A3C implementation

Notification : 실험하는 과정에서 Evaluation Process가 너무 빠르게 끝나버림, 이에 따라 학습 상황을 제대로 파악하지 못하는 일이 생기는거 같음

1. 환경

```
ENV = gym.make("InvertedPendulumSwingupBulletEnv-v0")
OBS_DIM = ENV.observation_space.shape[0]
ACT_DIM = ENV.action_space.shape[0]
ACT_LIMIT = ENV.action_space.high[0]
ENV.close()
GAMMA = 0.95
```

이 과제에서는 InvertedPendulumSwingupBulletEnv-v0 환경에 대해 실험을 진행했다.
state는 5 가지 값으로 구성되어 있고, 환경에 가해져야 하는 action의 수는 1가지이다.
action 값의 범위는 -1~1 사이 값을 유지하도록 한다.

2. 기본 Hyperparameter

```
GAMMA = 0.95
```

```
if n_steps < 3:
    optimizer = optim.Adam(local_actor.parameters(), lr=3e-5, betas=(0.95, 0.999))
else:
    optimizer = optim.Adam(local_actor.parameters(), lr=4e-4, betas=(0.95, 0.999))
```

gamma 값은 0.95로 설정했고, optimizer는 adam optimizer로 선정했다. 그리고 learning rate는 1~3 step일 때는 3e-5로, 그 외의 n-step일 때는 4e-4로 learning rate을 조금 변경했고, beta는 0.95, 0.999로 설정했다. 여기서 여러 가지 learning rate를 실험해본 결과 평균적으로 3e-5에서 좋은 성능이 도출됐다.

3. Network 구성

```
class ActorCritic(nn.Module):
    def __init__(self):
        super(ActorCritic, self).__init__()

        self.relu = nn.ReLU()
        self.layer1 = nn.Linear(OBS_DIM, 512)
        self.layer2 = nn.Linear(512, 256)

        self.mu_layer = nn.Linear(256, ACT_DIM)
        self.log_std_layer = nn.Linear(256, ACT_DIM)
        self.tanh = nn.Tanh()
        self.softplu = nn.Softplus()

        self.crilayer = nn.Linear(256, 1)

    def act(self, x):
        x = self.relu(self.layer1(x))
        x = self.relu(self.layer2(x))
        mu = 2 * self.tanh(self.mu_layer(x))
        mu = torch.clamp(mu, -1.0, 1.0)
        std = self.softplu(self.log_std_layer(x)) + 1e-5

        return mu, std

    def cri(self, x):
        x = self.relu(self.layer1(x))
        x = self.relu(self.layer2(x))
        out = self.crilayer(x)
        return out
```

우선, actor-critic 네트워크는 act와 cri 총 2가지로 구성되어 있다.
act와 cri는 layer1과 layer2를 공유하며, layer1은 (5, 256), layer2는 (256, 128)로 input output이 설정되어 있고 relu로 output을 조정해준다.
act 부분에서는 히든 레이어는 위와 같고, 최종 output은 2가지를 return 한다. 하나는 action의 평균은 도출하고, 하나는 action의 분산을 추출하는 layer이다. 최종 return 값은 mu와 std로 return된다.
cri 부분에서는 히든 레이어는 위와 같고, 최종 ouput은 action에 대한 value 1가지를 return한다. 이것은 간단한 layer기에 설명을 넘기겠다.

