“Atari Games with Double-DQN”

1. Environment

- “Breakout-v0”

Maximize your score in the Atari 2600 game Breakout. In this environment, the observation is an RGB image of the screen, which is an array of shape (210, 160, 3) Each action is repeatedly performed for a duration of *k* frames, where *k* is uniformly sampled from {2,3,4}.

수업시간에 배운 Atari Game(PacMan)과 비슷한 환경을 가지는 Breakout-v0을 선택하여 DQN의 improvement인 Double-DQN을 적용해보고자 하였다.

- Action

```
env.action_space
Discrete(4)
```

4가지 종류의 Discrete action을 가진다.
[**0: No-op** (아무것도 안 함), **1: Fire** (발사), **2: Left** (왼쪽 이동), **3: Right** (오른쪽 이동)]
4개 중 하나의 Action이 선택되면 *k*={2,3,4}에서 k값이 sampling되어서 해당 Action을 *k*-frame동안 지속하게 된다.

- State

```
env.observation_space
Box(0, 255, (210, 160, 3), uint8)
```

각각의 screen captured화면을 state로 사용한다. RGB channel을 가지는 210*160크기의 화면이 observation이 되지만, 실제 학습시에는 grayscale로 변환해서 사용한다.

- Game

화면 상단에는 무지개색의 벽돌들이 위치해있고, 아래쪽에 agent가 움직일 발판이 놓여있다. 발판은 횡방향(Left, Right)으로만 이동할 수 있고, 공을 튕겨내서 벽돌을 맞추면 해당 벽돌을 없앨 수 있다. 하나의 벽돌을 없앨 때마다 score를 획득할 수 있고(위쪽에 있는 블록일수록 높은 점수), Agent가 공을 떨어뜨리지 않고 최대한 많은 score를 획득하도록 만드는 것이 목표이다. (Maximizing)

2. Algorithm

- Double-DQN (DQN+Double Q-learning)

Double-DQN은 수업시간에도 배웠듯이, 기존 DQN의 단점을 해결하기 위해 제안된 DQN Improvent 알고리즘이다.

기존의 Q-learning 알고리즘은 Q-value를 overestimate한다는 단점이 있었고, 이 특징을 그대로 가지고 있는 DQN은 종종 좋지 않은 성능을 보였다.

$$Y_t^{DQN} \equiv R_{t+1} + \gamma \max_a Q(S_{t+1}, a; \theta_t^-)$$

DQN은 S_{t+1}에서 취할 수 있는 actions중에서 expected value가 가장 큰 action을 선택하는 maximum estimation을 사용한다. 하지만 DQN은 실제 Q-value가 아니라 대략적으로 추정하여 approximate하는 방식이기 때문에 Non-negative bias가 발생한다.

$$Q^{approx}(s,a) = R_{t+1} + \gamma max_a Q^{approx}(s',a)$$

$$Q^{target}(s,a) = R_{t+1} + \gamma max_a Q^{target}(s',a)$$

Q^{approx}를 근사값, Q^{target}을 실제값이라고 하면 각각 추정치와 실제값을 통해 위와 같은 식으로 나타낼 수 있고, Q^{approx}는 Q^{target} value에서 정규분포를 따르는 어떠한 Noise(Y)를 더한 값이라는 가정을 따른다.

