

HW#1

- 코드 환경: 운영체제 (윈도우 10), 패키지(Python 3.8.5, Tensorflow 2.4.1)

[참고] <https://github.com/pasus/Reinforcement-Learning-Book-Revision>

- 테스트 환경은 OpenAI Gym의 Pendulum-v0임

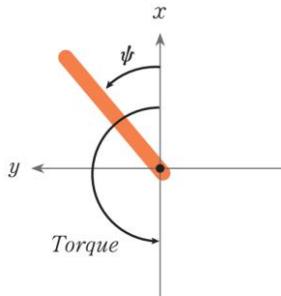


그림 4.9 Pendulum-v0

- 에이전트의 목표는 길이가 1인 진자를 위로 수직으로 세워서 오래 유지하는 것임
- 행동인 조인트 코크의 크기가 크지 않기 때문에 토크만으로 진자를 수직으로 세울 수가 없어서 진자를 흔들면서 중력의 도움을 받아야함.
- 에이전트가 측정할 수 있는 파라미터는 수직축 좌표인 $\cos \psi$, 수평축 좌표인 $\sin \psi$, 각속도인 $\dot{\psi}$ 등 3개임.
- ψ 는 진자와 수직축이 이루는 각도이며 $-\pi \leq \psi \leq \pi$ 의 범위임. 진자가 위로 수직일 때 $\psi = 0$
- 각속도는 $-8 \leq \dot{\psi} \leq 8$ 의 범위를 갖음. 행동은 $-2 \leq a \leq 2$ 의 범위를 갖음.
- 보상은 다음과 같음:

$$r = -\psi^2 - 0.1\dot{\psi}^2 - 0.001a^2 \quad [4.27]$$

- 한 시간스텝에서 받을 수 있는 보상의 최솟값은 -16.2736, 최댓값은 0임.
- 일정 시간스텝이 경과하면 에피소드가 자동으로 종결됨.

1. 참고 A2C 코드는 액터 신경망과 크리틱 신경망을 두 개로 나누어서 구현하였다.

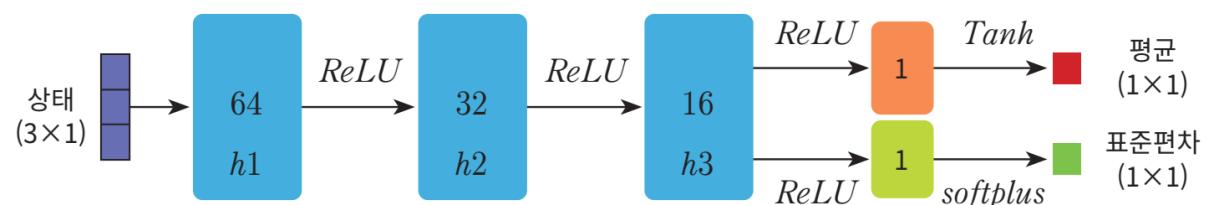


그림 4.11 액터 신경망 구조

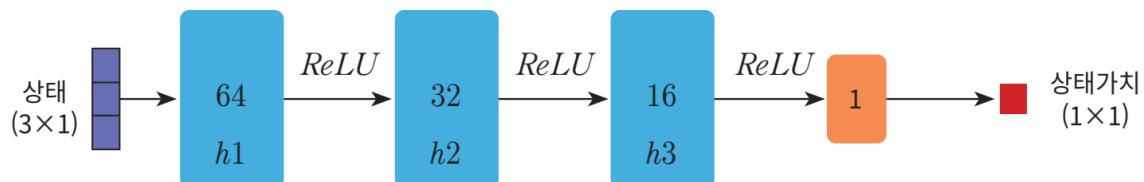


그림 4.13 크리틱 신경망 구조

a). 중간층이 64-32-16으로 되어 있는 데 실험을 통해서 최적구조를 찾아보세요 (성능비교 테이블과 x축은 에피소드 y축이 보상인 학습곡선을 그려서 비교분석하세요.).

b). a)에서 찾은 최적구조를 아래와 같이 합쳐진 정책 신경망과 가치 신경망의 형태로 변경하여 성능을 비교하세요.

x축은 에피소드 y축이 보상인 학습곡선을 그려서 비교하세요.

c). 변경된 코드도 제출하세요.

