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

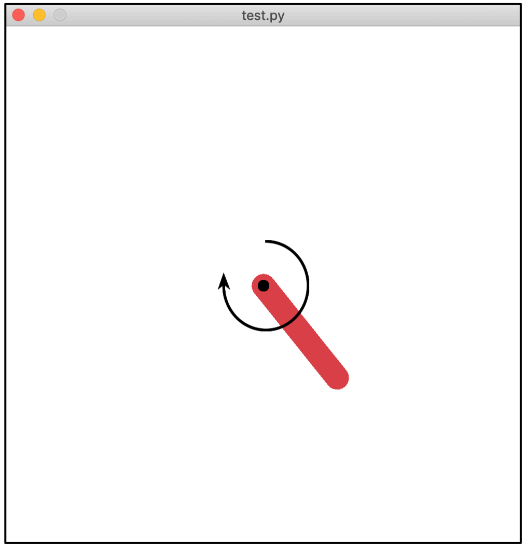
Pendulum

- 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	theta dot	-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*theta_dt^2 + 0.001*action^2)
```

Theta is normalized between -pi and pi. Therefore, the lowest cost is `-(pi^2 + 0.1*8^2 + 0.001*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

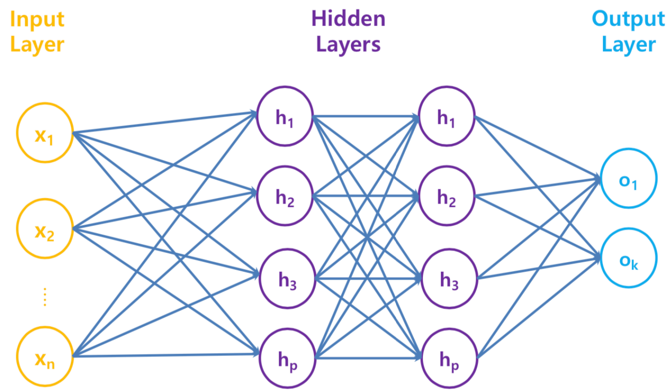
---

source : https://arxiv.org/pdf/1509.02971.pdf

Deep Deterministic Policy Gradient (DDPG)

- Actor network

State s

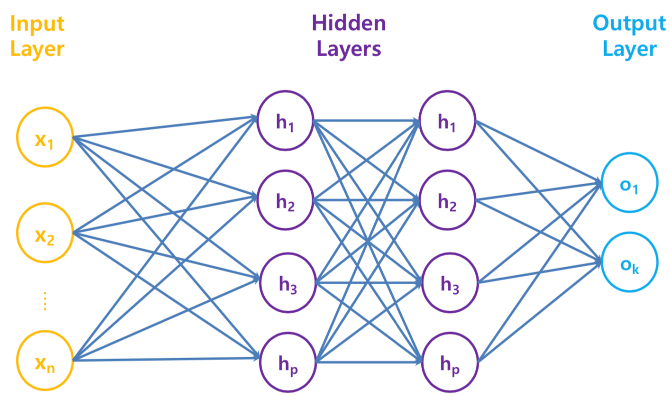


π(State s)

- Critic network

State s

Action a



Q(State s, Action a)

Deep Deterministic Policy Gradient (DDPG)

- 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.relu(self.fc1(x))
        x = torch.relu(self.fc2(x))
        policy = self.fc3(x)

        return policy
```

Deep Deterministic Policy Gradient (DDPG)

- Critic network

```
class Critic(nn.Module):
    def __init__(self, state_size, action_size, args):
        super(Critic, self).__init__()
        self.fc1 = nn.Linear(state_size + action_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, states, actions):
        x = torch.cat([states, actions], dim=1)
        x = torch.relu(self.fc1(x))
        x = torch.relu(self.fc2(x))
        q_value = self.fc3(x)

        return q_value
```

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

Learning process

1. 상태에 따른 행동 선택

```
policy = actor(torch.Tensor(state))
action = get_action(policy, ou_noise)
```

```
def get_action(policy, ou_noise):
    action = policy.detach().numpy() + ou_noise.sample()

    return action
```

