강화 학습 Term Project

학과	학번	성명
로봇학부	2014741002	이종수

목차

 # CartPole Environment Setting

 # 1) Friction

: friction은 추가하지 않았음 -> REINFORCE와 A2C에서 Epsilon Greedy Policy를 사용하지 않으므로, 모험을 하지 않는다. 따라서 문제 #3 에서 목표 위치 1m에 다다르도록 하기 위해서는 최종적으로 학습에 필요한 Episode가 늘어날지라도 여러 State를 경험할 필요가 있으므로 이를 위해 마찰을 없애 큰 동작을 수행하게 하여 많은 State를 경험하도록 한다.

(cartpole environment.py)

```
# Without Viscous Friction
xacc = temp - self.polemass_length * thetaacc * costheta /
self.total_mass

# With Viscous Friction
xacc2 = temp - self.polemass_length * thetaacc * costheta - 5 * x_dot /
self.total_mass

# 마찰을 사용하지 않을 것이므로, xacc 를 사용한다 !!
if self.kinematics_integrator == "euler":
    x = x + self.tau * x_dot                    # 위치 = 위치 + 속도적분
    x_dot = x_dot + self.tau * xacc             # 속도 = 속도+가속도적분
    theta = theta + self.tau * theta_dot        # 각도 = 각도 + 각속도적분
    theta_dot = theta_dot + self.tau * thetaacc  # 각속도 = 각속도 +
각가속도적분
else:  # semi-implicit euler
    x_dot = x_dot + self.tau * xacc
    x = x + self.tau * x_dot
    theta_dot = theta_dot + self.tau * thetaacc
    theta = theta + self.tau * theta_dot
```

 # 2) Target Score : 400 ~ 600

 # 3) Maximum Episode : DQN : 400 , REINFORCE, A2C : 1000

 # 4) Target Position

: **문제 #1, #2**에서는 목표 위치 없음 - cartpole environment.py에서 done flag에 위치에 대한 별다른 코드를 추가 하지 않았다.

```
# Simulation 종료 Signal
done = bool(
    x < -self.x_threshold
    or x > self.x_threshold
    or theta < -self.theta_threshold_radians
    or theta > self.theta_threshold_radians
)

if not done:
    reward = 1

elif self.steps_beyond_done is None:
    # Pole just fell!
```

```
    self.steps_beyond_done = 0
    reward = 1

else:
    if self.steps_beyond_done == 0:
        logger.warn(
            "You are calling 'step()' even though this "
            "environment has already returned done = True. You "
            "should always call 'reset()' once you receive 'done = "
            "True' -- any further steps are undefined behavior."
        )
    self.steps_beyond_done += 1
    reward = 0.0
```

: **문제 #3** 에서는 **목표 위치 1m**

Done flag에 1m에 다다르면 reward를 더 주거나, 거리가 크게 벌어지면 reward를 줄이는 등의 코드를 추가하여 agent가 1m 목표에 최대한 오래 머무를 수 있도록 Setting 함. 또한 현재 위치를 받아오기 위하여, info dictionary에 위치 값 x를 전달받아 매번 출력하도록 수정함

```
# 수정
destination = 1 # [m]
if not done:

    # 목표 위치와 떨어진 거리만큼 가중치를 두어 Reward 주기
    reward = 1 + ( 1 -  abs(destination - x) ) * 1.5

    if abs(destination - x) > 0.2: # 거리가 0.2 이상이면 절반만 주자.
        reward = reward * 0.5

    elif abs(destination - x) <= 0.2: # 거리가 0.2 이하면 그대로 다 주자
        reward = reward

    elif abs(destination - x) == 0: # 위치가 1m 라면 reward 증폭
        reward = reward * 2

elif self.steps_beyond_done is None:
    # Pole just fell!
    self.steps_beyond_done = 0
    reward = 1.0 * ( 1 -  abs(destination - x) ) * 1.5
else:
    if self.steps_beyond_done == 0:
        logger.warn(
            "You are calling 'step()' even though this "
            "environment has already returned done = True. You "
            "should always call 'reset()' once you receive 'done = "
            "True' -- any further steps are undefined behavior."
        )
    self.steps_beyond_done += 1
    reward = 0.0

# 수정 --- 거리 값 X 를 출력
return np.array(self.state, dtype=np.float32), reward, done, {'X' : x}
```

 # 문제 1) REINFORCE

REINFORCE: Monte-Carlo Policy-Gradient Control (episodic) for π_*

Input: a differentiable policy parameterization $\pi(a|s,\boldsymbol{\theta})$
Algorithm parameter: step size $\alpha > 0$
Initialize policy parameter $\boldsymbol{\theta} \in \mathbb{R}^{d'}$ (e.g., to $\mathbf{0}$)

Loop forever (for each episode):
  Generate an episode $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$, following $\pi(\cdot|\cdot,\boldsymbol{\theta})$
  Loop for each step of the episode $t = 0, 1, \ldots, T-1$:
    $G \leftarrow \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k$ (G_t)
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha \gamma^t G \nabla \ln \pi(A_t|S_t,\boldsymbol{\theta})$

#REINFORCE의 수행 Flow

1. **현재 State에 대해서 Agent가 Action을 선택하고 수행한다.**
2. **Action을 수행한 다음의 State와 Reward를 관찰한다.**
3. **1~2 과정을 Episode가 끝날 때 까지 반복한다. (Monte Carlo Method..)**
4. **Episode가 끝나면 학습을 진행한다.**

#학습 Flow

1. **Episode가 끝날 때까지의 받은 보상들에 대해 Return을 계산한다.**
2. **목적함수의 미분에 Q함수 대신 Return(G)을 넣어 Gradient를 계산한다.**
3. **계산된 Gradient를 Cross Entropy Loss x G 함수에 적용하여 Policy를 업데이트한다.**

 # Code 분석

먼저 Policy를 학습시키는 REINFORCE class를 정의한다.
해당 class는 2개의 hidden layer와 1개의 output layer로 구성되어 있고, 마지막 layer의 node 수는 행동할 수 있는 가지 수 (Left or Right)인 2이다. 해당 신경망에는 입력으로 State를 넣어주고, 출력으로 Action을 받게 된다. 따라서 이것은 Policy를 구하는 과정이며 결과적으로 행동에 대한 확률을 얻을 것이기 때문에 Activation function으로 Softmax(확률의 총합 = 1)를 사용한다.

```
class REINFORCE(tf.keras.Model):
    def __init__(self, action_size):
        super(REINFORCE, self).__init__()
        self.fc1 = Dense(24, activation = 'relu')
        self.fc2 = Dense(24, activation = 'relu')
        self.fc_out = Dense(action_size, activation = 'softmax') # 출력층의
Activation function : Softmax
```

```
    def call(self, x): # x : state vector
        x = self.fc1(x)
        x = self.fc2(x)

        policy = self.fc_out(x) # 출력으로는 해당 State 에서의 각 행동에 대한
확률을 return !!!
        return policy
```

$$\pi_\theta(a|s) = \frac{e^{\phi(s,a)^\top \theta}}{\sum_b e^{\phi(s,b)^\top \theta}}$$

Softmax 함수

agent가 환경을 훈련을 진행하며, Object function의 Gradient를 계산한 후 업데이트하여 해당 class를 수행한다.

