Continuous Action Space

한양대학교 최솔비

Action space

- continuous action 은 특정 범위를 지니고 있다.
- 매 스텝마다 해당 범위 내에서 구체적인 action 값을 정해 environment 로 전달해야한다.
- `env.step()` 에 특정 shape 의 Numpy vector를 action 값으로 전달하게 된다.
- discrete 과 continuous action 을 함께 전달해야할 때는 `gym.spaces.Tuple` 을 사용한다.

Environments

- 주로 물리 환경에서 continuous action space 를 많이 사용하기 때문에 물리 시뮬레이션 패키지를 이용해 본다. (로보틱스)
- Mujoco 는 유료이기 때문에 비슷한 기능을 제공하는 **Pybullet** 을 이용하도록 한다.
- `pip install pybullet` 명령어로 설치.
- 아래 샘플 코드로 환경 생성 테스트.

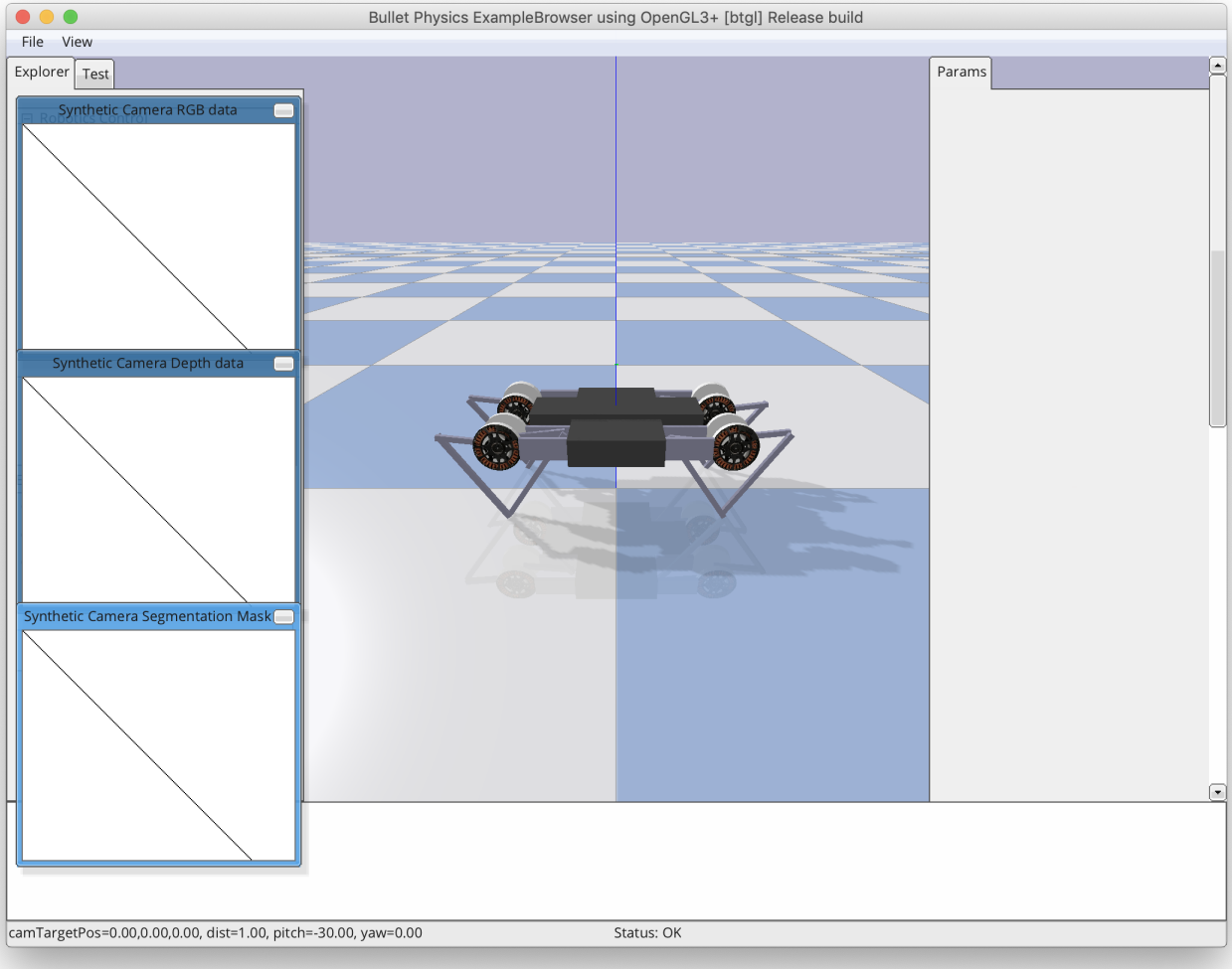
```
import gym
import pybullet_envs

ENV_ID = "MinitaurBulletEnv-v0"
RENDER = True

if __name__ == "__main__":
    spec = gym.envs.registry.spec(ENV_ID)
    spec._kwargs['render'] = RENDER
    env = gym.make(ENV_ID)

    print("Observation space:", env.observation_space)
    print("Action space:", env.action_space)
    print(env)
    print(env.reset())
    input("Press any key to exit\n")
    env.close()
```

