인공지능프로그래밍

Lab 07: Tabular Q Learning

학 번 : 2021202058

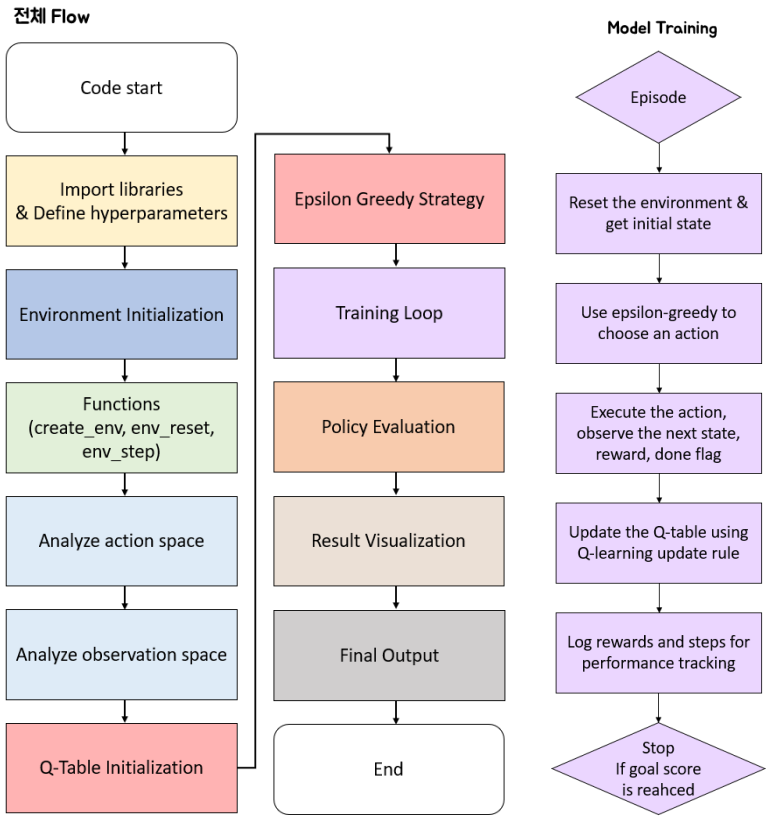
성 명 : 송채영

날 짜 : 2024.11.25

Lab Objective

이번 과제에서는 Tabular Q-learning을 직접 구현해본다. 구현한 모델을 바탕으로 gymnasium library의 Frozen lake, Taxi, Blackjack, 3개의 environment에서 적용해보며 결과를 확인한다. 또한 off-policy의 TD control과 policy에 대해 이해하며, policy 에서는 Behavior policy와 Target policy에 대해 알아볼 수 있다.

Program flow



우선 왼쪽 사진은 전체적인 코드의 흐름이다. 노란색 박스는 setup 및 configuration에 해당한다. Gymnasium, numpy, tqdm등의 필요한 라이브러리를 가져오며 Frozen, lake, Taxi, Blackjack에 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 재설정하고 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 space

를 분석해서 상태 공간의 크기와 구조를 결정한다. 핑크색 박스는 Q-learning setup 부분에 해당한다. 우선 Q-table을 초기화한다. [num_states, num_actions]를 사용하여 0으로 구성된 Q-table을 생성한다. 이후 Epsilon Greedy 부분인데 exponential decay를 사용해서 클래스를 정의하였다. 보라색 박스는 Training Loop에 해당하며 오른쪽 사진에 과정을 자세하게 나타내었다. 각 에피소드에 대해 환경을 재설정하고 초기 상태를 가져온다. 이후 epsilon-greedy를 사용해 action을 선택한다. 이때 Q 값이 가장 높은 action을 선택하고 임의의 action을 선택하도록 한다. Action을 실행한 후 next_state, reward, done flag를 확인한다. Q-learning update rule을 바탕으로 Q-table을 업데이트한다. 이때 수식은 $Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma a' \max Q(s',a') - Q(s,a)]$ 다음과 같다. 이후 performance tracking을 위한 rewards와 steps을 기록한다. 에피소드에 대해 반복하다가 goal score에 도달하면 훈련을 중지하도록 한다. 이어서 주황색 박스는 Policy Evaluation에 해당한다. 여러 에피소드에 걸쳐 학습된 policy를 evaluate_policy 함수를 사용해 최적의 policy, argmax(Q)를 사용해 에피소드를 실행한 후 평균 reward와 step을 계산한다. 갈색 박스는 Result Visualization에 해당한다. 이부분에서는 환경에 따라 비디오로 녹화 및 이미지로 렌더링을 한다. 마지막으로 회색 박스는 Final output 부분이다. 이부분에서는 최종 평가 결과(평균 보상)을 print 한 후 선택된 환경에 따라 녹화된 영상이나 렌더링 된 이미지를 보여준다.

Tabular Q-learning

Tabular Q-learning은 강화학습 중 하나의 알고리즘으로 policy를 명시적으로 저장하지 않고 Q-value를 학습하여 최적의 action을 찾는 off-policy 학습방법이다. 이를 통해 agent는 어떤 상태에서 가장 높은 reward를 얻을 수 있는 action을 선택할 수 있다.