다음으로는 환경에 대해 훈련을 진행할 Agent를 정의하는 부분이다.

```
class REINFORCEAgent:
    def __init__(self, state_size, action_size): # state 크기와 action
크기를 입력으로 받는 Agent Class

        self.render = True  # Drawing

        # 상태와 행동의 크기 정의
        self.state_size = state_size
        self.action_size = action_size

        # REINFORCE Hyperparameter 정의
        self.discount_factor = 0.99
        self.learning_rate = 0.01

        self.model = REINFORCE(self.action_size)
        self.optimizer = Adam(lr = self.learning_rate)

        # 값을 다루기 위해 빈 공간 할당
        self.states, self.actions, self.rewards = [], [], []
```

먼저 Agent는 환경에 대한 정보를 받아야한다. State는 위치, 속도, 각도, 각속도 총 4가지를 가지고 있으며, 행동은 왼쪽, 오른쪽으로 총 2가지를 가지고 있다. 이를 init 함수에서 가장 먼저 정의해준다. 또한 REINFORCE는 목적함수를 업데이트하는 과정에서 Q 함수 대신 Monte Carlo의 Return을 사용하기 때문에 이후 이를 위해 사용할 discount factor 즉, gamma 값을 정의해준다. 덧붙여 모델 학습에 필요한 학습률을 정의하고, REINFORCE 신경망과 역전파를 수행할 Optimizer를 정의한다.

계속해서 Agent 내부에 있는 함수들이다.

```
# 인공신경망으로 학습한 Policy를 따라 행동을 결정
def get_action(self, state):
    policy = self.model(state)[0] # ----------- [[위치, 속도, 각도, 각속도]]
이렇게 받을 거니까 [0] 해주기 --- 왜? NN에 넣어서 학습시키려면 차원 맞춰야지
    policy = np.array(policy) # 반환된 각 행동에 대한 확률들 numpy type으로


    # Policy 분포를 따라서 1개를 뽑자 (Optimal Policy가 아니니 [0.2 0.6] 이렇게
나올 것)
    return np.random.choice(self.action_size, 1, p = policy)[0]

# 에피소드가 끝나면 아래 함수로 Return 값을 계산하고, 이를 이용해서 업데이트를 진행
def discount_rewards(self, rewards): # Monte - Carlo method ...
    discounted_rewards = np.zeros_like(rewards) # rewards와 같은 크기의
영행렬
    running_add = 0

    for t in reversed(range(0, len(rewards))): # 뒤에서 부터 차례대로 ...
        running_add = running_add * self.discount_factor + rewards[t]
        discounted_rewards[t] = running_add

    return discounted_rewards
```

get_action 함수는 Policy에 따라 Action을 선택하는 부분이다.

np.random.choice(self.action_size, 1, p = policy)[0] 은 policy의 확률 분포를 따르며 action size 즉, 2개 중에서 1개를 뽑는 것을 수행한다.

다음에 등장하는 discount_rewards는 Monte – Carlo method에서 사용한 Total Reward를 계산하는 부분이며, 뒤에서부터 하나씩 discount factor를 곱하여 각 time step에 해당하는 return을 구하고, 이를 빈 행렬에 차곡차곡 쌓아 반환해준다. 이를 이용하여 목적함수의 Gradient를 구할 수 있다.

```
# 에피소드가 끝날 때 까지 상태, 행동, 보상 저장
# 신경망을 업데이트 할 때 (theta = theta + a * ~) Q 함수대신 Return을 사용할
것이다. 따라서 이 값을 매번 저장해두어야한다.
def append_sample(self, state, action, reward):
    self.states.append(state[0])
    self.rewards.append(reward)

    act = np.zeros(self.action_size)
    act[action] = 1 # Make one hot Vector
    self.actions.append(act)
```

위의 append_sample함수가 Episode를 진행하며 경험하는 모든 상태, 행동, 보상을 저장해두기 때문에 이 값들을 이용해서 discount_rewards 함수가 제 역할을 수행한다.

다음으로 Policy를 학습시키는 훈련 코드이다.

```
def train_model(self):

    model_params = self.model.trainable_variables
    with tf.GradientTape() as tape:
        tape.watch(model_params)

        policies = self.model(np.array(self.states))
        actions = np.array(self.actions) # 크기 action_size, 값 = One hot
vector

        action_prob = tf.reduce_sum(actions*policies, axis = 1) # 행동을 할
확률과, one-hot vector를 곱하면 해당하는 확률 값 얻을 수 있다.
        # CE = -sum(y*log(p)) --- y=1 일 때, CE = -log(p_action)
        cross_entropy = - tf.math.log(action_prob + 1e-5)
        loss = tf.reduce_sum(cross_entropy * discounted_rewards)
        entropy = - policies * tf.math.log(policies)

    grads = tape.gradient(loss, model_params)
    self.optimizer.apply_gradients(zip(grads, model_params))

    # 다시 싹 비우기
    self.states, self.actions, self.rewards = [], [], []

    return np.mean(entropy)
```

시작부에서 trainable_variable함수를 통해 학습을 수행할 Parameter들을 가져온다.
이 함수들에만 gradient를 계산하고 업데이트를 진행할 것이다. 위 코드를 한 줄 씩 살펴보며 코드를 이해해보자.

```
policies = self.model(np.array(self.states))
actions = np.array(self.actions) # 크기 action_size, 값 = One hot vector

action_prob = tf.reduce_sum(actions*policies, axis = 1) # 행동을 할 확률과,
one-hot vector를 곱하면 해당하는 확률 값 얻을 수 있다.
```