하지만 위의 두 식을 통해 두 값의 차이 Z를 구해보면,

$$Z_s = \gamma(max_a Q^{approx}(s',a) - max_a Q^{target}(s',a))$$

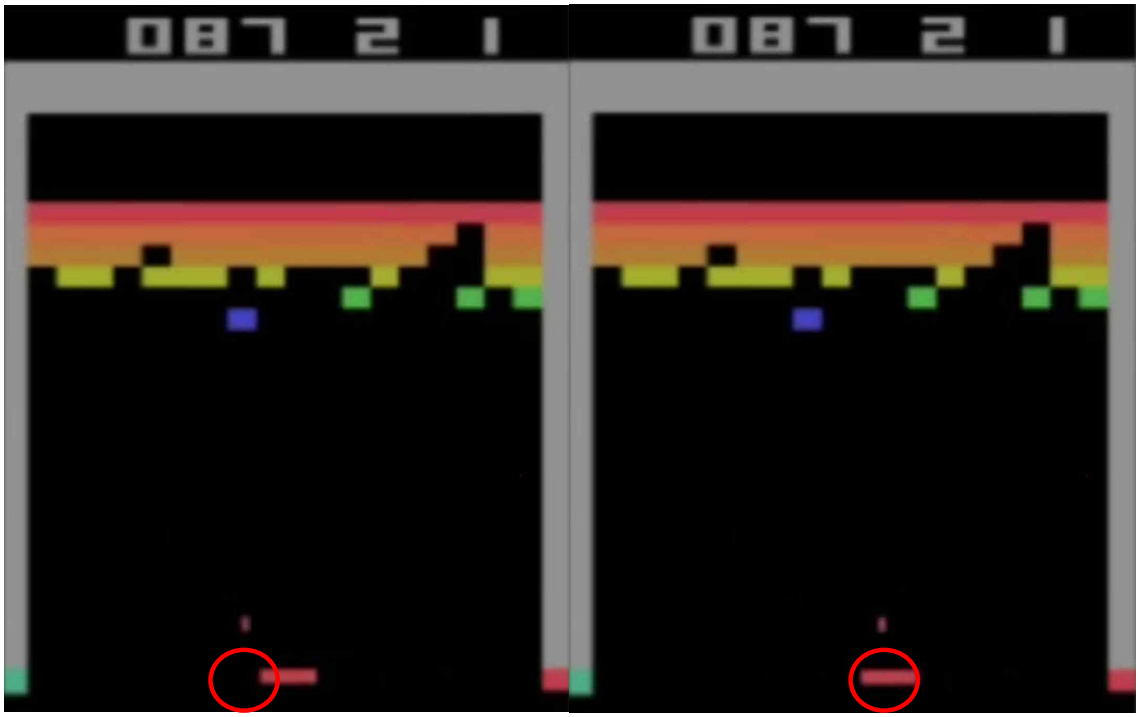
항상 Max값을 선택하는 연산으로 인해 두 값의 차이는 평균=0의 정규분포를 따르지 않게 되고, 어떤 Action a의 실제 action-state value보다 더 높은 값으로 판단해 버리는 Overestimation이 발생한다.

Double DQN은 이 Overestimation 문제점을 단일한 estimator로 인해서 나오는 것이라고 규정해서, double Q-learning에 쓰였던 아이디어를 활용하여(DQN+double Q-learning) 제안된 알고리즘이다.

$$Y_t^{DDQN} \equiv R_{t+1} + \gamma Q(S_{t+1}, \arg\max_a Q(S_{t+1}, a; \theta_t); \theta_t')$$

따라서 Double-DQN에서는 위와 같이 두 개의 분리된 estimator를 사용하여 Target을 재정의하게 된다. 즉, Selection과 Evaluation이 서로 다른 네트워크를 통해 이루어진다.

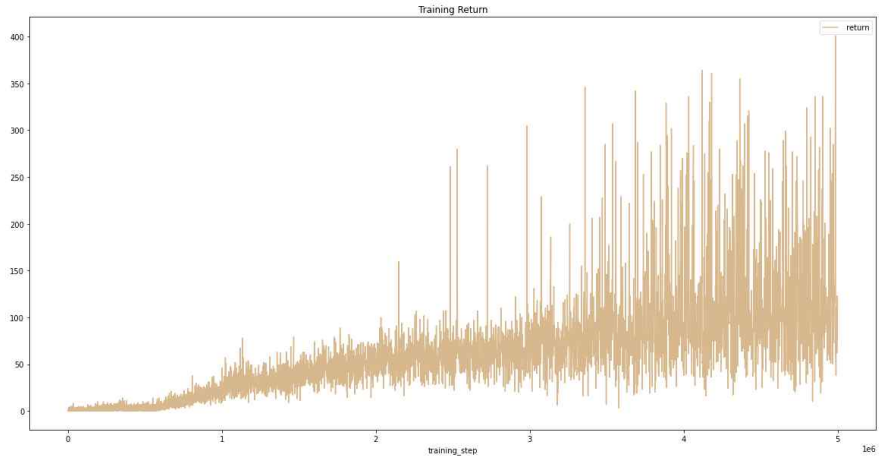
3. Results



agent가 좌우로 이동해가며 위치를 조절해서 위쪽에서 날아오는 공을 튕겨냄

튕겨진 공이 연두색 블록을 맞춰서 블록이 사라지고 score가 1점(연두색=1점) 오름. 공이 허공에서 돌아다니는 동안에도 agent는 가만히만 있지 않고 계속해서 action을 선택하는 수행을 한다. (영상자료는 따로 첨부하였습니다)

