인공지능프로그래밍

Lab 08: Deep Q Learning Network (a.k.a DQN)

학 번 : 2021202058

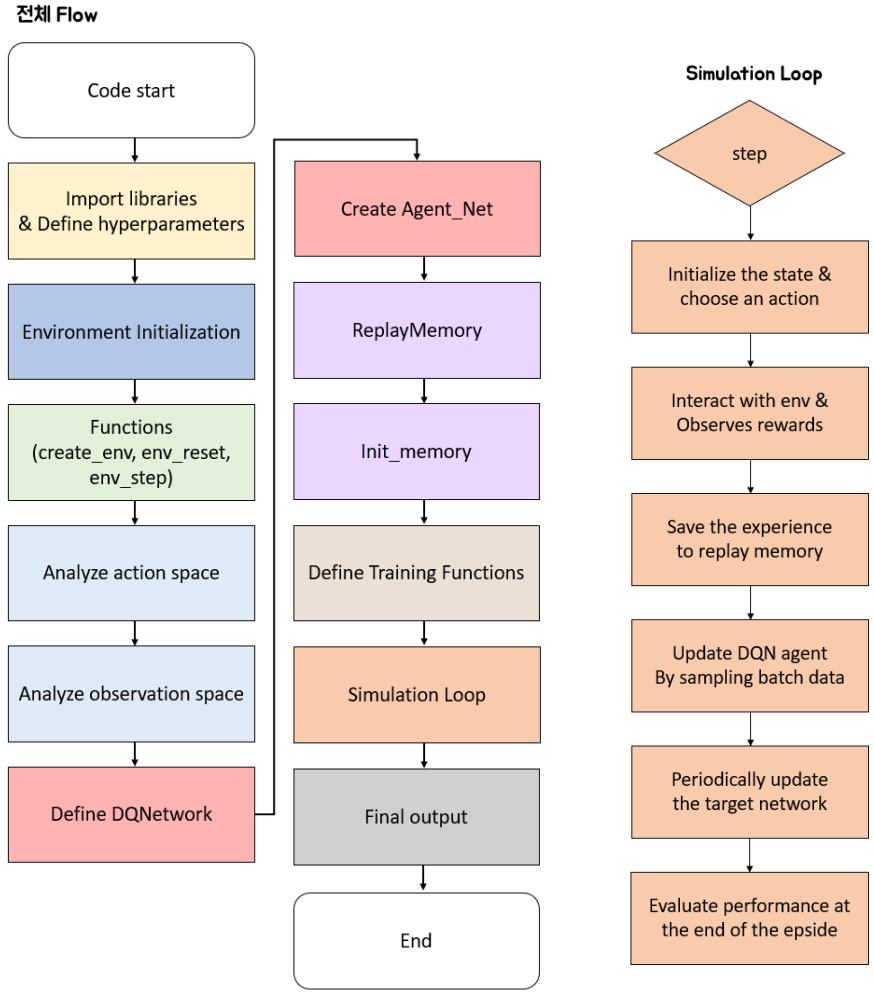
성 명 : 송채영

날 짜 : 2024.12.03

Lab Objective

이번 과제에서는 Deep Q-learning Network을 직접 구현해보며 DQN의 동작방식을 이해해본다. 구현한 모델을 바탕으로 gymnasium library의 Cart Pole, Lunar Lander, 2개의 environment에서 적용해보며 결과를 시각화해서 확인한다. 또한 시각화한 결과, 그래프, mp4 결과를 분석해보며 Q-learning을 각 environment에 적용되는 방식을 이해한다.

Program flow



우선 왼쪽 사진은 전체적인 코드의 흐름이다. 노란색 박스는 setup 및 configuration에 해당한다. Gymnasium, numpy, tqdm등의 필요한 라이브러리를 가져오며 CartPole, Lunar Lander에 대한 하이퍼파라미터를 정의한다. 하이퍼파라미터에는 에피소드 수, 학습률, 목표 점수 등이 있다. 이어서 파랑색 박스는 Environment Initialization에 해당한다. 사용자의 선택인 SELECT_ENV에 따라 적절한 gymnasium 환경을 생성하고 구성한다. 연두색 박스는 Function 부분이다. 선택한 환경을 생성하고 반환하는 create_env(), 환경을 재설정

하고 초기상태를 반환하는 env_reset(), action을 실행해서 next_state, reward, done flag를 반환하는 env_step() 함수를 만드는 과정에 해당한다. 하늘색 박스는 Environment state, action space analysis 부분에 해당한다. 우선 action space(discrete or continuous)를 분석해서 action dimensions, range 및 배치 형태를 설정한다. 이어서 observation space를 분석해서 상태 공간의 크기와 구조를 결정한다. 다음으로 핑크색 박스는 모델 정의 및 네트워크 구성에 해당한다. 우선 DQNet 부분은 DQnetwor를 생성하며 상태 입력을 받아 Q값, action value를 출력하는 다중 신경망이다. 다음으로 Agent_Net은 학습을 수행하는 에이전트로 policy_q와 target_q를 정의한다. 다음으로 보라색 박스 부분은 Replay Memory와 Init Memory로 나눌 수 있는데, ReplayMemory는 Experience Replay 메모리를 관리하는 것이고 init_memory는 초기 경험 데이터를 수집하는 것이다. 이어서 갈색 박스는 training 부분으로 test_training, dqn_train, evaluate_policy 이렇게 3개가 있다. 첫 번째는 Bellman 방정식 기반의 target Q-값 을 계산하며 두 번째는 정책 네트워크를 학습한다. 마지막은 에이전트의 성능을 평가한다. 주황색 부분은 simulation loop 부분으로 오른쪽 사진에 자세하게 풀었다. 시뮬레이션 루프를 통해 에이전트를 훈련하며, 각 단계에서 상태를 초기화하고 행동을 선택한 후 환경과 상호작용하며 보상을 관찰한다. 이후 경험을 리플레이 메모리에 저장하고 배치 데이터를 샘플링하여 DQN 에이전트를 업데이트한다. 타겟 네트워크는 주기적으로 업데이트하며 에피소드가 종료되면 성능을 평가한다. 회색 박스는 시뮬레이션 종료 후 학습 결과인 에피소드 수, 손실, 최종 보상을 출력하도록 하는 흐름이다.