Q-learning의 기본 개념을 살펴보면, Q-value는 특정 상태 s에서 특정 행동 a를 했을 때 얻을 것으로 예상되는 누적 보상에 해당하며 Bellman Equation을 기반으로 Q-value를 업데이트한다. 이어서 Tabular Q-learning은 모든 state-action 쌍의 Q-value를 테이블 배열에 저장한다. 따라서 Q-table의 크기는 state의 수 x action의 수가 된다. Exploration과 Exploitation를 비교하면 학습 초반에는 Exploration 비율을 높이고 학습 후반에는 Exploitation 비율을 높이기 위해 epsilon-greedy를 사용한다. 또한 off-policy를 사용하는데 action을 선택할 때는 Exploration를 포함한 Behavior policy를 따르지만 Q-value를 업데이트 할 때는 Exploitation만 고려하여 Target policy를 학습한다. 또한 state space가 작고 모든 state-action 쌍을 충분히 탐험해야 제대로 학습이 가능하기 때문에 모든 state를 방문할 필요가 있다.

Tabular Q-learning은 단순하지만 이번 과제와 같이 작은 환경에서는 강력한 성능을 낼 수 있고 deep Q-learning인 DQN과 같은 알고리즘의 기본 원리로 작용한다.

Result

```
[1]  # 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 gymnasium
         from tqdm.notebook import tqdm
     else:
         from tqdm import tqdm
```

```
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.
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 7.8 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
```

우선 현재 코드가 어떤 환경에서 실행 중인지를 확인한다. get_ipython의 결과가 goolge.colab 문자열을 포함하는지 체크하는 방식을 사용한다. Google Colab환경에서 실행될 때는 Jupyter notebook과 같은 환경에 맞게 조정된 tqdm 라이브러리를 가져오며 로컬 환경에서 실행될 때는 일반 tqdm 라이브러리를 가져온다. Google Colab환경에서 실행될 때 gymnasium을 설치해주는데, gymnasium은 강화학습을 위한 오픈소스 라이브러리로, OpenAI Gym의 기본 기능을 확장하고 개선해서 더 다양한 환경과 새로운 기능을 제공한다고 한다.

```
[2]  # Importing libraries
     import numpy as np
     import matplotlib.pyplot as plt

     import gymnasium as gym
     from gymnasium import wrappers
```

```
[4]  # Check the installed version
     gym.__version__
```

```
'1.0.0'
```

이어서 Numpy, Matplotlib.pyplot, gymnasium, 필요한 라이브러리를 가져온 후 gymnasium 의 설치된 버전을 확인해보았다. 간단하게 사용한 라이브러리에 대해 설명해보자면 다음과 같다.

Numpy – 배열, 행렬 및 수학 연산 작업에 사용되는 수치 계산용 라이브러리

Matplotlib.pyplot – 선 플롯, 막대 차트, 히스토그램과 같은 시각화를 생성하기 위한 플로팅 라이브러리

Gynmnasium -강화 학습 환경을 생성하고 상호작용하기 위한 라이브러리

```
# select evironment
# Text Game: 0 for FrozenLake 4x4, 1 for Taxi, 2 for Blackjack

SELECT_ENV = 1 # Choose the environment
```

```
# Frozen Lake configuration settings
if SELECT_ENV == 0:
    env_name, res_prefix = 'FrozenLake-v1', 'lak4'           # Environment name and result file prefix
    max_episodes, max_ep_steps, goal_score = 3000, 500, 0.8  # Number of learning episodes, maximum steps per episode and goal score
    val_ep_num, lr_rate, discount_rate = 20, 0.8, 0.95       # Number of evaluation episodes, learning rate, discount rate
# Taxi configuration settings
elif SELECT_ENV == 1:
    env_name, res_prefix = 'Taxi-v3', 'taxi'
    max_episodes, max_ep_steps, goal_score = 1000, 500, 8.0
    val_ep_num, lr_rate, discount_rate = 20, 0.8, 0.95
# Blackjack configuration settings
elif SELECT_ENV == 2:
    env_name, res_prefix = 'Blackjack-v1', 'blkj'
    max_episodes, max_ep_steps, goal_score = 5000, 10, -0.05
    val_ep_num, lr_rate, discount_rate = 1000, 0.1, 0.95
# Error in invalid selection
else: assert False, 'environment selection error'

# Create a Gym environment to the selected environment
def create_env():
    if   SELECT_ENV == 0:   # Create Forzen Lake Environment
        env = gym.make(env_name, desc=None, map_name='4x4', is_slippery=True, render_mode='rgb_array')
    elif   SELECT_ENV == 1: # Create Taxi Environment
        env = gym.make(env_name, render_mode='rgb_array')
    elif   SELECT_ENV == 2: # Create Blackjack Environment
        env = gym.make(env_name, natural=False, sab=False, render_mode='rgb_array')
    else: pass              # Do not process incorrect choices
    return env
```

select_env 변수는 사용자가 선택한 환경을 나타내며 0일 때 FrozenLake, 1일 때 Taxi, 2일 때 Blackjack에 해당한다.

