MPO(Maximum a Posteriori Policy Optimization)

Paper Link: https://arxiv.org/abs/1806.06920

Key Features

- MPO is an algorithm that learns policy with the maximum posterior probability as optimal policy. This method can be thought as a combination of Bayesian inference and RL.
- MPO finds optimal policy using two processes: 'control' and 'estimation'. The EM algorithm is used in this process.
- MPO is an Off-Policy algorithm suitable for high-dimensional control.
- MPO shows good sample efficiency for continuous action space and robustness in relation to hyperparameters.

Background

Difficulty : High-dimensional action space control and Sample efficiency

Before MPO was presented, TRPO or PPO algorithm was robust to high-dimensional control environments. However, TRPO and PPO have the disadvantage of poor sample efficiency, because they are On-Policy RL algorithms. In contrast, Off-Policy algorithm which has good data efficiency(e.g. DDPG, SVG, NAF) cannot be considered because they are generally unsuitable for high-dimensional control.

Method

MPO : Maximum a Posteriori Optimization

MPO uses auxiliary distribution q which is sample's likelihood distribution to calculate value function Q and optimize policy π during EM-algorithm. In E-step, objective function J based on q can be obtained in condition π_i is fixed(reverse KL divergence). This J is sub-optimal Q because it is based on present π_i which is not global optimal. Thereafter in M-step, the Q which extracted from E-step is fixed as a target and the J based on q is updated to find next optimal π_{i+1} through policy improvement(forward KL divergence). Through EM algorithm, MPO can ensure monotonous improvement in performance.

Evidence Lower Bound

During EM steps, Evidence Lower Bound(ELBO) is used as objective function. ELBO represents the lower bound of log-likelihood for the sample. The ELBO can be used even when population distribution is the multi-modal distribution that is difficult to handle. The equation below shows intuitively log-likelihood of $(O = 1)$. It can be interpreted as the event of obtaining maximum reward by choosing an action.

$$
\begin{aligned}
&log p_\pi(O=1) \\
&= log \int p_\pi(\tau) p(O=1|\tau) d\tau \geq \int q(\tau)[log[(O=1|\tau) + log \frac{p_\pi}{q(\tau)}]d\tau \\
&= \mathbb{E}_q[\sum r_t/\alpha] - KL(q(\tau)||p_\pi(\tau)) = \mathcal{J}(q,\pi)
\end{aligned}
$$

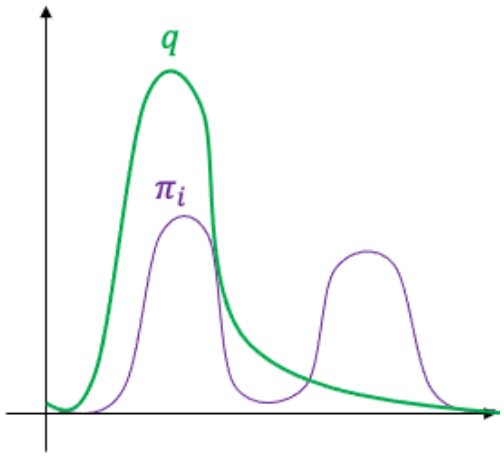
At the above equation, $J(q,\pi)$ is maximum when there is no difference between distribution q and π. If you find q that KL-term is the smallest with respect to π, it is the optimal objective function $J(q,\pi)$.

Expectation and Maximization algorithm

MPO applies EM algorithm to optimize policy. Expectation and Maximization(EM) is an algorithm used to estimate the population distribution of a sample without a label. In the Expectation(E-step) within the EM algorithm, labeling is performed by estimating the population distribution of the samples based on likelihood. In the Maximization(M-step), reconstruction of the estimated population distribution is conducted with form of specific distribution which is generally Gaussian form. As a result of M-step, new labels are set on samples and then the process is conducted to the next Expectation step. By repeating this process, MPO can find a distribution with maximum likelihood for samples and estimates it as a distribution which is approximated population distribution.

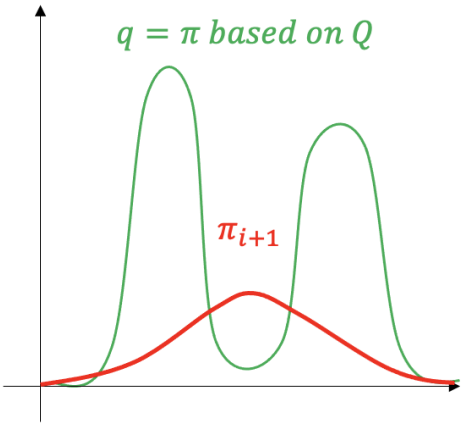
KL-divergence-based EM algorithm

On the E-step, MPO computes reverse KL-divergence term from q to π_i. The figure for this is as follows.



