# Proximal Policy Optimization (PPO)

# Motivation

- In policy gradient method, we used gradient ascent to iteratively maximize the objective function.

$$\nabla_\theta J(\theta) = \frac{1}{N}\sum_{i=1}^{N}\left[\sum_{t=0}^{T-1}\nabla_\theta \log \pi_\theta\,(a_t|s_t)R(\tau)\right]$$

$$\theta = \theta + \alpha\nabla_\theta J(\theta)$$

- The gradient $\nabla_\theta J(\theta)$ tells you the direction of the steepest ascent.
- You take a step in the direction. The step size depends on the learning rate $\alpha$.

- The question is: what is the proper step size?

# Motivation

- If the step size is too large, we can fall of the cliff.
  - If we fall of the cliff, we resume exploration from a poorly performing state. It may take a long time to recover.

- However, if the step size is too small, it will take a long time for us to get to the top (optimal point).

# Motivation

- In order to decide how far we will move, we define a notion of "trusted region".
- We want to make sure that if we move (change policy) within the trusted region, the return is always better than or at least equal to the previous location (policy).

# Trusted Region Policy Optimization (TRPO)

- Expected discounted return $\eta$ following the policy $\pi$

$$\eta(\pi) = \mathbb{E}_{\tau\sim\pi}\left[\sum_{t=0}^{\infty}\gamma^t r_t\right]$$

  - improving the policy means we increase $\eta$.

- Suppose we have changed our policy from an old policy $\pi$ to a new policy $\tilde{\pi}$. Then, the following equation is true.

$$\eta(\tilde{\pi}) = \eta(\pi) + \mathbb{E}_{\tau\sim\tilde{\pi}}\left[\sum_{t=0}^{\infty}\gamma^t A_\pi(s_t, a_t)\right]$$

*S. Kakade and J. Langford, "Approximately Optimal Approximate Reinforcement Learning", 2002.*

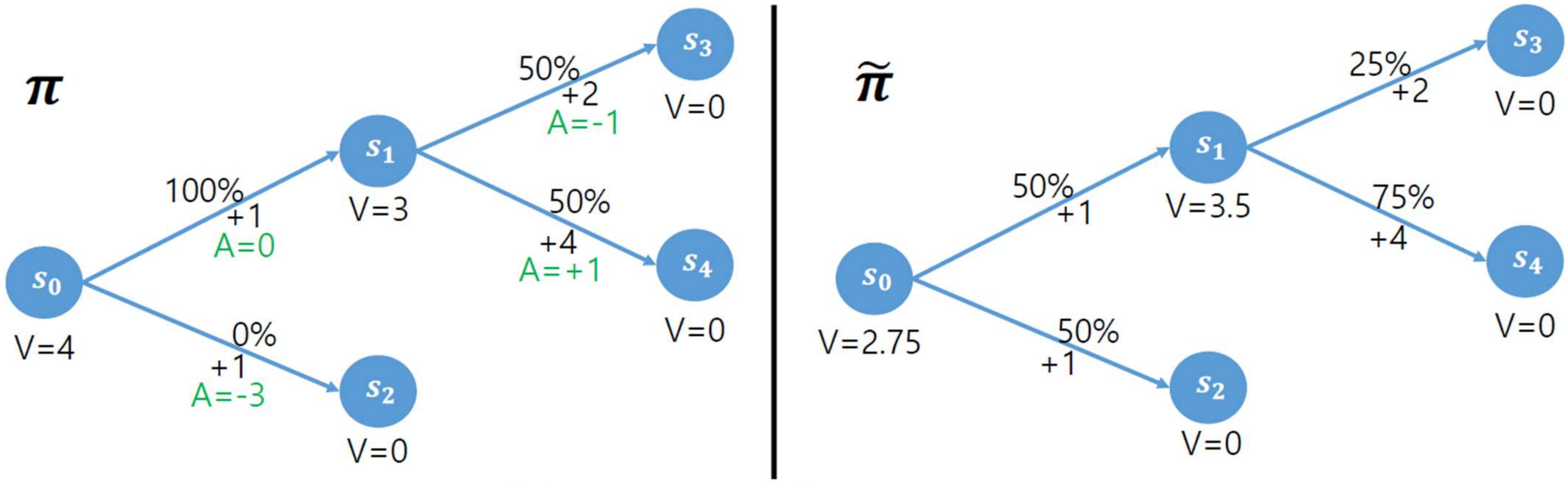
# Trusted Region Policy Optimization (TRPO)

