Qraft Technolongies (AXE)

BEAR and BRAC

RL Research

1 Overview of BCQ

1.1 Introduction

Extrapolation Error is an error in off-policy value learning which is introduced by the mismatch between the dataset and true state-action visitatoin of the current policy.

Deep Q-learning tend to use near-on-policy exploratory policies, such as epsilion-greedy, in conjuction with a replay buffer. However, when learning off-policy, or in a batch setting, extrapolation error will never be corrected due to the inability to collect new data

1.2 Proposed Idea

To avoid extrapolation error, a policy should induce a similar state-actoin visitation to batch. Policies, which satisfy theis condition are called batch-constrained

Policies are trained to select actions with respect to three objectives:

1. Minimize the distance for selected actions to the data in the batch
2. Lead to states where familiar data can be observed
3. Maximize the value function

1.3 Method

BCQ approaches the notion of batch-constrained through a generative model, which generates plausible actions with high similarity to the batch, and then selects the highest valued action through a learned Q-network.

VAE is used as a generative model in BCQ, The generative model $G\omega$, alongside the value function $Q\theta$, can be used as a policy by sampling n actions from $G\omega$ and selecting the highest valued action ac-cording to the value estimate $Q\theta$.

To increase the diversity of seen actions, they introduced a perturbation model $\xi\varphi(s, a, \Phi)$, which outputs an adjustment to an action a in the range $[-\Phi, \Phi]$

This results in the policy:

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

2 BEAR

2.1 Abstract

- Analysis of error accumulation in the bootstrapping process due to out-of-distribution inputs in off-policy offline reinforcement learning
- Unlike BCQ, BEAR doesn't constraint the distribution of the learned policy to be close to the behavior policy.
- Instead, BEAR restricts the policy to ensure that the actions of the policy lies in the support of the training distribution.

2.2 Background of BEAR

2.2.1 What is bootstrapping in RL

Definition 2.1. Bootstrapping is using one or more estimated values in the update step for the same kind of estimated value, bootstrapping in RL means that you update a value based on some estimates and not on some exact values.

In most TD updates rules you can see SARSA update like,

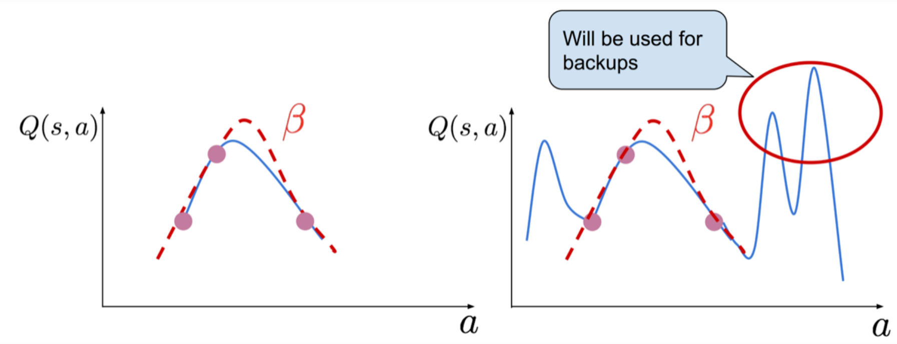
$$Q(s, a) \leftarrow Q(s, a) + \alpha \left(R_{t+1} + \gamma Q\left(s^{\prime}, a^{\prime}\right) - Q(s, a)\right)$$

The value $R_{t+1} + \gamma Q\left(s^{\prime}, a^{\prime}\right)$ is an estimate for the true value of $Q(s, a)$, and also called the TD target. It is a bootstrap method because we are in part using a Q value to update another Q value. There is a small amount of real observed data in the form of R_{t+1}, the immediate reward for the step, and also in the state transition $s \rightarrow s^{\prime}$

The main disadvantage of bootstrapping is that it is biased towards whatever your starting values of $Q\left(s^{\prime}, a^{\prime}\right)$ (or $V\left(s^{\prime}\right)$) are. Those are most likely wrong, and the update system can be unstable as a whole because of too much self-reference and not enough real data - this is a problem with off-policy learning (e.g. Q-learning) using neural networks.

