

인공지능프로그래밍

Lab8

Deep Q Learning

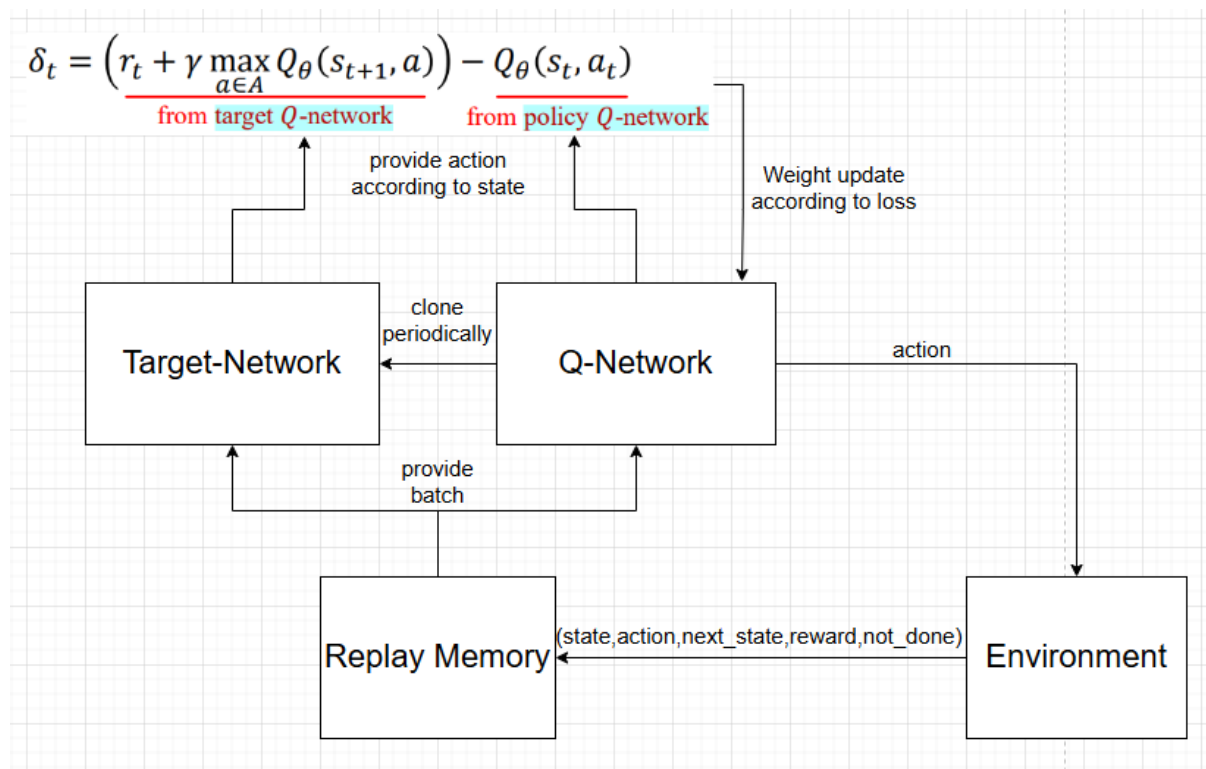
2024/12/01

2019202050 이강현

Lab Objective

Deep Q-learning Network을 직접 구현해보며 Q-learning에 어떻게 function approximation을 적용하는지 이해한다. 그 과정에서 Replay Memory를 활용하는 방법과 코드상에서 어떻게 state와 action을 처리하는지 이해한다. Cart pole과 luna lander environment에서 Q-learning이 적용되는 방식을 이해한다. 추가적으로 agent의 모습을 시각화하여 학습된 모습을 확인한다.

Program Flow



위는 전체적인 DQN의 흐름을 나타낸 것이다. 최상단의 수식을 통해 target Q와 policy Q의 차이를 구하고 이를 통해 네트워크를 학습시키는 과정이다. Q-Network와 Target-Network는 동일한 네트워크의 객체이고 Q-Network만을 학습시킨 후 주기적으로 가중치를 Target-Network에 복사하는 방식이다. 네트워크 내부의 층 구성은 아래 코드에서 자세히 설명한다. Replay Memory에는 Q-Network에서 나온 action을 통해 environment에서 얻은 정보가 experience로 저장되며 이후 batch의 형태로 네트워크의 입력으로 사용하게 된다.

Deep Q-learning Network(DQN)

$$\delta_t = (r_t + \gamma \max_{\hat{a}_{t+1}} Q_T(s_{t+1}, \hat{a}_{t+1}; \theta_T)) - Q_P(s_t, a_t; \theta_P)$$

위는 DQN에서 사용되는 수식이다. 수식은 Tabular Q-learning에서 사용된 수식과 동일하다. 한 가지 달라지는 점은 table을 사용하는 것과 함수를 사용하는 것의 차이이다. DQN은 table을 사용할 때 공간을 많이 사용한다는 단점을 보완하고자 function approximation을 적용하여 이를 해결하고자 하였다. 하지만 시간이 중요한 강화학습의 특성상 학습과정에서 함수의 가중치가 시간에 독립적이지 못하는 점이 발생하여 이를 replay memory를 이용하여 (state,action) pair를 모으고 랜덤하게 뽑아내 함수를 학습시키는 방식을 택했다. 결과적으로 연속된 sample들의 상관관계를 없앨 수 있다. 또한 함수를 학습시키는 과정에서 하나의 네트워크로 학습할 때 발생할 수 있는 발산 문제를 두 개의 네트워크로 나눔으로서 해결한다. 이때 각각의 네트워크에서 얻어낸 값을 통해 위 수식을 계산하여 함수를 최적화한다.