- Proof

$$
\begin{aligned}
&\mathbb{E}_{\tau|\tilde{\pi}}\left[\sum_{t=0}^{\infty}\gamma^t A_\pi(s_t,a_t)\right] \\
&= \mathbb{E}_{\tau|\tilde{\pi}}\left[\sum_{t=0}^{\infty}\gamma^t (r(s_t)+\gamma V_\pi(s_{t+1})-V_\pi(s_t))\right] \\
&= \mathbb{E}_{\tau|\tilde{\pi}}\left[-V_\pi(s_0)+\sum_{t=0}^{\infty}\gamma^t r(s_t)\right] \\
&= -\mathbb{E}_{s_0}\left[V_\pi(s_0)\right]+\mathbb{E}_{\tau|\tilde{\pi}}\left[\sum_{t=0}^{\infty}\gamma^t r(s_t)\right] \\
&= -\eta(\pi)+\eta(\tilde{\pi})
\end{aligned}
$$

*S. Kakade and J. Langford, "Approximately Optimal Approximate Reinforcement Learning", 2002.*

# Trusted Region Policy Optimization (TRPO)




$$\eta(\tilde{\pi}) = \eta(\pi) + \mathbb{E}_{s_0,a_0,\cdots\sim\tilde{\pi}}\left[\sum_{t=0}^{\infty}\gamma^t A_\pi(s_t,a_t)\right]$$

$$
\begin{aligned}
\text{좌변} &= 2.75 \\
\text{우변} &= 4 + 0.5 * -3 + 0.5 * (0 + 0.25 * -1 + 0.75 * 1) \\
&= 4 - 1.5 \qquad + 0.5 * 0.5 \\
&= 2.5 \qquad + 0.25 \\
&= 2.75
\end{aligned}
$$

*https://github.com/minyoungjun/Pang-yo*

# Trusted Region Policy Optimization (TRPO)

- We change the equation in terms of state visitation frequency.

$$\eta(\tilde{\pi}) = \eta(\pi) + \mathbb{E}_{\tau\sim\tilde{\pi}}\left[\sum_{t=0}^{\infty}\gamma^t A_\pi(s_t, a_t)\right]$$

$$\Downarrow$$

$$\begin{aligned}
\eta(\tilde{\pi}) &= \eta(\pi) + \sum_{t=0}^{\infty}\sum_{s} P(s_t = s|\tilde{\pi}) \sum_{a} \tilde{\pi}(a|s)\gamma^t A_\pi(s,a) \\
&= \eta(\pi) + \sum_{s}\sum_{t=0}^{\infty}\gamma^t P(s_t = s|\tilde{\pi}) \sum_{a} \tilde{\pi}(a|s) A_\pi(s,a) \\
&= \eta(\pi) + \sum_{s}\rho_{\tilde{\pi}}(s) \sum_{a} \tilde{\pi}(a|s) A_\pi(s,a).
\end{aligned}$$

- $\rho_\pi(s)$ is the discounted visitation frequency

$$\rho_\pi(s) = P(s_0 = s) + \gamma P(s_1 = s) + \gamma^2 P(s_2 = s) + \ldots$$

# Trusted Region Policy Optimization (TRPO)

- Now we have the following equation.

$$\eta(\tilde{\pi}) = \eta(\pi) + \underline{\sum_s \rho_{\tilde{\pi}}(s) \sum_a \tilde{\pi}(a|s) A_\pi(s,a)}$$

- If we make $\sum_s \rho_{\tilde{\pi}}(s) \sum_a \tilde{\pi}(a|s) A_\pi(s,a)$ positive, $\eta$ will be increased.

# Trusted Region Policy Optimization (TRPO)

- The problem is we cannot get $\rho_{\tilde{\pi}}(s)$ because we are sampling trajectories using the old policy.

$$\eta(\tilde{\pi}) = \eta(\pi) + \sum_s \rho_{\tilde{\pi}}(s) \sum_a \tilde{\pi}(a|s) A_\pi(s,a)$$

- Thus we approximate $\eta(\tilde{\pi})$ like this.

$$L_\pi(\tilde{\pi}) = \eta(\pi) + \sum_s \rho_\pi(s) \sum_a \tilde{\pi}(a|s) A_\pi(s,a)$$

- It can be shown that the followings are true.
  - assuming that $\pi_\theta(a|s)$ is differentiable with respect to $\theta$.

$$L_{\pi_{\theta_0}}(\pi_{\theta_0}) = \eta(\pi_{\theta_0}),$$
$$\nabla_\theta L_{\pi_{\theta_0}}(\pi_\theta)\big|_{\theta=\theta_0} = \nabla_\theta \eta(\pi_\theta)\big|_{\theta=\theta_0}$$

# Trusted Region Policy Optimization (TRPO)

- It means that $L_\pi(\tilde{\pi})$ is similar to $\eta(\tilde{\pi})$ when $\theta$ is similar to $\theta_0$.

