

# Trust Region Policy Optimization (TRPO)

이동민

삼성전자 서울대 공동연구소  
Jul 18, 2019

# Outline

---

- Trust Region Policy Optimization (TRPO)
  - Learning process
  - Hyperparameter
  - Main loop
  - Train model
  - Train & TensorboardX
  - Learning curve & Test

# Trust Region Policy Optimization (TRPO)

- TRPO Algorithm

---

**Algorithm 1** Trust Region Policy Optimization

---

- 1: Input: initial policy parameters  $\theta_0$ , initial value function parameters  $\phi_0$
- 2: Hyperparameters: KL-divergence limit  $\delta$ , backtracking coefficient  $\alpha$ , maximum number of backtracking steps  $K$
- 3: **for**  $k = 0, 1, 2, \dots$  **do**
- 4:   Collect set of trajectories  $\mathcal{D}_k = \{\tau_i\}$  by running policy  $\pi_k = \pi(\theta_k)$  in the environment.
- 5:   Compute rewards-to-go  $\hat{R}_t$ .
- 6:   Compute advantage estimates,  $\hat{A}_t$  (using any method of advantage estimation) based on the current value function  $V_{\phi_k}$ .
- 7:   Estimate policy gradient as

$$\hat{g}_k = \frac{1}{|\mathcal{D}_k|} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^T \nabla_\theta \log \pi_\theta(a_t | s_t)|_{\theta_k} \hat{A}_t,$$

- 8:   Use the conjugate gradient algorithm to compute

$$\hat{x}_k \approx \hat{H}_k^{-1} \hat{g}_k,$$

- where  $\hat{H}_k$  is the Hessian of the sample average KL-divergence.
- 9:   Update the policy by backtracking line search with

$$\theta_{k+1} = \theta_k + \alpha^j \sqrt{\frac{2\delta}{\hat{x}_k^T \hat{H}_k \hat{x}_k}} \hat{x}_k,$$

- where  $j \in \{0, 1, 2, \dots K\}$  is the smallest value which improves the sample loss and satisfies the sample KL-divergence constraint.
- 10:   Fit value function by regression on mean-squared error:

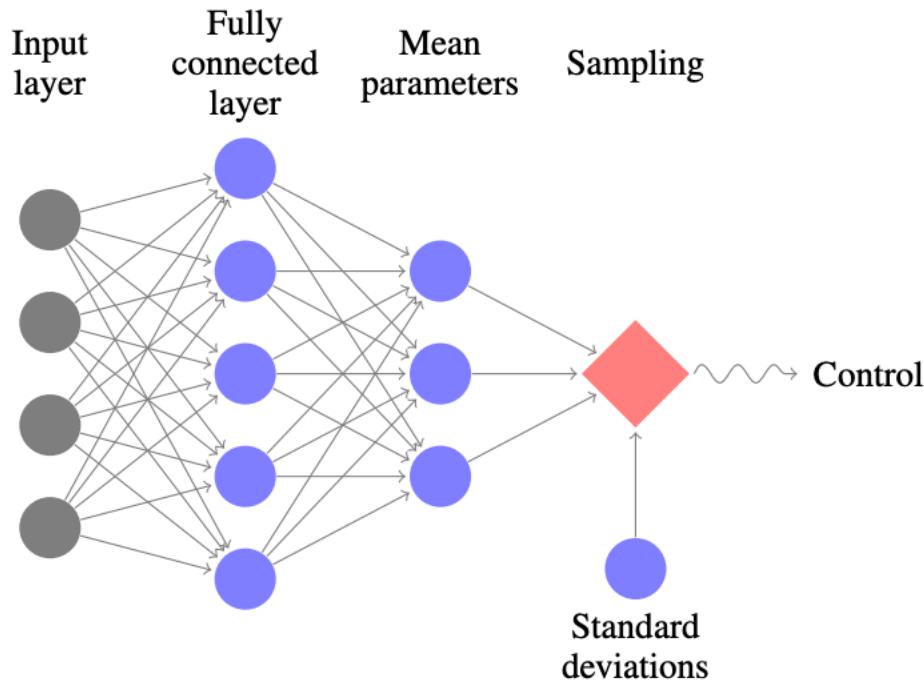
$$\phi_{k+1} = \arg \min_{\phi} \frac{1}{|\mathcal{D}_k| T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^T \left( V_\phi(s_t) - \hat{R}_t \right)^2,$$

- typically via some gradient descent algorithm.
- 11: **end for**
- 

source : <https://spinningup.openai.com/en/latest/algorithms/trpo.html>

# Trust Region Policy Optimization (TRPO)

- Actor network



# Trust Region Policy Optimization (TRPO)

- Actor network

```
class Actor(nn.Module):
    def __init__(self, state_size, action_size, args):
        super(Actor, self).__init__()
        self.fc1 = nn.Linear(state_size, args.hidden_size)
        self.fc2 = nn.Linear(args.hidden_size, args.hidden_size)
        self.fc3 = nn.Linear(args.hidden_size, action_size)

    def forward(self, x):
        x = torch.tanh(self.fc1(x))
        x = torch.tanh(self.fc2(x))

        mu = self.fc3(x)
        log_std = torch.zeros_like(mu)
        std = torch.exp(log_std)

    return mu, std
```

# Trust Region Policy Optimization (TRPO)

- Actor network

```
class Actor(nn.Module):
    def __init__(self, state_size, action_size, args):
        super(Actor, self).__init__()
        self.fc1 = nn.Linear(state_size, args.hidden_size)
        self.fc2 = nn.Linear(args.hidden_size, args.hidden_size)
        self.fc3 = nn.Linear(args.hidden_size, action_size)

    def forward(self, x):
        x = torch.tanh(self.fc1(x))
        x = torch.tanh(self.fc2(x))

        mu = self.fc3(x)
        log_std = torch.zeros_like(mu)
        std = torch.exp(log_std)

    return mu, std
```

