Trust Region Policy Optimization

TRPO
置信域策略优化
第二部分

东川路第一可爱猫猫虫

主要内容

- $\nabla_\theta L_{\theta_{old}}(\theta_{old})$的实现
- 用KKT条件求解不等式约束的优化问题
- Hessian Vector Product
- 共轭梯度法求Hx=g
- 回溯线搜索

感谢
蕾西亚万岁啊我透
的续费包月充电

Policy Net

$$\max \; g^T(\theta - \theta_{old})$$

$$s.t. \; \frac{1}{2}(\theta - \theta_{old})^T H(\theta - \theta_{old}) \leq \delta$$

- g是$\nabla_\theta L_{\theta_{old}}(\theta_{old})$

$$L_{\theta_{\text{old}}}(\theta) = \mathbb{E}_{s\sim\rho_{\theta_{\text{old}}}, a\sim\pi_{\theta_{\text{old}}}} \left[\frac{\pi_\theta(a|s)}{\pi_{\theta_{\text{old}}}(a|s)} \cdot A_{\pi_{\theta_{\text{old}}}}(s,a) \right]$$

log_probs = policy_net.get_log_prob(states, actions)

$$\exp(\log \pi_\theta - \log \pi_{\theta_{\text{old}}}) = \frac{\pi_\theta(a|s)}{\pi_{\theta_{\text{old}}}(a|s)}$$

ratio = torch.exp(log_probs - old_log_probs)

$$g^T$$

$$L_{\theta_{\text{old}}}(\theta) = \mathbb{E}_{s\sim\rho_{\theta_{\text{old}}},\, a\sim\pi_{\theta_{\text{old}}}}\left[\frac{\pi_\theta(a|s)}{\pi_{\theta_{\text{old}}}(a|s)} \cdot A_{\pi_{\theta_{\text{old}}}}(s,a)\right]$$

loss = (ratio * advantages).mean()

- g是$\nabla_\theta L_{\theta_{old}}(\theta_{old})$，下一步对$L_{\theta_{old}}(\theta)$求导

loss_grads = autograd.grad(loss, policy_net.parameters())

- 把梯度张量展平成一维张量，拼接得到g^T

loss_grad = torch.cat([grad.view(-1) for grad in loss_grads]).detach()

求解不等式约束的优化问题

$$\max \ g^T(\theta - \theta_{old})$$

$$s.t. \ \frac{1}{2}(\theta - \theta_{old})^T H(\theta - \theta_{old}) \leq \delta$$

$$\mathcal{L}(\theta, \lambda) = g^T(\theta - \theta_{old}) - \lambda\left(\frac{1}{2}(\theta - \theta_{old})^T H(\theta - \theta_{old}) - \delta\right)$$

- 对θ求偏导并置为0

$$\nabla_\theta \mathcal{L}(\theta, \lambda) = g - \lambda H(\theta - \theta_{old}) = 0$$

$$\theta - \theta_{old} = H^{-1} \cdot \frac{g}{\lambda}$$

- 将$\theta - \theta_{old}$的表达式带入约束条件

$$\frac{1}{2}\left(H^{-1} \cdot \frac{g}{\lambda}\right)^T H \left(H^{-1} \cdot \frac{g}{\lambda}\right) \leq \delta$$

先求$H^{-1}g$

$$\frac{1}{2}\cdot\frac{g^TH^{-1}g}{\lambda^2}\leq\delta \qquad \lambda^2\geq\frac{g^TH^{-1}g}{2\delta} \qquad \lambda\geq\sqrt{\frac{g^TH^{-1}g}{2\delta}}$$

$$\theta-\theta_{old}=H^{-1}\cdot\frac{g}{\sqrt{\frac{g^TH^{-1}g}{2\delta}}}=H^{-1}g\cdot\sqrt{\frac{2\delta}{g^TH^{-1}g}}$$

- 思路是先求$H^{-1}g$
- 即求解Hx=g
- H是平均KL散度的Hessian矩阵
- 共轭梯度法

Hessian Vector Product

- 无需显式构造H

  就能求出Hessian矩阵和向量的乘积

- Hvp

  Hessian矩阵与向量v的乘积

  =一阶导数与向量内积对参数的梯度

- $\theta_{n\times 1}$为参数　　标量函数$f(\theta)$

  一阶导 $g(\theta) = \left[\frac{\partial f}{\partial \theta_1}, \frac{\partial f}{\partial \theta_2}, \dots \frac{\partial f}{\partial \theta_n}\right]^T$

  二阶导 $H(\theta)$　$H_{ij} = \frac{\partial^2 f}{\partial \theta_i \partial \theta_j}$

下面证明：Hessian矩阵与向量v的乘积=一阶导数与向量内积对参数的梯度

Hessian矩阵与向量v的乘积
=一阶导数与向量内积对参数的梯度

- $H_{ij} = \frac{\partial^2 f}{\partial \theta_i \partial \theta_j}$ $v_{n\times 1}$
- $[H(\theta)v]_k$

$$= \Sigma_{i=1}^{n} H_{ki} v_i$$

$$= \Sigma_{i=1}^{n} \frac{\partial^2 f}{\partial \theta_k \partial \theta_i} v_i$$

- 一阶导数与向量内积

$$v^T g = \Sigma_{i=1}^{n} v_i g_i = \Sigma_{i=1}^{n} v_i \frac{\partial f}{\partial \theta_i}$$

