V-MPO(On-Policy Maximum a Posteriori Policy Optimization)

Paper Link: https://arxiv.org/abs/1909.12238

Key Features

- V-MPO is an on-policy reinforcement learning algorithm that can learn policy through policy iteration based on value function.
- V-MPO is effective in controlling high-dimensional action space without using policy gradient, importance sampling and entropy regularization.
- V-MPO is applicable to both discrete and continuous action space tasks.

Background

Limitation of policy gradient method

In the policy gradient method, the performance of the high-dimensional action space control problem may be limited by the bias of the policy gradient. To solve this problem, entropy regularization is commonly used. However, optimization is performed through an iterative policy improvement based on a state value function V in V-MPO.

Method

Characteristic of V-MPO

V-MPO is an approximate policy iteration algorithm with a specific prescription for the policy improvement step. In general, policy iteration uses the fact that the true state value function V_π corresponding to policy π can be used to obtain an improved policy π'. Thus V-MPO can do the following standard process.

1. Sample a trajectory using target policy

2. Evaluate target policy based on learned state value function V and calculate advantage A for each action.

In V-MPO, one more process is added to improve behavioral policy B based on the advantage A of the target policy.

The optimization method of V-MPO sets a non-parametric target and partially moves the parameter of policy in the direction of the non-parametric target according to KL constraint. In this process, the trajectory consists of n-step batch data and last of the data is bootstrapped.

Loss function

The total loss function of V-MPO is as follows.

$$\mathcal{L}_\pi(\phi, \theta, \eta, \alpha) = \mathcal{L}_V(\phi) + \mathcal{L}_{V-MPO}(\theta, \eta, \alpha)$$

where ϕ is a parameter of a value function V, θ is a parameter of a policy, and η and α are Lagrangian multipliers. Usually, parameter of policy θ and value function ϕ are shared.

It should be noted that the ϕ is fixed as a constant in calculating policy improvement loss $\mathcal{L}_{V-MPO}(\theta, \eta, \alpha)$.

The policy evaluation loss is a standard regression of the n-step batch data and is derived as follows. This is a loss function for parametric state function $V_\phi^\pi(s)$.

$$\mathcal{L}_V(\phi) = \frac{1}{2|\mathcal{D}|} \sum_{s_t \sim \mathcal{D}} (V_\phi^\pi(s_t) - G_t^{(n)})^2$$

The policy improvement loss is as follows.

$$\mathcal{L}_{V-MPO}(\theta, \eta, \alpha) = \mathcal{L}_\pi(\theta) + \mathcal{L}_\eta(\eta) + \mathcal{L}_\alpha(\theta, \alpha)$$

The policy loss term of the above is the weighted maximum likelihood loss as follows**.**

$$\mathcal{L}_\pi(\theta) = -\sum_{s,a\sim\tilde{\mathcal{D}}} \psi(s,a)\log\pi_\theta(a|s), \quad \psi(s,a) = \frac{exp(\frac{A^{target}(s,a)}{\eta})}{\sum_{s,a\sim\tilde{\mathcal{D}}} exp(\frac{A^{target}(s,a)}{\eta})}$$

In this case, A^{target} is advantage of target, and $\tilde{\mathcal{D}}$ represents data which is top 50% of the priority among the batch data.

In addition, the loss of η and α are as follows.

$$\mathcal{L}_\eta(\eta) = \eta\epsilon_\eta + \eta\log[\frac{1}{\tilde{\mathcal{D}}}\sum_{s,a\sim\tilde{\mathcal{D}}} exp(\frac{A^{target}(s,a)}{\eta})]$$

KL constraint, which represents trust region, is as follows. The term of sg[[·]] indicates a stop gradient. In this equation, batch data is full-batch.

$$\mathcal{L}_\alpha(\theta,\alpha) = \frac{1}{|\mathcal{D}|}\sum_{s\in\mathcal{D}}[\alpha(\epsilon_\alpha - sg[[D_{KL}(\pi_{\theta_{target}}(a|s)||\pi_\theta(a|s))]]) + sg[[\alpha]]D_{KL}(\pi_{\theta_{target}}(a|s)||\pi_\theta(a|s))]$$

Policy Evaluation

In V-MPO, policy evaluation means learning the approximate state value function V when a policy $\pi(a|s)$ is given. The batch data is fixed in the target learning stage, and V corresponding to the target policy is instantiated in the online network receiving gradient updates. According to this setting, the bootstrapping technique may obtain an optimal value.