DQN

Deep Q-learning Network, DQN은 Q-learning과 딥러닝 기술을 결합한 강화 학습 알고리즘이다. 모든 상태와 행동에 대한 큐 함수 값을 따로 저장하여 이를 이용해 학습을 수행하고 행동을 결정해 매우 많은 상태와 행동이 존재하는 환경에서 사용이 어렵다는 점에서 기존 Q learning의 문제점으로 작용하는데, DQN을 사용하면 CNN을 통해 각 상태와 행동에 대한 큐 함수 값을 근사해 모든 상태와 행동에 대한 큐 함수 값을 따로 저장하지 않고 큐 함수 값에 대한 추정을 수행할 수 있다. 즉 많은 상태와 행동이 존재하는 환경에서도 학습이 가능하다.

DQN 알고리즘은 현재 상태에 여러 행동에 대한 Q 함수 값을 근사하고 최적의 행동을 선택한다. 수행한 행동에 따라 보상을 받고, 그에 따른 타겟 값(Q 함수 값)과 손실함수를 계산하며 타겟 값과 예측 값의 차이가 최소화되도록 학습을 수행한다.

DQN 알고리즘에는 Experience Replay와 Target Network가 있는데 Experience Replay는 경험 데이터를 매 스텝마다 리플레이 메모리에 저장하는 것을 말한다. 경험 데이터는 현

재 상태, 현재 행동, 보상, 다음 상태, 게임 종료 정보를 뜻한다. 매 스텝마다 리플레이 메모리의 경험 데이터를 임의로 일정 개수만큼 추출해 mini-batch 학습을 수행하도록 한다. 이렇게 하면 데이터의 상관관계를 고려하지 않고 랜덤으로 샘플링 해서 전체적인 데이터의 경향성을 잘 반영한 결과를 얻을 수 있다. 다음으로 Target Network에 대한 설명이다. DQN에서는 구조가 완전히 동일한 두 종류의 네트워크를 사용하는데, 일반 네트워크와(행동을 결정하거나 큐 함수 값을 예측) 타겟 네트워크(학습에 필요한 타겟 값을 계산)가 있다. 일반 네트워크는 매 스텝마다 업데이트가 이루어지며, 행동에 따른 Q 함수 값과 최적의 행동을 계산한다. 즉 같은 입력을 해도 매 스텝마다 네트워크가 다른 답을 줄 수도 있음을 의미한다. 타겟 네트워크는 매 특정 스텝마다 한번씩 일반 네트워크를 복제하며 타겟 값을 계산한다. 이렇게 함으로써 학습의 목표가 되는 타겟 값을 최대한 일정하게 유지하며 안정적인 학습이 가능하도록 한다.

Result

```
# Check if this code runs in Colab
RunningInCOLAB = 'google.colab' in str(get_ipython())

# Installing the required library and Import tqdm for notebook if Colab
if RunningInCOLAB:
    !pip install swig
    !pip install gymnasium
    !pip install gymnasium[box2d]
    from tqdm.notebook import tqdm
else:
    from tqdm import tqdm
```

```
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 49.5 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 30.2 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 14.3 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
```

```
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=2376476 sha256=00c7e58fb718ea6be6c15b6f1f1a1a73566c13350ec5e0938efe5bcfd325ce8c
  Stored in directory: /root/.cache/pip/wheels/db/8f/6a/eaaadf056fba10a98d986f6dce954e6201ba3126926fc5ad9e
Successfully built box2d-py
Installing collected packages: box2d-py
Successfully installed box2d-py-2.3.5
```

위 코드는 Google Colab 환경에서 실행 중인지 확인한 후 Colab인 경우 필요한 라이브러리와 tqdm을 가져온다. Colab이 없을 경우 표준 tqdm을 가져온다. 설치 로그를 살펴보면 패키지 설치 및 Box2D 기반 환경도 성공적으로 설치된 것을 확인할 수 있다.

```
import os
# Setting the Keras backend to TensorFlow
os.environ["KERAS_BACKEND"] = "tensorflow"

# Importing necessary libraries
import numpy as np
import tensorflow as tf
import keras
import matplotlib.pyplot as plt

import gymnasium as gym
from gymnasium import wrappers

from collections import deque
import random
```

```
# List available GPU devices
physical_devices = tf.config.list_physical_devices('GPU')
print(physical_devices)
# Enable memory growth for the first GPU
try:
    tf.config.experimental.set_memory_growth(physical_devices[0], True)
# Print an error message if no GPU is detected
except:
    print('GPU is not detected.')
```

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

```
# Check the installed version
gym.__version__
```

```
'1.0.0'
```

Keras 백엔드를 Tensorflow로 설정한 후 필요한 라이브러리를 가져오는 코드이다. 코드에서 사용한 라이브러리를 간단하게 설명하면, numpy는 배열 및 수학 연산을 처리하는 데 사용되는 수치 계산용 라이브러리, tensorflow는 Google에서 개발한 기계 학습 프레임워크, keras는 TensorFlow(또는 다른 백엔드) 위에서 실행되는 고급 신경망 API, matplotlib.pyplot은 그래프 및 시각화 생성에 사용되는 플로팅 라이브러리, gymnasium은 다양한 RL 환경을 생성하고 상호 작용하는 데 사용되는 강화 학습(RL) 라이브러리 등이 있다. 이어서 사용 가능한 GPU 장치 목록을 확인하는 코드로 출력 결과를 확인하면 한 개의 GPU 장치를 사용할 수 있음을 알 수 있다. 이어서 gymnasium 라이브러리 버전을 확인하는 코드로, 설치된 버전이 1.0.0임을 알 수 있다.

```
# select evironment
# Discrete Action Space:    0 for Cartpole, 1 for LunarLander

SELECT_ENV = 0
```

```
# If the selected environment is 0 (CartPole)
if SELECT_ENV == 0:
    # Set environment configurations
    env_name, res_prefix = 'CartPole-v1', 'cart'
    max_episodes, max_ep_steps, goal_score = 400, 500, 450
    b_size, h_size = 128, 1000
    network_type, state_width, state_height, state_depth = 'dense', 0, 0, 0
    kwargs = {'render_mode':'rgb_array'}
# If the selected environment is 1 (LunarLander)
elif SELECT_ENV == 1:
    # Set environment configurations
    env_name, res_prefix = 'LunarLander-v2', 'lunD'
    max_episodes, max_ep_steps, goal_score = 400, 1000, 200
    b_size, h_size = 128, 10000
    network_type, state_width, state_height, state_depth = 'dense', 0, 0, 0
    kwargs = {'continuous':False, 'render_mode':'rgb_array'}
# If the environment selection is invalid
else: assert False, 'environment selection error'

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

SELECT_ENV 값에 따라 CartPole 또는 LunarLander, 두 가지 강화 학습 환경 중에서 택할 수 있으며 선택한 환경에 따라 다른 구성이 설정된다. CartPole인 경우 환경은 CartPole, 학습을 위한 최대 에피소드 수는 400, 에피소드 당 최대 스탭 수는 500, 목표 점수는 450, 배치 크기는 128, 신경망의 숨겨진 레이어 크기는 1000, dense 레이어를 사용해야 함 등으로 구성이 설정되었고, LunarLander의 경우 학습을 위한 최대 에피소드 수는 400, 스탭 수는 1000, 목표 점수는 200 등으로 구성이 설정된 것을 확인할 수 있다. SELECT_ENV가 0 또는 1이 아닐 경우 오류 메시지를 표시한다.

create_env 함수는 환경을 생성하고 반환하며 Gymnasium의 make 메소드를 사용해 환경을 초기화한다.