- In order to quantify the "similarity" between $L_\pi(\tilde{\pi})$ and $\eta(\tilde{\pi})$, we use the notion of total variation divergence $D_{TV}(p \parallel q)$.

$$D_{TV}(p \parallel q) = \frac{1}{2}\sum_i |p_i - q_i|$$

$$D_{TV}^{\max}(\pi, \tilde{\pi}) = \max_s D_{TV}\big(\pi(\cdot\,|s) \parallel \tilde{\pi}(\cdot\,|s)\big)$$

- It can be shown that the following bound holds.

$$\eta(\pi_{\text{new}}) \geq L_{\pi_{\text{old}}}(\pi_{\text{new}}) - \frac{4\epsilon\gamma}{(1-\gamma)^2}\alpha^2 \quad \textit{where } \epsilon = \max_{s,a} |A_\pi(s,a)|$$

$$\alpha = D_{\text{TV}}^{\max}(\pi_{\text{old}}, \pi_{\text{new}})$$

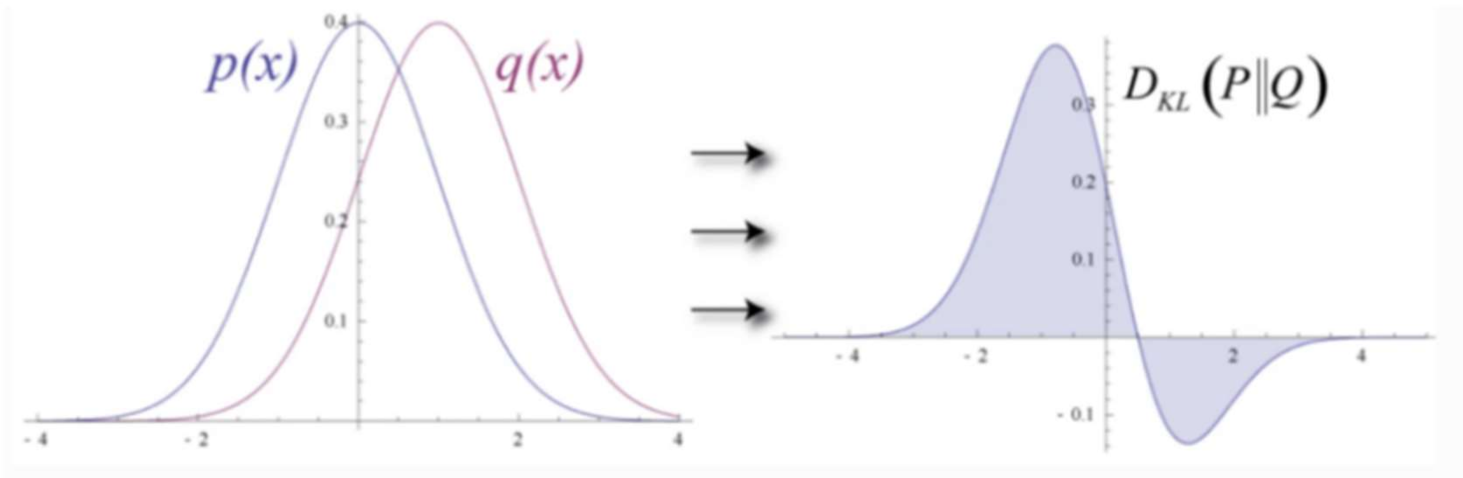
# Trusted Region Policy Optimization (TRPO)

- It can also be shown that

$$D_{TV}(p \parallel q)^2 \leq D_{KL}(p \parallel q)$$

  - $D_{KL}$ is the KL divergence which measure the distance between two distributions.

$$D_{KL}(P||Q) = \mathbb{E}_x \log \frac{P(x)}{Q(x)}$$



# Trusted Region Policy Optimization (TRPO)

- Using KL-divergence, we can rewrite the equation as follows.

$$\eta(\tilde{\pi}) \geq L_{\pi}(\tilde{\pi}) - C D_{KL}^{\max}(\pi, \tilde{\pi}), \qquad \text{where } C = \frac{4\epsilon\gamma}{(1-\gamma)^2}$$

- Now, in each step, we are going to choose a new policy that will maximize $L_{\pi}(\tilde{\pi}) - C D_{KL}^{\max}(\pi, \tilde{\pi})$.

- In the worst case, we can select the old policy $\pi$ as the new policy $\tilde{\pi}$, which means $L_{\pi}(\tilde{\pi}) - C D_{KL}^{\max}(\pi, \tilde{\pi}) = \mathrm{L}_{\pi}(\pi) = \eta(\pi)$
- Otherwise, we are always increasing $\eta$!

- We can see that the objective function $L_{\pi}(\tilde{\pi}) - C D_{KL}^{\max}(\pi, \tilde{\pi})$ becomes smaller when the difference between $\pi$ and $\tilde{\pi}$ is large. This limits the amount of policy change to a region we call "trusted region".

# Parameterizing the Policies

- Since we are using neural networks to parameterize policies, we can write our optimization goal as:

$$\text{maximize}_\theta \left[L_{\theta_{old}}(\theta) - C D_{KL}^{\max}(\theta_{old}, \theta)\right]$$

- Instead of using the penalty term, we can change this into an optimization with constraint.

$$\text{maximize}_\theta \; L_{\theta_{old}}(\theta)$$
$$\text{subject to } D_{KL}^{\max}(\theta_{old}, \theta) \leq \delta$$

- $C$ is typically a very large number, which makes the step size very small.
- Instead of $C$, we use a parameter $\delta$ which limits the difference between the old policy and the new policy.