Result

아래는 사용할 라이브러리를 설치하는 부분이다. colab에서 사용중인지를 체크하고 분기하여 swig, gymnasium, tqdm을 설치한다.

```
▶ #import library
# Check if this code runs in Colab
RunningInCOLAB = 'google.colab' in str(get_ipython()) # If running in colab

if RunningInCOLAB:
    !pip install swig # A library that allows code written in C to run in Python
    !pip install gymnasium # for reinforcement learning
    !pip install gymnasium[box2d] # for box environment
    from tqdm.notebook import tqdm # to check progress
else: # if not running in colab
    from tqdm import tqdm # to check progress
```

```

Collecting swig
  Downloading swig-4.3.0-py2.py3-none-manylinux_2_5_x86_64.manylinux1_x86_64.whl.metadata (3.5 kB)
Downloading swig-4.3.0-py2.py3-none-manylinux_2_5_x86_64.manylinux1_x86_64.whl (1.9 MB)
1.9/1.9 MB 32.1 MB/s eta 0:00:00
Installing collected packages: swig
Successfully installed swig-4.3.0
Collecting gymnasium
  Downloading gymnasium-1.0.0-py3-none-any.whl.metadata (9.5 kB)
Requirement already satisfied: numpy>=1.21.0 in /usr/local/lib/python3.10/dist-packages (from gymnasium) (1.26.4)
Requirement already satisfied: cloudpickle>=1.2.0 in /usr/local/lib/python3.10/dist-packages (from gymnasium) (3.1.0)
Requirement already satisfied: typing-extensions>=4.3.0 in /usr/local/lib/python3.10/dist-packages (from gymnasium) (4.12.2)
Collecting farama-notifications>=0.0.1 (from gymnasium)
  Downloading Farama_Notifications-0.0.4-py3-none-any.whl.metadata (558 bytes)
Downloading gymnasium-1.0.0-py3-none-any.whl (958 kB)
958.1/958.1 kB 12.9 MB/s eta 0:00:00
Downloading Farama_Notifications-0.0.4-py3-none-any.whl (2.5 kB)
Installing collected packages: farama-notifications, gymnasium
Successfully installed farama-notifications-0.0.4 gymnasium-1.0.0
Requirement already satisfied: gymnasium[box2d] in /usr/local/lib/python3.10/dist-packages (1.0.0)
Requirement already satisfied: numpy>=1.21.0 in /usr/local/lib/python3.10/dist-packages (from gymnasium[box2d]) (1.26.4)
Requirement already satisfied: cloudpickle>=1.2.0 in /usr/local/lib/python3.10/dist-packages (from gymnasium[box2d]) (3.1.0)
Requirement already satisfied: typing-extensions>=4.3.0 in /usr/local/lib/python3.10/dist-packages (from gymnasium[box2d]) (4.12.2)
Requirement already satisfied: farama-notifications>=0.0.1 in /usr/local/lib/python3.10/dist-packages (from gymnasium[box2d]) (0.0.4)
Collecting box2d-py==2.3.5 (from gymnasium[box2d])
  Downloading box2d-py-2.3.5.tar.gz (374 kB)
374.4/374.4 kB 11.3 MB/s eta 0:00:00
Preparing metadata (setup.py) ... done
Requirement already satisfied: pygame>=2.1.3 in /usr/local/lib/python3.10/dist-packages (from gymnasium[box2d]) (2.6.1)
Requirement already satisfied: swig==4.* in /usr/local/lib/python3.10/dist-packages (from gymnasium[box2d]) (4.3.0)
Building wheels for collected packages: box2d-py
  Building wheel for box2d-py (setup.py) ... done
  Created wheel for box2d-py: filename=box2d-py-2.3.5-cp310-cp310-linux_x86_64.whl size=2376422 sha256=17c1cc08494d80a37048845674289e715ada5aae928e5ed1fe195d9c118a894c
  Stored in directory: /root/.cache/pip/wheels/db/8f/6a/eaadf056fba10a98d986f6dce954e6201ba3126926fc5ad9e
Successfully built box2d-py
Installing collected packages: box2d-py
Successfully installed box2d-py-2.3.5

```

위와 같이 존재하지 않는 라이브러리들이 잘 설치되는 것을 확인한다.

```

▶ #Import various libraries
import os # To use functions supported by the operating system
os.environ["KERAS_BACKEND"] = "tensorflow" # for tensorflow environment

import numpy as np # For useful array uses
import tensorflow as tf # Deep learning library
import keras # Deep learning library with tensorflow
import matplotlib.pyplot as plt #for visualizing

import gymnasium as gym #for reinforcement learning
from gymnasium import wrappers #for recoding video

from collections import deque # to use queue algorithm
import random # for randomizing

```

위는 설치한 라이브러리와 과제에 필요한 다양한 라이브러리들을 import하는 부분이다.

```

▶ #Setting GPU
physical_devices = tf.config.list_physical_devices('GPU')# available GPU
print(physical_devices) # print
try:
    tf.config.experimental.set_memory_growth(physical_devices[0], True) #to setting GPU environment
except:
    print('GPU is not detected.')# cannot detect GPU

↵ [PhysicalDevice(name='/physical_device:GPU:0', device_type='GPU')]

```

```

[4] gym.__version__# gymnasium library version

↵ '1.0.0'

```

위는 현재 연결된 GPU가 사용가능한지 체크하고 탐지되지 않을 시 오류를 출력한다. Gpu가 연결되어 있고 강화학습 라이브러리의 버전을 출력하는 것을 확인한다.

```

[5] # select environment
    # Discrete Action Space:    0 for Cartpole, 1 for LunarLander

    SELECT_ENV = 1

```

위는 학습에 사용할 environment를 결정하는 부분이다. 0이면 cartpole, 1이면 lunarlander라는 환경을 사용한다.

```

▶ #select environment
if SELECT_ENV == 0:
    env_name, res_prefix = 'CartPole-v1', 'cart' #set relevant environment and saved name
    max_episodes, max_ep_steps, goal_score = 400, 500, 450 # training parameter
    b_size, h_size = 128, 1000 # set batch_size and replay memory_size
    network_type, state_width, state_height, state_depth = 'dense', 0, 0, 0 # these values are not use in code
    kwargs = {'render_mode':'rgb_array'} # rendering is rgb
elif SELECT_ENV == 1:
    env_name, res_prefix = 'LunarLander-v3', 'lunD' #set relevant environment and saved name
    max_episodes, max_ep_steps, goal_score = 400, 1000, 200 # training parameter
    b_size, h_size = 128, 10000 # set batch_size and replay memory_size
    network_type, state_width, state_height, state_depth = 'dense', 0, 0, 0 # these values are not use in code
    kwargs = {'continuous':False, 'render_mode':'rgb_array'} # rendering is rgb
else: assert False, 'environment selection error' #can't find environment

def create_env():
    env = gym.make(env_name, **kwargs) # create environment
    return env

```

위는 각각의 environment에 따라 붙여질 이름과 추후 저장될 영상의 prefix, batch_size, replay memory의 size등 전체적인 DQN의 설정값들을 선언하는 부분이다. Create_env는 gymnasium 라이브러리를 이용하여 environment를 make한다.

```

▶ #define environment reset and one step
def env_reset(env):
    observation = env.reset()
    state = observation[0] if type(observation)==tuple else observation
    return state

def env_step(env, action):
    observation = env.step(action) #when one step progressed, observation have three value, state,reward,done
    state = observation[0]
    reward = observation[1]
    done = observation[2] or observation[3] if len(observation)>4 else observation[2]
    return state, reward, done

```

위는 environment를 reset하여 초기 state를 가져오는 env_reset과 action이 주어졌을때 state,reward,done을 반환하는 env_step을 정의하는 부분이다.

```

▶ # environment reset and one step
env = create_env()
state = env_reset(env)
state, reward, done = env_step(env, env.action_space.sample())# use sample