결과 : GUI 창 생성


Bullet Physics ExampleBrowser using OpenGL3+ [btgl] Release build
File
View
Explorer
Test
Synthetic Camera RGB data
Synthetic Camera Depth data
Synthetic Camera Segmentation Mask
Params
camTargetPos=0.00,0.00,0.00, dist=1.00, pitch=-30.00, yaw=0.00
Status: OK

결과: environment information

```
Observation space: Box(28,)
Action space: Box(8,)
<TimeLimit<MinitaurBulletEnv<MinitaurBulletEnv-v0>>>
[ 1.46701772e+00  1.45132945e+00  1.46136290e+00  1.44365373e+00
  1.47285021e+00  1.47840277e+00  1.47020185e+00  1.47215955e+00
  1.53663963e+00  1.59704334e+00  1.55490607e+00  1.60954856e+00
  1.57085014e+00  1.51026007e+00  1.56602058e+00  1.53683121e+00
  6.11996164e-01  7.30885124e-01  6.55958296e-01  7.92429732e-01
  5.57251898e-01  5.21320501e-01  5.80350028e-01  5.68792656e-01
 -3.29284113e-03  6.60923886e-04 -1.45173773e-04  9.99994350e-01]
Press any key to exit
```

- observation 28개 : 로봇의 물리 파라미터(속도, 위치, 가속 등)
- action 8개 : 모터의 파라미터(다리 마다 2개)
- reward = (로봇이 이동한 거리) - (사용한 에너지)

The Actor-Critic(A2C) method

- $\nabla J = \nabla_\theta \log \pi_\theta(a|s)(R - V_\theta(s))$
- π_θ policy는 action의 확률 분포를 나타냄.
- $V_\theta(s)$ 는 critic. state value 와 같으며, MSE로 학습된다.
- 위 식에 엔트로피 보너스 $L_H = \pi_\theta(s) \log \pi_\theta(s)$ 를 더해 exploration을 한다.

The Actor-Critic(A2C) method

- 위 식의 policy 부분을 continuous space 에 알맞은 방법으로 바뀌어야 한다. (나머지 부분은 기존의 A2C 방법과 같다.)
    - discrete action space 에서 policy는 각 action에 대한 probability distribution이었음.
    - continuous action space 에서는 policy를 모든 action에 대한 특정 value로 표현하도록 한다. 이는state value(각 state 에서의 기대 reward) 와 혼동하지 않도록 한다.
- 이렇게 특정 value로 나타내게 된다면 exploration을 하기 어렵다는 단점이 있다. -> 확률적으로 value 를 리턴하는 방법.