# Parameterizing the Policies

- Now $D_{KL}^{\max}(\theta_{old}, \theta) = \max_s D_{KL}(\theta_{old} \parallel \theta)$, which cannot be calculated because we potentially have a very large number of states.

- As a heuristic approximation, we use average KL divergence.

$$\overline{D}_{\mathrm{KL}}^{\rho}(\theta_1, \theta_2) := \mathbb{E}_{s\sim\rho}\left[D_{\mathrm{KL}}(\pi_{\theta_1}(\cdot|s) \parallel \pi_{\theta_2}(\cdot|s))\right]$$

- Thus, we change the optimization goal as follows:

$$\begin{aligned}&\underset{\theta}{\text{maximize}}\, L_{\theta_{\text{old}}}(\theta)\\&\text{subject to } \overline{D}_{\mathrm{KL}}^{\rho_{\theta_{\text{old}}}}(\theta_{\text{old}}, \theta) \le \delta.\end{aligned}$$

# Sample-based Estimation

- Now we need a way to solve the optimization problem.

$$\underset{\theta}{\text{maximize}}\, L_{\theta_{\text{old}}}(\theta)$$
$$\text{subject to } \overline{D}_{\text{KL}}^{\rho_{\theta_{\text{old}}}}(\theta_{\text{old}}, \theta) \le \delta.$$

- We can write out the equation for $L_{\theta_{old}}(\theta)$.

$$\underset{\theta}{\text{maximize}} \sum_s \rho_{\theta_{\text{old}}}(s) \sum_a \pi_\theta(a|s) A_{\theta_{\text{old}}}(s,a)$$
$$\text{subject to } \overline{D}_{\text{KL}}^{\rho_{\theta_{\text{old}}}}(\theta_{\text{old}}, \theta) \le \delta.$$

- We would like to simplify this equation by getting rid of the two summations using sampling.

# Sample-based Estimation

- The first term $\sum_s \rho_{\theta_{old}}(s)$ expresses the summation over state visitation frequency.

- We can replace it by sampling states from state visitation as $\mathbb{E}_{s\sim\rho(\pi_{\theta_{old}})}$.
- Our equation becomes:

$$\mathbb{E}_{s\sim\rho(\pi_{\theta_{\mathrm{old}}})}\left[\sum_a \pi_\theta(a|s) A_{\pi_{\theta_{\mathrm{old}}}}(s,a)\right]$$

서강대학교 SOGANG UNIVERSITY

# Sample-based Estimation

- In order to replace the sum over actions $\sum_a \pi_\theta(a|s)$, we use importance sampling.

- Let $q$ be the sampling distribution, and $a$ is sampled from $q$. Then, we can rewrite our equation as:

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

- Since we are sampling actions from the old policy, our equation is:

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

# Sample-based Estimation

- Finally, our objective function becomes:

$$\underset{\theta}{\text{maximize}}\ \mathbb{E}_{s,a\sim\pi_{\theta_{\text{old}}}}\left[\frac{\pi_\theta(a|s)}{\pi_{\theta_{\text{old}}}(a|s)}\,A_{\pi_{\theta_{\text{old}}}}(s,a)\right]$$
$$\text{subject to}\ \ \mathbb{E}_{s\sim\pi_{\theta_{\text{old}}}}\left[D_{KL}\left(\pi_{\theta_{\text{old}}}(\cdot\,|s)\,\|\,\pi_\theta(\cdot\,|s)\right)\right]\le\delta$$

- The TRPO (Trusted Region Policy Optimization) algorithm solves this constrained optimization problem with Lagrange multipler and line search.

*J. Schulman et al. "Trust Region Policy Optimization", 2015.*

# Proximal Policy Optimization

- PPO is a more simple and practical version of TRPO.
- We start from the TRPO objective function.

$$\underset{\theta}{\text{maximize}}\ \mathbb{E}_{s,a\sim\pi_{\theta_{\text{old}}}}\left[\frac{\pi_\theta(a|s)}{\pi_{\theta_{\text{old}}}(a|s)}\,A_{\pi_{\theta_{\text{old}}}}(s,a)\right]$$
$$\text{subject to}\ \mathbb{E}_{s\sim\pi_{\theta_{\text{old}}}}\left[D_{KL}\left(\pi_{\theta_{\text{old}}}(\cdot\,|s)\|\,\pi_\theta(\cdot\,|s)\right)\right]\le\delta$$

- Unlike TRPO, PPO does not use any constraints in the objective function.