The number of parameters

Layer	Weight	Bias
1	$3 \times 64 = 192$	64
2	$64 \times 64 = 4096$	64
3	$64 \times 1 = 64$	1
Sum	4352	129
Total		4481

# Trust Region Policy Optimization (TRPO)

- Critic network

```
class Critic(nn.Module):
    def __init__(self, state_size, args):
        super(Critic, self).__init__()
        self.fc1 = nn.Linear(state_size, args.hidden_size)
        self.fc2 = nn.Linear(args.hidden_size, args.hidden_size)
        self.fc3 = nn.Linear(args.hidden_size, 1)

    def forward(self, x):
        x = torch.tanh(self.fc1(x))
        x = torch.tanh(self.fc2(x))
        value = self.fc3(x)

    return value
```

# Trust Region Policy Optimization (TRPO)

- Learning process
  1. 상태에 따른 행동 선택
  2. 환경에서 선택한 행동으로 한 time step을 진행한 후, 다음 상태와 보상을 받음
  3. Sample  $(s, a, r, s')$ 을 trajectories set에 저장
  4. 일정 sample들이 모이면 trajectories set으로 Actor & Critic network 업데이트
    - Step 1 : Return 구하기
    - Step 2 : Critic network 업데이트
    - Step 3 : Actor loss의 gradient를 구하고, conjugate gradient method (CGM)을 통해 search direction 구하기
    - Step 4 : Step size와 maximal step 구하기
    - Step 5 : Actor network를 업데이트하고, backtracking line search를 통해 trust region을 잡고 trust region안에서 업데이트

# Learning process

## 1. 상태에 따른 행동 선택

```
156 | mu, std = actor(torch.Tensor(state))  
157 | action = get_action(mu, std)
```

```
def get_action(mu, std):  
    normal = Normal(mu, std)  
    action = normal.sample()  
  
    return action.data.numpy()
```

- **Normal(mu, std)**
  - Normal(Gaussian) distribution에서 sampling을 할 경우, 분산을 일정하게 유지하면서 지속적인 exploration이 가능
  - std를 1로 고정함으로써 일정한 폭을 가지는 normal distribution에서 sampling

# Learning process

2. 환경에서 선택한 행동으로 한 time step을 진행한 후, 다음 상태와 보상을 받음

```
159 | | | | next_state, reward, done, _ = env.step(action)
```

3. Sample  $(s, a, r, s')$ 을 trajectories set에 저장

```
139 | | | | trajectories = deque()  
161 | | | | mask = 0 if done else 1  
162 | | | |  
163 | | | | trajectories.append((state, action, reward, mask))
```

# Learning process

4. 일정 sample들이 모이면 trajectories set으로 Actor & Critic network 업데이트

- Step 1 : Return 구하기

- time step 6까지 진행하고 episode가 끝났을 경우를 가정

$$G_1 = R_2 + \gamma R_3 + \gamma^2 R_4 + \gamma^3 R_5 + \gamma^4 R_6$$

$$G_2 = R_3 + \gamma R_4 + \gamma^2 R_5 + \gamma^3 R_6$$

$$G_3 = R_4 + \gamma R_5 + \gamma^2 R_6$$

$$G_4 = R_5 + \gamma R_6$$

$$G_5 = R_6$$

- 거꾸로 계산하며 계산해놓은 return값을 이용

$$G_5 = R_6$$

$$G_4 = R_5 + \gamma G_5$$

$$G_3 = R_4 + \gamma G_4$$

$$G_2 = R_3 + \gamma G_3$$

$$G_1 = R_2 + \gamma G_2$$

# Learning process

4. 일정 sample들이 모이면 trajectories set으로 Actor & Critic network 업데이트

- Step 1 : Return 구하기

- 거꾸로 계산하며 계산해놓은 return값을 이용

$$G_5 = R_6$$

$$G_4 = R_5 + \gamma G_5$$

$$G_3 = R_4 + \gamma G_4$$

$$G_2 = R_3 + \gamma G_3$$

$$G_1 = R_2 + \gamma G_2$$

```
def get_returns(rewards, masks, gamma):
    returns = torch.zeros_like(rewards)
    running_returns = 0

    for t in reversed(range(0, len(rewards))):
        running_returns = rewards[t] + masks[t] * gamma * running_returns
        returns[t] = running_returns

    returns = (returns - returns.mean()) / returns.std()

    return returns
```

# Learning process

4. 일정 sample들이 모이면 trajectories set으로 Actor & Critic network 업데이트
  - Step 2 : Critic network 업데이트
    - Critic Loss

$$J_V(\phi) = \frac{(V_\phi(s) - R)^2}{\text{Prediction} \quad \text{Target}}$$

```
47      # -----
48      # step 2: update critic
49      criterion = torch.nn.MSELoss()
50
51      values = critic(torch.Tensor(states))
52      targets = returns.unsqueeze(1)
53
54      critic_loss = criterion(values, targets)
55      critic_optimizer.zero_grad()
56      critic_loss.backward()
57      critic_optimizer.step()
```

# Learning process

- Optimization problem of theoretical TRPO → Surrogate objective function

$$\text{maximize}_{\theta} \quad \mathcal{L}_{\theta_{old}}(\theta) = \mathbb{E}_{s,a \sim \pi_{\theta_{old}}} \left[ \frac{\pi_{\theta}(a|s)}{\pi_{\theta_{old}}(a|s)} A_{\pi_{\theta_{old}}}(s,a) \right]$$

$$s.t. \quad \bar{D}_{KL}^{\rho_{\theta_{old}}}(\theta_{old}, \theta) \leq \delta$$

