강화학습 프로젝트

OpenAI Gym에 있는 Mspacman-v0 환경에서 Deep Q-Network Algorithm을 적용

- 10 팀 -

제출일	2020. 07. 02.	**전공**	컴퓨터공학과
과목	지능시스템 소프트웨어	**학번**	15109339
담당교수	한지형 교수님	**이름**	신준수

선택한 환경 ; MsPacman-v0

1. Mspacman-v0 환경을 선택한 간단한 배경

저는 알파고, 머신 러닝 등 평소 인공지능에 관심은 많았지만, 수학에 자신이 없어서 시도해보지 못했었습니다. 하지만 이번 학기에 지능 시스템 소프트웨어라는 강의가 신설되어 시도해보자 마음먹고 신청해서 수강하였습니다. 이번 강의로 처음 접해본 분야라서 그런지 교수님께서 강의로 보여주시던 실습 자료를 따라 실습할 때에 학습이 되는 것을 보고 인공지능 분야에 재미를 느끼게 되었습니다. 그래서 이번 프로젝트를 제대로 진행하고 싶었습니다.

프로젝트의 주제로 정할 환경을 선택할 때, 어릴 적에 했던 팩맨 게임이 생각났습니다. 또한, Supervised Learning 즉, 머신 러닝에도 평소 관심이 많았기 때문에 Supervised Learning 요소와 Reinforcement Learning 요소를 어느 정도 결합한 DQN 알고리즘을 구현해보고 싶어서 이번 기회에 DQN 알고리즘으로 학습하는 팩맨 환경을 구현해보고자 선택하였습니다.

2. Mspacman-v0 환경 소스코드 설명을 포함한 이론적 설명

Mspacman-v0 환경은 OpenAI Gym에서 제공하는 Atari 2600 games 환경 중 하나입니다. OpenAI Gym은 강화 학습을 위한 플랫폼의 하나입니다. 명령어 한, 두 개로 설치할 수 있어 설치하기 쉽고, 표준적인 방식의 다양한 샘플 환경을 제공하기 때문에 강화 학습 연구에 많이 활용되고 있습니다.

OpenAI Gym의 Atari game 환경은 Stella Atari 에뮬레이터를 사용하는 아케이드 학습 환경(The Arcade Learning Environment;ALE)을 통해 시뮬레이션 됩니다. Stella는 무료로 배포되고 있는 다중 플랫폼 Atari 2600 VCS 에뮬레이터로써, Atari 2600 게임을 PC 환경에서 실행할 수 있도록 해주는 에뮬레이터입니다. ALE는 Atari 2600 게임에서 작동하는 인공지능 에이전트를 개발할 수 있는 간단한 객체지향 프레임워크입니다. 앞서 언급한 Atari 2600 에뮬레이터인 Stella 위에 구현되어 에뮬레이션 세부 사항을 에이전트 디자인과 분리하여 주는 역할을 합니다.

Mspacman-v0 환경은 기본적으로 Atari 2600 Pacman의 score, 즉 점수를 최대화하기 위한 환경입니다. Mspacman-v0 환경에서는 세로 210픽셀, 가로 160픽셀의 RGB, 세 개의 채널을 가진 게임 스크린샷 이미지를 사용하여 학습합니다. 게임 스크린샷 이미지의 정보를 (210, 160, 3)의 배열의 형태로 저장한 것을 Network의 Input으로 사용합니다. 또한, 각각의 액션들은 프레임마다 액션이 선택되는 것이 아니라 2, 3, 그리고 4프레임 중 한 프레임 기간을 정하여 일정 프레임 동안 지속하게 되도록 설정되어 있습니다.

다음은 Mspacman-v0 환경의 코드 분석을 통한 조금 더 자세한 Mspacman-v0의 구조에 대한 설명입니다. Mspacman-ram-v0와 같은 코드를 공유하기 때문에 Mspacman-ram-v0에서 쓰이는 부분의 몇몇 코드는 따로 첨부 및 설명을 하지 않았습니다.

Import Necessary Libraries

: 필요한 모듈 및 라이브러리를 import하고 학습에 필요한 설정을 합니다.

```
import numpy as np
import os
import gym
from gym import error, spaces
from gym import utils
from gym.utils import seeding
try:
    import atari_py
except ImportError as e:
    raise error.DependencyNotInstalled(
            "{}. (HINT: you can install Atari dependencies by running "
            "'pip install gym[atari].)".format(e))
```

〉 필요한 라이브러리를 불러옵니다. 이후에 atari_py 라이브러리를 불러오고, 불러오는 중에 Error 발생 시 예외 처리를 해주어 처리합니다.

Define AtariEnv Class

:gym.Env와 utils.EzPickle을 인자로 가지는 AtariEnv 클래스를 작성합니다.

```
class AtariEnv(gym.Env, utils.EzPickle):
    metadata = {'render.modes': ['human', 'rgb_array']}

    def __init__(
            self,
            game='pong',
            mode=None,
            difficulty=None,
            obs_type='ram',
            frameskip=(2, 5),
            repeat_action_probability=0.,
            full_action_space=False):
        """Frameskip should be either a tuple (indicating a random range to
        choose from, with the top value exclude), or an int."""

        utils.EzPickle.__init__(
                self,
                game,
                mode,
                difficulty,
                obs_type,
                frameskip,
                repeat_action_probability)
        assert obs_type in ('ram', 'image')

        self.game = game
        self.game_path = atari_py.get_game_path(game)
        self.game_mode = mode
        self.game_difficulty = difficulty
```

〉 생성자와 비슷한 역할을 하는 __init__ 함수입니다. __init__ 함수는 한 객체에 대한 인스턴스를 생성할 때 호출되는 함수이며, 프레임 스킵 간격, Action 반복의 확률, 게임 모드 등을 인자로 가집니다.

〉 이 오브젝트를 선택하지 않으면 제공된 여러 인자들을 생성자에 전달하여 새 AtariEnv를 생성합니다. 그다음, obs_type이 'ram'이나 'image'이 아닌 경우에 AssertError를 발생시킵니다. Error가 발생하지 않았다면, 해당 클래스의 game, game_path, game_mode, game_difficulty 변수에 해당 변수를 대입해 줍니다.

```
if not os.path.exists(self.game_path):
    msg = 'You asked for game %s but path %s does not exist'
    raise IOError(msg % (game, self.game_path))
self._obs_type = obs_type
self.frameskip = frameskip
self.ale = atari_py.ALEInterface()
self.viewer = None
```

〉 os.path.exists()을 이용하여 해당 경로에 파일이 존재하지 않으면, IOError를 발생시킵니다. Error가 발생하지 않았다면, 해당 클래스의 _obs_type, frameskip에 각각의 해당 변수를, ale에는 atari_py.ALEInterface()를, viewer에는 None을 대입해 줍니다.

```
# Tune (or disable) ALE's action repeat:
# https://github.com/openai/gym/issues/349
assert isinstance(repeat_action_probability, (float, int)), \
        "Invalid repeat_action_probability: {!r}".format(repeat_action_probability)
self.ale.setFloat(
        'repeat_action_probability'.encode('utf-8'),
        repeat_action_probability)
```