```

위는 정의한 함수들을 실행하는 모습이다. 이때는 sample을 사용한다.

```

▶ #set action_space and observation_space
action_shape = env.action_space.shape #action_space shape
action_space_type = type(env.action_space)

if action_space_type==gym.spaces.discrete.Discrete: #we use discrete action
    actn_space = 'DISCRETE'
    action_shape = (1,)
    action_dims = 1
    action_range = env.action_space.n
    num_actions = action_range # number of actions is action range for DISCRETE actions
    action_batch_shape = (None, action_range) # for replaymemory
elif action_space_type==gym.spaces.box.Box: #don't use this space in code
    actn_space = 'CONTINUOUS'
    action_dims = action_shape[0]
    actn_uppr_bound = env.action_space.high[0]
    actn_lowr_bound = env.action_space.low[0]
    action_range = (actn_uppr_bound - actn_lowr_bound) # x0.5 for tanh output
    action_batch_shape = tuple([None]+[x for x in action_shape]) # for replaymemory
    num_actions = action_dims # number of actions is action dimension for CONTINUOUS actions
else: assert False, 'other action space type are not supported'

observation_space_type = type(env.observation_space)
observation_shape = env.observation_space.shape #observation_space shape

if observation_space_type==gym.spaces.discrete.Discrete: #don't use this space in code
    observation_shape = (1,)
    num_states = env.observation_space.n
elif observation_space_type==gym.spaces.box.Box: #Cart Pole and Lunar Lander have this observation
    num_states = observation_shape[0]
else: print('observation space type error')

state_shape = observation_shape
state_batch_shape = tuple([None]+[x for x in observation_shape]) # for replaymemory

value_shape = (1,)
num_values = 1

```

위는 action_space와 observation_space가 달라질 때 해당값들을 저장하는 부분이다. Action은 discrete와 continuous가 있는데 이번 과제에서 사용할 environment 들은 모두 discrete에 속한다. Observation은 discrete와 box형태가 존재하는데 이번 과제에서 두 environment는 box형태를 사용한다. 위 코드를 통해 action, observation의 각각 shape,dimension 등의 값들을 얻어낼 수 있다. 또한 추후 replay memory의 사용을 위한 batch의 shape을 얻는 것도 확인한다.

아래와 같이 위에서 얻은 값들을 모두 출력하여 현재 environment에서 얻을 수 있는 정보를 한눈에 확인할 수 있다.

```
[10] #check action,observation,state,value
print('Action space ', action_space_type)
print('Action shape ', action_shape)
print('Action dimensions ', action_dims)
print('Action range ', action_range)
if action_space_type==gym.spaces.box.Box:
    print('Max Value of Action ', actn_uppr_bound)
    print('Min Value of Action ', actn_lowr_bound)
else: pass
print('Action batch shape ', action_batch_shape)

print('Observation space ', observation_space_type)
print('Observation shape ', observation_shape)
print('Size of State Space ', num_states)
print('State shape ', state_shape)
print('State batch shape ', state_batch_shape)

print('Vallue shape ', value_shape)
print('Value dimensions ', num_values)
```

아래는 cartpole의 정보들이 담긴 출력이다. 핵심적인 정보들을 살펴보면 Action shape은 1이고 좌로 움직이거나 우로 움직이는 action이므로 range는 2임을 확인한다. Batch size는 batch 크기에 따라 추후 정해지니 none,2로 되어있는 것을 확인한다. Observation shape은 state,reward,terminate,truncate로 4를 나타내고 box형태이다. size of state는 물체의 현재 위치, 속도, 기울어진 각, 각속도 총 4가지의 정보를 담기 때문에 4인 것을 확인한다.

```

Action space <class 'gymnasium.spaces.discrete.Discrete'>
Action shape (1,)
Action dimensions 1
Action range 2
Action batch shape (None, 2)
Observation space <class 'gymnasium.spaces.box.Box'>
Observation shape (4,)
Size of State Space 4
State shape (4,)
State batch shape (None, 4)
Vallue shape (1,)
Value dimensions 1

```

아래는 lunar lander라는 environment에 대한 정보이다.

핵심적인 부분만 살펴보면 action_range는 가만히 있기, 위로 추진, 좌로 추진, 우로 추진으로 4이다. Size of state는 현재의 위치를 나타내는 x,y, x와 y방향 각각의 속도, 현재 비행체의 각과 각속도, 착지를 위한 왼쪽 다리, 오른쪽 다리 각각의 지면에 닿음 유무로 총 8개의 정보를 담기에 8이다.

```

⇄ Action space <class 'gymnasium.spaces.discrete.Discrete'>
Action shape (1,)
Action dimensions 1
Action range 4
Action batch shape (None, 4)
Observation space <class 'gymnasium.spaces.box.Box'>
Observation shape (8,)
Size of State Space 8
State shape (8,)
State batch shape (None, 8)
Vallue shape (1,)
Value dimensions 1

```

```

▶ #define DQNet function
def DQNet(hiddens, act_fn, out_fn, init_fn): # hiddens = (layer1 units, layer2 units)
    inputs = keras.Input(shape=state_shape) # input layer

    ### START CODE HERE ###

    l1 = tf.keras.layers.Dense(units=hiddens[0], activation=act_fn, kernel_initializer=init_fn)(inputs) # first fully connected layer
    # (units=, activation=, kernel_initializer=)
    l2 = tf.keras.layers.Dense(units=hiddens[1], activation=act_fn, kernel_initializer=init_fn)(l1) # second fully connected layer

    outputs = tf.keras.layers.Dense(units=num_actions, activation=out_fn, kernel_initializer=init_fn)(l2) # output (third) layer

    ### END CODE HERE ###

    model = keras.Model(inputs=inputs, outputs=outputs, name='q_net')
    return model

def build_DQNet():
    model = DQNet(hiddens=(32,32), act_fn='relu', out_fn='linear', init_fn='he_uniform')
    return model

```

위는 사용할 네트워크이다.

네트워크는 4개의 층을 사용하는데 먼저 입력층을 통해 shape을 조정하고 2개의 fully-connected layer를 지난다. 이때 fully-connected layer는 32개의 unit을 가진다.

이후 마지막 출력층으로 agent가 취할 수 있는 action을 내놓게 된다. 입력층을 제외한 모든 층은 relu함수를 사용하며 초기에는 He 초기화를 진행한다.

아래는 네트워크의 구성을 확인하기 위한 코드이다.

```
#test model and check inside
test_model = build_DQNet()
test_model.summary()
del test_model
```

Model: "q_net"

Layer (type)	Output Shape	Param #
input_layer (InputLayer)	(None, 4)	0
dense (Dense)	(None, 32)	160
dense_1 (Dense)	(None, 32)	1,056
dense_2 (Dense)	(None, 2)	66

Total params: 1,282 (5.01 KB)
Trainable params: 1,282 (5.01 KB)
Non-trainable params: 0 (0.00 B)

위는 state가 4인 cartpole에 대한 모델로 입력이 batch로 들어와 (none,4)라고 생성됨을 확인한다. 이후 dense층을 통과한 후 batch 형태(none,2)로 나가는 것을 확인한다. 이때 2는 action_range에 따른 q값을 저장하기 위한 크기이다.