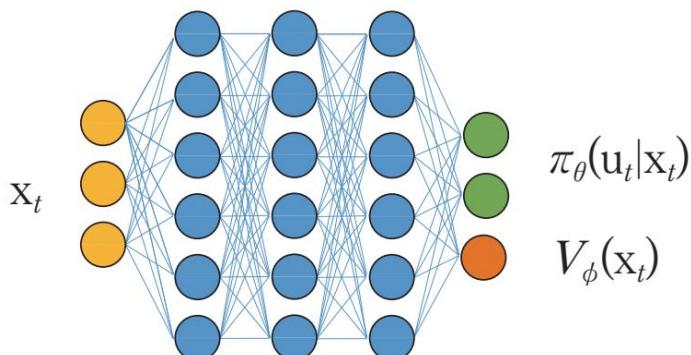


그림 4.4 합쳐진 정책 신경망과 가치 신경망

■ 코드개요:

참고 A2C 코드는 액터-크리틱 신경망을 구현하고 학습시키기 위한 `a2c_learn.py`, 이를 실행시키기 위한 `a2c_main.py`, 그리고 학습을 마친 신경망 파라미터를 읽어와 에이전트를 구동하기 위한 `a2c_load_play.py`로 구성됨

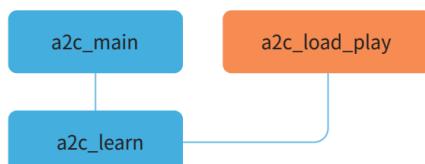


그림 4.10 A2C 코드 구조

a2c_learn.py

```
# A2C learn (tf2 subclassing API version)
# coded by St.Watermelon

# 필요한 패키지 임포트
import tensorflow as tf

from tensorflow.keras.models import Model
from tensorflow.keras.layers import Dense, Lambda
from tensorflow.keras.optimizers import Adam

import numpy as np
import matplotlib.pyplot as plt

## A2C 액터 신경망
class Actor(Model):

    def __init__(self, action_dim, action_bound):
        super(Actor, self).__init__()
        self.action_bound = action_bound

        self.h1 = Dense(64, activation='relu')
        self.h2 = Dense(32, activation='relu')
        self.h3 = Dense(16, activation='relu')
        self.mu = Dense(action_dim, activation='tanh')
        self.std = Dense(action_dim, activation='softplus')

    def call(self, state):
        x = self.h1(state)
        x = self.h2(x)
        x = self.h3(x)
        mu = self.mu(x)
        std = self.std(x)

        # 평균값을 [-action_bound, action_bound] 범위로 조정
        mu = Lambda(lambda x: x*self.action_bound)(mu)

    return [mu, std]
```

```

## A2C 크리틱 신경망
class Critic(Model):

    def __init__(self):
        super(Critic, self).__init__()

        self.h1 = Dense(64, activation='relu')
        self.h2 = Dense(32, activation='relu')
        self.h3 = Dense(16, activation='relu')
        self.v = Dense(1, activation='linear')

    def call(self, state):
        x = self.h1(state)
        x = self.h2(x)

        x = self.h3(x)
        v = self.v(x)
        return v


## A2C 에이전트 클래스
class A2Cagent(object):

    def __init__(self, env):

        # 하이퍼파라미터
        self.GAMMA = 0.95
        self.BATCH_SIZE = 32
        self.ACTOR_LEARNING_RATE = 0.0001
        self.CRITIC_LEARNING_RATE = 0.001

        # 환경
        self.env = env
        # 상태변수 차원
        self.state_dim = env.observation_space.shape[0]
        # 행동 차원
        self.action_dim = env.action_space.shape[0]
        # 행동의 최대 크기
        self.action_bound = env.action_space.high[0]
        # 표준편차의 최솟값과 최댓값 설정
        self.std_bound = [1e-2, 1.0]