By reverse KL-divergence computation($KL(q||\pi_i = Q)$), MPO can find an optimal-Q that is tractable for policy with a multi-modal distribution. In other words, feature can be extracted through this process. This method is called as modal-seeking.

On the M-step, MPO computes forward KL-divergence term from π_{i+1} to $q = \pi$. The figure of this process is shown as follows.



By forward KL-divergence computation($KL(\pi_i||q=\pi)$), MPO can find a next optimal-π. The policy like this reduces tendency of policy to avoid collapse. This method is called as mean-seeking.

The reverse KL-divergence and forward KL-divergence terms are used to calculate the objective function J in E-step and M-step, respectively to calculate Q, π.

Policy Evaluation with Retrace algorithm

To calculate value during above policy improvement, the MPO uses replay buffer. MPO uses Retrace algorithm in addition to importance sampling essential on using Off-Policy algorithm as follows.

$$\min_{\phi} L(\phi) = \min_{\phi} E_{\mu_b(s),b(a|s)}[(Q_{\theta_i}(s_t,a_t,\phi) - Q_t^{ret})^2]$$

$$Q_t^{ret} = Q_{\phi'}(s_t,a_t) + \sum_{j=t}^{\infty}\gamma^{j-t}(\prod_{k=t+1}^{j} c_k)[r(s_j,a_j) + E_{\pi(a|s_{j+1}}[Q_{\phi'}(s_{j+1},a)] - Q_{\phi'}(s_j,a_j)]$$

$$c_k = \min(1, \frac{\pi(a_k|s_k)}{b(a_k|s_k)})$$

C_k is importance sampling term. Without Retrace algorithm, the loss function can be diverge by Q_t during computation. To solve this divergence problem, retrace algorithm, which constraints upper bound of C_k value to 1, is used.

Policy Improvement

In variational distribution $q(\tau)$, MPO considers ELBO as follows.

$$\mathcal{J}(q,\theta) = \mathbb{E}_q[\sum_{t=0}^{\infty}\gamma^t[r_t - \alpha KL(q(a|s_t)||\pi(a|s_t,\theta))]] + logp(\theta)$$

In addition, the regularized Q-function associated with above ELBO can be defined as follows.

$$Q_\theta^q(s,a) = r_0 + \mathbb{E}_{q(\tau), s_0=s, a_0=a}[\sum_{t\geq 1}^{\infty}\gamma^t[r_t - \alpha KL(q_t||\pi_t)]]$$

In above equation, $KL(q_t||\pi_t)$ can be replaced by $KL(q(a|s_t))||\pi(a|s_t,\theta))$. MPO optimizes J with respect to q is equivalent to solving an expected reward RL problem with augmented reward $\tilde{r} = r_t - \alpha log\frac{q(a_t|s_t)}{\pi(a_t|s_t,\theta)}$.

E-step : optimize J with respect to Q

The ELBO J with respect to Q is shown as follows. It is '1-step' KL regularized objective function.

$$\begin{aligned}&\max_q J(q,\theta_i) = \max_q T^{\pi,q}Q_{\theta_i}(s,a)\\&= \max_q \mathbb{E}_{\mu(s)}[\mathbb{E}_{q(\cdot|s)}[Q_{\theta_i}(s,a)] - \alpha D_{KL}(q||\pi_i)]\end{aligned}$$

Even if this maximizing equation obtains $q = argmaxJ$, it is not optimal. Because it treats Q as a constant with respect to present $q = \pi_i$. In addition, This maximizing equation can be thought of separating the term of target term $V_{\theta_i}(s) = E_{q(\cdot|s)}[Q_{\theta_i}(s,a)]$ used in conventional RL and the term of surrogate objective $\alpha D_{KL}(Q||\pi)$. The equation means that the optimal-q has to maximize cumulative reward Q and at the same time the optimal-q should be most similar to the distribution of policy π_i.