Model: "q_net"

Layer (type)	Output Shape	Param #
input_layer (InputLayer)	(None, 8)	0
dense (Dense)	(None, 32)	288
dense_1 (Dense)	(None, 32)	1,056
dense_2 (Dense)	(None, 4)	132

Total params: 1,476 (5.77 KB)
Trainable params: 1,476 (5.77 KB)
Non-trainable params: 0 (0.00 B)

위는 state가 8인 lunarlander에 대한 모델로 입력이 batch로 들어와 (none,8)라고

생성됨을 확인한다. 이후 dense층을 통과한 후 batch 형태(none,4)로 나가는 것을 확인한다. 이때 4는 action_range에 따른 q값을 저장하기 위한 크기이다.

결론적으로 네트워크는 현재 state를 확인하고 추천하는 action을 q값으로 표시하는 역할을 함을 알 수 있다.

```
#agent(Q network, target network)
class Agent_Net:
    def __init__(self):
        self.policy_q = build_DQNNet()           # build policy network
        self.target_q = build_DQNNet()           # build target network
        self.target_update()                     # copy weights from policy to target

    def policy(self, state, epsilon, exploring):   # e-greedy policy if exploring==True
        state_input = tf.convert_to_tensor(state[None,...], dtype=tf.float32) # make the state network-ready

        ### START CODE HERE ###

        if exploring:                            # if e-greedy policy
            if tf.random.uniform(()) > epsilon:   # exploit if random>epsilon
                action_q = self.policy_q(state_input) # get actions for input state
                action = tf.math.argmax(action_q[0]) # find an action for maximum q value
            else:
                action = tf.random.uniform((), minval=0, maxval=action_range, dtype=tf.int64) # random action
        else:
            action_q = self.policy_q(state_input) # else greedy policy (exploitation)
            action = tf.math.argmax(action_q[0]) # get actions for input state
            # find an action for maximum q value

        ### END CODE HERE ###

        return action.numpy()

    def target_update(self):
        self.target_q.set_weights(self.policy_q.get_weights()) # copy weights from policy network to target network
        return
```

Agent_Net은 초기에 두 네트워크의 인스턴스를 생성하고 가중치를 복사한다. 이후 state, epsilon, exploring이 입력되었을 때 exploration과 exploitation을 epsilon, exploring 인자를 통해 결정하고 네트워크로부터 반환받은 q값을 통해 action을 결정하는 함수이다. 추후 함수를 최적화할 때 사용할 target network로의 가중치 복사 함수 또한 존재하는 것을 볼 수 있다.

```

# Check whether the implemented code works well
def test_policy(exploring):

    epsilon = 0.1
    state_input = tf.random.uniform((1,3)) # 1x3 state input
    action_range = 7

    def policy_q(x): # pretending policy_q network
        x = tf.random.uniform((tf.shape(x)[0],action_range))
        return x

    ### START CODE HERE ###

    if exploring:
        if tf.random.uniform(()) > epsilon:
            action_q = policy_q(state_input)
            action = tf.math.argmax(action_q[0])
        else:
            action = tf.random.uniform((),minval=0,maxval=action_range,dtype=tf.int64)
    else:
        action_q = policy_q(state_input)
        action = tf.math.argmax(action_q[0])

    ### END CODE HERE ###

    return action

tf.random.set_seed(2) # Use consistent random values
for _ in range(10): print(test_policy(True).numpy(), ' ', end='')
for _ in range(10): print(test_policy(False).numpy(), ' ', end='')

```

5 0 6 5 1 3 1 4 5 0 3 1 6 0 1 2 0 3 6 4

위는 Agent_Net의 테스트 버전이다. 임의로 인자들과 네트워크를 구성하고 q값을 통해 action을 선택한 것이 올바르게 작동하는 것을 확인하기 위해 존재한다. 아래와 같이 결과가 나왔고 이는 제시된 값과 일치하여 올바르게 동작함을 확인했다.

```

▶ #define replay memory
class ReplayMemory:
    def __init__(self, memory_size):
        self.experiences = deque(maxlen=memory_size) # allocate replay memory
        self.num_episodes = 0 # set number of episode to zero

    def put_experience(self, experience): # put an experience into replay memory
        state, action, next_state, reward, not_done = experience
        self.experiences.append((state, action, next_state, reward, not_done)) #bundle and save
        return

    def get_batch(self, num_samples): # get a batch of randomly sampled experiences
        state_batch, next_state_batch, action_batch, reward_batch, not_done_batch = [], [], [], [], []

        sample_batch = random.sample(self.experiences, num_samples) # shuffle

        for sample in sample_batch: # Take each item out of the sample and save it to the list
            state, action, next_state, reward, not_done = sample
            state_batch.append(state)
            action_batch.append(action)
            next_state_batch.append(next_state)
            reward_batch.append(reward)
            not_done_batch.append(not_done)

        batch = (tf.convert_to_tensor(state_batch, dtype=tf.float32),
                 tf.convert_to_tensor(action_batch, dtype=tf.int32),
                 tf.convert_to_tensor(next_state_batch, dtype=tf.float32),
                 tf.convert_to_tensor(reward_batch, dtype=tf.float32),
                 tf.convert_to_tensor(not_done_batch, dtype=tf.float32))
        return batch #return tensor type

```

위는 replay memory에 대한 코드이다. Replay memory는 deque 라이브러리를 통해 구현되며 이는 replay memory의 크기에 제한을 두어 랜덤하게 값을 추출하기는 하나 비교적 최근 데이터를 사용하여 함수를 최적화하는데 목적을 두기 위함이다. Put_experience는 state,action,next_state,reward,not_done으로 이루어진 experience를 queue에 넣는 함수이다. Get_batch는 queue에서 random으로 batch 크기만큼의 experience를 뽑아내고 이를 각각 나누어 저장하는 것을 확인한다. 이후 나누어진 정보들을 tensor형태로 변환하고 다시 하나의 batch로 묶어 반환한다.

```

[16] #initialize replay memory
def init_memory(mem, env, agent, num_samples):
    state = env.reset(env) # initialize environment
    for _ in range(num_samples):

        ### START CODE HERE ###

        action = agent.policy(state, epsilon=1.0, exploring=True) # get an action with the policy
        next_state, reward, done = env.step(env, action) # observe the environment reaction
        experience = (state, action, next_state, reward, not done) # pack observations into experience tuple, done is converted to not done
        mem.put_experience(experience) # put the experience to replay memory
        state = env.reset(env) if done else next_state # set the next state (reset env if done)

        ### END CODE HERE ###

    return

```

위는 memory를 초기화한다. 초기 state를 env_reset으로 얻고 agent를 통해 얻은 action으로 experience를 형성하고 이를 통해 replay memory를 채운다. 이때 초기에는 done에 not을 붙여서 저장한다. 이유는 추후 q-learning 수식을 구현하는 부분에서 설명한다.