The Actor-Critic(A2C) method

- 네트워크가 Gaussian distribution의 파라미터를 리턴하게끔 하자!
- 평균값 벡터, 분산 벡터.
- N개의 action이라면 -> $[\mu_1, \mu_2, ..., \mu_N]$, $[\sigma_1^2, \sigma_2^2, ..., \sigma_N^2]$
- 벡터 각각의 파라미터들은 서로 연관 관계 없다(uncorrelated).
- 가우시안 확률 분포의 확률 밀도 함수(pdf)
  - $f(x|\mu, \sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$
  - numerical stability를 위해 위 식에 log 를 씌워 사용한다.
  - $\log \pi_\theta(a|s) = -\frac{(x-\mu)^2}{2\sigma^2} - \log \sqrt{2\pi\sigma^2}$
- 가우시안 확률 분포의 엔트로피: differential entropy 정의 이용
  - $ln\sqrt{2\pi e\sigma^2}$

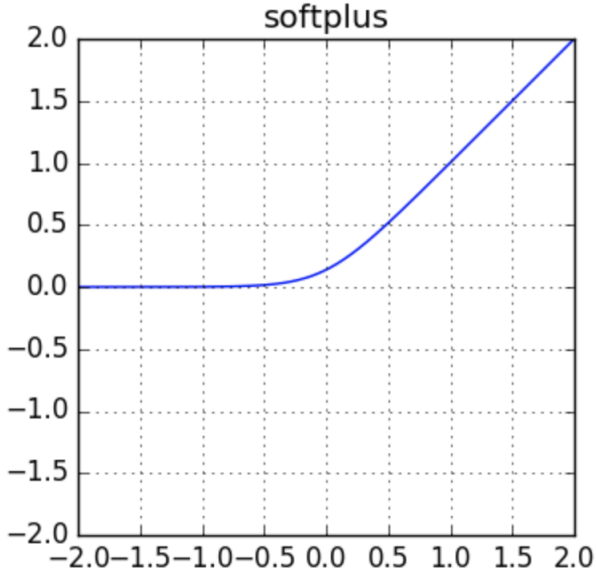
Implementation

```
class ModelA2C(nn.Module):
    def __init__(self, obs_size, act_size):
        super(ModelA2C, self).__init__()

        self.base = nn.Sequential(
            nn.Linear(obs_size, HID_SIZE),
            nn.ReLU(),
        )
        # 평균: 하이퍼볼릭 탄젠트 함수(-1~1 값 리턴)
        self.mu = nn.Sequential(
            nn.Linear(HID_SIZE, act_size),
            nn.Tanh(),
        )
        # 분산: softplus 함수
        self.var = nn.Sequential(
            nn.Linear(HID_SIZE, act_size),
            nn.Softplus(),
        )
        # state value(critic)
        self.value = nn.Linear(HID_SIZE, 1)
```

Soft Plus function (as activation function)

- $f(x) = \log(1 + e^x)$
- 매끄럽게 만든 ReLU 함수 모양이다.
- 분산 값이 0 이상이 되게 한다.
- 미분하면 sigmoid 함수가 된다.



(출처 : https://sefiks.com/2017/08/11/softplus-as-a-neural-networks-activation-function/)

```
def forward(self, x):
    base_out = self.base(x)
    return self.mu(base_out), self.var(base_out),
                              self.value(base_out)
```

- 공통 layer 를 먼저 적용한 후 , 각각의 layer 를 계산한다.

```python
class AgentA2C(ptan.agent.BaseAgent):
    def __init__(self, net, device="cpu"):
        self.net = net
        self.device = device

    def __call__(self, states, agent_states):
        states_v = ptan.agent.float32_preprocessor(
                                    states).to(self.device)

        mu_v, var_v, _ = self.net(states_v)
        mu = mu_v.data.cpu().numpy()
        sigma = torch.sqrt(var_v).data.cpu().numpy()
        # normal distribution으로 랜덤 샘플링
        actions = np.random.normal(mu, sigma)
        # action 의 범위가 -1~1 가 되도록 자름.
        actions = np.clip(actions, -1, 1)
        return actions, agent_states
```

- Agent class는 ptan의 BaseAgent를 상속받아 `__call__` 메서드를 오버라이드 한다.
- `np.clip` 은 범위가 벗어나면 각각 하한, 상한 값으로 맞춰준다.

