딥 강화학습 Project: A2C with Mujoco

인공지능학과 2023320001 윤준영

1. A2C Network using Mujoco Walker2d

A2C 네트워크는 Advantage Actor-Critic 네트워크로 Actor-Critic을 기반으로 Advantage를 통해 예상 reward와 현재 policy의 value를 이용하여 policy를 업데이트한다. 여러 에이전트가 작동하여 수집한 경험으로 동시에 업데이트를 진행해, 에이전트 간의 경험을 동기화하고 gradient update 시 충돌을 줄여 병렬적으로 학습할 수 있어 빠르고 안정적이다.

1.1. Critic Network

```
class Critic(nn.Module):
    def __init__(self, input_dim, num_heads, hidden_dim, dropout_rate):
        super(Critic, self).__init__()
        self.linear1 = nn.Linear(input_dim, hidden_dim)
        self.encoder_layer = nn.TransformerEncoderLayer(d_model=hidden_dim,
                                                        nhead=num_heads,
                                                        dim_feedforward=512,
                                                        dropout=dropout_rate,
                                                        activation='gelu')
        self.transformer_encoder = nn.TransformerEncoder(self.encoder_layer, num_layers=6)
        self.value_head = nn.Linear(hidden_dim, 1)
    def forward(self, x):
        x = x.unsqueeze(0)  # Add a batch dimension
        x = self.linear1(x)
        x = self.transformer_encoder(x)
        x = x.squeeze(0)  # Remove the batch dimension
        value = self.value_head(x)
        return value
```

Critic Network는 state를 입력받아 value를 평가하는 네트워크이다. Walker2d의 입력 차원인 17차원의 입력을 linear layer를 통해 hidden_dimension 차원인 128차원으로 늘리고, 6개의 layer를 가진 트랜스포머 인코더로 구성하였다. 다시 128차원의 값을 value head linear layer를 통해 1차원으로 줄였다.

1.2. Actor Network

```
class Actor(nn.Module):
    def __init__(self, input_dim, action_dim, num_heads, hidden_dim, dropout_rate):
        super(Actor, self).__init__()
        self.linear1 = nn.Linear(input_dim, hidden_dim)
        self.encoder_layer = nn.TransformerEncoderLayer(d_model=hidden_dim,
                                                        nhead=num_heads,
```

```
                                                    dim_feedforward=512,
                                                    dropout=dropout_rate,
                                                    activation='gelu')
        self.transformer_model = nn.TransformerEncoder(self.encoder_layer, num_layers=6)
        self.mean_head = nn.Linear(hidden_dim, action_dim)
        self.log_std_head = nn.Linear(hidden_dim, action_dim)
    def forward(self, x):
        x = x.unsqueeze(0)  # Add a batch dimension
        x = self.linear1(x)
```

Actor Network는 현재 상태를 통해 다음 step의 행동을 결정하는 네트워크이다. Actor network와 동일하게 linear layer와 트랜스포머 인코더 레이어를 구성한 뒤, 연속적인 값을 가지기 위해 mean head와 log std head를 통해 actuator의 평균과 표준편차를 예측한다. Walker2d의 actuator는 6개이므로 각각의 head는 6개의 output을 가진다.

1.3. Select Action – Continuous Action

Walker2d의 6개의 actuator는 이산적인 값을 가지는 것이 아니라 연속적인 action space를 가진다. 따라서 continuous action을 다루기 위하여 위의 Actor Network에서는 평균과 표준편차를 출력하는 mean head와 log std head를 출력으로 가진다. 하지만 이렇게 sample된 action은 미분이 불가능하여 network의 gradient를 이용하여 학습할 수 없다. 따라서 reparameterization trick을 이용하여 미분이 가능하게 만들어야 한다. 다음과 같이 간단하게 구현할 수 있다.

```
class Actor(nn.Module):
(...)
    def sample_action(self, state):
        mean, std = self(state)
        normal_dist = Normal(mean, std)
        action = normal_dist.rsample()  # Differentiable sampling
        log_prob = normal_dist.log_prob(action)
        entropy = normal_dist.entropy()
        return action, log_prob, entropy
```

1.4. Getting Losses – n-step Return TD error

n-step Return TD error는 에이전트가 현재 상태에서 n-step 행동 후 상태에 도달할 때까지 받는 누적 보상으로 TD error를 계산한다. 특정 step까지의 실제 보상과 그 이후 상태까지의 추정치를 사용하여 갱신하게된다.

