Mountain Car Problem

학 번	120210203	이 름	권태현
강 의 명	컴퓨터네트워크1	연 구 실	빅데이터처리 및 DB연구실
담당교수님	소정민 교수님	지도교수님	정성원 교수님

Contents

0. Overview

1. A description of the algorithm and key idea

2. The documented code

3. The result graph

4. Lessons learned while implementing and running the code

5. README

6. Reference

0. Overview

*** Overview**

- In this project, you will solve the problem of "Mountain Car Climbing" using reinforcement learning.

*** Requirements**

- You must implement a model-free RL algorithm to train the agent.
- If you train neural networks (such as in DQN or policy gradient), you may use Tensorflow/Keras, but Pytorch is recommended.

1. A description of the algorithm and key idea

*** Algorithm used**

- Deep Q Network (이하 DQN)

*** Explain of Algorithm used**

- Q-Table의 한계를 극복하고자, Function Approximator 역할을 하는 Neural Network를 활용한 방법
- Q-Learning을 통해 {State, Action} pair에 따른 Q-Table(=Q function)이 도출한 value 값이 좋아지는 것이 목표
- Current state가 Next state에 영향을 주는 강화학습에 딥러닝 적용 시, 딥러닝의 데이터 샘플이 서로 독립적이라는 가정이 성립되지 않으므로, Q-Table을 딥러닝 모델에 적용하였을 때 학습이 안되는 문제 발생

*** Experience Replay**

- 위와 같은 문제점을 해결하기 위해 사용하는 방법
- Episode를 진행하면서, 딥러닝과 같이 Weight를 학습하지 않고, Time Step마다 Transition을 학습하는 방법[1)]
- Replay Buffer에 Transition이 일정 이상 누적되었을 경우, Correlation을 피하기 위해 랜덤하게 Sampling하는 방법[1)]

1) Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., and Riedmiller, M. (Dec 2013). Playing Atari with deep reinforcement learning. Technical Report arXiv:1312.5602 [cs.LG], Deepmind Technologies.

*** Replay Buffer of DQN**

- DQN의 Training set인 Transition을 저장하는 저장소로,
  각각의 Transition은 { State, Action, Reward, Next state }로 구성

2. The documented code

*** Import Modules**

```
import numpy as np

import tensorflow as tf
from tensorflow.keras.layers import Dense, Input
from tensorflow.keras import Model
from tensorflow.keras.optimizers import Adam
from keras.models import Sequential

from tensorflow import keras
from collections import deque          # To Saving Transitions
import matplotlib.pyplot as plt        # To Plotting graphs

from tqdm import tqdm
```

*** Make "MountainCar-v0" Environment**

```
import gym
env = gym.make('MountainCar-v0')

env.observation_space
```

```
Box([-1.2  -0.07], [0.6  0.07], (2,), float32)
```

- "MountainCar-v0" 환경 생성

*** Implement the "__init__" function**

```
    def __init__(self, init_state, init_action, episodes):
        self.init_state = init_state
        self.init_action = init_action
        self.episodes = episodes

        self.mini_batch = 64   # 16  32  64  128
        self.gamma = 0.95      # 0.85   0.9  0.95
        self.loss = tf.keras.losses.mean_squared_error
        self.optimizer = tf.keras.optimizers.Adam(learning_rate=0.01)
```

- Batch Size = 64
- Parameter Gamma = 0.95
- Loss Function = Mean Squared Error
- Optimizer = Adam
- Learning rate = 0.01

*** Structure of Model**

```
    def create_model(self):
        state_input = self.init_state

        model = Sequential()
        model.add(Dense(32, input_dim = state_input, activation = 'relu'))
        model.add(Dense(32, activation = 'relu'))

        model.add(Dense(self.init_action))
        return model
```

- Input Layer Size = State size (= env.observation_space.shape[0])
- Hidden Layer1 Size는 32로 주었으며, Relu를 사용
- Hidden Layer2 Size는 32로 주었으며, Relu를 사용
- Output Layer Size = Action space size (= env.action_space.n)