The reward and the KL terms are on an arbitrary relative scale. This can make it difficult to choose α. MPO replace the soft KL regularization with a hard constraint with parameter. If constraint is added on this ELBO J that q is a parametric variational distribution with parameter θ and q is on Off-Policy setting, it can be similar to the TRPO equation as follows. In contrast, the unconstrained equation shown above is similar to the objective used by PPO.

$$\max_q \mathbb{E}_{\mu(s)}[\mathbb{E}_{q(a|s)}[Q_{\theta_i}(s,a)]], \quad s.t. \mathbb{E}_{\mu(s)}[KL(q(a|s), \pi(a|s,\theta_i))] < \epsilon$$

M-step : optimize J with respect to π

The MPO finds ELBO J with respect to π to obtain an updated policy $\pi_{i+1} = argmax_\theta(q_i, \theta)$ during the M-step. In M-step, the π_{t+1}'s parameter θ_{t+1} is obtained based on the previously computed $q_i = Q$. In the paper, $p(\theta)$ is set as Gaussian prior around the current policy.

$$
\begin{aligned}
&objective\ function \\
&\max_\theta J(q_i, \theta) = \max_\theta \mathbb{E}_{\mu_q(s)}[\mathbb{E}_q(a|s)[log\pi(a|s,\theta)]] + logp(\theta) \\
&p(\theta) \approx \mathcal{N}(\mu = \theta_i, \sum = \frac{F_{\theta_i}}{\lambda})
\end{aligned}
$$

The optimal objective function is as follows. It can be divided into 2 kinds of versions depending on the constraint. The constraint is effective to minimize the risk of overfitting.

$$
\begin{aligned}
&generalized\ M\ step\ equation \\
&= \max_\pi \mathbb{E}_{\mu_q(s)}[E_{q(a|s)}[log\pi(a|s,\theta)] - \lambda KL(\pi(a|s,\theta_i), \pi(a|s,\theta))]
\end{aligned}
$$

$$
\begin{aligned}
&Hard\ constraint\ ver. \\
&= \max_\pi \mathbb{E}_{\mu_q(s)}[E_{q(a|s)}[log\pi(a|s,\theta)]] \quad s.t.\ \mathbb{E}_{\mu_q(s)}[KL(\pi(a|s,\theta_i), \pi(a|s,\theta))] < \epsilon
\end{aligned}
$$

Surrogate objective constraint by Decoupling

For better optimization, MPO divides the term of surrogate objective as mean term and variance term. This technique called ‘Decoupling’ allows MPO to converge more elaborately by specifying ϵ respectively for mean and variance. By Decoupling, the generalized equation called Lagrangian equation can be obtain as follows. η_μ and $\eta_\sum$ are Lagrangian multipliers.

$$
L(\theta^q, \eta_\mu, \eta_\sum) = \int \mu_q(s) \int q(a|s;\theta^q) A_i(a,s) da ds + \eta_\mu(\epsilon_\mu - C_\mu) + \eta_\sum(\epsilon_\sum - C_\sum)
$$

Algorithm

Algorithm 1 MPO (chief)

Input G number of gradients to average
while True **do**
    initialize N = 0
    initialize gradient store $s_\phi = \{\}$, $s_\eta = \{\}$, $s_{\eta_\mu} = \{\}$, $s_{\eta\Sigma} = \{\}$ $s_\theta = \{\}$
    while $N < G$ **do**
        receive next gradient from worker w
        $s_\phi = s_\phi + [\delta\phi^w]$
        $s_\phi = s_\theta + [\delta\theta^w]$
        $s_\eta = s_\eta + [\delta\eta^w]$
        $s_{\eta_\mu} = s_{\eta_\mu} + [\delta\eta_\mu^w]$
        $s_{\eta_\theta} = s_{\eta_\theta} + [\delta\eta_\theta^w]$
    update parameters with average gradient from
    s_ϕ, s_η, s_{η_μ}, $s_{\eta\Sigma}$ s_θ
    send new parameters to workers