Training

```
def test_net(net, env, count=10, device="cpu"):
    rewards = 0.0
    steps = 0
    for _ in range(count):
        obs = env.reset()
        while True:
            obs_v = ptan.agent.float32_preprocessor(
                                    [obs]).to(device)
            mu_v = net(obs_v)[0]
            action = mu_v.squeeze(dim=0).
                                   data.cpu().numpy()
            action = np.clip(action, -1, 1)
            obs, reward, done, _ = env.step(action)
            rewards += reward
            steps += 1
            if done:
                break
    return rewards / count, steps / count
```

- test를 위한 함수. exploration 을 하지 않고 model로 부터 리턴받은 평균 값을 그대로 사용한다. (random sampling X)

```
def calc_logprob(mu_v, var_v, actions_v):
    p1 = - ((mu_v - actions_v) ** 2) /
                            (2*var_v.clamp(min=1e-3))
    p2 = - torch.log(torch.sqrt(2 * math.pi * var_v))
    return p1 + p2
```

- 취한 action 의 log probability 계산.
- $\log \pi_\theta(a|s) = -\frac{(x-\mu)^2}{2\sigma^2} - \log \sqrt{2\pi\sigma^2}$
- `torch.clamp()` 함수를 이용하여 0으로 나누는 것을 막는다. (분산이 너무 작아서 underflow 나는 경우)

- 나머지 부분은 전 단원의 A2C과 유사.

```
states_v, actions_v, vals_ref_v = \
      common.unpack_batch_a2c(batch,
      net, last_val_gamma=GAMMA ** REWARD_STEPS,
      device=device)
batch.clear()

optimizer.zero_grad()
mu_v, var_v, value_v = net(states_v)

loss_value_v = F.mse_loss(value_v.squeeze(-1),
                          vals_ref_v)

adv_v = vals_ref_v.unsqueeze(dim=-1) - value_v.detach()
log_prob_v = adv_v * calc_logprob(mu_v, var_v, actions_v)
loss_policy_v = -log_prob_v.mean()
# entropy 계산 공식이 다르다.
entropy_loss_v = ENTROPY_BETA *
        (-(torch.log(2*math.pi*var_v) + 1)/2).mean()

loss_v = loss_policy_v + entropy_loss_v + loss_value_v
loss_v.backward()
optimizer.step()
```

Result

- 수렴하는데 이틀 걸릴 뿐 더러, 성능이 좋지 않다.

Deterministic policy gradients

- A2C method의 variation이지만, off-policy 특성을 가지고 있다.
- 기존의 A2C method에서 actor 는 확률적 policy를 추정하여 리턴한다.
- 이러한 방법에서 policy는 특정 distribution 에 따라 action을 sampling 하게 된다.
- 반면, Deteministic policy gradients에서의 policy는 deterministic함.
    - state 로부터 action 이 바로 결정됨.
    - 그라디언트를 적용할 때 Q value 에 chain rule을 쓸 수 있게 된다.
    - Q를 최대화 하면 policy가 향상됨.

DPG(How differently actor and critic work)

- continuous action 문제에서 actor network 는 state 를 인풋으로 받아 N(action개수)개의 값을 리턴한다.
- DPG 에서는 이 값이 deterministic하다.(같은 network, 같은 input이라면 output 값이 같음)
- critic network 는 input 으로 두 가지 값을 받는다 : **state, action**
- critic 의 output은 state 에 대한 Q value 1개 값이다.
    - DQN 에서는 state 에서 모든 action 에 대한 각각의 Q value 였음. (deterministic)
- state 값을 받아 action으로 바꾸어 주는 actor 함수를 $\mu(s)$라고 하고, 주어진 state,action 으로 하나의 Q value 를 계산하는 critic 함수를 $Q(s, a)$ 라고 한다면, 두 식을 합칠 수 있다.
    - $Q(s, \mu(s))$: s에 관한 일 변수 함수가 됨.

DPG(cont.)

- critic network의 아웃풋 $Q(s, \mu(s))$ 을 최대화하도록 옵티마이징한다.
- 이 과정에서 actor, critic network 모두의 weight 을 업데이트 해야한다.
  -> policy gradient
- chain rule 을 적용하여 그라디언트를 얻을 수 있다.
  - $\nabla_a Q(s, a) \nabla_{\theta_\mu} \mu(s)$
- 기존의 A2C 와 DDPG 는 critic을 사용하는 방법이 다르다.
  - A2C : baseline for a reward(optional)
    이처럼 stochastic 한 policy 에서는 랜덤 샘플링 한 것을 back propagation(미분)할 수 없다.
  - DDPG : critic 으로부터 얻은 Q value 를 이용해 actor's weight 까지 계산 할 수 있다.(전체 미분 가능)
- off-policy 의 장점을 가지게 된다. (replay buffer 등)

