Off-policy Learning to Rank (BCQ implementation)

Outline

This document is used to describe the implementation of BCQ on off-policy learning to rank task. As for the results, please refer to *BCQ Results* documentation for details.

Task Formulation

In order to apply BCQ to off-policy learning to rank task, we first need to formulate our MDP model. In this case, we choose to model ***the process user browsing a ranking list*** as an MDP, where its sequential nature inspired us to do so.

- ***step***: each rank in the list, the length of the list is based on hyperparameter `END_POS`, usually 10.
- ***state***: the (weighted) average feature of docs in previous and current positions. Intuitively, the closer to current position, the weight is heavier (*1/distance*).

  ```
  states[position] = states[position-1] * position / (position+1) +
     dataset.get_features_by_query_and_docid(qid, result_list[position])
     if position > 0 else dataset.get_features_by_query_and_docid(qid,
  result_list[0])
  ```

- ***action***: the feature of chosen doc at current position.
- ***next state***: next state after chosen action by policy, similarly defined as state.
- ***reward***: metric dcg@k. In the training session, we use click as reward signal, and optionally subtract oracle propensity to debias if needed. In evaluating session, true relevance label is used.

Algorithm

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 $\mathcal{B}$
  $\mu, \sigma = E_{\omega_1}(s, a), \quad \tilde{a} = D_{\omega_2}(s, z), \quad z \sim \mathcal{N}(\mu, \sigma)$
  $\omega \leftarrow \text{argmin}_\omega \sum (a - \tilde{a})^2 + D_{\text{KL}}(\mathcal{N}(\mu, \sigma) || \mathcal{N}(0, 1))$
  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)
  $\theta \leftarrow \text{argmin}_\theta \sum (y - Q_\theta(s, a))^2$
  $\phi \leftarrow \text{argmax}_\phi \sum Q_{\theta_1}(s, a + \xi_\phi(s, a, \Phi)), a \sim G_\omega(s)$
  Update target networks: $\theta'_i \leftarrow \tau\theta + (1 - \tau)\theta'_i$
  $\phi' \leftarrow \tau\phi + (1 - \tau)\phi'$
end for

Overall Training Procedure

Step 1: Off-policy data collection

For off-policy training, data is collected before training via a *behavior policy*, which is implemented by an *online MDPranker with randomly initialized parameters*. Specifically,

1. Rearrange train set (fold1/train.txt) and test set (fold1/test.txt) into a handy form: *PreprocessDataset* class.
2. Apply ranker to train set and get result ranking lists for each query.
3. Simulate user clicks via user click model (perfect, informational, navigational).
4. Repeat step 2 and step 3 for `NUM_SAMPLE` times, to get enough samples.

After the four steps above, off-policy data collection is done.

Step 2: BCQ training and evaluation

With collected data, we then implement BCQ algorithm to train a ranking policy. Specifically, we train five networks:

- ***policy Q network***: a.k.a. actor, choose the best action under current state based on Q value.
- ***two target Q network***: a.k.a. critic, criticize the action policy network chooses, and guide the policy network to do better. Here we implement two target network to prevent

overestimation bias.

- ***VAE network***: including *encoding* and *decoding* network, where the former encodes specific state and action pair into a Gaussian distribution, which is represented by its expectation and covariance; and the latter generates an action based on the distribution. Briefly speaking, VAE network acts as an *action generator* for every state.
- ***perturbation network***: perturb each action to increase the diversity of seen actions, which enables access to actions in a constrained region, without having to sample from the generative model a prohibitive number of times.

Every training epoch follows BCQ algorithm (details in *Algorithm* section). And we evaluate the current policy every `EVAL_EPISODE` epochs. The whole training process maintains `EPOCHS` epochs. The details of training process can be seen in *Detail* section.

Details

In this section, we tend to introduce the details in the BCQ implementation.