〉 https://github.com/openai/gym/issues/349 에 의하면, Atari game 환경은 두 가지의 비결정론적 요소가 존재한다고 합니다. 클래스의 _step 메소드에서 OpenAI에 의해 구현된 2 ~ 4 번의 Action을 반복한다는 것이 존재함과 동시에 Gym 코드에 의해 명시적으로 설정되지 않고 기본값으로 0.25를 갖는 ALEInterface의 repeat_action_probability가 존재합니다. 이를 조치해 주기 위해 assert 문으로 확인해 줍니다.

```
self.seed()
```

〉 이후에 나오는 클래스 내의 seed 함수를 호출하여 random으로 seed를 설정 후 게임 ROM을 로드합니다.

```
self._action_set = (self.ale.getLegalActionSet() if full_action_space
                    else self.ale.getMinimalActionSet())
self.action_space = spaces.Discrete(len(self._action_set))
```

〉 이 클래스의 _action_set을 ALE 프레임워크의 getLegalActionSet()을 사용하여 full_action_space나 ale.getMinimalActionSet()으로 정의하고, 그 길이를 이용하여 이 클래스의 action_space를 정의합니다.

```
(screen_width, screen_height) = self.ale.getScreenDims()
if self._obs_type == 'ram':
    self.observation_space = spaces.Box(low=0, high=255, dtype=np.uint8, shape=(128,))
elif self._obs_type == 'image':
    self.observation_space = spaces.Box(low=0, high=255, shape=(screen_height, screen_width, 3), dtype=np.uint8)
else:
    raise error.Error('Unrecognized observation type: {}'.format(self._obs_type))
```

〉 ALE 프레임워크의 ale.getScreenDims()을 통해 가져온 Screen의 Dimension을 튜플 자료형에 저장합니다. 이후에 이 객체의 _obs_type가 'ram'이나 'image'인지 판단하여 만약 둘 다 아니라면, Error를 발생시킵니다. Error가 발생하지 않았다면 spaces.Box를 이용하여 앞서 정의한 튜플 자료형으로 observation_space를 정의해 줍니다.

```
def seed(self, seed=None):
    self.np_random, seed1 = seeding.np_random(seed)
    # Derive a random seed. This gets passed as a uint, but gets
    # checked as an int elsewhere, so we need to keep it below
    # 2**31.
    seed2 = seeding.hash_seed(seed1 + 1) % 2**31
    # Empirically, we need to seed before loading the ROM.
    self.ale.setInt(b'random_seed', seed2)
    self.ale.loadROM(self.game_path)
```

〉 numpy 모듈의 np_random를 통해 random seed를 정의하는 함수 seed입니다. seed1을 seeding.hash_seed 하여 생성한 seed2는 uint 자료형으로 전달되지만 받는 변수가 int 자료형인 경우가 존재하기 때문에, seed2의 값은 2^{31} 이하로 유지해야 합니다.

〉 game_path를 ROM에서 불러오기 전에 환경이 초기화되면 인수를 설정할 수 있습니다. ALE 프레임워크의 ale.setInt()를 통해 환경 매개 변수를 설정한 후 ale.loadROM()를 통해서 게임 ROM을 로드합니다.

```
if self.game_mode is not None:
    modes = self.ale.getAvailableModes()

    assert self.game_mode in modes, (
        "Invalid game mode \"{}\" for game {}.\nAvailable modes are: {}"
    ).format(self.game_mode, self.game, modes)
    self.ale.setMode(self.game_mode)
```

〉 객체의 game_mode 변수를 확인하여 해당 변수의 값이 유효한 게임 모드가 아

니라면 assert 문으로 에러를 발생시킵니다. 에러가 발생하지 않았다면 객체의 game_mode 변수에 들어 있는 값으로 게임의 모드를 설정합니다.

```
if self.game_difficulty is not None:
    difficulties = self.ale.getAvailableDifficulties()

    assert self.game_difficulty in difficulties, (
        "Invalid game difficulty \"{}\" for game {}.\nAvailable difficulties are: {}"
    ).format(self.game_difficulty, self.game, difficulties)
    self.ale.setDifficulty(self.game_difficulty)

return [seed1, seed2]
```

〉 객체의 game_difficulty 변수를 확인하여 해당 변수의 값이 유효한 게임 난이도가 아니라면 assert 문으로 에러를 발생시킵니다. 에러가 발생하지 않았다면 이 클래스의 game_difficulty 변수에 들어 있는 값으로 게임의 난이도를 설정합니다.

```
def step(self, a):
    reward = 0.0
    action = self._action_set[a]

    if isinstance(self.frameskip, int):
        num_steps = self.frameskip
    else:
        num_steps = self.np_random.randint(self.frameskip[0], self.frameskip[1])
    for _ in range(num_steps):
        reward += self.ale.act(action)
    ob = self._get_obs()

    return ob, reward, self.ale.game_over(), {"ale.lives": self.ale.lives()}
```

〉 정해둔 범위 안에서 랜덤으로 변수 frameskip 값을 정해 그 값만큼 건너 뛰어 선택된 각 스텝의 리워드를 계산하는 함수 step입니다. step 함수는 ob, reward, ale.game_over(), ale.lives()를 반환합니다. DQN 알고리즘 코드에서 반환받은 값을 각각 obs, reward, done, info 변수에 저장하는 식으로 사용하고, *obs, reward, done, info = env.step(variable)*의 형태로 사용했습니다.

```
def _get_image(self):
    return self.ale.getScreenRGB2()
```

〉 ale.getScreenRGB2()를 이용하여 이미지를 반환하는 함수 _get_image입니다.

```
@property
def _n_actions(self):
    return len(self._action_set)
```

〉 @property를 사용하여 이 객체의 _action_set의 길이 값을 가져와 반환하는 함수 _n_actions입니다.

```
def _get_obs(self):
    if self._obs_type == 'ram':
        return self._get_ram()
    elif self._obs_type == 'image':
        img = self._get_image()
    return img
```

〉 _obs_type을 확인하여 _obs_type이 'image'이면, 앞서 정의한 _get_imag를 호출하여 img 변수에 대입하고 img를 반환하는 함수 _get_obs입니다.

```
# return: (states, observations)
def reset(self):
    self.ale.reset_game()
    return self._get_obs()
```

〉 ale.reset_game()을 통해 게임을 초기화하고 반환 값으로 _get_obs 함수를 실행하는 함수 reset입니다. DQN 알고리즘 코드에서 학습할 때와 테스트할 때 등 게임이 종료되어 다시 시작할 때 환경을 초기화해주기 위하여 *obs = env.reset()* 형태로 사용했습니다.

```
def render(self, mode='human'):
    img = self._get_image()
    if mode == 'rgb_array':
        return img
    elif mode == 'human':
        from gym.envs.classic_control import rendering
        if self.viewer is None:
            self.viewer = rendering.SimpleImageViewer()
        self.viewer.imshow(img)
        return self.viewer.isopen
```