2.2.2 Why Bootstrapping Error is accumulated?

The agent is unable to correct these bootstrapping errors as it cannot gather ground truth by exploring the environment, which leads to Q-values not converging to the correct values and making to final performance that is much worse than expected.



Incorrectly high Q-values for OOD actions may be used for backups, leading to accumulation of error. Figure and caption credit: Aviral Kumar.

2.3 Method

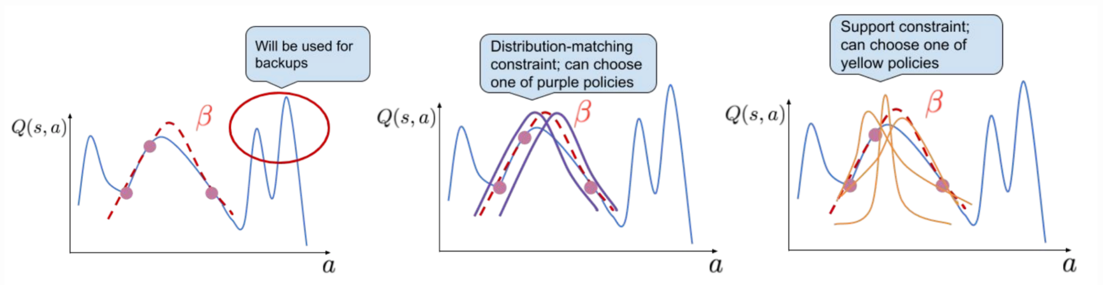


Illustration of support constraint (BEAR) (right) and distribution-matching constraint (middle). Figure and caption credit: Aviral Kumar.

1. The key idea behind BEAR is to constrain the learned policy to lie within the support of the behavior policy distribution.
2. In contrast to distribution matching (middle - BCQ, BRAC) – BEAR does not constrain the learned policy to be close in distribution to the behavior policy.
3. This is referred as support constraint. As an example, in a setting with a uniform at random behavior policy, a support constraint allows dynamic programming to learn an optimal, deterministic policy.
4. However, a distribution matching constraint will require that the learned policy be highly stochastic (and thus not optimal), for instance in middle graph, the learned policy is constrained to be one of the stochastic purple policies, however in right figure, the learned policy can be a (near)deterministic yellow policy.
5. theoretical insight behind this choice is that a support constraint enables us to control error propagation by upper bounding **concentrability** under the learned policy, while providing the capacity to reduce divergence from the optimal policy.
6. Only requiring a support match is a much weaker assumption, which enables Offline RL to more flexibly consider a wider range of actions so long as the batch of data has used those actions at some point with non-negligible probability.

Using the sampled Maximum Mean Discrepancy (MMD) distance between actions as a measure of support divergence. Letting $X = \{x_1, \cdots, x_n\}, Y = \{y_1, \cdots, y_n\}$ and k be any RBF kernel, we have:

$$\mathrm{MMD}^2(X,Y) = \frac{1}{n^2}\sum_{i,i'} k\left(x_i, x_{i'}\right) - \frac{2}{nm}\sum_{i,j} k\left(x_i, y_j\right) + \frac{1}{m^2}\sum_{j,j'} k\left(y_j, y_{j'}\right)$$

A simple code snippet for computing MMD is shown below:

