❓

RND(Random Network Distillation)

Paper Link: https://arxiv.org/pdf/1810.12894.pdf

Key Features

- Random Network Distillation(RND) is a technique that increases property of exploration by applying an exploration bonus as intrinsic reward term.
- The exploration bonus used by RND is the prediction error of agent with respect to the target which extracted from fixed and random neural network. The agent is learned by approximating its own prediction of the next state to the target.
- RND defines a state which is difficult to predict as a 'novel state' that requires learning and encourages exploration.
- Reward can be evaluated by separating intrinsic reward and extrinsic reward. Separate discount rates are applied to the each reward.
- RND is easy to implement and can be applied to the environment which has high-dimensional observation. It is also available with all policy optimization algorithms.

Background

Limitation of conventional RL for sparse-reward environment

In the return-based learning method of conventional RL, agent is difficult to learn the environment with a low reward density. To solve this problem, a method that encourages

the agent to the good state is needed. RND uses two kinds of network to solve this problem. It is target network and predicator network respectively.

Method

How can measure the novelty of state? → Exploration bonus

The conventional RL agent is trained by only extrinsic reward. Unlike conventional RL agent, agent using RND adds intrinsic reward that quantifies ‘novelty of state’. It is called as ‘exploration bonus’. Then, how can this exploration bonus be calculated? The procedure is as follows.

1. The next observation is an input into the target network which embeds the observation as $f : \mathcal{O} \rightarrow \mathbb{R}^k$ to predict the hidden vector of the next state. Target network is initialized and fixed during training process. Next state extracted from the target network is used as the target value for the agent to learn the state transition of the environment.
2. The next state, which is extracted from the predictor network, is embedded as $\hat{f} : \mathcal{O} \rightarrow \mathbb{R}^k$.
3. Calculate the exploration bonus by computing the error between the predicted value and the target value. If a state is repeatedly visited, the predictor network is trained many times to predict the hidden state obtained by the target network. Then, the prediction becomes easier and the exploration bonus gradually decreases. This shows the meaning of "the state which is difficult to predict is a novel state".

Combination of the extrinsic reward and the intrinsic reward

In the case of using intrinsic reward, non-episodic reward is effective in in improving property of exploration.

However, if extrinsic reward is used as non-episodic reward, the agent can be trained to choose short-sighted policy. For example, agent chooses the current reward over future uncertainty by terminating the task early.

Therefore, RND separates value function for extrinsic and intrinsic reward. Due to the separation of the value function, different discount rates can be applied for each reward.

Normalization with respect to reward and observation

To improve performance, RND normalizes for reward and observation. At first, the reward size varies according to environment or training step. It makes hard to set proper hyperparameters of agent for training. To solve this, the reward is maintained on a consistent scale by dividing it by a running estimate of the standard deviation.

When normalization for observation is insufficient, it may be difficult to transmit input information due to low embedding variance. Because the parameters of target network are fixed and hence cannot adjust to the scale of different datasets. To solve this, agent normalizes observation. This is performed by subtracting the running mean for each observation and dividing it by the running standard deviation. Thereafter, the observation is limited to the range of [-5,+5]. The same observation normalization is applied to both predictor and target networks but not the policy network.

