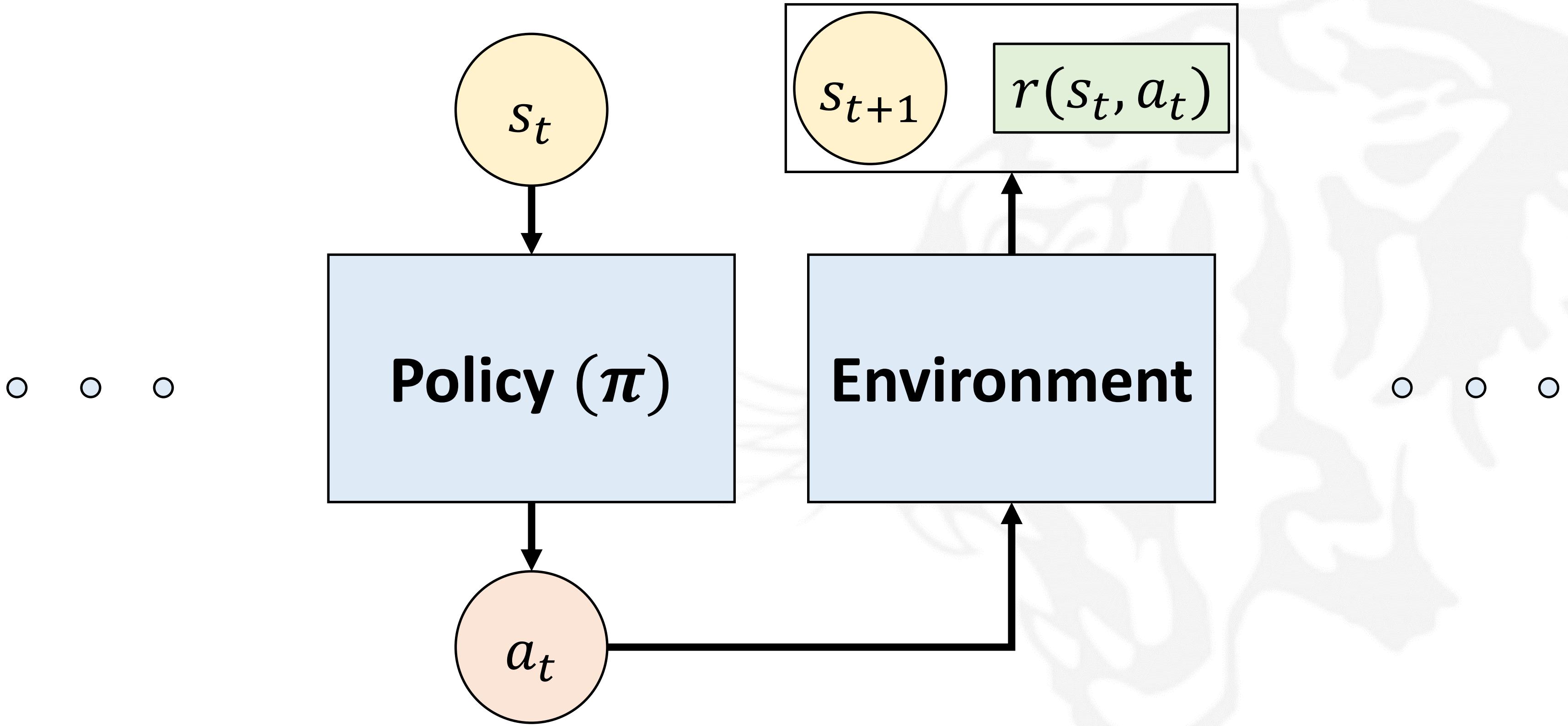
정책 기반 강화학습 이론 및 응용

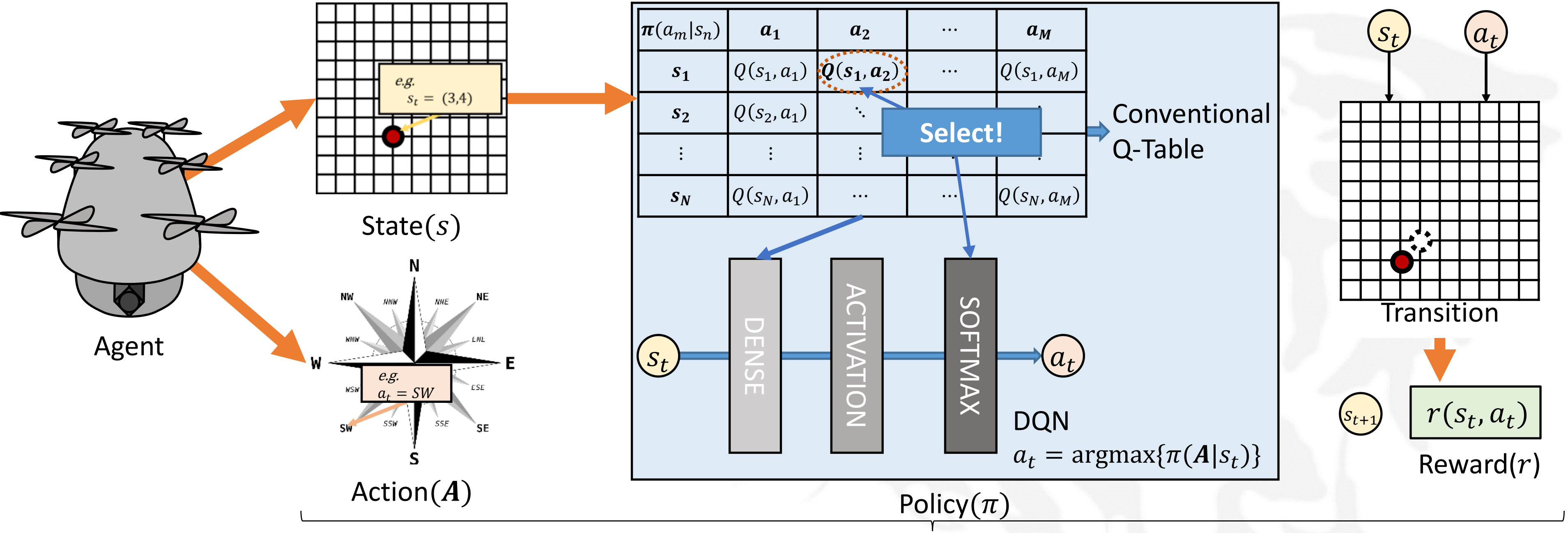
Policy-based Reinforcement Learning Theory and its Application

Won Joon Yun
Korea University, School of Electrical Engineering
Artificial Intelligence and Mobility Laboratory

Review: Reinforcement Learning Mechanism



Review: Reinforcement Learning Mechanism



Iterate until the scenario terminated!

Trajectory(Dataset): $\tau = \{s_0, a_0, r_0, s_1, a_1, \ldots, s_T\}$

Objective Function: $J(\theta) = E_\tau[\sum_{t=0}^{T} \gamma^t \cdot r(s_t, a_t)]$ ← **Maximize!**

Now, we have curiosity about…

Q1. Should we wait for the scenario terminated?

Trajectory(Dataset): $\tau = \{s_0, a_0, r_0, s_1, a_1, \ldots.., s_T\}$

Q2. How can I maximize objective function efficiently?

Objective Function: $J(\theta) = E_\tau[\sum_{t=0}^{T} \gamma^t \cdot r(s_t, a_t)]$

Q3.What about design DQN?

Q4. Any new idea?

Now, we have curiosity about…

Q1. Should we wait for the scenario terminated?
Trajectory(Dataset): $\tau = \{s_0, a_0, r_0, s_1, a_1, \ldots, s_T\}$

A1. No, I will introduce *A2C*. It will make objective function optimized FASTER.

Q2. How can I maximize objective function efficiently?
Objective Function: $J(\theta) = E_\tau[\sum_{t=0}^{T} \gamma^t \cdot r(s_t, a_t)]$

A2. I will introduce *PPO* and *DDPG*. If you use it, you can maximize the objective function with BETTER PERFORMANCE.

Q3.What about design DQN?

A3. I will introduce *CommNet* and *G2ANet*.

Q4. Any new idea?

A4. I will introduce Value Decomposition Network.

Now, we have curiosity about…

Q1. Should we wait for the scenario terminated?
Trajectory(Dataset): $\tau = \{s_0, a_0, r_0, s_1, a_1, \ldots.., s_T\}$

A1. No, I will introduce *A2C*. It will make objective function optimized FASTER.

Q2. How can I maximize objective function efficiently?
Objective Function: $J(\theta) = E_\tau[\sum_{t=0}^{T} \gamma^t \cdot r(s_t, a_t)]$

A2. I will introduce *PPO* and *DDPG*. If you use it, you can maximize the objective function with BETTER PERFORMANCE.

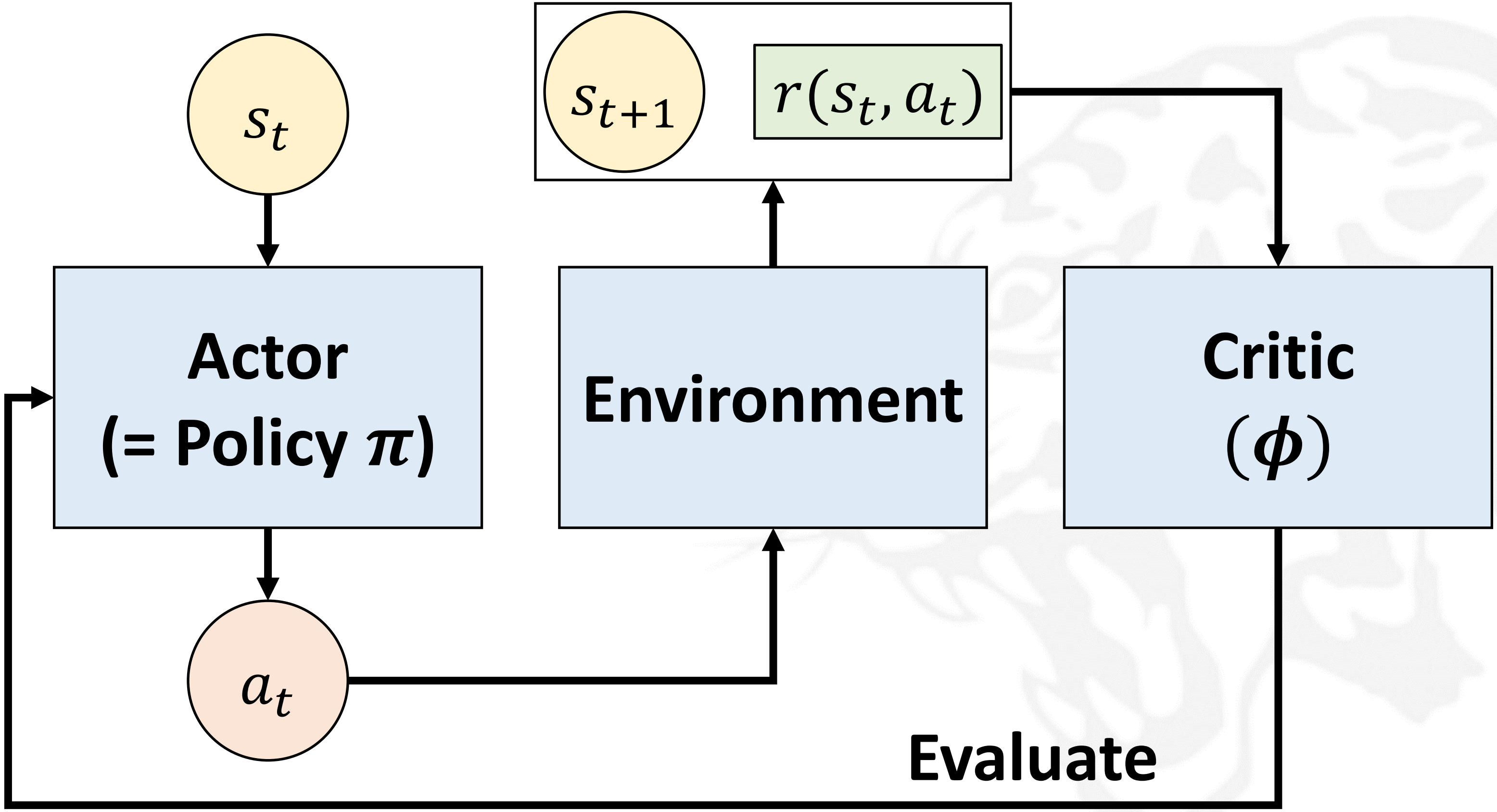
Q3.What about design DQN?

A3. I will introduce *CommNet* and *G2ANet*.