- There are two types of PPO algorithm
  - PPO-clipped
  - PPO-penalty

# PPO with a clipped objective

- From the TRPO objective function, we first remove the constraint.
- Now, our goal is to maximize $L(\theta)$.

$$L(\theta) = \mathbb{E}_t \left[ \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)} A_t \right]$$

- In the equation, $\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}$ is the probability ratio of the new policy to the old policy. We denote this term as $r_t(\theta)$.

- Now the objective function becomes:

$$L(\theta) = \mathbb{E}_t \left[ r_t(\theta) A_t \right]$$

# PPO with a clipped objective

- Since we have removed the constraint, updating policy according to the objective function may move the policy out of the trusted region.

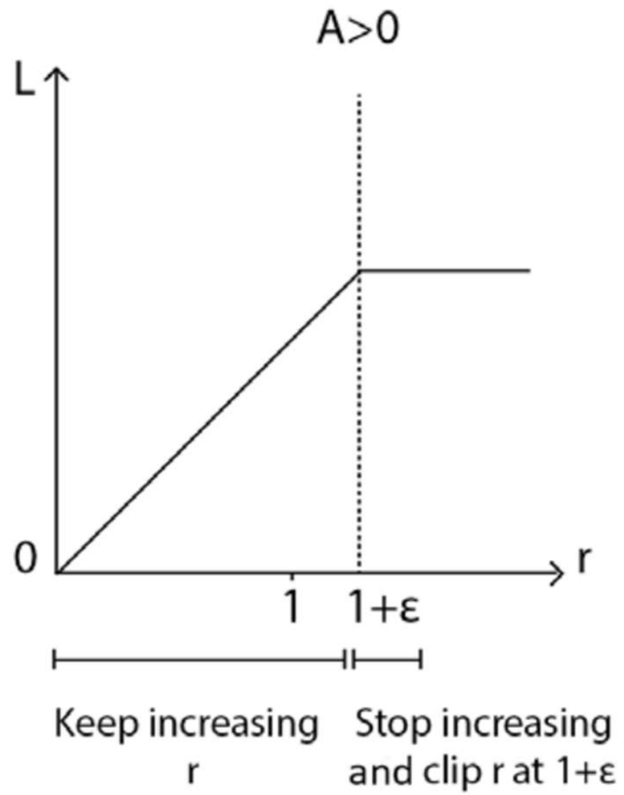
- To avoid that, we modify our objective function using a clipping function.
- The goal of using the clipping function is to make the new policy close to the old policy.

$$L(\theta) = \mathbb{E}_t[\min(r_t(\theta)A_t, \mathrm{clip}(r_t(\theta), 1 - \epsilon, 1 + \epsilon)A_t)]$$

- The first term, $r_t(\theta)A_t$, is the original objective.
- The second term, $\mathrm{clip}(r_t(\theta), 1 - \epsilon, 1 + \epsilon)A_t$, is the clipped objective.

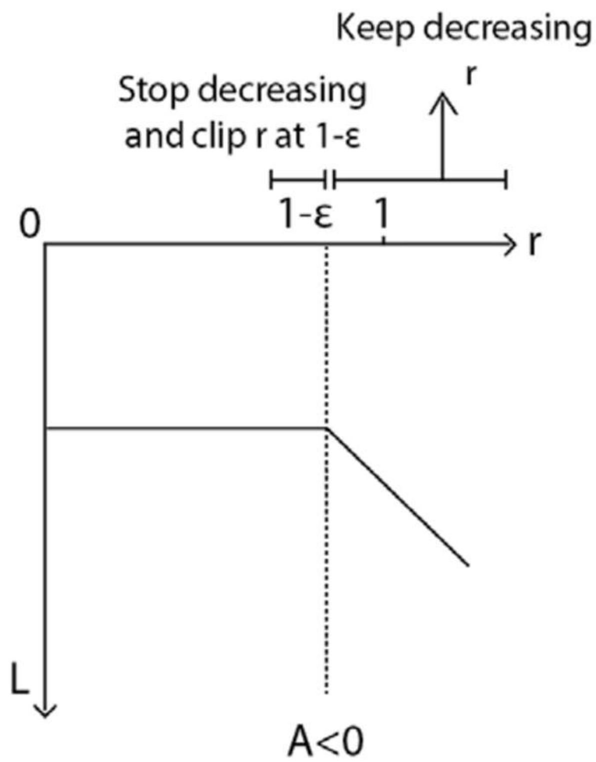
# PPO with a clipped objective

- Suppose the advantage $A_t$ is positive, $A_t > 0$.
- It means that the corresponding action should be preferred over the average of all other actions.
- So we can increase the value of $r_t(\theta)$ for that action so that it will have a greater chance of being selected.
- However, while increasing $r_t(\theta)$, we should not increase it too much that it goes far away from the old policy.
- To prevent this, we clip $r_t(\theta)$ at $1 + \epsilon$.
  - $\epsilon$ is a hyperparameter.



