Munchausen-RL

Paper Link: https://arxiv.org/pdf/2007.14430.pdf

Key Features

- M-RL(Munchausen-RL) is a method of optimizing based on immediate reward augmented by adding scaled log-policy of the agent using bootstrapping method.
- M-RL learns for maximizing the entropy of policies and cumulative reward.
- DQN was selected as the subject of application for M-RL. However, DQN cannot calculate log-policy because it uses deterministic policy. Therefore, M-RL is applied to Soft-DQN(S-DQN), so M-DQN(Munchausen-DQN) can be implemented.
- M-RL also supports the M-IQN agent which is the version of IQN.

Background

Modification of regression target : M-DQN

In the paper, M-RL method is applied at DQN to evaluate its effect (M-DQN).

M-DQN assumes usage of stochastic policies while DQN, which is baseline of M-DQN, computes Q-function by deterministic policies as follows.

$$\hat{q}_{dqn}(r_t, s_{t+1}) = r_t + \gamma \sum_{a' \in \mathcal{A}} \pi_{\bar{\theta}}(a'|s_{t+1}) q_{\bar{\theta}}(s_{t+1}, a') \; with \; \pi_{\bar{\theta}} \in \mathcal{G}(q_{\bar{\theta}})$$

Therefore, the proposed method changed the setting of DQN and it is called S-DQN(Soft-DQN). The Q-function equation of S-DQN is as follows.

$$
\begin{aligned}
&\hat{q}_{s-dqn}(r_t, s_{t+1}) \\
&= r_t + \gamma \sum_{a' \in \mathcal{A}} \pi_{\bar{\theta}}(a'|s_{t+1})(q_{\bar{\theta}}(s_{t+1}, a') - \tau \ln \pi_{\bar{\theta}}(a'|s_{t+1}))\ with\ \pi_{\bar{\theta}} = sm(\frac{q_{\bar{\theta}}}{\tau})
\end{aligned}
$$

Q-function of M-DQN can be derived by adding a scaled log-policy term to the equation of S-DQN.

$$
\begin{aligned}
&\hat{q}_{m-dqn}(r_t, s_{t+1}) \\
&= r_t + \alpha\tau \ln \pi_{\bar{\theta}}(a_t, s_t) + \gamma \sum_{a' \in \mathcal{A}} \pi_{\bar{\theta}}(a'|s_{t+1})(q_{\bar{\theta}}(s_{t+1}, a') - \tau \ln \pi_{\bar{\theta}}(a'|s_{t+1})) \\
&where\ scaling\ factor\ \alpha \in [0,\ 1]
\end{aligned}
$$

Since the difference from S-DQN is only term of $\alpha\tau \ln \pi_{\bar{\theta}}(a_t, s_t)$, M-DQN can be obtained by slightly modifying regression target.

The method of M-RL can also be applied to the IQN algorithm. It is called M-IQN.

$$
\begin{aligned}
&\hat{q}_{m-iqn}(r_t, s_{t+1}) \\
&= r_t + \lambda\tau \ln \pi_{\beta}(a_t, s_t) + \gamma \sum_{a' \in \mathcal{A}} \pi_{\beta}(a'|s_{t+1})(Z_{\tau'}(s_{t+1}, \pi_{\beta}(a'|s_{t+1})) - \tau \ln \pi_{\beta}(a'|s_{t+1})) \\
&where\ scaling\ factor\ \lambda \in [0, 1], \\
&distortion\ risk\ measure\ \beta \in U([0, 1]), \\
&and\ new\ distribution\ \tau' \sim U([0, 1])
\end{aligned}
$$

Method

Algorithm

Algorithm 1 Munchausen DQN

Require: $T \in \mathbb{N}^*$ the number of environment steps, $C \in \mathbb{N}^*$ the update period, $F \in \mathbb{N}^*$ the interaction period.
  Initialize θ at random
  $\mathcal{B} = \{\}$
  $\bar{\theta} = \theta$
  for $t = 1$ **to** T **do**
    Collect a transition $b = (s_t, a_t, r_t, s_{t+1})$ from $\mathcal{G}_e(\theta)$
    $\mathcal{B} \leftarrow \mathcal{B} \cup \{b\}$
    if $t \mod F == 0$ **then**
      On a random batch of transitions $B_t \subset \mathcal{B}$, update θ with one step of SGD on $\mathcal{L}_{\text{m-dqn}}$, see (7)
    end if
    if $k \mod C == 0$ **then**
      $\bar{\theta} \leftarrow \theta$
    end if
  end for
  return $\mathcal{G}_0(\theta)$

