ICM(Intrinsic Curiosity Module)

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

Key Features

- Exploration based on curiosity is a technique which encourage agent to explore to new state without randomness.
- Curiosity is obtained by a module called ICM(Intrinsic Curiosity Module) that derives intrinsic rewards by learning the dynamic model of the environment.
- The ICM consists of an Inverse module and a Forward module. Inverse module predicts action from an encoded state and encoded next state. Forward module predicts encoded next state from an encoded state and action.

Background

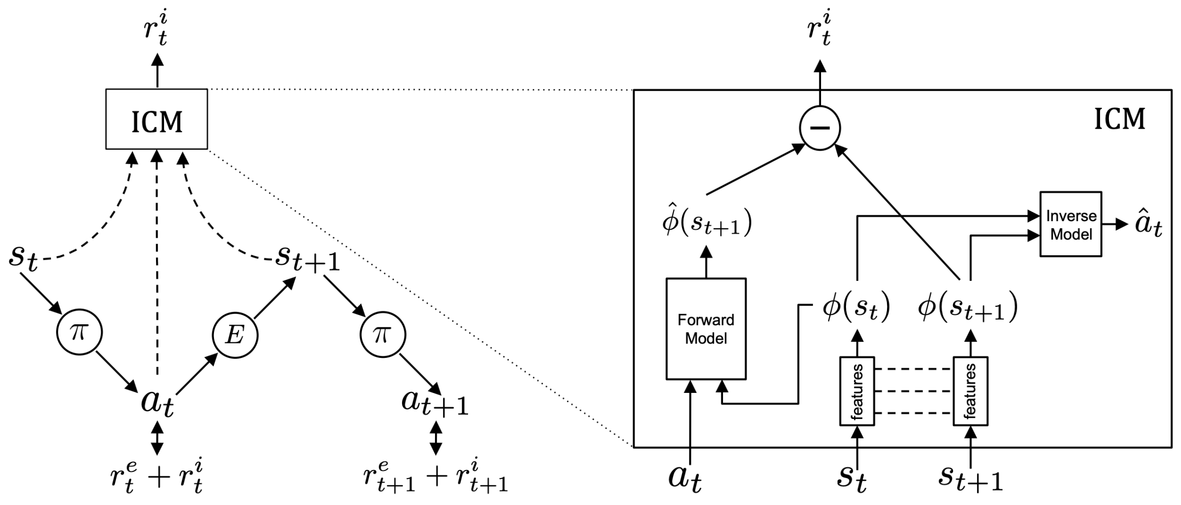
Limitation of single goal-oriented reward system

The goal of conventional RL is ‘maximization of cumulative reward(return)’. Each reward, which constitutes the return, increases the performance of the agent with respect to target task. However, the blind spot of the reward structure is that the policy is updated only when it succeeds in reaching the ideal state. Because most exploration policies are random. Reaching the ideal state based on randomness cannot effectively work in an environment where learning is difficult. ICM suggests a method of conducting extensive environmental exploration based on the material called 'curiosity' . This curiosity is expressed as an intrinsic reward existing inside the agent, not a reward given from the environment. This intrinsic reward plays an important role especially when the extrinsic reward is sparse.

Method

Subsystems of agent

The agent uses ICM as subsystem. The figure is shown as follows.



First of all, the policy π is updated based on reward as in the conventional RL. At this time, the reward obtained by summation of the agent's intrinsic reward r_t^i extracted from ICM and the extrinsic reward r_t^e given from the environment. So, objective function of policy π parameterized with θ_P is as follows.

$$J(s_t, \theta_P) = \underset{\theta_P}{max} \mathbb{E}_\pi (s_t; \theta_P)[\sum_t (r_t^i + r_t^e)]$$

ICM is effective for sparse reward environment. The sparse reward environment is an environment in which good rewards rarely occur. ICM induces the agent to a new state based on curiosity in such a difficult environment to receive extrinsic rewards.

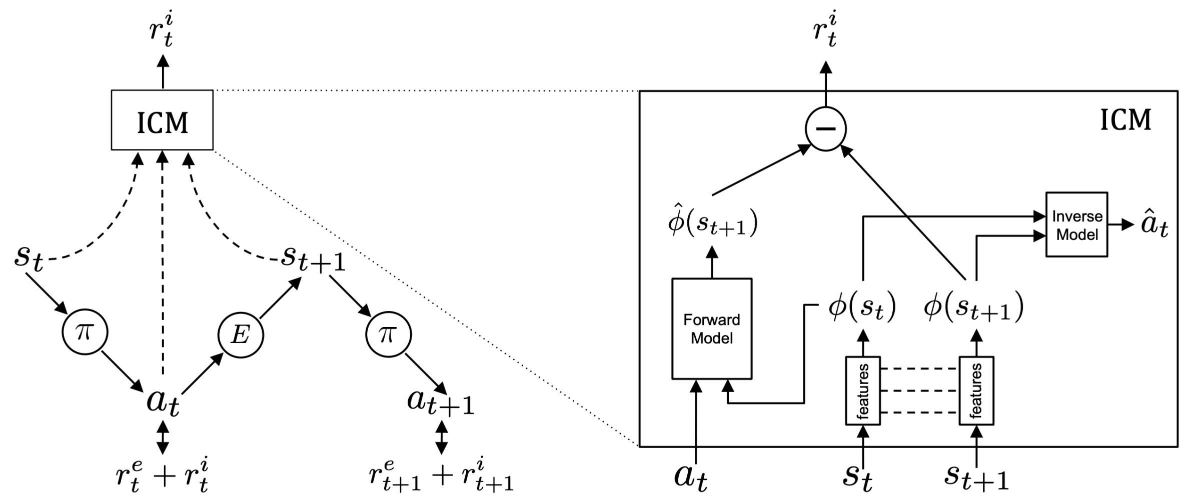
Why ICM learns with respect to feature input?