〉 렌더링을 하기 위한 함수 render를 정의합니다. mode가 'rgb_array'일 경우엔 img를 반환하고, 'human'일 경우에는 rendering 라이브러리를 불러와 이 클래스의 viewer가 None이라면 rendering.SimpleImageViewer()를 통해 렌더링을 수행하고 viewer.isopen을 반환합니다. DQN 알고리즘 코드에서는 학습이 진행 중이거나, 완료된 이후에 학습 결과를 눈으로 확인하기 위해 사용했습니다. render 함수의 인자인 mode를 'rgb_array'로 설정해 주어 img를 반환하여 반환받은 값을 DQN 코드의 img 변수에 저장합니다. 실제 코드에선 *img = env.render(mode="rgb_array")* 형태로 사용했습니다.

```
def close(self):
    if self.viewer is not None:
        self.viewer.close()
        self.viewer = None
```

〉 앞서 정의한 render 함수에서 정의하였던 self.veiwer가 None이 아니라면 viewer.close()를 실행하여 viewer를 닫고 None으로 값을 변경해줍니다.

```
def get_action_meanings(self):
    return [ACTION_MEANING[i] for i in self._action_set]
```

〉 코드의 마지막에 정의된 딕셔너리 ACTION_MEANING에서 값을 가져와서 반환하는 함수 get_action_meanings입니다.

```
def get_keys_to_action(self):
    KEYWORD_TO_KEY = {
        'UP':      ord('w'),
        'DOWN':    ord('s'),
        'LEFT':    ord('a'),
        'RIGHT':   ord('d'),
        'FIRE':    ord(' '),
    }

    keys_to_action = {}

    for action_id, action_meaning in enumerate(self.get_action_meanings()):
        keys = []
        for keyword, key in KEYWORD_TO_KEY.items():
            if keyword in action_meaning:
                keys.append(key)
        keys = tuple(sorted(keys))

        assert keys not in keys_to_action
        keys_to_action[keys] = action_id

    return keys_to_action
```

〉 에이전트가 할 수 있는 액션의 집합과 키보드의 자판으로 나타낼 수 있는 키를 매핑해주는 함수 get_keys_to_action입니다. 'w', 's', 'a', 'd', ' '를 아스키코드 값으로 반환해 주는 Python의 내장함수 ord를 이용하여 딕셔너리 자료형으로 에이전트가 게임에서 할 수 있는 행동을 정의합니다. 그리고 앞에서 정의된 함수 get_action_meanings의 값을 이용하여 새로운 딕셔너리 keys_to_action에 아스키코드로 변환된 키의 조합에 ACTION_MEANING의 키값인 action_id를 매핑하여 저장하고 반환합니다. 다음은 DQN 알고리즘 코드에서 *env.get_keys_to_action()*을 실행한 결과입니다.

```
env.get_keys_to_action()
```

```
{(): 0,
 (97,): 3,
 (97, 115): 8,
 (97, 119): 6,
 (100,): 2,
 (100, 115): 7,
 (100, 119): 5,
 (115,): 4,
 (119,): 1}
```

```
def clone_state(self):
    """Clone emulator state w/o system state. Restoring this state will
    *not* give an identical environment. For complete cloning and restoring
    of the full state, see `{clone,restore}_full_state()`."""
    state_ref = self.ale.cloneState()
    state = self.ale.encodeState(state_ref)
    self.ale.deleteState(state_ref)
    return state
```

〉system states 없이 에뮬레이터 상태를 복제하는 함수 clone_state입니다. 이 함수를 통하여 상태를 복원해도 동일한 환경이 제공되지는 않습니다. 전체 상태를 완전히 복제하고 복원하려면`{clone, restore} _full_state ()`를 참조해야 합니다.

```
def restore_state(self, state):
    """Restore emulator state w/o system state."""
    state_ref = self.ale.decodeState(state)
    self.ale.restoreState(state_ref)
    self.ale.deleteState(state_ref)
```

〉system states 없이 에뮬레이터 상태를 복원하는 함수 restore_state입니다.

```
def clone_full_state(self):
    """Clone emulator state w/ system state including pseudorandomness.
    Restoring this state will give an identical environment."""
    state_ref = self.ale.cloneSystemState()
    state = self.ale.encodeState(state_ref)
    self.ale.deleteState(state_ref)
    return state
```

〉Pseudorandomness(의사 난수)를 포함한 system states로 전체 에뮬레이터 상태를 복제하는 함수 clone_full_state입니다. 이 함수를 통하여 상태를 복원하면 동일한 환경이 제공됩니다.

```
def restore_full_state(self, state):
    """Restore emulator state w/ system state including pseudorandomness."""
    state_ref = self.ale.decodeState(state)
    self.ale.restoreSystemState(state_ref)
    self.ale.deleteState(state_ref)
```

〉Pseudorandomness(의사 난수)를 포함한 system states로 전체 에뮬레이터 상태를 복원하는 함수 restore_full_state입니다.

Define ACTION_MEANING Dictionary

: 에이전트의 액션을 숫자 key값에 맵핑하는 딕셔너리입니다.

```
ACTION_MEANING = {
    0: "NOOP",
    1: "FIRE",
    2: "UP",
    3: "RIGHT",
    4: "LEFT",
    5: "DOWN",
    6: "UPRIGHT",
    7: "UPLEFT",
    8: "DOWNRIGHT",
    9: "DOWNLEFT",
    10: "UPFIRE",
    11: "RIGHTFIRE",
    12: "LEFTFIRE",
    13: "DOWNFIRE",
    14: "UPRIGHTFIRE",
    15: "UPLEFTFIRE",
    16: "DOWNRIGHTFIRE",
    17: "DOWNLEFTFIRE",
}
```

〉0부터 17까지의 숫자 key값에 Atari game에서 에이전트가 할 수 있는 액션을 맵핑한 딕셔너리 ACTION_MEANING입니다.

적용한 강화학습 알고리즘 ; Deep Q-Network

1. DQN에 대한 이론적 설명

이 DQN은 매우 유명하고 널리 쓰이는 Deep Reinforcement Learning(DRL) algorithm입니다. DQN이라는 개념은 AlphaGo를 개발한 DeepMind의 *"Playing Atari with Deep Reinforcement Learning"*이라는 논문에서 처음 소개되었습니다. 해당 논문에서는 강화학습의 Q-network에 아래 세 가지 사항을 반영하여 Q-networks에 Deep Learning을 적용한 Deep Q-network를 정의하였습니다.

1. CNN을 적용하여 게임 스크린샷의 픽셀을 Input data로 입력받는 네트워크.
2. Experience Replay를 사용하는 네트워크.
- Replay memory에 이전에 경험한 Transition pair들을 저장하고 재사용.
3. Q value를 별개로 구성한 Target Network에서 학습시키는 네트워크.