```

```

# 액터 신경망 및 크리틱 신경망 생성
self.actor = Actor(self.action_dim, self.action_bound)
self.critic = Critic()
self.actor.build(input_shape=(None, self.state_dim))
self.critic.build(input_shape=(None, self.state_dim))

self.actor.summary()
self.critic.summary()

# 옵티마이저 설정
self.actor_opt = Adam(self.ACTOR_LEARNING_RATE)
self.critic_opt = Adam(self.CRITIC_LEARNING_RATE)

# 에피소드에서 얻은 총 보상값을 저장하기 위한 변수
self.save_epi_reward = []

## 로그-정책 확률밀도함수
def log_pdf(self, mu, std, action):
    std = tf.clip_by_value(std, self.std_bound[0], self.std_bound[1])
    var = std ** 2
    log_policy_pdf = -0.5 * (action - mu) ** 2 / var - 0.5 * tf.math.log(var * 2 * np.pi)
    return tf.reduce_sum(log_policy_pdf, 1, keepdims=True)

## 액터 신경망에서 행동 샘플링
def get_action(self, state):
    mu_a, std_a = self.actor(state)
    mu_a = mu_a.numpy()[0]
    std_a = std_a.numpy()[0]
    std_a = np.clip(std_a, self.std_bound[0], self.std_bound[1])
    action = np.random.normal(mu_a, std_a, size=self.action_dim)
    return action

## 액터 신경망 학습
def actor_learn(self, states, actions, advantages):
    with tf.GradientTape() as tape:

```

```

# 정책 확률밀도함수
mu_a, std_a = self.actor(states, training=True)
log_policy_pdf = self.log_pdf(mu_a, std_a, actions)

# 손실함수
loss_policy = log_policy_pdf * advantages
loss = tf.reduce_sum(-loss_policy)

# 그래디언트
grads = tape.gradient(loss, self.actor.trainable_variables)
self.actor_opt.apply_gradients(zip(grads, self.actor.trainable_variables))

## 크리틱 신경망 학습
def critic_learn(self, states, td_targets):
    with tf.GradientTape() as tape:
        td_hat = self.critic(states, training=True)
        loss = tf.reduce_mean(tf.square(td_targets-td_hat))

        grads = tape.gradient(loss, self.critic.trainable_variables)
        self.critic_opt.apply_gradients(zip(grads, self.critic.trainable_variables))

## 시간차 타깃 계산
def td_target(self, rewards, next_v_values, dones):
    y_i = np.zeros(next_v_values.shape)
    for i in range(next_v_values.shape[0]):
        if dones[i]:
            y_i[i] = rewards[i]
        else:
            y_i[i] = rewards[i] + self.GAMMA * next_v_values[i]
    return y_i

## 신경망 파라미터 로드
def load_weights(self, path):
    self.actor.load_weights(path + 'pendulum_actor.h5')
    self.critic.load_weights(path + 'pendulum_critic.h5')

## 배치에 저장된 데이터 추출
def unpack_batch(self, batch):

```

```

unpack = batch[0]
for idx in range(len(batch)-1):
    unpack = np.append(unpack, batch[idx+1], axis=0)

return unpack

## 에이전트 학습
def train(self, max_episode_num):

    # 에피소드마다 다음을 반복
    for ep in range(int(max_episode_num)):

        # 배치 초기화
        batch_state, batch_action, batch_reward, batch_next_state, batch_done = [], [], [], [], []
        # 에피소드 초기화
        time, episode_reward, done = 0, 0, False
        # 환경 초기화 및 초기 상태 관측
        state = self.env.reset()

        while not done:

            # 학습 가시화
            #self.env.render()

            # 행동 샘플링
            action = self.get_action(tf.convert_to_tensor([state], dtype=tf.float32))
            # 행동 범위 클리핑
            action = np.clip(action, -self.action_bound, self.action_bound)
            # 다음 상태, 보상 관측
            next_state, reward, done, _ = self.env.step(action)
            # shape 변환
            state = np.reshape(state, [1, self.state_dim])
            action = np.reshape(action, [1, self.action_dim])
            reward = np.reshape(reward, [1, 1])
            next_state = np.reshape(next_state, [1, self.state_dim])
            done = np.reshape(done, [1, 1])
            # 학습용 보상 계산
            train_reward = (reward + 8) / 8

