



CORE

Control + Optimization Research Lab

Deep Deterministic Policy Gradient (DDPG)

이동민

삼성전자 서울대 공동연구소

Jul 17, 2019

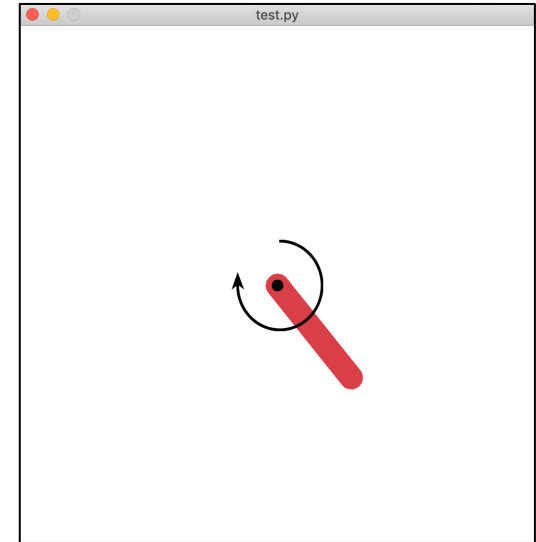
Outline

- Environment : Pendulum
- Deep Deterministic Policy Gradient (DDPG)
 - Learning process
 - Hyperparameter
 - Main loop
 - Train model
 - Train & TensorboardX
 - Learning curve & Test
 - DDPG video

Pendulum

- Env name : Pendulum-v0
- States : Continuous observation spaces

| Num | Observation | Min | Max |
|-----|----------------|------|-----|
| 0 | $\cos(\theta)$ | -1.0 | 1.0 |
| 1 | $\sin(\theta)$ | -1.0 | 1.0 |
| 2 | $\dot{\theta}$ | -8.0 | 8.0 |



- Actions : **Continuous** action spaces

| Num | Action | Min | Max |
|-----|--------------|------|-----|
| 0 | Joint effort | -2.0 | 2.0 |

```
state: [0.88760426 0.46060687 0.60495138] | action: [1.5591747] |  
next_state: [0.8587901 0.5123276 1.18428273] | reward: -0.268161  
02634748273 | done: False  
state: [0.8587901 0.5123276 1.18428273] | action: [1.0582479] |  
next_state: [0.81139809 0.58449392 1.72726561] | reward: -0.430701  
2460303101 | done: False  
state: [0.81139809 0.58449392 1.72726561] | action: [0.7929939] |  
next_state: [0.73948894 0.6731687 2.28458514] | reward: -0.688669  
3000621688 | done: False  
state: [0.73948894 0.6731687 2.28458514] | action: [-0.6580073] |  
next_state: [0.64251266 0.76627507 2.69076057] | reward: -1.06772  
66789916096 | done: False  
state: [0.64251266 0.76627507 2.69076057] | action: [-1.4092577] |  
next_state: [0.51847703 0.85509156 3.05407822] | reward: -1.48817  
52666467798 | done: False
```



Pendulum

- Reward

The precise equation for reward:

$$-(\theta^2 + 0.1 \cdot \dot{\theta}^2 + 0.001 \cdot a^2)$$

Theta is normalized between $-\pi$ and π . Therefore, the lowest cost is $-(\pi^2 + 0.1 \cdot 8^2 + 0.001 \cdot 2^2) = -16.2736044$, and the highest cost is 0 . In essence, the goal is to remain at zero angle (vertical), with the least rotational velocity, and the least effort.

- Episode Termination

There is no specified termination. Adding a maximum number of steps might be a good idea.

NOTE: Your environment object could be wrapped by the TimeLimit wrapper, if created using the "gym.make" method. In that case it will terminate after 200 steps.

Deep Deterministic Policy Gradient (DDPG)

- DDPG Algorithm

Algorithm 1 DDPG algorithm

Randomly initialize critic network $Q(s, a|\theta^Q)$ and actor $\mu(s|\theta^\mu)$ with weights θ^Q and θ^μ .