```
/usr/local/lib/python3.10/dist-packages/gymnasium/envs/registration.py:517: DeprecationWarning: WARN: The environment LunarLander-v2 is out of date. You should consider upgrading to version `v3`.
  logger.deprecation(
---------------------------------------------------------------------------
DeprecatedEnv                             Traceback (most recent call last)
<ipython-input-8-89891ef0812d> in <cell line: 3>()
      1 # Create the environment, reset it to get the initial state,
      2 # and take a random step to get the new state, reward, and done flag
----> 3 env = create_env()
      4 state = env_reset(env)
      5 state, reward, done = env_step(env, env.action_space.sample())

                                  3 frames
/usr/local/lib/python3.10/dist-packages/gymnasium/envs/registration.py in _check_version_exists(ns, name, version)
    429
    430     if latest_spec is not None and version < latest_spec.version:
--> 431         raise error.DeprecatedEnv(
    432             f"Environment version v{version} for `{get_env_id(ns, name, None)}` is deprecated. "
    433             f"Please use `{latest_spec.id}` instead."

DeprecatedEnv: Environment version v2 for `LunarLander` is deprecated. Please use `LunarLander-v3` instead.
```

주어진 코드에서 LunarLander-v2는 지원이 안되었기 때문에 -v3으로 바꾸어 진행하였다.

```
# Function to reset the environment
def env_reset(env):
    observation = env.reset()
    state = observation[0] if type(observation)==tuple else observation
    return state

 # Function to take a step in the environment
def env_step(env, action):
    observation = env.step(action)
    state = observation[0]
    reward = observation[1]
    done = observation[2] or observation[3] if len(observation)>4 else observation[2]
    return state, reward, done
```

```
# Create the environment, reset it to get the initial state,
# and take a random step to get the new state, reward, and done flag
env = create_env()
state = env_reset(env)
state, reward, done = env_step(env, env.action_space.sample())
```

env_reset 함수는 환경을 초기 상태로 재설정 하고 환경에 따라 튜플일 수 있는 초기 관찰을 반환한다. env_step 함수는 환경에서 조치를 취하고 새로운 상태, 보상, 에피소드 완료 여부를 반환한다.

앞서 정의한 create_env 함수를 사용해 환경을 생성하고 환경을 재설정하고 초기상태를 state에 저장한다. 이어서 환경에서 임의의 작업을 수행하고 결과가 env_step에 전달되어 새로운 상태, 보상, 에피소드 완료 여부를 반환한다.

```
# Set action space parameters based on its type
action_shape = env.action_space.shape
action_space_type = type(env.action_space)

# For discrete actions: define action range and shape
if action_space_type==gym.spaces.discrete.Discrete:
    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 continuous actions: define action range, bounds, and shape
elif action_space_type==gym.spaces.box.Box:
    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])
    num_actions = action_dims   # number of actions is action dimension for CONTINUOUS actions
else: assert False, 'other action space type are not supported'
```

```
# Set observation space parameters based on its type
observation_space_type = type(env.observation_space)
observation_shape = env.observation_space.shape

# For discrete observations: set number of states
if observation_space_type==gym.spaces.discrete.Discrete:
    observation_shape = (1,)
    num_states = env.observation_space.n
# For continuous observations: set number of states
elif observation_space_type==gym.spaces.box.Box:
    num_states = observation_shape[0]
else: print('observation space type error')

# Set batch shapes for state and value
state_shape = observation_shape
state_batch_shape = tuple([None]+[x for x in observation_shape])

value_shape = (1,)
num_values = 1
```

env.action_space type을 확인하여 sapce가 discrete인지 continuous인지를 확인하는 코드이다. Discrete인 경우 gym.spaces.discrete.Discrete, 환경에 가능한 행동의 유한한 집합이 있다고 가정한다. Action space의 유형, action 형태, action의 차원, action의 범위 등을 설정하고 action batch shape을 (none, action_range)로 설정하여 두 번째 차원이 action space의 크기에 해당하는 action batch를 처리할 수 있게 한다. Continuous인 경우 gym.spaces.box.Box, 동작이 실제값의 벡터로 표현된다고 가정한다. Action의 범위는 action 값의 상한과 하한의 차이를 계산한 후 크기를 조정하는데 사용된다. 두 케이스 모두 아닌 경우 오류를 출력한 후 observation space type을 확인한다.

이어서 observation sapce 부분으로, gym.spaces.discrete.Discrete 유형은 observation sapce가 유한한 개수의 정수값으로 이루어진 경우로 observation sapce 형태는 크기를 1로 설정하고 가능한 상태의 개수를 저장한다. gym.spaces.box.Box 유형은 연속적인 값으로 이루어진 경우이다. gym.spaces.tuple.Tuple 이 유형은 여러 개의 관찰 공간을 조합한 경우이며 지원되지 않는 관찰 공간의 경우 알맞은 출력문을 넣어주었다. 이후 환경의 타입에 따라 state shape과 batch shape을 설정하였다. 마지막으로 value 크기를 설정한다.

아래의 코드는 action space 정보 및 observation space의 정보 및 값 형태를 포함하여 환경의 다양한 구성 요소에 대한 정보를 출력하는 코드이다.

- Cart Pole

