作业

Homework5

庄镇华 502022370071

A RL Homework Assignment

2022 年 12 月 30 日

实验环境

本次作业的数据集为 d4rl 的 Hopper 数据集，共有三个任务：hopper-random，hopper-medium，hopper-expert。random 为随机策略在环境中采样获得的数据集，medium 为策略学习到中等性能时在环境中采样获得的数据集，expert 为专家策略在环境采样获得的数据集。

状态空间：11 维，各维度取值范围连续。

动作空间：3 维，各维度取值范围 (-1,1)，连续。

奖励函数：奖励函数为 x 方向上前进的长度、对动作幅度的惩罚（减去动作各维度的平方和）以及是否存活的奖励（如果当前步没有倒下则 +1）。

转移函数：几乎是确定性的转移。

最大步长：1000。

问题 1 思考探究值外推问题

首先我们举个例子来简要说明值外推带来的问题。在之前的作业中，我们尝试了 Q-table 求解强化学习。

那么现在我们暂时不考虑 Q-value 的泛化能力，先思考如下的迷宫问题，在该例子中，黄色为出发点，棕色为墙壁，粉色为可通行区域，红色为离线数据集中的数据。每个格子都有两个数值，第一个数值为单步的奖励，第二个数值为我们的 Q-table 各个动作下初始化的数值。

请思考：

1. 当 $x = -1$ 时，我们直接在离线数据集上进行动态规划，Q-value 会发生什么变化？我们能否获得一个可以走到最大奖励的策略？

2. 那么当 $x = 20$ 呢？（认为折扣因子为 1）。

-5, x	0, x	-2, x	0, x	0, x	-1, x	0, x	0, x	0, x	7, x
0, x	0, x	0, x	0, x	0, x	0, x	0, x	0, x	0, x	5, x
0, x	0, x	0, x	0, x	0, x	0, x	0, x	0, x	0, x	0, x
0, x	0, x	0, x	0, x	0, x	0, x	0, x	0, x	0, x	-4, x
0, x	0, x	0, x	0, x	0, x	0, x	0, x	0, x	0, x	0, x
0, x	0, x	0, x	0, x	0, x	0, x	0, x	0, x	0, x	-2, x

表 1: 迷宫问题

基于该例子，我们可以对离线强化学习值外推误差有一个初步的认识：由于无法对数据集外的状态-动作对进行采样，在进行策略迭代时，如果需要使用数据集外的状态-动作值计算 Q-target，来更新当前的 Q 值时，一旦该外推的值过大，将可能导致整个数据集上的 Q 值估计错误，从而导致导出的策略完全失败。

以往的工作对值外推的解决方法有 Model-based 方法。（**请思考**，为什么当前的 model-based 算法可以一定程度上缓解该问题？提供一个思考角度：学到的模型的泛化性要强于 Q-function，一定程度上覆盖到了优化策略采样的轨迹终端）

问题 2

实现任意一种已有的离线强化学习算法（如 BCQ、CQL、MOPO、BRAC、BEAR、MORel 等），并在给定的数据集上训练并导出模型。

问题 3

在训练过程中，由于没有真实环境可供测试策略，尝试思考算法何时应该停止（不一定是收敛）。

问题 4 选做

尝试评估训练时的策略。

问题 5 选做

尝试改进上述的离线强化学习算法。

解答：

1 作业内容

探究离线强化学习值外推误差的问题，并在 d4rl 的 Hopper 数据集上训练离线强化学习算法，并思考如何评估策略性能以及判断算法的收敛。

2 算法原理

2.1 CQL

离线强化学习面对的巨大挑战是如何减少外推误差。实验证明，外推误差主要会导致在远离数据集的点上函数的过高估计。因此，如果能用某种方法将算法中偏离数据集的点上的

函数保持在很低的值，或许能消除部分外推误差的影响，这就是保守 Q-learning（conservative Q-learning, CQL）算法的基本思想。CQL 是直接限制函数的算法的代表。