Exploration

- deterministic 하기 때문에 exploration을 따로 해주어야 한다.
- actor로부터 받은 action을 환경에 적용하기 전에 noise 를 더해주는 방법.
    - random noise 사용
    - stochastic noise 사용(OU process)
- OU process 는 이전 state 에서 사용한 noise 를 이용하여 다음 noise 를 계산한다.

$$x_{t+1} = x_t + \theta(\mu - x_t) + \sigma N$$

Implemetation

```
class DDPGActor(nn.Module):
    def __init__(self, obs_size, act_size):
        super(DDPGActor, self).__init__()

        self.net = nn.Sequential(
            nn.Linear(obs_size, 400),
            nn.ReLU(),
            nn.Linear(400, 300),
            nn.ReLU(),
            nn.Linear(300, act_size),
            nn.Tanh() # -1~1 사이의 output
        )

    def forward(self, x):
        return self.net(x)
```

```python
class DDPGCritic(nn.Module):
    def __init__(self, obs_size, act_size):
        super(DDPGCritic, self).__init__()

        self.obs_net = nn.Sequential(
            nn.Linear(obs_size, 400),
            nn.ReLU(),
        )
        # observation을 먼저 네트워크에 넣고
        # 그 다음 action 을 합침.
        self.out_net = nn.Sequential(
            nn.Linear(400 + act_size, 300),
            nn.ReLU(),
            nn.Linear(300, 1)
        )

    def forward(self, x, a):
        obs = self.obs_net(x)
        return self.out_net(torch.cat([obs, a], dim=1))
                                    # concatenate
```