Hyperparameter

The hyperparameter is collected as a dictionary, where the meaning of each hyperparameter is commented after.

```
HYPERPARAMETERS = dict(
    # State and action dim
    # STATE_DIM = 46,  # MQ2007/MQ2008
    # ACTION_DIM = 46,
    STATE_DIM=136,  # MSLR_WEB10K
    ACTION_DIM=136,

    # Offline RL training hyperparams
    BATCH_SIZE = 256,
    GAMMA = 0.99,
    TARGET_UPDATE = 100,
    LR = 1e-3,
    # EPOCHS = 5000,  # for MQ2007/2008
    EPOCHS = 10000,  # for MSLR_WEB10K/30K
    TAU = 0.005,  # soft target update

    # Hyperparameters related to sample
    # MQ2007/MQ2008 (300 queries)
    # NUM_SAMPLE = 200,  # num of lists under each query
    # N_EVAL = 30,  # num of evaluation samples
    # EVAL_EPISODE = 10,  # evaluate interval
    # MEMORY_SIZE = 1000000,
    # MSLR_WEB10K/30K (10000 queries/30000 queries)
    NUM_SAMPLE = 30,  # num of lists under each query
    N_EVAL = 30,  # num of evaluation samples
    EVAL_EPISODE = 10,  # evaluate interval
    MEMORY_SIZE = 2000000,

    # BCQ hyperparams
    TARGET_LAMBDA = 0.75,  # soft target trade-off parameter
    PERTURB_RANGE = 0.2,  # perturb range of perturbation model
```

```
    # Click model
    CLICK_MODEL = ['perfect', 'informational', 'navigational'],
    END_POS = 10  # ndcg@10
)
```

Network Structure

The structure of the networks are defined below:

VAE network

```
class Encoder(nn.Module):
    r'''encoder net in VAE'''

    def __init__(self,
                 state_dims,
                 action_dims,
                 device) -> None:
        super(Encoder, self).__init__()
        self.device = device
        self.encoder = nn.Sequential(
            nn.Linear(state_dims + action_dims, 750, dtype=torch.float32),
            nn.ReLU(),
            nn.Linear(750, 750, dtype=torch.float32),
            nn.ReLU(),
            nn.Linear(750, 2 * action_dims, dtype=torch.float32)
        )

    def forward(self,
                state,
                action):
        state = state.to(self.device)
        action = action.to(self.device)
        return self.encoder(torch.cat((state, action), dim=1))

class Decoder(nn.Module):
    r'''decoder net in VAE'''

    def __init__(self,
                 state_dims,
                 latent_dims,
                 device) -> None:
        super(Decoder, self).__init__()
        self.latent_dims = latent_dims
        self.device = device
        self.decoder = nn.Sequential(
            nn.Linear(state_dims + latent_dims, 750, dtype=torch.float32),
            nn.ReLU(),
            nn.Linear(750, 750, dtype=torch.float32),
            nn.ReLU(),
            nn.Linear(750, latent_dims, dtype=torch.float32)
        )

    def forward(self,
                state,
```

```
                latent_vec=None):
        if latent_vec == None:
            latent_vec = torch.randn((state.shape[0],
self.latent_dims)).to(self.device).clamp(-0.5,0.5)
        state = state.to(self.device)
        latent_vec = latent_vec.to(self.device)
        return self.decoder(torch.cat((state, latent_vec), dim=1))

class VAE(object):
    r'''VAE is defined by two networks -- `encoder` and `decoder`.

        `encoder E(s,a)`: inputs a state-action pair and outputs the mean
        and standard deviation of a Gaussian distribution, with which we
        can construct latent variable `z` through reparameterization trick.

        `decoder D(s,z)`: inputs state and latent variable and output an action.

        While training, the loss is constructed by L2 loss, along with the KL
        divergence between standard normal distribution and normal distribution
        induced by encoder. See `Eq.28~30` in paper `Off-Policy Deep
Reinforcement
        Learning without Exploration` for details.'''

    def __init__(self,
                memory,
                hyperparameters) -> None:

        super(VAE, self).__init__()
        self.device = torch.device("cuda" if torch.cuda.is_available() else
"cpu")
        self.state_dims = hyperparameters['STATE_DIM']
        self.action_dims = hyperparameters['ACTION_DIM']
        self.batch_size = hyperparameters['BATCH_SIZE']
        self.memory = memory
        self.encoder = Encoder(hyperparameters['STATE_DIM'],\
            hyperparameters['ACTION_DIM'], self.device).to(self.device)
        self.decoder = Decoder(hyperparameters['STATE_DIM'], \
            hyperparameters['ACTION_DIM'], self.device).to(self.device)
        self.optimizer = optim.Adam(list(self.encoder.parameters()) +
list(self.decoder.parameters()))
        self.loss = nn.MSELoss()

    def sample(self, next_state_batch):
        r'''sample n actions from VAE'''

        # decode and get simulated action vectors for each state
        return self.decoder(next_state_batch, latent_vec=None)

    def train_vae(self,
                state_batch,
                action_batch):
        r'''train VAE in one epoch'''

        # encode and get mean and deviation vector
        mean_and_deviation = self.encoder(state_batch, action_batch)
```

```
        # sample latent vector for each mean-deviation pair
        normal_dist = normal.Normal(torch.tensor([0.0]), torch.tensor([1.0]))
        samples =
normal_dist.sample(torch.tensor([self.batch_size])).to(self.device)
        latent_vecs = (mean_and_deviation[:,:self.action_dims] \
            + samples * mean_and_deviation[:,self.action_dims:])

        # decode and get simulated action vectors for each state
        sim_action_batch = self.decoder(state_batch, latent_vecs)

        # loss calculation
        loss_reconstruct = self.loss(action_batch, sim_action_batch)
        square = mean_and_deviation**2
        loss_KL = -0.5 * (2 * self.action_dims +
torch.log(square[:,self.action_dims:])\
            .sum(dim=1, keepdim=True) - torch.sum(square, dim=1, keepdim=True))
        loss = loss_reconstruct + 1 / 2 / self.action_dims * torch.sum(loss_KL)

        # optimization
        self.optimizer.zero_grad()
        loss.backward()
        self.optimizer.step()
```