Initialize target network Q' and μ' with weights $\theta^{Q'} \leftarrow \theta^Q, \theta^{\mu'} \leftarrow \theta^\mu$

Initialize replay buffer R

for episode = 1, M **do**

 Initialize a random process \mathcal{N} for action exploration

 Receive initial observation state s_1

for t = 1, T **do**

 Select action $a_t = \mu(s_t|\theta^\mu) + \mathcal{N}_t$ according to the current policy and exploration noise

 Execute action a_t and observe reward r_t and observe new state s_{t+1}

 Store transition (s_t, a_t, r_t, s_{t+1}) in R

 Sample a random minibatch of N transitions (s_i, a_i, r_i, s_{i+1}) from R

 Set $y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'})$

 Update critic by minimizing the loss: $L = \frac{1}{N} \sum_i (y_i - Q(s_i, a_i|\theta^Q))^2$

 Update the actor policy using the sampled policy gradient:

$$\nabla_{\theta^\mu} J \approx \frac{1}{N} \sum_i \nabla_a Q(s, a|\theta^Q)|_{s=s_i, a=\mu(s_i)} \nabla_{\theta^\mu} \mu(s|\theta^\mu)|_{s_i}$$

 Update the target networks:

$$\theta^{Q'} \leftarrow \tau \theta^Q + (1 - \tau) \theta^{Q'}$$

$$\theta^{\mu'} \leftarrow \tau \theta^\mu + (1 - \tau) \theta^{\mu'}$$

end for

end for

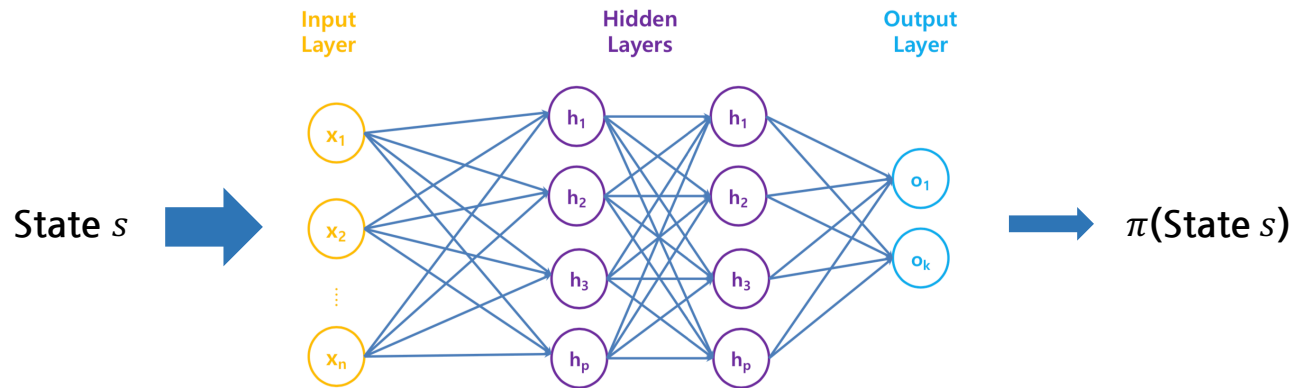
Deep Deterministic Policy Gradient (DDPG)

- 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 replay buffer에 저장
 4. Replay buffer에서 랜덤으로 sample을 추출
 5. 추출한 sample로 Actor & Critic network 업데이트
 6. Actor & Critic에 대해 Soft target 업데이트