- But how do we solve it? Solution → Approximation!

$$\mathcal{L}_{\theta_{old}}(\theta) \approx g^T(\theta - \theta_{old}) \quad g \doteq \nabla_{\theta} \mathcal{L}_{\theta_{old}}(\theta) |_{\theta_{old}}$$

$$\bar{D}_{KL}^{\rho_{\theta_{old}}}(\theta_{old}, \theta) \approx \frac{1}{2}(\theta - \theta_{old})^T H(\theta - \theta_{old}) \quad H \doteq \nabla_{\theta}^2 \bar{D}_{KL}^{\rho_{\theta_{old}}}(\theta_{old}, \theta) |_{\theta_{old}}$$

# Learning process

- But how do we solve it? Solution → Approximation!

$$\mathcal{L}_{\theta_{old}}(\theta) \approx g^T(\theta - \theta_{old}) \quad g \doteq \nabla_{\theta} \mathcal{L}_{\theta_{old}}(\theta) |_{\theta_{old}}$$

$$\bar{D}_{KL}^{\rho_{\theta_{old}}}(\theta_{old}, \theta) \approx \frac{1}{2}(\theta - \theta_{old})^T H(\theta - \theta_{old}) \quad H \doteq \nabla_{\theta}^2 \bar{D}_{KL}^{\rho_{\theta_{old}}}(\theta_{old}, \theta) |_{\theta_{old}}$$

- Approximate optimization problem of practical TRPO

$$\text{maximize}_{\theta} \ g^T(\theta - \theta_{old})$$

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

# Learning process

- Approximate optimization problem of practical TRPO (argmax form)

$$\theta_{k+1} = \underset{\theta}{\operatorname{argmax}} g^T(\theta - \theta_k)$$

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

- Solution to approximate problem

$$\theta_{k+1} = \theta_k + \sqrt{\frac{2\delta}{g^T H^{-1} g}} H^{-1} g$$

Step size      Search direction

# Learning process

- Solution to approximate problem

$$\theta_{k+1} = \theta_k + \underbrace{\sqrt{\frac{2\delta}{g^T H^{-1} g}} H^{-1} g}_{\text{Maximal step}}$$

- TRPO adds a modification to this update rule → Backtracking line search

$$\theta_{k+1} = \theta_k + \underbrace{\alpha^j \sqrt{\frac{2\delta}{g^T H^{-1} g}} H^{-1} g}_{\text{Backtracking coefficient}}$$

# Learning process

## ❖ 정리

- 1) Find search direction and step size through CGM and Hessian of KL
  - Search direction :  $H^{-1}g$  (use CGM to solve  $Hx = g$  for  $x = H^{-1}g$ )
  - Step size :  $\sqrt{\frac{2\delta}{g^T H^{-1} g}}$
- 2) Do line search on that direction inside trust region through Backtracking line search

$$\theta_{k+1} = \theta_k + \alpha^j \sqrt{\frac{2\delta}{g^T H^{-1} g}} H^{-1} g$$



# Learning process

4. 일정 sample들이 모이면 trajectories set으로 Actor & Critic network 업데이트

- Step 3 : Actor loss의 gradient를 구하고, CGM을 통해 search direction 구하기

```
59      # -----
60      # step 3: get gradient of actor loss and search direction through conjugate gradient method
61      mu, std = actor(torch.Tensor(states))
62      old_policy = get_log_prob(actions, mu, std)
```

```
def get_log_prob(actions, mu, std):
    normal = Normal(mu, std)
    log_prob = normal.log_prob(actions)

    return log_prob
```

- `get_log_prob`

$$\pi_{\theta}(a|s) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(a - \mu_{\theta}(s))^2}{2\sigma^2}\right)$$

# Learning process

4. 일정 sample들이 모이면 trajectories set으로 Actor & Critic network 업데이트
- Step 3 : Actor loss의 gradient를 구하고, CGM을 통해 search direction 구하기

```
59      # -----
60      # step 3: get gradient of actor loss and search direction through conjugate gradient method
61      mu, std = actor(torch.Tensor(states))
62      old_policy = get_log_prob(actions, mu, std)
63      actor_loss = surrogate_loss(actor, values, targets, states, old_policy.detach(), actions)
```

```
def surrogate_loss(actor, values, target, states, old_policy, actions):
    mu, std = actor(torch.Tensor(states))
    new_policy = get_log_prob(actions, mu, std)

    advantages = target - values

    surrogate_loss = torch.exp(new_policy - old_policy) * advantages
    surrogate_loss = surrogate_loss.mean()

    return surrogate_loss
```

- **surrogate\_loss**

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

# Learning process

- 일정 sample들이 모이면 trajectories set으로 Actor & Critic network 업데이트
  - Step 3 : Actor loss의 gradient를 구하고, CGM을 통해 search direction 구하기

```
59      # -----
60      # step 3: get gradient of actor loss and search direction through conjugate gradient method
61      mu, std = actor(torch.Tensor(states))
62      old_policy = get_log_prob(actions, mu, std)
63      actor_loss = surrogate_loss(actor, values, targets, states, old_policy.detach(), actions)
64
65      actor_loss_grad = torch.autograd.grad(actor_loss, actor.parameters())
66      actor_loss_grad = flat_grad(actor_loss_grad)
```

- torch.autograd.grad(outputs, inputs) - actor.parameter : 4481(weight : 4352, bias : 129)

```
torch.autograd.grad(outputs, inputs, grad_outputs=None, retain_graph=None,
create_graph=False, only_inputs=True, allow_unused=False)
```

[SOURCE]

Computes and returns the sum of gradients of outputs w.r.t. the inputs.