在一般的 Q-learning 中，Q 的更新方程可以写为：

$$\hat{Q}^{k+1} \leftarrow \operatorname{argmin}_Q \mathbb{E}_{(s,a)\sim\mathcal{D}}\left[\left(Q(s,a) - \hat{\mathcal{B}}^\pi \hat{Q}^k(s,a)\right)^2\right]$$

为了防止 Q 值在各个状态上（尤其是不在数据集中的状态上）的过高估计，我们要对某些状态上的高 Q 值进行惩罚。我们希望 Q 在某个特定分布 $\mu(s,a)$ 上的期望值最小。因此对数据集中的状态 s 按策略 μ 得到的动作进行惩罚：

$$\hat{Q}^{k+1} \leftarrow \operatorname{argmin}_Q \beta\mathbb{E}_{s\sim\mathcal{D},a\sim\mu(a|s)}[Q(s,a)] + \frac{1}{2}\mathbb{E}_{(s,a)\sim\mathcal{D}}\left[\left(Q(s,a) - \hat{\mathcal{B}}^\pi \hat{Q}^k(s,a)\right)^2\right]$$

其实只要求 Q 在 $\pi(a|s)$ 上的期望值 V^π 比真实值小就可以，即对于符合用于生成数据集的行为策略的数据点，不必限制让值很小。

$$\hat{Q}^{k+1} \leftarrow \operatorname{argmin}_Q \beta\cdot\left(\mathbb{E}_{s\sim\mathcal{D},a\sim\mu(a|s)}[Q(s,a)] - \mathbb{E}_{s\sim\mathcal{D},a\sim\hat{\pi}_b(a|s)}[Q(s,a)]\right) + \frac{1}{2}\mathbb{E}_{(s,a)\sim\mathcal{D}}\left[\left(Q(s,a) - \hat{\mathcal{B}}^\pi \hat{Q}^k(s,a)\right)^2\right]$$

考虑计算的时间开销，用使 Q 取最大值的 μ 去近似 π，再加上正则项 $\mathcal{R}(\mu)$，综合起来得到完整的迭代方程：

$$\begin{aligned}\hat{Q}^{k+1} \leftarrow \operatorname{argmin}_Q \max_\mu \beta \cdot &\left(\mathbb{E}_{s\sim\mathcal{D},a\sim\mu(a|s)}[Q(s,a)] - \mathbb{E}_{s\sim\mathcal{D},a\sim\hat{\pi}_b(a|s)}[Q(s,a)]\right) \\ &+ \frac{1}{2}\mathbb{E}_{(s,a)\sim\mathcal{D}}\left[\left(Q(s,a) - \hat{\mathcal{B}}^\pi \hat{Q}^k(s,a)\right)^2\right] + \mathcal{R}(\mu)\end{aligned}$$

正则项采用和一个先验策略 $\rho(a|s)$ 的 KL 距离，即 $\mathcal{R}(\mu) = -D_{KL}(\mu,\rho)$。一般来说，取 $\rho(a|s)$ 为均匀分布 $\mathcal{U}(a)$ 即可，这样可以将迭代方程化简为：

$$\hat{Q}^{k+1} \leftarrow \operatorname{argmin}_Q \beta\cdot\mathbb{E}_{s\sim\mathcal{D}}\left[\log\sum_a \exp(Q(s,a)) - \mathbb{E}_{a\sim\hat{\pi}_b(a|s)}[Q(s,a)]\right] + \frac{1}{2}\mathbb{E}_{(s,a)\sim\mathcal{D}}\left[\left(Q(s,a) - \hat{\mathcal{B}}^\pi \hat{Q}^k(s,a)\right)^2\right]$$