가장 먼저 현재 정책에 따른 Policy 결과를 받아온다. 이 정책은 tf.reduce_sum(actions*policies, axis =1) 이라는 함수에 들어가 계산이 수행되는데, 이 코드의 의미는 다음과 같다.
actions라는 변수는 append_sample 함수에서 One-hot 인코딩되어 반환되었다. 따라서 Policies 변수에 각 행동에 대해서 확률이 들어있는데 이와 곱해지므로, 선택된 행동과 그 행동에 대한 확률값이 곱해지게 된다. 즉, [0 1] * [0.2, 0.8] = 0.8 이 된다.

$$-\sum_{c=1}^{C} L_c log(P_c)$$

Cross Entropy ...

```
# CE = -sum(y*log(p)) --- y=1 일 때, CE = -log(p_action)
cross_entropy = - tf.math.log(action_prob + 1e-5)
```

이 값은 Cross Entropy 계산에 그대로 이용되는데, Cross Entropy는 현재 확률분포와 정답 확률분포 간의 차이를 의미한다. 여기서는 해당 행동에 대한 Cross Entropy의 정답 값을 항상 1로 설정해두었다. 따라서 CE = -1*(logP)가 된다. 한 가지 중요한 점은 우리는 목적 함수에 대해 Gradient Ascent 를 수행해야 하는데, 기존 목적 함수에 -를 곱해주면 Convex한 함수로 볼 수 있다. 따라서 이 함수에 Gradient Descent 를 수행하는 것이 동일한 결과를 나타낸다.

추가적으로 볼 것은 log 내부의 값이 0이 되지 않도록 매우 작은 값을 추가한 모습이다.
결국 우리는 아래의 식을 계산하여 목적함수를 업데이트 해야한다.

$$\nabla_\theta J(\theta) = F_{\pi_\theta}[\nabla_\theta \log \pi_\theta(a|s) q^{\pi_\theta}(s,a)].$$

```
loss = tf.reduce_sum(cross_entropy * discounted_rewards)
```

위의 코드를 수행하여 위에서 구해둔 Cross_entropy와 Q함수 대신 Return을 넣어 우리가 원하는 목적함수의 Gradient를 계산할 수 있다.

```
grads = tape.gradient(loss, model_params)
self.optimizer.apply_gradients(zip(grads, model_params))

# 다시 싹 비우기
self.states, self.actions, self.rewards = [], [], []

return np.mean(entropy)
```

이 다음으로는 계산해둔 Gradient를 업데이트할 변수에 적용시키고, 다음 Episode에서 사용하기 위해 기존에 사용한 Array를 초기화 시켜준다. 반환 값으로는 Entropy 값을 주는데, 이 값을 현재 추정하고 있는 분포와 정답 분포가의 차이를 알려준다. 1에 가까울수록 정답에 가깝다.

다음으로는 REINFORCE를 수행시키는 Main 함수이다.
훈련에서 최대로 사용할 Episode는 400으로 설정해두었으며, 학습을 위해 취할 수 있는 Action의 수, 그리고 상태의 수를 가져온다. 이는 Agent 클래스 객체를 선언할 때 매개변수로써 넘겨준다.

```
if __name__ == "__main__":

    env = gym.make('CartPole-v1')
    state_size = env.observation_space.shape[0]
    action_size = env.action_space.n
```

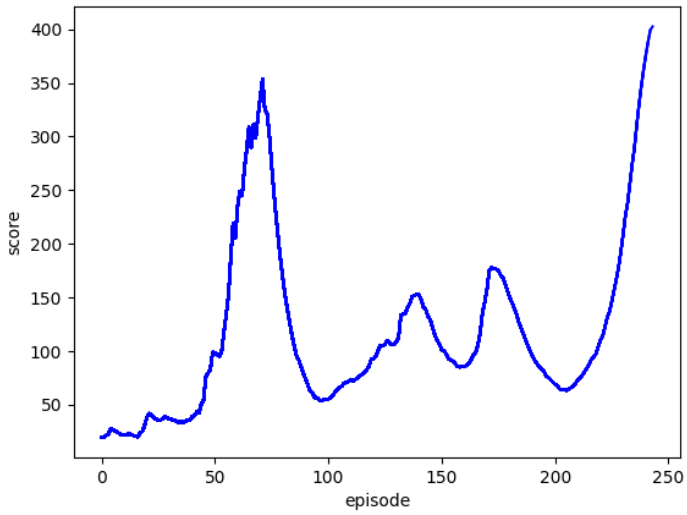
```
    agent = REINFORCEAgent(state_size, action_size)
    scores , episodes = [], []
    score_avg = 0
    num_episode = 1000
    for e in range(num_episode):
        done = False
        score = 0
        state = env.reset()
        state = np.reshape(state, [1, state_size])
        while not done:
            if agent.render:
                env.render()
            # 현재 상태로 행동을 선택
            action = agent.get_action(state)
            # 선택한 행동으로 환경에서 한 타임스텝 진행
            next_state, reward, done, info = env.step(action)
            next_state = np.reshape(next_state, [1, state_size])

            agent.append_sample(state, action, reward)
            score += reward
            reward = 0.1 if not done or score == 500 else -1
            state = next_state
            if done:
                # 정책 신경망 업데이트
                entropy = agent.train_model()
                score_avg = 0.9 * score_avg + 0.1 * score if score_avg != 0
else score
                print("episode: {:3d} | score avg: {:3.2f} | entropy:
{:.3f}".format(
                      e, score_avg, entropy))

                scores.append(score_avg)
                episodes.append(e)

                if score_avg > 400:
                    agent.model.save_weights("./model", save_format="tf")
                    sys.exit()
```

마찬 가지로 main 문 내의 코드이며, 이 코드는 최대 에피소드 1000이 되기 전까지 또는 목표 보상인 400을 달성하기 전까지 반복되는 For 문이다.
Done 변수는 목표 보상이 달성되었거나, 아니면 수직봉이 넘어졌거나 특정 threshold 범위를 넘어갔을 때 True로 바뀌며 Epsiode를 종료시키기 위해 사용된다.