CNN ; Convolutional Neural Network

Atari game의 raw pixel을 직접적으로 Input data로 사용하기 위해 CNN Network를 사용하였습니다. 이는 픽셀들을 독립적인 Input으로 정의하기보다는 주변 픽셀과 함께 하나의 Region으로 인식하기 위함입니다. 또한, CNN이 인간의 시각세포를 이용하여 시각을 얻는 방법과 유사하다는 연구결과도 존재합니다.

CNN은 Deep Learning에 가장 많이 사용되는 알고리즘으로, 이미지에서 객체를 인식하기 위해 패턴을 찾는 것에 유용하게 쓰이는 알고리즘입니다. CNN은 데이터에서 직접 학습하며, 패턴을 사용하여 이미지를 분류하고 특징을 수동으로 추출할 필요가 없습니다. CNN은 각 레이어의 입출력 데이터의 형상이 유지된다는 점, 이미지의 공간 정보를 유지하면서 인접 이미지와의 특징을 효과적으로 인식한다는 점, 복수의 필터를 사용하여 이미지의 특징 추출 및 학습을 한다는 점과 같은 기존의 Fully Conneted Neural Network와 차별성을 갖습니다.

DQN에서는 첫 번째 레이어로 Convolutional network를 사용하고 있는데, 앞서 언급하였듯이 게임의 이미지를 network의 State로 사용하고 210X160 픽셀의 RGB 컬러를 가진 원래의 게임 이미지를 다운 샘플링하여 84X84 픽셀의 흑백 컬러로 전환하여 사용합니다. 여기서 일반 CNN 과는 다르게 Pooling layer는 사용하지 않는데, 그 이유는 Pooling layer에서는 보통 이미지에 있는 내용을 압축하는 역할을 하는데 이것은 이미지에 오브젝트가 있는지 만을 판단할 때는 유용하지만 DQN에서는 이미지에서 오브젝트들의 공간적 관계를 인식하기 때문에 오브젝트들의 포지션이 중요하므로 Polling layer를 사용하지 않습니다.

Experience Replay a.k.a. replay buffer

DQN 이전의 environment와 상호작용하며 얻어진 on-policy sample들을 통해 파라미터를 업데이트하던 방법은 sample에 대한 의존성이 커서 policy가 converge 하지

못하고 oscillate 할 수 있게 됩니다. 이러한 문제를 해결하고자, DQN에는 Experience Replay라는 개념이 도입됩니다.

Experience Replay은 DQN에서 중요한 개념입니다. 쉽게 말하면, 에이전트의 경험을 저장하는 버퍼로, 트랜지션을 〈s, a, r, s'〉의 형태로 버퍼에 저장하는 것을 말합니다. 큐 형태에 가까운 Experience Replay은 DQN의 입력으로 리플레이 버퍼에 저장되어 있는 트랜지션의 일부를 random batch로 샘플링해서 입력으로 넣습니다. 그 이유는 Atari game의 경우, 학습 데이터가 시간의 흐름에 따라 순차적으로 수집되기 때문에, 에이전트의 경험이 시간 기준으로 Correlation(연관성)이 강합니다. 그렇기 때문에 앞의 행동과 그다음 행동들의 Correlation을 줄이면서 학습하기 위함입니다. 에이전트의 액션들의 Correlation을 줄여야 하는 이유는 Correlation이 강한 상태로 학습을 시키면 네트워크가 그 상황에서만 동작하는 일반화되지 않은 네트워크로 학습이 되는 Overfitting 문제의 발생 때문입니다.

Target Network

하나의 Network를 사용하지 않고 분리된 Target Network를 둔다는 것은 업데이트를 하는 동안 Target Q value을 계산하는 $y_i = E_{s\sim\varepsilon}[r + \gamma \max_a Q(s', a; \theta_{i-1}) \mid s, a]$ 식에서, 계산에 사용되는 파라미터를 고정한다는 것입니다. 즉, Network에 사용되는 파라미터들을 θ_i라고 할 때, 파라미터 θ를 Loss function인 $L_i(\theta_i) = E_{s,\ a\sim\rho(\cdot)}[(y_i - Q(s, a; \theta_i))^2]$에 의해 업데이트하는 동안 θ_{i-1}은 고정합니다. 이렇게 Target Network을 따로 두는 이유 즉, 이전의 파라미터를 고정하는 이유는, Target의 값을 구할 때 더욱 안정적으로 계산하기 위해서입니다. 위의 식에서 Target value는 파라미터의 영향을 크게 받기 때문에 별개의 Network를 구성하여 안정적인 계산을 할 수 있습니다. 만약 이렇게 하지 않는다면, Target Q value를 구할 때, 파라미터에 영향을 크게 받아 무한한 Feedback Loop에 빠지게 될 수 있고, Estimated Q value가 Target으로 안정적으로 업데이트되지 않을 수 있습니다. 따라서, DQN에서는 우리가 업데이트할 Target Network의 파라미터를 고정해두고 업데이트합니다.

이 프로젝트에선 간소화한 식인 $Loss = \left(y_i - Q(s, a; \theta)\right)^2$ 식을 사용하여 Loss를 줄이는 방향으로 θ를 업데이트하고, Weight를 업데이트할 때에는 Gradient descent를 사용하여 업데이트합니다.

또한, Output layer에서 출력으로 뉴런의 개수를 액션의 개수만큼 넣어서 모든 각각의 행동에 대한 Q value를 계산하는 역할을 하게 합니다. 이 방법은 일반적인 네트워크를 한 번 돌려야 Q value가 나오는 네트워크와 비교하면 훨씬 경제적인 방법이라 할 수 있습니다.

이 프로젝트에서 사용한 DQN 알고리즘 이외에도, 논문에서 DQN이 소개된 이후, 일반적인 DQN 알고리즘의 문제점을 개선하고 성능을 향상시킨 Double DQN, Dueling DQN, Prioritized Experience Replay를 사용하는 DQN 등의 많은 DQN Architecture가 연구되었습니다.

2. 작성한 DQN 알고리즘 소스코드 설명

Import Necessary Libraries

: 필요한 모듈 및 라이브러리를 import하고 학습에 필요한 설정을 합니다.

```
from __future__ import division, print_function, unicode_literals
import gym
import numpy as np
import os
import sys
import tensorflow as tf
```

〉 사용할 모듈 gym, numpy, os, sys, tensorflow 등을 불러왔습니다.

```
# matplotlib 설정
from IPython.display import HTML
import matplotlib
import matplotlib.animation as animation
import matplotlib.pyplot as plt
plt.rcParams['axes.labelsize'] = 14
plt.rcParams['xtick.labelsize'] = 12
plt.rcParams['ytick.labelsize'] = 12
```

〉 matplotlib을 import하고, rcParams를 통하여 label의 크기를 설정해주었습니다.

```
# 일관된 출력을 위해 유사난수 초기화
def reset_graph(seed=42):
    tf.reset_default_graph()
    tf.set_random_seed(seed)
    np.random.seed(seed)
```

〉 numpy의 random 모듈을 이용하여 유사난수를 초기화해주는 reset_graph() 함수를 정의해주었습니다.

```
env = gym.make("MsPacman-v0")
obs = env.reset()
obs.shape
env.action_space
```

〉 openAI Gym의 Mspacman-v0 환경을 불러오고, 변수 obs에 초기화한 환경을 저장하였습니다. 그 이후에 obs의 배열 크기와 Mspacman-v0 환경의 Action 개수를 확인해주었습니다.