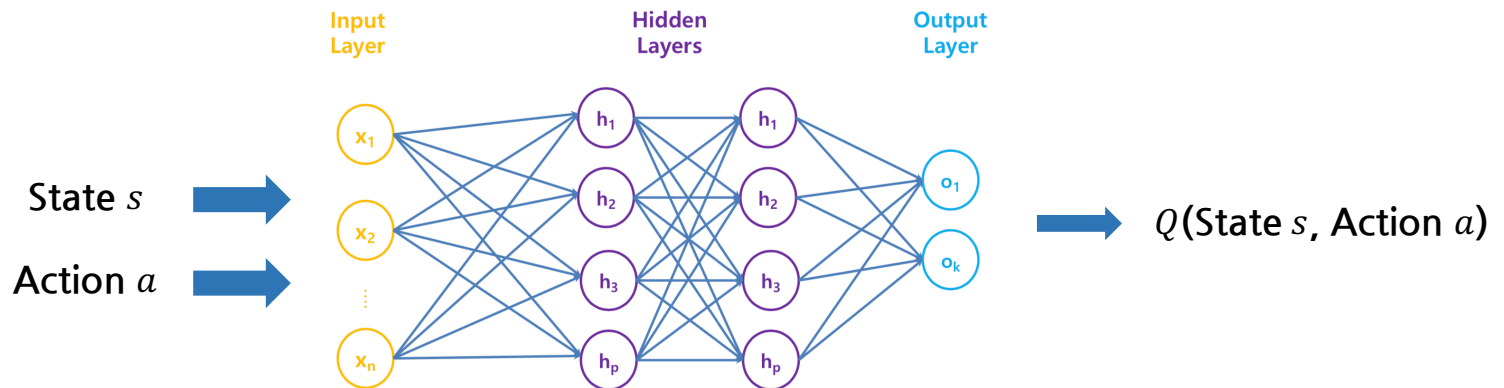


Deep Deterministic Policy Gradient (DDPG)

- Actor network



- Critic network



Deep Deterministic Policy Gradient (DDPG)

- Actor network

```
4  class Actor(nn.Module):
5      def __init__(self, state_size, action_size, args):
6          super(Actor, self).__init__()
7          self.fc1 = nn.Linear(state_size, args.hidden_size)
8          self.fc2 = nn.Linear(args.hidden_size, args.hidden_size)
9          self.fc3 = nn.Linear(args.hidden_size, action_size)
10
11     def forward(self, x):
12         x = torch.relu(self.fc1(x))
13         x = torch.relu(self.fc2(x))
14         policy = self.fc3(x)
15
16         return policy
```



Deep Deterministic Policy Gradient (DDPG)

- Critic network

```
18 class Critic(nn.Module):
19     def __init__(self, state_size, action_size, args):
20         super(Critic, self).__init__()
21         self.fc1 = nn.Linear(state_size + action_size, args.hidden_size)
22         self.fc2 = nn.Linear(args.hidden_size, args.hidden_size)
23         self.fc3 = nn.Linear(args.hidden_size, 1)
24
25     def forward(self, states, actions):
26         x = torch.cat([states, actions], dim=1)
27         x = torch.relu(self.fc1(x))
28         x = torch.relu(self.fc2(x))
29         q_value = self.fc3(x)
30
31         return q_value
```



Learning process

1. 상태에 따른 행동 선택

```
114 policy = actor(torch.Tensor(state))
115 action = get_action(policy, ou_noise) train.py
```

```
19 def get_action(policy, ou_noise):
20     action = policy.detach().numpy() + ou_noise.sample()
21
22     return action utils.py
```

- Ornstein-Uhlenbeck noise (OU noise) - theta : 0.15, mu : 0.0, sigma : 0.2
 - $dx_t = \theta(\mu - x_t)dt + \sigma dW_t$

```
4 class OUNoise:
5     def __init__(self, action_size, theta, mu, sigma):
6         self.action_size = action_size
7         self.theta = theta
8         self.mu = mu
9         self.sigma = sigma
10        self.X = np.zeros(self.action_size)
11
12    def sample(self):
13        dx = self.theta * (self.mu - self.X)
14        dx = dx + self.sigma * np.random.randn(len(self.X))
15        self.X = self.X + dx
16
17    return self.X
```

```
dx [-0.25447483]
X [-0.25447483]

dx [-0.20936269]
X [-0.46383752]

dx [0.1268112]
X [-0.33702632]

dx [0.18123875]
X [-0.15578757]

dx [-0.0240699]
X [-0.17985747]
```

utils.py



Learning process

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

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

3. Sample (s, a, r, s') 을 replay buffer에 저장

```
97         replay_buffer = deque(maxlen=10000)
```

```
120         mask = 0 if done else 1
```

```
121
```

```
122         replay_buffer.append((state, action, reward, next_state, mask))
```

4. Replay buffer에서 랜덤으로 sample을 추출 (Batch size : 64)

```
128         mini_batch = random.sample(replay_buffer, args.batch_size)
```