*** Implement the "train_model" function**

```
    def train_model(self, model):

        states, actions, rewards, next_states, dones = sample_experiences(self.mini_batch)
        mask = tf.one_hot(actions, self.init_action)

        next_Q_values = model.predict(next_states)
        max_next_Q_values = np.max(next_Q_values, axis=1)
        target_Q_values = (rewards + (1 - dones) * self.gamma * max_next_Q_values)
        target_Q_values = target_Q_values.reshape(-1, 1)

        with tf.GradientTape() as tape:
            all_Q_values = model(states)
            Q_values = tf.reduce_sum(all_Q_values * mask, axis=1, keepdims=True)
            loss = tf.reduce_mean(self.loss(target_Q_values, Q_values))

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

- 전체 Experiences 세트로 학습을 하지 않고, "sample_experiences" 함수를 사용하여 Time step마다 Replay Buffer의 Transition에 기반하여 학습을 진행
- One-hot Encoding에 의해 나온 결과에 대해 Reduce Sum을 수행한 Q value와 Target Q value 간 Mean Squared Error를 적용하여 Loss값을 계산
- 계산한 Loss값에 대해 "Adam" optimizer를 적용 (Learning rate = 0.01)

*** Create Model**

```
best_score = 200    # Fixed "200"
episodes = 1100     # 1000  1100  1500  2000
timestep = 200      # Fixed "200"

agent = DQNagent(env.observation_space.shape[0], env.action_space.n, episodes)
model = agent.create_model()
rewards = []
```

- 앞서 구현한 클래스 내 함수들을 기반으로 모델 생성

*** Implement the "sample_experiences" function with Replay Buffer**

```
# Create Replay Buffer To Saving Transitions
Replay_Buffer = deque(maxlen=1000)

def sample_experiences(mini_batch):
    idx = np.random.randint(len(Replay_Buffer), size=mini_batch)          # mini_batch = 64
    batch = []
    for index in idx:
        batch.append(Replay_Buffer[index])

    # Transition
    states, actions, rewards, next_states, dones = [np.array([experience[field_index] for experience in batc
    return states, actions, rewards, next_states, dones
```

- 앞서 1-2에서 Correlation 관련 문제를 해결하기 위해 도입한 개념으로,
  Deque 자료형으로 정의한 Replay Buffer 사용
- "sample_experiences" 함수는 Deque로 구현된 Replay Buffer 내 데이터가 Maxsize를
  초과하면 호출
- "mini_batch"로 정의한 값에 따라 랜덤으로 샘플링 (mini_batch = 64)
- { State, Action, Reward, Next state }로 구성된 Transition을 반환

*** Implement the "add_Buffer" function**

```
def add_Buffer(state, action, reward, next_state, done):
    Replay_Buffer.append((state, action, reward, next_state, done))      # Add To Replay_Buffer
```

- "__init__"에서 정의한 Replay Buffer에 Transition을 Add
- Deque는 FIFO 구조이므로, Replay Buffer의 Size 초과 시, 오래된 데이터가 먼저 제거

*** Starting Mountain Car**

```
for episode in tqdm(range(episodes)):
    state = env.reset()

    for step in range(timestep):
        # Step action by using Epsilon-Greedy Policy
        epsilon = max(1 - episode / (episodes * 0.8), 0.01)

        if np.random.rand() >= epsilon:
            action = np.argmax(model.predict(state[np.newaxis])[0])
        else:
            action = np.random.randint(env.action_space.n)

        next_state, reward, done, info = env.step(action)
```

- 각 Episode에 대하여 200번의 step을 진행하며,
  Epsilon-Greedy Policy를 기반으로 action을 취함

*** Score Policy**