```
# Print details about action space, observation space, state space, and value shape
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)
```

```
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
```

주요 부분만 이야기 하면 CartPole은 Discrete action space을 사용하며 작업 공간은 카트를 왼쪽으로 밀기, 오른쪽으로 밀기 이렇게 2가지로 구성된다. 액션 형태는 0 또는 1 두 개의 개별 액션을 나타내며 작업 차원은 1차원이다. 관찰 형태는 4가지 값으로 카트 위치, 카트 속도, 기울어진 각도, 각속도에 해당한다. 상태 공간은 관찰 공간과 동일한 형태인 4차원이다.

- Lunar Lander

```
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
```

LunarLander의 주요 부분만 살펴보면 Discrete action space를 사용하고 작업 공간은 아무것도 하지 않음, 왼쪽 엔진 발사, 오른쪽 엔진 발사, 주엔진 발사 이렇게 4가지가 존재하며 관찰 공간은 x 위치, y 위치, x 속도, y 속도, 각도, 각속도, 왼쪽 엔진 상태, 오른쪽 엔진 상태 이렇게 8개가 있다.

Exercise: Define Deep Q-network (TensorFlow)

A NN of three fully-connected layers is enough for classic control problems.

Parameters for layer definition are:
hiddens = (unit # for layer1, unit # for layer2),
act_fn: activation function,
out_fn: activation function for output layer,
init_fn: kernel initialization function

```
# DQNet defines a deep Q-network model
def DQNet(hiddens, act_fn, out_fn, init_fn):    # hiddends = (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
    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') # Create Keras model object
    return model

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

Deep Q-Network(DQNet) 모델을 정의한 코드로, Q-learning Agent의 구조를 정의한다. 입력 형태는 환경의 상태 크기를 나타내는 state_shape에 의해 결정되며 cartpole의 경우 4, lunarlander의 경우 8이다. Hidden layer 부분은 dense 레이어를 사용하며 출력 레이어의 경우 작업의 Q 값을 나타낸다. Build_DQNet 함수는, 32개 단위로 구성된 히든 레이어, 활성화는 relu, output 활성화는 linear, 초기화는 he_uniform, 이러한 매개변수를 바탕으로 모델을 생성한다.

아래는 모델의 구조를 출력한 것으로 입력 레이어의 모양은 환경에 따라 다르며 cartpole의 경우 (none, 4)이고 lunarlander의 경우 (none, 8)이다. None은 배치 크기가 다양할 수 있다는 것을 나타낸다. 이때 매개변수는 입력 레이어이므로 0개가 있다. 이어서 첫 번째 hidden layer 부분에서 32개 단위가 있는 것을 볼 수 있고 매개변수는 cartpole의 경우 4*32+32 = 160, lunarlender의 경우 8*32+32 = 228인 것을 확인할 수 있다. 두 번째 hidden layer 부분에서도 32개 단위가 있으며 이전 레이어 출력 * 32 + 32로 계산했을 때 1056이 된다. 출력 레이어에서는 작업 수에 따라 다르므로 2 또는 4가 되며 매개변수는 32 * num_actions + num_actions에 따라 각각 66, 132가 되는 것을 확인할 수 있다.

- Cart Pole

```
# Create the DQNet model and display
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)

- Lunar Lander

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)

```
# Agent_Net defines a reinforcement learning agent with policy and target networks for Q-learning
class Agent_Net:
    def __init__(self):
        self.policy_q = build_DQNet()                       # Build policy network
        self.target_q = build_DQNet()                       # Build target network
        self.target_update()                                # Copy weights from policy to target

    def policy(self, state, epsilon, exploring):            # Epsilon-greedy policy if exploring
        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.argmax(action_q[0])             # Find an action for maximum q value
            else:                                           # Explore else
                action = tf.random.uniform((), 0, action_range, tf.int64)       # Random action
        else:                                               # Else greedy policy (exploitation)
            action_q = self.policy_q(state_input)           # Get actions for input state
            action = tf.argmax(action_q[0])                 # 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 Deep Q-learning을 사용해 강화학습 에이전트를 정의하는 코드이다. policy network와 target network 이렇게 두 개의 신경망을 사용하는데 policy network는 현재 상태에 대한 Q값을 예측하며 target network는 학습의 불안정성을 줄이기 위해 주기적으로 업데이트 된다. 에이전트는 epsilon-greedy policy을 사용해 Exploration, Exploitation을 선택할 수 있다.

```
# Test_policy simulates an epsilon-greedy policy for action selection
def test_policy(exploring):

    epsilon = 0.1                           # Set epsilon value for exploration
    state_input = tf.random.uniform((1,3))  # Randomly generate a state input
    action_range = 7                        # Number of possible actions

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

    ### START CODE HERE ###

    if exploring:                                           # If using epsilon-greedy policy
        if tf.random.uniform(()) > epsilon:                 # Exploit if random > epsilon
          action_q = policy_q(state_input)                  # Get Q-values for input state
          action = tf.argmax(action_q[0])                   # Find an action for maximum q value
        else:                                               # Explore else take random action
          action = tf.random.uniform((), 0, action_range, tf.int64) # Random action selection
    else:                                                   # Else greedy policy (exploitation)
         action_q = policy_q(state_input)                   # Get Q-values for input state
         action = tf.argmax(action_q[0])                    #  Find an action for maximum q value

    ###  END CODE HERE  ###

    return action # Return selected action

tf.random.set_seed(2) # Set seed for reproducibility
# Print actions when exploring
for _ in range(10): print(test_policy(True).numpy(), ' ', end='')
# Print actions when exploiting
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
```

작업 선택을 위해 epsilon-greedy policy을 시뮬레이션 하는 코드로 주어진 상태에 대해 무작위 Q 값을 생성하는 policy_q를 사용한다. Exploirng은 epsilon-greedy policy를 사용할지 아님 greedy policy를 사용할지 결정하는 플래그로 이것이 true인 경우 90%의 확률로 가장 높은 Q-값에 해당하는 동작이 선택되며 10%의 확률로 무작위 행동이 선택된다. False인 경우 항상 Q값이 가장 높은 작업을 선택한다.