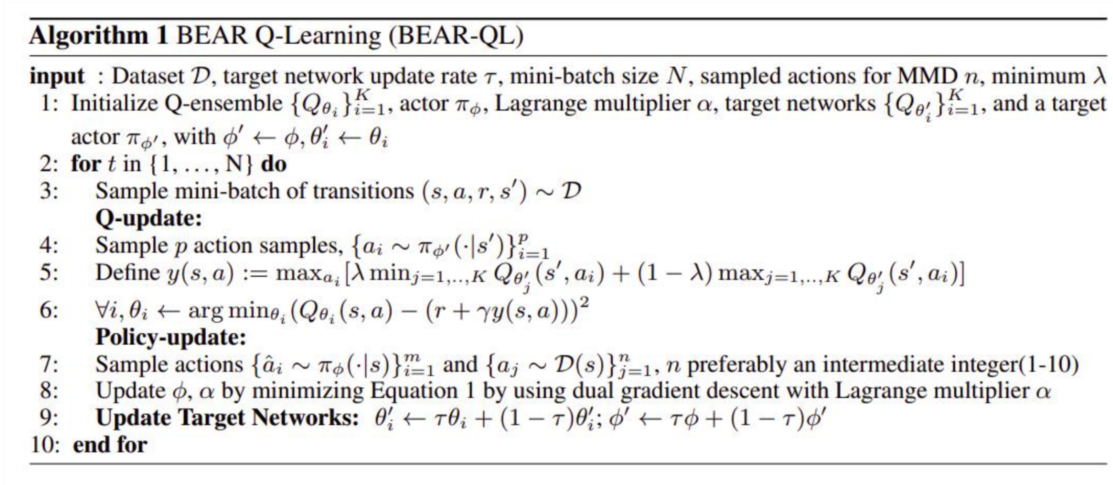
```
def gaussian_kernel(x, y, sigma=0.1):
  return exp(-(x - y).pow(2).sum() / (2 * sigma.pow(2)))

def compute_mmd(x, y):
  k_x_x = gaussian_kernel(x, x)
  k_x_y = gaussian_kernel(x, y)
  k_y_y = gaussian_kernel(y, y)
  return sqrt(k_x_x.mean() + k_y_y.mean() - 2*k_x_y.mean())
```

MMD is amenable to stochastic gradient-based training and we show that computing $\mathrm{MMD}(P, Q)$ using only few samples from both distributions P and Q provides sufficient signal to quantify differences in support but not in probability density, hence making it a preferred measure for implementing the support constraint. To sum up, the new (constrained)policy improvement step in an actor-critic setup is given by:

$$\pi_\phi := \max_{\pi \in \Delta_{|S|}} \mathbb{E}_{s \sim \mathcal{D}} \mathbb{E}_{a \sim \pi(\cdot|s)} \left[Q_\theta(s, a)\right] \quad \text{s.t.} \quad \mathbb{E}_{s \sim \mathcal{D}}[\mathrm{MMD}(\beta(\cdot \mid s), \pi(\cdot \mid s))] \leq \varepsilon$$

- $\mathcal{D}$ represents the static data of transitions collected by behavioral policy
- β, and the j subscripts are from the ensemble of Q-functions used to compute a conservative estimate of Q-values.



```
Algorithm 1 BEAR Q-Learning (BEAR-QL)
input : Dataset $\mathcal{D}$, target network update rate $\tau$, mini-batch size $N$, sampled actions for MMD $n$, minimum $\lambda$
 1: Initialize Q-ensemble $\{Q_{\theta_i}\}_{i=1}^K$, actor $\pi_\phi$, Lagrange multiplier $\alpha$, target networks $\{Q_{\theta'_i}\}_{i=1}^K$, and a target
    actor $\pi_{\phi'}$, with $\phi' \leftarrow \phi, \theta'_i \leftarrow \theta_i$
 2: for $t$ in $\{1, \ldots, \mathrm{N}\}$ do
 3:     Sample mini-batch of transitions $(s, a, r, s') \sim \mathcal{D}$
        Q-update:
 4:     Sample $p$ action samples, $\{a_i \sim \pi_{\phi'}(\cdot|s')\}_{i=1}^p$
 5:     Define $y(s, a) := \max_{a_i} [\lambda \min_{j=1,..,K} Q_{\theta'_j}(s', a_i) + (1 - \lambda) \max_{j=1,..,K} Q_{\theta'_j}(s', a_i)]$
 6:     $\forall i, \theta_i \leftarrow \arg\min_{\theta_i} (Q_{\theta_i}(s, a) - (r + \gamma y(s, a)))^2$
        Policy-update:
 7:     Sample actions $\{\hat{a}_i \sim \pi_\phi(\cdot|s)\}_{i=1}^m$ and $\{a_j \sim \mathcal{D}(s)\}_{j=1}^n$, $n$ preferably an intermediate integer(1-10)
 8:     Update $\phi$, $\alpha$ by minimizing Equation 1 by using dual gradient descent with Lagrange multiplier $\alpha$
 9:     Update Target Networks: $\theta'_i \leftarrow \tau\theta_i + (1 - \tau)\theta'_i$; $\phi' \leftarrow \tau\phi + (1 - \tau)\phi'$
10: end for
```

Figure 1: Full Algorithm

3 BRAC

3.1 Abstract

- The authors introduce their framework, **Behavior Regularized Actor Critic**, to empirically evaluate recently proposed methods as well as a number of simple baselines across a variety of offline continuous control tasks.
- Surprisingly, they find that many of the technical complexities introduced in recent methods(BCQ, BEAR) are unnecessary to achieve strong performance.