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

```
class OUNoise:
    def __init__(self, action_size, theta, mu, sigma):
        self.action_size = action_size
        self.theta = theta
        self.mu = mu
        self.sigma = sigma
        self.X = np.zeros(self.action_size)

    def sample(self):
        dx = self.theta * (self.mu - self.X)
        dx = dx + self.sigma * np.random.randn(len(self.X))
        self.X = self.X + dx

        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]
```

Control + Optimization Research Lab

Learning process

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

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

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

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

```
mask = 0 if done else 1

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

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

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

Learning process

5. 추출한 sample로 Actor & Critic network 업데이트
    - Critic Loss

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

```
# update critic
criterion = torch.nn.MSELoss()

# get Q-value
q_value = critic(torch.Tensor(states), actions).squeeze(1)

# get target
target_next_policy = target_actor(torch.Tensor(next_states))
target_next_q_value = target_critic(torch.Tensor(next_states), target_next_policy).squeeze(1)
target = rewards + masks * args.gamma * target_next_q_value

critic_loss = criterion(q_value, target.detach())
critic_optimizer.zero_grad()
critic_loss.backward()
critic_optimizer.step()
```

Learning process

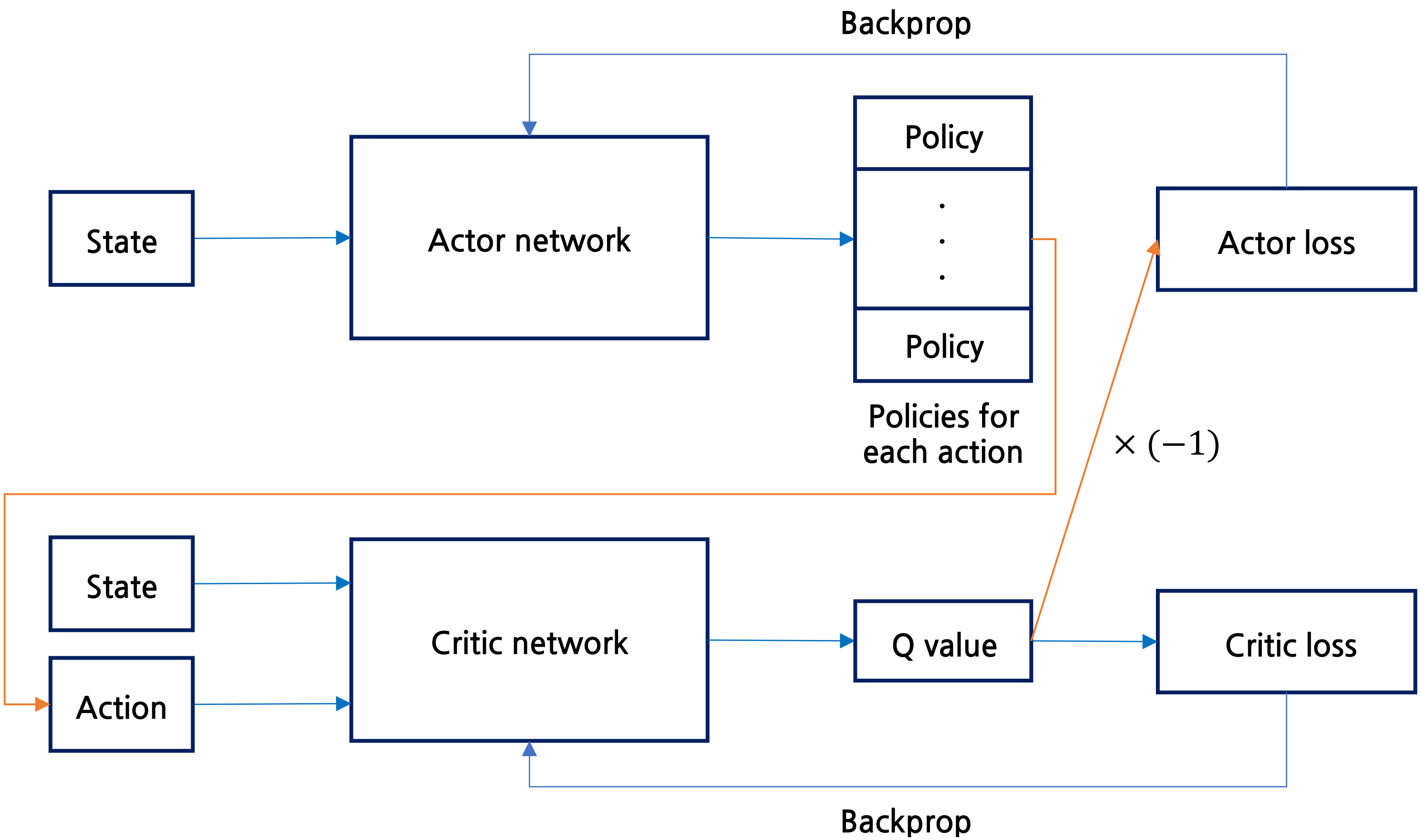
5. 추출한 sample로 Actor & Critic network 업데이트
   - Actor Loss

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