3. Actor Critic (Single Step, Single Process) + (Multi Step, Single Process) + (Multi Process)

```
for train_episode in range(3000):
    ## update the weigth from global network
    local_actor.load_state_dict(global_actor.state_dict())

    ## start the episode
    state = env.reset()

    ## Episode total score
    score = 0

    ## Set memory
    memory = deque()
```

한 에피소드를 시작하기 앞서, 글로벌 actor로부터 weight를 모두 받아온다
그리고, 초기 state를 받고 n-step learning에서 사용할 memory를 reset해준다.

```
for step in range(10001):
    ## ========================
    ## Here is one step
    ## ========================
    state = torch.FloatTensor(state)

    ## Get learnable features
    mu, std = local_actor.act(state)
    normal = Normal(mu.view(1, ).data, std.view(1, ).data)
    action = normal.sample().numpy()

    torchAction = torch.FloatTensor(action)
    act = Variable(torchAction)
    prob = getProb(act, mu, std)

    entropy = 0.5 * ((std * 2 * pi.expand_as(std)).log()+1)

    log_prob = (prob + 1e-6).log()

    ## move one action
    action = action.clip(-1, 1)
    next_state, reward, done, _ = env.step(action)
    next_state = torch.FloatTensor(next_state)
```

이 부분은 매 step 마다 실행되는 것이다.
우선 현재 state를 기반으로 mu와 std를 계산하고 mu와 std를 기반으로 분포를 도출한다.
이 분포를 기반으로 action을 선정하며, action은 -1과 1 사이의 값으로 cliping된다. 선정한 action을 기반으로 log probality를 계산하고, std로 entropy를 계산한다.
이후, 선정한 action을 환경에 가하고 next state, reward, done에 대한 정보를 얻는다.

```
## Add memory for n-step
memory.append([state, next_state, action, reward, log_prob, entropy])

if done : break

score += reward
```

해당 step이 종료됐다면, 이 step에서 얻은 정보를 memory에 저장한다.
만약 해당 step으로 인해 게임이 종료될 경우 새로운 episode를 위해 탈출한다.
그리고 score는 현재 episode에서 얻은 reward를 저장한다.

```
actor_loss = 0
critic_loss = 0
temp = torch.zeros(1, 1)
lastValue = local_actor.cri(next_state)
R = Variable(lastValue)

## Here start learning
if len(memory) == n_steps:
    for i in reversed(range(n_steps)):
        R = GAMMA * R + memory[i][3]
        advantage = R - local_actor.cri(memory[i][0])
        critic_loss += 0.5 * advantage.pow(2)

        deltaT = memory[i][3] + GAMMA * \
                 local_actor.cri(memory[i][1]).data - local_actor.cri(memory[i][0]).data

        temp = temp * GAMMA + deltaT

        actor_loss = actor_loss - \
                     (memory[i][4].sum() * Variable(temp)) - \
                     (0.01 * memory[i][5].sum())

    ## Train
    optimizer.zero_grad()
    (actor_loss + 0.5 * critic_loss).backward()
    optimizer.step()

    ## Out one memory
    memory.clear()
    #memory.popleft()
```

만약 현재 memory에 저장된 memory의 수가 n_step의 수와 같은 경우, actor loss와 critic loss를 계산해 모델 weight를 업데이트 해준다. 여기서, critic 부분을 업데이트할 때에는 advantage를 계산하는 방식을 사용했고, actor loss를 계산할 때에는 기존 SARSA와 유사한 방식을 사용해 loss를 계산했지만, exploration을 위해 entropy를 조금 추가했다. 한 차례의 learning을 마쳤다면, memory는 모두 지워진다.

```
if n_steps > 0 and len(memory) > 0:
    actor_loss = 0
    critic_loss = 0
    temp = torch.zeros(1, 1)
    lastValue = local_actor.cri(memory[-1][1])
    R = Variable(lastValue)

    for i in reversed(range(len(memory))):
        R = GAMMA * R + memory[i][3]
        advantage = R - local_actor.cri(memory[i][0])
        critic_loss += 0.5 * advantage.pow(2)

        deltaT = memory[i][3] + GAMMA * \
                 local_actor.cri(memory[i][1]).data - local_actor.cri(memory[i][0]).data

        temp = temp * GAMMA + deltaT

        actor_loss = actor_loss - \
                     (memory[i][4].sum() * Variable(temp)) - \
                     (0.01 * memory[i][5].sum())

    ## Train
    optimizer.zero_grad()
    (actor_loss + 0.5 * critic_loss).backward()
    optimizer.step()

    ## Out one memory
    memory.clear()
```