3.2 Brief discussion two recent papers BEAR and BCQ

3.3 Off-Policy offline Rl

Offline RL (also known as batch RL) considers the problem of learning a policy π from a fixed dataset $\mathbf{D}$ consisting of single-step transitions $(\mathrm{s}, \mathrm{a}, \mathrm{r}, \mathrm{s}')$.

Behavior policy $\pi_b(\mathrm{a} \mid \mathrm{s})$ is defined as the conditional distribution $\mathrm{p}(\mathrm{a} \mid \mathrm{s})$ observed in the dataset distribution $\mathbf{D}$.

$$\hat{\pi}_b := \underset{\hat{\pi}}{\operatorname{argmax}} \mathbb{E}_{(s,a,r,s') \sim \mathcal{D}}[\log \hat{\pi}(a \mid s)] \ .$$

This is the learned policy and is referred to as **"cloned policy"**.

3.3.1 BEAR

Bootstrapping Error Accumulation Reduction (BEAR) paper identifies bootstrapping error as a key source of instability in current off-policy methods, which is due to bootstrapping from actions that lie outside of the training data distribution.

BEAR learns a policy to maximize Q-values while penalizing it from diverging from behavior policy support. They measure the divergence from behavioral policy using a Kernel Maximum Mean Discrepancy (MMD).

$$\operatorname{MMD}_k^2\left(\pi(\cdot \mid s), \pi_b(\cdot \mid s)\right)=\underset{x, x^{\prime} \sim \pi(\cdot \mid s)}{\mathbb{E}}\left[K\left(x, x^{\prime}\right)\right]-2 \mathbb{E}_{\substack{x \sim \pi(\cdot \mid s) \\ y \sim \pi_b(\cdot \mid s)}}[K(x, y)]+\underset{y, y^{\prime} \sim \pi_b(\cdot \mid s)}{\mathbb{E}}\left[K\left(y, y^{\prime}\right)\right]$$

where K is a Kernel function. Target Q-value function is calculated as:

$$\bar{Q}\left(s^{\prime}, a^{\prime}\right):=0.75 \cdot \min _{j=1, \ldots, k} Q_{\psi_j^{\prime}}\left(s^{\prime}, a^{\prime}\right)+0.25 \cdot \max _{j=1, \ldots, k} Q_{\psi_j^{\prime}}\left(s^{\prime}, a^{\prime}\right)$$

where $\psi'\mathbf{j}$ is an updated ensemble of target Q-functions. The authors of the BEAR paper have used $\mathbf{k} = 4$ in its implementation.

3.3.2 BCQ

In this paper the authors introduce Batch Constrained Reinforcement Learning, to overcome the extrapolation error, a phenomenon in which unseen state-action pairs are erroneously estimated to have unrealistic values, by training agents to maximize reward while minimizing the mismatch between the state-action pair visitation of the policy and the state-action pairs contained in the batch.

BCQ forces the policy π to be close to behavioral policy π_b with a specific parameterization:

$$\pi_\theta(a \mid s):=\underset{a_i+\xi_\theta\left(s, a_i\right)}{\operatorname{argmax}} Q_\psi\left(s, a_i+\xi_\theta\left(s, a_i\right)\right) \text { for } a_i \sim \pi_b(a \mid s), i=1, \ldots, N,$$

3.4 Behavior Regularized Actor Critic

Previous approaches to off-policy RL encourage learned policy to be close to behavior policy. BRAC generalizes these approaches while providing more implementation options.