The policy makes new target based on the updated V. While update of V is performed for the current value function parameter ϕ, which can be shared with current policy parameter θ. Therefore, the policy π is to be treated as an $\pi_{\theta_{old}}$ while V is evaluated on the current parameter ϕ. Accordingly, the loss function of the V_ϕ is as follows.

$$\mathcal{L}_V(\phi) = \frac{1}{2|\mathcal{D}|}\sum_{s_t\sim\mathcal{D}}(V_\phi^\pi(s_t) - G_t^{(n)})^2$$

G is the bootstrapped n-step return and advantage A, which is the main interest in the V-MPO policy improvement step, is calculated as $G-V$.

Policy Improvement

This section shows how to evaluate improved new policy when the old policy based advantage A is given. Prior to the explanation, $\mathcal{I}$ is an indicator of a binary event for policy improvement. If $\mathcal{I}=1$, it represents an event that successfully improves the policy. If $I=0$, it is the opposite. V-MPO seeks for the posterior distribution mode of parameter θ which is maximum a posterior probability(MAP) based on this event. the equation of optimal θ for MAP is as follows.

$$\theta^* = \underset{\theta}{argmax}[\log p_\theta(\mathcal{I}=1) + \log p(\theta)]$$

In this process, the calculation of $\log p(X)$ is calculated for the latent distribution as follows.

$$\log p(X) = \mathbb{E}_{\psi(Z)}[\log \frac{p(X,Z)}{\psi(Z)}] + D_{KL}(\psi(Z)||p(Z|X))$$

The first term of this equation means lower bound. This is because the KL term value is always non-negative. Therefore, the equation for calculating $\log p_\theta(\mathcal{I}=1)$ is as follows.

$$\log p_\theta(\mathcal{I}=1) = \sum_{s,a}\psi(s,a)\log\frac{p_\theta(\mathcal{I}=1,s,a)}{\psi(s,a)} + D_{KL}(\psi(s,a)||p_\theta(s,a|\mathcal{I}=1))$$

Based on the above equations, V-MPO can improve policy through EM algorithm.

E-step

In the E-step, goal of V-MPO is to choose the variational distribution $\psi(s,a)$ like first equation such that the lower bound on $\log p_\theta(\mathcal{I}=1)$ is as tight as possible. It is the case when the KL term on the third equation is zero.

$$\psi(s,a) = \frac{p_{\theta_{old}}(s,a)p_{\theta_{old}}(\mathcal{I}=1|s,a)}{p_{\theta_{old}}(\mathcal{I}=1)}, \quad p_{\theta_{old}} = \sum_{s,a} p_{\theta_{old}}(s,a)p_{\theta_{old}}(\mathcal{I}=1|s,a)$$

$$\log p_\theta(\mathcal{I}=1) = \sum_{s,a} \psi(s,a) \log \frac{p_\theta(\mathcal{I}=1,s,a)}{\psi(s,a)} + D_{KL}(\psi(s,a)||p_\theta(s,a|\mathcal{I}=1))$$

This solution weights the probability of each state-action pair based on the probability of case $(\mathcal{I}=1)$. The equation completed by applying Advantage A on above $p_{\theta_{old}}$ has the following proportional relationship. Especially, η is greater than zero.

$$p_{\theta_{old}}(\mathcal{I}=1|s,a) \propto exp(\frac{A^{\pi_{\theta_{old}}}(s,a)}{\eta})$$

This can control the diversity of actions that contribute to weights through modifying η, but now η set to arbitrary value.