```
# update actor
policy = actor(torch.Tensor(states))

actor_loss = -critic(torch.Tensor(states), policy).mean()
actor_optimizer.zero_grad()
actor_loss.backward()
actor_optimizer.step()
```

Learning process



Learning process

6. Actor & Critic에 대해 Soft target 업데이트

- Initialize target model

```
hard_target_update(actor, critic, target_actor, target_critic)
```

```
def hard_target_update(actor, critic, target_actor, target_critic):
    target_critic.load_state_dict(critic.state_dict())
    target_actor.load_state_dict(actor.state_dict())
```

- Soft target update (τ : 0.001)

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

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

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

```
def soft_target_update(actor, critic, target_actor, target_critic, tau):
    soft_update(critic, target_critic, tau)
    soft_update(actor, target_actor, tau)

def soft_update(net, target_net, tau):
    for param, target_param in zip(net.parameters(), target_net.parameters()):
        target_param.data.copy_(tau * param.data + (1.0 - tau) * target_param.data)
```

Hyperparameter

```python
parser = argparse.ArgumentParser()
parser.add_argument('--env_name', type=str, default="Pendulum-v0")
parser.add_argument('--load_model', type=str, default=None)
parser.add_argument('--save_path', default='./save_model/', help='')
parser.add_argument('--render', action="store_true", default=False)
parser.add_argument('--gamma', type=float, default=0.99)
parser.add_argument('--hidden_size', type=int, default=64)
parser.add_argument('--batch_size', type=int, default=64)
parser.add_argument('--actor_lr', type=float, default=1e-3)
parser.add_argument('--critic_lr', type=float, default=1e-3)
parser.add_argument('--theta', type=float, default=0.15)
parser.add_argument('--mu', type=float, default=0.0)
parser.add_argument('--sigma', type=float, default=0.2)
parser.add_argument('--tau', type=float, default=0.001)
parser.add_argument('--max_iter_num', type=int, default=1000)
parser.add_argument('--log_interval', type=int, default=10)
parser.add_argument('--goal_score', type=int, default=-300)
parser.add_argument('--logdir', type=str, default='./logs',
                    help='tensorboardx logs directory')
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

```python
def main():
    env = gym.make(args.env_name)
    env.seed(500)
    torch.manual_seed(500)

    state_size = env.observation_space.shape[0]
    action_size = env.action_space.shape[0]
    print('state size:', state_size)
    print('action size:', action_size)

    actor = Actor(state_size, action_size, args)
    target_actor = Actor(state_size, action_size, args)
    critic = Critic(state_size, action_size, args)
    target_critic = Critic(state_size, action_size, args)

    actor_optimizer = optim.Adam(actor.parameters(), lr=args.actor_lr)
    critic_optimizer = optim.Adam(critic.parameters(), lr=args.critic_lr)

    hard_target_update(actor, critic, target_actor, target_critic)
    ou_noise = OUNoise(action_size, args.theta, args.mu, args.sigma)

    writer = SummaryWriter(args.logdir)

    replay_buffer = deque(maxlen=10000)
    recent_rewards = deque(maxlen=100)
    steps = 0
```