실행 결과는 탐색 시에는 무작위 작업과 Q 값이 가장 높은 작업이 모두 나타나야하며 탐색하지 않을 때에는 Q 값이 가장 높은 작업만 선택되어야한다.

Define and Initialize Replay Memory

The replay memory (or replay buffer) is implemented with `deque`, which maintians the fixed number of elements by discarding the oldest element automatically.
The inputs of the `put_experience` function are from environments and the outputs of the `get_batch` function are fed to the NN. Therefore the output types should be tensors.

```python
# ReplayMemory stores and manages experiences for replay buffer in reinforcement learning
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)) # Store experience
        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) # Randomly sample experiences

        # Extract and store individual components of each sample
        for sample in sample_batch:
            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)

        # Convert batches to tensors
        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 the batch of experiences
```

강화 학습 에이전트에 대한 경험을 저장하고 관리하는 데 사용되는 ReplayMemory 클래스를 정의한 코드로, 초기화 부분에서 memory_size가 있는 deque는 경험을 저장하는데 사용되므로 새 경험이 추가될 때 오래된 경험이 제거된다. 이후 put_experience 부분이 경험이 추가되는 부분으로 경험 튜플을 가져와 버퍼에 추가하고 버퍼는 deque의 maxlen으로 인해 가장 오래된 항목을 제거할 수 있다. 버퍼에서 저장된 수의 경험을 무작위로 샘플링 하고 샘플링된 경험의 구성요소를 상태, 작업, 다음 상태, 보상, 완료플래그와 같은 배치로 추출하고 분리한다. 이후 훈련 루프에서 직접 사용할 수 있는 텐서의 튜플로 반환한다.

아래의 코드는 재현성을 위한 코드로 무작위 시드를 설정하고, 에이전트, 메모리 및 환경을 초기화하고, 초기 경험으로 메모리를 채우고, 메모리 배치를 샘플링 하고, 배치의 특정 구성 요소를 조건부로 print하여 보여주는 것이다. 결과를 살펴보면 cartpole의 경우 첫 번째 벡터는 state, 두번 째 배열은 action, 세번쨰 벡터는 next state, 네 번째 배열은 reward, 다섯 번째 배열은 done에 해당한다. Lunarlender의 경우 첫 번째 및 세 번째 벡터는 state 및 next state를 나타내며 네 번째 배열에서 reward가 음수로, 특정 행동이 부정적인 결과를 초래했음을 나타낸다.

- Cart Pole

```
# Set random seeds for reproducibility and initialize components for testing
tf.random.set_seed(2)
keras.utils.set_random_seed(2)
test_agent = Agent_Net()
test_mem = ReplayMemory(4)
test_env = create_env()
test_state = test_env.reset(seed=3)
init_memory(test_mem, test_env, test_agent, 4)

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())

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.]
```

- Lunar Lander

```
[-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.]
```

아래의 코드는 Q값, 목표 보상을 계산하고 에이전트 training을 위한 정답 라벨을 생성하는 함수이다. 우선 state_b는 각각 크기가 4인 3개의 무작위 상태를 배치하며 action_b는 각 상태에 대해 수행되는 작업을 나타내는 0~6 사이의 값을 가지는 3개의 무작위 작업 배치이다. next_state_b는 각각 크기가 4인 3개의 무작위 다음 상태 배치, reward_b는 3개의 무작위 보상 배치, not_done_b는 각 에피소드의 종료 여부를 나타내는 플래그이다. test_agent 클래스는 DQN에서 사용되는 Q network와 target Q network를 시뮬레이션 하기 위한 클래스로 policy_q는 policy network에서 Q 값 추정을 시뮬레이션하며 target_q는 target network에서 Q 값 추정을 시뮬레이션 한다. Q 값 계산은 벨만 방정식을 이용하며 curr_q는 정책 네트워크에서 얻은 현재 상태에 대한 Q값, next_q는 대상 네트워크

에서 얻은 다음 상태에 대한 Q 값이다. target_reward는 bellman 방정식을 사용하여 Q 값을 업데이트한다. 이후 action_v는 원-핫 인코딩 형식으로 변환되므로 수행되지 않은 action에 해당하는 Q 값들을 필터링 할 수 있다. label_q는 정답 레이블로 수식을 기반으로 구현하였다.

```
# Computes Q-values, target rewards, and generates ground truth labels for training
def test_training():
    state_b = tf.random.uniform((3,4)) # Batch of states (b, s)
    action_b = tf.random.uniform((3,), minval=0, maxval=7, dtype=tf.int32) # Batch of actions (b,)
    next_state_b = tf.random.uniform((3,4)) # Batch of next states (b, s)
    reward_b = tf.random.uniform((3,)) # Batch of rewards (b,)
    not_done_b = tf.random.uniform((3,)) # Indicator for whether the episode has ended (b,)
    gamma = tf.random.uniform((1,)) # Discount factor (scalar)
    action_range = 7 # The number of possible actions

    class test_agent:
        def __init__(self):
            pass
        def policy_q(self, x):
            return tf.reduce_sum(x, axis=-1, keepdims=True) # Q-value from policy network (b, 1)
        def target_q(self, x):
            return tf.reduce_sum(x, axis=-1, keepdims=True) # Q-value from target network (b, 1)

    agent = test_agent()

    ### START CODE HERE ###

    # Get action probability with current (think of WHY!!) policy (b,a)
    # Get the current Q-values (b,1) from the policy network
    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,)
    # Get the Q-values (b,1) from the target network for the next state
    next_q = agent.target_q(next_state_b)
    max_next_q = tf.reduce_max(next_q, axis=-1)

    # Calculate target reward using the Bellman equation (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, action_range)

    # Make ground true labels for training (b,a)
    label_q = curr_q * (1 - action_v) + target_reward[:, None] * action_v
    ###  END CODE HERE  ###

    return label_q
```

```
# Set random seed for reproducibility
tf.random.set_seed(2)

# Define the expected result for the training labels
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 ]])
# Call the function to get training labels
res = test_training()
print('Training lable test passed.') if np.allclose(res,ans) else print('Training lable test failed.')
```

```
Training lable test passed.
```