DDPG actor and critic networks

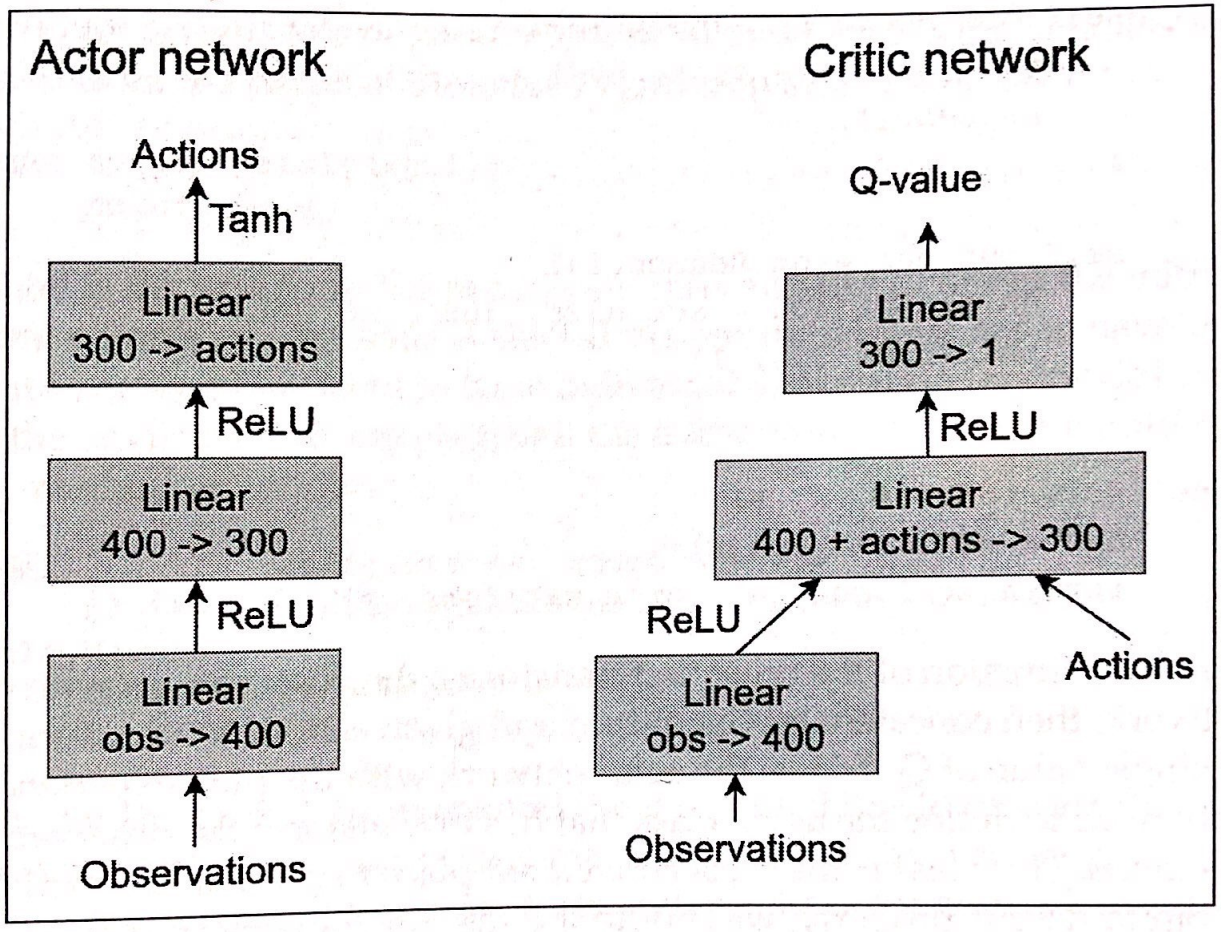


Figure 3: DDPG actor and critic networks

```
class AgentDDPG(ptan.agent.BaseAgent):

    # Orstein-Uhlenbeck exploration process
    def __init__(self, net, device="cpu",
    ou_enabled=True, ou_mu=0.0, ou_teta=0.15,
    ou_sigma=0.2, ou_epsilon=1.0):
        self.net = net
        self.device = device
        self.ou_enabled = ou_enabled
        self.ou_mu = ou_mu
        self.ou_teta = ou_teta
        self.ou_sigma = ou_sigma
        self.ou_epsilon = ou_epsilon

    def initial_state(self):
        return None
```

- optional statefulness: OU process 를 사용하기 위해선, observations 사이에서 OU value(noise)가 어떻게 변했는지 추적해야 함.
- state 추적 <-> stateless

```
def __call__(self, states, agent_states):
    states_v = ptan.agent.float32_preprocessor(
                  states).to(self.device)
    mu_v = self.net(states_v)
    actions = mu_v.data.cpu().numpy()
        # 여기서부터 OU process(noise 생성)
    if self.ou_enabled and self.ou_epsilon > 0:
        new_a_states = []
        for a_state, action in zip(agent_states,
                                   actions):
            if a_state is None:
                a_state = np.zeros(shape=action.shape,
                                   dtype=np.float32)
            a_state += self.ou_teta *
                       (self.ou_mu - a_state)
            a_state += self.ou_sigma *
                np.random.normal(size=action.shape)
            # action에 생성한 noise 를 더한 후, noise 저장
            action += self.ou_epsilon * a_state
            new_a_states.append(a_state)
    else:
        new_a_states = agent_states

    actions = np.clip(actions, -1, 1)
    return actions, new_a_states
```

train code

```
    act_net = model.DDPGActor(env.observation_space.
        shape[0], env.action_space.shape[0]).to(device)
    crt_net = model.DDPGCritic(env.observation_space.
        shape[0], env.action_space.shape[0]).to(device)
    # actor , critic 각각 타겟 네트워크 사용
    tgt_act_net = ptan.agent.TargetNet(act_net)
    tgt_crt_net = ptan.agent.TargetNet(crt_net)

    agent = model.AgentDDPG(act_net, device=device)
    exp_source = ptan.experience.
          ExperienceSourceFirstLast(env, agent,
          gamma=GAMMA, steps_count=1)
    # replay buffer 사용
    buffer = ptan.experience.ExperienceReplayBuffer(
        exp_source, buffer_size=REPLAY_SIZE)
    act_opt = optim.Adam(act_net.parameters(),
                         lr=LEARNING_RATE)
    crt_opt = optim.Adam(crt_net.parameters(),
                         lr=LEARNING_RATE)
```