- example

$$y = \sum_{i=1}^4 3 \times x_i$$

$$\frac{\partial y}{\partial x_i} = 3$$



# Learning process

## 4. 일정 sample들이 모이면 trajectories set으로 Actor & Critic network 업데이트

- Step 3 : Actor loss의 gradient를 구하고, CGM을 통해 search direction 구하기

```
59      # -----
60      # step 3: get gradient of actor loss and search direction through conjugate gradient method
61      mu, std = actor(torch.Tensor(states))
62      old_policy = get_log_prob(actions, mu, std)
63      actor_loss = surrogate_loss(actor, values, targets, states, old_policy.detach(), actions)
64
65      actor_loss_grad = torch.autograd.grad(actor_loss, actor.parameters())
66      actor_loss_grad = flat_grad(actor_loss_grad)
```

- flat\_grad(actor\_loss\_grad)

```
def flat_grad(grads):
    grad_flatten = []
    for grad in grads:
        grad_flatten.append(grad.view(-1))
    grad_flatten = torch.cat(grad_flatten)
    return grad_flatten
```

# Learning process

4. 일정 sample들이 모이면 trajectories set으로 Actor & Critic network 업데이트

- Step 3 : Actor loss의 gradient를 구하고, CGM을 통해 search direction 구하기

```
59      # -----
60      # step 3: get gradient of actor loss and search direction through conjugate gradient method
61      mu, std = actor(torch.Tensor(states))
62      old_policy = get_log_prob(actions, mu, std)
63      actor_loss = surrogate_loss(actor, values, targets, states, old_policy.detach(), actions)
64
65      actor_loss_grad = torch.autograd.grad(actor_loss, actor.parameters())
66      actor_loss_grad = flat_grad(actor_loss_grad)
67
68      search_dir = conjugate_gradient(actor, states, actor_loss_grad.data, nsteps=10)
69
70      actor_loss = actor_loss.data.numpy()
```

# Learning process

## 4. 일정 sample들이 모이면 trajectories set으로 Actor & Critic network 업데이트

- Step 3 : Actor loss의 gradient를 구하고, CGM을 통해 search direction 구하기
  - Use CGM to solve  $Hx = g$  for  $x = H^{-1}g$  ( $Hx = \nabla_{\theta}((\nabla_{\theta}\bar{D}_{KL}(\theta_{old} \parallel \theta))^T \cdot x)$ )

### Iterative method

```
r0 := b - Ax0
if r0 is sufficiently small, then return x0 as the result
p0 := r0
k := 0
repeat
    alpha_k := r_k^T r_k / p_k^T A p_k
    x_{k+1} := x_k + alpha_k p_k
    r_{k+1} := r_k - alpha_k A p_k
    if r_{k+1} is sufficiently small, then exit loop
    beta_k := r_{k+1}^T r_{k+1} / r_k^T r_k
    p_{k+1} := r_{k+1} + beta_k p_k
    k := k + 1
end repeat
return x_{k+1} as the result
```

[https://en.wikipedia.org/wiki/Conjugate\\_gradient\\_method#As\\_an\\_iterative\\_method](https://en.wikipedia.org/wiki/Conjugate_gradient_method#As_an_iterative_method)

```
def conjugate_gradient(actor, states, b, nsteps, residual_tol=1e-10):
    x = torch.zeros(b.size())
    r = b.clone()
    p = b.clone()
    rdotr = torch.dot(r, r)

    for i in range(nsteps): # nsteps = 10
        Ap = hessian_vector_product(actor, states, p, cg_damping=1e-1)
        alpha = rdotr / torch.dot(p, Ap)

        x += alpha * p
        r -= alpha * Ap

        new_rdotr = torch.dot(r, r)
        betta = new_rdotr / rdotr

        p = r + betta * p
        rdotr = new_rdotr

        if rdotr < residual_tol: # residual_tol = 0.0000000001
            break

    return x
```

# Learning process

4. 일정 sample들이 모이면 trajectories set으로 Actor & Critic network 업데이트

- Step 3 : Actor loss의 gradient를 구하고, CGM을 통해 search direction 구하기

```
def hessian_vector_product(actor, states, p, cg_damping=1e-1):
    p.detach()
    kl = kl_divergence(new_actor=actor, old_actor=actor, states=states)
    kl = kl.mean()

    kl_grad = torch.autograd.grad(kl, actor.parameters(), create_graph=True)
    kl_grad = flat_grad(kl_grad)

    kl_grad_p = (kl_grad * p).sum()
    kl_hessian = torch.autograd.grad(kl_grad_p, actor.parameters())
    kl_hessian = flat_hessian(kl_hessian)

    return kl_hessian + p * cg_damping # cg_damping = 0.1
```

- $Hx = \nabla_{\theta}((\nabla_{\theta}\bar{D}_{KL}(\theta_{old} \parallel \theta))^T \cdot x)$
- CGM의 Iteration을 돌 때마다 매번  $A_p$ 의 값을 업데이트해주기 위해서  
kl\_grad를 구한 후에 p를 곱해서 kl\_hessian을 구함

# Learning process

4. 일정 sample들이 모이면 trajectories set으로 Actor & Critic network 업데이트

- Step 3 : Actor loss의 gradient를 구하고, CGM을 통해 search direction 구하기

```
def hessian_vector_product(actor, states, p, cg_damping=1e-1):
    p.detach()
    kl = kl_divergence(new_actor=actor, old_actor=actor, states=states)
    kl = kl.mean()

    kl_grad = torch.autograd.grad(kl, actor.parameters(), create_graph=True)
    kl_grad = flat_grad(kl_grad)

    kl_grad_p = (kl_grad * p).sum()
    kl_hessian = torch.autograd.grad(kl_grad_p, actor.parameters())
    kl_hessian = flat_hessian(kl_hessian)

    return kl_hessian + p * cg_damping # cg_damping = 0.1
```

```
def flat_hessian(hessians):
    hessians_flatten = []
    for hessian in hessians:
        hessians_flatten.append(hessian.contiguous().view(-1))
    hessians_flatten = torch.cat(hessians_flatten).data
    return hessians_flatten
```