# PPO with a clipped objective

- Suppose the advantage $A_t$ is negative, $A_t < 0$.
- It means that the corresponding action should not be preferred over the average of all other actions.
- So we can decrease the value of $r_t(\theta)$ for that action so that it will have a lower chance of being selected.
- However, while increasing $r_t(\theta)$, we should not decrease it too much that it goes far away from the old policy.
- To prevent this, we clip $r_t(\theta)$ at $1 - \epsilon$.



# PPO with a penalized objective

- In PPO-penalty method, we use the penalty term instead of the constraint in the opmization problem.

$$L(\theta) = \mathbb{E}_t \left[ \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)} A_t - \beta \text{KL}[\pi_{\theta_{\text{old}}}(\cdot|s_t), \pi_\theta(\cdot|s_t)] \right]$$

- $\beta$ is the penalty coefficient which is dynamically adjusted.

- Let $d = KL[\pi_{\theta_{old}}(\cdot|s_t), \pi_\theta(\cdot|s_t)]$, and let $\delta$ be the target KL convergence. Then, we set the value of $\beta$ as follows.

  – If $d$ is greater than or equal to $1.5\delta$, then we set $\beta_{i+1} = 2\beta_i$.
  – If $d$ is less than or equal to $\delta/1.5$, then we set $\beta_{i+1} = \beta_i/2$.

서강대학교 SOGANG UNIVERSITY

# Solving CartPole with PPO-clipped [ex022]

- CartPole is an environment with discrete actions.

- Library imports and hyper-parameters

```
# https://github.com/seungeunrho/minimalRL/blob/master/ppo.py

import gym
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torch.distributions import Categorical

#Hyperparameters
learning_rate = 0.0005
gamma         = 0.98
lmbda         = 0.95
eps_clip      = 0.1
K_epoch       = 3
T_horizon     = 20
```

# Solving CartPole with PPO-clipped [ex022]

- main function (1/2)
  - We create a PPO agent and start running the episodes.
  - In each episode, we go through T_horizon steps and collect transitions.
  - We select an action based on the stochastic policy calculated from the model.

```
def main():
    env = gym.make('CartPole-v1')
    model = PPO()
    score = 0.0
    print_interval = 20

    for n_epi in range(10000):
        s = env.reset()
        done = False
        while not done:
            for t in range(T_horizon):
                prob = model.pi(torch.from_numpy(s).float())
                m = Categorical(prob)
                a = m.sample().item()
                s_prime, r, done, info = env.step(a)
```

# Solving CartPole with PPO-clipped [ex022]

- main function (2/2)
  - When we save the transition, we include the probability of action.
  - Reward is scaled down to 1/100.
  - After each T_horizon steps, we update the network by calling model.train_net().

```
                model.put_data((s, a, r/100.0, s_prime, prob[a].item(), done))
                s = s_prime

                score += r
                if done:
                    break

            model.train_net()

        if n_epi%print_interval==0 and n_epi!=0:
            print("# of episode :{}, avg score : {:.1f}".format(n_epi, score/print_interval))
            score = 0.0

    env.close()

if __name__ == '__main__':
    main()
```

# Solving CartPole with PPO-clipped [ex022]

- Definition of class PPO
  - We have a shared hidden layer of 256 neurons.
  - After hidden layer, we have a layer that calculates the probability of actions.
  - Also, we have a layer that calculates value of the states.

```
class PPO(nn.Module):
    def __init__(self):
        super(PPO, self).__init__()
        self.data = []

        self.fc1   = nn.Linear(4,256)
        self.fc_pi = nn.Linear(256,2)
        self.fc_v  = nn.Linear(256,1)
        self.optimizer = optim.Adam(self.parameters(), lr=learning_rate)

    def pi(self, x, softmax_dim = 0):
        x = F.relu(self.fc1(x))
        x = self.fc_pi(x)
        prob = F.softmax(x, dim=softmax_dim)
        return prob

    def v(self, x):
        x = F.relu(self.fc1(x))
        v = self.fc_v(x)
        return v
```

# Solving CartPole with PPO-clipped [ex022]

- When we store transition, we add prob_a, which is the probability of action.
- This is required when we calculate the policy ratio.

$$\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}$$

```python
def put_data(self, transition):
    self.data.append(transition)

def make_batch(self):
    s_lst, a_lst, r_lst, s_prime_lst, prob_a_lst, done_lst = [], [], [], [], [], []
    for transition in self.data:
        s, a, r, s_prime, prob_a, done = transition

        s_lst.append(s)
        a_lst.append([a])
        r_lst.append([r])
        s_prime_lst.append(s_prime)
        prob_a_lst.append([prob_a])
        done_mask = 0 if done else 1
        done_lst.append([done_mask])

    s,a,r,s_prime,done_mask, prob_a = torch.tensor(s_lst, dtype=torch.float), torch.tensor(a_lst), \
                                      torch.tensor(r_lst), torch.tensor(s_prime_lst, dtype=torch.float), \
                                      torch.tensor(done_lst, dtype=torch.float), torch.tensor(prob_a_lst)
    self.data = []
    return s, a, r, s_prime, done_mask, prob_a
```

# Solving CartPole with PPO-clipped [ex022]

- GAE (Generalized Advantage Estimation)
  - A better way to calculate advantage in trusted region methods.