훈련 레이블에 대한 예상 결과를 정의하고 test_training함수를 호출해 훈련 레이블을 계산한다. 이후 test_training 함수에서 계삳뇐 결과 res를 예상 결과인 ans와 비교한 코드이다. 이때 허용 범위 내에서 충분히 가까우면 훈련 라벨 테스트 통과를 출력하며 결과도 통과하였음을 확인할 수 있다.

```
# Trains the DQN agent by updating its policy network using the given batch of experiences
def dqn_train(agent, batch, config):
    state_b, action_b, next_state_b, reward_b, not_done_b = batch # Unpack the batch
    gamma = config.gamma # Discount factor from the config

    ### START CODE HERE ###

    # get action probability with current (think of WHY!!) policy (b,a)
    # Get the current Q-values from the policy network (b, a) where a is the number of actions
    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,)
    # Get the Q-values from the target network for the next states (b, a)
    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, action_range)

    # Make ground true labels for training (b,a)
    label_q = curr_q * (1 - action_v) + target_reward[:, None] * action_v

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

    ###  END CODE HERE  ###

    # Get the loss from the training logs
    loss = logs.history['loss'][-1]
    return loss
```

Test_trianing 코드와 비슷하지만 다른 점은 위 코드는 DQN 에이전트를 훈련시키는 함수이며 배치를 사용하여 state, action, next_state, reward, not_done을 가져오며, fit 함수를 사용하여 훈련한다는 점에서 차이점이 있다.

```
# To evaluate the performance of the agent by averaging rewards over multiple episodes
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)        # Reset environment and get initial state
        done = False
        episode_reward = 0.0
        while not done:

            ### START CODE HERE ###

            action = agent.policy(state, epsilon=0.0, exploring=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           # Update state
            episode_reward += reward     # Accumulate reward for the episode
        total_reward += episode_reward   # Add the episode reward to total reward

    # Calculate the average reward across all episodes
    average_reward = total_reward / episodes_to_play

    return average_reward
```

위 코드는 여러 에피소드에 대한 총 보상을 평균하여 강화 학습 에이전트의 성능을 평가하는데 사용하는 함수로, 에피소드 루프 부분을 살펴보면 각 에피소드 내에서 에이전트는 정책에 따라 조치를 취한다. 에이전트는 정책을 사용해 작업을 선택하고, 환경에서 작업을 수행하고 next_sate, reward 및 done 플래그를 받는다. 이후 다음 시간 단계에 대해 상태가 업데이트 되며 현재 에피소드에 대한 보상이 누적된다. 각 에피소드가 끝나면 에피소드 전체의 총 보상이 업데이트되고 모든 에피소드가 끝나면 total_reward를 episodes_to_play로 나누어 평균 보상을 계산 후 이를 반환한다.

Define Epsilon Function

This is an example of exponential decay epsilon function. One of easist epsilon decay functions is simply to multiply 0.9. You can define your own epsilon function.

```
# 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                # Episode counter

    def get_epsilon(self):
        # Calculate epsilon based on exponential decay
        eps = (self.explore_stop
            + (self.explore_start - self.explore_stop) * tf.math.exp(-self.decay_rate * self.episode_cnt))
        self.episode_cnt += 1
        return eps
```

위 코드는 epsilon-greedy exploration strategy를 구현한 것으로 get_epsilon함수를 보면 exponential decay을 사용하여 epsilon을 계산한다. 또한 시간이 지남에 따라 탐색을 감소시켜 균형을 조정한다.

Define and Initialize Hyperparameters

```
# Configuration class for storing hyperparameters
class configuration:
    def __init__(self):
        self.gamma = 0.99   # Discount factor for future rewards
        self.lr = 2e-4      # Learning rate for the optimizer

# Create a configuration object
config = configuration()
```

```
# Total steps in the simulation
max_steps = max_episodes * max_ep_steps
batch_size = b_size
memory_size = h_size

# Initialize the agent and replay memory
agent = Agent_Net()
memD = ReplayMemory(memory_size)
# Initialize memory with experiences from the environment
init_memory(memD, env, agent, memory_size)
# Initialize epsilon decay function and optimizer
epsF = Epsilon(max_episodes, 10.0)
opt = tf.optimizers.Adam(learning_rate=config.lr, clipvalue=2.0)

# Compile the policy network with optimizer and loss function
agent.policy_q.compile(optimizer=opt, loss='mse', jit_compile=False)
```

첫 번째 코드는 훈련 과정에 사용되는 매개변수를 저장하는데 사용되는 코드로 gamma 및 learning rate를 설정한다. 두 번째 코드는 시뮬레이션 매개변수를 설정하고 에이전트 및 replaymemory, init_memory, epsilon decay, optimizer를 초기화한다. 이후 policy network를 컴파일 한다.

Define Main Training Loop

Exercise: Complete Main Training Loop

```
]  logs = keras.callbacks.History()     # Initialize a history object to store logs during training
   logs.history.update({'pi_loss':[]})  # Initialize empty lists for policy loss
   logs.history.update({'ereward':[]})  # Initialize empty lists for episode reward
   logs.history.update({'e-steps':[]})  # Initialize empty lists for episode steps
   logs.history.update({'vreward':[]})  # Initialize empty lists for validation reward

   # Variables for simulation
   num_episodes = 0
   val_episodes = 2  # Exit condition

   # Variables for episode logging
   pi_loss = 0.0
   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}')
```

우선 훈련 로그를 저장할 객체, 에피소드 및 평가 관련 변수, epsilon 값을 초기화한 코드이다.

아래의 코드는 훈련 루프로서 각 시뮬레이션 스탭에서는 현재 상태를 얻고 에이전트가 탐험적 행동을 통해 액션을 선택한다.(agent.policy) 그 액션을 통해 환경에서 다음 상태, 보상, 완료 여부를 받고(env_step), 새로운 경험을 리플레이 메모리에 저장한다.(memD.put_experience) 이후 리플레이 메모리에서 배치를 가져와 DQN 훈련을 수행한다.(dqn_train) 에피소드 종료 후 타겟 네트워크, epsilon 값을 업데이트 한 후 각 에피소드에 대한 훈련 결과를 로그에 저장한다. 반복하다가 goal_score를 초과하면 훈련을 종료하며 종료 조건은 목표 점수에 도달하거나 최대 에피소드에 도달했을 때이다.

```
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=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

    # 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

        # Update logs with the results of the episode
        logs.history['pi_loss'].extend([pi_loss])
        logs.history['ereward'].extend([epis_reward])
        logs.history['e-steps'].extend([epis_steps])
        logs.history['vreward'].extend([eval_reward])
```

```
        # Reset episode tracking variables
        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 the goal score is reached
    if eval_reward > goal_score:
        eval_reward = evaluate_policy(env, agent, val_episodes) # Evaluate policy multiple times
    if eval_reward > goal_score: break
    # Stop simulation if the maximum number of episodes is reached
    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)
```

- Cart Pole

```
  3%|▌         | 6619/200000 [10:28<4:36:38, 11.65it/s, episode=209, loss=4.03, reward=500, steps=0]
episodes:  209, loss:11.61478, val_reward 500.00
total steps: 6620
```

우선 cartpole의 경우 훈련이 전체 200,000 스탭 중 3% 완료 되었으며 (6619/200000)는 현재까지 훈련한 스탭 수를 의미한다. 초당 11.65개의 스탭이 처리되었고 현재 에피소드

는 209번째, 현재 에피소드에서 에이전트가 받은 총 보상은 500임을 확인할 수 있다.