아래는 agent, memory, environment를 test하는 부분이다. 결과적으로 제시된 값들과 동일하게 나온 것을 확인하였고 구현한 코드가 정상적으로 작동함을 확인했다.

```
#test replay memory
tf.random.set_seed(2) #extract random value
keras.utils.set_random_seed(2)
test_agent = Agent_Net() #target network
test_mem = ReplayMemory(4) #replay memory
test_env = create_env() #environment
test_state = test_env.reset(seed=3) #state
init_memory(test_mem, test_env, test_agent, 4) #initialize memory
# There is no difference between if statement and else statement.
# When the value is 1, the state has 8 dimensions, so extracting 8 seems to be meaningful as a branch.
if SELECT_ENV==1:
    print(test_mem.get_batch(4)[0][0][:4].numpy())
    print(test_mem.get_batch(4)[1].numpy())
    print(test_mem.get_batch(4)[2][0][:4].numpy())
    print(test_mem.get_batch(4)[3].numpy())
    print(test_mem.get_batch(4)[4].numpy())
else:
    print(test_mem.get_batch(4)[0][0][:4].numpy())
    print(test_mem.get_batch(4)[1].numpy())
    print(test_mem.get_batch(4)[2][0][:4].numpy())
    print(test_mem.get_batch(4)[3].numpy())
    print(test_mem.get_batch(4)[4].numpy())

# finishing work
del test_agent, test_mem
test_env.close()
```

```
[[-0.04072088  0.18846463 -0.00277539 -0.32736924]
 [1 0 1 0]
 [-0.02550636 -0.00617038 -0.02836518 -0.04544564]
 [1. 1. 1. 1.]
 [1. 1. 1. 1.]
```

```
[[-0.01505842  1.4191642 -0.76619244  0.17035668]
 [1 0 1 0]
 [-0.03796177  1.427054 -0.7755038  0.08992036]
 [-1.0618883 -1.0290995 -1.9180926 -1.4797792]
 [1. 1. 1. 1.]
```

```

# for test
def test_training():
    state_b = tf.random.uniform((3,4))
    action_b = tf.random.uniform((3,)), minval=0, maxval=7, dtype=tf.int32)
    next_state_b = tf.random.uniform((3,4))
    reward_b = tf.random.uniform((3,))
    not_done_b = tf.random.uniform((3,))
    gamma = tf.random.uniform((1,))
    action_range = 7

    class test_agent:
        def __init__(self):
            pass
        def policy_q(self, x):
            return tf.reduce_sum(x, axis=-1, keepdims=True) # The way to obtain the q value through the state is through a Q network, but since this is a test, it is implemented as a simple sum.
        def target_q(self, x):
            return tf.reduce_sum(x, axis=-1, keepdims=True) # The way to obtain the q value through the state is through a target network, but since this is a test, it is implemented as a simple sum.

    agent = test_agent()

    ### START CODE HERE ###

    # get action probability with current (think of WHY!!) policy (b,a)
    curr_q = agent.policy_q(state_b) # calculate Q value

    # get action probability with target policy (b,a),
    # and then find the max Q value with it (b,)
    next_q = agent.target_q(next_state_b) # calculate Q value
    max_next_q = tf.reduce_max(next_q, axis=-1) # remove axis

    # calculate target reward (b,1)
    target_reward = reward_b + gamma * max_next_q + not_done_b

    # make one-hot actions (b,a) to filter out other actions
    action_v = tf.one_hot(action_b, depth=action_range)

    # make ground true labels for training (b,a)
    label_q = curr_q + (tf.expand_dims(target_reward,-1) - curr_q) * action_v
    ### END CODE HERE ###

    return label_q

```

위는 training의 테스트 버전이다. 올바른 training을 위해 q-learning 수식을 구현하고 테스트하기 위함이다. 실제 agent에서 얻어낸 action의 q값을 여기서 사용하지 않고 test_agent의 policy_q와 target_q를 통해 action을 가져왔다고 가정한다.


Curr_q를 현재 state를 통해 agent의 Q-network에서 뽑아서 얻었다고 가정한다. 이후 target-network에서도 마찬가지로 state를 통해 Q값을 뽑아내고 이중 max값으로 target의 reward를 계산한다. 이때 reward도 사전에 정의되었지만 원래는 replay memory에서 가져온 reward이다. 그리고 replay_memory에 done을 넣지 않고 not_done으로 넣는 이유가 바로 여기서 나타난다. Target_reward를 계산할 때 종료 flag가 활성화되면 우리는 next_state에 대한 계산을 할 필요가 없다. 따라서 not_done을 next_state를 통한 reward 계산 항에 곱해줌으로서 종료조건이 되었을때는 항이 사라지도록 구현한다.

이후 action을 one-hot 인코딩하는데 이 이유는 q값을 update할 때 특정 action에 따른 q값의 차이만이 반영되어야 하므로 action_v를 일종의 마스킹으로서 사용한다. 따라서 label_q는 함수의 최적화시 label로 사용되는데 수식을 살펴보면 target_reward에서 curr_q를 빼고 이를 action_v와 곱한 것을 다시 curr_q에서 빼는 방식으로 구현되어 있다. 이는 target_reward가 정답이라고 가정하고 이것에서

curr_q를 빼면 정답 q값과 현재 q값의 차이가 되는데 이 차이는 모든 action에 반영되면 안되고 현재 action에만 반영되어야 하므로 action_v를 곱해주어 현재 action이 아닌 부분은 모두 0이 되게끔 만든다. 이후 이 값을 curr_q에서 빼주어 이를 label로 사용하는 것이다.

```
[19] # Check if q-learning is working well
      tf.random.set_seed(2)

      ans = np.array([[1.1889474 , 1.1889474 , 1.1889474 , 1.1889474 , 1.1889474 , 1.1889474 , 0.40853113],
                      [2.713482  , 2.713482  , 2.4351463 , 2.713482  , 2.713482  , 2.713482  , 2.713482  ],
                      [3.3669732 , 1.4427134 , 3.3669732 , 3.3669732 , 3.3669732 , 3.3669732 , 3.3669732 ]])
      res = test_training()
      print('Training lable test passed.') if np.allclose(res,ans) else print('Training lable test failed.')
```

 Training lable test passed.