```
        if action == 2 and next_state[0] - state[0] > 0:
            reward = 1
        if action == 0 and next_state[0] - state[0] < 0:
            reward = 1

        add_Buffer(state, action, reward, next_state, done)
        state = next_state.copy()

        if done:
            break

    rewards.append(step)

    if step < best_score:
        best_weights = model.get_weights()
        best_score = step
```

- goal에 도달하는데 걸린 iteration step 수로 점수를 측정
- 매 episode마다 최고점에 도달하면 해당 가중치를 "best_weight"로 저장

*** Training Model by using "sample_experiences" function**

```
    if episode > 25:
        agent.train_model(model)

model.set_weights(best_weights)

env.close()
```

- 앞서 설명하였던 Correlation 관련 문제를 해결하기 위해 "sample_experiences" 함수를 사용하여 전체 Experiences 세트로 학습을 하지 않고, episode가 25번 진행된 이후에 Replay Buffer의 Transition에 기반하여 학습을 진행

3. The result graph

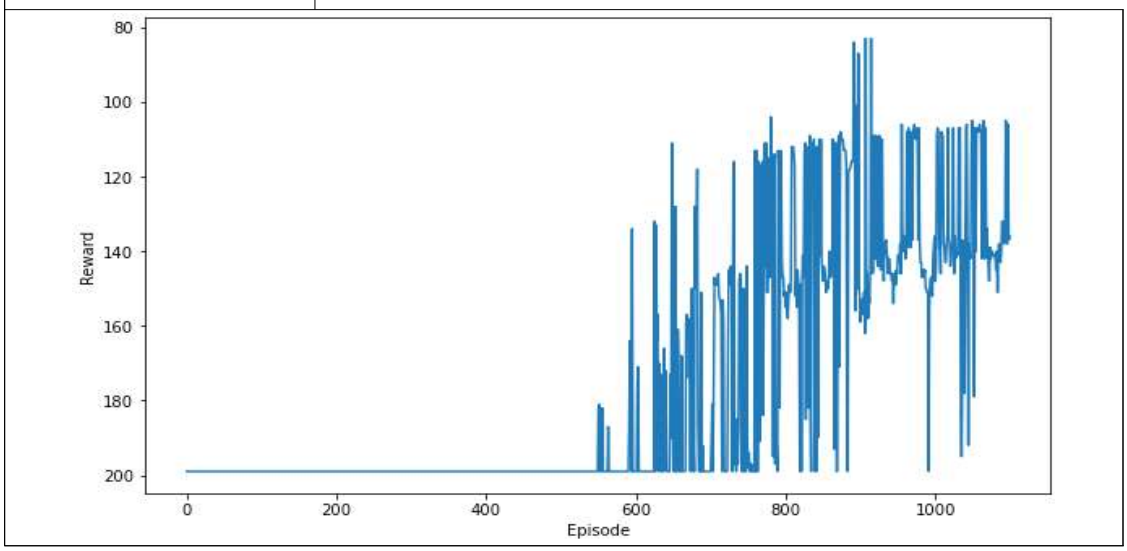
*** Fixed Values**

- learning rate = 0.01
- Episodes = 1100
- Best score, timestep = 200

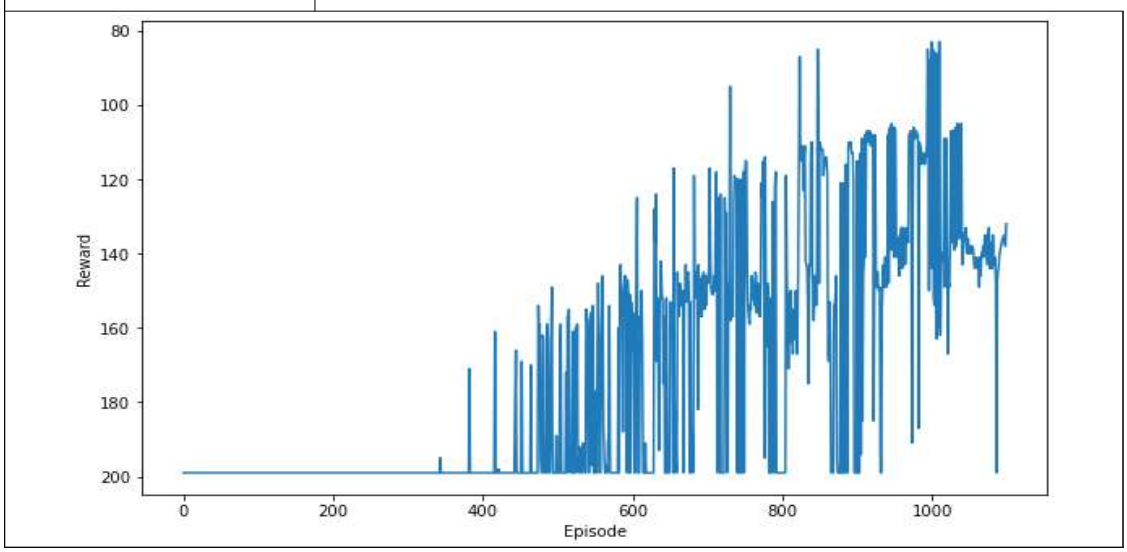
*** Batch Size**

- Gamma = 0.95 Fixed

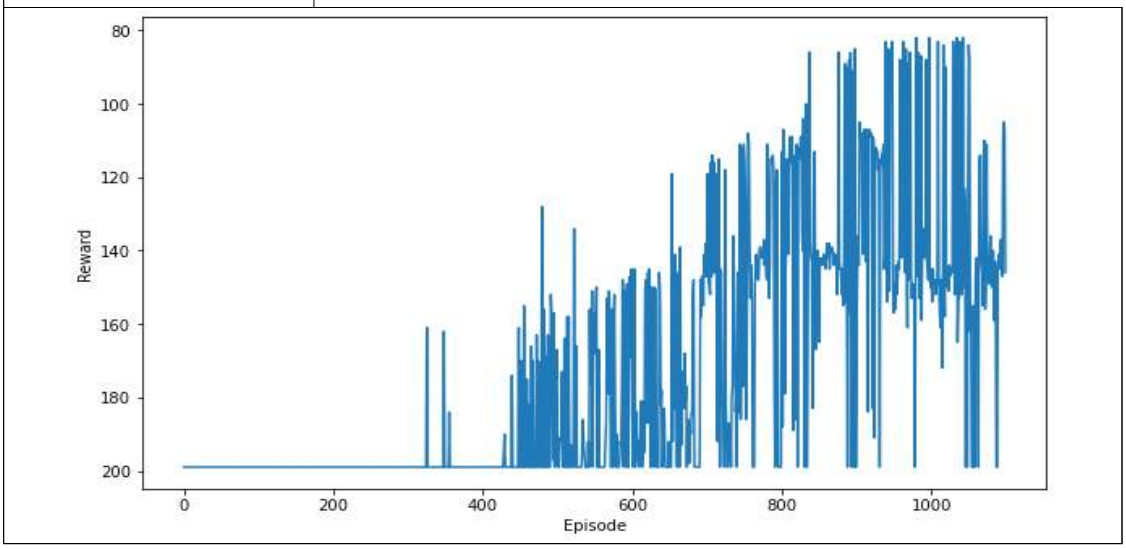
Batch Size = 16



Batch Size = 32



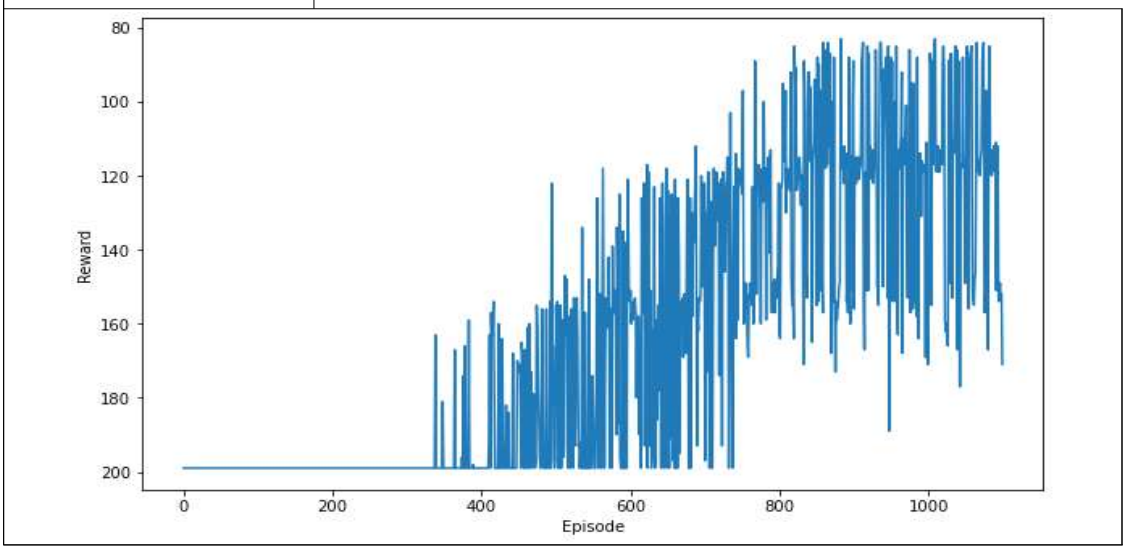
Batch Size = 64



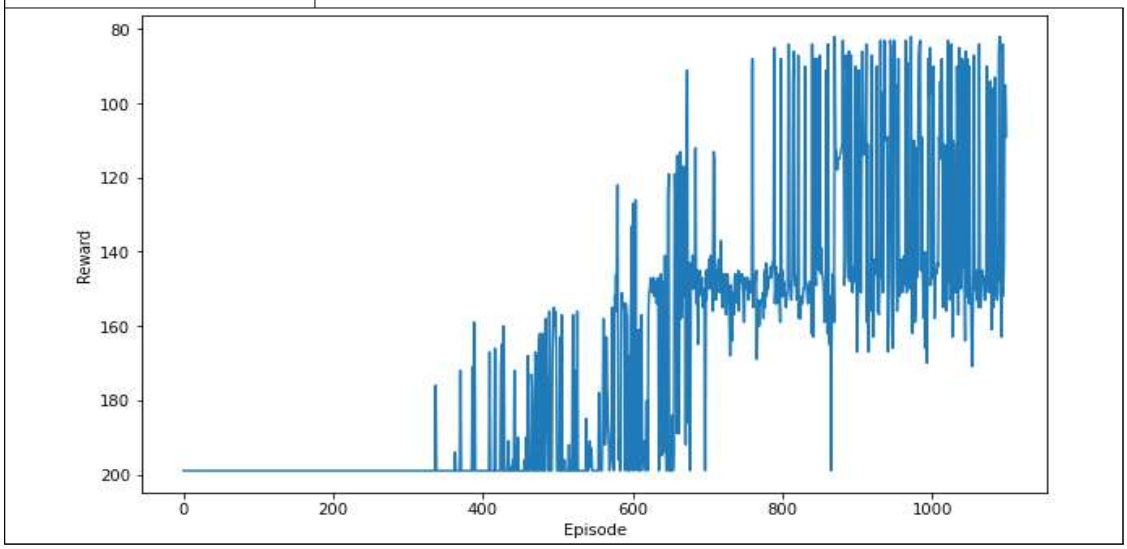
*** Gamma**

- Batch Size = 64 Fixed

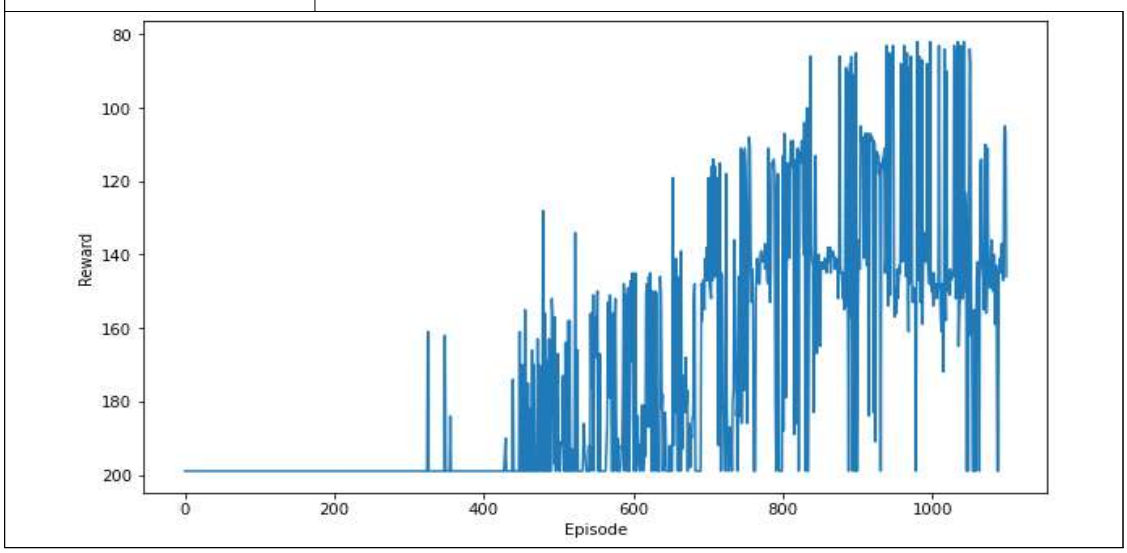
Gamma = 0.85



Gamma = 0.90