- Lunar Lander

```
9%|▉         |
34055/400000 [53:42<8:11:15, 12.42it/s, episode=71, loss=29.7, reward=224, steps=0]
episodes:   71, loss:1.46242, val_reward 210.16
total steps: 34056
```

Lunalander의 경우 훈련이 전체 400,000 스탭 중 9% 완료 되었으며 (34055/400000)는 현재까지 훈련한 스텝 수를 의미한다. 초당 12.42개의 스탭이 처리되었고 현재 에피소드는 71번째, 현재 에피소드에서 에이전트가 받은 총 보상은 224임을 확인할 수 있다.

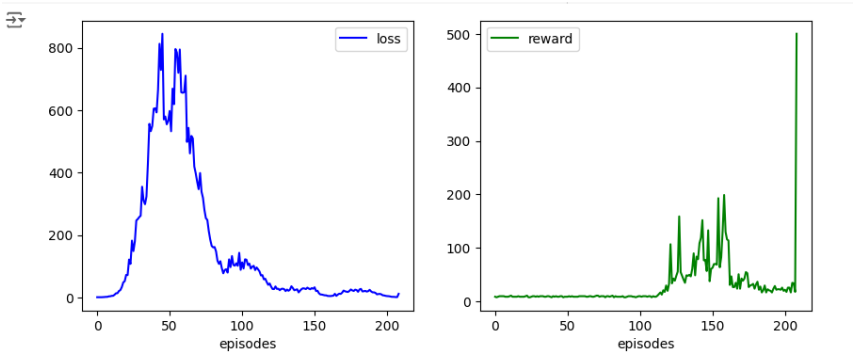
- Cart Pole

Plot Training Histories

```python
# 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

# Define the labels
log_labels    = ['pi_loss', 'vreward']
label_strings = ['loss', 'reward']
label_colors  = ['b-', 'g-']
plot_graphs(logs.history, log_labels, label_strings, label_colors)
```

위 코드는 훈련 과정에서 기록된 pi_loss, 손실 값과 vreward, 보상 값을 시각화 하는 함수이다.



위 그래프는 cart pole의 시각화 결과이다.

Evaluate the Agent

Since a single evaluation try often takes some time, evaluate the agent here to show the progress bar

```
[ ]  evaluate_episodes = 20 # Set the number of episodes for evaluation
     sum_episode_rewards = 0.0 # Initialize the sum of episode rewards
     pbar = tqdm(range(evaluate_episodes))

     for i in pbar:
         # Evaluate the policy for each episode
         sum_episode_rewards += evaluate_policy(env, agent, 1)

     env.close()

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

```
100% 20/20 [00:38<00:00,  1.89s/it]
Evaluation Result: 500.0
```

위 코드는 주어진 에이전트의 성능을 여러 에피소드에 걸쳐 평가하고 평균 보상을 출력하는 코드로 결과를 살펴보면 20번의 평가가 완료되었고 평가에 걸린 시간은 약 0.38초이다. Evaluation Result를 보면 에이전트가 20번의 평가 에피소드에서 얻은 평균 보상은 500이다. Reward 종료 조건을 만족함을 알 수 있다.

```
env = create_env() # Create the environment
# Wrap the environment to record videos
env = wrappers.RecordVideo(env, video_folder='./gym-results/', name_prefix=res_prefix)
# Evaluate the agent's policy
eval_reward = evaluate_policy(env, agent, 1)

print('Sample Total Reward:', eval_reward)

env.close()
```

```
Sample Total Reward: 500.0
```

위 코드는 강화학습 환경을 생성하고 이를 비디오로 저장한 후 평가 결과로 얻은 reward를 출력하는 코드이다.

결과를 살펴보면 에이전트가 환경에서 한 번의 에피소드를 수행한 후 얻은 총 보상은 500이다.

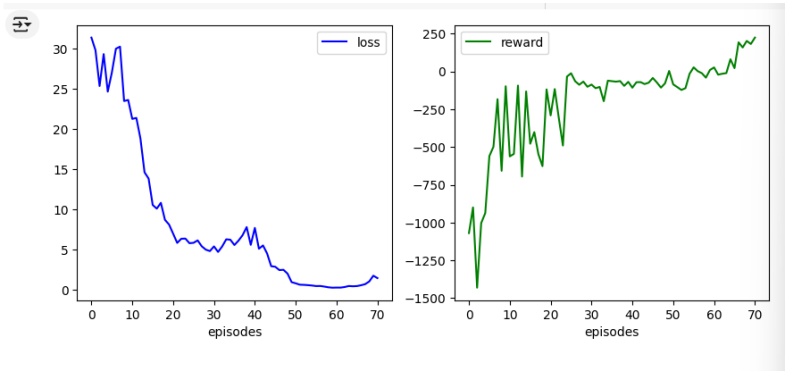
```
from IPython.display import HTML
from base64 import b64encode

def show_video(video_path, video_width = 320):
  # Read the video file in binary mode
  video_file = open(video_path, "r+b").read()
  # Encode the video in base64 format to embed in HTML
  video_url = f"data:video/mp4;base64,{b64encode(video_file).decode()}"
  # Return HTML video player with the embedded base64 video
  return HTML(f"""<video width={video_width} controls><source src="{video_url}"></video>""")

# Display the video using the function
show_video('./gym-results/' + res_prefix + '-episode-0.mp4')
```

위 코드는 위에서 저장한 비디오를 Jupyter 노트북에서 재생하는 코드이다. 코드 아래의 그림은 cart pole의 비디오가 출력된 결과이다.

- Lunar Lander



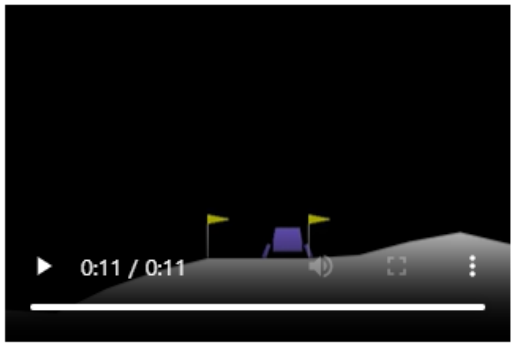
위 그래프는 lunar lender의 시각화 결과이다.