Two common ways to incorporate regularization are:

- Penalty in the value function
- Penalty solely on the policy

3.5 Method

1. The first way is the value penalty (vp), where they add a term to target Q-value calculation that regularizes π towards π_b and define **the penalized value function** as:

$$V_D^\pi(s)=\sum_{t=0}^{\infty} \gamma^t \mathbb{E}_{s_t \sim P_t^\pi(s)}\left[R^\pi\left(s_t\right)-\alpha D\left(\pi\left(\cdot \mid s_t\right), \pi_b\left(\cdot \mid s_t\right)\right)\right]$$

   - where D is the Divergence function. **Q-value objective** is given as:

$$\min _{Q_\psi} \mathbb{E}_{\substack{\left(s, a, r, s^{\prime}\right) \sim \mathcal{D} \\ a^{\prime} \sim \pi_\theta\left(\cdot \mid s^{\prime}\right)}}\left[\left(r+\gamma\left(\bar{Q}\left(s^{\prime}, a^{\prime}\right)-\alpha \hat{D}\left(\pi_\theta\left(\cdot \mid s^{\prime}\right), \pi_b\left(\cdot \mid s^{\prime}\right)\right)\right)-Q_\psi(s, a)\right)^2\right]$$

- and **Policy learning objective** is given as:

$$\max_{\pi_\theta} \mathbb{E}_{(s,a,r,s')\sim\mathcal{D}}\left[\mathbb{E}_{a''\sim\pi_\theta(\cdot|s)}\left[Q_\psi\left(s,a''\right)\right]-\alpha\hat{D}\left(\pi_\theta(\cdot\mid s),\pi_b(\cdot\mid s)\right)\right]$$

2. The second way is by policy regularization (pr), where they add a regularizer only during policy optimization, which means they use the same **Q-value objective** and **Policy learning objective**, but use $\boldsymbol{\alpha}=\mathbf{0}$ in the Q update while using a non-zero $\mathbf{a}$ in the policy update.

In addition to the choice of these two ways, the choice of $\mathbf{D}$ and how to perform a sample-estimate of D are key designs of BRAC.

1. **Kernel MMD**

   We can compute a sample-based estimate of kernel MMD by drawing samples from both π_θ and $\mathbf{\pi}$b. As we don't have access to multiple samples from π_b, this requires a pre-estimated cloned policy.

$$\mathrm{MMD}_k^2\left(\pi(\cdot\mid s),\pi_b(\cdot\mid s)\right)=\underset{x,x'\sim\pi(\cdot|s)}{\mathbb{E}}\left[K\left(x,x'\right)\right]-2\mathbb{E}_{\substack{x\sim\pi(\cdot|s)\\ y\sim\pi_b(\cdot|s)}}[K(x,y)]+\underset{y,y'\sim\pi_b(\cdot|s)}{\mathbb{E}}\left[K\left(y,y'\right)\right]$$

2. **KL Divergence**

$$D_{\mathrm{KL}}\left(\pi_\theta(\cdot\mid s),\pi_b(\cdot\mid s)\right)=\mathbb{E}_{a\sim\pi_\theta(\cdot|s)}\left[\log\pi_\theta(a\mid s)-\log\pi_b(a\mid s)\right]$$

   We can use the pre-estimated cloned policy in place of behavioral policy. Alternatively, we can avoid estimating behavioral policy by using the dual form of KL Divergence.

$$D_f(p,q)=\mathbb{E}_{x\sim p}[f(q(x)/p(x))]=\max_{g:\mathcal{X}\mapsto\mathrm{dom}(f^*)}\mathbb{E}_{x\sim q}[g(x)]-\mathbb{E}_{x\sim p}\left[f^*(g(x))\right]$$

   where $\mathbf{f}^*$ is the Fenchel dual of $\mathbf{f}$

$$f(x)=-\log x \text{ and } f^*(t)=-\log(-t)-1$$

3. **Wasserstein Distance**

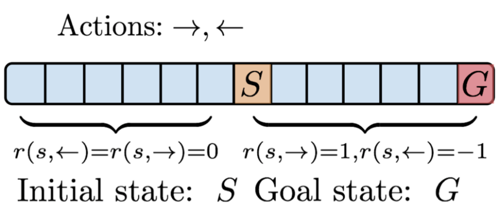
$$W(p,q)=\sup_{g:\|g\|_L\leq 1}\mathbb{E}_{x\sim p}[g(x)]-\mathbb{E}_{x\sim q}[g(x)]$$

4 Support constraint vs Distribution-matching constraint

4.1 Author of BEAR

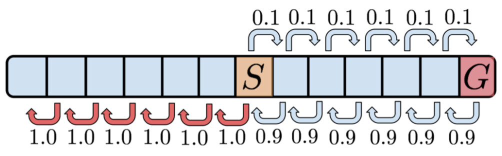
some works (BRAC, BCQ) argue that that using a distribution-matching constraint might suffice in such fully off-policy RL problems. But we provide an instance of an MDP where distribution-matching constraint might lead to arbitrarily suboptimal behavior while support matching does not suffer from this issue.