算法 1 基于 SAC 框架的 CQL 算法

初始化 Q 网络 Q_θ、目标 Q 网络 $Q_{\theta'}$ 和策略 π_ϕ、熵正则系数 α
for 训练次数 $t = 1 \to T$ **do**
  更新熵正则系数：$\alpha_t \leftarrow \alpha_{t-1} - \eta_\alpha \nabla_\alpha \mathbb{E}_{s\sim\mathcal{D},a\sim\pi_\phi(a|s)}\left[-\alpha_{t-1}\log\pi_\phi(a\mid s) - \alpha_{t-1}\mathcal{H}\right]$
  更新函数 Q：

$$\begin{aligned}\theta_t \leftarrow \theta_{t-1} - \eta_Q\nabla_\theta\Bigg(&\alpha\cdot\mathbb{E}_{s\sim\mathcal{D}}\left[\log\sum_a\exp\left(Q_\theta(s,a)\right) - \mathbb{E}_{a\sim\hat{\pi}_b(a|s)}\left[Q_\theta(s,a)\right]\right] \\ &+ \frac{1}{2}\mathbb{E}_{(s,a)\sim\mathcal{D}}\left[\left(Q_\theta(s,a) - \mathcal{B}^{\pi_\phi}Q_\theta(s,a)\right)^2\right]\Bigg)\end{aligned}$$

  更新策略：$\phi_t \leftarrow \phi_{t-1} - \eta_\pi\nabla_\phi\mathbb{E}_{s\sim\mathcal{D},a\sim\pi_\phi(a|s)}\left[\alpha\log\pi_\phi(a\mid s) - Q_\theta(s,a)\right]$
end for

3 实现过程

3.1 实现算法

在 agent 文件夹下的 cql.py 文件实现基于 SAC 框架的 CQL 算法。算法基于原始 SAC 算法实现，修改的部分如下：

首先初始化需要指定额外的超参数：CQL 损失函数中的系数以及 CQL 的动作采样数。

```python
class CQL:

    def __init__(self, args):
        # ... #
        # @zhuangzh
        self.beta = args.beta  # CQL损失函数中的系数
        self.num_random = args.num_random  # CQL中的动作采样数
```

然后是更新 critic 网络时，依据迭代方程，损失函数需要加入对 Q 值限制的部分。

$$\hat{Q}^{k+1} \leftarrow \operatorname{argmin}_Q \underbrace{\beta \cdot \mathbb{E}_{s\sim\mathcal{D}}\left[\log \sum_a \exp(Q(s,a)) - \mathbb{E}_{a\sim\hat{\pi}_b(a|s)}[Q(s,a)]\right]}_{\text{限制 Q 值}} + \frac{1}{2}\mathbb{E}_{(s,a)\sim\mathcal{D}}\left[\left(Q(s,a) - \hat{\mathcal{B}}^{\pi}\hat{Q}^k(s,a)\right)^2\right]$$

其中 $\mathbb{E}_{s\sim\mathcal{D}}[\cdot]$ 利用随机采样估计，实验中 batch_size 为 256，num_random 为 20。

```python
    def update_critic(self, state, action, next_state, reward, done):
        with torch.no_grad():
            next_action, next_logprobs, _ = self.actor_eval(next_state,
    get_logprob=True)
            q_t1, q_t2 = self.critic_target(next_state, next_action)
            # take min to mitigate positive bias in q-function training
            q_target = torch.min(q_t1, q_t2)
            value_target = reward + (1.0 - done) * self.gamma * (q_target - self.
    alpha * next_logprobs)
        q_1, q_2 = self.critic_eval(state, action)
        loss_1 = F.mse_loss(q_1, value_target)
        loss_2 = F.mse_loss(q_2, value_target)

        # @zhuangzh
        # 以上与SAC相同,以下Q网络更新是CQL的额外部分
        batch_size = state.shape[0]
        random_unif_action = torch.rand([batch_size * self.num_random, action.
    shape[-1]], dtype=torch.float).uniform_(-1, 1).to(self.device)
        random_unif_log_pi = np.log(0.5 ** next_action.shape[-1])
        tmp_state = state.unsqueeze(1).repeat(1, self.num_random, 1).view(-1,
    state.shape[-1])
        tmp_next_state = next_state.unsqueeze(1).repeat(1, self.num_random, 1).
    view(-1, next_state.shape[-1])
        random_curr_action, random_curr_log_pi, _ = self.actor_eval(tmp_state,
    get_logprob=True)
        random_next_action, random_next_log_pi, _ = self.actor_eval(tmp_next_state
    , get_logprob=True)

        def util(x, num = self.num_random):
            x1, x2 = x
            return x1.view(-1, num, 1), x2.view(-1, num, 1)
```

```
        q1_unif, q2_unif = util(self.critic_eval(tmp_state, random_unif_action))
        q1_curr, q2_curr = util(self.critic_eval(tmp_state, random_curr_action))
        q1_next, q2_next = util(self.critic_eval(tmp_state, random_next_action))
        q1_cat = torch.cat([q1_unif - random_unif_log_pi,
            q1_curr - random_curr_log_pi.detach().view(-1, self.num_random, 1),
            q1_next - random_next_log_pi.detach().view(-1, self.num_random, 1)
        ], dim=1)
        q2_cat = torch.cat([q2_unif - random_unif_log_pi,
            q2_curr - random_curr_log_pi.detach().view(-1, self.num_random, 1),
            q2_next - random_next_log_pi.detach().view(-1, self.num_random, 1)
        ], dim=1)

        loss_1 += self.beta * (torch.logsumexp(q1_cat, dim=1).mean() - q_1.mean())
        loss_2 += self.beta * (torch.logsumexp(q2_cat, dim=1).mean() - q_2.mean())
        # @zhuangzh

        q_loss_step = loss_1 + loss_2
        self.critic_optim.zero_grad()
        q_loss_step.backward()
        self.critic_optim.step()

        return q_loss_step.item()
```