# Learning process

4. 일정 sample들이 모이면 trajectories set으로 Actor & Critic network 업데이트
- Step 3 : Actor loss의 gradient를 구하고, CGM을 통해 search direction 구하기

```
def kl_divergence(new_actor, old_actor, states):
    mu, std = new_actor(torch.Tensor(states))

    mu_old, std_old = old_actor(torch.Tensor(states))
    mu_old = mu_old.detach()
    std_old = std_old.detach()

    # kl divergence between old policy and new policy : D( pi_old || pi_new )
    # pi_old -> mu_old, std_old / pi_new -> mu, std
    # be careful of calculating KL-divergence. It is not symmetric metric.
    kl = torch.log(std / std_old) + (std_old.pow(2) + (mu_old - mu).pow(2)) / (2.0 * std.pow(2)) - 0.5
    return kl.sum(1, keepdim=True)
```

- KL-Divergence between two univariate Gaussians

$$KL(p, q) = - \int p(x) \log q(x) dx + \int p(x) \log p(x) dx$$

$$= \frac{1}{2} \log(2\pi\sigma_2^2) + \frac{\sigma_1^2 + (\mu_1 - \mu_2)^2}{2\sigma_2^2} - \frac{1}{2}(1 + \log 2\pi\sigma_1^2)$$

$$= \log \frac{\sigma_2}{\sigma_1} + \frac{\sigma_1^2 + (\mu_1 - \mu_2)^2}{2\sigma_2^2}.$$

<https://stats.stackexchange.com/questions/7440/kl-divergence-between-two-univariate-gaussians>



**CORE**  
Control + Optimization Research Lab

# Learning process

## 4. 일정 sample들이 모이면 trajectories set으로 Actor & Critic network 업데이트

- Step 3 : Actor loss의 gradient를 구하고, CGM을 통해 search direction 구하기

```
59      # -----
60      # step 3: get gradient of actor loss and search direction through conjugate gradient method
61      mu, std = actor(torch.Tensor(states))
62      old_policy = get_log_prob(actions, mu, std)
63      actor_loss = surrogate_loss(actor, values, targets, states, old_policy.detach(), actions)
64
65      actor_loss_grad = torch.autograd.grad(actor_loss, actor.parameters())
66      actor_loss_grad = flat_grad(actor_loss_grad)
67
68      search_dir = conjugate_gradient(actor, states, actor_loss_grad.data, nsteps=10)
69
70      actor_loss = actor_loss.data.numpy()
```

# Learning process

## ❖ 정리

- 1) Find search direction and step size through CGM and Hessian of KL
  - Search direction :  $H^{-1}g$  (use CGM to solve  $Hx = g$  for  $x = H^{-1}g$ ) **Clear!**
  - Step size :  $\sqrt{\frac{2\delta}{g^T H^{-1} g}}$
- 2) Do line search on that direction inside trust region through Backtracking line search

$$\theta_{k+1} = \theta_k + \alpha^j \sqrt{\frac{2\delta}{g^T H^{-1} g}} H^{-1} g$$

# Learning process

4. 일정 sample들이 모이면 trajectories set으로 Actor & Critic network 업데이트
  - Step 4 : Step size와 maximal step 구하기

```
72      # -----
73      # step 4: get step size and maximal step
74      gHg = (hessian_vector_product(actor, states, search_dir) * search_dir).sum(0, keepdim=True)
75      step_size = torch.sqrt(2 * args.max_kl / gHg)[0]
76      maximal_step = step_size * search_dir
```

- Step size :  $\sqrt{\frac{2\delta}{g^T H^{-1} g}} \quad (Hx = g)$
- Maximal step :  $\sqrt{\frac{2\delta}{g^T H^{-1} g}} H^{-1} g \quad (\text{max\_kl } \delta : 0.01)$

# Learning process

---

## ❖ 정리

- 1) Find search direction and step size through CGM and Hessian of KL
  - Search direction :  $H^{-1}g$  (use CGM to solve  $Hx = g$  for  $x = H^{-1}g$ ) **Clear!**
  - Step size :  $\sqrt{\frac{2\delta}{g^T H^{-1} g}}$  **Clear!**
- 2) Do line search on that direction inside trust region through Backtracking line search

$$\theta_{k+1} = \theta_k + \alpha^j \sqrt{\frac{2\delta}{g^T H^{-1} g}} H^{-1} g$$



# Learning process

4. 일정 sample들이 모이면 trajectories set으로 Actor & Critic network 업데이트
- Step 5 : Actor network를 업데이트하고, backtracking line search를 통해 trust region을 잡고 trust region안에서 업데이트

```
78      # -----
79      # step 5: update actor and perform backtracking line search for n iteration
80      params = flat_params(actor)
81
82      old_actor = Actor(state_size, action_size, args)
83      update_model(old_actor, params)
84
85      expected_improve = (actor_loss_grad * maximal_step).sum(0, keepdim=True)
86      expected_improve = expected_improve.data.numpy()
```

```
def flat_params(model):
    params = []
    for param in model.parameters():
        params.append(param.data.view(-1))
    params_flatten = torch.cat(params)
    return params_flatten
```