위는 테스트 결과를 정답과 비교하여 목적에 맞게 구현되었는지 확인하는 부분이다.

```

▶ # train network
def dqn_train(agent, batch, config):
    state_b, action_b, next_state_b, reward_b, not_done_b = batch #load from batch
    gamma = config.gamma

    ### START CODE HERE ###

    # get action probability with current (think of WHY!!) policy (b,a)
    curr_q = agent.policy_q(state_b)

    # get action probability with target policy (b,a),
    # and then find the max Q value with it (b,)
    next_q = agent.target_q(next_state_b)
    max_next_q = tf.reduce_max(next_q,axis=-1)

    # calculate target reward (b,1)
    target_reward = reward_b + gamma * max_next_q * not_done_b

    # make one-hot actions (b,a) to filter out other actions
    action_v = tf.one_hot(action_b,depth=action_range)

    # make ground true labels for training (b,a)
    label_q = curr_q + (tf.expand_dims(target_reward,-1) - curr_q) * action_v

    # training with model.fit()
    logs = agent.policy_q.fit(state_b, label_q, epochs=1, verbose=0)

    ### END CODE HERE ###

    loss = logs.history['loss'][-1]
    return loss

```

위는 테스트 버전과 동일하게 구현되었고 실제 훈련시 사용하도록 네트워크를 최적화할 fit함수와 state, action, next_state, reward, not_done은 실제로 replay_memory에서 얻고 이는 batch 단위로 가져온다는 부분이 달라진 부분이다.


```
[21] # Evaluation after training
def evaluate_policy(env, agent, num_avg):

    total_reward = 0.0
    episodes_to_play = num_avg
    for i in range(episodes_to_play): # Play n episode and take the average
        state = env.reset(env)
        done = False
        episode_reward = 0.0
        while not done:

            ### START CODE HERE ###

            action = agent.policy(state,0,False) # get an action with policy
            next_state, reward, done = env_step(env,action) # take action and observe outcomes

            ### END CODE HERE ###

            state = next_state
            episode_reward += reward
            total_reward += episode_reward
            average_reward = total_reward / episodes_to_play


    return average_reward
```

위는 policy를 평가하는 부분으로 학습된 상태를 확인하기 위해 exploring인자를 false로 하여 구현하였다. 평균 reward를 계산하여 추후 학습시에 종료조건을 결정하게 된다.

```
[22] # Exploration parameters for epsilon greedy strategy
class Epsilon:
    def __init__(self, max_episodes, decay_speed=1.0):
        self.explore_start = 1.0 # exploration probability at start
        self.explore_stop = 0.01 # minimum exploration probability
        self.decay_rate = decay_speed/max_episodes # exp decay rate for exploration prob (10/max ≈ 0.99)
        self.episode_cnt = 0

    def get_epsilon(self):
        eps = (self.explore_stop
              + (self.explore_start - self.explore_stop) * tf.math.exp(-self.decay_rate * self.episode_cnt)) # Decrease epsilon gradually (early exploratory, late follow learned)
        self.episode_cnt += 1
        return eps
```

위는 epsilon class로 exploring시에 사용할 epsilon값을 결정하고 학습에 따라 점점 감소하게끔 구현되어 있다. 따라서 초기에는 탐험적인 시도를 많이하고 이후에는 최적화가 되었다고 가정하여 네트워크의 값을 비교적 더 많이 사용하게 되는 효과를 가져온다.

```
 #set hyperparameter
class configuration:
    def __init__(self):
        self.gamma = 0.99 # discount rate
        self.lr = 2e-4 # learning rate

config = configuration()
```

위는 target_reward를 계산할 때 사용되는 gamma값과 learning_rate를 설정하는 부분이다. 훈련시 사용된다.

```
▶ #Replay memory size setting, batch size setting, related optimizer setting
max_steps = max_episodes * max_ep_steps
batch_size = b_size
memory_size = h_size

agent = Agent_Net()

memD = ReplayMemory(memory_size)
init_memory(memD, env, agent, memory_size) #replay memory init
epsF = Epsilon(max_episodes, 10.0) #epsilon setting
opt = tf.optimizers.Adam(learning_rate=config.lr, clipvalue=2.0)

agent.policy_q.compile(optimizer=opt, loss='mse', jit_compile=False) #precompile
```

위는 전체 학습의 step을 계산하고 agent, replay memory, Epsilon, optimizer등 구현했던 함수와 클래스를 사용하여 훈련에 준비하는 부분이다.

```
▶ #final main train code
#log
logs = keras.callbacks.History()
logs.history.update({'pi_loss': []})
logs.history.update({'reward': []})
logs.history.update({'e-steps': []})
logs.history.update({'vreward': []})

# variables for simulation
num_episodes = 0
val_episodes = 2 # exit condition

# variables for episode logging
pi_loss = 0.0
#initial reward and loss value
loss_sum = 0.0
epis_steps = 0
epis_reward = 0.0
eval_reward = -float('inf')

# initialize training variables
epsilon = 1.0
next_state = None
done = True

pbar = tqdm(range(max_steps), bar_format='{l_bar}{bar:10}{r_bar}{bar:-10b}') #progress bar
```

위는 훈련 초기에 결과들이 저장될 log와 훈련된 episode 수, 평가할 episode 수,

loss, reward 등등을 저장할 변수들을 선언하는 부분이다. 초기 epsilon은 1.0으로 설정되어있고 done은 True로 해둔 후 훈련에 진입한다. Tqdm 라이브러리를 통해 progress bar를 만든다.

```
for sim_steps in pbar:

    ### START CODE HERE ###

    state = env.reset(env) if done else next_state          # get the current state
    action = agent.policy(state, epsilon, exploring=True)     # find an action with e-greedy
    next_state, reward, done = env.step(env, action)         # take action and observe outcomes

    experience = (state, action, next_state, reward, not done) # pack observations into a new experience
    memD.put_experience(experience)                           # put a new experience to replay buffer

    batch = memD.get_batch(batch_size)                       # get a new batch from replay buffer
    step_pi_loss = dqn_train(agent, batch, config)           # train DQN for a step

    ### END CODE HERE ###

    loss_sum += step_pi_loss                                # accumulate policy loss for a step
    epis_reward += reward                                    # accumulate reward for a step
    epis_steps += 1                                          # increase the number of steps for an episode

    # episode termination conditions
    if epis_steps > max_ep_steps: done = True
```

위는 훈련이 진행되는 반복문이다. 초기 state는 done이 True이면 episode의 처음 반복이므로 environment를 초기화하고 얻은 state를 사용한다. False라면 next_state로 이후 state를 사용한다. Action은 구현한 agent의 policy를 따른다. Action을 통해 얻어낸 environment의 정보들을 replay memory에 저장하고 이를 batch 단위로 꺼내어 train에 사용하는 것을 볼 수 있다. 이를 반복하여 loss와 reward를 계산하고 이후 episode step이 최대 step을 넘어가면 done을 True로 만든다.

```

# summarize episode
if done:
    agent.target_update()                # update target network whenever episode ends
    memD.num_episodes += 1              # increase number of episode simulated
    epsilon = epsF.get_epsilon()         # update decay epsilon value

    pi_loss = loss_sum / epis_steps      # average policy loss for an episode

    pbar.set_postfix({'episode':num_episodes, 'loss':step_pi_loss, 'reward':eval_reward, 'steps':epis_steps, 'evaluating':val_episodes})
    eval_reward = evaluate_policy(env, agent, 1) # evaluate policy one time

    #log append
    logs.history['pi_loss'].extend([pi_loss])
    logs.history['reward'].extend([epis_reward])
    logs.history['e-steps'].extend([epis_steps])
    logs.history['vreward'].extend([eval_reward])

    loss_sum = 0.0
    epis_reward = 0.0
    epis_steps = 0
    num_episodes += 1

else: pass

pbar.set_postfix({'episode':num_episodes, 'loss':step_pi_loss, 'reward':eval_reward, 'steps':epis_steps})

# coditions to stop simulation
if eval_reward > goal_score:
    eval_reward = evaluate_policy(env, agent, val_episodes) # evaluate policy multiple times
if eval_reward > goal_score: break
if num_episodes > max_episodes: break


print('episodes:{0:5d}, loss:{1:7.5f}, val_reward {2:4.2f}'.format(num_episodes, pi_loss, eval_reward))
print('total steps:', sim_steps+1)