replay buffer 에서 샘플링

```
batch = buffer.sample(BATCH_SIZE)
```

train critic

```
crt_opt.zero_grad()
q_v = crt_net(states_v, actions_v)
last_act_v = tgt_act_net.target_model(last_states_v)
q_last_v = tgt_crt_net.target_model(last_states_v,
                                    last_act_v)
q_last_v[dones_mask] = 0.0
# one step Bellman equation을 이용해 계산
q_ref_v = rewards_v.unsqueeze(dim=-1) + q_last_v * GAMMA
# target networks에서 구한 q value 와
# critic network에서 구한 값의 차이를 이용해 학습.
critic_loss_v = F.mse_loss(q_v, q_ref_v.detach())
critic_loss_v.backward()
crt_opt.step()
```

train actor

```
act_opt.zero_grad()
cur_actions_v = act_net(states_v)
# Q value 가 최대가 되는 방향으로 학습.
actor_loss_v = -crt_net(states_v, cur_actions_v)
actor_loss_v = actor_loss_v.mean()
actor_loss_v.backward()
act_opt.step()

# target network 업데이트
tgt_act_net.alpha_sync(alpha=1 - 1e-3)
tgt_crt_net.alpha_sync(alpha=1 - 1e-3)
```

- critic network 는 여기서 학습시키지 않는다.
- actor optimizer 만 부르도록 한다.
- **soft sync** : continuous action 문제에서는 매 스텝마다 타겟 네트워크를 업데이트 한다. 적은 비율(alpha)로 업데이트 된다 -> smooth and slow transition from old weight to the new ones.

Result

- 하루정도 걸림. continuous A2C에 비해 나은 결과.
- 하지만 여전히 noise 가 많으며 unstable 함.

Distributional policy gradients

- 2018 년 논문(https://openreview.net/forum?id=SyZipzbCb)
- Distributed Distributional Deep Deterministic Policy Gradient (D4PG)
    i. critic으로부터 받는 single Q-value를 probability distribution 으로 바꾼다. Bellman 방정식은 Bellman operator 로 교체한다.
    ii. n-step Bellman equation (unroll)
    iii. prioritized replay buffer

Architecture

- critic network가 Q value 값 하나를 리턴하는 대신 `N_ATOMS` 값을 리턴하도록 한다. 미리 Q value 값의 범위를 지정 해놓은 후 각 범위에 해당 되는 값이 나올 확률을 리턴하도록 한다.
- D4PG 는 exploration 에 있어서 OU noise 와 랜덤 noise 사용했을 때의 결과가 같다(논문 참고). 따라서 더 간단한 uniformly random noise 를 사용한다.
- D4PG 는 train시에 확률 분포를 비교하기 위해 cross-entropy loss 를 사용한다. critic의 아웃풋 값과 Bellman operator 결과의 차이를 학습시킨다.

Implementation

- actor class 는 같은 아키텍쳐

```
class D4PGCritic(nn.Module):
    def __init__(self, obs_size, act_size, n_atoms,
                                    v_min, v_max):
        super(D4PGCritic, self).__init__()

        self.obs_net = nn.Sequential(
            nn.Linear(obs_size, 400),
            nn.ReLU(),
        )
        # output size 가 1이 아닌 n_atoms.
        self.out_net = nn.Sequential(
            nn.Linear(400 + act_size, 300),
            nn.ReLU(),
            nn.Linear(300, n_atoms)
        )
        # q value 의 범위: v_min ~ v_max. delta는 각 구간.
        delta = (v_max - v_min) / (n_atoms - 1)
        self.register_buffer("supports",
                torch.arange(v_min, v_max + delta, delta)
```

```
def forward(self, x, a):
    obs = self.obs_net(x)
    return self.out_net(torch.cat([obs, a], dim=1))

def distr_to_q(self, distr):
    # support buffer 에서 확률 분포를 가져와
    # 평균 Q value 값을 계산한다.
    weights = F.softmax(distr, dim=1) * self.supports
    res = weights.sum(dim=1)
    return res.unsqueeze(dim=-1)
```

```
class AgentD4PG(ptan.agent.BaseAgent):

    def __init__(self, net, device="cpu", epsilon=0.3):
        self.net = net
        self.device = device
        self.epsilon = epsilon

    def __call__(self, states, agent_states):
        states_v = ptan.agent.float32_preprocessor(
                           states).to(self.device)
        mu_v = self.net(states_v)
        actions = mu_v.data.cpu().numpy()
        # Gaussian noise
        actions += self.epsilon *
                np.random.normal(size=actions.shape)
        actions = np.clip(actions, -1, 1)
        return actions, agent_states
```