Implementation on JORLDY

- M-DQN JORLDY Implementation

```
### M-DQN's learn function ###
# calcuate M-DQN's target q = reward + munchausen_term + gamma * maximum_entropy_term
# munchausen_term = alpha * (tau) * log_policy
# maximum_entropy_term = next_policy * (next_target_q - (tau) * next_log_policy)

def learn(self):
...

    eye = torch.eye(self.action_size).to(self.device)
    one_hot_action = eye[action.view(-1).long()]
    q = (self.network(state) * one_hot_action).sum(1, keepdims=True)

    with torch.no_grad():
        max_Q = torch.max(q).item()
        next_target_q = self.target_network(next_state)

        # calculate M-DQN's target q
        target_q = self.target_network(state)
```

```
        log_policy = (
            stable_scaled_log_softmax(target_q, self.tau) * one_hot_action
        ).sum(-1, keepdims=True)
        clipped_log_policy = torch.clip(log_policy, min=self.l_0, max=0)

        next_log_policy = stable_scaled_log_softmax(next_target_q, self.tau)
        next_policy = stable_softmax(next_target_q, self.tau)

        munchausen_term = self.alpha * clipped_log_policy
        maximum_entropy_term = (
            next_policy * (next_target_q - next_log_policy)
        ).sum(-1, keepdims=True)

        target_q = (
            reward
            + munchausen_term
            + (1 - done) * self.gamma * maximum_entropy_term
        )

...
```

- M-IQN JORLDY Implementation

```
### M-IQN's learn function ###
# calcuate M-IQN's target theta = reward + munchausen_term + gamma * maximum_entropy_term
# munchausen_term = self.alpha * clipped_log_policy
# maximum_entropy_term = next_policy * (logits_target - next_log_policy)

def learn(self):
...

    # Get Theta Pred, Tau
    logit, tau = self.network(state)
    logits, q_action = self.logits2Q(logit)
    action_eye = torch.eye(self.action_size, device=self.device)
    action_onehot = action_eye[action.long()]

    theta_pred = action_onehot @ logits
    tau = torch.transpose(tau, 1, 2).contiguous()

    with torch.no_grad():
        # Get Theta Target
        logit_next, _ = self.network(next_state)
        _, q_next = self.logits2Q(logit_next)

        logit_target, _ = self.target_network(next_state)
        logits_target, next_target_q = self.logits2Q(logit_target)

        max_a = torch.argmax(q_next, axis=-1, keepdim=True)
        max_a_onehot = action_eye[max_a.long()]

        # calculate M-IQN's target theta & loss
```

```
        logit, _ = self.network(state)
        _, target_q = self.logits2Q(logit)

        log_policy = (
            stable_scaled_log_softmax(target_q, self.tau) * action_onehot.squeeze()
        ).sum(-1, keepdims=True)
        clipped_log_policy = torch.clip(log_policy, min=self.l_0, max=0)

        munchausen_term = self.alpha * clipped_log_policy

        next_log_policy = (
            stable_scaled_log_softmax(next_target_q, self.tau)
            .unsqueeze(2)
            .repeat(1, 1, self.num_support)
        )
        next_policy = (
            stable_softmax(next_target_q, self.tau)
            .unsqueeze(2)
            .repeat(1, 1, self.num_support)
        )

        maximum_entropy_term = (
            next_policy * (logits_target - next_log_policy)
        ).sum(1)

        theta_target = (
            reward
            + munchausen_term
            + (1 - done) * self.gamma * maximum_entropy_term
        )

        theta_target = torch.unsqueeze(theta_target, 2)

    error_loss = theta_target - theta_pred
    huber_loss = F.smooth_l1_loss(
        *torch.broadcast_tensors(theta_pred, theta_target), reduction="none"
    )

    # Get Loss
    loss = torch.where(error_loss < 0.0, 1 - tau, tau) * huber_loss
    loss = torch.mean(torch.sum(loss, axis=2))

    max_Q = torch.max(q_action).item()
    max_logit = torch.max(logit).item()
    min_logit = torch.min(logit).item()

...
```

References

Relevant papers

- Implicit Quantile Networks for Distributional Reinforcement Learning(IQN)
  (W. Dabney et al, 2018)

Public implementations

- munchausen-rl
  (google-research)