ICM do not learn based on raw image pixels, because raw image is more likely to have noise. This characteristic of raw image data can interfere the effective exploration of the agent. Then, what kind of feature space is appropriate for learning? The answers are as follows.

- The change which can be controlled by the agent
- The change which cannot be controlled but affect on the agent

To solve the above-described problem, the agent needs a generalized mechanism capable of valid feature expressions during learning. In this paper, the raw image is encoded as a vector using the neural network.

ICM = Inverse + Forward



The above figure shows the structure of ICM. The ICM consists of two models. All of the models are implemented as neural network.

The first model is the inverse model. The inverse model estimates action $\hat{a}_t$ based on the encoded state $\hat{\phi}(s_t)$ and the encoded next state $\hat{\phi}(s_{t+1})$. The agent can estimate its own action based on the state transition through the ICM.

$$\hat{a}_t = g(s_t, s_{t+1}; \theta_I)$$
$$\underset{\theta_I}{min} L_I(\hat{a}_t, a_t)$$

In the above equation, θ_I which is a parameter of Inverse model learns in the direction of minimizing the difference between real action a_t and predicted action $\hat{a}_t$. At this time, if the predicted action $\hat{a}_t$ is a discrete value, the output of dynamics is softmax output(probability), which is the same as having the maximum likelihood in a multinomial distribution.

The second model of ICM is the forward model. The forward model estimates the encoded next state $\hat{\phi}(s_{t+1})$ based on the encoded state $\hat{\phi}(s_t)$ and action a_t as follows.

$$\hat{\phi}(s_{t+1}) = f(\phi(s_t), a_t; \theta_F)$$

The above equation shows that the forward model is parameterized as a neural network θ_F. The loss function L_F for this neural network is as follows.

$$L_F(\phi(s_t), \hat{\phi}(s_{t+1})) = \frac{1}{2}||\hat{\phi}(s_{t+1}) - \phi(s_{t+1})||_2^2$$

The equation of the intrinsic reward is as follows.

$$r_t^i = \frac{\eta}{2}||\hat{\phi}(s_{t+1}) - \phi(s_{t+1})||_2^2$$

In the above equation, η is a scaling element and is always greater than zero.

Loss function with ICM

If all loss functions above are combined, total loss function can be derived as follows.

$$\underset{\theta_P,\theta_I,\theta_F}{min}[-\lambda\mathbb{E}_{\pi(s_t;\theta_P)}[\sum_t r_t] + (1-\beta)L_I + \beta L_F]$$

The coefficient β is the weight indicating the importance of the inverse model against the forward model, and the coefficient λ is the weight indicating the importance of policy gradient loss against the intrinsic reward. λ ranges from $(0,1]$.

Implementation on JORLDY

- ICM-PPO JORLDY Implementation

```
### learn function ###
# 1. Set policy_old and advantage
# 2. Train iteration(PPO)
#    - Update Actor, Critic
#    - After that, update ICM

def learn(self):
...

    # set prob_a_old and advantage
    with torch.no_grad():
        # ICM
        self.icm.update_rms_obs(next_state)
        r_i, _, _ = self.icm(state, action, next_state, update_ri=True)
        reward = (
            self.extrinsic_coeff * reward + self.intrinsic_coeff * r_i.unsqueeze(1)
        )

        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

        next_value = self.network(next_state)[-1]
        delta = reward + (1 - done) * self.gamma * next_value - value
        adv = delta.clone()
        adv, done = adv.view(-1, self.n_step), 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]
            )

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

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

    mean_ret = ret.mean().item()

    # start train iteration
    actor_losses, critic_losses, entropy_losses, ratios, probs = [], [], [], [], []
    idxs = np.arange(len(reward))
    for _ 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, _ret, _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, ret, 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)
            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()

            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_loss = torch.max(critic_loss1, critic_loss2).mean()
            entropy_loss = -m.entropy().mean()

            # ICM
            _, l_f, l_i = self.icm(_state, _action, _next_state)

            loss = self.lamb * (
                actor_loss
                + self.vf_coef * critic_loss
                + self.ent_coef * entropy_loss
            )

            icm_loss = self.beta * l_f + (1 - self.beta) * l_i

            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.icm_optimizer.zero_grad(set_to_none=True)
            icm_loss.backward()
            torch.nn.utils.clip_grad_norm_(
                self.icm.parameters(), self.clip_grad_norm
            )
            self.icm_optimizer.step()

...
```