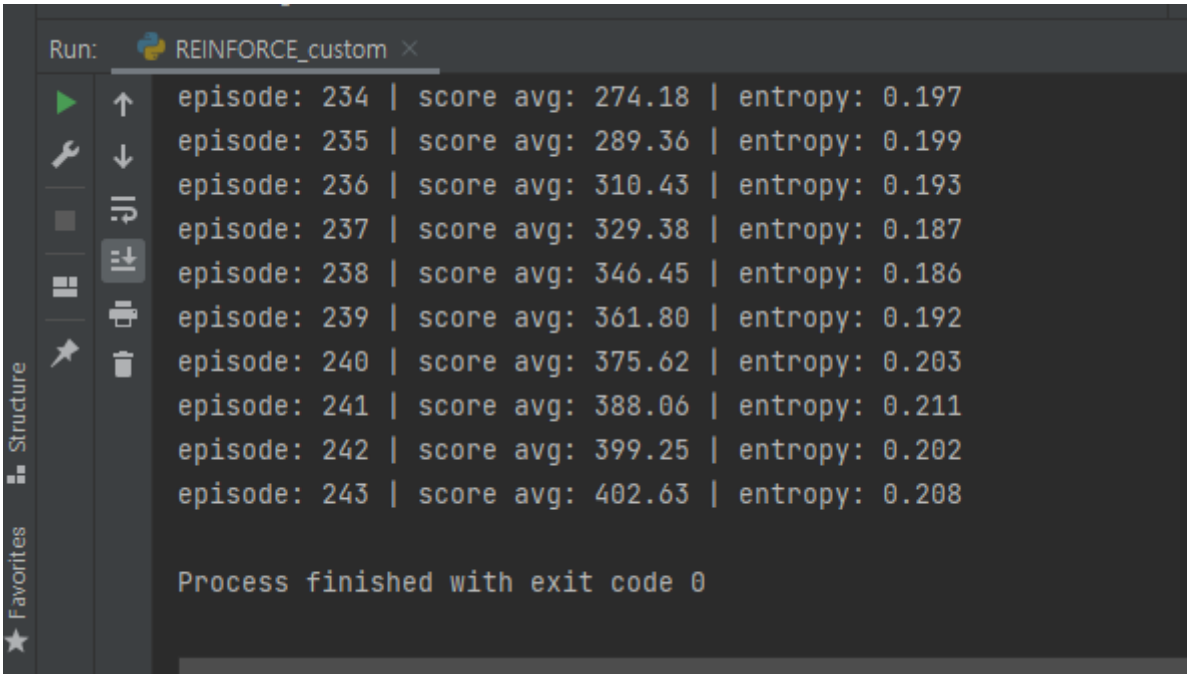
가장 먼저 객체로 선언한 agent에서 현재 state에 기반한 행동을 가져온다. 이 선택된 행동을 기반으로 다음 step을 수행하고 그 때의 상태, 보상, done 변수를 가져온다. Done 변수가 True 가 되었을 때, 즉 Epsiode가 종료되었을 때 저장해 두었던 상태, 행동, 보상들을 기반으로 Policy를 업데이트 시키는 코드가 작성되어 있다.

마지막으로 목표 보상을 달성하였을 때에는 해당 모델을 저장하여 재사용이 가능하게 하였다.

결과 분석



총 수행 Episode = 243 회



REINFORCE 특성상 학습 시 Value function을 근사하여(G) 사용하기 때문에 Variance가 매우 큰 단점이 있다. 자세한 이유는 Monte Carlo Method가 Terminated Episode를 기반으로 수행된다는 것인데, 예를 들어 Episode가 매우 길게 진행된다면 Return 값이 그에 맞게 굉장히 커진다. 또한 상대적으로 일찍 끝나버린 Episode는 Return 값이 매우 작다.

다시 말해 Episode에 따라 Return 값이 들쭉 날쭉하기 때문에 학습의 Variance가 크게 된다. 이러한 결과가 학습 과정에 영향을 미치게 되어, 결과적으로 학습이 점진적으로 매끄럽게 되지 않고 출력 Score가 들쭉 날쭉 한 모습을 확인할 수 있다.

문제 2) A2C

A2C

: REINFORCE의 큰 분산을 보완할 수 있으며, 특징으로는 REINFORCE에서 Q함수를 Return으로 근사하여 사용하였던 것과 달리, Q 함수를 Value function Approximation 을 통해 직접 근사하여 목적함수를 업데이트한다. 더 나아가 A2C에서는 q 함수로 근사한 것에서 unbiased estimated한 함수인 Advantage function을 사용한다.

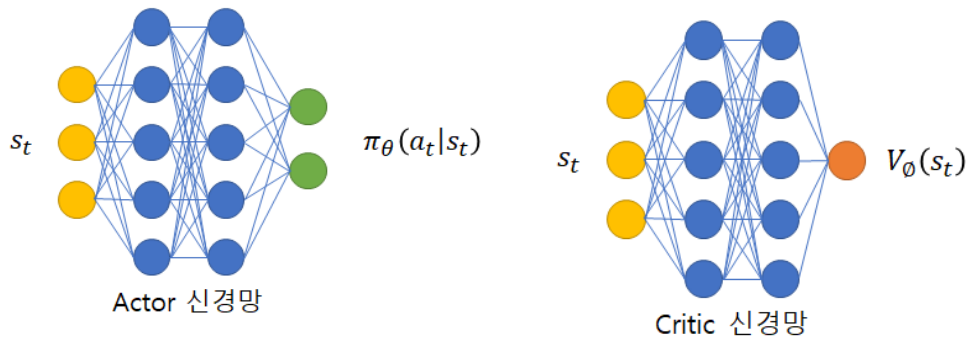
One-step Actor–Critic (episodic), for estimating $\pi_\theta \approx \pi_*$

Input: a differentiable policy parameterization $\pi(a|s,\boldsymbol{\theta})$
Input: a differentiable state-value function parameterization $\hat{v}(s,\mathbf{w})$
Parameters: step sizes $\alpha^{\boldsymbol{\theta}} > 0$, $\alpha^{\mathbf{w}} > 0$
Initialize policy parameter $\boldsymbol{\theta} \in \mathbb{R}^{d'}$ and state-value weights $\mathbf{w} \in \mathbb{R}^{d}$ (e.g., to $\mathbf{0}$)
Loop forever (for each episode):
  Initialize S (first state of episode)
  $I \leftarrow 1$
  Loop while S is not terminal (for each time step):
    $A \sim \pi(\cdot|S,\boldsymbol{\theta})$
    Take action A, observe S', R
    $\delta \leftarrow R + \gamma \hat{v}(S',\mathbf{w}) - \hat{v}(S,\mathbf{w})$ (if S' is terminal, then $\hat{v}(S',\mathbf{w}) \doteq 0$)
    $\mathbf{w} \leftarrow \mathbf{w} + \alpha^{\mathbf{w}} \delta \nabla \hat{v}(S,\mathbf{w})$
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha^{\boldsymbol{\theta}} I \delta \nabla \ln \pi(A|S,\boldsymbol{\theta})$
    $I \leftarrow \gamma I$
    $S \leftarrow S'$

A2C의 수행 및 학습 Flow

:A2C는 두 개의 신경망을 동시에 업데이트 시킨다. 1. Value function 2. Policy

1. 학습한 Policy에 따라 현재 상태에서 행동을 선택하고 수행
2. 다음 상태와 보상을 관찰
3. S, A, R, S' Sample을 통해 TD Error를 구하고 Advantage function을 업데이트
4. TD 에러를 통해 Value function을 업데이트
5. Advantage function과 CrossEntropy를 통해 Policy 업데이트



두 신경망 모두 상태를 입력으로 받지만, 하나는 정책 또다른 하나는 Value 를 반환한다.

