HW3 A3C 알고리즘 구현

InvertedPendulumSwingBulletEnv-v0

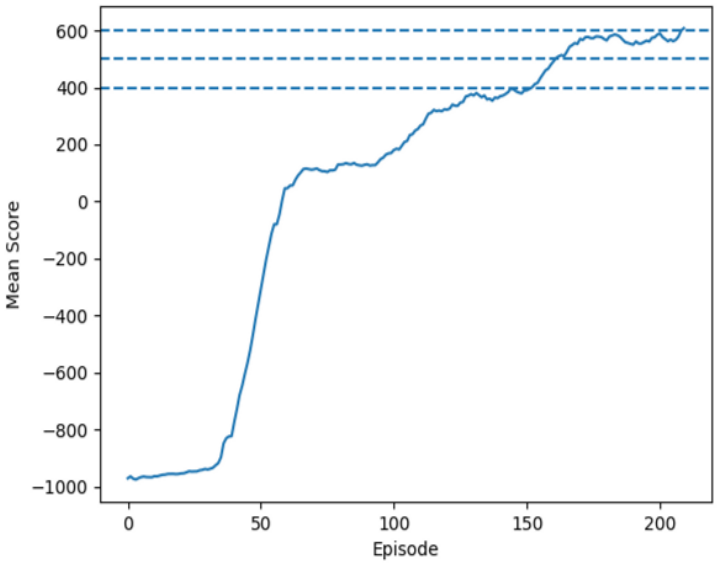
2023710158 박경빈

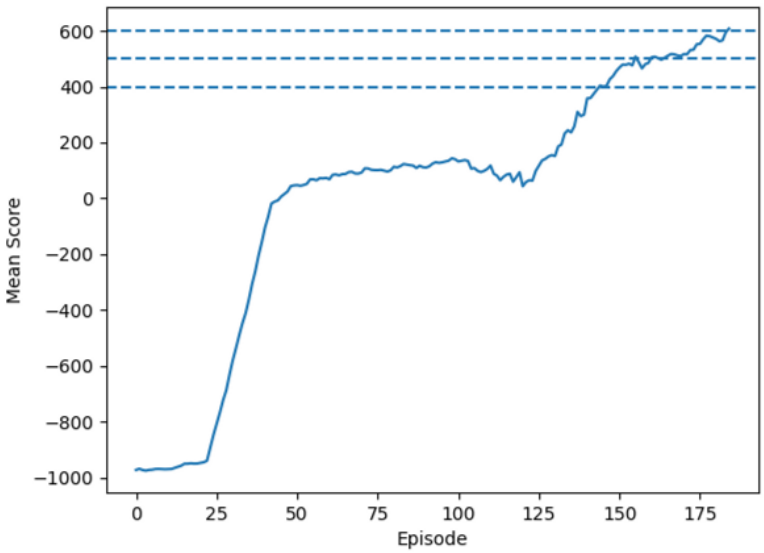
1. 학습환경

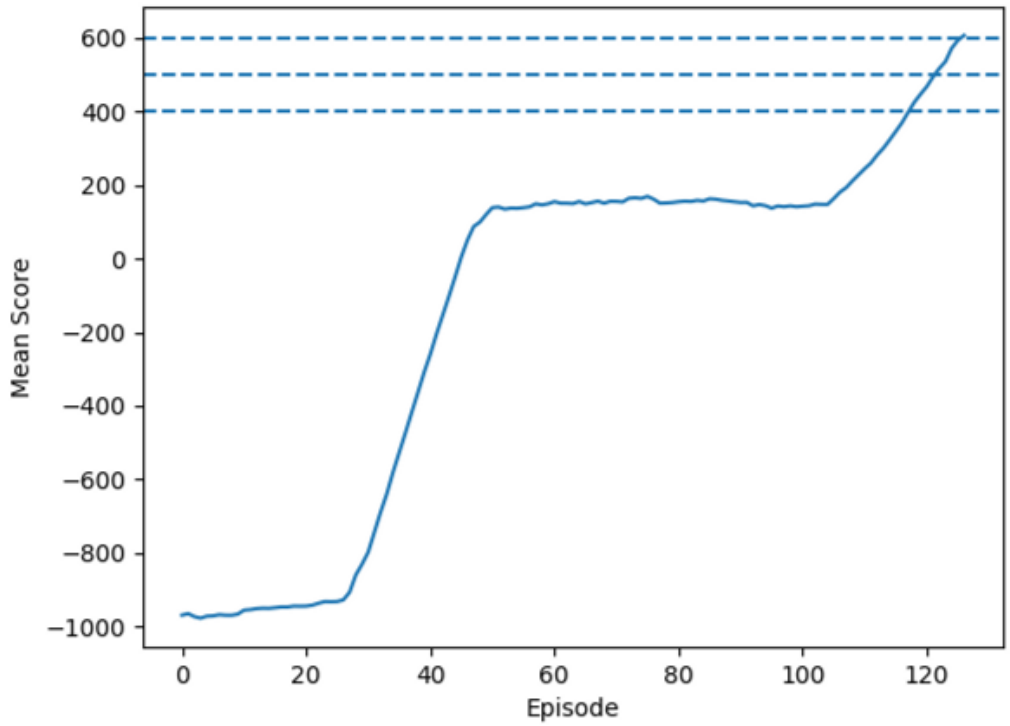
- **python==3.9.16**
- **pytorch==1.8.1**
- **numpy==1.24.3**
- **matplotlib==3.7.1**
- **gym==0.22.0**
- **pybullet==3.2.5**