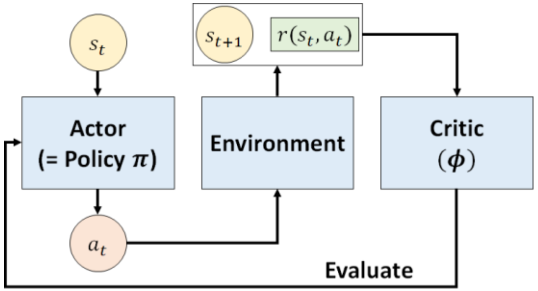
Q4. Any new idea?

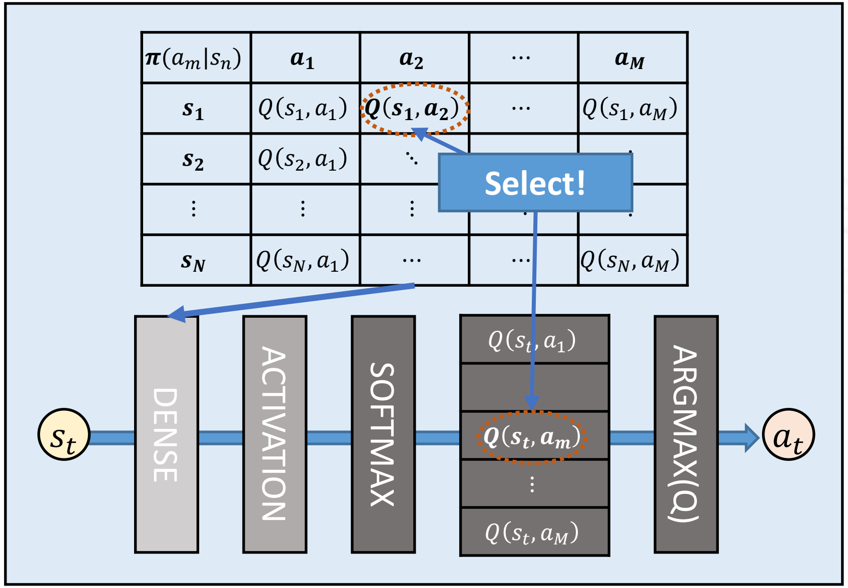
A4. I will introduce Value Decomposition Network.

A2C: Advantage Actor Critic



Preliminaries of A2C.





$\boldsymbol{\pi}(a_m \mid s_n)$	$\boldsymbol{a_1}$	$\boldsymbol{a_2}$	$\cdots$	$\boldsymbol{a_M}$
$\boldsymbol{s_1}$	$Q(s_1, a_1)$	$\boldsymbol{Q(s_1, a_2)}$	$\cdots$	$Q(s_1, a_M)$
$\boldsymbol{s_2}$	$Q(s_2, a_1)$	$\ddots$		
$\vdots$	$\vdots$	$\vdots$		
$\boldsymbol{s_N}$	$Q(s_N, a_1)$	$\cdots$	$\cdots$	$Q(s_N, a_M)$

1. Action is index of maximum Q-value

$$a_t = \text{argmax}_a\{\pi(\boldsymbol{A}|s_t)\}$$
$$= \text{argmax}_a\{Q(s_t, \boldsymbol{A})\}$$

2. $Q^\pi(s_t, a_t)$ and $\pi_\theta(s_t, a_t)$ are same!

3. Notice Q is action value function, and V is state value function. We will use advantage function.

$$A = Q(s_t, a_t) - V(s_t)$$

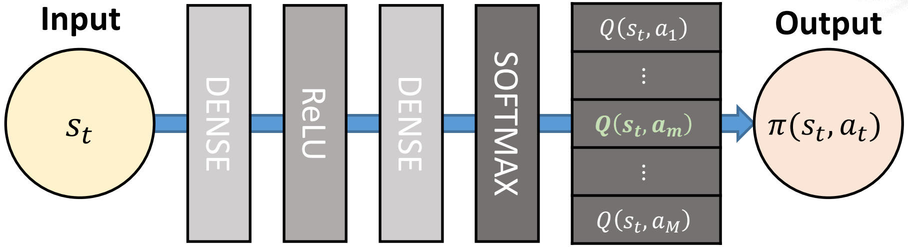
4. With Bellman equation, the present value can be estimated by the present reward and the future value .

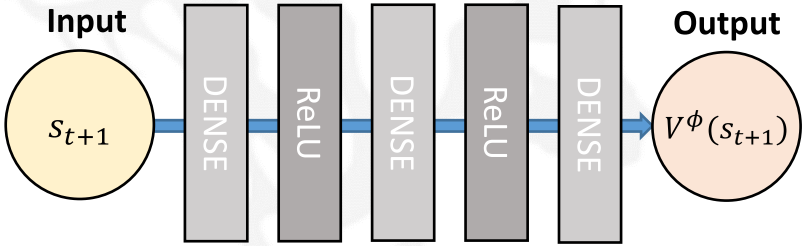
$$V^{\phi*}(s_t) = r_t + \gamma \cdot V^\phi(s_{t+1})$$

A2C: Description of A2C Architecture

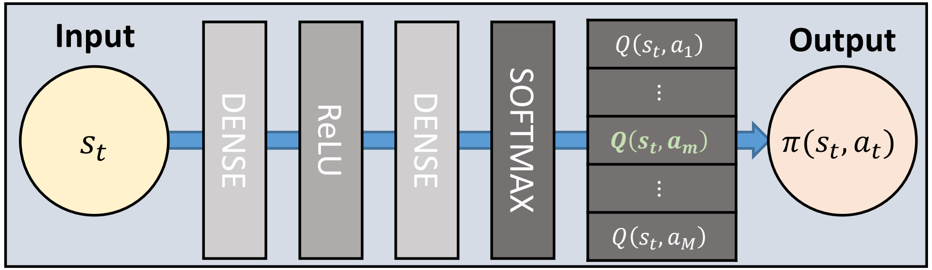
```
class Actor(nn.Module):
    def __init__(self):
        super(Actor, self).__init__()
        self.fc1 = nn.Linear(s_dim, 64)
        self.fc2 = nn.Linear(64,a_dim)
    def forward(self, s):
        s = F.ReLU(self.fc1(s))
        q = self.fc2(s)
        q = nn.Softmax(q)
        return q
```

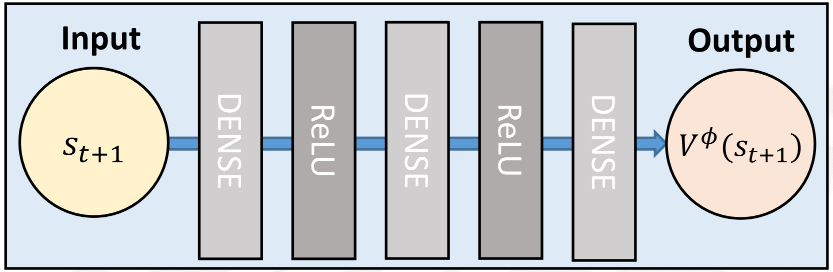
```
class Critic(nn.Module):
    def __init__(self):
        super(Critic, self).__init__()
        self.fc1 = nn.Linear(s_dim, 64)
        self.fc2 = nn.Linear(64, 8)
        self.state_value= nn.Linear(8, 1)
    def forward(self, x):
        x = F.ReLU(self.fc1(x))
        x = F.ReLU(self.fc2(x))
        value = self.state_value(x)
        return value
```





A2C: Description of A2C Process

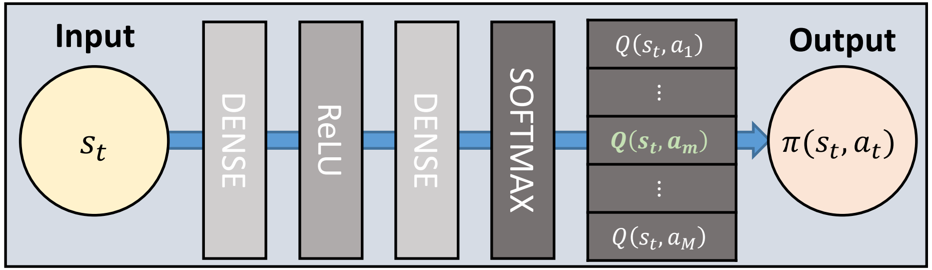


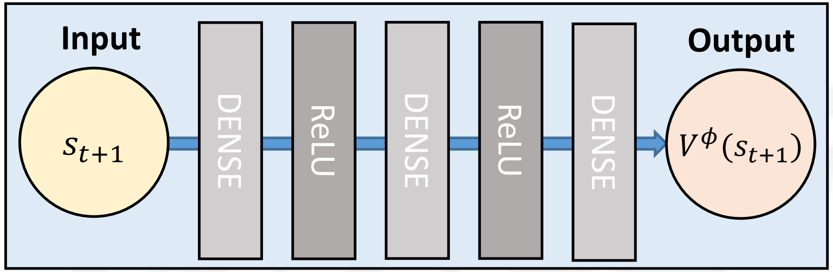



```
q, v       = Actor(state), Critic(state)
action     = q.sample()
next_state, reward = envs.step(action)
log_prob   = q.log_prob(action)
v_next     = Critic(next_state)
target     =  reward + gamma * v_next
advantage =  target - v
A_loss = -(log_probs*advantage)
C_loss = advantage
```

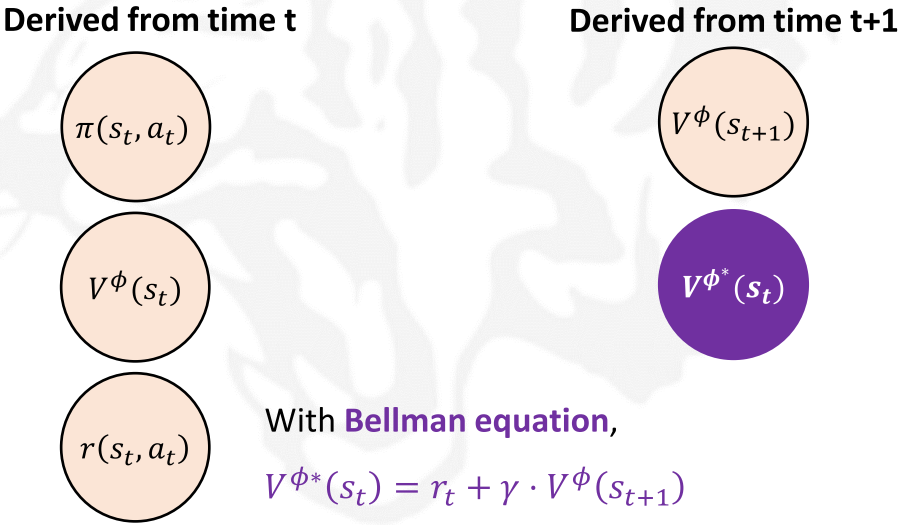
Derived from time t **Derived from time t+1**

A2C: Description of A2C Process



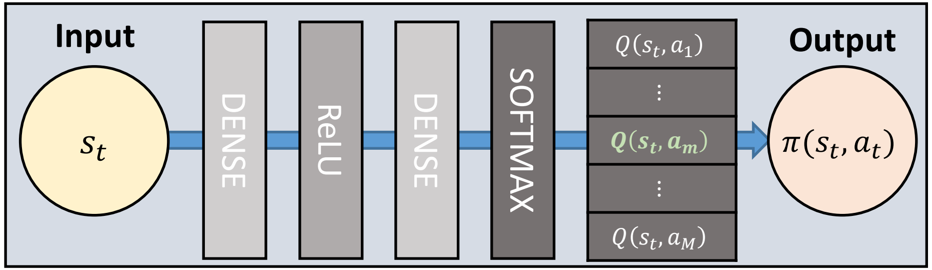


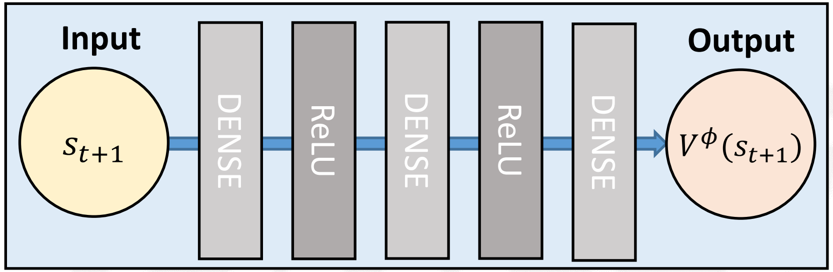

```
q, v        = Actor(state), Critic(state)
action      = q.sample()
next_state, reward = envs.step(action)
log_prob    = q.log_prob(action)
v_next      = Critic(next_state)
target      =  reward + gamma * v_next
advantage =  target - v
A_loss = -(log_probs*advantage)
C_loss = advantage
```



With **Bellman equation**,

$$V^{\phi*}(s_t) = r_t + \gamma \cdot V^{\phi}(s_{t+1})$$






```
q, v      = Actor(state), Critic(state)
action    = q.sample()
next_state, reward = envs.step(action)
log_prob  = q.log_prob(action)
v_next    = Critic(next_state)
target    =  reward + gamma * v_next
advantage =  target - v
A_loss = -(log_probs*advantage)
C_loss = advantage
```

Derived from time t

$\pi(s_t, a_t)$

$V^{\phi}(s_t)$

$r(s_t, a_t)$

Advantage Function(A^{ϕ})

$A^{\phi} = Q^{\phi}(s_t, a_t) - V^{\phi}(s_t)$

With action value formula,

$Q(s_t) = r_t + \gamma \cdot V(s_{t+1})$

Advantage Function(A^{ϕ})

$\boldsymbol{A^{\phi}(s_t, a_t)}$

$\boldsymbol{= r_t + \gamma \cdot V^{\phi}(s_{t+1}) - V^{\phi}(s_t)}$

Derived from time t+1

$V^{\phi}(s_{t+1})$

$V^{\phi^*}(s_t)$

$A^{\phi}(s_t, a_t)$

Now, we have curiosity about…

Q1. Should we wait for the scenario terminated?