n-step learning이 중간에 끝났을 경우, 남아있는 memory가 있을 수 있다. 따라서, 남아있는 memory 만을 사용해 n-step learning을 실시한다. 모든 learning이 끝난 이후 memory는 비워준다.
(actor_loss는 온전히 반영하지만, critic_loss는 반만 반영한다. 그 이유는 actor에 더 많은 중점을 두기 위해서이다.)

```
        ## ===============
        ## Here one episode END
        ## ===============
        state = next_state

    print("Episode {0}, Total Score : {1}".format(train_episode, score))
    global_actor.load_state_dict(local_actor.state_dict())
```

한 step이 완전하게 끝났다면, next state를 현재 state로 변경해준다.
그리고 episode가 끝났다면, 해당 episode에서 얻은 reward를 보여주고, 향후 multiprocess를 위해 update하는 느낌으로 코드를 작성해놨다.

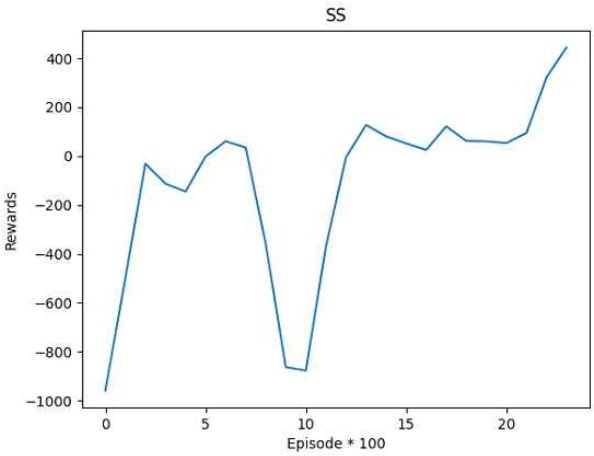
```
def update(model, global_model):
    for param, global_param in zip(model.parameters(), global_model.parameters()):
        if global_param.grad is not None:
            return
        else:
            global_param._grad = param.grad
```

이 함수를 사용해 global actor에 gradient를 share하는 효과를 냈다.

4. Result (Single step, Single process)

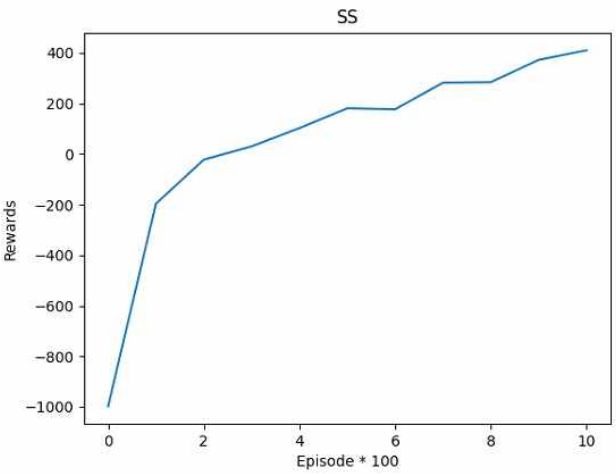
Experiment 1 :

```
Episode 180, Total Score : 265.4516825420919
Episode: 2400, avg score: 444.5
Solved (1)!!!, Time : 1138.60
Episode 181, Total Score : 430.8628717544072
```