```

```

# 배치에 저장
batch_state.append(state)
batch_action.append(action)
batch_reward.append(train_reward)
batch_next_state.append(next_state)
batch_done.append(done)

# 배치가 채워질 때까지 학습하지 않고 저장만 계속
if len(batch_state) < self.BATCH_SIZE:
    # 상태 업데이트
    state = next_state[0]
    episode_reward += reward[0]

    time += 1
    continue

# 배치가 채워지면 학습 진행
# 배치에서 데이터 추출
states = self.unpack_batch(batch_state)
actions = self.unpack_batch(batch_action)
train_rewards = self.unpack_batch(batch_reward)
next_states = self.unpack_batch(batch_next_state)
dones = self.unpack_batch(batch_done)

# 배치 비움
batch_state, batch_action, batch_reward, batch_next_state, batch_done = [], [], [], [], []

# 시간차 타깃 계산
next_v_values = self.critic(tf.convert_to_tensor(next_states, dtype=tf.float32))
td_targets = self.td_target(train_rewards, next_v_values.numpy(), dones)

# 크리틱 신경망 업데이트
self.critic_learn(tf.convert_to_tensor(states, dtype=tf.float32),
                  tf.convert_to_tensor(td_targets, dtype=tf.float32))

# 어드밴티지 계산
v_values = self.critic(tf.convert_to_tensor(states, dtype=tf.float32))
next_v_values = self.critic(tf.convert_to_tensor(next_states, dtype=tf.float32))
advantages = train_rewards + self.GAMMA * next_v_values - v_values

```

```

# 액터 신경망 업데이트
self.actor_learn(tf.convert_to_tensor(states, dtype=tf.float32),
                 tf.convert_to_tensor(actions, dtype=tf.float32),
                 tf.convert_to_tensor(advantages, dtype=tf.float32))

# 상태 업데이트
state = next_state[0]
episode_reward += reward[0]
time += 1

# 에피소드마다 결과 출력
print('Episode: ', ep+1, 'Time: ', time, 'Reward: ', episode_reward)

self.save_epi_reward.append(episode_reward)

# 에피소드 10번마다 신경망 파라미터를 파일에 저장
if ep % 10 == 0:
    self.actor.save_weights("./save_weights/pendulum_actor.h5")
    self.critic.save_weights("./save_weights/pendulum_critic.h5")

# 학습이 끝난 후, 누적 보상값 저장
np.savetxt('./save_weights/pendulum_epi_reward.txt', self.save_epi_reward)
print(self.save_epi_reward)

## 에피소드와 누적 보상값을 그려주는 함수
def plot_result(self):
    plt.plot(self.save_epi_reward)
    plt.show()

```

a2c_main.py

```

# A2C main
# coded by St.Watermelon

## 에이전트를 학습하고 결과를 도시하는 파일
# 필요한 패키지 임포트
from a2c_learn import A2Cagent
import gym

```

```

def main():

    max_episode_num = 1000    # 최대 에피소드 설정
    env_name = 'Pendulum-v0'
    env = gym.make(env_name)  # 환경으로 OpenAI Gym의 pendulum-v0 설정
    agent = A2Cagent(env)    # A2C 에이전트 객체

    # 학습 진행
    agent.train(max_episode_num)

    # 학습 결과 도시
    agent.plot_result()

if __name__=="__main__":
    main()