각 환경에 맞는 설정을 조건문에 따라 다르게 정의한 코드로 env_name은 환경의 이름, res_prefix는 환경에 맞는 결과 파일의 접두어, ex)lak4, taxi, blkj, max_episodes는 학습에 사용할 최대 에피소드 수, max_ep_steps는 한 에피소드에서의 최대 step 수, goal_score는 목표 점수, val_ep_num은 평가 에피소드 수, lr_rate는 학습률, discount_rate는 discount factor에 해당한다. 잘못된 환경 선택의 경우 오류를 발생시킨다. 이어서 환경을 생성하는 함수는 사용자가 선택한 환경을 gym 라이브러리를 사용해 생성한 후 그것을 반환한다. FrozenLake의 경우 4x4 크기의 맵을 선택한 후 is_slippery는 미끄러운 얼음 환경을 활성화하는 것을 의미한다. Render_mode는 화면을 렌더링 할 때를 말하며 RGB 배열 형식으로 반환하였다. Taxi의 경우 Render_mode를 RGB로 설정하였고, Blackjack의 경우 natural=False로 하여 처음에 에이스와 10으로 시작하는 경우 추가 보상을 제공하지 않음을 뜻한다. 또한 Render_mode를 RGB로 설정하였다.

```
# Environmental Initialization Function
def env_reset(env):
    observation = env.reset() # Initialize the environment and bring in initial observations
    obs = observation[0] if type(observation)==tuple else observation # Use the first value if the observation is a tuple
    if SELECT_ENV == 2: # In a Blackjack environment
        state = obs[0] + 32 * obs[2] + 64 * obs[1] # State calculation
    else: state = obs  # Use as is in other environments
    return state

# One-step execution function in the environment
def env_step(env, action):
    observation = env.step(action) # Use a given action to take a step forward in environment
    if SELECT_ENV == 2:
        state = observation[0][0] + 32 * observation[0][2] + 64 * observation[0][1]
    else: state = observation[0]
    reward = observation[1] # Get rewards
    done = observation[2] or observation[3] if len(observation)>4 else observation[2] # Determininig whether to done
    return state, reward, done
```

```
env = create_env() # Call function that creates environment
state = env_reset(env) # Initialize the environment and get the initial state
state, reward, done = env_step(env, env.action_space.sample()) # Select a random action and apply it to the environment to receive a new condition, reward, done
```

우선 env_reset 함수는 환경을 초기화하는 함수로 env.reset함수를 호출해 환경을 초기화하고 observation을 받는다. Observation이 튜플인 경우 observation[0]을 사용하며 아닌 경우 observation 그대로를 사용한다. Blackjack인 경우는 observation이 Player current sum, Dealer showing card value, Usable Ace 이렇게 3개의 값으로 구성되어 있다. 각 요소에 가중치를 주어 state를 계산한다. 다른 환경에서는 그대로 사용하며 최종적으로 계산된 state를 반환한다. 이어서 env_step 함수는 env.step(actio)을 호출해서 주어진 action을 환경에 적용하고 한 단계를 진행한다. Blackjack의 경우 observation이 위 함수와 같이 동작하며 다른 환경의 경우도 마찬가지이다. 이후 reward를 얻은 후 done을 통해 종료여부를 확인한다. Blackjack 환경에서는 observation[3]을 사용할 수도 있으며, 환경이 반환하는 값의 길이가 4보다 길다면 역시 observation[3]을 사용한다. 이후 state, reward, done을 반환한다.

이어서 환경을 생성하고 실행하는 예시의 코드이다. create_env 함수를 호출해 env를 생성한 후 환경을 초기화하고 무작위로 action을 선택한 후 env_step 함수를 호출해 그 환경에서 한 단계를 진행한다. 이때 새로운 state, reward, done을 받는다.

Check and Show Environment Variables

```
[8]  # Check the action space type and set corresponding parameters
     action_shape = env.action_space.shape
     action_space_type = type(env.action_space)

     # For Discrete action space
     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 action space
     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
     # If neither is the case
     else: assert False, 'other action space type are not supported'

     # Check the observation space type and set corresponding parameters
     observation_space_type = type(env.observation_space)
     observation_shape = env.observation_space.shape
```

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.

```
# For Discrete observation space
if observation_space_type==gym.spaces.discrete.Discrete:
    observation_shape = (1,)
    num_states = env.observation_space.n
# For Box (continuous) observation space
elif observation_space_type==gym.spaces.box.Box:
    num_states = observation_shape[0]
# For Tuple observation space (e.g., multiple spaces combined)
elif observation_space_type==gym.spaces.tuple.Tuple:
    observation_shape = tuple([x.n for x in env.observation_space])
    num_states = np.prod(observation_shape)
# Unsupported
else: print('observation space type error')

# Define state shape and batch shape depending on environment
if SELECT_ENV == 2:
    state_shape = (1,)
    state_batch_shape = (None,1)
else:
    state_shape = observation_shape
    state_batch_shape = tuple([None]+[x for x in observation_shape])

# Set value shape
value_shape = (1,)
num_values = 1
```