# Learning process

4. 일정 sample들이 모이면 trajectories set으로 Actor & Critic network 업데이트
- Step 5 : Actor network를 업데이트하고, backtracking line search를 통해 trust region을 잡고 trust region안에서 업데이트

```
78      # -----
79      # step 5: update actor and perform backtracking line search for n iteration
80      params = flat_params(actor)
81
82      old_actor = Actor(state_size, action_size, args)
83      update_model(old_actor, params)
84
85      expected_improve = (actor_loss_grad * maximal_step).sum(0, keepdim=True)
86      expected_improve = expected_improve.data.numpy()
```

```
def update_model(model, new_params):
    index = 0
    for params in model.parameters():
        params_length = len(params.view(-1))
        new_param = new_params[index: index + params_length]
        new_param = new_param.view(params.size())
        params.data.copy_(new_param)
        index += params_length
```

params_length	Weight	Bias
192		
64		
4096		
64	$3 \times 64 = 192$	64
64		
64	$64 \times 64 = 4096$	64
1		
192		
64		
4096		
64	$64 \times 1 = 64$	1
64		
64		
129	4352	129
1		

# Learning process

4. 일정 sample들이 모이면 trajectories set으로 Actor & Critic network 업데이트
  - Step 5 : Actor network를 업데이트하고, backtracking line search를 통해 trust region을 잡고 trust region안에서 업데이트

```
78      # -----
79      # step 5: update actor and perform backtracking line search for n iteration
80      params = flat_params(actor)
81
82      old_actor = Actor(state_size, action_size, args)
83      update_model(old_actor, params)
84
85      expected_improve = (actor_loss_grad * maximal_step).sum(0, keepdim=True)
86      expected_improve = expected_improve.data.numpy()
```

- **expected\_improve** : maximal step만큼 parameter space에서 움직였을 때 예상되는 performance의 변화

# Learning process

4. 일정 sample들이 모이면 trajectories set으로 Actor & Critic network 업데이트
  - Step 5 : Actor network를 업데이트하고, backtracking line search를 통해 trust region을 잡고 trust region안에서 업데이트

---

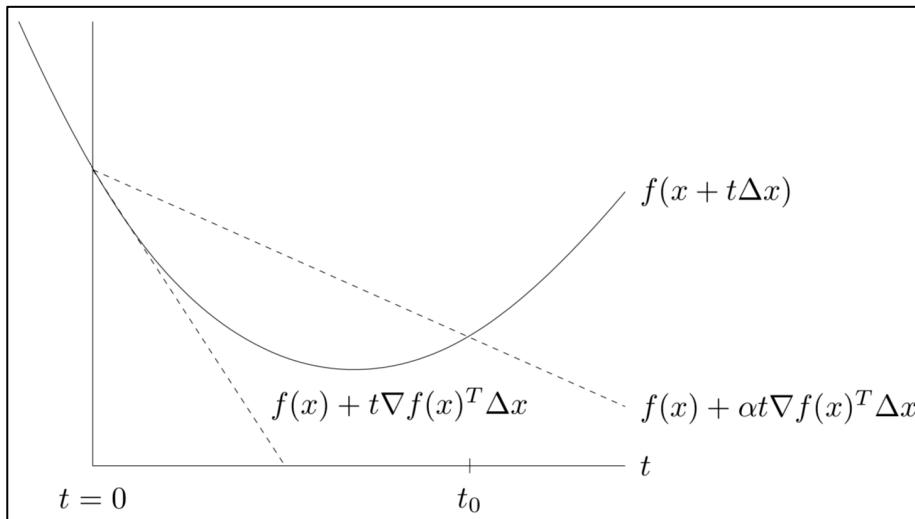
**Algorithm 9.2** Backtracking line search.

given a descent direction  $\Delta x$  for  $f$  at  $x \in \text{dom } f$ ,  $\alpha \in (0, 0.5)$ ,  $\beta \in (0, 1)$ .

$t := 1$ .

**while**  $f(x + t\Delta x) > f(x) + \alpha t \nabla f(x)^T \Delta x$ ,     $t := \beta t$ .

---



source : [https://web.stanford.edu/~boyd/cvxbook/bv\\_cvxbook.pdf](https://web.stanford.edu/~boyd/cvxbook/bv_cvxbook.pdf)

# Learning process

4. 일정 sample들이 모이면 trajectories set으로 Actor & Critic network 업데이트
- Step 5 : Actor network를 업데이트하고, backtracking line search를 통해 trust region을 잡고 trust region안에서 업데이트

```
88     backtrac_coef = 1.0
89     alpha = 0.5
90     beta = 0.5
91     flag = False
92
93     for i in range(10):
94         new_params = params + backtrac_coef * maximal_step
95         update_model(actor, new_params)
96
97         new_actor_loss = surrogate_loss(actor, values, targets, states, old_policy.detach(), actions)
98         new_actor_loss = new_actor_loss.data.numpy()
99
100        loss_improve = new_actor_loss - actor_loss
101        expected_improve *= backtrac_coef
102        improve_condition = loss_improve / expected_improve
103
104        kl = kl_divergence(new_actor=actor, old_actor=old_actor, states=states)
105        kl = kl.mean()
106
107        if kl < args.max_kl and improve_condition > alpha:
108            flag = True
109            break
110
111        backtrac_coef *= beta
112
113    if not flag:
114        params = flat_params(old_actor)
115        update_model(actor, params)
116        print('policy update does not impove the surrogate')
```