$$r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \cdots + \gamma^{n-1} r_{t+n} + \gamma^n V_w(s_{t+n})$$

n-step TD error를 사용하게 되면 Monte-Carlo 방법과 1-step TD error의 장점을 취할 수 있어 단기적인 정보에 의존하지 않고 장기적인 결과를 예측하고 행동하여, 조금 더 멀리 미래의 존재하는 reward를 가질 수 있게 한다. 다음과 같이 구현하였다.

```
    def get_losses(self, rewards: torch.Tensor, action_log_probs: torch.Tensor,
                   value_preds: torch.Tensor, entropy: torch.Tensor, masks: torch.Tensor,
                   gamma: float, n_steps: int, ent_coef: float, device: torch.device,
                   ) -> tuple[torch.Tensor, torch.Tensor]:
        T = len(rewards)
        n_action = action_log_probs.shape[1]
        returns = torch.zeros(T, n_action, self.n_envs, device=device)
        advantages = torch.zeros_like(returns)
        # compute the returns using n-step returns
        for t in range(T - n_steps):
            n_step_return = value_preds[t + n_steps] * (gamma ** n_steps)
            for step in reversed(range(n_steps)):
                n_step_return = (rewards[t + step]
                                 + gamma * n_step_return * masks[t + step])
            returns[t] = n_step_return
            advantages[t] = returns[t] - value_preds[t]
        # calculate the critic loss
        critic_loss = advantages.pow(2).mean()
        # calculate the actor loss
        # using the advantages and including entropy bonus for exploration
        actor_loss = (-(advantages.detach() * action_log_probs).mean()
                      - ent_coef * entropy.mean())
        return critic_loss, actor_loss
```

2. Domain Randomization

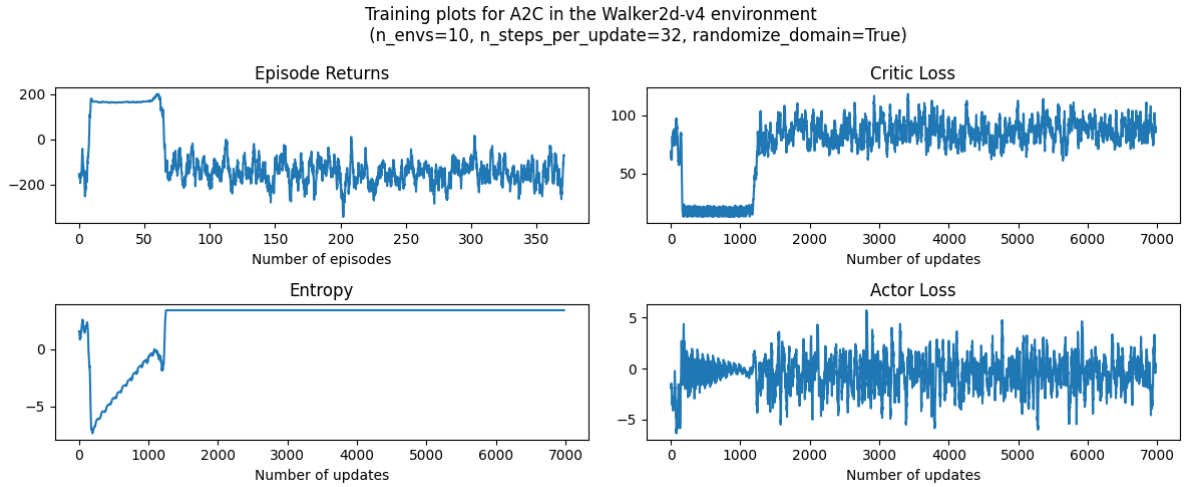
Walker2d 환경에서는 LunarLander와 다르게 바람과 같은 외부 조건들이 존재하지 않기 때문에 domain randomization 방법은 walker가 생성될 때의 초기 속도와 위치에 대한 pertubation을 조절하는 reset_noise_scale밖에 없었다. 그와 별개로 학습 시 사용할 수 있는 reward weight argument가 존재하였기 때문에 forward reward, control cost, healthy reward weight를 변화시켜 학습의 domain을 randomization하였다.