 # Code 분석

먼저 A2C의 신경망을 정의하는 Class이다. 해당 Class에는 두 개의 신경망이 구현되어있으며, 각각 Policy를 근사하는 Actor 신경망, Value function을 근사하는 Critic 신경망이 있다.

```
class A2C(tf.keras.Model):
    def __init__(self, action_size):
        super(A2C,self).__init__()

        self.actor_fc = Dense(24, activation = 'tanh')
        # 출력이 행동이기 때문에 Softmax activation 함수 사용
        self.actor_out = Dense(action_size, activation='softmax')# 가치
신경망 정의 -- relu 보다 tanh 가 더 성능이 좋았다.

        self.critic_fc1 = Dense(24, activation='tanh')
        self.critic_fc2 = Dense(24, activation='tanh')

        self.critic_out = Dense(1)
```

강의 서적의 Reference Code에서는 기존의 relu가 아닌 tanh activation function을 단지 성능의 문제로 선택하였다고 한다. 먼저 Keras module의 최상위 class를 가져온 후 내부에서 두 개의 신경망을 구현한다. Actor는 간단히 1개의 24개 node를 가진 hidden layer로 구성되있고, Critic은 2개의 Hidden layer로 구성되어있다. 각 Network의 출력은 Policy, value이므로 activation function 은 Softmax와 Linear로 각각 정의되있다.

```
def call(self, x):

    # 정책 신경망을 통해 Policy Improvement 수행
    actor_x = self.actor_fc(x)
    policy = self.actor_out(actor_x) # 출력 : Policy

    # 가치 신경망을 통해 Policy Evaluation 을 수행 --- Delta function Update!
    critic_x = self.critic_fc1(x)
    critic_x = self.critic_fc2(critic_x)

    value = self.critic_out(critic_x) # 출력 : 가치 함수


    return policy, value
```

다음으로 위 신경망 내부에서 입력으로 들어온 상태 값이 네트워크를 흘러가며 계산이 수행되도록 구현된 코드이다. A2C 객체를 수행하면 자동적으로 Call 함수가 수행된다.
Init 함수에서 구현한 것과 같이 각각 1개의 layer, 2개의 layer로 구성되어 있고, Return 값으로 policy와 value를 돌려준다.

```
class A2CAgent:
    def __init__(self, action_size):
        self.render = True

        self.action_size = action_size
        self.discount_factor = 0.99
        self.learning_rate = 0.001

        # 정책신경망 가치신경망 생성
        self.model = A2C(self.action_size)

        # 최적화 알고리즘 설정, 미분값이 너무 커지는 현상을 막기 위해 clipnorm 설정
        # Clipnorm :
        self.optimizer = Adam(lr=self.learning_rate, clipnorm=5.0)
```

다음으로 Agent를 정의하는 부분이다. 사실 REINFORCE 알고리즘과 A2C이 크게 다르지 않아 REINFORCE에서 설명한 내용이 대부분 동일하다. 따라서 해당 알고리즘이 가지는 특징이 두드러지게 나타나는 부분에 대해서만 설명을 붙이도록 한다.

```
# 정책신경망의 출력을 받아 확률적으로 행동을 선택
def get_action(self, state):

    policy, _ = self.model(state) # 정책 신경망의 출력값
    policy = np.array(policy[0])
    return np.random.choice(self.action_size, 1, p=policy)[0] # Policy 에
따라 Action 을 선택
```

해당 함수는 역시 특정 상태에서 다음 Action을 가져오는 함수이며, 코드에서 볼 수 있듯, 객체로 생성한 model에 상태 변수를 입력 인자로 넣어 2개의 반환값을 받는다. 해당 함수에서는 policy만을 필요로 하므로, 두 번째 반환값인 value 값은 사용하지 않는다.

다음으로 실제 모델을 훈련시키는 함수이다. 역시 가장 먼저 업데이트할 파라미터들의 목록을 가져온다.

```
# 각 타임스텝마다 정책신경망과 가치신경망을 업데이트
def train_model(self, state, action, reward, next_state, done):

    model_params = self.model.trainable_variables # Update 할 파라미터들
가져오기
```

본격적으로 tensorflow에서 제공하는 Gradient Descent 방법에 따라 코드를 수행한다.

```
with tf.GradientTape() as tape:
    policy, value = self.model(state)  # 정책 신경망과 가치 신경망의 출력 V(s)
가져오기
    _, next_value = self.model(next_state) # 모델에 다음 Step 의 상태를 넣어
V(s+1) 가져오기
```

```
    # Advantage function 계산 delta = R + gamma*V(S_t+1) - V(S_t)
    target = reward + (1 - done) * self.discount_factor * next_value[0]
```

위의 코드에서 발견할 수 있는 것은 target이라고 되어있는 변수인데, 바로 다음을 의미한다.

$$\delta_{\mathbf{v}} = r + \gamma v_{\mathbf{v}}(s') - v_{\mathbf{v}}(s)$$

Advantage function / TD Error

$$r + \gamma v_{\mathbf{v}}(s')$$

TD Target

즉, target 이라는 변수는 위의 TD Target을 의미하며, 조금 아래 코드에서 나머지 value function을 빼주는 과정으로 Advantage function을 완성한다. 물론 TD Target에서 의미하는 Value function은 다음 Step에 해당하는 다음 state의 value이다.

```
# 정책 신경망 오류 함수 구하기
one_hot_action = tf.one_hot([action], self.action_size)
action_prob = tf.reduce_sum(one_hot_action * policy, axis=1) # 해당하는
행동의 확률 Get
cross_entropy = - tf.math.log(action_prob + 1e-5)
```

위의 코드는 REINFORCE와 완벽히 동일하므로 간단히 생략하도록 한다.
바로 아래 코드가 매우 중요하다.

```
# Advantage function 계산 delta = R + gamma*V(S_t+1) - V(S_t)
advantage = tf.stop_gradient(target - value[0]) # 정책신경망의 오류함수를
구하는 과정에서 가치 함수를 업데이트 하지 않기 위함!
actor_loss = tf.reduce_mean(cross_entropy * advantage)
```