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 정보를 출력하는 코드이다. Action space의 유형, 모양, 차원 수, 가능한 동작 범위, continuous 공간인 경우 동작 값의 상한 하한, 배치 모양등을 출력하고, observation space의 유형, 모양, 상태 공간의 크기, 상태의 모양, 상태 공간의 배치 모양을 출력한다. 이후 value function 정보를 출력한 것이다. 출력 결과를 살펴보면 동작 공간이 discrete임을 알 수 있고 이는 동작이 고정된 정수 값으로 제한됨을 의미한다. 동작 공간의 모양은 크기 1의 튜플이며 동작 차원수는 1이다. 가능한 동작의 개수는 6개, 배치 크기는 미니배치 단위로 none x 6의 형태이다.

관찰 공간의 경우 discrete임을 알 수 있고 이 역시 가능한 상태가 고정된 개수의 정수 값으로 제한됨을 의미한다. 모양은 크기 1의 튜플이다. 상태의 개수는 500개 상태의 모양은 크기 1의 튜플, 배치 크기는 none x 1의 형태이다. Value 의 출력 모양은 크기 1의 튜플, 차원 수는 1이다.

```
# Prints action space, shape, dimensions, range, max/min values, batch shape,
# observation space, shape, state size, state shape, batch shape, and value dimensions
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  6
Action batch shape  (None, 6)
Observation space  <class 'gymnasium.spaces.discrete.Discrete'>
Observation shape  (1,)
Size of State Space  500
State shape  (1,)
State batch shape  (None, 1)
Vallue shape  (1,)
Value dimensions  1
```

```
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.discrete.Discrete'>
Observation shape  (1,)
Size of State Space  16
State shape  (1,)
State batch shape  (None, 1)
Vallue shape  (1,)
Value dimensions  1
```

```
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.tuple.Tuple'>
Observation shape  (32, 11, 2)
Size of State Space  704
State shape  (1,)
State batch shape  (None, 1)
Vallue shape  (1,)
Value dimensions  1
```

첫 번째 사진은 taxi의 경우이고 아래는 차례대로 frozen lake, blackjack의 경우이다.

Define and Initialize The Agent

Exercise: Define Q-table (Numpy)

```
[10] ### START CODE HERE ###

     Q_table = np.zeros((num_states, num_actions)) # Q-table initialized with zeros based on the number of states and actions

     ###  END CODE HERE  ###

     print(Q_table.shape)

     (500, 6)
```

Define and Initialize Hyperparameters

```
[11] total_episodes = max_episodes   # Total episodes
     max_steps = max_ep_steps        # Max steps per episode
     learning_rate = lr_rate         # Learning rate
     gamma = discount_rate           # Discounting rate
     val_episodes = val_ep_num       # Number of validation episodes
```

우선 Q-table을 초기화한다. Np.zeros 함수를 사용해 0으로 테이블을 초기화한다. 이때 크기는 num_states x num_actions이다. 이를 출력해보면 Q-table의 크기는(500,6)인 것을 알 수 있다. 다음 코드는 Hyperparameters를 설정하는 코드이다. 총 에피소드 수는 에이전트가 환경을 반복적으로 탐색하며 학습하는 횟수를 의미하며 max_episodes 변수에 의해 정의되었다. 이어서 각 에피소드에서 수행할 최대 단계 수는 하나의 에피소드 내에서 에이전트가 행동할 수 있는 최대 횟수를 의미하며 max_ep_steps 변수에 의해 정의된다. Learning_rate는 학습률로 새롭게 얻은 정보가 Q-table에 얼마나 빠르게 반영되는지를 제어하며 lr_rate에 의해 정의된다. Gamma는 미래에 보상의 중요도를 조정하며 감마가 0일 경우 현재 보상만 고려하며 1일 경우 장기적인 보상만 고려함을 의밎나다. 이는 discount_rate에 의해 정의된다. 마지막으로 검증 에피소드 수는 학습 후 에이전트의 성능을 평가하는데 사용되며 val_ep_num 변수에 의해 정의된다. (환경이 taxi일 때의 결과)

Exercise: Define Q-table (Numpy)

```
[10] ### START CODE HERE ###

     Q_table = np.zeros((num_states, num_actions)) # Q-table initialized with zeros based on the number of states and actions

     ###  END CODE HERE  ###

     print(Q_table.shape)

     (16, 4)
```

Exercise: Define Q-table (Numpy)

```
### START CODE HERE ###

Q_table = np.zeros((num_states, num_actions)) # Q-table initialized with zeros based on the number of states and actions

###  END CODE HERE  ###

print(Q_table.shape)
```

```
(704, 2)
```

Frozen lake와 blackjack의 경우의 shape이다.