- Training Return

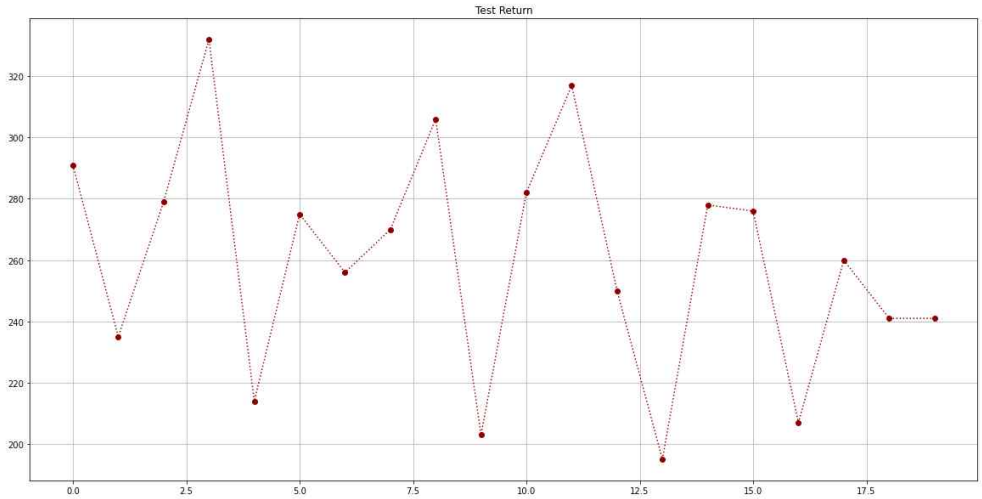


총 500,000epoch의 학습을 수행했고, 학습시간은 colab(GPU)환경에서 275분정도 소요되었다. 100~200학습주기마다 csv파일로 로그를 출력하여 Reward를 확인할 수 있도록 했는데, 출력된 점수를 시각화해보니 위와 같은 그래프를 볼 수 있었다.

초반에는 거의 0에 수렴하는 score만 나왔었는데, render()를 통해 agent의 움직임을 직접 확인해보아도 공을 단 한번도 튕겨내지 못하고 reset되는 것을 볼 수 있었다.

학습이 진행될수록 평균적으로 얻는 score가 상승하여 평균return이 100점 이상으로 넘어가기도 했는데, episode마다의 score 편차가 매우 크고 불안정한 것을 볼 수 있었다.