```

Done이 true가 되면 q-network의 가중치를 target-network의 가중치로 복사하고 epsilon 값을 감소시킨다. 이후 이번 episode에서 나온 loss나 reward같은 history를 log에 저장하고 다시 변수들을 초기화하는 과정을 거친다. 목표 reward에 도달했는지 확인하고 그렇지 않으면 새로운 episode를 시작하고 도달했다면 종료한 후 전체 훈련 step과 각종 정보를 출력한다.

 27% | 53191/200000 [1:23:03<3:17:05, 12.41it/s, episode=366, loss=422, reward=500, steps=0]
 episodes: 366, loss:65.78466, val_reward 491.00
 total steps: 53192

위는 cartpole에 대한 결과이다.

 10% | 40248/400000 [1:04:06<8:05:29, 12.35it/s, episode=77, loss=0.618, reward=208, steps=0]
 episodes: 77, loss:1.34991, val_reward 221.89
 total steps: 40249

위는 lunar lander에 대한 결과이다.

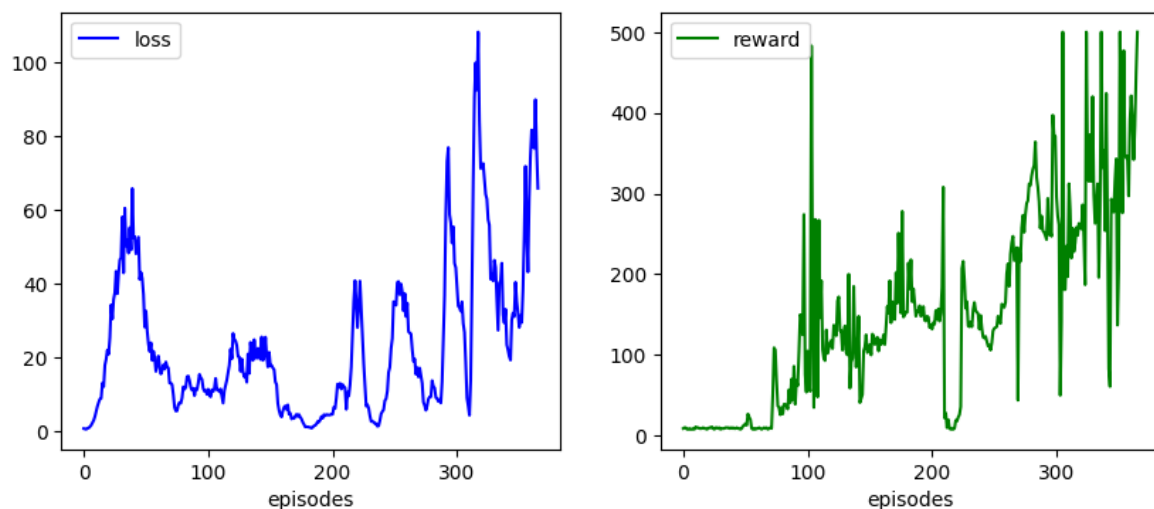
```

▶ # plot loss and accuracy
def plot_graphs(log_history, log_labels, graph_labels, graph_colors=['b-', 'g-']):
    num_graphs = len(log_labels)
    plt.figure(figsize=(5*num_graphs, 4))
    for i in range(num_graphs):
        plt.subplot(1, num_graphs, i+1)
        plt.plot(log_history[log_labels[i]], graph_colors[i], label=graph_labels[i])
        plt.xlabel('episodes')
        plt.legend()
    plt.show()
    return

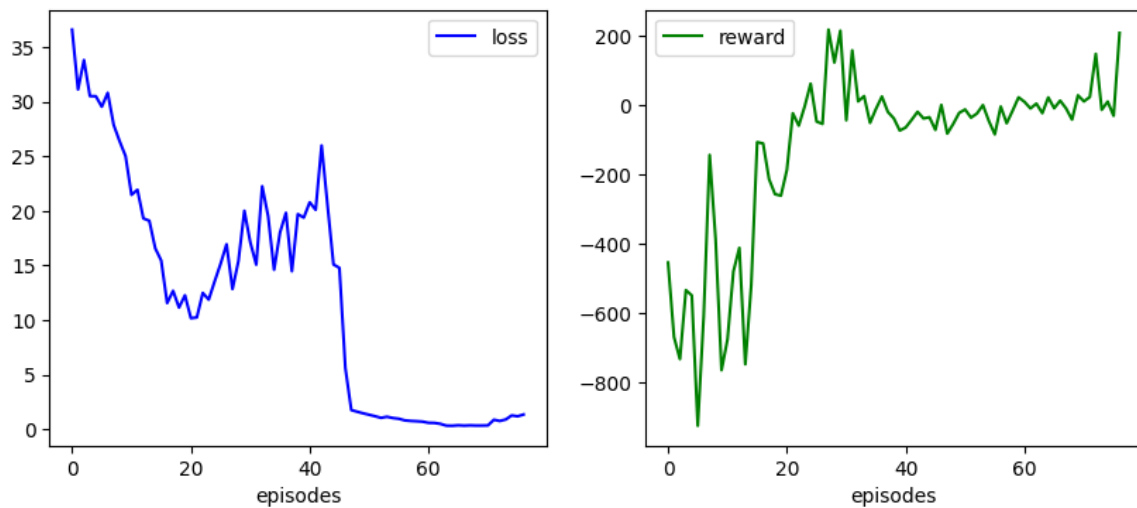
log_labels = ['pi_loss', 'vreward'] # Correlation between loss and reward
label_strings = ['loss', 'reward']
label_colors = ['b-', 'g-'] # blue and green
plot_graphs(logs.history, log_labels, label_strings, label_colors)

```

위 코드를 통해 episode에 따라 loss와 reward를 확인할 수 있고 상관 관계를 확인할 수 있다.



위는 cartpole에 대한 결과이다. 점점 reward가 높아져 탈출하게 되었다.



Lunar lander에 경우이다. 비교적 빠르게 학습되었다.

```
#evaluate agent
evaluate_episodes = 20
sum_episode_rewards = 0.0
pbar = tqdm(range(evaluate_episodes))

for i in pbar:
    sum_episode_rewards += evaluate_policy(env, agent, 1) #Evaluate 20 episodes once each

env.close()

print('Evaluation Result:', sum_episode_rewards/evaluate_episodes)
```

100% 20/20 [00:37<00:00, 1.90s/it]
Evaluation Result: 483.25

위는 cartpole을 20의 episode로 평가했을 때 평균 reward를 보여준다. 450의 reward 종료조건을 평균적으로 만족하는 것을 확인한다.