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.

```
[12] # Exploration parameters for epsilon greedy strategy
     class Epsilon:
         def __init__(self, max_steps):
             self.explore_start = 1.0            # Initial exploration probability at start
             self.explore_stop = 0.01            # Minimum exploration probability
             self.decay_rate = 20.0/max_steps    # Exponential decay rate for reducing exploration prob (4.6 ~ max_step = 0.01)
             self.steps = 0                      # Counter to track the number of steps taken

         def get_epsilon(self):
             # Calculate the current exploration probability using exponential decay formula
             eps = (self.explore_stop
                 + (self.explore_start - self.explore_stop) * np.exp(-self.decay_rate * self.steps))
             self.steps += 1                     # Increment the step counter
             return eps                          # Return the current exploration probability
```

Learning Procedures

Q-learning is an off-policy TD control algorithm.

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha(r_t + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t))$$

The target policy π is greedy w.r.t. $Q(s, a)$

$$\pi(s_t) = \arg\max_{a'} Q(s_t, a')$$

Epsilon greddy exploration에서 사용되는 exploration 확률을 정의하고 update하는 클래스를 정의한 코드이다. 우선 클래스를 초기화하는 부분에서는 초기 탐험 확률, 최소 탐험 확률, 탐험 확률 감소 속도, 현재까지의 step 수를 정의하였다. 이후 탐험 확률을 계산하는 함수 부분에서는 $\epsilon = \epsilon_{stop} + (\epsilon_{start} - \epsilon_{stop}) \cdot e^{-decay\ rate \cdot steps}$ 수식을 이용한다. 초기에는 ϵ이 높아 임의의 행동을 만힝 수행하지만 학습이 진행되며 ϵ이 감소해 더 많이 학습된 정책에 따라 행동하게 된다. 탐험 확률을 계산한 후 step을 1 증가시켜 학습 진행 상황을 추적한다. 그 아래의 수식은 Q-learning 업데이트 수식과 정책 π를 정의한 것이다. 이부분에 대해서는 위에서 Tabular Q-learning을 설명하며 했으므로 생략한다.

Exercise: Define Training Loop and Evaluation Loop

```
def evaluate_policy(env, qtable, num_average, images=None):

        total_reward = 0.0              # Accumulates total reward over all evaluation episodes
        total_steps = 0                 # Accumulates total steps taken across episodes
        episodes_to_play = num_average  # Number of episodes to play for evaluation
        for _ in range(episodes_to_play):  # Play n episode and calculate average performance
            state = env_reset(env)         # Reset the environment to its initial state
            if images!=None: images.append(env.render()) # If images is provided, capture the environment's initial render
            done = False  # Track whether the episode is finished
            episode_reward = 0.0  # Accumulates reward for the current episode
            while not done:  # Continue until the episode is finished

                ### START CODE HERE ###

                action = np.argmax(qtable[state])                   # Select the best action from the Q-table for the current state
                next_state, reward, done = env_step(env, action)    # Take action and observe the next state, reward, and done flag

                ###  END CODE HERE  ###

                if images!=None: images.append(env.render()) # If images is provided, capture the environment's render after each step
                state = next_state                          # Update current state to the next state
                episode_reward += reward                    # Accumulate the reward for this episode
                total_steps += 1                            # Increment the total step count
            total_reward += episode_reward                  # Add the episode's reward to the total reward
        average_reward = total_reward / episodes_to_play    # Take the average reward of evaluations
        average_steps = total_steps / episodes_to_play      # Take the average steps of evaluations

        return average_reward, average_steps, images        # Return the averages and images
```

주어진 정책 Q-table을 평가하는 함수로 주어진 환경에서 num average개의 평가 에피소드를 실행하여 평균 보상과 평균 스탭수를 계산한다. Images 매개변수가 제공되면 각 에피소드 동안 환경의 상태를 렌더링하여 시각적인 출력을 저장하도록 한다. 구현한 코드 부분을 살펴보면 Q-table을 통해 행동을 선택하도록 하였다. 현재 상태에서 Q-table을 참조하여 가장 높은 값, 즉 가장 좋은 행동을 선택한다. 이는 greedy policy를 따르는 방식으로 가장 높은 Q-value를 가진 행동을 선택하게 되는 것이다. 이어서 env_step 함수를 통해 선택된 행동을 환경에 적용하고 next_state, reward, done을 반환한다.

```
history = {'rewards' : []}                          # Dictionary to logs of rewards
epsF = Epsilon(total_episodes)                      # Generate epsilon object for epsilon-greedy strategy

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

for episodes in pbar: # Iterate through all episodes

    # initialize variables for a new episode
    epis_rewards = 0                                # Total rewards accumulated in episode
    epis_steps = 0                                  # Total steps token in episode
    done = False                                    # Episode end flag

    state = env_reset(env)                          # Reset the environment to the initial state

    epsilon = epsF.get_epsilon()                    # Get the current epsilon value for exploration

    while not done:                                 # Simulate until the terminal state of the episode

        ### START CODE HERE ###

        # Get a random number for exploration decision
        random_number = np.random.rand()
        # If random number > greater than epsilon --> exploitation
        if random_number > epsilon:
            #  Find an index of the biggest Q value for this state
            action = np.argmax(Q_table[state])
         # Else doing a random choice --> exploration
        else:
            #  Get a random integer in [0,action_range)
            action = np.random.randint(0, action_range)

        # Take the action to environment and observe the next state, reward, and done flag
        next_state, reward, done = env_step(env, action)
        # Update Q-table using Q-learning update rule
        Q_table[state, action] += learning_rate * (reward + gamma * np.max(Q_table[next_state] - Q_table[state, action]))


        ###  END CODE HERE  ###
```

```
            epis_rewards += reward                    # Accumulate rewards to calculate the episode reward
            state = next_state                        # Set state for the next state

            if epis_steps > max_steps: break          # Terminate if the episode exceeds the maximum steps
            else: epis_steps += 1                     # Increment step count

        # Evaluate the policy after each episode
        eval_reward, eval_steps, _ = evaluate_policy(env, Q_table, val_episodes) # Evaluate the policy

        history['rewards'].append(eval_reward) # Log the evaluation reward

        pbar.set_postfix({'reward':eval_reward, 'steps':eval_steps}) # Update progress bar

        if eval_reward>goal_score: break              # Exit if convergence

    print('episodes:{0:5d}, final_reward {1:4.2f}'.format(episodes, eval_reward))
```

```
52%|
522/1000 [00:47<00:14, 33.08it/s, reward=8.35, steps=12.7]
episodes:  522, final_reward 8.35
```