Advantage function의 모양이 보인다. 위에서 구해두었던 target 변수에 현재 state에 대한 value를 빼는 모습이 보이며, tf.stop_gradient를 사용한 이유는 이 과정이 Policy에 대한 신경망을 업데이트 하는 부분인데 이 때 가치 함수를 업데이트 하지 않게 하기 위함이다.
Actor loss는 기존 목적함수를 구하는 과정과 동일하지만, REINFORCE와 다르게 G가 아니라 advantage 함수가 곱해져있음을 확인할 수 있다.

```
# 가치 신경망 오류 함수 구하기
critic_loss = 0.5 * tf.square(tf.stop_gradient(target) - value[0])
critic_loss = tf.reduce_mean(critic_loss)
```

다음으로 동일하게 Value function을 근사하는 신경망도 업데이트 시켜줘야한다.
Value function approximation에서 사용한 방법 그대로 MSE Loss를 사용하여 그 오차를 줄이는 방향으로 Gradient Descent 방법을 사용하여 업데이트 해준다.

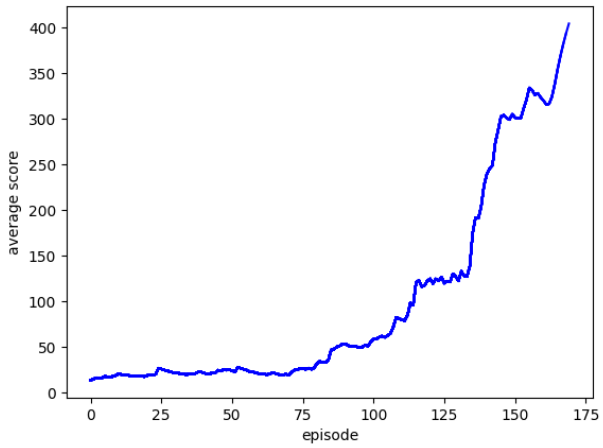
```
# 하나의 오류 함수로 만들기
loss = 0.2 * actor_loss + critic_loss
```

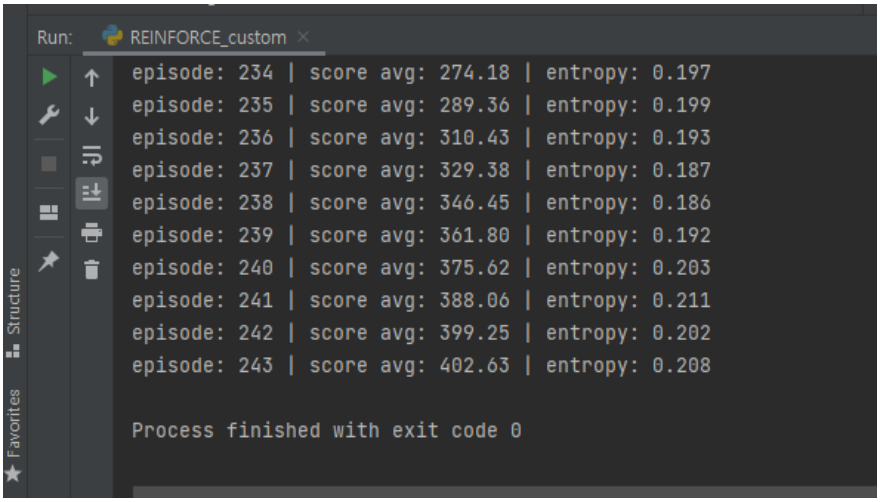
끝으로 전체 loss에 대해 모든 파라미터를 업데이트 해야하기에 하나의 loss 함수로 합쳐주는 과정을 거치면 된다.

```
# 오류함수를 줄이는 방향으로 모델 업데이트
grads = tape.gradient(loss, model_params)
self.optimizer.apply_gradients(zip(grads, model_params))
return np.array(loss)
```

이제 이전 REINFORCE와 동일하게 loss에 대해서 gradient를 계산하고, 기존 파라미터에 해당 Gradient 계산 결과를 적용시켜 업데이트 해준다. 해당 부분 아래의 main 문은 REINFORCE와 완벽히 일치하므로 생략하도록 한다.

 # 결과 분석





총 수행 Episode = 243회

REINFORCE와 동일한 조건에서 수행한 A2C의 결과 그래프이다.
A2C를 여러 번 수행해보았으며, 가장 깔끔하게 나온 그래프를 결과그래프로 사용하였다.
깔끔하지 못한 그래프들도 대체적으로 REINFORCE에 비해 매우 매끄럽게 나왔다. 학습이 수행되는 과정이 Score 그래프에 그대로 나타났지만, REINFORCE와 A2C 모두 Epsilon Greedy 방법을 사용하지 않아 탐험을 하지 않음에도 불구하고 REINFORCE는 기존의 정책을 벗어나 완전히 다른 행동을 할 때가 많았다. 이는 분명 Return의 Variance가 매우 큼직하게 차이가 났기에 그에 따른 부작용이라고 생각된다. 따라서 Variance를 줄인 A2C같은 경우, 한 번 제대로 학습 방향을 잡게 되면 이를 놓치지 않고 꾸준히 성장시키는 것을 확인할 수 있다.

 # 문제 3) DQN / REINFORCE / A2C를 사용하여 1m 위치에 Cart 위치시키기

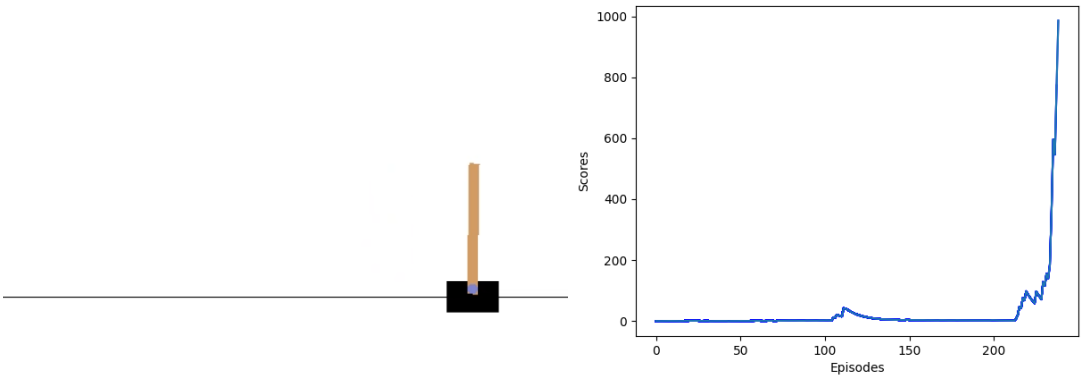
 # Environment Setting

: Gym module 내부의 cartpole.py 코드에서 done이라는 flag 내부에 어떤 상황에서 reward를 줄 지 결정할 수 있는데, 기존에는 Pole이 쓰러지지 않는다면 reward를 1씩 주는 형태로 구성되어있다. 이 부분은 다음과 같이 변경하여 코드를 수행해본다. (DQN / REINFORCE / A2C의 코드는 그대로 사용한다.)