2. 학습 결과

- **N step은 16으로, Multiprocess는 2개로 진행하였고 3번의 학습 결과는 다음과 같습니다.**







총 3번의 학습결과 모두 episode 200정도안에 score가 600점에 도달하는 것을 확인할 수 있었다. (N_Step = 16, Multiprocess = 2)

3. Hyperparameters

Parameters	**Value**
Learning Rate	0.00045
gamma	0.95
entropy_coef	0.5
N step	16
Multiprocess	2
Number of Hidden Layers	2
Number of Hidden Layers Unit	32
Activation Function	Tanh
Gradient Norm	0.5

4. 구현 설명

4-1 Class NstepBuffer

- NstepBuffer는 다음과 같이 init에서 빈 리스트를 만들고 add에서 append를 통해 list에 저장한다. sample에서는 저장된 것을 불러오고 reset은 다시 빈 list로 만들면서 reset을 진행한다. 코드는 다음과 같다.

```
class NstepBuffer:
    '''
    Save n-step transitions to buffer
    '''
    def __init__(self):
        self.states = []
        self.actions = []
        self.rewards = []
        self.next_states = []
        self.dones = []

    def add(self, state, action, reward, next_state, done):
        '''
        Add a sample to the buffer
        '''
        self.states.append(state)
        self.actions.append(action)
        self.rewards.append(reward)
        self.next_states.append(next_state)
        self.dones.append(done)

    def sample(self):
        '''
        Sample transitions from the buffer
        '''
        return self.states, self.actions, self.rewards, self.next_states, self.dones

    def reset(self):
        '''
        Reset the buffer
        '''
        self.states = []
        self.actions = []
        self.rewards = []
        self.next_states = []
        self.dones = []
```

4-2 Class ActorCritic

- hidden layer는 총 2개로 구성하고 input은 OBS_DIM(5), output은 ACT_DIM(1)으로 설정한다. 또한 환경이 간단하다고 생각하여 hidden layer의 unit은 32로 설정하였다. Actor와 Critic은 Tanh를 사용하였다. Actor에서는 Action을 선정하기 위한

평균과 표준편차를 출력하였고 Critic에서는 Value값을 출력하였다. 이때 표준편차는 F.softplus함수를 사용하여 계산하였으며 값이 0에 가까워지면 로그함수의 입력으로 들어갈 때 문제가 발생할 수 있습니다. 따라서 작은 양수 값인 0.001을 더해주어 표준편차가 0이 되지 않도록 보정을 하였습니다. 코드는 다음과 같습니다.

```
class ActorCritic(nn.Module):
    def __init__(self):
        super(ActorCritic, self).__init__()
        self.a1 = nn.Linear(OBS_DIM, 32)
        self.a2 = nn.Linear(32, 32)
        self.mu = nn.Linear(32, ACT_DIM)
        self.sigma = nn.Linear(32, ACT_DIM)
        self.c1 = nn.Linear(OBS_DIM, 32)
        self.c2 = nn.Linear(32, 32)
        self.v = nn.Linear(32, ACT_DIM)
        self.set_init([self.a1, self.mu, self.sigma, self.c1, self.v])
        self.distribution = torch.distributions.Normal

    def set_init(self, layers):
        for layer in layers:
            nn.init.normal_(layer.weight, mean=0., std=0.1)
            nn.init.constant_(layer.bias, 0.)

    def actor(self, states):
        a1 = torch.tanh(self.a1(states))
        a2 = torch.tanh(self.a2(a1))
        mean = self.mu(a2)
        std = F.softplus(self.sigma(a2)) + 0.001    #to avoid 0
        return mean, std

    def critic(self, states):
        c1 = torch.tanh(self.c1(states))
        c2 = torch.tanh(self.c2(c1))
        values = self.v(c2)
        return values
```