위 코드는 Q-learning 알고리즘을 이용해 강화학습을 수행하는 부분이다. 우선 변수를 초기화한다. History는 학습 중 얻은 보상들을 저장할 딕셔너리, epsF는 epsilon 값을 관리하는 객체이다. 이후 에피소드를 반복한다. epis_rewards, epis_steps, done, state, epsilon을 정의하고 random_number > epsilon, 즉 epsilon 값보다 큰 랜덤 값이 나오면 exploitation, 현재까지 학습된 Q-table에 따라 가장 좋은 행동을 선택하고, 그 반대이면 random_number <= epsilon: exploration, 무작위로 행동을 선택해 새로운 경험을 얻는다. 이후 행동을 환경에 적용하고 결과를 얻고 q-table을 업데이트한다. Q-learning의 핵심부분으로 현재 상태에서 선택된 행동에 대한 Q 값을 갱신한다. $\delta = r_t + \gamma \cdot a' \max Q(s_{t+1}, a') - Q(s_t, a_t)$ 수식은 다음과 같고, np.max(Q_table[next_state])부분은 다음 상태에서 가능한 행동들 중 가장 큰 Q 값을 선택해 미래의 보상을 고려하는 것이다. 이후 보상 및 스텝 수를 누적하고 정책을 평가한다. 반복문의 종료 조건은 eval_reward > goal_score으로 만약 평가 보상이 목표 점수를 초과하면 학습을 종료한다. 이후 최종 결과를 출력하여 확인한 것으로 52%는 완료된 전체 에피소드 수의 비율을 나타내고 522/1000은 총 에피소드 수와 현재까지 완료된 에피소드 수에 해당한다. 보상은 8.35로 에이전트가 가장 최근 에피소드 세트에서 받은 평균 보상을 나타내며 steps의 경우 에이전트가 각 에피소드를 완료하기 위한 평균 단계수를 나타낸다. (taxi의 경우)

```
10%|          | 292/3000 [00:10<02:03, 22.00it/s, reward=0.85, steps=43]
episodes:  292, final_reward 0.85
```

```
print('episodes:{0:5d}, final_reward {1:4.2f}'.format(episodes, eval_reward))
```

```
16%|          | 780/5000 [02:20<10:31,  6.68it/s, reward=-0.031, steps=1.42]
episodes:  780, final_reward -0.03
```

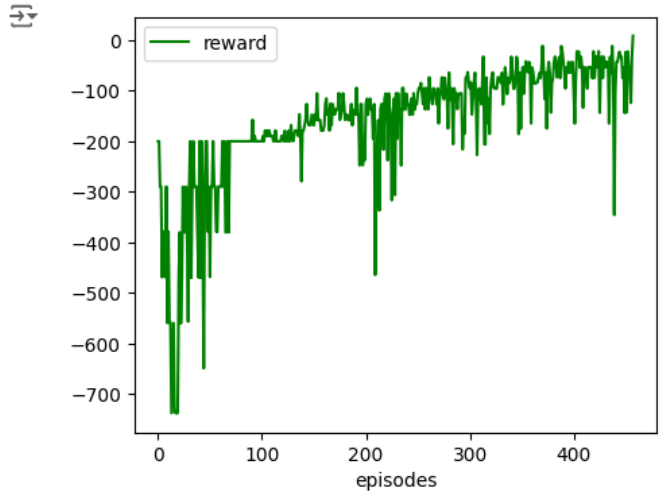
위 사진은 frozen lake, blackjack의 경우이다.