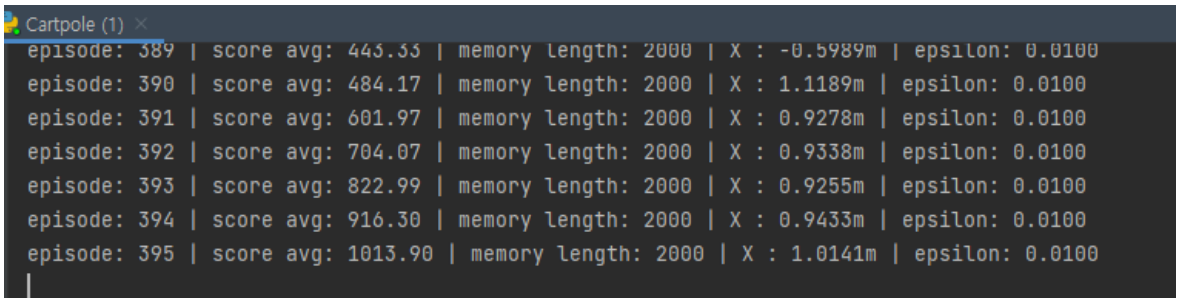
**** 해당 코드에 대한 설명은 목차 #1 환경 세팅에서 설명해두었으므로 생략한다.**

```
# 수정
destination = 1 # [m]
if not done:

    # 목표 위치와 떨어진 거리만큼 가중치를 두어 Reward 주기
    reward = 1 + ( 1 -  abs(destination - x) ) * 1.5

    if abs(destination - x) > 0.2: # 거리가 0.2 이상이면 절반만 주자.
        reward = reward * 0.5

    elif abs(destination - x) <= 0.2: # 거리가 0.2 이하면 그대로 다 주자
        reward = reward

    elif abs(destination - x) == 0: # 위치가 1m 라면 reward 증폭
        reward = reward * 2

elif self.steps_beyond_done is None:
    # Pole just fell!
    self.steps_beyond_done = 0
    reward = 1.0 * ( 1 -  abs(destination - x) ) * 1.5
else:
    if self.steps_beyond_done == 0:
        logger.warn(
            "You are calling 'step()' even though this "
            "environment has already returned done = True. You "
            "should always call 'reset()' once you receive 'done = "
            "True' -- any further steps are undefined behavior."
        )
    self.steps_beyond_done += 1
    reward = 0.0

# 수정 --- 거리 값 X 를 출력
return np.array(self.state, dtype=np.float32), reward, done, {'X' : x}
```

DQN 실행 결과



1m 거리에서 정지한 Cart 와 Score Graph

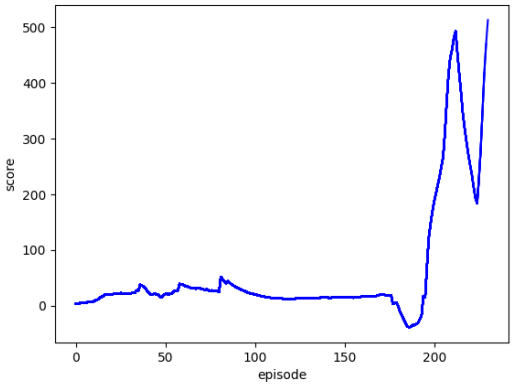


위의 그래프는 250회 정도까지 수행하였을 때 저장한 그래프이다.
추가로 100회정도 더 수행하였을 경우, 아래 X 변수에서 볼 수 있 듯, 1m에 근사하게 pole이 정지하여 유지하는 것을 확인할 수 있다. DQN은 Variance 관점에서 A2C와 비슷하다. Replay memory을 통해 Batch 학습을 수행하기 때문에 Variance가 적다. 즉, batch 샘플들이 서로 상관관계가 없기 때문에, 정규화와 같은 효과를 내고, 결과적으로 bias와 variance를 줄일 수 있다. Score Graph또한 A2C와 비슷한 것을 확인할 수 있다.

또한 DQN이 REINFORCE와 A2C에 비해서는 250 ~ 300회의 Episode안에서 대부분 학습이 안정적으로 마무리 되는 모습을 볼 수 있다. 이 부분에 대해서는 분명 Epsilon Greedy의 영향이 클 것이라고 예상된다. REINFOECE와 A2C와는 다르게 이따금씩 정책과 다른 행동을 통해 다양한 State를 경험하도록 한다. 따라서 우리가 1m라는 목표 위치에 다다르기 위해서는 1m 근방으로 이동해야하는데, 다른 알고리즘에 비해 DQN의 탐험성이 이 목표지점 근방을 탐색하는데 매우 도움을 준다고 생각된다.

이 때문에 이후 다른 알고리즘에서 마찰을 줄이는 방식으로 동작을 크게 하여 1m 근방을 탐색할 수 있도록 하였다.