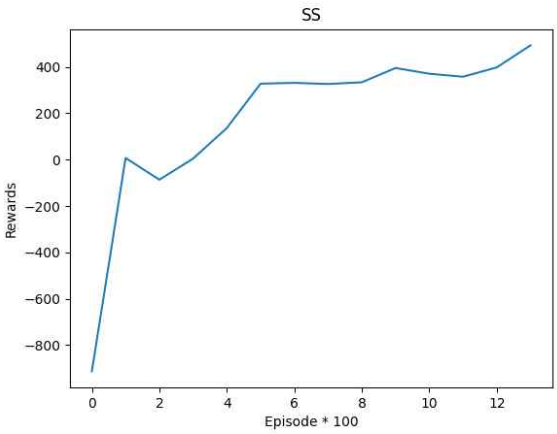
Experiment 2 :

```
Episode 149, Total Score : 425.62484919026303
Episode 150, Total Score : 110.21947616551581
Episode 151, Total Score : 493.5188780698327
Episode 152, Total Score : 241.12549468037506
Episode 153, Total Score : 693.7592451022194
Episode: 1100, avg score: 410.9
Solved (1)!!!, Time : 671.57
Episode 154, Total Score : 517.3300962075532
Episode 155, Total Score : 144.3335023690773
```



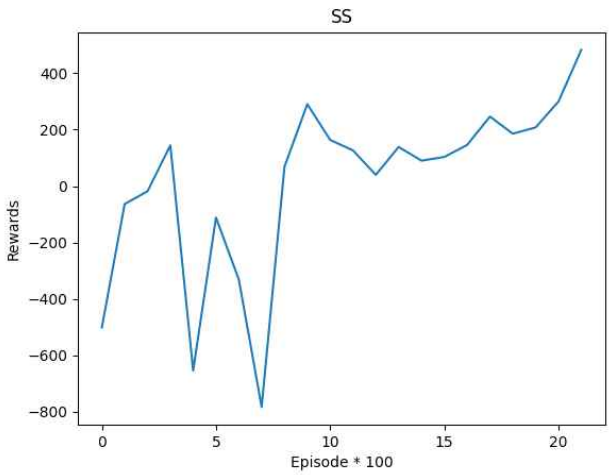
Experiment 3 :

```
Episode 196, Total Score : 702.0359593664483
Episode 197, Total Score : 234.8345674581921
Episode: 1400, avg score: 493.3
Solved (1)!!!, Time : 2263.66
Episode 198, Total Score : 398.06260891465166
Episode 199, Total Score : 781.0871810674469
Episode 200, Total Score : 399.01829659874767
Episode 201, Total Score : 829.0054408492393
```



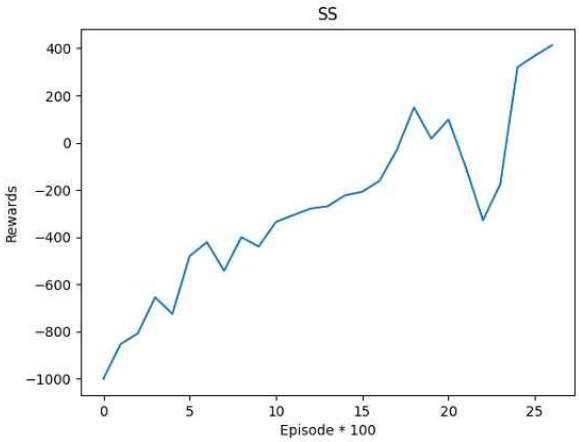
Experiment 4 :

```
Episode 298, Total Score : 393.5376879204047
Episode 299, Total Score : 369.2306587014709
Episode 300, Total Score : 693.181248264286
Episode: 2200, avg score: 483.3
Solved (1)!!!, Time : 3594.08
Episode 301, Total Score : 642.4454373922155
Episode 302, Total Score : 585.1630039707735
Episode 303, Total Score : 887.8938253705015
Episode 304, Total Score : 775.6664722909878
Episode 305, Total Score : 888.3954347903334
```