# Hyperparameter

```
14 parser = argparse.ArgumentParser()
15 parser.add_argument('--env_name', type=str, default="Pendulum-v0")
16 parser.add_argument('--load_model', type=str, default=None)
17 parser.add_argument('--save_path', default='./save_model/', help=' ')
18 parser.add_argument('--render', action="store_true", default=False)
19 parser.add_argument('--gamma', type=float, default=0.99)
20 parser.add_argument('--hidden_size', type=int, default=64)
21 parser.add_argument('--critic_lr', type=float, default=1e-3)
22 parser.add_argument('--max_kl', type=float, default=1e-2)
23 parser.add_argument('--max_iter_num', type=int, default=500)
24 parser.add_argument('--total_sample_size', type=int, default=2048)
25 parser.add_argument('--log_interval', type=int, default=5)
26 parser.add_argument('--goal_score', type=int, default=-300)
27 parser.add_argument('--logdir', type=str, default='./logs',
28 | | | | | help='tensorboardx logs directory')
29 args = parser.parse_args()
```

# Main loop

- Initialization
  - Seed - random number 고정
  - Actor network
  - TensorboardX
  - Recent rewards

```
119  def main():
120      env = gym.make(args.env_name)
121      env.seed(500)
122      torch.manual_seed(500)
123
124      state_size = env.observation_space.shape[0]
125      action_size = env.action_space.shape[0]
126      print('state size:', state_size)
127      print('action size:', action_size)
128
129      actor = Actor(state_size, action_size, args)
130      critic = Critic(state_size, args)
131      critic_optimizer = optim.Adam(critic.parameters(), lr=args.critic_lr)
132
133      writer = SummaryWriter(args.logdir)
134
135      recent_rewards = deque(maxlen=100)
136      episodes = 0
```

# Main loop

- Episode 진행
  - Initialize trajectories set
  - 상태에 따른 행동 선택
  - 다음 상태와 보상을 받음
  - Trajectories set에 저장

```
138     for iter in range(args.max_iter_num):  
139         trajectories = deque()  
140         steps = 0  
141  
142         while steps < args.total_sample_size:  
143             done = False  
144             score = 0  
145             episodes += 1  
146  
147             state = env.reset()  
148             state = np.reshape(state, [1, state_size])  
149  
150             while not done:  
151                 if args.render:  
152                     env.render()  
153  
154                 steps += 1  
155  
156                 mu, std = actor(torch.Tensor(state))  
157                 action = get_action(mu, std)  
158  
159                 next_state, reward, done, _ = env.step(action)  
160  
161                 mask = 0 if done else 1  
162  
163                 trajectories.append((state, action, reward, mask))  
164  
165                 next_state = np.reshape(next_state, [1, state_size])  
166                 state = next_state  
167                 score += reward  
168  
169                 if done:  
170                     recent_rewards.append(score)
```

# Main loop

- Train model
- Print & Visualize log
- Termination : 최근 100개의 episode의 평균 score가 -300보다 크다면
  - Save model
  - 학습 종료

```
172     actor.train(), critic.train()
173     train_model(actor, critic, critic_optimizer,
174                  | | |
174                  | | | trajectories, state_size, action_size)
175
176     writer.add_scalar('log/score', float(score), episodes)
177
178     if iter % args.log_interval == 0:
179         print('{0} iter | {0} episode | score_avg: {1:.2f}'.format(iter, episodes, np.mean(recent_rewards)))
180
181     if np.mean(recent_rewards) > args.goal_score:
182         if not os.path.isdir(args.save_path):
183             os.makedirs(args.save_path)
184
185         ckpt_path = args.save_path + 'model.pth'
186         torch.save(actor.state_dict(), ckpt_path)
187         print('Recent rewards exceed -300. So end')
188         break
```

# Train model

- Trajectories → Numpy array
  - Trajectories에 있는 2200개의 sample들을 각각 나눔
    - state - (2200, 3)
    - action - (2200, 1)
    - reward - (2200)
    - mask - (2200)

```
31 def train_model(actor, critic, critic_optimizer,
32                 |         |         |     trajectories, state_size, action_size):
33     trajectories = np.array(trajectories)
34     states = np.vstack(trajectories[:, 0])
35     actions = list(trajectories[:, 1])
36     rewards = list(trajectories[:, 2])
37     masks = list(trajectories[:, 3])
38
39     actions = torch.Tensor(actions).squeeze(1)
40     rewards = torch.Tensor(rewards).squeeze(1)
41     masks = torch.Tensor(masks)
```

# Train model

- **returns** - (2200)
- **values** - (2200, 1)
- **targets** - (2200, 1)

```
43     # -----
44     # step 1: get returns
45     returns = get_returns(rewards, masks, args.gamma)
46
47     # -----
48     # step 2: update critic
49     criterion = torch.nn.MSELoss()
50
51     values = critic(torch.Tensor(states))
52     targets = returns.unsqueeze(1)
53
54     critic_loss = criterion(values, targets)
55     critic_optimizer.zero_grad()
56     critic_loss.backward()
57     critic_optimizer.step()
```

# Train model

- **old\_policy** - (2200, 1)
- **actor\_loss\_grad** - (4481)
- **search\_dir** - (4481)

```
59      # -----
60      # step 3: get gradient of actor loss and search direction through conjugate gradient method
61      mu, std = actor(torch.Tensor(states))
62      old_policy = get_log_prob(actions, mu, std)
63      actor_loss = surrogate_loss(actor, values, targets, states, old_policy.detach(), actions)
64
65      actor_loss_grad = torch.autograd.grad(actor_loss, actor.parameters())
66      actor_loss_grad = flat_grad(actor_loss_grad)
67
68      search_dir = conjugate_gradient(actor, states, actor_loss_grad.data, nsteps=10)
69
70      actor_loss = actor_loss.data.numpy()
71
72      # -----
73      # step 4: get step size and maximal step
74      gHg = (hessian_vector_product(actor, states, search_dir) * search_dir).sum(0, keepdim=True)
75      step_size = torch.sqrt(2 * args.max_kl / gHg)[0]
76      maximal_step = step_size * search_dir
```