Define Preprocess Function

: 학습 효율을 높이기 위해 게임 스크린샷 이미지를 전처리합니다.

```
mspacman_color = 210 + 164 + 74

def preprocess_observation(obs):
    img = obs[1:176:2, ::2] # 자르고 크기를 줄입니다.
    img = img.sum(axis=2) # 흑백 스케일로 변환합니다.
    img[img==mspacman_color] = 0 # 대비를 높입니다.
    img = (img // 3 - 128).astype(np.int8) # -128~127 사이로 정규화합니다.
    return img.reshape(88, 80, 1)

img = preprocess_observation(obs)
```

〉 Input으로 들어오는 이미지들이 학습하기에 크기 때문에 이미지를 다운 샘플링하여 88X80 픽셀로 줄이고 Mspacman의 대비를 향상시키기 위한 간단한 전처리 함수를 만들었습니다. 수업에서 쓰인 전처리 함수와는 다르게 64비트 부동 소수를 -1.0~1.0 사이로 나타내지 않고 부호가 있는 바이트(-128~127)로 표현하였습니다. 이렇게 한 이유는 정밀도를 감소시켜도 눈에 띄게 학습이 미치는 영향이 없고, 재생 메모리가 52GB에서 6.5GB로 매우 적게 소모되기 때문입니다.

〉 게임 스크린샷 이미지의 전처리 작업은 선택 사항이지만 훈련 속도를 크게 높

여주고 비용을 절감할 수 있습니다.

Define Q-network

: 3개의 Convolutional Layer로 구성되고 그 뒤에 2개의 Fully-connected Layer와 Output Layer가 이어지는 구조로 DQN을 구성합니다.

```
reset_graph()

input_height = 88
input_width = 80
input_channels = 1
conv_n_maps = [32, 64, 64]
conv_kernel_sizes = [(8,8), (4,4), (3,3)]
conv_strides = [4, 2, 1]
conv_paddings = ["SAME"] * 3
conv_activation = [tf.nn.relu] * 3
n_hidden_in = 64 * 11 * 10  # conv3은 11x10 크기의 64개의 맵을 가집니다
n_hidden = 512
hidden_activation = tf.nn.relu
n_outputs = env.action_space.n  # 9개의 행동이 가능합니다
initializer = tf.variance_scaling_initializer()
```

```
def q_network(X_state, name):
    prev_layer = X_state / 128.0 # 픽셀 강도를 [-1.0, 1.0] 범위로 스케일 변경합니다.
    with tf.variable_scope(name) as scope:
        for n_maps, kernel_size, strides, padding, activation in zip(
                conv_n_maps, conv_kernel_sizes, conv_strides,
                conv_paddings, conv_activation):
            prev_layer = tf.layers.conv2d(
                prev_layer, filters=n_maps, kernel_size=kernel_size,
                strides=strides, padding=padding, activation=activation,
                kernel_initializer=initializer)
        last_conv_layer_flat = tf.reshape(prev_layer, shape=[-1, n_hidden_in])
        hidden = tf.layers.dense(last_conv_layer_flat, n_hidden,
                                 activation=hidden_activation,
                                 kernel_initializer=initializer)
        outputs = tf.layers.dense(hidden, n_outputs,
                                  kernel_initializer=initializer)
    trainable_vars = tf.get_collection(tf.GraphKeys.TRAINABLE_VARIABLES,
                                       scope=scope.name)
    trainable_vars_by_name = {var.name[len(scope.name):]: var
                              for var in trainable_vars}
    return outputs, trainable_vars_by_name
```

〉 이 부분에서 DQN 구조의 입력의 크기, 채널 수, 커널 크기, 스트라이드, Activation Function 등의 여러 하이퍼 파라미터를 정의하였습니다.

〉 DQN을 만들기 위해 Mspacman 환경의 State인 X_State와 변수 범위의 이름을 입력으로 받는 q_network() 함수를 정의해 주었습니다.

〉 이 네트워크는 State-Action Pair을 입력받고 이에 맞는 Q Value Q(s, a)의 예측값을 출력합니다. 동일한 구조이지만 모델 파라미터는 다른 DQN 2개를 사용하였습니다. Actual DQN은 Mspacman을 조정하는 방법을 학습하고, Target DQN은 Actual DQN을 훈련하기 위한 Target Q value를 만드는데 사용합니다. 그리고 일정한 간격으로 Actual DQN을 Target DQN으로 복사하여 파라미터를 교체합니다. 그리고 Actual DQN과 Target DQN은 같은 구조를 가진 Network이어야 하므로 같은 두 개의 DQN을 구성하기 위해 q_network() 함수를 만들어서 구성했습니다.

〉 위 코드의 trainable_vars_by_name 딕셔너리는 이 DQN에 있는 모든 훈련 가능한 변수를 담고 있습니다. 이 딕셔너리는 Actual DQN을 Target DQN으로 복사하는 연산을 만들 때 사용하기 위해 구현하였습니다.

```
X_state = tf.placeholder(tf.float32, shape=[None, input_height, input_width,
                                            input_channels])
online_q_values, online_vars = q_network(X_state, name="q_networks/online")
target_q_values, target_vars = q_network(X_state, name="q_networks/target")

copy_ops = [target_var.assign(online_vars[var_name])
            for var_name, target_var in target_vars.items()]
copy_online_to_target = tf.group(*copy_ops)
```

〉 위의 코드에서 Input placeholder와 두 개의 DQN, 그리고 Actual DQN을 Target DQN으로 복사하는 연산을 구현해 주었습니다.

〉 Placeholder는 데이터를 상숫값을 전달함과 같이 할당하는 것이 아니라 다른 텐서를 placeholder에 매핑시키는 tensorflow의 자료형이라고 할 수 있습니다.

〉 copy_online_to_target 연산을 Actual DQN의 모든 trainable 변수를 이에 상응하는 Target DQN의 변수에 복사하기 위해 구현하였습니다. 또한, tensorflow의 tf.group() 함수를 사용하여 모든 할당 연산을 사용이 쉽도록 하나의 연산으로 묶어주었습니다.

```
learning_rate = 0.001
momentum = 0.95

with tf.variable_scope("train"):
    X_action = tf.placeholder(tf.int32, shape=[None])
    y = tf.placeholder(tf.float32, shape=[None, 1])
    q_value = tf.reduce_sum(online_q_values * tf.one_hot(X_action, n_outputs),
                            axis=1, keep_dims=True)
    error = tf.abs(y - q_value)
    clipped_error = tf.clip_by_value(error, 0.0, 1.0)
    linear_error = 2 * (error - clipped_error)
    loss = tf.reduce_mean(tf.square(clipped_error) + linear_error)

    global_step = tf.Variable(0, trainable=False, name='global_step')
    optimizer = tf.train.MomentumOptimizer(learning_rate, momentum,
                                           use_nesterov=True)
    training_op = optimizer.minimize(loss, global_step=global_step)

init = tf.global_variables_initializer()
saver = tf.train.Saver()
```

〉 Learning rate와 optimizer에 사용할 momentum을 정의해 주었습니다.