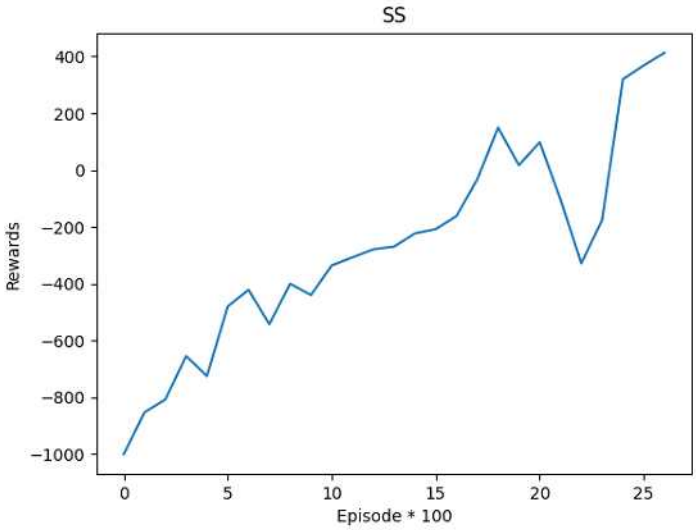
Experiment 5 :

```
Episode 372, Total Score : 281.9250395351068
Episode 373, Total Score : 299.51611610541534
Episode 374, Total Score : 362.4352666856126
Episode: 2700, avg score: 412.9
Solved (1)!!!, Time : 1617.49
Episode 375, Total Score : 475.85440183582483
Episode 376, Total Score : 476.629719441238
Episode 377, Total Score : 400.3031660110068
Episode 378, Total Score : 457.76021837993267
```



Experiment 6 :

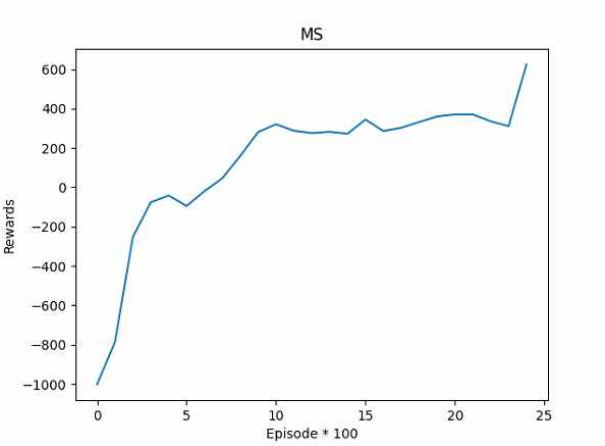
```
Episode 296, Total Score : 747.6115234130418
Episode 297, Total Score : 832.8503401107438
Episode 298, Total Score : 744.799085639609
Episode 299, Total Score : 241.26720065089998
Episode 300, Total Score : 644.96736657869
Episode: 2100, avg score: 434.2
Solved (1)!!!, Time : 1265.20
Episode 301, Total Score : 229.26795031458445
Episode 302, Total Score : 452.68469912426764
```



5. Result (Multi step, Single process)

n-step -> 8

```
Episode 330, Total Score : 883.2939843066056
Episode 331, Total Score : 645.871126551315
Episode 332, Total Score : 884.1043799794235
Episode 333, Total Score : 886.6860382659935
Episode 334, Total Score : 884.1801071798836
Episode: 2500, avg score: 624.7
Solved (2)!!!, Time : 1471.09
Episode 335, Total Score : 885.0276714943379
Episode 336, Total Score : 738.8638754807448
```



n-step -> 16
실험 하는 과정에서 과제 제출 전에 400은 일단 넘었다. 아마 이 추세면 향후 값이 500 이상을 도달할 것 같다.

실험을 추가로 진행하고 싶었으나, 현재 가지고 있는 하드웨어 자원이 충분하지 못해서 많이 하지 못했다.

6. Result (Multi step, Multi Process)

이 부분은 시간이 충분하지 않아 실험을 진행하지 못했다. multi-process 부분에 대한 코드는 많지 않았기에 어느정도 성능이 나올 것이라 생각이 된다.