Q network

```
class DQN(nn.Module):
    r'''Q-network implementation

        `Input`: state(shape=[batch_size, STATE_DIM])
                action(shape=[batch_size, ACTION_DIM])

        `Output`: Q value(shape=[batch_size, 1])'''

    def __init__(self,
                state_dims,
                action_dims,
                device) -> None:
        super(DQN, self).__init__()
        self.device = device
        self.ln1 = nn.Linear(state_dims + action_dims, 64, dtype=torch.float32)
        self.ln2 = nn.Linear(64, 32, dtype=torch.float32)
        self.output = nn.Linear(32, 1, dtype=torch.float32)
        self.net = nn.Sequential(
            self.ln1,
            nn.ELU(),
            self.ln2,
            nn.ELU(),
            self.output,
        )

    def forward(self, state, action):
        state = state.to(self.device)
        action = action.to(self.device)
        return self.net(torch.cat((state, action), dim=1))
```

Perturbation network

```
class Perturbation(nn.Module):
    r'''perturb each simulated action'''

    def __init__(self,
                 state_dims,
                 action_dims,
                 perterb_range) -> None:
        super(Perturbation, self).__init__()
        self.device = torch.device("cuda" if torch.cuda.is_available() else
"cpu")
        self.perterb_range = perterb_range
        self.perturbation = nn.Sequential(
            nn.Linear(state_dims + action_dims, 400, dtype=torch.float32),
            nn.ReLU(),
            nn.Linear(400, 300, dtype=torch.float32),
            nn.ReLU(),
            nn.Linear(300, action_dims, dtype=torch.float32),
            nn.Tanh()  # serve as constraint
        )

    def forward(self, state, action):
        state = state.to(self.device)
        action = action.to(self.device)
        return self.perturbation(torch.cat((state, action),
dim=1))*self.perterb_range
```

Replay Buffer

In our replay buffer, different from the traditional `(state, action, reward, next_state, done)` pairs, we delete `done` signal and add `chosen` and `qid` in each pair.