Learning process

5. 추출한 sample로 Actor & Critic network 업데이트

- Critic Loss

$$J_Q(\phi) = (\underbrace{r + \gamma Q_{\phi}(s', \pi_{\theta}(s'))}_{\text{Target}} - \underbrace{Q_{\phi}(s, a)}_{\text{Prediction}})^2$$

```
49 # update critic
50 criterion = torch.nn.MSELoss()
51
52 # get Q-value
53 q_value = critic(torch.Tensor(states), actions).squeeze(1)
54
55 # get target
56 target_next_policy = target_actor(torch.Tensor(next_states))
57 target_next_q_value = target_critic(torch.Tensor(next_states), target_next_policy).squeeze(1)
58 target = rewards + masks * args.gamma * target_next_q_value
59
60 critic_loss = criterion(q_value, target.detach())
61 critic_optimizer.zero_grad()
62 critic_loss.backward()
63 critic_optimizer.step()
```



Learning process

5. 추출한 sample로 Actor & Critic network 업데이트

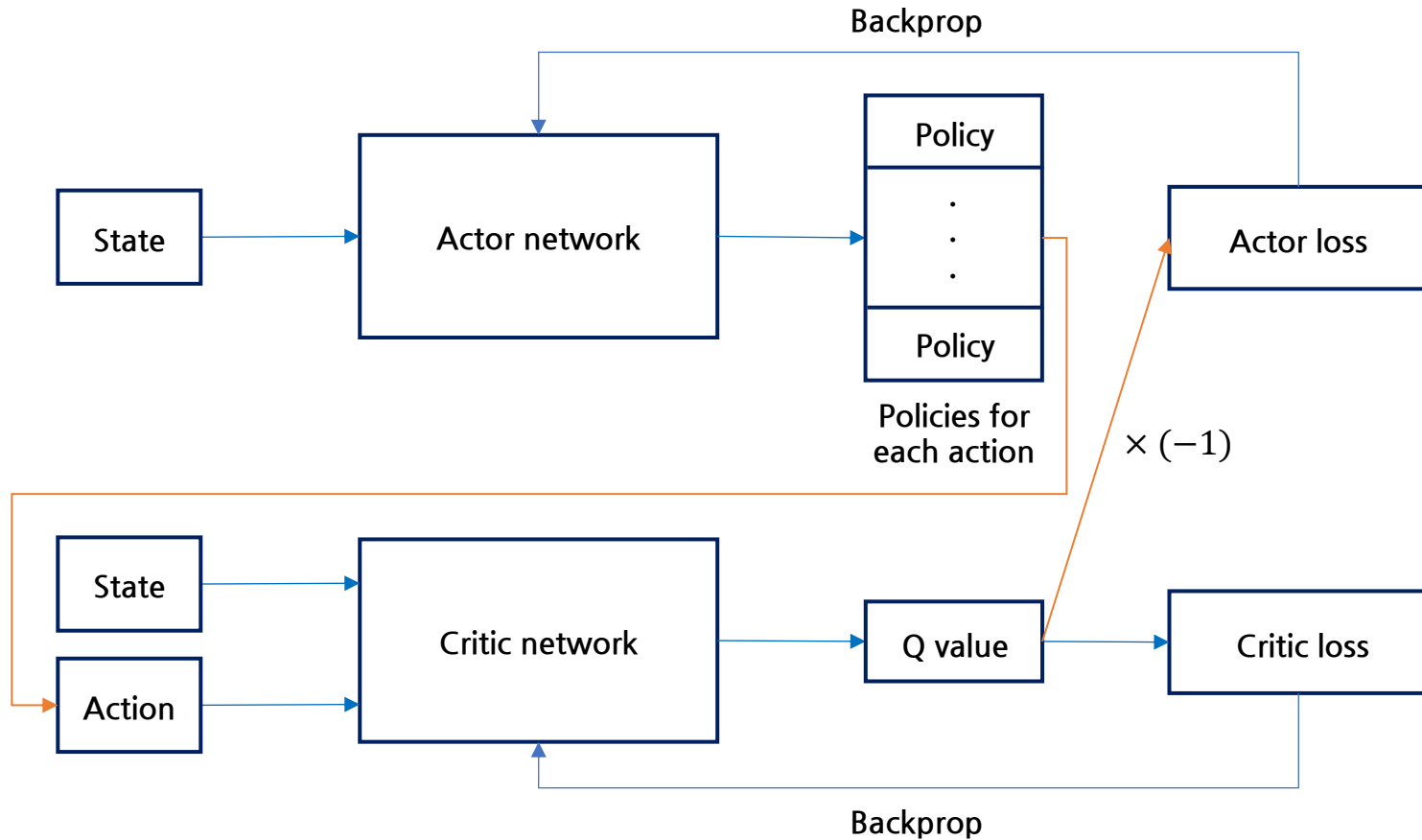
- Actor Loss

$$J_{\pi}(\theta) = -\frac{1}{N} \sum Q_{\phi}(s, \pi_{\theta}(s))$$

```
65     # update actor
66     policy = actor(torch.Tensor(states))
67
68     actor_loss = -critic(torch.Tensor(states), policy).mean()
69     actor_optimizer.zero_grad()
70     actor_loss.backward()
71     actor_optimizer.step()
```