Main loop

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

```
for episode in range(args.max_iter_num):
    done = False
    score = 0

    state = env.reset()
    state = np.reshape(state, [1, state_size])

    while not done:
        if args.render:
            env.render()

        steps += 1

        policy = actor(torch.Tensor(state))
        action = get_action(policy, ou_noise)

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

        next_state = np.reshape(next_state, [1, state_size])
        mask = 0 if done else 1

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

        state = next_state
        score += reward
```

Main loop

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

```
if steps > args.batch_size:
    mini_batch = random.sample(replay_buffer, args.batch_size)

    actor.train(), critic.train()
    target_actor.train(), target_critic.train()
    train_model(actor, critic, target_actor, target_critic,
                actor_optimizer, critic_optimizer, mini_batch)

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

if done:
    recent_rewards.append(score)
```

Main loop

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

```
if episode % args.log_interval == 0:
    print('{} episode | score_avg: {:.2f}'.format(episode, np.mean(recent_rewards)))
    writer.add_scalar('log/score', float(score), episode)

if np.mean(recent_rewards) > args.goal_score:
    if not os.path.isdir(args.save_path):
        os.makedirs(args.save_path)

    ckpt_path = args.save_path + 'model.pth'
    torch.save(actor.state_dict(), ckpt_path)
    print('Recent rewards exceed -300. So end')
    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)

```
def train_model(actor, critic, target_actor, target_critic,
                actor_optimizer, critic_optimizer, mini_batch):
    mini_batch = np.array(mini_batch)
    states = np.vstack(mini_batch[:, 0])
    actions = list(mini_batch[:, 1])
    rewards = list(mini_batch[:, 2])
    next_states = np.vstack(mini_batch[:, 3])
    masks = list(mini_batch[:, 4])

    actions = torch.Tensor(actions).squeeze(1)
    rewards = torch.Tensor(rewards).squeeze(1)
    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
criterion = torch.nn.MSELoss()

# get Q-value
q_value = critic(torch.Tensor(states), actions).squeeze(1)

# get target
target_next_policy = target_actor(torch.Tensor(next_states))
target_next_q_value = target_critic(torch.Tensor(next_states), target_next_policy).squeeze(1)
target = rewards + masks * args.gamma * target_next_q_value

critic_loss = criterion(q_value, target.detach())
critic_optimizer.zero_grad()
critic_loss.backward()
critic_optimizer.step()
```

Train model

- Update critic - MSE Loss
  - $J_Q(\theta) = \big(\underbrace{r + \gamma Q_{\theta^-}\left(s', \pi_{\phi^-}(s')\right)}_{\text{Target}} - \underbrace{Q_\theta(s,a)}_{\text{Prediction}}\big)^2$

```
# update critic
criterion = torch.nn.MSELoss()

# get Q-value
q_value = critic(torch.Tensor(states), actions).squeeze(1)

# get target
target_next_policy = target_actor(torch.Tensor(next_states))
target_next_q_value = target_critic(torch.Tensor(next_states), target_next_policy).squeeze(1)
target = rewards + masks * args.gamma * target_next_q_value