train critic

```
crt_opt.zero_grad()
crt_distr_v = crt_net(states_v, actions_v)
last_act_v = tgt_act_net.target_model(last_states_v)
last_distr_v = F.softmax(tgt_crt_net.target_model(
                  last_states_v, last_act_v), dim=1)
proj_distr_v = distr_projection(last_distr_v,
                  rewards_v, dones_mask,
                  gamma=GAMMA**REWARD_STEPS, device=device)
# cross entropy loss 함수.
prob_dist_v = -F.log_softmax(crt_distr_v, dim=1) *
                                          proj_distr_v
critic_loss_v = prob_dist_v.sum(dim=1).mean()
critic_loss_v.backward()
crt_opt.step()
```

train actor

```
act_opt.zero_grad()
cur_actions_v = act_net(states_v)
crt_distr_v = crt_net(states_v, cur_actions_v)
# 확률분포를 평균 Q 값으로 바꿈.
actor_loss_v = -crt_net.distr_to_q(crt_distr_v)
actor_loss_v = actor_loss_v.mean()
actor_loss_v.backward()
act_opt.step()

tgt_act_net.alpha_sync(alpha=1 - 1e-3)
tgt_crt_net.alpha_sync(alpha=1 - 1e-3)
```

distr_projection 함수

- Bellman operator를 이용하여 확률 projection

```
def distr_projection(next_distr_v, rewards_v,
                     dones_mask_t, gamma, device="cpu"):
    next_distr = next_distr_v.data.cpu().numpy()
    rewards = rewards_v.data.cpu().numpy()
    dones_mask = dones_mask_t.cpu().numpy().
                                    astype(np.bool)
    batch_size = len(rewards)
    proj_distr = np.zeros((batch_size, N_ATOMS),
                          dtype=np.float32)
```

```
for atom in range(N_ATOMS):
    # Bellman operator 로 계산
    tz_j = np.minimum(Vmax, np.maximum(Vmin,
        rewards + (Vmin + atom * DELTA_Z) * gamma))
    # 위 값의 인덱스 구하기
    b_j = (tz_j - Vmin) / DELTA_Z
    l = np.floor(b_j).astype(np.int64)
    u = np.ceil(b_j).astype(np.int64)
    # 경계에 있다면 해당 atom 에 추가.
    eq_mask = u == l
    proj_distr[eq_mask, l[eq_mask]] +=
                        next_distr[eq_mask, atom]
    # atom 사이에 있다면 비율을 구해 양쪽 atom 에 추가.
    ne_mask = u != l
    proj_distr[ne_mask, l[ne_mask]] +=
        next_distr[ne_mask, atom] * (u - b_j)[ne_mask]
    proj_distr[ne_mask, u[ne_mask]] +=
        next_distr[ne_mask, atom] * (b_j - l)[ne_mask]
```

```
if dones_mask.any():
    proj_distr[dones_mask] = 0.0
    tz_j = np.minimum(Vmax, np.maximum(Vmin,
                                       rewards[dones_mask]))
    b_j = (tz_j - Vmin) / DELTA_Z
    l = np.floor(b_j).astype(np.int64)
    u = np.ceil(b_j).astype(np.int64)
    eq_mask = u == l
    eq_dones = dones_mask.copy()
    eq_dones[dones_mask] = eq_mask
    if eq_dones.any():
        proj_distr[eq_dones, l[eq_mask]] = 1.0
    ne_mask = u != l
    ne_dones = dones_mask.copy()
    ne_dones[dones_mask] = ne_mask
    if ne_dones.any():
        proj_distr[ne_dones, l[ne_mask]] =
                                       (u - b_j)[ne_mask]
        proj_distr[ne_dones, u[ne_mask]] =
                                       (b_j - l)[ne_mask]
return torch.FloatTensor(proj_distr).to(device)
```

Results

- D4PG 가 가장 좋은 결과를 보여준다.
- 수렴속도가 빠르며 reward 또한 높다.