Learning process



Learning process

6. Actor & Critic에 대해 Soft target 업데이트

- Initialize target model

```
92     hard_target_update(actor, critic, target_actor, target_critic) train.py
```

```
24 def hard_target_update(actor, critic, target_actor, target_critic):  
25     target_critic.load_state_dict(critic.state_dict())  
26     target_actor.load_state_dict(actor.state_dict()) utils.py
```

- Soft target update ($\tau : 0.001$)

$$\phi^{Q'} \leftarrow \tau \phi^Q + (1 - \tau) \phi^{Q'}$$

$$\theta^{\pi'} \leftarrow \tau \theta^{\pi} + (1 - \tau) \theta^{\pi'}$$

train.py

```
135     soft_target_update(actor, critic, target_actor, target_critic, args.tau)
```

```
28 def soft_target_update(actor, critic, target_actor, target_critic, tau):  
29     soft_update(critic, target_critic, tau)  
30     soft_update(actor, target_actor, tau)  
31  
32 def soft_update(net, target_net, tau):  
33     for param, target_param in zip(net.parameters(), target_net.parameters()):  
34         target_param.data.copy_(tau * param.data + (1.0 - tau) * target_param.data) utils.py
```



Hyperparameter

```
15 parser = argparse.ArgumentParser()
16 parser.add_argument('--env_name', type=str, default="Pendulum-v0")
17 parser.add_argument('--load_model', type=str, default=None)
18 parser.add_argument('--save_path', default='./save_model/', help='')
19 parser.add_argument('--render', action="store_true", default=False)
20 parser.add_argument('--gamma', type=float, default=0.99)
21 parser.add_argument('--hidden_size', type=int, default=64)
22 parser.add_argument('--batch_size', type=int, default=64)
23 parser.add_argument('--actor_lr', type=float, default=1e-3)
24 parser.add_argument('--critic_lr', type=float, default=1e-3)
25 parser.add_argument('--theta', type=float, default=0.15)
26 parser.add_argument('--mu', type=float, default=0.0)
27 parser.add_argument('--sigma', type=float, default=0.2)
28 parser.add_argument('--tau', type=float, default=0.001)
29 parser.add_argument('--max_iter_num', type=int, default=1000)
30 parser.add_argument('--log_interval', type=int, default=10)
31 parser.add_argument('--goal_score', type=int, default=-300)
32 parser.add_argument('--logdir', type=str, default='./logs',
33 | | | | | help='tensorboardx logs directory')
34 args = parser.parse_args()
```



Main loop

- Initialization
 - Seed - random number 고정
 - Actor & Critic network
 - Target actor & critic network
 - Actor & Critic optimizer
 - Hard target update
 - OU noise
 - TensorboardX
 - Replay buffer
 - Recent rewards

```
74 def main():
75     env = gym.make(args.env_name)
76     env.seed(500)
77     torch.manual_seed(500)
78
79     state_size = env.observation_space.shape[0]
80     action_size = env.action_space.shape[0]
81     print('state size:', state_size)
82     print('action size:', action_size)
83
84     actor = Actor(state_size, action_size, args)
85     target_actor = Actor(state_size, action_size, args)
86     critic = Critic(state_size, action_size, args)
87     target_critic = Critic(state_size, action_size, args)
88
89     actor_optimizer = optim.Adam(actor.parameters(), lr=args.actor_lr)
90     critic_optimizer = optim.Adam(critic.parameters(), lr=args.critic_lr)
91
92     hard_target_update(actor, critic, target_actor, target_critic)
93     ou_noise = OUNoise(action_size, args.theta, args.mu, args.sigma)
94
95     writer = SummaryWriter(args.logdir)
96
97     replay_buffer = deque(maxlen=10000)
98     recent_rewards = deque(maxlen=100)
99     steps = 0
```