Advantage : $\hat{A}_t = -V(s_t) + r_t + \gamma r_{t+1} + \cdots + \gamma^{T-t+1} r_{T-1} + \gamma^{T-t} V(s_T)$

GAE : $\hat{A}_t = \delta_t + (\gamma\lambda)\delta_{t+1} + \cdots + \cdots + (\gamma\lambda)^{T-t+1}\delta_{T-1},$

where $\delta_t = r_t + \gamma V(s_{t+1}) - V(s_t)$

*J. Schulman et al. "High-Dimensional Continuous Control Using Generalized Advantage Estimation", 2015.*

# Solving CartPole with PPO-clipped [ex022]

- Training the network (1/3)
  - Since T_horizon is set to 20, we have 20 transitions in a batch.
  - Using this batch, we update the network K_epoch times. Here, K_epoch is 3.
  - First, we calculate $\delta$ in GAE.

```
def train_net(self):
    s, a, r, s_prime, done_mask, prob_a = self.make_batch()

    for i in range(K_epoch):
        td_target = r + gamma * self.v(s_prime) * done_mask
        delta = td_target - self.v(s)
        delta = delta.detach().numpy()
```

$$\begin{pmatrix} \delta_1 \\ \delta_2 \\ \delta_3 \\ \vdots \\ \delta_t \end{pmatrix} = \begin{pmatrix} r_1 \\ r_2 \\ r_3 \\ \vdots \\ r_t \end{pmatrix} + \gamma \begin{pmatrix} V(s_2) \\ V(s_3) \\ V(s_4) \\ \vdots \\ V(s_{t+1}) \end{pmatrix} - \begin{pmatrix} V(s_1) \\ V(s_2) \\ V(s_3) \\ \\ V(s_t) \end{pmatrix}$$

# Solving CartPole with PPO-clipped [ex022]

- Training the network (2/3)
  - Using the $\delta$s, we calculate the GAE $\hat{A}_t$

```
advantage_lst = []
advantage = 0.0
for delta_t in delta[::-1]:
    advantage = gamma * lmbda * advantage + delta_t[0]
    advantage_lst.append([advantage])
advantage_lst.reverse()
advantage = torch.tensor(advantage_lst, dtype=torch.float)
```

$$\hat{A}_t = \delta_t + (\gamma\lambda)\delta_{t+1} + \cdots + \cdots + (\gamma\lambda)^{T-t+1}\delta_{T-1},$$
$$\text{where} \quad \delta_t = r_t + \gamma V(s_{t+1}) - V(s_t)$$

# Solving CartPole with PPO-clipped [ex022]

- Training the network (3/3)
  - We calculate the ratio $\mathrm{r_t}(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}$
  - surr1 is $r_t(\theta)A_t$
  - surr2 is $\mathrm{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)A_t$
  - The first term of loss is $\min(r_t(\theta)A_t, \mathrm{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)A_t)$
  - The second term of loss is the MSE (or smooth_l1_loss) between target and predicted value of $s$.

```
pi = self.pi(s, softmax_dim=1)
pi_a = pi.gather(1,a)
ratio = torch.exp(torch.log(pi_a) - torch.log(prob_a))  # a/b == exp(log(a)-log(b))

surr1 = ratio * advantage
surr2 = torch.clamp(ratio, 1-eps_clip, 1+eps_clip) * advantage
loss = -torch.min(surr1, surr2) + F.smooth_l1_loss(self.v(s) , td_target.detach())

self.optimizer.zero_grad()
loss.mean().backward()
self.optimizer.step()
```

# Solving Pendulum with PPO-clipped [ex023]

- Pendulum-v0 is an environment with continuous actions.

```
# https://github.com/seungeunrho/minimalRL/blob/master/ppo-continuous.py

import gym
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torch.distributions import Normal
import numpy as np

#Hyperparameters
learning_rate  = 0.0003
gamma           = 0.9
lmbda           = 0.9
eps_clip        = 0.2
K_epoch         = 10
rollout_len    = 3
buffer_size    = 30
minibatch_size = 32
```

# Solving Pendulum with PPO-clipped [ex023]

- PPO agent (1/4)

```
class PPO(nn.Module):
    def __init__(self):
        super(PPO, self).__init__()
        self.data = []

        self.fc1   = nn.Linear(3,128)
        self.fc_mu = nn.Linear(128,1)
        self.fc_std  = nn.Linear(128,1)
        self.fc_v = nn.Linear(128,1)
        self.optimizer = optim.Adam(self.parameters(), lr=learning_rate)
        self.optimization_step = 0

    def pi(self, x, softmax_dim = 0):
        x = F.relu(self.fc1(x))
        mu = 2.0*torch.tanh(self.fc_mu(x))
        std = F.softplus(self.fc_std(x))
        return mu, std

    def v(self, x):
        x = F.relu(self.fc1(x))
        v = self.fc_v(x)
        return v

    def put_data(self, transition):
        self.data.append(transition)
```