```

a2c_load_play.py

```

# A2C load and play (tf2 version)
# coded by St.Watermelon

## 학습된 신경망 파라미터를 가져와서 에이전트를 실행시키는 파일
# 필요한 패키지 임포트
import gym
import tensorflow as tf
from a2c_learn import A2Cagent

def main():

    env_name = 'Pendulum-v0'
    env = gym.make(env_name)

    agent = A2Cagent(env)

    agent.load_weights('./save_weights/')  # 신경망 파라미터를 가져옴

    time = 0
    state = env.reset() # 환경을 초기화하고 초기 상태 관측

    while True:
        env.render()

```

```

action = agent.actor(tf.convert_to_tensor([state], dtype=tf.float32))[0][0] # 행동 계산
state, reward, done, _ = env.step(action) # 환경으로 부터 다음 상태, 보상 받음
time += 1

print('Time: ', time, 'Reward: ', reward)

if done:
    break

env.close()

if __name__=="__main__":
    main()

```

2. 참조 A3C를 활용하여

- 동기적 방법과 비동기적 방법을 비교분석하세요. (성능비교 테이블과 x축은 에피소드 y축이 보상인 학습곡선을 그려서 비교분석하세요.).
- 기존 그래디언트 병렬화 방법을 데이터 병렬화 방법으로 변경하고 비교분석하세요. (성능비교 테이블과 x축은 에피소드 y축이 보상인 학습곡선을 그려서 비교분석하세요.).
- 변경된 코드를 모두 제출하세요.

참조 코드개요:

그래디언트 병렬화 방식의 A3C 코드는 액터-크리틱 신경망을 구현하고 학습시키기 위한 a3c_learn.py, 이를 실행시키기 위한 a3c_main.py, 그리고 학습을 마친 신경망 파라미터를 읽어와 에이전트를 구동하기 위한 a3c_load_play.py로 구성됨

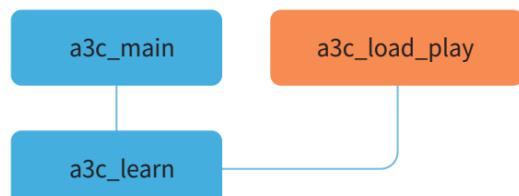


그림 5.11 A3C 코드 구조

a3c_learn.py

```
# A3C learn (tf2 subclassing API version: Gradient)
# coded by St.Watermelon

# 필요한 패키지 임포트
import gym
import tensorflow as tf

from tensorflow.keras.models import Model
from tensorflow.keras.layers import Dense, Lambda
from tensorflow.keras.optimizers import Adam

import numpy as np
import matplotlib.pyplot as plt

import threading
import multiprocessing

## 액터 신경망
class Actor(Model):

    def __init__(self, action_dim, action_bound):
        super(Actor, self).__init__()
        self.action_bound = action_bound

        self.h1 = Dense(64, activation='relu')
        self.h2 = Dense(32, activation='relu')
        self.h3 = Dense(16, activation='relu')
        self.mu = Dense(action_dim, activation='tanh')
        self.std = Dense(action_dim, activation='softplus')

    def call(self, state):
        x = self.h1(state)
        x = self.h2(x)
        x = self.h3(x)
        mu = self.mu(x)
        std = self.std(x)
```

```

# 평균값을 [-action_bound, action_bound] 범위로 조정
mu = Lambda(lambda x: x*self.action_bound)(mu)

return [mu, std]

## 크리틱 신경망
class Critic(Model):

def __init__(self):
super(Critic, self).__init__()

self.h1 = Dense(64, activation='relu')
self.h2 = Dense(32, activation='relu')
self.h3 = Dense(16, activation='relu')

self.v = Dense(1, activation='linear')

def call(self, state):
x = self.h1(state)
x = self.h2(x)
x = self.h3(x)
v = self.v(x)
return v

# 모든 워커에서 공통으로 사용할 글로벌 변수 설정
global_episode_count = 0
global_step = 0
global_episode_reward = [] # save the results

## a3c 에이전트 클래스
class A3Cagent(object):

def __init__(self, env_name):

# 학습할 환경 설정
self.env_name = env_name
self.WORKERS_NUM = multiprocessing.cpu_count() # 워커의 개수
env = gym.make(env_name)
# 상태변수 차원
self.state_dim = env.observation_space.shape[0]

# 행동 차원
self.action_dim = env.action_space.shape[0]