REINFORCE 실행 결과



약 1m 거리에서 정지한 Cart 와 Score Graph

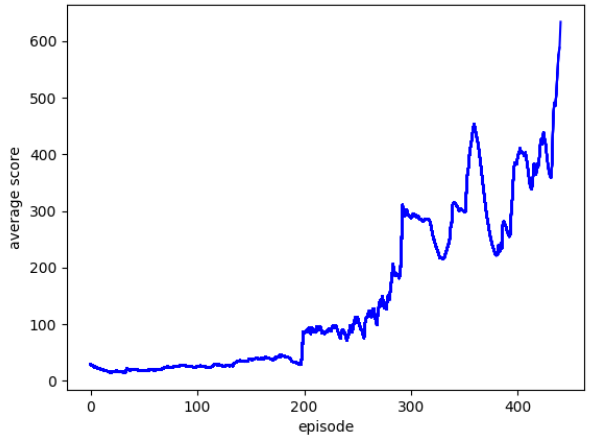
```
episode: 215 | score avg: 383.34 | X : 2.0516m | | entropy: 0.210
episode: 216 | score avg: 346.22 | X : -0.1265m | | entropy: 0.194
episode: 217 | score avg: 319.37 | X : 2.4213m | | entropy: 0.202
episode: 218 | score avg: 295.52 | X : 2.4337m | | entropy: 0.192
episode: 219 | score avg: 273.27 | X : 2.4189m | | entropy: 0.186
episode: 220 | score avg: 254.08 | X : 2.3420m | | entropy: 0.209
episode: 221 | score avg: 236.68 | X : 2.0539m | | entropy: 0.204
episode: 222 | score avg: 214.04 | X : -0.0478m | | entropy: 0.193
episode: 223 | score avg: 193.85 | X : -0.1028m | | entropy: 0.203
episode: 224 | score avg: 183.60 | X : 1.7429m | | entropy: 0.228
episode: 225 | score avg: 228.52 | X : 1.1862m | | entropy: 0.227
episode: 226 | score avg: 281.46 | X : 1.4366m | | entropy: 0.231
episode: 227 | score avg: 348.91 | X : 1.2968m | | entropy: 0.238
episode: 228 | score avg: 417.97 | X : 1.0293m | | entropy: 0.237
```

REINFORCE는 평균적으로 400 ~ 600 회 Episode를 수행하면서 학습을 완료했다.
드물게는 위와 같이 200회 정도의 Episode에서도 학습을 완료하는 모습을 보여준다.

이 알고리즘의 특징을 바로 여기서 확인할 수 있다. 즉, 학습이 완료되기 위해 필요한 에피소드의 양이 매우 편차가 크다는 점이다. 이에 대한 이유는 위에서 설명한 바와 같이 Monte Carlo Method를 사용함에 따라 Terminated Episode들을 사용하는데, 만약 해당 에피소드 들의 길이가 들쭉날쭉하면 사용되는 Return G의 값이 크게 크게 변하게 된다.

따라서 이 값에 대한 영향으로 정책의 방향이 크게 틀어질 수 있기 때문에 일정한 에피소드를 유지하는 경우에는 위 경우와 같이 200 회 정도의 에피소드로도 학습을 완료할 수 있지만, 중간에 적은 횟수로 짧거나, 완전히 다른 방향의 에피소드를 수행하게 되면 정책의 학습이 원점으로 돌아가는 경우도 발생한다. 따라서 REINFORCE는 매우 편차가 큰 학습 알고리즘이므로, 환경을 완벽히 설계하거나 아니면 많은 시행착오를 통해 최적의 결과를 얻는 등으로 사용해야 할 것이다.

A2C 실행 결과



총 440 회의 Episode를 학습

```
episode: 432 | score avg: 383.56 | X : 1.8665m
episode: 433 | score avg: 460.36 | X : 1.2322m
episode: 434 | score avg: 491.07 | X : 1.4463m
episode: 435 | score avg: 485.48 | X : 2.4076m
episode: 436 | score avg: 506.89 | X : 1.9199m
episode: 437 | score avg: 543.18 | X : 1.0228m
episode: 438 | score avg: 575.15 | X : 1.2847m
episode: 439 | score avg: 588.19 | X : 0.9952m
episode: 440 | score avg: 633.88 | X : 1.3037m

Process finished with exit code 0
```

A2C는 REINFORCE의 경우와 달리 Value와 Policy를 같이 학습해야 하는 알고리즘이다.
이를 이유로 DQN, REINFORCE와 비교하여 상대적으로 더 많은 Episode를 필요로 한다.
(평균 500 ~ 700회 정도) 처음에는 다른 알고리즘 들에 비해 상대적으로 학습이 매우 불안하고, 값이 튀는 모습을 보여주지만, 어느정도 안정한 궤도에 올라서게 되면 REINFORCE보다 더 일관적인 학습을 보여준다. 이는 REINFORCE에 비해 Variance가 적어 학습한 것을 토대로 조금씩 더 나아가며, 다른 이상한 행동을 하도록 큰 변화가 일어나지 않기 때문이다. 즉, 방향성이 계속 유지 되는 것을 확인할 수 있다.

하지만 이러한 특성 때문에, 해당 문제의 목적인 1m 위치에 정지시키는 것에 문제가 있다.
즉, 1m라는 지점 주변을 탐색하여 해당 state를 경험해보아야 하는데, Epsilon greedy 정책을 사용하지도 않고, 또 현재 학습과정에서 크게 차이나는 행동도 하지 않기 때문에 목표 지점 근방으로 Cart를 옮기기 쉽지 않았다. 이를 이유로 Friction을 제거하고, 목표 지점 이외에 부분에서는 reward를 현저히 낮게 주어 조금이라도 행동을 크게 해서 목표 지점 근방에 이르면 reward를 크게 받을 수 있게 만들어 주었다.
한 번 해당 state들을 경험한 이후로는 꽤 준수하게 학습을 진행하는 모습을 볼 수 있다.

 # 고찰

문제 #1, #2
Cartpole을 단순히 수행하는 것은 DQN, REINFORCE, A2C 모두 좋은 성능을 보인다.
하지만 안정한 궤도에 올라선 뒤 그것을 유지하는 것은 A2C 알고리즘이 가장 좋은 성능을 보였으며, 그 다음으로 DQN이다. REINFORCE는 조금이라도 이상한 에피소드를 수행하는 순간 Return 값이 크게 달라져 안정성이 무너진다.

문제 #3
Cartpole을 1m 목표 지점으로 옮기는 것은 매우 다른 문제다. 이를 수행하는 것은 DQN 알고리즘에는 큰 문제가 없었지만 REINFORCE와 A2C 알고리즘은 Epsilon Greedy Policy 방법을 사용하지 않으므로 해당 지점 근처를 탐색하게 하기 위한 방법이 추가로 필요하다.
따라서 목표 지점을 탐색하는 측면에서 DQN이 가장 사용한 에피소드가 작았다.
하지만 한 번 해당 지점을 탐색한 이후로 A2C , REINFORCE 모두 괜찮은 성능을 보였다.
동일하게 REINFORCE는 한 번의 Outliar로도 안정성이 무너진다. A2C는 뚝심있게 본인이 학습한 것을 토대로 점진적으로 안정성을 높여나간다.

이번 프로젝트를 통해서 각 알고리즘이 어떤 특성을 가지고 있는지 확인할 수 있었고,
Dynamics를 다르게 꾸며 이후 시뮬레이션이 아닌, 실제 프로젝트에 사용해볼 수 있을 것이라 기대된다.