Trajectory(Dataset): $\tau = \{s_1, a_1, r_1, s_2, a_2, \ldots, s_T\}$

A1. **No, I will introduce *A2C*. It will make objective function optimized FASTER.**

Q2. How can I maximize objective function efficiently?

Objective Function: $J(\theta) = E_\tau[\sum_{t=0}^{T} \gamma^t \cdot r(s_t, a_t)]$

A2. **I will introduce *PPO* and *DDPG*. If you use it, you can maximize the objective function with BETTER PERFORMANCE.**

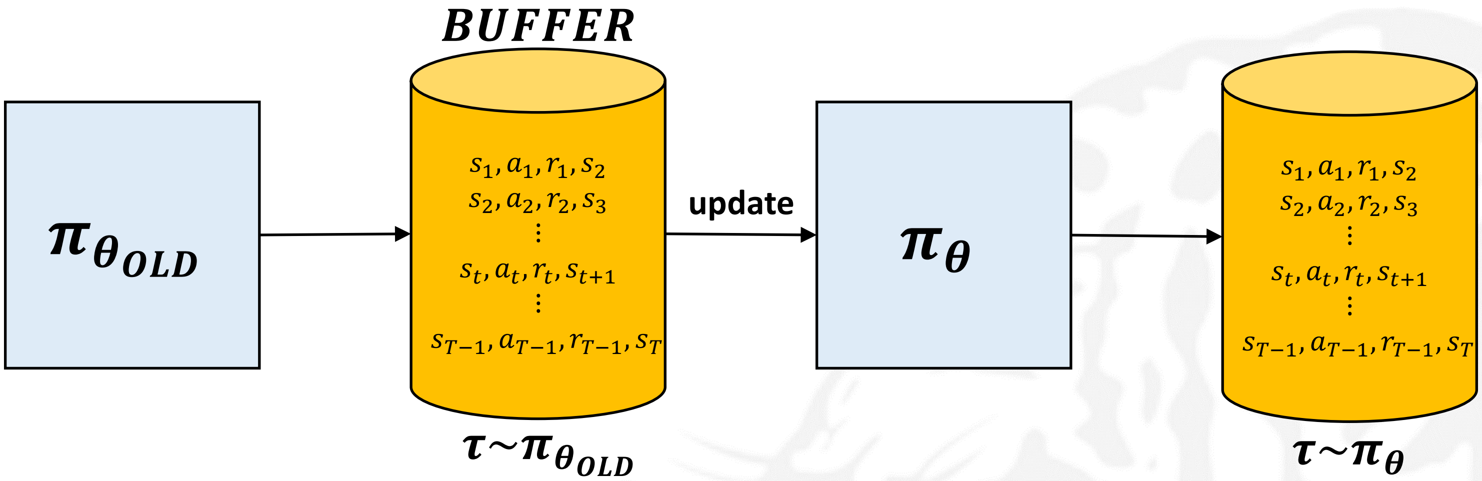
Q3.What about design DQN?

A3. **I will introduce *CommNet* and *G2ANet*.**

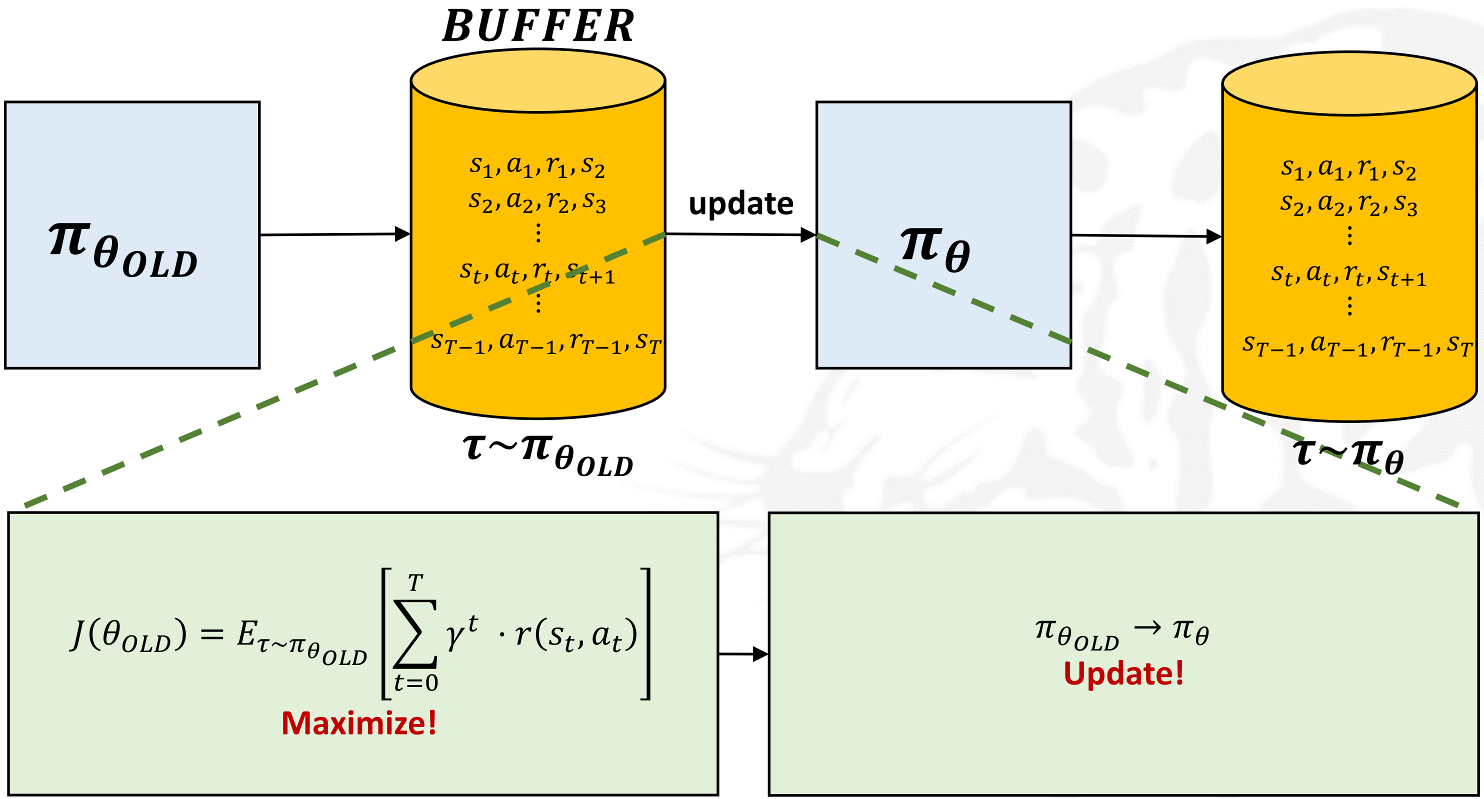
Q4. Any new idea?

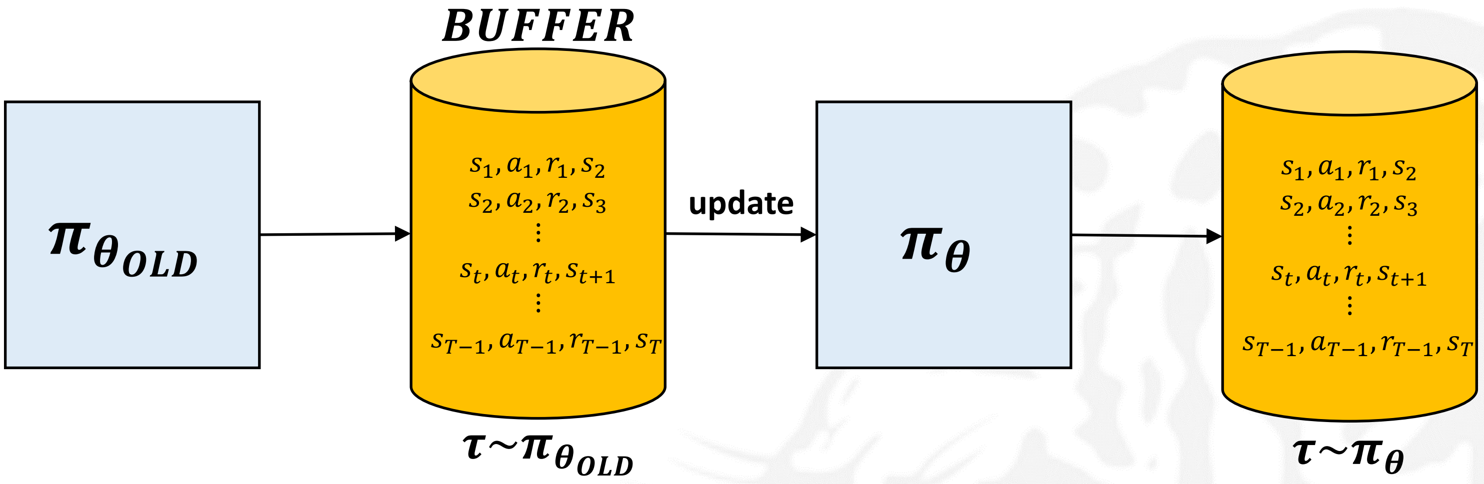
A4. **I will introduce Value Decomposition Network.**

PPO: Proximal Policy Optimization



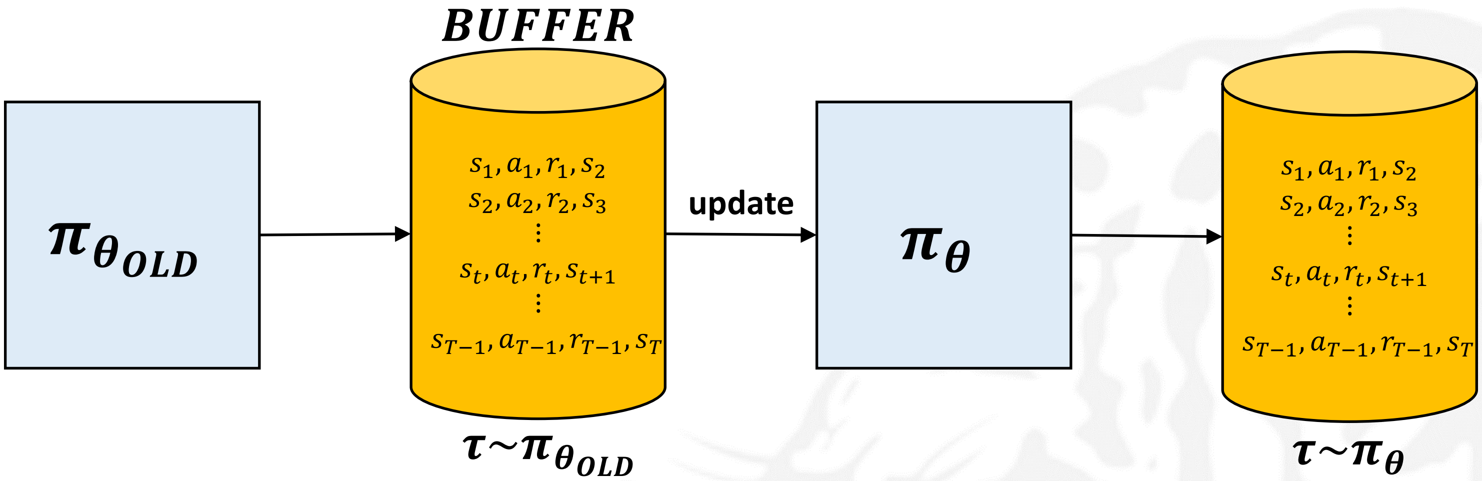
PPO: Proximal Policy Optimization





What if there is huge difference between $\pi_{\theta_{OLD}}$ and π_θ?