# Solving Pendulum with PPO-clipped [ex023]

- PPO agent (2/4)

```
def make_batch(self):
    s_batch, a_batch, r_batch, s_prime_batch, prob_a_batch, done_batch = [], [], [], [], [], []
    data = []

    for j in range(buffer_size):
        for i in range(minibatch_size):
            rollout = self.data.pop()
            s_lst, a_lst, r_lst, s_prime_lst, prob_a_lst, done_lst = [], [], [], [], [], []

            for transition in rollout:
                s, a, r, s_prime, prob_a, done = transition

                s_lst.append(s)
                a_lst.append([a])
                r_lst.append([r])
                s_prime_lst.append(s_prime)
                prob_a_lst.append([prob_a])
                done_mask = 0 if done else 1
                done_lst.append([done_mask])

            s_batch.append(s_lst)
            a_batch.append(a_lst)
            r_batch.append(r_lst)
            s_prime_batch.append(s_prime_lst)
            prob_a_batch.append(prob_a_lst)
            done_batch.append(done_lst)

        mini_batch = torch.tensor(s_batch, dtype=torch.float), torch.tensor(a_batch, dtype=torch.float), \
                      torch.tensor(r_batch, dtype=torch.float), torch.tensor(s_prime_batch, dtype=torch.float), \
                      torch.tensor(done_batch, dtype=torch.float), torch.tensor(prob_a_batch, dtype=torch.float)
        data.append(mini_batch)

    return data
```

서강대학교
SOGANG UNIVERSITY

# Solving Pendulum with PPO-clipped [ex023]

- PPO agent (3/4)

```
def calc_advantage(self, data):
    data_with_adv = []
    for mini_batch in data:
        s, a, r, s_prime, done_mask, old_log_prob = mini_batch
        with torch.no_grad():
            td_target = r + gamma * self.v(s_prime) * done_mask
            delta = td_target - self.v(s)
        delta = delta.numpy()

        advantage_lst = []
        advantage = 0.0
        for delta_t in delta[::-1]:
            advantage = gamma * lmbda * advantage + delta_t[0]
            advantage_lst.append([advantage])
        advantage_lst.reverse()
        advantage = torch.tensor(advantage_lst, dtype=torch.float)
        data_with_adv.append((s, a, r, s_prime, done_mask, old_log_prob, td_target, advantage))

    return data_with_adv
```

# Solving Pendulum with PPO-clipped [ex023]

- PPO agent (4/4)

```
def train_net(self):
    if len(self.data) == minibatch_size * buffer_size:
        data = self.make_batch()
        data = self.calc_advantage(data)

        for i in range(K_epoch):
            for mini_batch in data:
                s, a, r, s_prime, done_mask, old_log_prob, td_target, advantage = mini_batch

                mu, std = self.pi(s, softmax_dim=1)
                dist = Normal(mu, std)
                log_prob = dist.log_prob(a)
                ratio = torch.exp(log_prob - old_log_prob)  # a/b == exp(log(a)-log(b))

                surr1 = ratio * advantage
                surr2 = torch.clamp(ratio, 1-eps_clip, 1+eps_clip) * advantage
                loss = -torch.min(surr1, surr2) + F.smooth_l1_loss(self.v(s) , td_target)

                self.optimizer.zero_grad()
                loss.mean().backward()
                nn.utils.clip_grad_norm_(self.parameters(), 1.0)
                self.optimizer.step()
                self.optimization_step += 1
```

# Solving Pendulum with PPO-clipped [ex023]

- main function (1/2)

```
def main():
    env = gym.make('Pendulum-v0')
    model = PPO()
    score = 0.0
    print_interval = 20
    rollout = []

    for n_epi in range(10000):
        s = env.reset()
        done = False
        while not done:
            for t in range(rollout_len):
                mu, std = model.pi(torch.from_numpy(s).float())
                dist = Normal(mu, std)
                a = dist.sample()
                log_prob = dist.log_prob(a)
                s_prime, r, done, info = env.step([a.item()])
```

# Solving Pendulum with PPO-clipped [ex023]

- main function (2/2)

```
                rollout.append((s, a, r/10.0, s_prime, log_prob.item(), done))
                if len(rollout) == rollout_len:
                    model.put_data(rollout)
                    rollout = []

                s = s_prime
                score += r
                if done:
                    break

            model.train_net()

        if n_epi%print_interval==0 and n_epi!=0:
            print("# of episode :{}, avg score : {:.1f}, opt step: {}".format(n_epi, score/print_interval, model.optimization_step))
            score = 0.0

    env.close()

if __name__ == '__main__':
    main()
```

# End of Class

Questions?

Email: jso1@sogang.ac.kr