M-step

V-MPO finds a nonparametric variational distribution on the E-step that gives a lower bound constrained according to the following equations. So, V-MPO can maximize this lower bound together with the prior $\log p(\theta)$ with respect to parameters θ in M-step.

$$\log p_\theta(\mathcal{I}=1) = \sum_{s,a} \psi(s,a) \log \frac{p_\theta(\mathcal{I}=1,s,a)}{\psi(s,a)} + D_{KL}(\psi(s,a)||p_\theta(s,a|\mathcal{I}=1))$$

$$\psi(s,a) = \frac{p_{\theta_{old}}(s,a)p_{\theta_{old}}(\mathcal{I}=1|s,a)}{p_{\theta_{old}}(\mathcal{I}=1)}, \quad p_{\theta_{old}} = \sum_{s,a} p_{\theta_{old}}(s,a)p_{\theta_{old}}(\mathcal{I}=1|s,a)$$

The loss function for the updated parameter θ is as follows.

$$\mathcal{L}(\theta) = -\sum_{s,a} \psi(s,a) \log \frac{p_\theta(\mathcal{I}=1,s,a)}{\psi(s,a)} - \log p(\theta)$$

It should be noted that ψ is expressed as a joint probability distribution, but conditional probability must be used to optimize the policy. Therefore, the first term of the equation is modified as follows.

$$\mathcal{L}_{\pi}(\theta) = -\sum_{s,a}\psi(s,a)\log\pi_{\theta}(a|s)$$

It is a weighted maximum likelihood policy loss function. It is assumed that an unbiased sample is used for this sample-based loss calculation.

As in the original MPO, The new prior keeps it close to the old one as follows.

$$\log p(\theta) \approx -\alpha\mathbb{E}_{s\sim p(s)}[D_{KL}(\pi_{\theta_{old}}(a|s))||(\pi_{\theta}(a|s))]$$

Then, this setting makes V-MPO optimize the constraint, because it is convenient to constrain on the KL term instead of tuning the α directly.

$$\theta^* = \underset{\theta}{argmin} -\sum_{s,a}\psi(s,a)\log\pi_{\theta}(a|s) \quad s.t. \underset{s\sim p(s)}{\mathbb{E}}[D_{KL}(\pi_{\theta_{old}}(a|s)||\pi_{\theta}(a|s))] < \epsilon_{\alpha}$$

In addition, the non-constraint target function is constructed as unconstrained objective using Lagrangian relaxation as follows in order to solve the problem of constrained optimization with the gradient descent method.

$$\mathcal{J}(\theta,\alpha) = \mathcal{L}_{\pi}(\theta) + \alpha(\epsilon_{\alpha} - \underset{s\sim p(s)}{\mathbb{E}}[D_{KL}(\pi_{\theta_{old}}(a|s)||\pi_{\theta}(a|s))])$$

As a result of composing the above equation, the constraint loss is as follows.

$$\mathcal{L}_{\alpha}(\theta,\alpha) = \alpha(\epsilon_{\alpha} - \underset{s\sim p(s)}{\mathbb{E}}[sg[[D_{KL}(\pi_{\theta_{old}}||\pi_{\theta})]]]) + sg[[\alpha]]\underset{s\sim p(s)}{\mathbb{E}}[D_{KL}(\pi_{\theta_{old}}||\pi_{\theta})]$$

If the second term of the above equation is replaced with a sample, the constraint loss is as follows.

$$\mathcal{L}_\alpha(\theta,\alpha)=\frac{1}{|\mathcal{D}|}\sum_{s\in\mathcal{D}}[\alpha(\epsilon_\alpha-sg[[D_{KL}(\pi_{\theta_{target}}(a|s)||\pi_\theta(a|s))]])+sg[[\alpha]]D_{KL}(\pi_{\theta_{target}}(a|s)||\pi_\theta(a|s))]$$