```
100% 20/20 [01:02<00:00,  3.34s/it]
Evaluation Result: 146.06925780077114
```

결과를 살펴보면 20번의 평가가 완료되었고 평가에 걸린 시간은 약 3.34초 이다. Evaluation Result를 보면 에이전트가 20번의 평가 에피소드에서 얻은 평균 보상은 약 146.07이다. 이는 Reward 종료 조건을 만족하지 못함을 알 수 있다.

```
/usr/local/lib/python3.10/dist-packages/gymnasium/wrappers/rendering.py:283: UserWarning: WARN: Overwriting existing vi
  logger.warn(
Sample Total Reward: 207.9398907515973
```

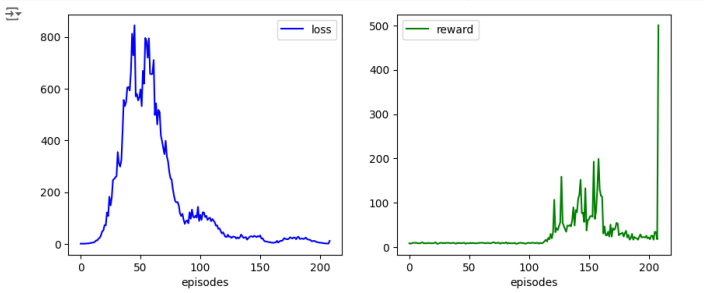
결과를 살펴보면 에이전트가 환경에서 한 번의 에피소드를 수행한 후 얻은 총 보상은 207,94이다. 또한 나타난 경고 메세지는 기존에 저장된 비디오 파일이 있을 경우 덮어 쓸 것이라고 경고 메시지를 주는 것으로 무시해도 괜찮다.

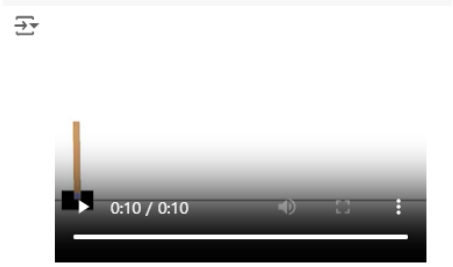


그림은 lunar lender의 비디오가 출력된 결과이다.

Simulation results & Discussion

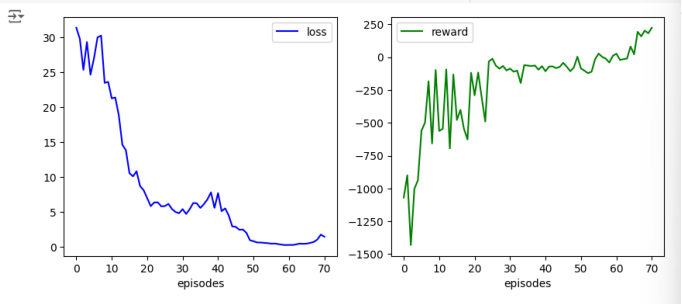
- Cart Pole

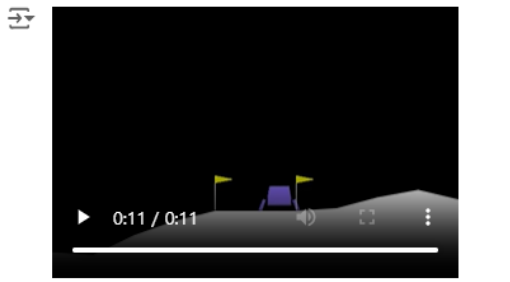




왼쪽은 cart pole의 loss, reward 그래프이고 오른쪽은 mp4파일이다. 우선 왼쪽부터 살펴보면, 파란색 곡선은 에피소드에 따라 훈련 손실이 어떻게 변하는지 보여주며 높게 시작되었지만 50 이후에는 크게 감소하는 것을 확인할 수 있다. 또한 100 에피소드 이후에는 손실이 낮은 값으로 안정화되어 모델이 효과적으로 학습되었다는 것을 알 수 있다. 이어서 녹색 곡선은 훈련 중 수집된 보상을 나타내며 초기 에피소드 동안에는 낮게 유지되었다가 100회 이후에서는 보상이 급격하게 증가하며 종료 조건인 500 부근에서 안정화가 된 것을 확인할 수 있다. 이는 에이전트가 지속적으로 높은 보상을 달성하였기에 훈련이 성공적으로 되었음을 확인했다. 오른쪽 영상은 에이전트의 성과를 시각적으로 보여주는데, 카트에서 기둥의 균형을 좌우로 움직이다가 성공적으로 맞추는 것을 확인하였다.

- Lunar Lander





왼쪽은 lunar lender의 loss, reward 그래프이고 오른쪽은 mp4파일이다. 우선 왼쪽부터 살펴보면, 파란색 곡선, 즉 손실은 높게 시작해서 처음 40개의 에피소드 내에서 크게 감소한다. 60회까지 손실이 0에 가깝게 안정화되어 에이전트가 수렴되었다는 것을 알 수 있다. 이어서 녹색 곡선, 즉 누적 보상 부분은, 처음에는 보상의 성과가 좋지 않은 것을 확인할 수 있었고, 시간이 지남에 따라 보상이 꾸준히 활성화되어 종료조건인 약 200에 도달하는 것을 확인하여 성공적인 학습이 되었다고 생각하였다. 오른쪽 영상은 뒤집히거나 충돌하지 않았고 가운데는 아니지만 깃발 사이에 성공적으로 착륙하는 것을 확인하였다.