100% 20/20 [01:09<00:00, 3.32s/it]
Evaluation Result: 127.19640124585742

위는 lunar lander을 20의 episode로 평가했을 때 평균 reward를 보여준다. 200의 reward 종료조건을 평균적으로 만족하진 못했다.

```

▶ #save video
env = create_env()
env = wrappers.RecordVideo(env, video_folder='./gym-results/', name_prefix=res_prefix)

eval_reward = evaluate_policy(env, agent, 1) #evaluate 1

print('Sample Total Reward:', eval_reward)

env.close()

```

위는 하나의 episode를 평가했을 때 결과를 비디오로 저장하고 reward를 출력하는 코드이다.

⇒ Sample Total Reward: 500.0

위는 cartpole의 결과 아래는 lunar lander의 결과이다.

Sample Total Reward: 224.52625436231193

```

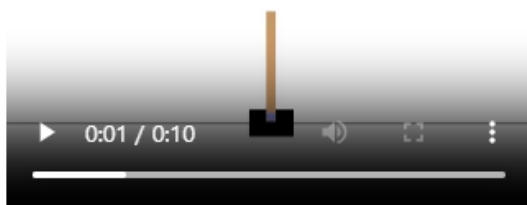
[29] #show video
from IPython.display import HTML
from base64 import b64encode

def show_video(video_path, video_width = 320):
    video_file = open(video_path, "r+b").read() #load saved video
    video_url = f"data:video/mp4;base64,{b64encode(video_file).decode()}" #encode,decode
    return HTML(f'<video width={video_width} controls><source src="{video_url}"></video>""')#put link

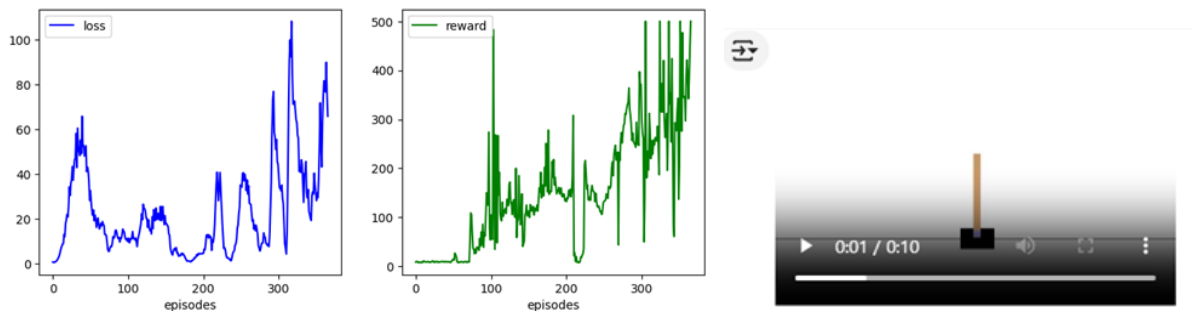
show_video('./gym-results/' + res_prefix + '-episode-0.mp4')

```

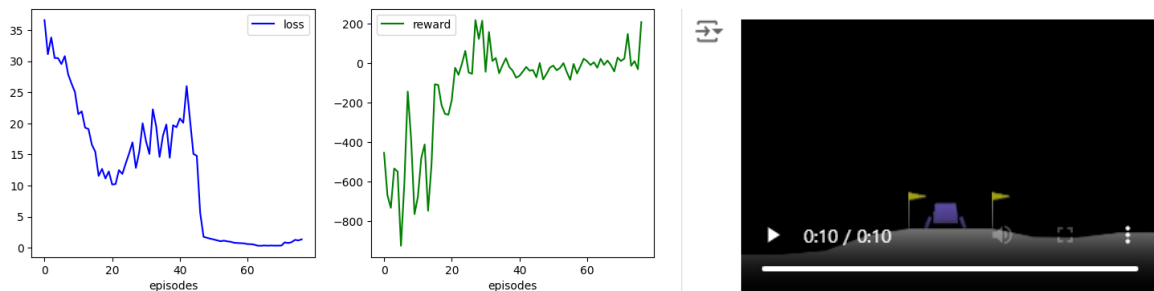
위는 저장된 video를 보여주는 코드이다. 아래와 같이 출력되었다.



Discussion



그래프에 대한 논의를 추가로 진행한다. 위의 cart pole에 대한 훈련 그래프이다. 초기 80회까지는 학습이 거의 잘 되지 않다가 급격하게 100회 지점에서 솟구치는 모양을 보인다. 이때 종료 조건을 만족했으나 이후 평가 부분에서 만족하지 못해 다시 학습이 재개되는 모습을 보였다. 이후에는 종료조건을 자주 만족했으나 평가에서 만족하지 못하는 모습을 많이 보였고 영상을 보았을 때 좌우를 많이 반복하는 것을 확인했다.



위의 lunar lander에 대한 그래프이다. Cart pole과는 달리 8개의 state를 정보를 가진 environment이고 action도 더 많은데 불구하고 더 빠른 최적화를 이루었다. Loss가 매우 빠르게 감소하였고 reward 또한 편차가 크지 않고 적절히 수렴하였다. 영상 또한 빠르게 내려와 중앙부에 안정적으로 착지하는 모습을 보였다.

더 복잡한 lunar lander가 더 빠르고 잘 수렴한 이유는 보상 조건의 정밀도가 크게 작용했다고 생각한다. Cart pole같은 경우 오랫동안 중심을 잃지 않는 것이 유일한 보상이다. 일정 범위를 벗어나거나 중심을 잃어도 깎이는 보상은 없으며 종료될 뿐이다. 이렇게 결과적으로 보상이 단순하게 구성되어 있어 최적화가 오래 걸렸다고 예상한다. Lunar lander같은 경우에는 착륙패드에서 멀어질 때 보상이

감소하고 가까워질 때 증가하며 측면부나 메인 엔진을 점화할 때 또한 보상이 감소한다. 또 다리가 지면에 닿을 때 보상이 존재하는 등 다양하게 구성된 보상이 더 빠르고 안정적으로 학습에 도움이 되었을 것이다.

두번째로 replay memory의 크기에 있다.

Cartpole은 replay memory의 크기가 1000인 반면 lunar lander는 10000이다. 따라서 더 많은 경험이 축적되어 있었을 것이고 이것이 학습에 도움이 되었을 것이라고 생각한다.

이번 과제에서는 DQN을 직접 구현해보고 다양한 environment에서 실제 잘 동작함을 확인할 수 있었다. 모든 시간의 state가 주어지지 않아도 현재 state와 다음 state의 관계성만으로도 모델이 학습될 수 있다는 것을 확인할 수 있어 유익했다.