〉 위의 코드에서 Actual Network을 학습시키기 위한 연산을 구현했습니다. 그러기 위해선 먼저 메모리 배치에 있는 각각의 State-Action Pair에 대한 estimated Q vaule를 계산해야 하기 때문에, 실제로 수행한 Action에 대응하는 Q Value만 저장해 주었습니다. 이렇게 하기 위해 Action을 one_hot 벡터로 바꾸고, Q value를 곱해 주어 현재 수행된 Action에 대한 Q Value를 제외하고 모두 0으로 만들어주었습니다. 그런 다음 각 메모리 배치에 대한 estimated Q value을 얻기 위해 첫 번째 축을 기준으로 더해주었습니다.

〉 자료를 검색하던 중 찾은 내용 중 Loss를 오로지 차이의 제곱 오차로 계산하는 것보다 작은 오차에 대해서만 제곱 오차를 이용하고, 큰 오차에 대해서는 오차의 2를 곱해주는 방법이 낫다는 내용을 참고해보았습니다. 또한, 이 방법을 사용 시, 수업 시에 등장했던 Adam Optimizer 대신에 Nesterov Optimizer를 사용하는 것이 더 낫다는 점도 참고하여 구현하였습니다.

〉 Q value를 제공하는 용도로 사용될 Placeholder y를 만들고 손실을 계산하였습니다. 이때, 검색해서 찾은 내용을 바탕으로 Loss를 차이의 제곱 오차로만 정의하지 않

고, 1.0을 기준으로 제곱과 2배수를 사용하여 Loss를 계산해 주었습니다.

Define Replay buffer

: 학습에 사용할 크기가 500,000인 Replay buffer입니다.

```
class ReplayMemory:
    def __init__(self, maxlen):
        self.maxlen = maxlen
        self.buf = np.empty(shape=maxlen, dtype=np.object)
        self.index = 0
        self.length = 0

    def append(self, data):
        self.buf[self.index] = data
        self.length = min(self.length + 1, self.maxlen)
        self.index = (self.index + 1) % self.maxlen

    def sample(self, batch_size, with_replacement=True):
        if with_replacement:
            indices = np.random.randint(self.length, size=batch_size) # 더 빠름
        else:
            indices = np.random.permutation(self.length)[:batch_size]
        return self.buf[indices]
```

〉 deque를 사용하는 방법도 있지만, 수업 중 DQN의 요소 중에서 Replay buffer가 중요한 요소라고 배웠기 때문에 ReplayMemory 클래스를 사용하여 한 번 구현해보고자 했습니다.

〉 ReplayMemory Class : 최근 관찰된 transition을 보관, 유지하는 제한된 크기의 순환 버퍼입니다. 또한, 연관성(correlation)을 줄이는 학습을 위한 transition을 random batch하는 sample() 함수를 제공하여 줍니다.

〉 Atari game의 경우, 에이전트의 경험이 시간 기준으로 연관성(correlation)이 강한 환경이기 때문에, 경험 간의 연관성을 줄이면서 학습하기 위해 random으로 batch하여 트레이닝 샘플을 선택하도록 구현하였습니다.

〉 검색해서 찾은 내용 중 중복을 허용하여 샘플링하면 큰 Replay buffer에서 중복을 허용하지 않고 샘플링하는 것보다 빠르다고 한 것을 참고하여 구현하였습니다.

```
replay_memory_size = 500000
replay_memory = ReplayMemory(replay_memory_size)
```

〉 Replay buffer의 사이즈를 500,000으로 정의해주었습니다.

```
def sample_memories(batch_size):
    cols = [[], [], [], [], []] # 상태, 행동, 보상, 다음 상태, 계속
    for memory in replay_memory.sample(batch_size):
        for col, value in zip(cols, memory):
            col.append(value)
    cols = [np.array(col) for col in cols]
    return cols[0],cols[1],cols[2].reshape(-1, 1),cols[3],cols[4].reshape(-1, 1)
```

〉 설정한 하이퍼파라미터인 batch_size 만큼의 경험을 메모리에서 샘플링하도록 구현하였습니다.

```
eps_min = 0.1
eps_max = 1.0
eps_decay_steps = 2000000

def epsilon_greedy(q_values, step):
    epsilon = max(eps_min, eps_max - (eps_max-eps_min) * step/eps_decay_steps)
    if np.random.rand() < epsilon:
        return np.random.randint(n_outputs) # exploration
    else:
        return np.argmax(q_values) # exploitation
```

〉에이전트가 따를 ε-greedy policy를 정의해 주었습니다. numpy 모듈의 rand() 함수를 통해 난수를 생성하여 ε보다 난수가 작을 경우엔 exploration을 하도록 설정하였고, 난수가 ε보다 클 경우에는 Optimal한 액션을 취하는 exploitation을 하도록 설정하였습니다.

〉ε은 초반 학습에 더욱 큰 비중을 주기 위해 2,000,000 개의 스텝을 지나면서 점진적으로 1.0에서 0.1까지 감소시키도록 구현하였습니다.

```
n_steps = 4000000  # 전체 훈련 스텝 횟수
training_start = 10000  # 10,000번 게임을 반복한 후에 훈련 시작
training_interval = 4  # 4번 게임을 반복하고 훈련 스텝 실행
save_steps = 1000  # 1,000번 훈련 스텝마다 모델 저장
copy_steps = 10000  # 10,000번 훈련 스텝마다 Actual DQN을 Target DQN으로 복사
discount_factor = 0.99
skip_start = 90  # 게임의 시작 부분은 스킵
batch_size = 50
iteration = 0  # 게임 반복횟수
checkpoint_path = "./my_dqn.ckpt"
done = True # 환경 리셋의 필요 여부 판단
```

〉대표적으로 Discount Factor와 학습을 진행할 스텝의 총 횟수, Actual DQN을 Target DQN으로 복사할 일정 간격 스텝, 체크포인트를 저장할 파일 등의 하이퍼파라미터와 변수를 정의해주었습니다.

```
loss_val = np.infty
game_length = 0
total_max_q = 0
mean_max_q = 0.0
```

〉이전의 하이퍼파라미터 이외에 학습과정을 추적하여 관측하기 위해 loss_val, game_length 등의 몇가지 변수를 추가로 정의해주었습니다.