3.2 评估算法

在 MoJoCo 物理仿真引擎中评估利用离线数据集训练的模型，首先需要下载并解压 MoJoCo 动态链接库到指定的文件夹 ~/.mojoco 中，然后添加一些环境变量：

```
export LD_LIBRARY_PATH=~/.mujoco/mujoco210/bin$LD_LIBRARY_PATH:+:$LD_LIBRARY_PATH
export MUJOCO_KEY_PATH=~/.mujoco$MUJOCO_KEY_PATH
```

最后利用 pip 下载 mojoco_py 库，`pip install mujoco_py`，这样才能使用 gym 中的 Hopper-v2 环境。

实验中使用的评估方式为：每经过 10k 次迭代训练后在环境中评估算法一次，每次执行五次情节，取五次情节奖励平均值作为本次评估结果。

```
env = gym.make('Hopper-v2')
def evaluate(env, policy, eval_runs=5):
    """
    Makes an evaluation run with the current policy
    """
    reward_batch = []
    for i in range(eval_runs):
        state = env.reset()
        rewards = 0
        while True:
            action = policy.inference(state, True)
            state, reward, done, _ = env.step(action)
            rewards += reward
            if done:
                break
        reward_batch.append(rewards)
    return np.mean(reward_batch)
```

4 复现方式

在 code 文件夹下运行 `python train.py`, 可训练、保存模型并评估算法在三种离线数据集上的性能。

在 code 文件夹下运行 `python main.py`, 可以直接加载已保存的模型并评估算法在三种离线数据集上的性能。

5 实验效果

问题 1 思考探究值外推问题

1. 当 $x = -1$ 时，我们直接在离线数据集上进行动态规划，Q-value 会发生什么变化？我们能否获得一个可以走到最大奖励的策略？
2. 那么当 $x = 20$ 呢？（认为折扣因子为 1）。

答：当 $x = -1$ 时，直接在离线数据集上进行动态规划，Q-value 会逐步收敛至最优值，假设最终走出迷宫获得奖励为 0，那么最终结果如下表所示，此时可以获得走到最大奖励为 12 的策略。

-5, -1	0, -1	-2, -1	0, -1	0, -1	-1, -1	0, -1	0, -1	0, -1	7, 7
0, -1	0, -1	0, -1	0, -1	0, -1	0, -1	0, -1	0, -1	0, 12	5, 12
0, -1	0, -1	0, -1	0, -1	0, -1	0, -1	0, 12	0, 12	0, 12	0, -1
0, -1	0, -1	0, -1	0, -1	0, 12	0, 12	0, 12	0, -1	0, -1	-4, -1
0, -1	0, 12	0, 12	0, 12	0, 12	0, -1	0, 0	0, 0	0, -1	0, -1
0, 12	0, 12	0, -1	0, -1	0, -1	0, -1	0, -1	0, 0	0, 0	-2, -1