PPO: Proximal Policy Optimization



Does not guarantee as follows:

→ Stability of Policy

→ Reward Convergence

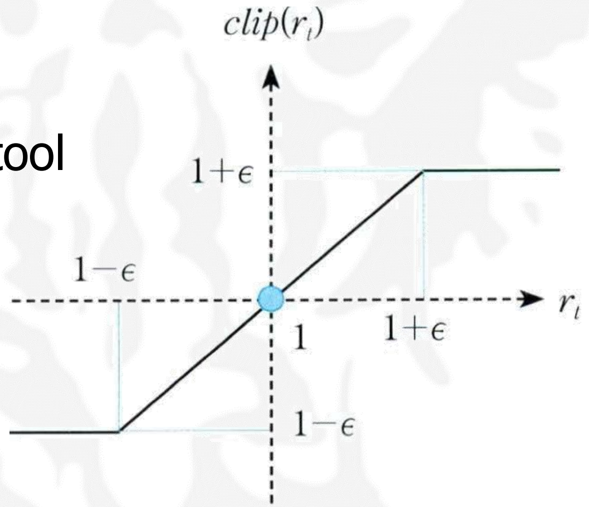
PPO: Overview of PPO

- Two objectives:
  - **Minimize** $\nabla_\theta \pi_\theta(s_t, a_t)$ → New objectives.
  - **Maximize** $J(\theta)$ → **Maximize** $L(\theta)$

- How? → Surrogate function($L(\theta)$)
  - Let's assume $\pi_\theta \cong \pi_{\theta_{OLD}}$ and $J(\theta) - J(\theta_{OLD}) = L(\theta) > 0$ where,
  - $L(\theta) = \sum_{t=0}^{\infty} E_{\tau_{x_0:u_t} \sim p_{\theta_{OLD}}(\tau_{x_0:u_t})} \left[\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{OLD}}(a_t|s_t)} \gamma^t A^{\pi_{\theta_{OLD}}}(s_t, a_t) \right]$

- Mathematical Tools for "$\pi_\theta \cong \pi_{\theta_{OLD}}$"
  - **Clip function**

- Let's define the ratio of π_θ and $\pi_{\theta_{OLD}}$ as follows:
- $r_t(\theta) = \dfrac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{OLD}}(a_t|s_t)}$
- Then let's define clip function which is the mathematical tool for "$\pi_\theta \cong \pi_{\theta_{OLD}}$" as follows:
- $clip(r_t(\theta), 1-\epsilon, 1+\epsilon) = \begin{cases} 1+\epsilon, & if\ r_t(\theta) \geq 1+\epsilon \\ 1-\epsilon, & if\ r_t(\theta) \leq 1-\epsilon \\ r_t(\theta), & otherwise. \end{cases}$



where ϵ is small number *e.g.* $\epsilon = 0.01$.

PPO: Clip function

1. **If** $\pi_\theta(a_t|s_t) \gg \pi_{\theta_{OLD}}(a_t|s_t)$ **:**

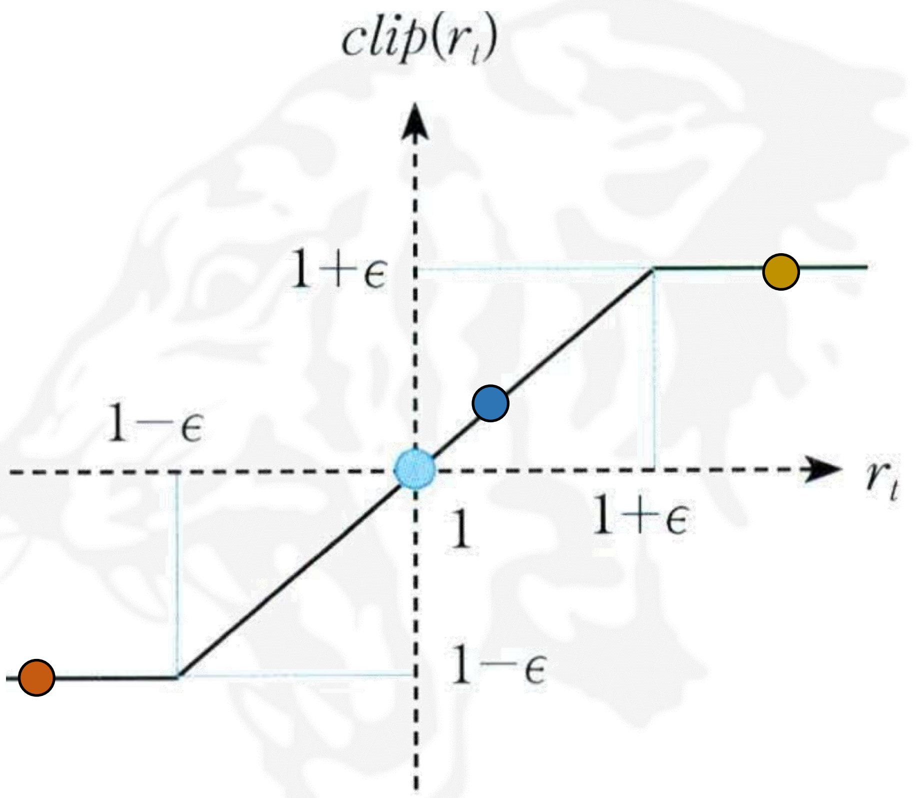
   $\rightarrow r_t(\theta) = 1 + \epsilon$

2. **Else if** $\pi_\theta(a_t|s_t) \ll \pi_{\theta_{OLD}}(a_t|s_t)$ **:**

   $\rightarrow r_t(\theta) = 1 - \epsilon$

3. **Else** $\pi_\theta(a_t|s_t) \cong \pi_{\theta_{OLD}}(a_t|s_t)$ **:**

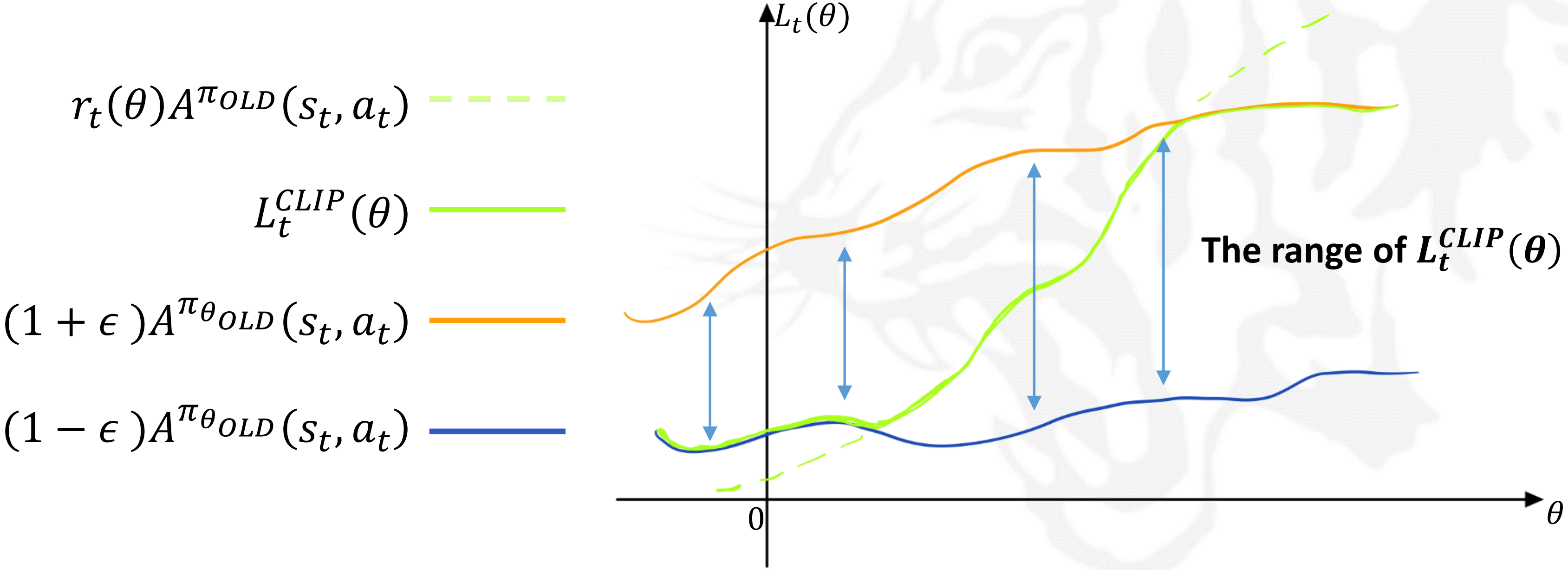
   $\rightarrow r_t(\theta) = r_t(\theta)$



PPO: Surrogate function

- Now the objective function can be expressed as follow:

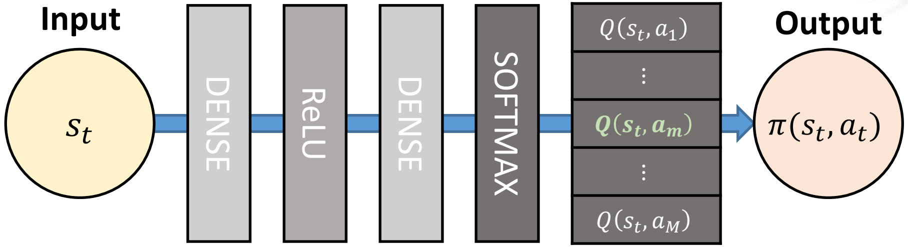
$$L_t^{CLIP}(\theta) = \min\{r_t(\theta)A^{\pi_{OLD}}(s_t, a_t), clip(r_t(\theta), 1-\epsilon, 1+\epsilon)A^{\pi_{\theta_{OLD}}}(s_t, a_t)\}$$

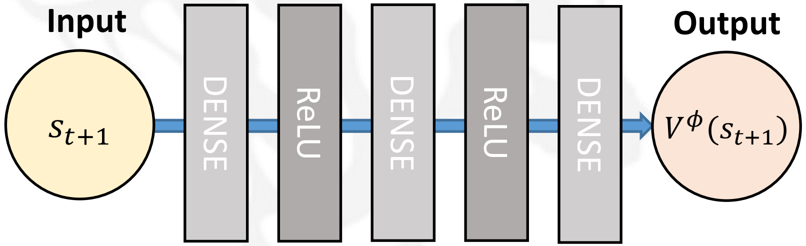


PPO: Actor, Critic Network Codes

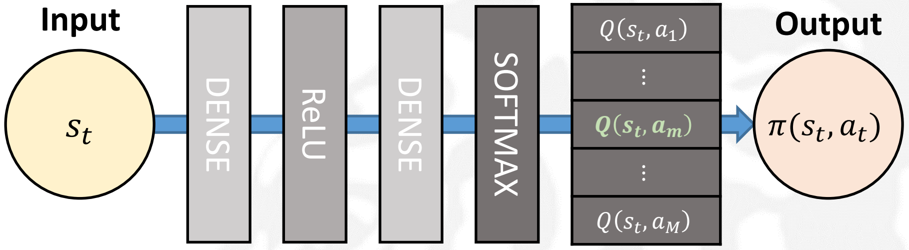
```
class Actor(nn.Module):
    def __init__(self):
        super(Actor, self).__init__()
        self.fc1 = nn.Linear(s_dim, 64)
        self.fc2 = nn.Linear(64,a_dim)
    def forward(self, s):
        s = F.ReLU(self.fc1(s))
        q = self.fc2(s)
        q = nn.Softmax(q)
        return q
```

```
class Critic(nn.Module):
    def __init__(self):
        super(Critic, self).__init__()
        self.fc1 = nn.Linear(s_dim, 64)
        self.fc2 = nn.Linear(64, 8)
        self.state_value= nn.Linear(8, 1)
    def forward(self, x):
        x = F.ReLU(self.fc1(x))
        x = F.ReLU(self.fc2(x))
        value = self.state_value(x)
        return value
```



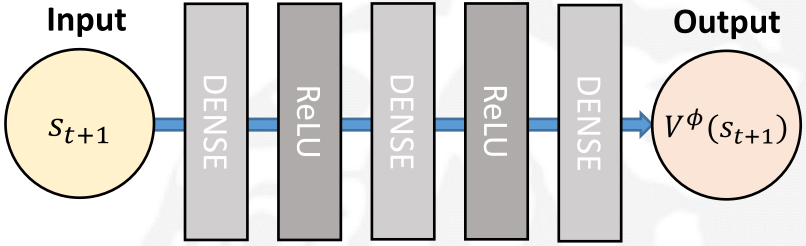


```
class PPO():
    clip_param = 0.2
    def __init__(self):
        super(PPO, self).__init__()
        self.actor_net = Actor()
        self.critic_net = Critic()
        self.buffer = []
        self.actor_optimizer = optim.Adam(self.actor_net.parameters(), 1e-3)
        self.critic_net_optimizer = optim.Adam(self.critic_net.parameters(), 3e-3)
        self.counter = 0
    def store_transition(self, transition):
        self.buffer.append(transition)
        self.counter += 1
    def get_value(self, state):
        state = torch.from_numpy(state)
        value = self.critic_net(state)
        return value.item()
```