```

```

# 행동의 최대 크기
self.action_bound = env.action_space.high[0]
# 글로벌 액터 및 크리틱 신경망 생성
self.global_actor = Actor(self.action_dim, self.action_bound)
self.global_critic = Critic()

self.global_actor.build(input_shape=(None, self.state_dim))
self.global_critic.build(input_shape=(None, self.state_dim))

self.global_actor.summary()
self.global_critic.summary()

## 신경망 파라미터 로드
def load_weights(self, path):
    self.global_actor.load_weights(path + 'pendulum_actor.h5')
    self.global_critic.load_weights(path + 'pendulum_critic.h5')

## 학습
def train(self, max_episode_num):
    workers = []
    # 워커 스레드를 생성하고 리스트에 추가
    for i in range(self.WORKERS_NUM):
        worker_name = 'worker%i' % i
        workers.append(A3Cworker(worker_name, self.env_name, self.global_actor,
                               self.global_critic, max_episode_num))

    # 리스트를 순회하면서 각 워커 스레드를 시작하고 다시 리스트를 순회하면서 조인함
    for worker in workers:
        worker.start()

    for worker in workers:
        worker.join()

    # 학습이 끝난 후, 글로벌 누적 보상값 저장
    np.savetxt('./save_weights/pendulum_epi_reward.txt', global_episode_reward)
    print(global_episode_reward)

```

```

## 에피소드와 글로벌 누적 보상값을 그려주는 함수
def plot_result(self):
    plt.plot(global_episode_reward)
    plt.show()

## a3c 워커 클래스
class A3Cworker(threading.Thread):

    def __init__(self, worker_name, env_name, global_actor, global_critic, max_episode_num):
        threading.Thread.__init__(self)

        # 하이퍼파라미터
        self.GAMMA = 0.95
        self.ACTOR_LEARNING_RATE = 0.0001
        self.CRITIC_LEARNING_RATE = 0.001
        self.t_MAX = 4 # n-스텝 시간차

        self.max_episode_num = max_episode_num

        # 워커의 환경 생성
        self.env = gym.make(env_name)
        self.worker_name = worker_name

        # 글로벌 신경망 공유
        self.global_actor = global_actor
        self.global_critic = global_critic

        # 상태변수 차원
        self.state_dim = self.env.observation_space.shape[0]
        # 행동 차원
        self.action_dim = self.env.action_space.shape[0]
        # 행동의 최대 크기
        self.action_bound = self.env.action_space.high[0]
        # 표준편차의 최솟값과 최댓값 설정
        self.std_bound = [1e-2, 1.0]

        # 워커 액터 및 크리틱 신경망 생성
        self.worker_actor = Actor(self.action_dim, self.action_bound)
        self.worker_critic = Critic()
        self.worker_actor.build(input_shape=(None, self.state_dim))
        self.worker_critic.build(input_shape=(None, self.state_dim))