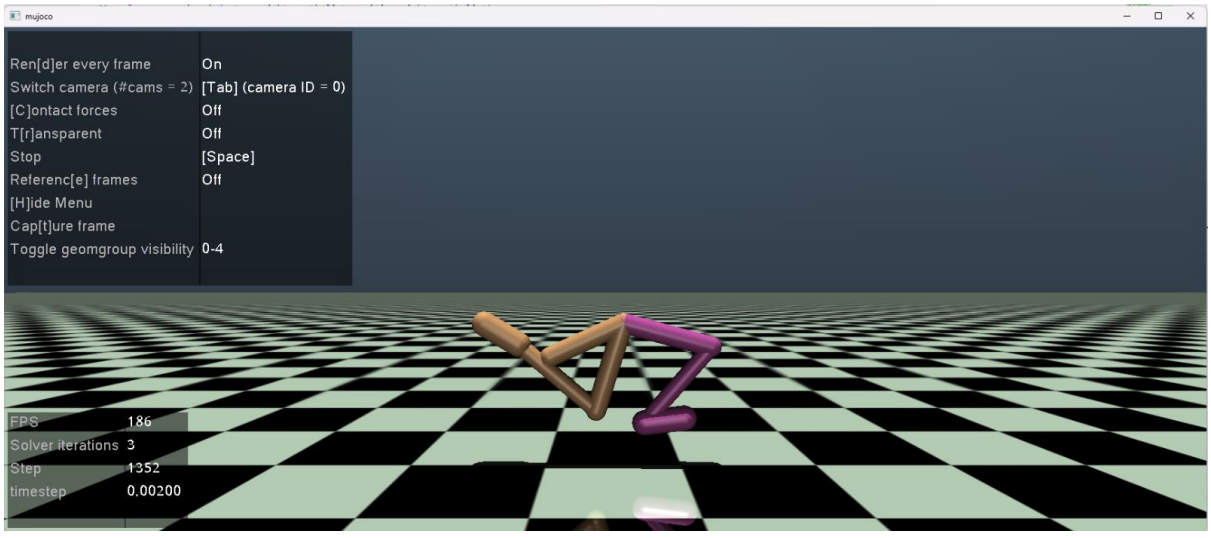
```
    for i in range(n_envs):
        ff.append(np.clip(np.random.normal(loc=1.0, scale=0.5), 0.5, 1.5))
        cc.append(np.clip(np.random.normal(loc=1e-3, scale=5e-4), 5e-4, 1.5e-3))
        hh.append(np.clip(np.random.normal(loc=1.0, scale=0.5), 0.5, 1.5))
        rr.append(np.clip(np.random.normal(loc=5e-3, scale=2e-3), 2e-3,7e-3))
        print(ff[i], cc[i], hh[i], rr[i])
    if randomize_domain:
        envs = gym.vector.AsyncVectorEnv([
            lambda: gym.make("Walker2d-v4",
                             forward_reward_weight=ff[i],
                             ctrl_cost_weight=cc[i]
```

```
                    healthy_reward=hh[i],
                    reset_noise_scale=rr[i],
                    terminate_when_unhealthy=True,
                    max_episode_steps=600)
                    for i in range(n_envs)])
```

3. Result – 1

위 학습 환경으로 실험한 결과는 다음과 같다.





4. Re-training with higher healthy return

실험 결과의 showcase를 살펴 본 결과 넘어진 후에도 계속 발을 이용하여 앞으로 나가는 시도는 좋았으나 결국 반동을 통해 굴러가는 것처럼 보였다. 그래서 이것이 Walker라고 할 수 있을지 의문이 들었다. 따라서 해당 actor와 critic의 weight를 기반으로 healthy reward를 높혀 조금 더 선

상태를 유지하도록 다음과 같이 healthy reward weight를 높혀 학습시켰다.

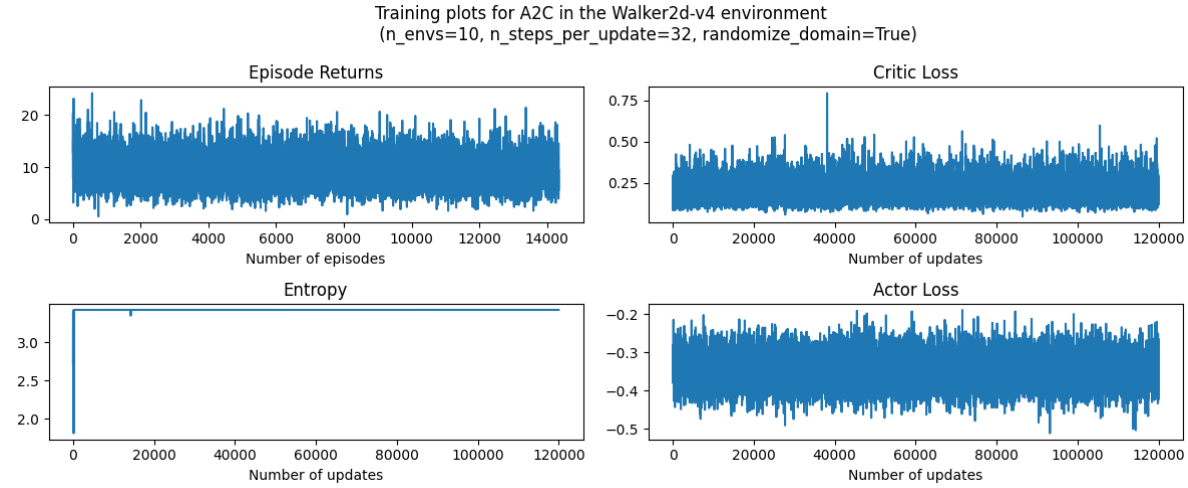
```
    for i in range(n_envs):
        ff.append(np.clip(np.random.normal(loc=1.0, scale=0.5), 0.5, 1.5))
        cc.append(np.clip(np.random.normal(loc=1e-3, scale=5e-4), 5e-4, 1.5e-3))
        hh.append(np.clip(np.random.normal(loc=3.0, scale=0.5), 2.5, 3.5))
        rr.append(np.clip(np.random.normal(loc=5e-3, scale=2e-3), 2e-3,7e-3))
        print(ff[i], cc[i], hh[i], rr[i])
    if randomize_domain:
        envs = gym.vector.AsyncVectorEnv([
            lambda: gym.make("Walker2d-v4",
                             forward_reward_weight=ff[i],
                             ctrl_cost_weight=cc[i],
                             healthy_reward=hh[i],
                             reset_noise_scale=rr[i],
                             terminate_when_unhealthy=True,
                             max_episode_steps=600)
                             for i in range(n_envs)])
```