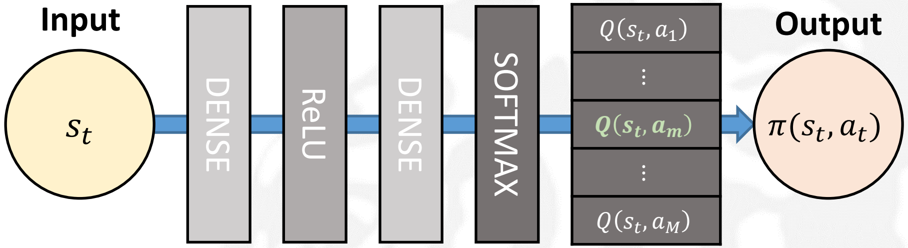


PPO: PPO Codes

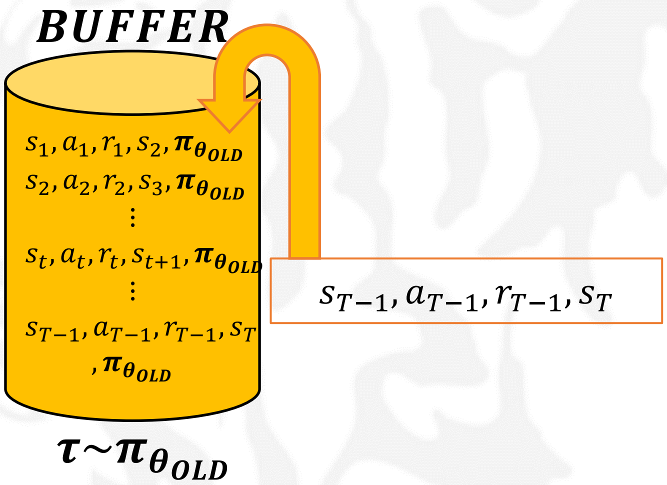
```
class PPO():
    clip_param = 0.2
    def __init__(self):
        super(PPO, self).__init__()
        self.actor_net = Actor()
        self.critic_net = Critic()
        self.buffer = []
        self.actor_optimizer = optim.Adam(self.actor_net.parameters(), 1e-3)
        self.critic_net_optimizer = optim.Adam(self.critic_net.parameters(), 3e-3)
        self.counter = 0
    def store_transition(self, transition):
        self.buffer.append(transition)
        self.counter += 1
    def get_value(self, state):
        state = torch.from_numpy(state)
        value = self.critic_net(state)
        return value.item()
```



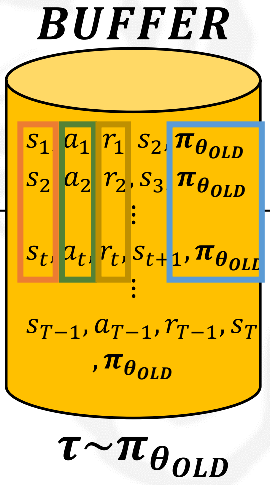
```python
class PPO():
    clip_param = 0.2
    def __init__(self):
        super(PPO, self).__init__()
        self.actor_net = Actor()
        self.critic_net = Critic()
        self.buffer = []
        self.actor_optimizer = optim.Adam(self.actor_net.parameters(), 1e-3)
        self.critic_net_optimizer = optim.Adam(self.critic_net.parameters(), 3e-3)
        self.counter = 0
    def store_transition(self, transition):
        self.buffer.append(transition)
        self.counter += 1
    def get_value(self, state):
        state = torch.from_numpy(state)
        value = self.critic_net(state)
        return value.item()
```



```python
class PPO():
    clip_param = 0.2
    def __init__(self):
        super(PPO, self).__init__()
        self.actor_net = Actor()
        self.critic_net = Critic()
        self.buffer = []
        self.actor_optimizer = …
        self.critic_net_optimizer = …
        self.counter = 0
    def store_transition(self, transition):
        self.buffer.append(transition)
        self.counter += 1
    def get_value(self, state):
        state = torch.from_numpy(state)
        value = self.critic_net(state)
        return value.item()
```



```python
class PPO():
    def update(self, i_ep):
        state = torch.tensor([t.state for t in self.buffer])
        action = torch.tensor([t.action for t in self.buffer])
        reward = [t.reward for t in self.buffer]
        old_action_log_prob = torch.tensor([t.a_log_prob for t in self.buffer])
        R = 0
        Gt = []
        for r in reward[::-1]:
            R = r + gamma * R
            Gt.insert(0, R)
        Gt = torch.tensor(Gt, dtype=torch.float)
```



```
class PPO():
    def update(self, i_ep):
        for i in range(self.ppo_update_time):
            for index in BatchSampler(XXXX):
                action_loss = -torch.min(surr1, surr2).mean()
                self.actor_optimizer.zero_grad()
                action_loss.backward()
                nn.utils.clip_grad_norm_(self.actor_net.parameters(), self.max_grad_norm)
                self.actor_optimizer.step()
                value_loss = F.mse_loss(Gt_index, V)
                self.critic_net_optimizer.zero_grad()
                value_loss.backward()
                nn.utils.clip_grad_norm_(self.critic_net.parameters(), self.max_grad_norm)
                self.critic_net_optimizer.step()
```

```
XXXX = SubsetRandomSampler(range(len(self.buffer))), self.batch_size, False
```

Now, we have curiosity about…

Q1. Should we wait for the scenario terminated?

Trajectory(Dataset): $\tau = \{s_1, a_1, r_1, s_2, a_2, \ldots., s_T\}$

A1. No, I will introduce *A2C*. It will make objective function optimized FASTER.

Q2. How can I maximize objective function efficiently?

Objective Function: $J(\theta) = E_\tau[\sum_{t=0}^{T} \gamma^t \cdot r(s_t, a_t)]$

A2. I will introduce *PPO* and *DDPG*. If you use it, you can maximize the objective function with BETTER PERFORMANCE.

Q3.What about design DQN?

A3. I will introduce *CommNet* and *G2ANet*.

Q4. Any new idea?

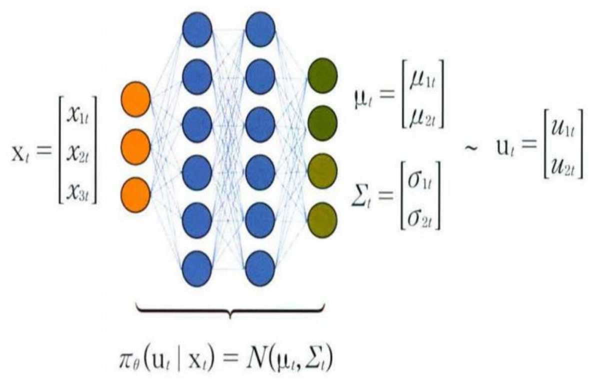
A4. I will introduce Value Decomposition Network.

Previous Algorithm.

- Conventional Q-Learning
- DQN, PPO

→ Stochastic Policy

→ Discrete Action

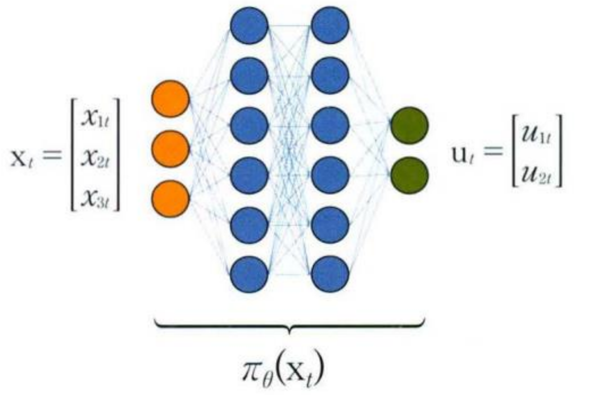


What we will learn.