- ***chosen***: a binary 1-D vector (shape=number of candidates(query id)), where 1 indicates the correspond document is available to click, 0 otherwise.
- ***qid***: str type, indicate the id of current query.

Then the pair restored in the replay buffer is: `(state, action, reward, next_state, chosen, qid)`

```
Transition = namedtuple('Transition', ('state', 'action', 'next_state', 'chosen',
'qid', 'reward'))

class ReplayMemory(object):
    '''Replay memory to store experiences'''

    def __init__(self, capacity) -> None:
        self.memory = deque([], maxlen=capacity)

    def push(self, *args):
        '''Save a transition in memory buffer'''
        self.memory.append(Transition(*args))
```

```
    def sample(self, batch_size):
        return random.sample(self.memory, batch_size)

    def __len__(self):
        return len(self.memory)
```

Training Procedure in One Epoch

```
def BCQTrainOneStep(self):
    '''train one batch via double BCQ alg'''

    trainsitions = self.memory.sample(self.hyperparameters['BATCH_SIZE'])
    batch = R.Transition(*zip(*trainsitions))

    # gather batch
    state_batch = torch.cat(batch.state).to(self.device)
    action_batch = torch.cat(batch.action).to(self.device)
    next_state_batch = torch.cat(batch.next_state).to(self.device)
    reward_batch = torch.cat(batch.reward).to(self.device)

    # train VAE to get action generator
    self.vae_net.train_vae(state_batch, action_batch)

    # sample new actions under next states and add perturbation
    sim_action_batch = self.vae_net.sample(next_state_batch)
    action_with_perturb = sim_action_batch + \
    self.perturbation(next_state_batch, sim_action_batch)

    # policy net optimization
    # set target and calculate loss
    min_Q = self.hyperparameters['TARGET_LAMBDA'] * \
        torch.min(self.target_net1(next_state_batch, action_with_perturb),\
              self.target_net2(next_state_batch, action_with_perturb))
    max_Q = (1 - self.hyperparameters['TARGET_LAMBDA']) * \
        torch.max(self.target_net1(next_state_batch, action_with_perturb),\
              self.target_net2(next_state_batch, action_with_perturb))
    target_value = reward_batch + self.hyperparameters['GAMMA'] *
torch.max(min_Q, max_Q)

    policyloss = self.policy_loss(self.policy_net(state_batch, action_batch),
target_value)
    self.policy_optimizer.zero_grad()
    policyloss.backward()
    self.policy_optimizer.step()

    # perturbation optimization
    # set target and calculate loss
    action_with_perturb_target = action_batch + \
    self.perturbation(state_batch, action_batch)
    perturbation_target = -torch.mean(self.policy_net(state_batch,
action_with_perturb_target))

    self.perturbation_optimizer.zero_grad()
    perturbation_target.backward()
    self.policy_optimizer.step()
```

```
    # target network soft update
    for param, target1_param in zip(self.policy_net.parameters(),
self.target_net1.parameters()):
        target1_param.data.copy_(self.hyperparameters['TAU'] * param.data + \
                                 (1 - self.hyperparameters['TAU']) *
target1_param.data)
    for param, target2_param in zip(self.policy_net.parameters(),
self.target_net2.parameters()):
        target2_param.data.copy_(self.hyperparameters['TAU'] * param.data + \
                                 (1 - self.hyperparameters['TAU']) *
target2_param.data)
    for param, perturb_param in zip(self.perturbation.parameters(),
self.perturbation_target.parameters()):
        perturb_param.data.copy_(self.hyperparameters['TAU'] * param.data + \
                                 (1 - self.hyperparameters['TAU']) *
perturb_param.data)

    return self.policy_net(state_batch, action_batch).mean().item(),\
        target_value.mean().item(),\
        policyloss.item()
```