Consider the 1 D-lineworld MDP shown in Figure 4 below. Two actions (left and right) are available to the agent at each state. The agent is tasked with reaching to the goal state G, starting from state S and the corresponding per-step reward values are shown in Figure $4(\mathrm{a})$. The agent is only allowed to learn from behavior data generated by a policy that performs actions with probabilities described in Figure $4(\ \mathrm{b})$, and in particular, this behavior policy executes the suboptimal action at states in-between S and G with a high likelihood of 0.9, however, both actions $\leftarrow$ and $\rightarrow$ are in-distribution at all these states.



Actions: $\rightarrow, \leftarrow$
S
G
$r(s,\leftarrow)=r(s,\rightarrow)=0$
$r(s,\rightarrow)=1, r(s,\leftarrow)=-1$
Initial state: S Goal state: G

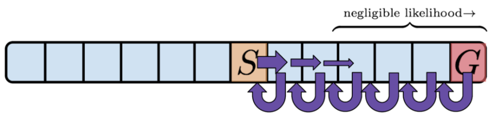
(a) 1D-Lineworld Environment



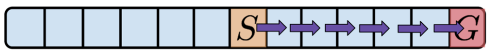
(b) Behavior Policy

Figure 4: Example 1D lineworld and the corresponding behavior policy.

In Figure $5(a)$, we show that the learned policy with a distribution-matching constraint can be arbitrarily suboptimal, infact, the probability of reaching goal G by rolling out this policy is very small, and tends to 0 as the environment is made larger. However, in Figure $5(\mathrm{~b})$, we show that a support constraint can recover an optimal policy with probability 1 .



(a) Learned Policy via distribution-matching



(b) Learned Policy via support-constraint

Figure 5: Policies learned via distribution-matching and support-matching in the 1D lineworld shown in Figure 4.

Why does distribution-matching fail here? Let us analyze the case when we use a penalty for distribution-matching. If the penalty is enforced tightly, then the agent will be forced to mainly execute the wrong action $(\leftarrow)$ in states between S and G, leading to suboptimal behavior. However, if the penalty is not enforced tightly, with the intention of achieving a better policy than the behavior policy, the agent will perform backups using the OOD-action $\rightarrow$ at states to the left of S, and these backups will eventually affect the Q-value at state S. This phenomenon will lead to an incorrect Q-function, and hence a wrong policy - possibly, one that goes to the left starting from S instead of moving towards G, as OOD-action backups combined with overestimation bias in Q-learning might make action $\leftarrow$ at state S look more preferable. Figure 6 shows that some states need a strong penalty/constraint (to prevent OOD backups) and the others require a weak penalty/constraint (to achieve optimality) for distribution-matching to work, however, this cannot be achieved via conventional distribution-matching approaches.

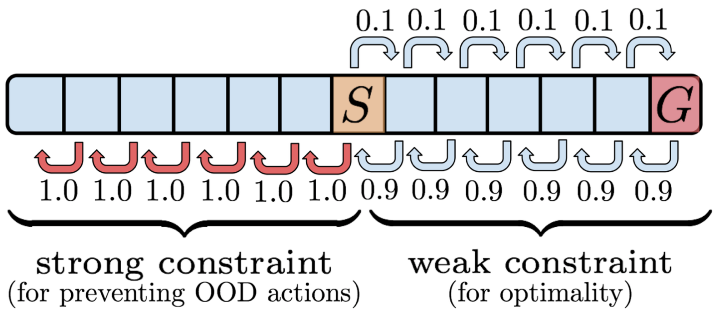


Figure 6: Analysis of the strength of distribution-matching constraint needed at different states.