求梯度

$$[\nabla_\theta (v^T g)]_k = \frac{\partial (v^T g)}{\partial (\theta_k)} = \Sigma_{i=1}^{n} v_i \frac{\partial^2 f}{\partial \theta_k \partial \theta_i}$$

Hvp应用于TRPO 求Hv

[DeepRL_Algorithms/Algorithms/pytorch/TRPO at master · RITCHIEHuang/DeepRL_Algorithms · GitHub]

- $f(\theta)$是KL散度的均值

```
kl = policy_net.get_kl(states)
kl = kl.mean()
```

- 求$f(\theta)$的一阶导数

```
grads = torch.autograd.grad(kl, policy_net.parameters(), create_graph=True)
flat_grad_kl = torch.cat([grad.view(-1) for grad in grads])
```

- 一阶导数与向量相乘得到内积

```
kl_v = (flat_grad_kl * v).sum()
```

- 内积对参数求梯度

```
grads = torch.autograd.grad(kl_v, policy_net.parameters())
flat_grad_grad_kl = torch.cat([grad.contiguous().view(-1)
for grad in grads]).detach()
```

Hvp

```
def Hvp(v):
    kl = policy_net.get_kl(states)
    kl = kl.mean()
    grads = torch.autograd.grad(kl, policy_net.parameters(),
create_graph=True)
    flat_grad_kl = torch.cat([grad.view(-1) for grad in grads])
    kl_v = (flat_grad_kl * v).sum()
    grads = torch.autograd.grad(kl_v, policy_net.parameters())
    flat_grad_grad_kl = torch.cat([grad.contiguous().view(-1) for grad in
grads]).detach()
    return flat_grad_grad_kl + v * damping
```

共轭梯度 Ax=b

- 初始化

```
x = torch.zeros_like(b, device=device)
r = - b.clone()
p = b.clone()
rdotr = r.t() @ r
```

迭代

- 步长 $\alpha_k = \frac{r_k^T r_k}{p_k^T H p_k}$
- 解的更新

  第k+1步迭代的解$x_{k+1} = x_k + \alpha_k p_k$
- 残差的更新

  $r_1 = g - Hx_1 = g - H(x_0 + \alpha_0 p_0) = g - Hx_0 - \alpha_0 H p_0$
  $= r_0 - \alpha_0 H p_0$
- 更新系数 $\beta_k = \frac{r_{k+1}^T r_{k+1}}{r_k^T r_k}$
- 搜索方向的更新 $p_{k+1} = r_{k+1} + \beta_k p_k$

迭代代码

```
for i in range(steps):
    Hvp = Hvp_f(p)  # A @ p
    alpha = rdotr / (p.t() @ Hvp)  # step length
    x += alpha * p  # update x
    r += alpha * Hvp  # new residual
    new_rdotr = r.t() @ r
    betta = new_rdotr / rdotr  # beta
    p = - r + betta * p
    rdotr = new_rdotr
    if rdotr < rdotr_tol:  # satisfy the threshold
        break
```

完整的CG共轭梯度函数

```
def conjugate_gradient(Hvp_f, b, steps=10, rdotr_tol=1e-10):
    x = torch.zeros_like(b, device=device)  # initialization approximation of x
    r = - b.clone()  # Hvp(x) - b : residual
    p = b.clone()  # b - Hvp(x) : steepest descent direction
    rdotr = r.t() @ r  # r.T @ r
    for i in range(steps):
        Hvp = Hvp_f(p)  # A @ p
        alpha = rdotr / (p.t() @ Hvp)  # step length
        x += alpha * p  # update x
        r += alpha * Hvp  # new residual
        new_rdotr = r.t() @ r
        betta = new_rdotr / rdotr  # beta
        p = - r + betta * p
        rdotr = new_rdotr
        if rdotr < rdotr_tol:  # satisfy the threshold
            break
    return x
```

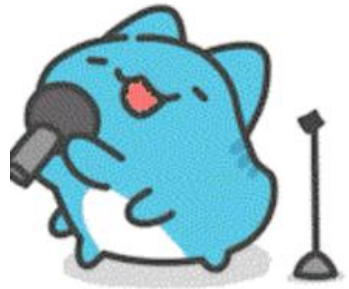
求参数的完整更新方向

$$\theta - \theta_{old} = H^{-1} \cdot \frac{g}{\sqrt{\frac{g^T H^{-1} g}{2\delta}}} = H^{-1} g \cdot \sqrt{\frac{2\delta}{g^T H^{-1} g}}$$

```
step_dir = conjugate_gradient(Hvp, loss_grad)
shs = Hvp(step_dir).t() @ step_dir
lm = torch.sqrt(2 * max_kl / shs)
```

- 参数的完整更新方向

```
step = lm * step_dir
```



回溯线搜索

```
def line_search(model, f, x, step_dir, expected_improve, max_backtracks=10,
accept_ratio=0.1):
    f_val = f(False).item()
    for step_coefficient in [.5 ** k for k in range(max_backtracks)]:
        x_new = x + step_coefficient * step_dir
        set_flat_params(model, x_new)
        f_val_new = f(False).item()
        actual_improve = f_val_new - f_val
        improve = expected_improve * step_coefficient
        ratio = actual_improve / improve
        if ratio > accept_ratio:
            return True, x_new
    return False, x
```