表 2: $x = -1$ 收敛结果

当 $x = 20$ 时，直接在离线数据集上进行动态规划，由于无法对数据集外的状态-动作对进行采样，在进行策略迭代时，如果需要使用数据集外的状态-动作值计算 Q-target，来更新当前的 Q 值时，由于外推的值过大，导致整个数据集上的 Q 值估计错误，假设最终走出迷宫获得奖励为 0，那么最终结果如下表所示，此时无法获得走到最大奖励为 12 的策略，而是最终策略选择的路径奖励可能是-1，-2，-4 或-5。

-5, 20	0, 20	-2, 20	0, 20	0, 20	-1, 20	0, 20	0, 20	0, 20	7, 7
0, 20	0, 20	0, 20	0, 20	0, 20	0, 20	0, 20	0, 20	0, 12	5, 12
0, 20	0, 20	0, 20	0, 20	0, 20	0, 20	0, 20	0, 20	0, 20	0, 20
0, 20	0, 20	0, 20	0, 20	0, 20	0, 20	0, 20	0, 20	0, 20	-4, 20
0, 20	0, 20	0, 20	0, 20	0, 20	0, 20	0, 20	0, 20	0, 20	0, 20
0, 20	0, 20	0, 20	0, 20	0, 20	0, 20	0, 20	0, 20	0, 20	-2, 20

表 3: $x = 20$ 收敛结果

请思考，为什么当前的 model-based 算法可以一定程度上缓解该问题？提供一个思考角度：学到的模型的泛化性要强于 Q-function，一定程度上覆盖到了优化策略采样的轨迹终端

答：外推误差，是指由于当前策略可能访问到的状态动作对与从数据集中采样得到的状态动作对的分布不匹配而产生的误差。关于外推误差问题，因为 model-based 算法学习到的模型泛化性要强于 Q-function，一定程度上覆盖到了优化策略采样的轨迹终端。

此外，该问题出现是因为无法对数据集外的状态-动作对进行采样，而 model-based 方法可以在 in-distribution 的 data 上，学习出 out-distribution 的 policy，例如 MOPO 用惩罚来限制 policy 的学习，构建出一个可有效使用在各个区域上的 policy，而不仅仅只针对 behavior policy 支持的区域。

问题 2

实现任意一种已有的离线强化学习算法（如 BCQ、CQL、MOPO、BRAC、BEAR、MORel 等），并在给定的数据集上训练并导出模型。

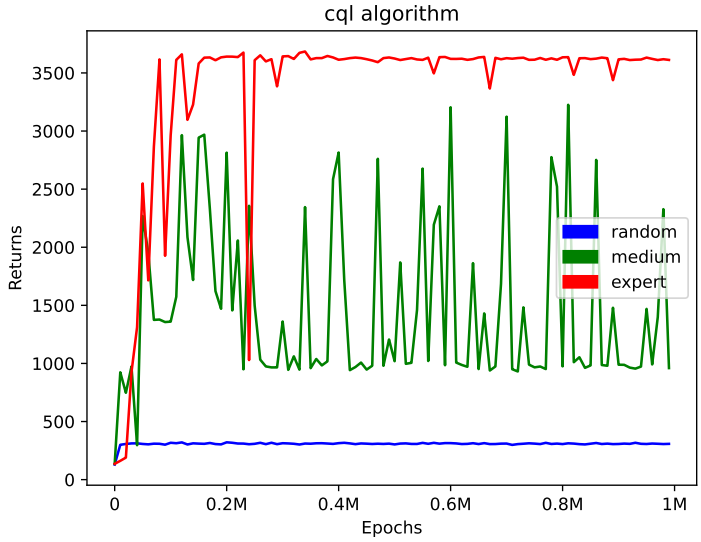

图 1: CQL 算法在离线数据集上训练的性能

本次实验实现的算法是 CQL 算法，该算法在 random、medium、expert 离线数据集上的训练性能如上图所示。

可以发现，因为不能通过与环境交互来获得新的数据，离线算法最终的效果和数据集有很大关系，并且波动会比较大。通常来说，调参后数据集中的样本质量越高，算法的表现就越好。