Gamma = 0.95



4. Lessons learned while implementing and running the code

실험 초반에 learning rate을 0.001로 두어 실험하였으나, 학습 시간이 너무 오래 걸려 0.01로 고정하고, Episode 또한 적당한 결과를 확인하기 위해 1100으로 고정하여 실험을 진행하였다. 이에 따라 3번의 결과로 미루어 보았을 때, Batch Size가 높았을 때 더 좋은 성능을 보였으나, 연산량이 많아짐에 따라 시간이 오래걸린다는 문제점이 발생했다. 또한, Gamma를 줄이거나 높일수록 목표물을 찾아가는 variance가 높아지고 낮아지는 반비례 관계를 보면서 감마의 값이 0.85 일 때 좋은 성능을 보인 것으로 판단했다.

주 연구분야가 데이터베이스라서 C++ 위주의 코딩과 연구밖에 할 기회가 없었는데 새로운 도전을 하고, 결과를 도출하며 흥미로운 시간을 가졌던 것 같다. 컴퓨터에서 코드를 실행하며 중간중간 수십 번의 꺼짐이 발생하고 했지만, 이번 프로젝트를 통해 Python과 Tensorflow, 그리고 NeuralNet까지 새로운 지식을 견고하게 다지는 계기가 되었다.

5. README

* Python : 3.8.0
* Numpy : 1.20.3
* Tensorflow : 2.7.0
* Jupyter NoteBook

6. Reference

[1] Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., and Riedmiller, M. (Dec 2013). Playing Atari with deep reinforcement learning. Technical Report arXiv:1312.5602 [cs.LG], Deepmind Technologies.

[2] Enginner-Ladder, 티스토리 링크 : https://engineering-ladder.tistory.com/68

[3] Lecture 07: Deep Q Network 강의자료, 소정민, 컴퓨터네트워크 1

[4] John King, Some Stuff, Driving Up A Mountain 링크 : https://jfking50.github.io/mountaincar/

[5] nitish-kalan, Github, MountainCar-v0 with Deep Q-Learning in Keras링크 : https://github.com/nitish-kalan/MountainCar-v0-Deep-Q-Learning-DQN-Keras

[6] 도전적인 HarimKang, DAVINCI-AI, 딥러닝 (9) - [RL2] 신경망 네트워크를 이용한 MountainCar 해결 링크 : https://davinci-ai.tistory.com/33