Main loop

- Episode 진행
 - 상태에 따른 행동 선택
 - 다음 상태와 보상을 받음
 - Replay buffer에 저장

```
101 for episode in range(args.max_iter_num):
102     done = False
103     score = 0
104
105     state = env.reset()
106     state = np.reshape(state, [1, state_size])
107
108     while not done:
109         if args.render:
110             env.render()
111
112         steps += 1
113
114         policy = actor(torch.Tensor(state))
115         action = get_action(policy, ou_noise)
116
117         next_state, reward, done, _ = env.step(action)
118
119         next_state = np.reshape(next_state, [1, state_size])
120         mask = 0 if done else 1
121
122         replay_buffer.append((state, action, reward, next_state, mask))
123
124         state = next_state
125         score += reward
```



Main loop

- Episode 진행
 - Replay buffer에서 랜덤으로 64개의 sample을 추출 → Mini batch
 - Train model
 - Soft target update

```
127         if steps > args.batch_size:
128             mini_batch = random.sample(replay_buffer, args.batch_size)
129
130             actor.train(), critic.train()
131             target_actor.train(), target_critic.train()
132             train_model(actor, critic, target_actor, target_critic,
133                         actor_optimizer, critic_optimizer, mini_batch)
134
135             soft_target_update(actor, critic, target_actor, target_critic, args.tau)
136
137         if done:
138             recent_rewards.append(score)
```

Main loop

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

```
140         if episode % args.log_interval == 0:
141             print('{} episode | score_avg: {:.2f}'.format(episode, np.mean(recent_rewards)))
142             writer.add_scalar('log/score', float(score), episode)
143
144         if np.mean(recent_rewards) > args.goal_score:
145             if not os.path.isdir(args.save_path):
146                 os.makedirs(args.save_path)
147
148             ckpt_path = args.save_path + 'model.pth'
149             torch.save(actor.state_dict(), ckpt_path)
150             print('Recent rewards exceed -300. So end')
151             break
```

Train model

- Mini batch → Numpy array
- Mini batch에 있는 64개의 sample들을 각각 나눔
 - state - (64, 3)
 - action - (64, 1)
 - reward - (64)
 - next_state - (64, 3)
 - mask - (64)

```
36 def train_model(actor, critic, target_actor, target_critic,
37                 actor_optimizer, critic_optimizer, mini_batch):
38     mini_batch = np.array(mini_batch)
39     states = np.vstack(mini_batch[:, 0])
40     actions = list(mini_batch[:, 1])
41     rewards = list(mini_batch[:, 2])
42     next_states = np.vstack(mini_batch[:, 3])
43     masks = list(mini_batch[:, 4])
44
45     actions = torch.Tensor(actions).squeeze(1)
46     rewards = torch.Tensor(rewards).squeeze(1)
47     masks = torch.Tensor(masks)
```



Train model

- Prediction
 - q_value - (64)
- Target
 - target_next_policy - (64, 1)
 - target_next_q_value - (64)
 - target - (64)
- Update critic - MSE Loss
 - $J_Q(\phi) = (r + \gamma Q_\phi(s', \pi_\theta(s')) - Q_\phi(s, a))^2$

```
49     # update critic
50     criterion = torch.nn.MSELoss()
51
52     # get Q-value
53     q_value = critic(torch.Tensor(states), actions).squeeze(1)
54
55     # get target
56     target_next_policy = target_actor(torch.Tensor(next_states))
57     target_next_q_value = target_critic(torch.Tensor(next_states), target_next_policy).squeeze(1)
58     target = rewards + masks * args.gamma * target_next_q_value
59
60     critic_loss = criterion(q_value, target.detach())
61     critic_optimizer.zero_grad()
62     critic_loss.backward()
63     critic_optimizer.step()
```