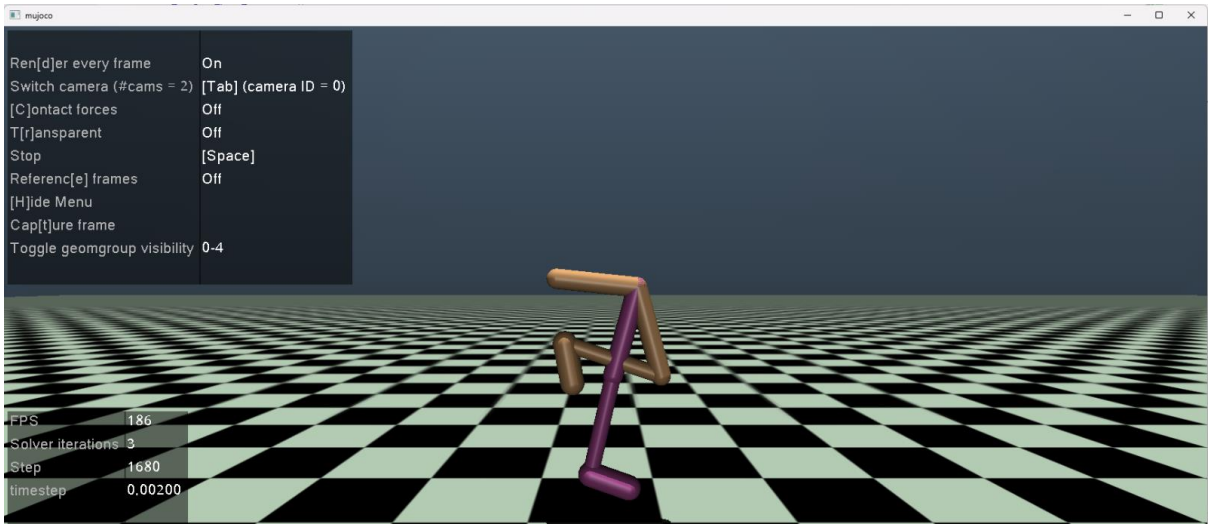
또한 학습 중 unhealthy state가 되었을 때 해당 env는 초기화 전까지 mask를 0으로 주어 넘어진 상태에서는 학습이 되지 않게끔 설정하였다.

```
    for sample_phase in tqdm(range(n_updates)):
(...)
        masks = torch.zeros(n_steps_per_update, n_envs, device=device)
        for step in range(n_steps_per_update):
(...)
            # for each env the mask is 1 if the episode is ongoing
            # and 0 if it is terminated (not by truncation!)
            if step == 0:
                masks[step] = torch.tensor([not term for term in terminated])
            else:
                masks[step] = masks[step - 1] * torch.tensor(
                    [not term for term in terminated])
            if masks[step].sum() == 0:
                break
```

5. Result – 2

위 학습 환경으로 실험한 결과는 다음과 같다.





실험 1과 비슷하게 앞으로 구르는 느낌으로 앞으로 나가는 현상을 보였다. 하지만 실험 1과는 다르게 종종 구르는 동안 자세가 세워졌을 때 잠깐 중심을 잡으려고 시도하는 것을 확인할 수 있었다. 결과적으로 reward에서 거리의 비중이 커졌을 때에는 앞으로 가는 속도가 더 느려져 실험 1의 reward가 더 높았고, healthy state의 비중이 커졌을 때에는 실험 2의 reward가 더 높았다.