- DDPG is…

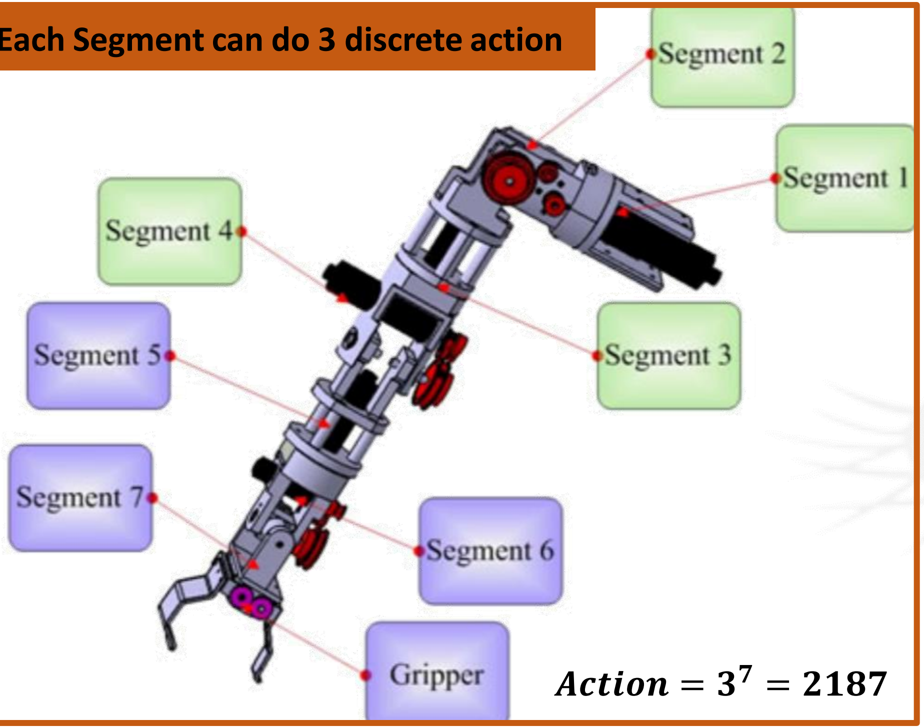
→ Deterministic Policy

→ Continuous Action



DDPG: Let's assume there is…

Each Segment can do 3 discrete action

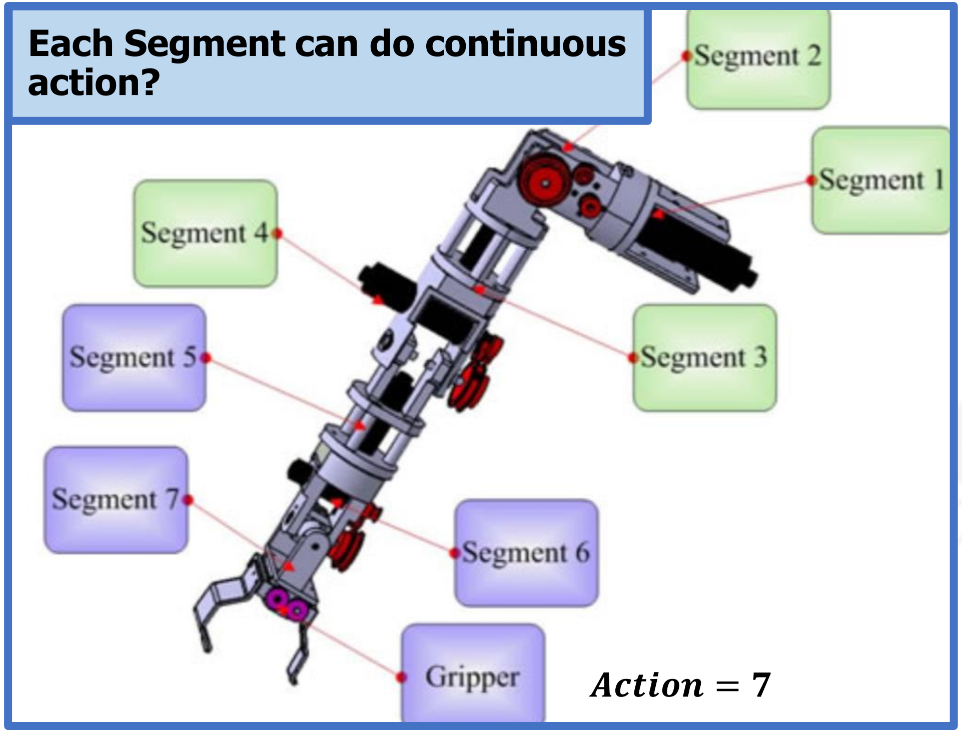


$$Action = 3^7 = 2187$$

The problem is…

1. Large Action Space
2. Loss by discretization
3. No sophistic action

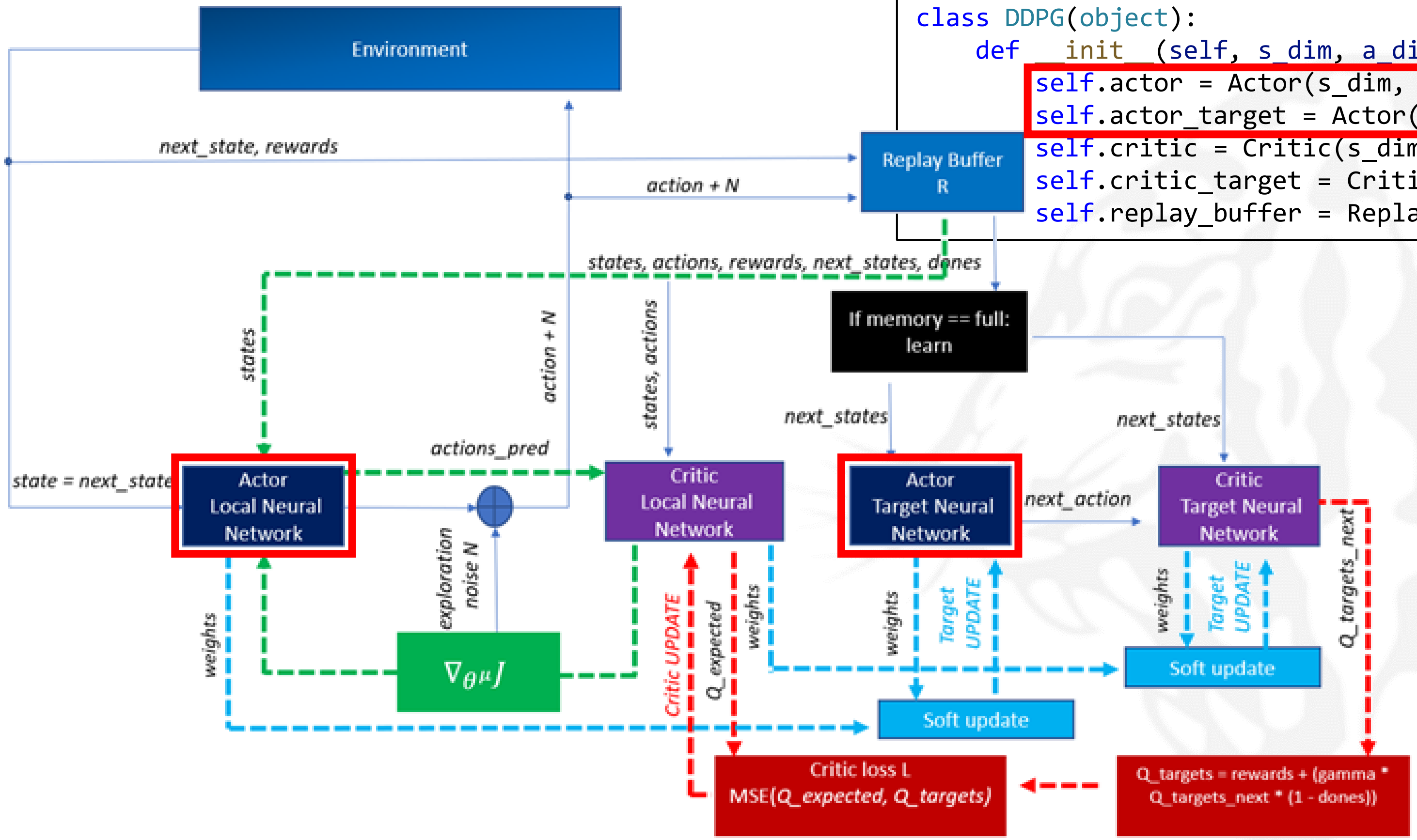
Each Segment can do continuous action?



$$\boldsymbol{Action = 7}$$

1. Small Action Space
2. Sophistic action

DDPG: New Architecture

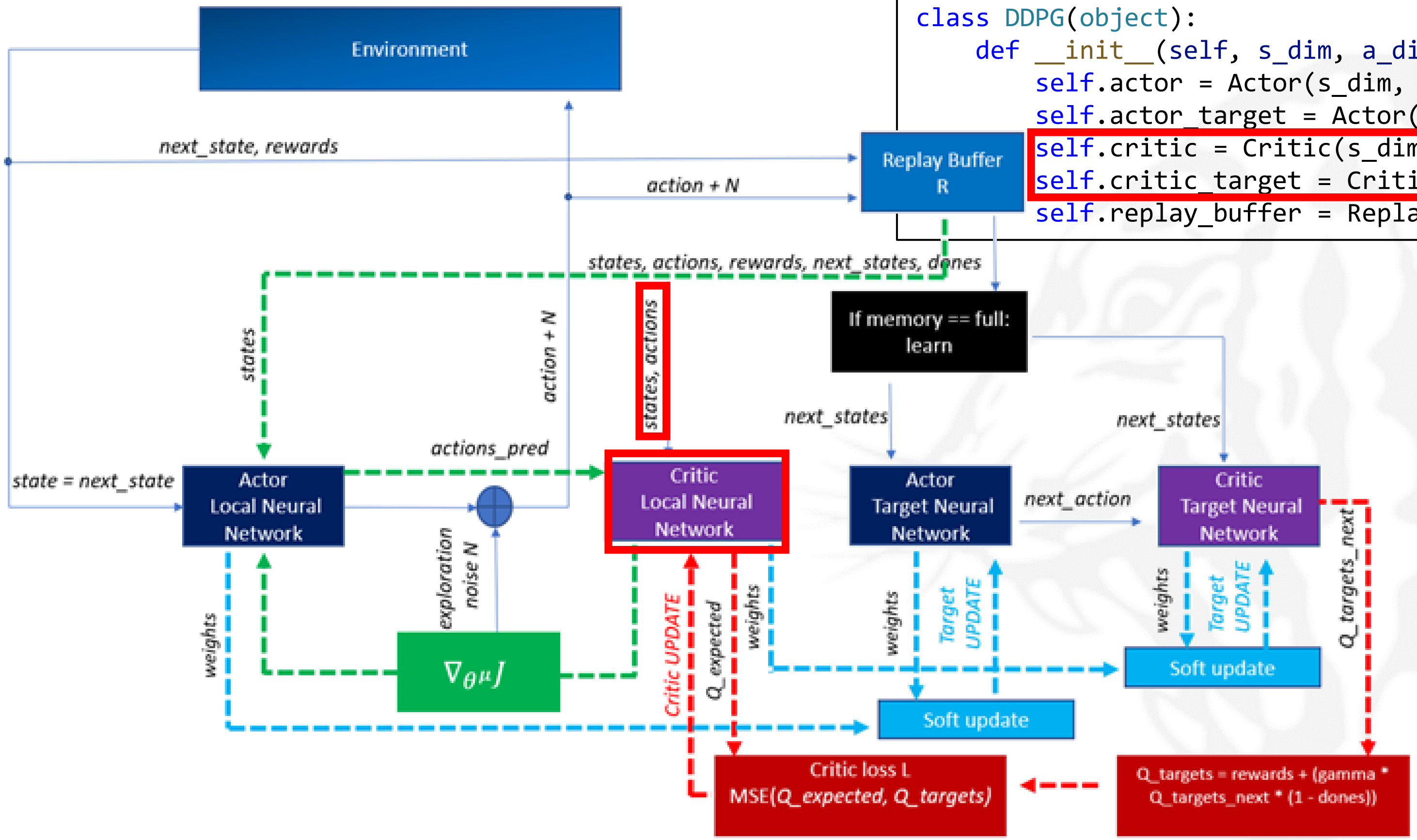



```python
class DDPG(object):
    def __init__(self, s_dim, a_dim, max_action):
        self.actor = Actor(s_dim, a_dim, max_action)
        self.actor_target = Actor(s_dim, a_dim, max_action)
        self.critic = Critic(s_dim, a_dim)
        self.critic_target = Critic(s_dim, a_dim)
        self.replay_buffer = Replay_buffer()
```

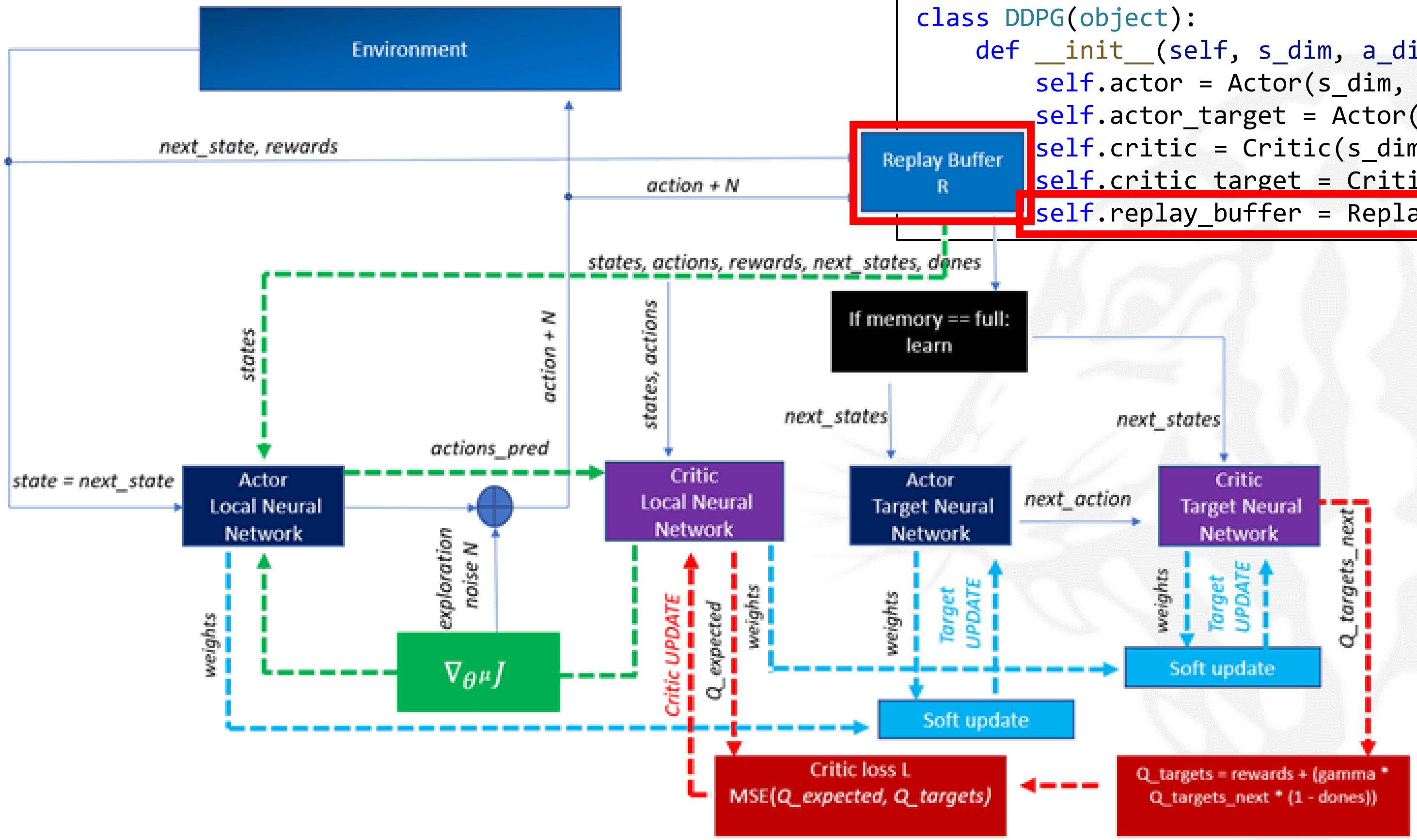
DDPG: New Architecture

Artificial Intelligence and Mobility Lab

```python
class DDPG(object):
    def __init__(self, s_dim, a_dim, max_action):
        self.actor = Actor(s_dim, a_dim, max_action)
        self.actor_target = Actor(s_dim, a_dim, max_action)
        self.critic = Critic(s_dim, a_dim)
        self.critic_target = Critic(s_dim, a_dim)
        self.replay_buffer = Replay_buffer()
```