Learning Start

: 학습 세션을 열고 학습 Loop를 실행합니다.

```
with tf.Session() as sess:
    if os.path.isfile(checkpoint_path + ".index"):
        saver.restore(sess, checkpoint_path)
    else:
        init.run()
        copy_online_to_target.run()
    while True:
        step = global_step.eval()
        if step >= n_steps:
            break
        iteration += 1
        print("\r반복 {}\t훈련 스텝 {}/{} ({:.1f})%\t손실 {:5f}\t평균 최대-Q {:5f}   ".format(
            iteration, step, n_steps, step * 100 / n_steps, loss_val, mean_max_q), end="")
        if done: # 게임이 종료되면 다시 시작합니다
            obs = env.reset()
            for skip in range(skip_start): # 게임 시작 부분은 스킵합니다
                obs, reward, done, info = env.step(0)
            state = preprocess_observation(obs)
```

〉학습이 매우 오래 걸리기 때문에 중간에 중단하더라도 이어서 학습할 수 있도록 학습을 수행하면서 학습 진행 상황을 checkpoint_path로 지정되어 있는 *"./my_dqn.ckpt"* 파일을 저장하도록 하였습니다. 파일에 학습 진행 상황을 저장하면서 세션 맨 처음에 체크포인트 파일이 있으면 해당 파일에 저장된 모델을 복원하고, 파일이 없으면 초기화하고 시작하도록 설정하였습니다.

〉While문이 시작하면 iteration으로 전체 학습 스텝 횟수를 카운트하고, step으로는 학습이 시작한 후부터의 전체 학습 스텝 횟수를 카운트합니다. 즉, 앞서 말한 체크포인트 파일을 통해 이전 학습을 불러와도 global_step 변수를 통해 step 변수를 불러올

수 있습니다.

〉 *obs = env.reset()* 코드를 통해 게임을 리셋하고 그 이후 코드를 통해 학습에 필요하지 않은 부분을 건너뛸 수 있도록 구현하였습니다.

```
# Actual DQN이 해야할 Action을 평가합니다
q_values = online_q_values.eval(feed_dict={X_state: [state]})
action = epsilon_greedy(q_values, step)

# Actual DQN으로 게임을 플레이합니다.
obs, reward, done, info = env.step(action)
next_state = preprocess_observation(obs)

# Replay buffer에 기록합니다
replay_memory.append((state, action, reward, next_state, 1.0 - done))
state = next_state
```

〉 Actual DQN이 어떤 Action을 할지 평가하고, 게임을 플레이하여 그 경험을 Replay buffer에 저장합니다.

```
# 트래킹을 위해 통계값을 계산합니다
total_max_q += q_values.max()
game_length += 1
if done:
    mean_max_q = total_max_q / game_length
    total_max_q = 0.0
    game_length = 0

if iteration < training_start or iteration % training_interval != 0:
    continue
```

〉 미리 설정한 하이퍼 파라미터인 training_start와 training_interval을 통해 일정한 간격으로 Actual DQN이 학습 step에 들어가도록 구현했습니다.

```
# Target DQN을 사용해서 메모리에서 샘플링하여 Target Q value를 구합니다
X_state_val, X_action_val, rewards, X_next_state_val, continues = (
    sample_memories(batch_size))
next_q_values = target_q_values.eval(
    feed_dict={X_state: X_next_state_val})
max_next_q_values = np.max(next_q_values, axis=1, keepdims=True)
y_val = rewards + continues * discount_factor * max_next_q_values

# Actual DQN을 학습시킵니다
_, loss_val = sess.run([training_op, loss], feed_dict={
    X_state: X_state_val, X_action: X_action_val, y: y_val})

# Actual DQN을 Target DQN으로 정해둔 일정 간격마다 복사합니다
if step % copy_steps == 0:
    copy_online_to_target.run()
```

〉 먼저 메모리에서 배치를 샘플링하고 Target DQN에 각 경험 데이터의 다음 state에 대한 가능한 모든 Action의 Q value을 estimate하도록 합니다. 그리고 나서 $y_i = r + \gamma max_{a'} Q(s', a'; \theta_i)$ 식을 사용하여 각각의 State-Action Pair에 대한 Target Q value를 담은 y_val을 계산하도록 구현하였습니다. 이 코드를 구현할 때 주의하여 구현했던 점은 max_next_q_values 벡터와 continues 벡터를 곱해 게임이 터미널 스테이트에 도달한 경우 Future reward을 0으로 만들어야 한다는 점이었습니다.

〉 위에서 계산한 y_val을 이용하여 Q value를 estimate하는 Actual DQN의 정확도를 향상시키기 위해 학습을 진행합니다.

〉 정해둔 하이퍼 파라미터인 step에서 copy_steps을 나눈 나머지 값이 0일 경우, copy_online_to_target을 호출하여 Actual DQN을 Target DQN으로 복사합니다.

```
# 정해둔 일정 간격으로 저장합니다
if step % save_steps == 0:
    saver.save(sess, checkpoint_path)
```

〉 정해둔 하이퍼 파라미터인 step에서 save_steps을 나눈 나머지 값이 0일 경우, saver의 save() 함수를 호출하여 진행상황을 checkpoint_path에 저장합니다.

Testing

: 학습 진행 도중이나 학습 종료 후, 진행 정도 및 결과를 테스트합니다.

```
frames = []
n_max_steps = 10000

with tf.Session() as sess:
    saver.restore(sess, checkpoint_path)

    obs = env.reset()
    for step in range(n_max_steps):
        state = preprocess_observation(obs)

        # Actual DQN이 해야할 Action을 평가합니다
        q_values = online_q_values.eval(feed_dict={X_state: [state]})
        action = np.argmax(q_values)

        # Actual DQN이 게임을 플레이합니다
        obs, reward, done, info = env.step(action)

        img = env.render(mode="rgb_array")
        frames.append(img)

        if done:
            break
```

〉 앞서 언급하였듯이 진행 상황을 checkpoint_path에 저장하여 언제든 학습을 중단하고 테스트해볼 수 있도록 구현하였습니다. 또한, 테스트 결과를 frames 배열에 저장하여 이후 코드에 있는 비디오 변환 코드를 사용할 수 있도록 하였습니다.

Changing to HTML5 Video

: 학습 결과를 비디오로 변환합니다.

```
def update_scene(num, frames, patch):
    plt.close()
    patch.set_data(frames[num])
    return patch,

def plot_animation(frames, figsize=(5,6), repeat=False, interval=40):
    fig = plt.figure(figsize=figsize)
    patch = plt.imshow(frames[0])
    plt.axis('off')
    return animation.FuncAnimation(fig, update_scene, fargs=(frames, patch),
                                   frames=len(frames), repeat=repeat, interval=interval)
video = plot_animation(frames, figsize=(5,6))
HTML(video.to_html5_video())  # HTML5 동영상으로 만들어 줍니다
```

〉 이전 코드에서 frames 배열에 저장한 학습 결과를 불러와 matplotlib의 animation 모듈을 사용하여 애니메이션 비디오로 변환할 수 있도록 구현하였습니다.