```

```

# 옵티마이저
self.actor_opt = Adam(self.ACTOR_LEARNING_RATE)
self.critic_opt = Adam(self.CRITIC_LEARNING_RATE)

# 글로벌 신경망의 파라미터를 워커 신경망으로 복사
self.worker_actor.set_weights(self.global_actor.get_weights())
self.worker_critic.set_weights(self.global_critic.get_weights())

## 로그-정책 확률밀도함수 계산
def log_pdf(self, mu, std, action):
    std = tf.clip_by_value(std, self.std_bound[0], self.std_bound[1])
    var = std ** 2
    log_policy_pdf = -0.5 * (action - mu) ** 2 / var - 0.5 * tf.math.log(var * 2 * np.pi)
    return tf.reduce_sum(log_policy_pdf, 1, keepdims=True)

## 액터 신경망에서 행동을 샘플링
def get_action(self, state):
    mu_a, std_a = self.worker_actor(state)
    mu_a = mu_a.numpy()[0]
    std_a = std_a.numpy()[0]
    std_a = np.clip(std_a, self.std_bound[0], self.std_bound[1])
    action = np.random.normal(mu_a, std_a, size=self.action_dim)
    return action

## 액터 신경망 학습
def actor_learn(self, states, actions, advantages):

    with tf.GradientTape() as tape:
        # 정책 확률밀도함수

        mu_a, std_a = self.worker_actor(states, training=True)
        log_policy_pdf = self.log_pdf(mu_a, std_a, actions)

        # 워커의 손실함수 계산
        loss_policy = log_policy_pdf * advantages
        loss = tf.reduce_sum(-loss_policy)

        # 워커의 그래디언트 계산
        grads = tape.gradient(loss, self.worker_actor.trainable_variables)
        # 그래디언트 클리핑
        grads, _ = tf.clip_by_global_norm(grads, 20)

```

```

# 워커의 그래디언트를 이용해 글로벌 신경망 업데이트
self.actor_opt.apply_gradients(zip(grads, self.global_actor.trainable_variables))

## 크리틱 신경망 학습
def critic_learn(self, states, n_step_td_targets):
    with tf.GradientTape() as tape:
        # 워커의 손실함수 계산
        td_hat = self.worker_critic(states, training=True)
        loss = tf.reduce_mean(tf.square(n_step_td_targets-td_hat))
    # 워커의 그래디언트 계산
    grads = tape.gradient(loss, self.worker_critic.trainable_variables)
    # 그래디언트 클리핑
    grads, _ = tf.clip_by_global_norm(grads, 20)
    # 워커의 그래디언트를 이용해 글로벌 신경망 업데이트
    self.critic_opt.apply_gradients(zip(grads, self.global_critic.trainable_variables))

## n-스텝 시간차 타깃 계산
def n_step_td_target(self, rewards, next_v_value, done):
    y_i = np.zeros(rewards.shape)
    cumulative = 0
    if not done:
        cumulative = next_v_value

    for k in reversed(range(0, len(rewards))):
        cumulative = self.GAMMA * cumulative + rewards[k]
        y_i[k] = cumulative
    return y_i

## 배치에 저장된 데이터 추출
def unpack_batch(self, batch):
    unpack = batch[0]
    for idx in range(len(batch)-1):
        unpack = np.append(unpack, batch[idx+1], axis=0)

    return unpack

## 파이썬에서 스레드를 구동하기 위해서는 함수명을 run으로 해줘야 함. 워커의 학습을 구현
def run(self):
    # 모든 워커에서 공통으로 사용할 글로벌 변수 선언
    global global_episode_count, global_step
    global global_episode_reward