问题 3

在训练过程中，由于没有真实环境可供测试策略，尝试思考算法何时应该停止（不一定是收敛）。

一般情况下，可以根据收敛情况决定是否停止算法。

还可以类似监督学习，划分训练集和验证集，当在验证集上达到较好性能时停止算法。

还可以根据经验，即依据数据集规模、问题的复杂程度得到大致的迭代轮数。

还可以借鉴 model-based 方法，构建仿真模型，利用模型生成的样本进行测试，当在模型生成的样本上达到较好性能时停止算法。

问题 4 选做

尝试评估训练时的策略。

在 MoJoCo 物理仿真引擎中评估利用离线数据集训练的模型，首先需要下载并解压 MoJoCo 动态链接库到指定的文件夹 ~/.mojoco 中，然后添加一些环境变量：

```
export LD_LIBRARY_PATH=~/.mujoco/mujoco210/bin$LD_LIBRARY_PATH:+:$LD_LIBRARY_PATH
export MUJOCO_KEY_PATH=~/.mujoco$MUJOCO_KEY_PATH
```

最后利用 pip 下载 mojoco_py 库，`pip install mujoco_py`，这样才能使用 gym 中的 Hopper-v2 环境。

实验中使用的评估方式为：每经过 10k 次迭代训练后在环境中评估算法一次，每次执行五次情节，取五次情节奖励平均值作为本次评估结果。

以下几幅图展示了 SAC 和 CQL 算法以及 TD3BC 算法的训练性能：

对比 SAC 算法和 CQL 算法效果，可以发现，朴素的 SAC 算法没有添加对 Q 值的约束项，容易导致值外推问题，所以算法在三种离线数据集上表现都不好。而基于 SAC 框架的 CQL 算法通过添加对 Q 值的约束项，很好地缓解了值外推问题，在 medium 和 expert 数据集上都取得了较好的结果。