Algorithm 3 MPO (worker) - parametric variational distribution

Input $= \epsilon_\Sigma, \epsilon_\mu, L_{\text{max}}$
$i = 0$, $L_{\text{curr}} = 0$
Initialise $Q_{\omega_i}(a, s)$, $\pi(a|s, \boldsymbol{\theta}_i)$, $\eta, \eta_\mu, \eta_\Sigma$
for each worker **do**
    while $L_{\text{curr}} < L_{\text{max}}$ **do**
        update replay buffer $\mathcal{B}$ with L trajectories from the environment
        $k = 0$
        // Find better policy by gradient descent
        while $k < 1000$ **do**
            sample a mini-batch $\mathcal{B}$ of N (s, a, r) pairs from replay
            sample M additional actions for each state from $\mathcal{B}$, $\pi(a|s, \boldsymbol{\theta}_k)$ for estimating integrals
            compute gradients, estimating integrals using samples
            // Q-function gradient:
            $\delta_\phi = \partial_\phi L'_\phi(\phi)$
            // E-Step gradient:
            $[\delta_{\eta_\mu}, \delta_{\eta\Sigma}] = \alpha\partial_{\eta_\mu, \eta\Sigma} L(\boldsymbol{\theta}_k, \eta_\mu, \eta_\Sigma)$
            $\delta_\theta = \partial_{\boldsymbol{\theta}} L(\boldsymbol{\theta}, \eta_{\mu\,k+1}, \eta_{\Sigma\,k+1})$
            // M-Step gradient: In practice there is no M-step in this case as policy and variatinal distribution q use a same structure.
            send gradients to chief worker
            wait for gradient update by chief
            fetch new parameters $\phi, \theta, \eta, \eta_\mu, \eta_\Sigma$
            $k = k + 1$
        $i = i + 1$, $L_{\text{curr}} = L_{\text{curr}} + L$
        $\boldsymbol{\theta}_i = \boldsymbol{\theta}, \phi' = \phi$