critic_loss = criterion(q_value, target.detach())
critic_optimizer.zero_grad()
critic_loss.backward()
critic_optimizer.step()
```

Train model

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

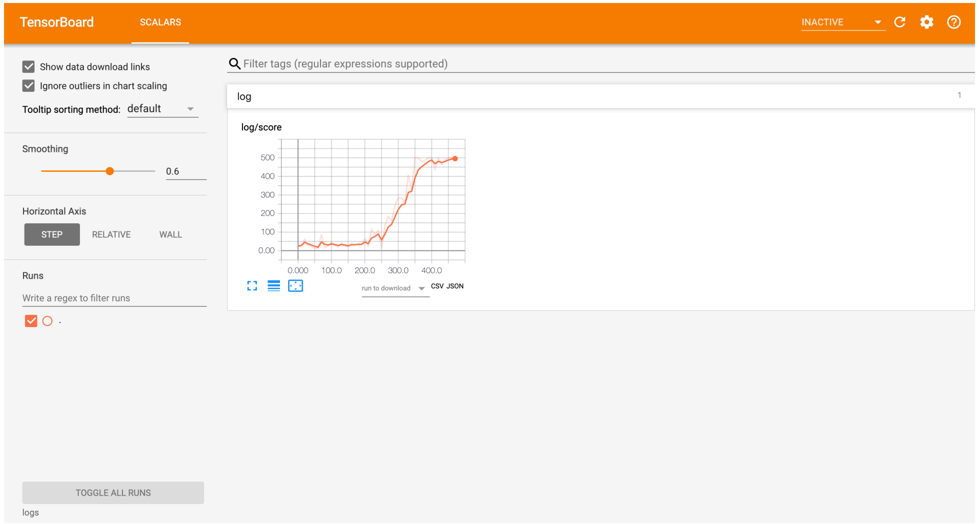
```
# update actor
policy = actor(torch.Tensor(states))

actor_loss = -critic(torch.Tensor(states), policy).mean()
actor_optimizer.zero_grad()
actor_loss.backward()
actor_optimizer.step()
```

Train & TensorboardX

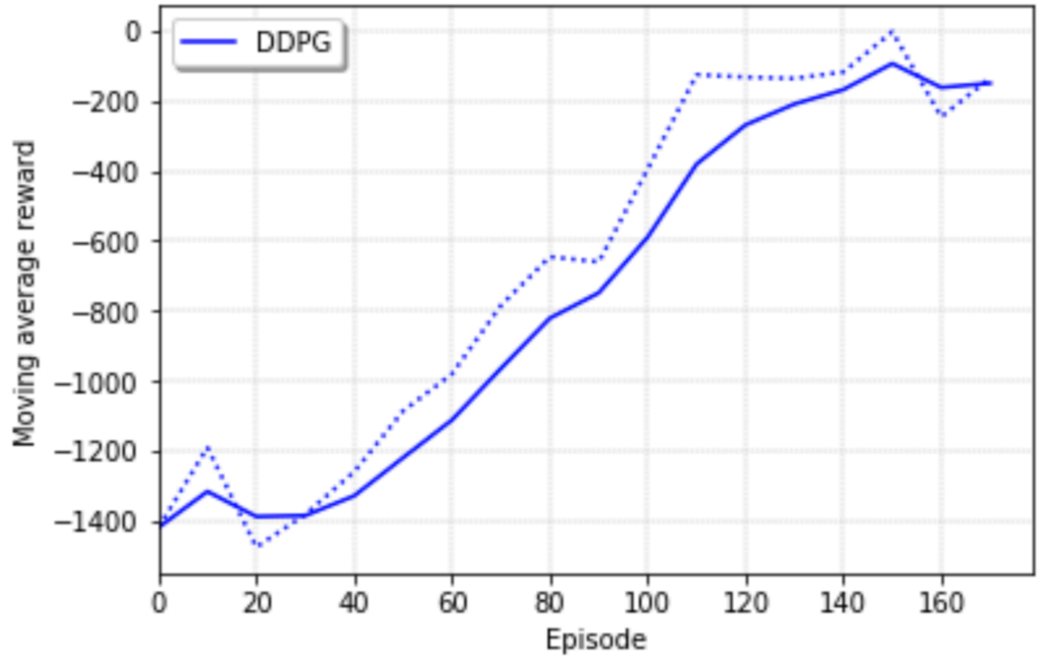
- Terminal A - train 실행
  - ➢ conda activate env_name
  - ➢ python train.py

- Terminal B - tensorboardX 실행
  - ➢ conda activate env_name
  - ➢ tensorboard --logdir logs
  - ➢ (웹에서) localhost:6006



Learning curve & Test

- Learning curve



- Test
  - python test.py

Thank you