학습 종료 후, 최종 결과

총 4,000,000번의 훈련 스텝을 실행했으며 학습 시간은 대략 96시간 정도 소요되었습니다. 학습 환경으로 Docker를 통해 Intel i7 CPU 용량의 50~60% 정도를 사용했습니다. 학습 진행 상황 테스트는 1/2 정도가 진행된 2,339,451번째 스텝에서 한 번, 4,000,000번의 훈련 스텝을 모두 실행한 학습이 종료되었을 때 한 번, 총 두 번 진행하여 학습 정도를 확인하고 이외의 구간에서는 육안으로 확인하였습니다.

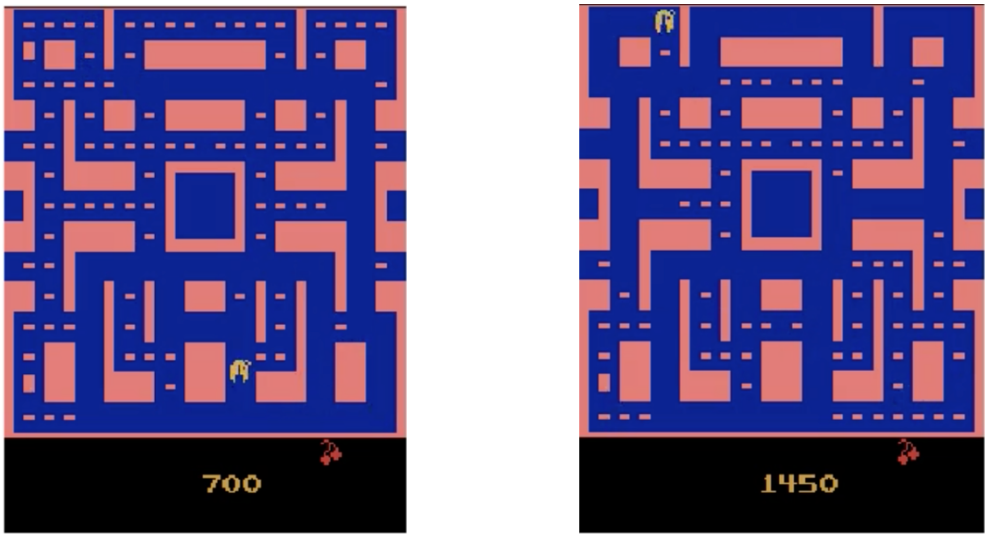
구분	결과
58.5%	손실 : 5.784453, 평균 최대-Q : 200.299690
100%	손실 : 8.244065, 평균 최대-Q : 191.150198

3.1 테스트를 진행한 스텝의 손실과 평균 최대-Q

처음 진행률 0~20% 구간에서는 Q value가 음수로 나타나기도 하고 양수로 나타나더라도 한자리 수가 대부분이었습니다. 학습이 크게 이루어진 구간은 진행률 20~50% 정도의 구간이었습니다. 이 구간에서부터 학습이 진행될 수록 Q value가 음수로 계산되는 스텝이 존재하지 않았고 양수인 Q value가 큰 폭으로 커지는 구간이었습니다. 50~100% 구간에서는 Q value가 185~200 정도로 고정되어 구간 내에서 늘어나고 줄어드는 것을 반복하기만 할 뿐 200 이상으로는 더 이상 늘어나지 않았습니다. 실제 학습 중 테스트를 진행한 스텝의 캡쳐본(3.2)과 테스트 동영상의 캡처본(3.3)을 아래에 첨부합니다.

```
INFO:tensorflow:Restoring parameters from ./my_dqn.ckpt
반복 9371308    훈련 스텝 2339451/4000000 (58.5)%      손실 5.784453   평균 최대-Q 200.299690
INFO:tensorflow:Restoring parameters from ./my_dqn.ckpt
반복 16015304   훈련 스텝 3999999/4000000 (100.0)%     손실 8.244065   평균 최대-Q 191.150198
```

3.2 테스트를 진행한 스텝의 캡처본 (위 : 58.5%, 아래 : 100%)



3.3 테스트를 진행한 동영상 캡처본 (왼쪽 : 58.5%, 오른쪽 : 100%)

결론 ; 느낀 점

서론에서 이야기하였듯이 저는 알파고, 머신 러닝 등 평소 인공지능에 관심은 많았지만 수학에 자신이 없어서 시도해보지 못했었습니다. 하지만, 이번 프로젝트로 처음 강화학습을 접하고 프로그래밍해볼 기회를 가지게 되었으며, TV로 여러 번 접했던 팩맨 게임을 다루면서 추억을 회상하게 되는 기회가 되어서 매우 뜻깊었습니다.

학습을 진행하기 전, 처음에는 100%가 될 때까지 Q value가 상승하여 어느정도는 최적인 Policy를 찾을 수 있을 거라고 생각했습니다. 학습을 진행하면서 50%의 진행률까지는 점점 늘어나는 Q value를 보면서 혼자 나름대로 '잘 프로그래밍했구나.'라고 생각했습니다. 그런데 50% 이후에 Q value가 늘어나지 않는 것을 보고 하이퍼 파라미터를 잘못 설정했을 수도 있겠다는 생각이 들었습니다. Q value가 더이상 늘어나지 않고 일정 구간 안에서 늘었다가 줄었다만을 반복했기 때문입니다. 테스트 동영상으로 확인하여도 분명 처음 학습을 시작할 때보다는 나아진 것이 눈에 보이긴 했지만 만족스러운 결과는 아니었습니다. 2,000,000번의 스텝에서는 어느정도 학습이 잘 되었는데 이후 2,000,000번의 스텝에서는 어떠한 이유에서 학습이 잘 안됐는지 파악하기가 어려웠고, 이 점이 매우 아쉬웠습니다.

개인적으로 프로젝트를 진행하면서 환경 소스코드 분석, 알고리즘 프로그래밍 등 힘들고 이해하는 것이 오래 걸리는 것들이 매우 많았습니다. 하지만, 그중에도 가장 힘들었던 건 아무래도 역시 학습시키는 시간이었습니다. Docker를 통해 Intel i7 CPU의 50~60% 정도를 사용하여 전체 학습의 1%에 해당하는 40,000번 학습시키는데에 대략 1시간 정도 걸렸고 총 96 시간 정도가 소요되었는데 DQN 알고리즘을 선택하여 학습시키는데 정말 여러 번 후회했습니다. 며칠 동안 컴퓨터를 혹사시키며 매일 틈틈이 확인하여 학습의 진행 상황과 Q value를 확인했어야 했습니다. DQN에 비해 금방 학습이 되는 Dynamic Programming, TD Learning 등 알고리즘을 바꾸고 싶다는 생각을 정말 여러 번 했지만, 하나의 프로젝트일 뿐이더라도 포기하지 않고 DQN 프로그래밍을 포기하지 않고 끝까지 해내게 되어 매우 기쁘고, 찾지 못한 학습이 잘 되지 않았던 점들을 해결해 제대로 사람처럼 혹은 사람보다 더 잘 플레이하는 팩맨의 DQN 알고리즘을 한 번 구현해보고 싶습니다.