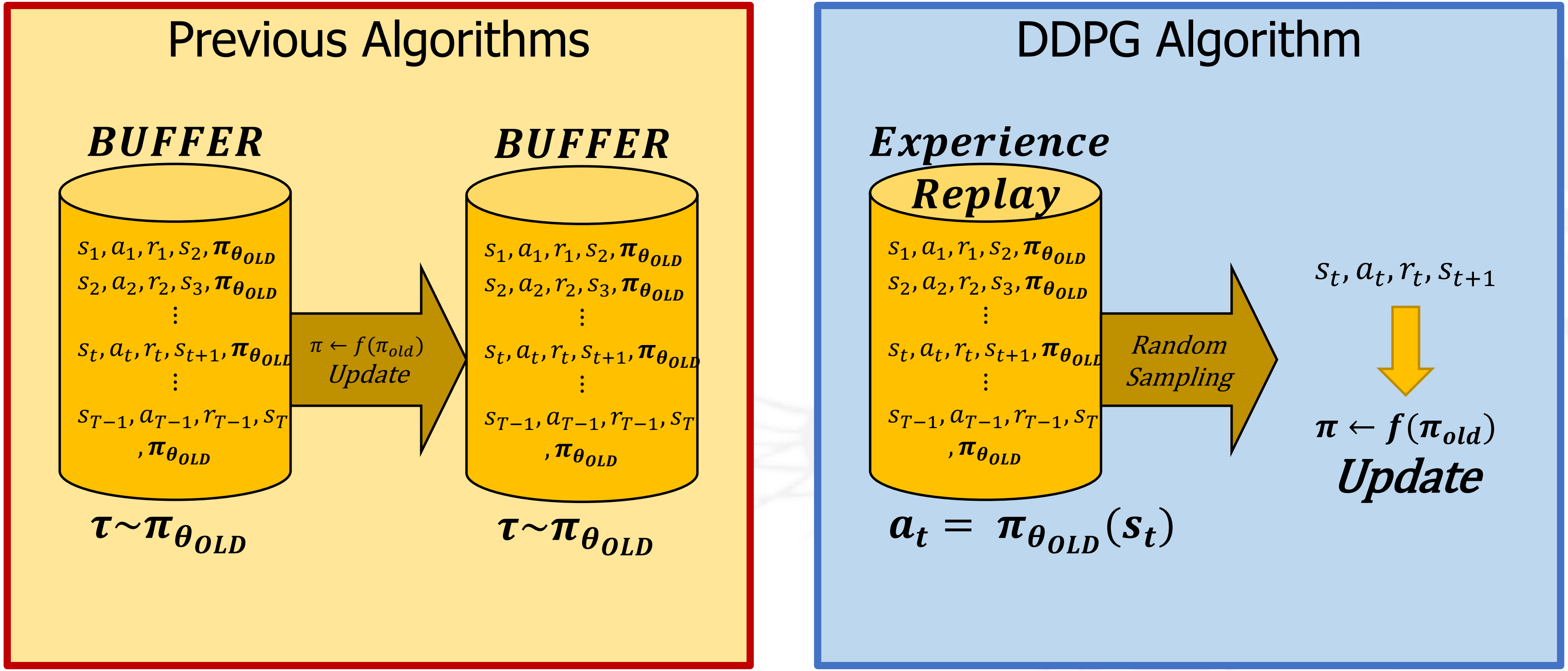


DDPG: New Architecture



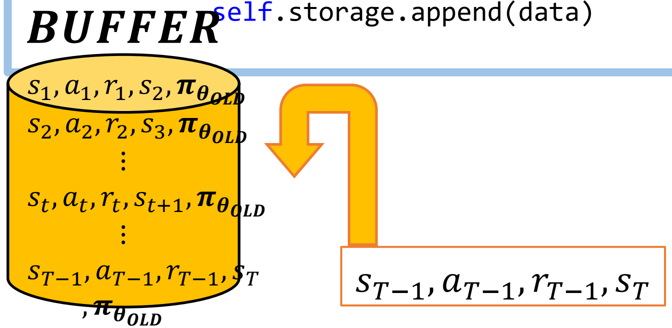

```python
class DDPG(object):
    def __init__(self, s_dim, a_dim, max_action):
        self.actor = Actor(s_dim, a_dim, max_action)
        self.actor_target = Actor(s_dim, a_dim, max_action)
        self.critic = Critic(s_dim, a_dim)
        self.critic_target = Critic(s_dim, a_dim)
        self.replay_buffer = Replay_buffer()
```

DDPG: Experience Replay

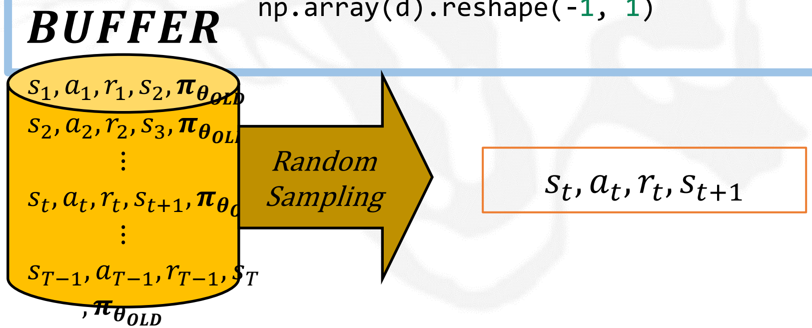


DDPG: Experience Replay

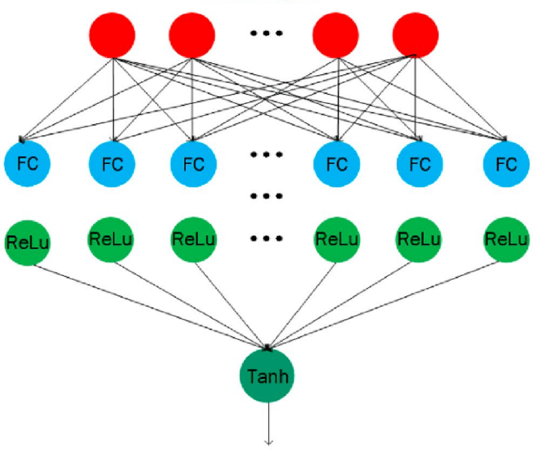
```python
class Replay_buffer():
    def __init__(self, max_size=args.capacity):
        self.storage = []
        self.max_size = max_size
        self.ptr = 0

    def push(self, data):
        if len(self.storage) == self.max_size:
            self.storage[int(self.ptr)] = data
            self.ptr = (self.ptr + 1) % self.max_size
        else:
            self.storage.append(data)
```




```python
    def sample(self, batch_size):
        ind = np.random.randint(0, len(self.storage), size=batch_size)
        s, s_next, a, r, d = [], [], [], [], []

        for i in ind:
            S, S_, A, R, D = self.storage[i]
            s.append(np.array(X, copy=False))
            s_next.append(np.array(Y, copy=False))
            a.append(np.array(U, copy=False))
            r.append(np.array(R, copy=False))
            d.append(np.array(D, copy=False))

        return np.array(x), np.array(y), np.array(u), np.array(r).reshape(-1, 1), np.array(d).reshape(-1, 1)
```



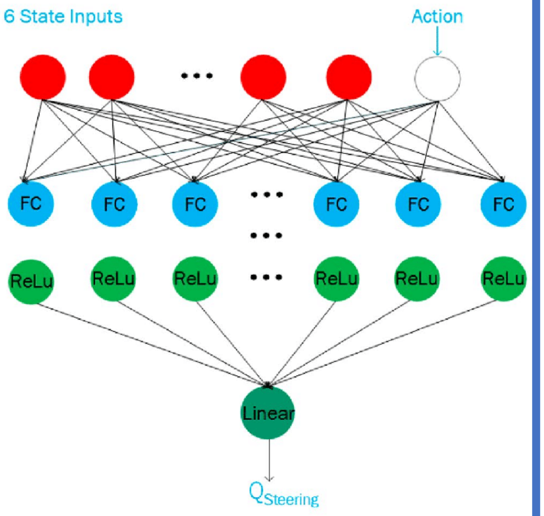
DDPG: Actor



(a) Actor Network

```python
class Actor(nn.Module):
    def __init__(self, state_dim, action_dim, max_action):
        super(Actor, self).__init__()

        self.l1 = nn.Linear(state_dim, 400)
        self.l2 = nn.Linear(400, 300)
        self.l3 = nn.Linear(300, action_dim)

        self.max_action = max_action

    def forward(self, x):
        x = F.relu(self.l1(x))
        x = F.relu(self.l2(x))
        x = self.max_action * torch.tanh(self.l3(x))
        return x
```