```

```
# 워커 실행 시 프린트
print(self.worker_name, "starts ---")
# 에피소드마다 다음을 반복
while global_episode_count <= int(self.max_episode_num):

    # 배치 초기화
    batch_state, batch_action, batch_reward = [], [], []
    # 에피소드 초기화
    step, episode_reward, done = 0, 0, False
    # 환경 초기화 및 초기 상태 관측
    state = self.env.reset()
    # 에피소드 종료 시까지 다음을 반복
    while not done:

        # 환경 가시화
        #self.env.render()
        # 행동 추출
        action = self.get_action(tf.convert_to_tensor([state], dtype=tf.float32))
        # 행동 범위 클리핑
        action = np.clip(action, -self.action_bound, self.action_bound)
        # 다음 상태, 보상 관측
        next_state, reward, done, _ = self.env.step(action)
        # shape 변환
        state = np.reshape(state, [1, self.state_dim])
        action = np.reshape(action, [1, self.action_dim])
        reward = np.reshape(reward, [1, 1])

        # 학습용 보상 범위 조정
        train_reward = (reward + 8) / 8
        # 배치에 저장
        batch_state.append(state)
        batch_action.append(action)
        batch_reward.append(train_reward)
        # 상태 업데이트
        state = next_state
```

```

episode_reward += reward[0]
step += 1

# 배치가 채워지면 워커 학습 시작
if len(batch_state) == self.t_MAX or done:

    # 배치에서 데이터 추출
    states = self.unpack_batch(batch_state)
    actions = self.unpack_batch(batch_action)
    rewards = self.unpack_batch(batch_reward)
    # 배치 비움
    batch_state, batch_action, batch_reward = [], [], []
    # n-스텝 시간차 타깃과 어드밴티지 계산
    next_state = np.reshape(next_state, [1, self.state_dim])
    next_v_value = self.worker_critic(tf.convert_to_tensor(next_state, dtype=tf.float32))
    n_step_td_targets = self.n_step_td_target(rewards, next_v_value.numpy(), done)
    v_values = self.worker_critic(tf.convert_to_tensor(states, dtype=tf.float32))
    advantages = n_step_td_targets - v_values

    # 글로벌 크리틱 신경망 업데이트
    self.critic_learn(states, n_step_td_targets)
    # 글로벌 액터 신경망 업데이트
    self.actor_learn(states, actions, advantages)
    # 글로벌 신경망 파라미터를 워커 신경망으로 복사
    self.worker_actor.set_weights(self.global_actor.get_weights())
    self.worker_critic.set_weights(self.global_critic.get_weights())
    # 글로벌 스텝 업데이트
    global_step += 1
    # 에피소드가 종료되면,
    if done:
        # 글로벌 에피소드 카운트 업데이트
        global_episode_count += 1
        # 에피소드마다 결과 보상값 출력
        print('Worker name:', self.worker_name, ', Episode: ', global_episode_count,
              ', Step: ', step, ', Reward: ', episode_reward)
        global_episode_reward.append(episode_reward)
        # 에피소드 10번마다 신경망 파라미터를 파일에 저장
        if global_episode_count % 10 == 0:
            self.global_actor.save_weights("./save_weights/pendulum_actor.h5")
            self.global_critic.save_weights("./save_weights/pendulum_critic.h5")

```

a3c_main.py

```

# A3C main
# coded by St.Watermelon
## A3C 에이전트를 학습하고 결과를 도시하는 파일

# 필요한 패키지 임포트
from a3c_learn import A3Cagent

def main():

    max_episode_num = 1000 # 최대 에피소드 설정
    env_name = 'Pendulum-v0' # 환경으로 OpenAI Gym의 pendulum-v0 설정
    agent = A3Cagent(env_name) # A3C 에이전트 객체

    # 학습 진행
    agent.train(max_episode_num)

    # 학습 결과 도시
    agent.plot_result()

if __name__=="__main__":
    main()

```

a3c_load_play.py

```

# A3C load and play (tf2 version)
# coded by St.Watermelon
## 학습된 신경망 파라미터를 가져와서 에이전트를 실행시키는 파일

# 필요한 패키지 임포트
import gym
import tensorflow as tf
from a3c_learn import A3Cagent

def main():

    env_name = 'Pendulum-v0'
    env = gym.make(env_name)

    agent = A3Cagent(env_name) # A3C 에이전트 객체
    # 글로벌 신경망 파라미터 가져옴
    agent.load_weights('./save_weights/')

```

```
time = 0
state = env.reset() # 환경을 초기화하고, 초기 상태 관측

while True:
    env.render()
    # 행동 계산
    action = agent.global_actor(tf.convert_to_tensor([state], dtype=tf.float32))[0][0]
    # 환경으로부터 다음 상태, 보상 받음

    state, reward, done, _ = env.step(action)
    time += 1

    print('Time: ', time, 'Reward: ', reward)

    if done:
        break

env.close()

if __name__=="__main__":
    main()
```