아래의 코드는 강화학습 에이전트의 훈련을 시각화하는 함수를 정의한 것으로 각 플롯은 에피소드에 대한 훈련 성과(reward)를 보여준다. plt.show 함수를 사용해 플롯을 표시한

결과로 나타난 플롯을 통해 매 스탭마다 저장해둔 reward를 확인할 수 있다. 이에 대한 분석은 아래에서 진행하겠다.

Plot Training Histories

```
# 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

# Configuration for plotting
log_labels    = ['rewards']
label_strings = ['reward']
label_colors  = ['g-']
plot_graphs(history, log_labels, label_strings, label_colors)
```



Evaluate the Agent

Evaluate the agent here to show the performance

```
# Evaluate the policy over multiple episodes and calculate the average reward
evaluate_episodes = 20
sum_episode_rewards = 0.0
pbar = tqdm(range(evaluate_episodes))

for i in pbar:
    rewards, _, _ = evaluate_policy(env, Q_table, 1)
    sum_episode_rewards += rewards

env.close()

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

```
100% 20/20 [00:00<00:00, 403.93it/s]
Evaluation Result: -3.15
```

코드는 여러 에피소드에 걸쳐 에이전트가 학습한 정책을 평가한다. evaluate_policy를 사용해 각 에피소드를 시뮬레이션 하고 각 에피소드에 대한 보상을 누적한다. 20개의 에피

소드에 대해 루프를 실행한 후 총 보상을 에피소드 수로 나누어 평균 보상을 얻게 되는 코드이다. 결과는 tqdm을 사용하여 표시된 것으로 20회 평가 후 평균 보상을 확인하는 것이다. (taxi의 경우)

위 사진은 frozen lake, blackjack의 경우이다.

See How The Agent Works

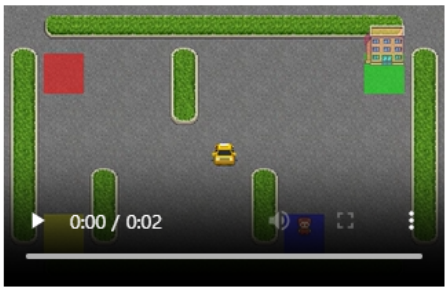
```
# Evaluate the policy in the selected environment and optionally record video or collect images
env = create_env()

if SELECT_ENV != 2:
    env = wrappers.RecordVideo(env, video_folder='./gym-results/', name_prefix=res_prefix)
    eval_reward, _, _ = evaluate_policy(env, Q_table, 1)
else:
    ims = []
    eval_reward, _, ims = evaluate_policy(env, Q_table, 1, ims)

print('Sample Total Reward:', eval_reward)

env.close()
```

```
Sample Total Reward: 12.0
```

선택된 환경에서 학습된 정책을 평가하는 코드로 blackjack이 아닐 경우 비디오를 저장한 후 evaluate_policy 함수가 호출되어 환경에 대한 에이전트의 정책을 평가하고 총 보상, 단계 수 및 비디오를 반환한다. 만약 blackjack인 경우 이미지를 수집하며 evaluate_policy는 환경, q-table, 이미지 저장을 위한 빈 목록과 gak께 호출되며 총 보상과 수집된 이미지를 반환하도록 한다. 마지막으로 총 보상을 출력하며 평가 중 에이전트가 총 12.0의 보상을 얻었음을 의미한다. (taxi의 경우)

위 사진은 frozen lake, blackjack의 경우이다. Frozen_lake의 경우 이미 존재하는 영상이 있어 경고 문고가 뜬 것이다.

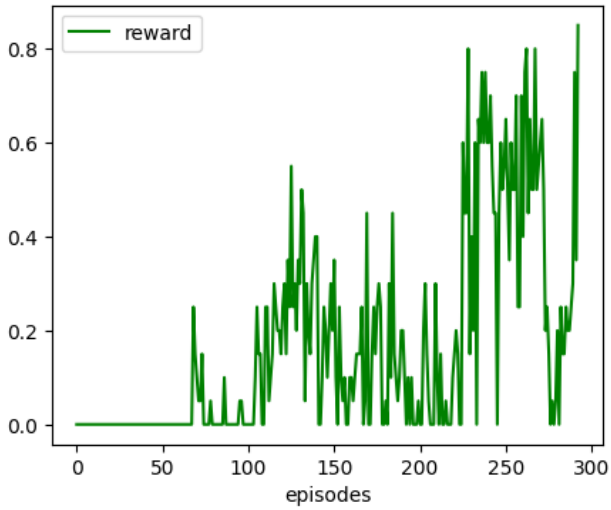
```python
# Display a recorded video or render images from the evaluation depending on the environment type
from IPython.display import HTML, display
from base64 import b64encode