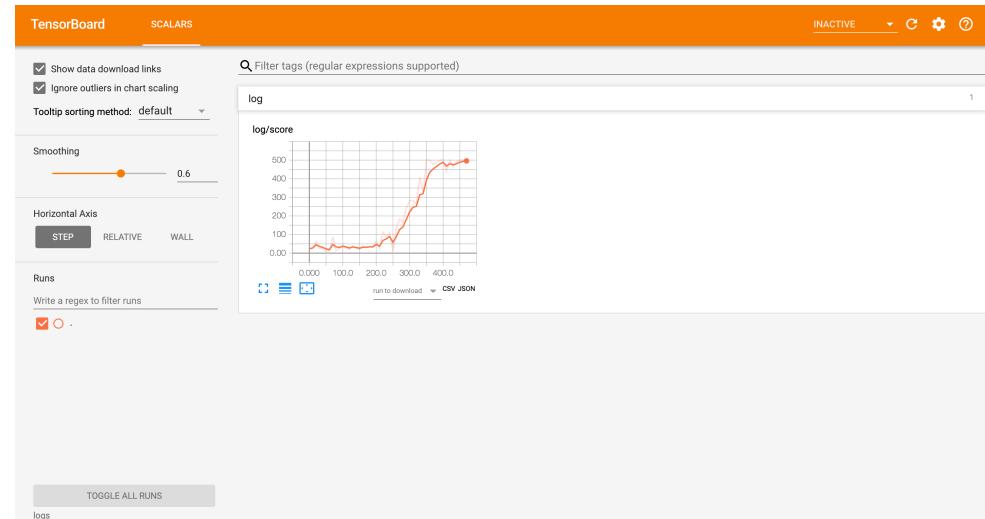
# Train model

- **params** - (4481)

```
78      # -----
79      # step 5: update actor and perform backtracking line search for n iteration
80      params = flat_params(actor)
81
82      old_actor = Actor(state_size, action_size, args)
83      update_model(old_actor, params)
84
85      expected_improve = (actor_loss_grad * maximal_step).sum(0, keepdim=True)
86      expected_improve = expected_improve.data.numpy()
87
88      backtrac_coef = 1.0
89      alpha = 0.5
90      beta = 0.5
91      flag = False
92
93      for i in range(10):
94          new_params = params + backtrac_coef * maximal_step
95          update_model(actor, new_params)
96
97          new_actor_loss = surrogate_loss(actor, values, targets, states, old_policy.detach(), actions)
98          new_actor_loss = new_actor_loss.data.numpy()
99
100         loss_improve = new_actor_loss - actor_loss
101         expected_improve *= backtrac_coef
102         improve_condition = loss_improve / expected_improve
103
104         kl = kl_divergence(new_actor=actor, old_actor=old_actor, states=states)
105         kl = kl.mean()
106
107         if kl < args.max_kl and improve_condition > alpha:
108             flag = True
109             break
110
111         backtrac_coef *= beta
```

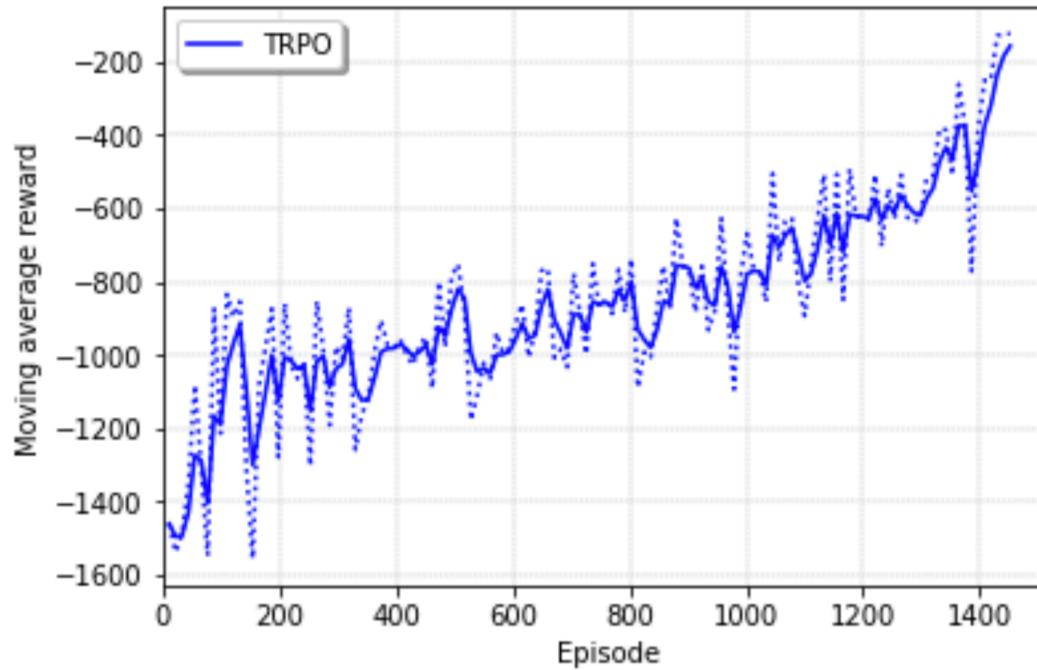
# Train & TensorboardX

- Terminal A - train 실행
  - conda activate env\_name
  - python train.py
- Terminal B - tensorboardX 실행
  - conda activate env\_name
  - tensorboard --logdir logs
  - (웹에서) localhost:6006



# Learning curve & Test

- Learning curve



- Test
  - `python test.py`

# Thank you



**CORE**  
Control + Optimization Research Lab