Implementation on JORLDY

- RND_PPO JORLDY Implementation

```
### learn function ###
# 1. Calculate intrinsic reward
# 2. Calculate policy_old, state-value, next state-value and advantage
# 3. Train iteration(PPO)
#    - Update Actor, Critic
#    - After that, update RND

def learn(self):
...

    # use extrinsic check
```

```
    if self.non_extrinsic:
        reward *= 0.0

    # set pi_old and advantage
    with torch.no_grad():
        # RND: calculate exploration reward, update moments of obs and r_i
        self.rnd.update_rms_obs(next_state)
        r_i = self.rnd(next_state, update_ri=True)

        if self.action_type == "continuous":
            mu, std, value = self.network(state)
            m = Normal(mu, std)
            z = torch.atanh(torch.clamp(action, -1 + 1e-7, 1 - 1e-7))
            log_prob = m.log_prob(z)
        else:
            pi, value = self.network(state)
            log_prob = pi.gather(1, action.long()).log()
        log_prob_old = log_prob
        v_i = self.network.get_v_i(state)

        next_value = self.network(next_state)[-1]
        delta = reward + (1 - done) * self.gamma * next_value - value

        next_v_i = self.network.get_v_i(next_state)
        episodic_factor = 1.0 if self.non_episodic else (1 - done)
        delta_i = r_i + episodic_factor * self.gamma_i * next_v_i - v_i

        adv, adv_i = delta.clone(), delta_i.clone()
        adv, adv_i = adv.view(-1, self.n_step), adv_i.view(-1, self.n_step)
        done = done.view(-1, self.n_step)

        for t in reversed(range(self.n_step - 1)):
            adv[:, t] += (
                (1 - done[:, t]) * self.gamma * self._lambda * adv[:, t + 1]
            )
            episodic_factor = 1.0 if self.non_episodic else (1 - done[:, t])
            adv_i[:, t] += (
                episodic_factor * self.gamma_i * self._lambda * adv_i[:, t + 1]
            )

        ret = adv.view(-1, 1) + value
        ret_i = adv_i.view(-1, 1) + v_i

        adv = self.extrinsic_coeff * adv + self.intrinsic_coeff * adv_i

        if self.use_standardization:
            adv = (adv - adv.mean(dim=1, keepdim=True)) / (
                adv.std(dim=1, keepdim=True) + 1e-7
            )

        adv, done = adv.view(-1, 1), done.view(-1, 1)

    mean_ret = ret.mean().item()
    mean_ret_i = ret_i.mean().item()
```

```
# start train iteration
actor_losses, critic_e_losses, critic_i_losses = [], [], []
entropy_losses, rnd_losses, ratios, probs = [], [], [], []
idxs = np.arange(len(reward))
for idx_epoch in range(self.n_epoch):
    np.random.shuffle(idxs)
    for offset in range(0, len(reward), self.batch_size):
        idx = idxs[offset : offset + self.batch_size]
        (
            _state,
            _action,
            _value,
            _v_i,
            _ret,
            _ret_i,
            _next_state,
            _adv,
            _log_prob_old,
        ) = map(
            lambda x: [_x[idx] for _x in x] if isinstance(x, list) else x[idx],
            [
                state,
                action,
                value,
                v_i,
                ret,
                ret_i,
                next_state,
                adv,
                log_prob_old,
            ],
        )

        if self.action_type == "continuous":
            mu, std, value_pred = self.network(_state)
            m = Normal(mu, std)
            z = torch.atanh(torch.clamp(_action, -1 + 1e-7, 1 - 1e-7))
            log_prob = m.log_prob(z)
        else:
            pi, value_pred = self.network(_state)
            m = Categorical(pi)
            log_prob = m.log_prob(_action.squeeze(-1)).unsqueeze(-1)
        value_i = self.network.get_v_i(_state)

        ratio = (log_prob - _log_prob_old).sum(1, keepdim=True).exp()
        surr1 = ratio * _adv
        surr2 = (
            torch.clamp(
                ratio, min=1 - self.epsilon_clip, max=1 + self.epsilon_clip
            )
            * _adv
        )
        actor_loss = -torch.min(surr1, surr2).mean()
```

```
            # Critic Clipping
            value_pred_clipped = _value + torch.clamp(
                value_pred - _value, -self.epsilon_clip, self.epsilon_clip
            )

            critic_loss1 = F.mse_loss(value_pred, _ret)
            critic_loss2 = F.mse_loss(value_pred_clipped, _ret)

            critic_e_loss = torch.max(critic_loss1, critic_loss2).mean()

            # Critic Clipping (intrinsic)
            value_i_clipped = _v_i + torch.clamp(
                value_i - _v_i, -self.epsilon_clip, self.epsilon_clip
            )

            critic_i_loss1 = F.mse_loss(value_i, _ret_i)
            critic_i_loss2 = F.mse_loss(value_i_clipped, _ret_i)

            critic_i_loss = torch.max(critic_i_loss1, critic_i_loss2).mean()

            critic_loss = critic_e_loss + critic_i_loss

            entropy_loss = -m.entropy().mean()
            loss = (
                actor_loss
                + self.vf_coef * critic_loss
                + self.ent_coef * entropy_loss
            )

            _r_i = self.rnd.forward(_next_state)
            rnd_loss = _r_i.mean()

            self.optimizer.zero_grad(set_to_none=True)
            loss.backward()
            torch.nn.utils.clip_grad_norm_(
                self.network.parameters(), self.clip_grad_norm
            )
            self.optimizer.step()

            self.rnd_optimizer.zero_grad(set_to_none=True)
            rnd_loss.backward()
            torch.nn.utils.clip_grad_norm_(
                self.rnd.parameters(), self.clip_grad_norm
            )
            self.rnd_optimizer.step()

...
```