DDPG: Critic



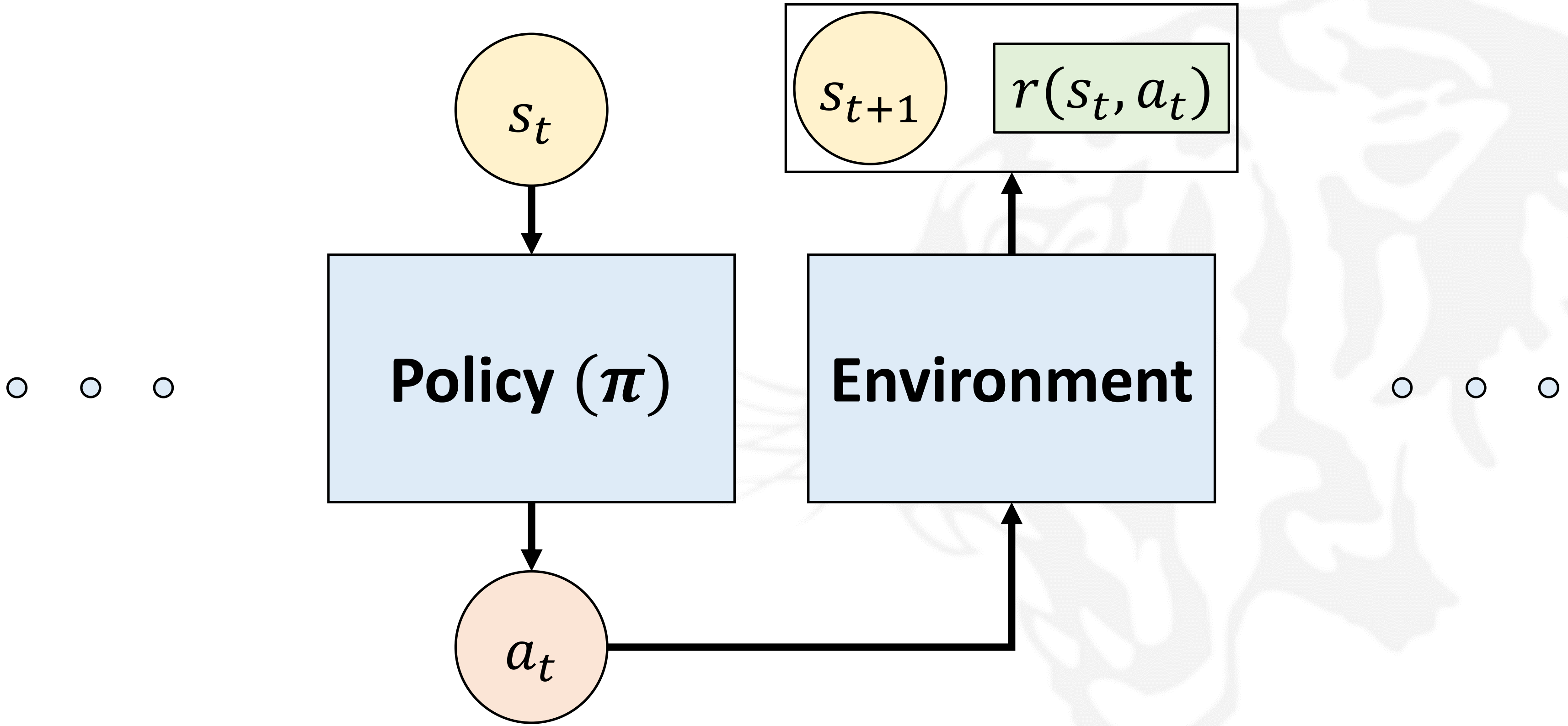
(b) Critic Network

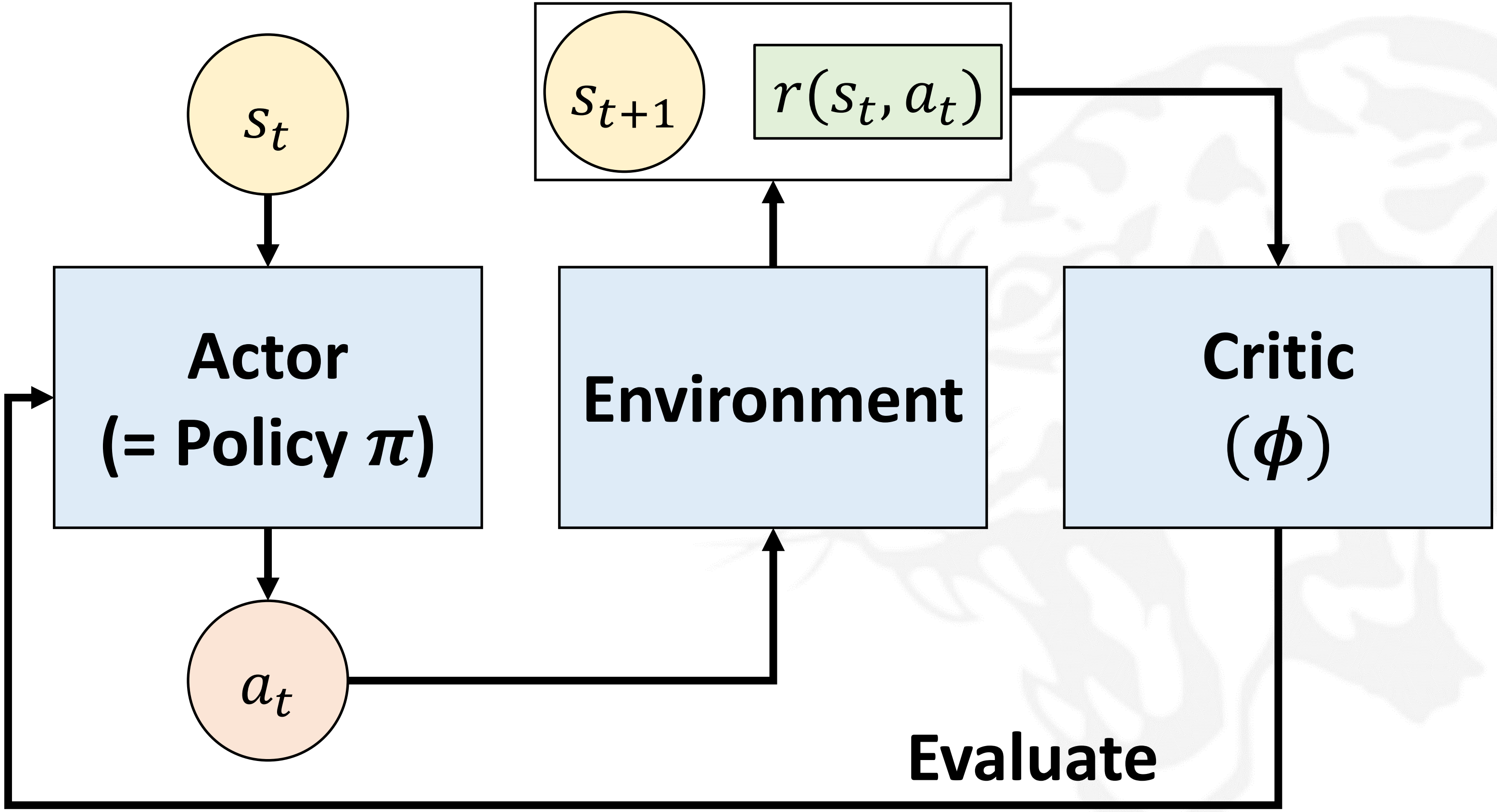
```
class Critic(nn.Module):
    def __init__(self, state_dim, action_dim):
        super(Critic, self).__init__()

        self.l1 = nn.Linear(state_dim + action_dim, 400)
        self.l2 = nn.Linear(400 , 300)
        self.l3 = nn.Linear(300, 1)

    def forward(self, x, u):
        x = F.relu(self.l1(torch.cat([x, u], 1)))
        x = F.relu(self.l2(x))
        x = self.l3(x)
        return x
```

DDPG: Update

```
def update(self):
    for it in range(args.update_iteration):
        # Sample replay buffer
        s, s_next, a, r, d = self.replay_buffer.sample(args.batch_size)
        a = (a + np.random.normal(0, noise, size=a_dim)).clip(a_min, a_max)
        target_Q = self.critic_target(s_next, self.actor_target(s_next))
        target_Q = r + ( (1-done) * args.gamma * target_Q)
        current_Q = self.critic(s, a)
        critic_loss = F.mse_loss(current_Q, target_Q)
        actor_loss = - self.critic(s, self.actor(s)).mean()

        # Update the frozen target models
        for param, target_param in zip(self.critic.parameters(), self.critic_target.parameters()):
            target_param.data.copy_(args.tau * param.data + (1 - args.tau) * target_param.data)

        for param, target_param in zip(self.actor.parameters(), self.actor_target.parameters()):
            target_param.data.copy_(args.tau * param.data + (1 - args.tau) * target_param.data)
```

Conventional Reinforcement Learning Method

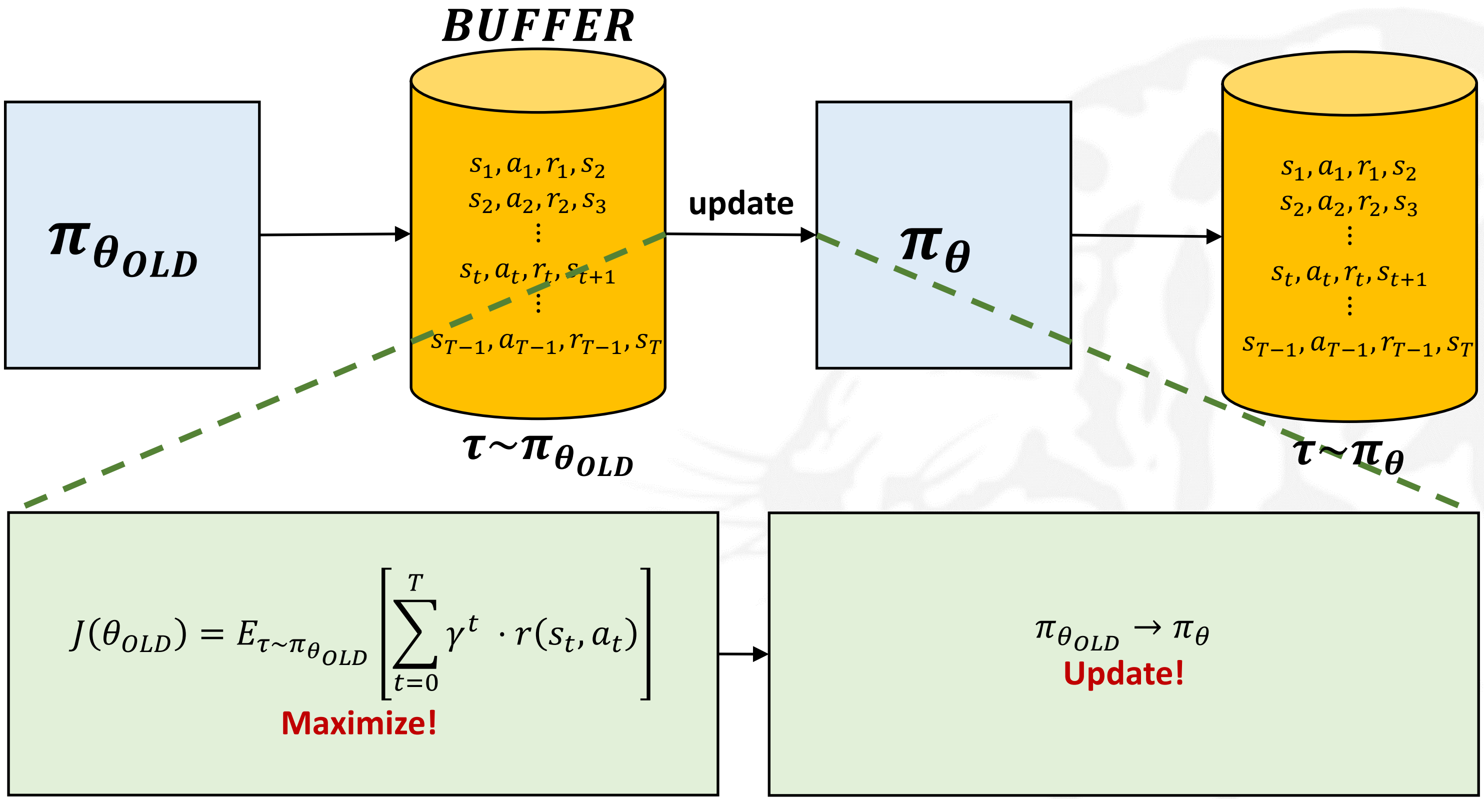




s_t
s_{t+1}
$r(s_t, a_t)$
Actor
(= Policy π)
Environment
Critic
(ϕ)
a_t
Evaluate

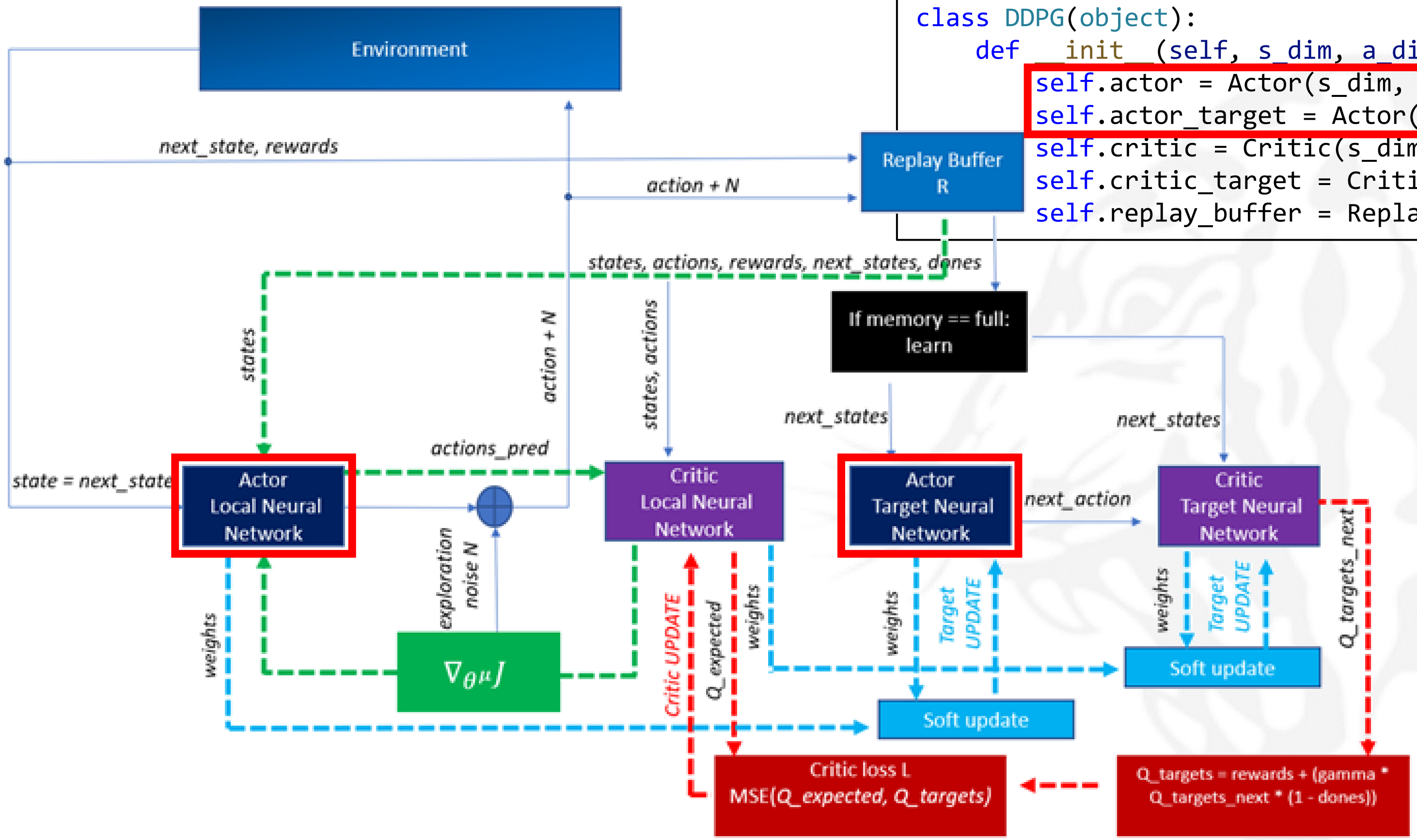
PPO: Proximal Policy Optimization

Artificial Intelligence and Mobility Lab



DDPG: Deep Deterministic Policy Gradient


```python
class DDPG(object):
    def __init__(self, s_dim, a_dim, max_action):
        self.actor = Actor(s_dim, a_dim, max_action)
        self.actor_target = Actor(s_dim, a_dim, max_action)
        self.critic = Critic(s_dim, a_dim)
        self.critic_target = Critic(s_dim, a_dim)
        self.replay_buffer = Replay_buffer()
```



Thank you for your attention!

- More questions?
  - ywjoon95@korea.ac.kr