Train model

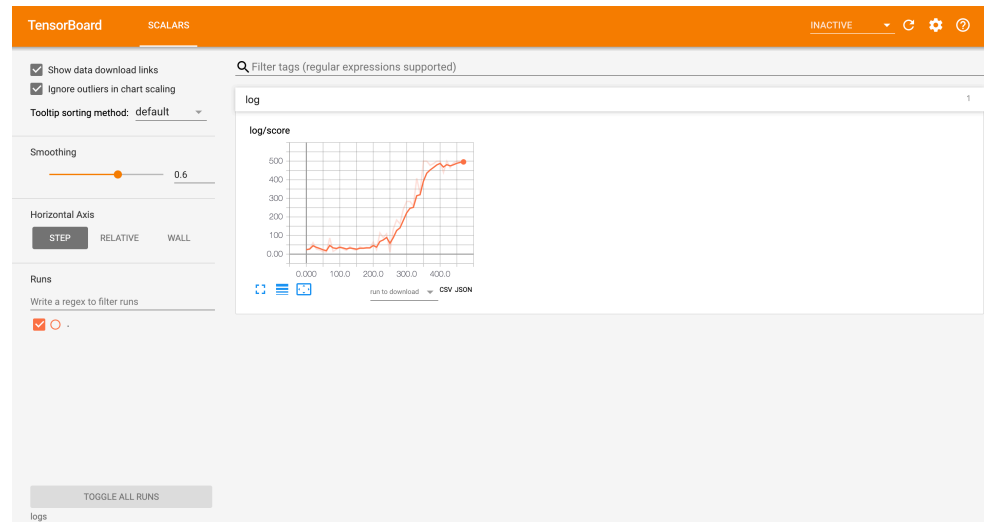
- Update actor
 - policy - (64, 1)
 - critic(torch.Tensor(state), policy) - (64, 1)
 - $J_{\pi}(\theta) = -\frac{1}{N} \sum Q_{\phi}(s, \pi_{\theta}(s))$

```
65     # update actor
66     policy = actor(torch.Tensor(states))
67
68     actor_loss = -critic(torch.Tensor(states), policy).mean()
69     actor_optimizer.zero_grad()
70     actor_loss.backward()
71     actor_optimizer.step()
```



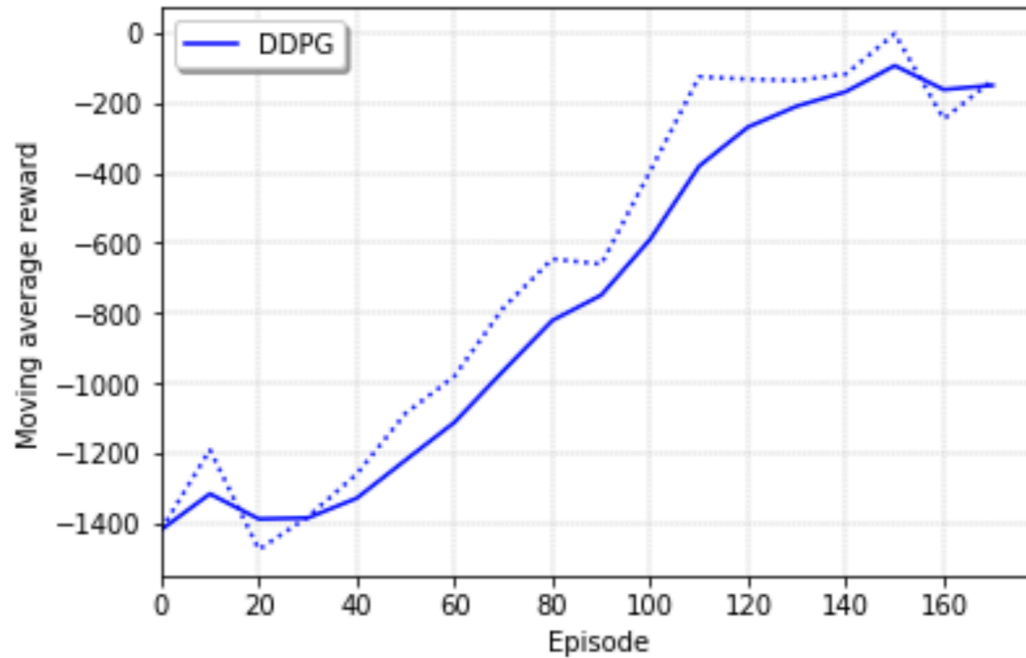
Train & TensorboardX

- 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`

DDPG video

- Learning to move: DDPG Algorithms on Gym MuJoCo
<https://www.youtube.com/watch?v=iFg5lcUzSYU&t=14>
- Deep RL for Robotic Manipulation (DDPG + HER)
<https://www.youtube.com/watch?v=K-foX756KTc&t=44>



Thank you