4-3 Class Worker

1) select_action

이 부분에서는 action을 선택하는 함수를 작성하였습니다. actor에서 mean과 std값을 구하게 되고 이를 Normal함수를 사용한 후 sample()을 통해 action을 선택합니다. 코드는 다음과 같습니다.

```
def select_action(self, state):
    '''
    selects action given state

    return:
        continuous action value
    '''
    state = torch.tensor(state, dtype=torch.float32)
    mean, std = self.local_actor.actor(state)
    action_dist = Normal(mean, std)
    action = action_dist.sample()
    return action
```

2) train_network

위에서 작성한 critic함수를 이용하여 next_state_value를 계산한 후 이를 이용해 target_value를 계산한 후 현재 value 값과 계산하여 advantage를 계산하였습니다. 앞에 actor함수와 같이 Normal함수를 이용하여 normal distribution을 생성합니다. 이후 주어진 actions에 대한 log_probs를 계산한 후 advantage와 같이 actor_loss를 계산합니다. 또한 critic_values를 구하고 mse_loss를 이용하여 critic_loss를 계산합니다. Action 분포의 엔트로피를 평균하여 entropy_loss를 계산하며 entropy는 분포의 불확실성을 나타내는 척도로서, 다양한 action을 탐색하도록 도움을 줍니다. 모두 계산을 완료했으면 total_loss를 계산한 후 학습의 안정성을 위해서 Gradient를 0.5이내로 제한하였습니다. 코드는 다음과 같습니다.

```
# Compute target values
with torch.no_grad():
    next_state_value = self.local_actor.critic(next_states)
    target_values = rewards + self.gamma * (1 - dones) * next_state_value
advantages = target_values - self.local_actor.critic(states)

# Actor loss
mean, std = self.local_actor.actor(states)
dist = Normal(mean, std)
log_probs = dist.log_prob(actions.unsqueeze(1))
actor_loss = -(log_probs * advantages.detach()).mean()

# Critic loss
critic_values = self.local_actor.critic(states)
critic_loss = F.mse_loss(critic_values, target_values.detach())

# Entropy loss
entropy_loss = (dist.entropy().mean())
total_loss = actor_loss + 0.5*critic_loss - self.entropy_coef * entropy_loss
nn.utils.clip_grad_norm_(self.global_actor.parameters(), 0.5)
```

```python
def train(self):
    step = 1

    while True:
        state = self.env.reset()
        done = False

        while not done:
            action = self.select_action(state)
            next_state, reward, done, _ = self.env.step(action)
            self.nstep_buffer.add(state, action.item(), reward, next_state, done)
            # n step에 도달하거나 에피소드가 종료된 경우 학습 진행
            if step % self.n_step == 0 or done:
                self.train_network(*self.nstep_buffer.sample())
                self.nstep_buffer.reset()
            state = next_state
            step += 1
```

수많은 시도 끝에 hyperparameter를 tuning을 하였으며 multiprocess를 4로 설정한 이유는 multiprocess 개수와 학습의 속도는 linear한 관계를 가진다고 하였기 때문에 최대인 4로 설정하였습니다. 또한 score가 300이상 올라갔다가 다시 -800까지 떨어지고 다시 올라가고 내려오기를 반복하였는데 이때 gradient를 제한하는 것이 학습과정에서 매우 중요했습니다. Gradient의 큰 변화는 policy에 불안정성을 야기했고 이것이 학습의 불안정한 결과를 가져오게 되었습니다. 하지만 gradinet를 0.5이내로 제한하면서 policy 변화를 억제하였고 그 결과로 안정적인 학습을 할 수 있게 되었습니다. 또한 환경이 간단하기 때문에 layer를 깊게 쌓지 않아야 하였으며 layer unit도 크게 설정하지 않고 32로 설정하면서 좋은 결과를 얻어낼 수 있었습니다.