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

if SELECT_ENV==2:
    for i in range(len(ims)):
        plt.figure(figsize=(3,3))
        plt.imshow(ims[i])
        plt.show()
else:
    display(show_video('./gym-results/' + res_prefix + '-episode-0.mp4'))
```



위 코드는 사용된 환경 타입에 따라 에이전트 평가 실행의 녹화된 비디오를 plot하거나 환경상태의 이미지를 렌더링한다. 출력된 결과에 대해서는 아래에서 설명하도록 하겠다.

Simulation results & Discussion

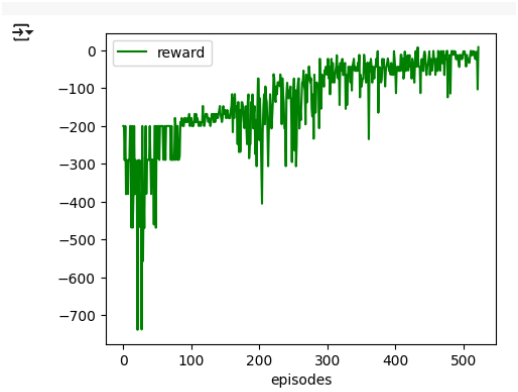
- FrozenLake

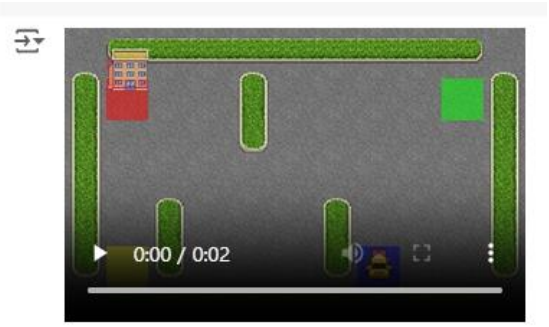


우선 왼쪽 사진은 Frozen lake의 reward로 Frozen lake의 goal score인 0.8로, 목표에 도달시 reward는 +1 ice hole에 도달시 reward는 0, 얼음에 도달했을 땐 reward 0을 주도록 한다. 그래프의 처음에는 평평한 추세를 보여주며 에이전트가 훈련 초기에는 보상을

얻기 위해 어려움이 있었음을 알 수 있다. 시간이 지남에 따라 에피소드 약 100쯤 에이전트가 최적의 경로를 발견하고 얼음 환경을 효과적으로 탐색하는 방법을 학습함에 따라 보상이 증가했음을 알 수 있다. 오른쪽 사진은 Frozen lake의 mp4 영상으로 FrozenLake 환경의 is_slippery가 True로 되어있기 때문에 성공 확률이 적었던 것 같다. 원하는 방향으로 갈 호가률이 1/3, 미끄러짐의 확률이 2/3이기 때문이다. 또한 reward의 차이가 없기 때문에 왔던 길을 반복해서 가게 되어 영상의 길이가 길다.

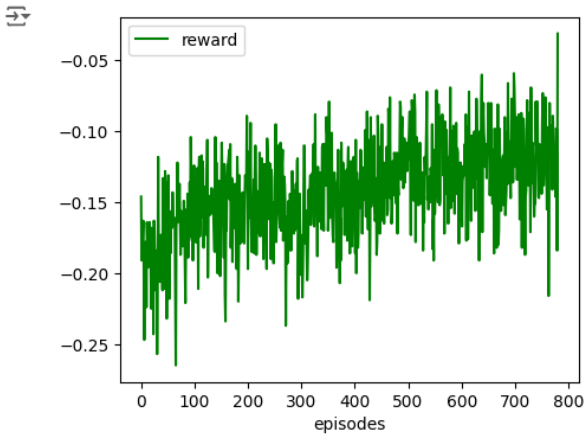
- Taxi





왼쪽 사진은 Taxi의 reward로 Taxi의 goal score인 0.8로, reward가 한 번 이동시 -1 목적지까지 승객을 데려갈 경우 +20, 잘못 내리거나 태우면 -10이다. 그래프에서도 볼 수 있듯이 뚜렷하게 상승하는 추세를 보여준다. 처음에는 reward가 매우 낮았으며 에이전트가 페널티를 받는 경우가 많았음을 확인할 수 있다. 에피소드가 진행됨에 따라 보상이 점차 증가해 페널티를 최소화하고 승객을 효율적으로 승하차했음을 보여준다. 오른쪽 사진은 Taxi의 mp4 영상으로 reward의 크기가 크기 때문에 학습이 쉬웠고 영상의 길이에서도 확인할 수 있듯이 최적의 경로를 찾았음을 확인할 수 있다.

- Blackjack



우선 이 사진은 Blackjack의 reward로 Blackjack의 goal score인 -0.05로, 이길 경우 +1

질 경우 -1 무승부면 0의 reward를 얻는다. 그래프에서 보면 보상이 마이너스 범위에 머물면서 느리고 꾸준한 개선을 보여준다. 운에 따라 결과가 크게 달라질 수 있기 때문에 이점이 그래프에 나타난 것 같다. 또한 그래프에서 볼 수 있듯이 비슷한 모양을 보여주지만 평균이 점점 상향하는 것을 확인할 수 있다. 아래의 그림은 Blackjack의 mp4대신 plot된 이미지로 4와 숨겨진 합이 7을 넘을 수 있을지를 판단했을 때 추가적으로 카드를 요청할지 말지에 대한 것으로 추가 패를 요청하였고 합이 17을 넘을 수 없으므로 더 이상의 요청 없이 종료된 것을 확인할 수 있다.