Implementation on JORLDY

- V-MPO JORLDY Implementation

```
### learn function ###
# 1. Calculate advantage and log_pi_old
# 2. Calculate loss of critic, eta, actor, alpha
# 3. Calculate total loss

def learn(self):
...

    # set advantage and log_pi_old
    with torch.no_grad():
        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_pi = m.log_prob(z)
            log_prob = log_pi.sum(axis=-1, keepdims=True)
            mu_old = mu
            std_old = std
        else:
            pi, value = self.network(state)
            pi_old = pi
            log_prob = torch.log(pi.gather(1, action.long()))
            log_pi_old = torch.log(pi)

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

    # start train iteration
    actor_losses, critic_losses, eta_losses, alpha_losses = [], [], [], []
    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, _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, ret, next_state, adv, log_prob_old],
            )

            if self.action_type == "continuous":
                _mu_old, _std_old = map(lambda x: x[idx], [mu_old, std_old])
            else:
                _log_pi_old, _pi_old = map(lambda x: x[idx], [log_pi_old, pi_old])

            # select top 50% of advantages
            idx_tophalf = _adv > _adv.median()
            tophalf_adv = _adv[idx_tophalf]
            # calculate psi
            exp_adv_eta = torch.exp(tophalf_adv / self.eta)
            psi = exp_adv_eta / torch.sum(exp_adv_eta.detach())

            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_pi = m.log_prob(z)
                log_prob = log_pi.sum(axis=-1, keepdims=True)
            else:
                pi, value = self.network(_state)
                log_prob = torch.log(pi.gather(1, _action.long()))
                log_pi = torch.log(pi)

            critic_loss = F.mse_loss(value, _ret).mean()

            # calculate loss for eta
            eta_loss = self.eta * self.eps_eta + self.eta * torch.log(
                torch.mean(exp_adv_eta)
            )

            # calculate policy loss (actor_loss)
            tophalf_log_prob = log_prob[idx_tophalf.squeeze(), :]
            actor_loss = -torch.sum(psi.detach().unsqueeze(1) * tophalf_log_prob)

            # calculate loss for alpha
            # NOTE: assumes that std are in the same shape as mu (hence vectors)
            #       hence each dimension of Gaussian distribution is independent
            if self.action_type == "continuous":
                ss = 1.0 / (std**2)  # (batch_size * action_dim)
                ss_old = 1.0 / (_std_old**2)  # (batch_size * action_dim)

                # mu
                d_mu = mu - _mu_old.detach()  # (batch_size * action_dim)
                KLD_mu = 0.5 * torch.sum(
                    d_mu * 1.0 / ss_old.detach() * d_mu, axis=1
                )
                mu_loss = torch.mean(
                    self.alpha_mu * (self.eps_alpha_mu - KLD_mu.detach())
                    + self.alpha_mu.detach() * KLD_mu
                )

                # sigma
                KLD_sigma = 0.5 * (
                    (
                        torch.sum(1.0 / ss * ss_old.detach(), axis=1)
                        - ss.shape[-1]
                        + torch.log(
                            torch.prod(ss, axis=1)
                            / torch.prod(ss_old.detach(), axis=1)
                        )
                    )
                )
```

```
                sigma_loss = torch.mean(
                    self.alpha_sigma * (self.eps_alpha_sigma - KLD_sigma.detach())
                    + self.alpha_sigma.detach() * KLD_sigma
                )

                alpha_loss = mu_loss + sigma_loss
            else:
                KLD_pi = _pi_old.detach() * (_log_pi_old.detach() - log_pi)
                KLD_pi = torch.sum(KLD_pi, axis=len(_pi_old.shape) - 1)
                alpha_loss = torch.mean(
                    self.alpha_mu * (self.eps_alpha_mu - KLD_pi.detach())
                    + self.alpha_mu.detach() * KLD_pi
                )

            loss = critic_loss + actor_loss + eta_loss + alpha_loss

...
```