Implementation on JORLDY

- MPO JORLDY Implementation

```
### learn function ###
# 1. Caluculates critic loss
# 2. Caluculates actor loss
# 3. Caluculates eta(Lagrangian coefficient) loss
# 4. Caluculates alpha loss

def learn(self):
...

    if self.action_type == "continuous":
        mu, std = self.actor(state)
        Q = self.critic(state, action)
        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)
        prob = torch.exp(log_prob)

        with torch.no_grad():
            mut, stdt = self.target_actor(state)
            mt = Normal(mut, stdt)
            zt = torch.atanh(torch.clamp(action, -1 + 1e-7, 1 - 1e-7))
            log_pit = mt.log_prob(zt)
            log_probt = log_pit.sum(axis=-1, keepdims=True)

            mu_old = mut
            std_old = stdt
            prob_t = torch.exp(log_probt)

            Qt_a = self.target_critic(state, action)

            next_mu, next_std = self.actor(next_state)
            mn = Normal(next_mu, next_std)
            zn = mn.sample(
                (self.num_sample,)
            )  # (num_sample, batch_size * len_tr, dim_action)
            next_action = torch.tanh(zn)

            Qt_next = self.target_critic(
                next_state.unsqueeze(0).repeat_interleave(self.num_sample, dim=0),
                next_action,
            )  # (num_sample, batch_size * len_tr, 1)

            c = torch.clip(prob / (prob_b + 1e-6), max=1.0)

            if self.critic_loss_type == "1step_TD":
                Qret = reward + self.gamma * (1 - done) * Qt_next.mean(axis=0)
            elif self.critic_loss_type == "retrace":
                Qret = reward + self.gamma * Qt_next.mean(axis=0) * (1 - done)

                # temporarily reshaping values
                # (batch_size * len_tr, item_dim) -> (batch_size, len_tr, item_dim)
                Qret = Qret.view(self.batch_size, -1, *Qret.shape[1:])
                Qt_a = Qt_a.view(self.batch_size, -1, *Qt_a.shape[1:])
                c = c.view(self.batch_size, -1, *c.shape[1:])
```

```
            done = done.view(self.batch_size, -1, *done.shape[1:])
            for i in reversed(range(Qret.shape[1] - 1)):
                Qret[:, i] += (
                    self.gamma
                    * c[:, i + 1]
                    * (1 - done[:, i])
                    * (Qret[:, i + 1] - Qt_a[:, i + 1])
                )
            Qret = Qret.view(-1, *Qret.shape[2:])

        zt_add = mt.sample(
            (self.num_sample,)
        )  # (num_sample, batch_size * len_tr, dim_action)
        action_add = torch.tanh(zt_add)
        log_pi_add = m.log_prob(zt_add)
        log_prob_add = log_pi_add.sum(axis=-1, keepdims=True)
        Qt_add = self.target_critic(
            state.unsqueeze(0).repeat_interleave(self.num_sample, dim=0), action_add
        )

        critic_loss = F.mse_loss(Q, Qret).mean()

        # Calculate Vt_add, At_add using Qt_add
        Vt_add = torch.mean(Qt_add, axis=0, keepdims=True)
        At_add = Qt_add - Vt_add
        At = At_add

        """ variational distribution q uses exp(At / eta) instead of exp(Qt / eta), for stable learning"""
        q = torch.exp(F.log_softmax(At_add / self.eta, dim=0))
        actor_loss = -torch.mean(torch.sum(q.detach() * log_prob_add, axis=0))

        eta_loss = self.eta * self.eps_eta + self.eta * torch.mean(
            torch.log(torch.exp((At_add) / self.eta).mean(axis=0))
        )

        ss = 1.0 / (std**2)  # (batch_size * len_tr, action_dim)
        ss_old = 1.0 / (std_old**2)

        """
        KL-Divergence losses(related to alpha) implemented using methods introduced from V-MPO paper
        https://arxiv.org/abs/1909.12238
        """

        # mu
        d_mu = mu - mu_old.detach()  # (batch_size * len_tr, 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:
        pi = self.actor(state)  # pi,Q: (batch_size, len_tr, dim_action)
        pi_next = self.actor(next_state)
        Q = self.critic(state)
        Q_a = Q.gather(1, action.long())

        with torch.no_grad():
            # calculate Q_ret using Retrace
            Qt = self.target_critic(state)  # Q_target
            Qt_next = self.target_critic(next_state)
            pit = self.target_actor(state)

            Qt_a = Qt.gather(1, action.long())
            prob_t = pi.gather(
                1, action.long()
            )  # (batch_size * len_tr, 1), target policy probability

            c = torch.clip(
                prob_t / (prob_b + 1e-6), max=1.0
            )  # (batch_size * len_tr, 1), prod of importance ratio and gamma

            if self.critic_loss_type == "1step_TD":
                Qret = reward + self.gamma * (1 - done) * torch.sum(
                    pi_next * Qt_next, axis=-1, keepdim=True
                )
            elif self.critic_loss_type == "retrace":
                Qret = reward + self.gamma * torch.sum(
                    pi_next * Qt_next, axis=-1, keepdim=True
                ) * (1 - done)

                # temporarily reshaping values
                # (batch_size * len_tr, item_dim) -> (batch_size, len_tr, item_dim)
                Qret = Qret.view(self.batch_size, -1, *Qret.shape[1:])
                Qt_a = Qt_a.view(self.batch_size, -1, *Qt_a.shape[1:])
                c = c.view(self.batch_size, -1, *c.shape[1:])
                done = done.view(self.batch_size, -1, *done.shape[1:])
                for i in reversed(
                    range(Qret.shape[1] - 1)
                ):  # along the trajectory length
                    Qret[:, i] += (
                        self.gamma
                        * c[:, i + 1]
                        * (Qret[:, i + 1] - Qt_a[:, i + 1])
                        * (1 - done[:, i])
                    )
                Qret = Qret.view(-1, *Qret.shape[2:])

            pi_old = pit

        critic_loss = F.mse_loss(Q_a, Qret).mean()

        # calculate V, Advantage of Qt
        Vt = torch.sum(pi_old * Qt, axis=-1, keepdims=True)
        At = Qt - Vt

        """ variational distribution q uses exp(At / eta) instead of exp(Qt / eta), for stable learning"""
```

```
        q = torch.exp(F.log_softmax(At / self.eta, dim=-1))
        actor_loss = -torch.mean(torch.sum(q.detach() * torch.log(pi), axis=-1))

        eta_loss = self.eta * self.eps_eta + self.eta * torch.mean(
            torch.log(torch.sum(pi_old * torch.exp(At / self.eta), axis=-1))
        )

        """
        KL-Divergence losses(related to alpha) implemented using methods introduced from V-MPO paper
        https://arxiv.org/abs/1909.12238
        """

        KLD_pi = pi_old.detach() * (torch.log(pi_old.detach()) - torch.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

...
```

```
### reset Lagrange multipliers: eta, alpha_{mu, sigma} ###

def reset_lgr_muls(self):
    self.eta.data = torch.max(self.eta, self.min_eta)
    self.alpha_mu.data = torch.max(self.alpha_mu, self.min_alpha_mu)
    self.alpha_sigma.data = torch.max(self.alpha_sigma, self.min_alpha_sigma)
```