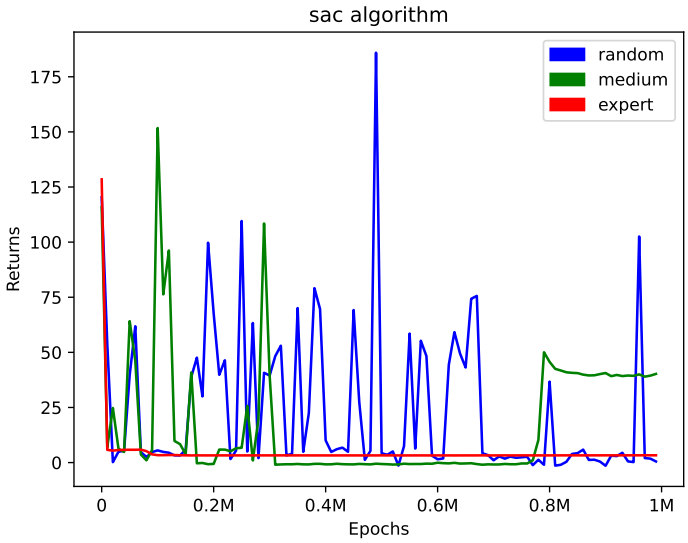

图 2: SAC 算法在三种离线数据集上训练的性能

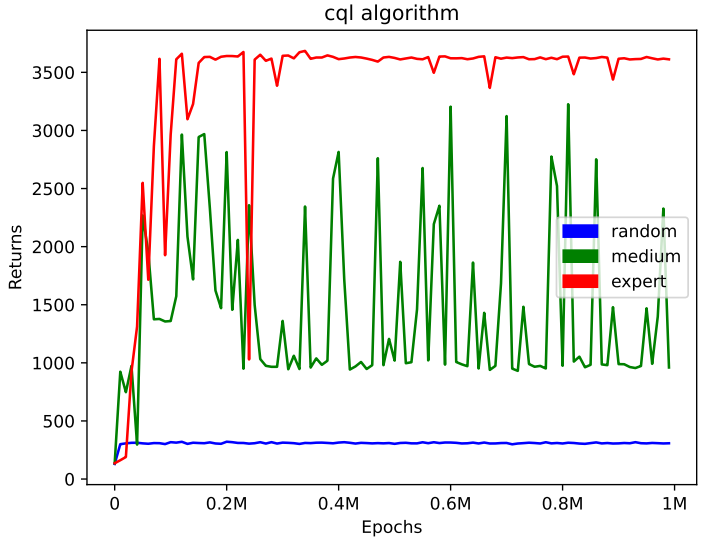

图 3: CQL 算法在三种离线数据集上训练的性能

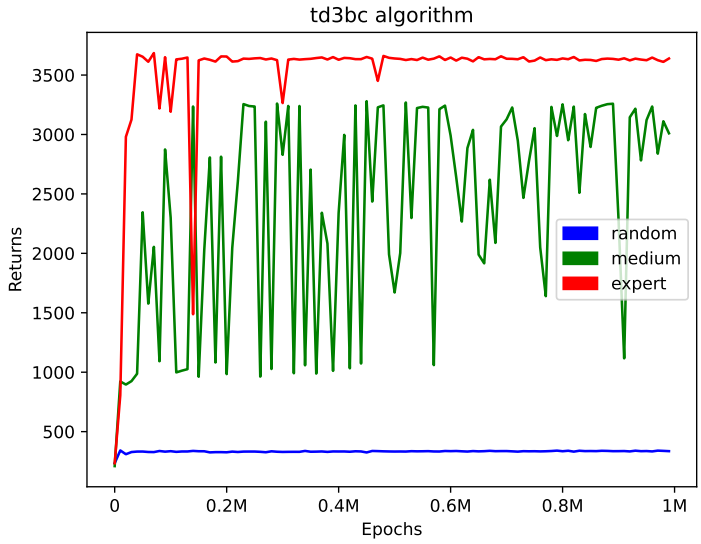

图 4: TD3BC 算法在三种离线数据集上训练的性能

对比 TD3BC 算法和 CQL 算法，可以发现，两种算法都缓解了值外推问题，在 random 和 expert 数据集上表现相似且出色，但在 medium 数据集上，TD3BC 算法略胜一筹。

最后，调取保存好的模型在环境中测试，三种数据集上训练出来的模型最终测试结果如下图所示。可以发现，数据集中的样本质量越高，算法的表现就越好。

```
(zhuangzh) root@ubuntu:~/zhuangzh/code# python main.py
/opt/anaconda3/envs/zhuangzh/lib/python3.7/site-packages/gym/logger.py:30:
  warnings.warn(colorize('%s: %s'%('WARN', msg % args), 'yellow'))
random: 315.7331563503584
medium: 2931.5557199715986
expert: 3660.2888251718564
```

图 5: CQL 离线数据集训练性能

问题 5 选做

尝试改进上述的离线强化学习算法

实行梯度裁剪，获得更稳定的性能。

```
        self.actor_optim.zero_grad()
        actor_loss.backward()
        # 梯度裁剪
        torch.nn.utils.clip_grad_norm_(self.actor_eval.parameters(), self.
    grad_norm_clip)
        self.actor_optim.step()

        self.temp_optim.zero_grad()
        temp_loss.backward()
        self.temp_optim.step()
```

6 小结

在这次实验中，我发现由于值外推误差的问题，如果不经处理，离线强化学习算法的效果会很差，并且算法通常对超参数极为敏感，非常难调参。

而且在实际复杂场景中通常不能像在模拟器中那样，每训练几轮就在环境中评估策略好坏，如何确定何时停止算法也是离线强化学习在实际应用中面临的一大挑战。

此外，离线强化学习在现实场景中的落地还需要关注离散策略评估和选择、数据收集策略的保守性和数据缺失性等现实问题。

不过无论如何，相对于其他前沿强化学习方向，例如多智能体强化学习和分层强化学习，离线强化学习和模仿学习目前更具有落地的潜力，两者都是为了解决在现实中训练智能体的困难而提出的，也都是强化学习实现工业应用的重要途径。