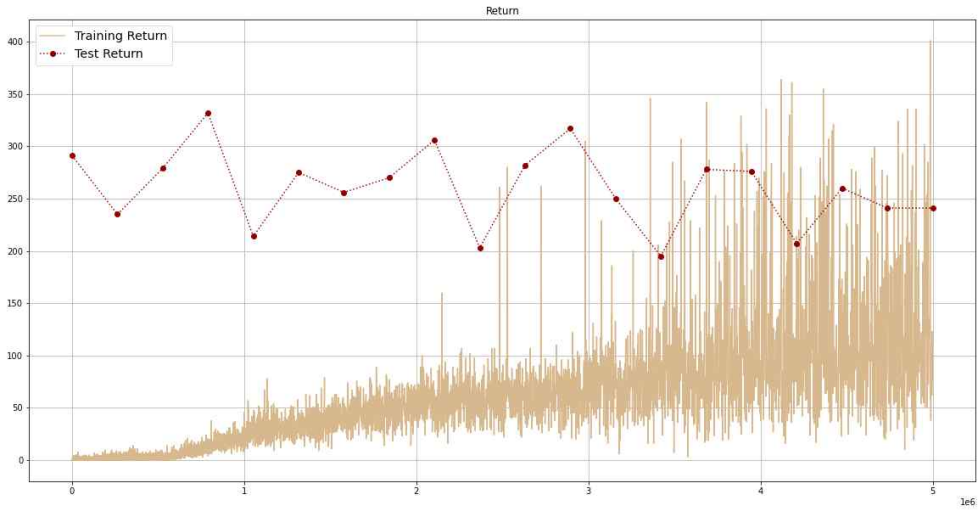
- Test return



학습이 종료된 후에 직접 render()를 통해 확인하면서 20번정도 게임을 플레이해보았는데, 평균적으로 260점을 획득했으나 Training과정에서 보았던것처럼 점수편차가 매우 큰 것을 볼 수 있었다.

```
Y_test = [291.0, 235.0, 279.0, 332.0, 214.0, 275.0, 256.0, 270.0, 306.0, 203.0,
          282.0, 317.0, 250.0, 195.0, 278.0, 276.0, 207.0, 260.0, 241.0, 241.0]
```

- Training+Test return



두 그래프를 같이 시각화하여 보았을 때, 학습량이 많고 그 중 낮은 score들이 많이 나와서 평균 score는 전체적으로 낮아보이는 경향이 있었으나 실제 Test시에는 꽤 좋은 score들을 얻었다.

매우 불안정하게 나타난 결과값이 알고리즘의 문제인가 싶어서 여러 자료들을 찾아보았는데, 학습량의 부족으로 나타난 문제인것 같아보였다(기본적으로 몇천만회 이상 epoch사용...). Colab에서도 사용 가능한 리소스에 한계가 있어서 더 이상 학습은 진행해보지 못했으나, Testing에서 게임 진행과정을 직접 눈으로 확인해보면 어느정도 잘 학습돼서 agent가 게임을 잘 플레이하는 것을 확인할 수 있었다.

4. Main Source Code

[기본 뼈대인 DQN network나 함수들은 강의자료와 Github등을 참고해서 구현하고, 비슷하게 사용가능한 환경에서 새롭게 적용(Pytorch)]

- class Atari_Wrapper

environment를 초기화하고 각 screenshot을 preprocessing해서 state로 활용하는 함수들 포함

```
def preprocess_observation(self, ob):

    ob = cv2.cvtColor(ob[self.frame_cutout_h[0]:self.frame_cutout_h[1],self.frame_cutout_w[0]:self.frame_cutout_w[1]], cv2.COLOR_BGR2GRAY)
    ob = cv2.resize(ob, dsize=self.dsize)

    return ob
```

- RGB(3차원)의 channel을 갖는 screenshot을 입력으로 사용하기 전에 grayscale로 변환해주고 지정된 크기로 crop해준다.

```
def step_frame_stack(self, frames):

    num_frames = len(frames)

    if num_frames == self.k:
        self.frame_stack = np.stack(frames)
    elif num_frames > self.k:
        self.frame_stack = np.array(frames[-k::])
    else:
        self.frame_stack[0: self.k - num_frames] = self.frame_stack[num_frames::]
        self.frame_stack[self.k - num_frames::] = np.array(frames)
```

- 하나의 action이 선택되면 한 frame에만 유효한 게 아니라 k={2,3,4}중 하나가 선택되어 k만큼 해당 action을 지속하기 때문에 k-frame만큼 유지시키는 함수를 사용한다.

```
def step(self, action):

    reward = 0
    done = False

    frames = []
    for i in range(self.k):

        ob, r, d, info = self.env.step(action)
        ob = self.preprocess_observation(ob)
        frames.append(ob)

        reward += r

        if d:
            done = True
            break

    self.step_frame_stack(frames)

    self.Return += reward
    if done:
        info["return"] = self.Return

    if reward > 0:
        reward = 1
    elif reward == 0:
        reward = 0
    else:
        reward = -1

    return self.frame_stack, reward, done, info
```

- 위에 정의한 함수를 통해 k-frame만큼 유지키시며 step하도록 만들고, reward가 정수형이 아니기 때문에 정수로 잘라낸다. 양수이면 1, 0이면 0, 음수이면 –1의 reward를 받고 전체 Return에 더해준다.

- class DQN

DQN network는 기존 DQN과 유사하게 사용하였다.

```
def __init__(self, in_channels, num_actions):
    super().__init__()

    network = [
        torch.nn.Conv2d(in_channels, 32, kernel_size=8, stride=4, padding=0),
        nn.ReLU(),
        torch.nn.Conv2d(32, 64, kernel_size=4, stride=2, padding=0),
        nn.ReLU(),
        torch.nn.Conv2d(64, 64, kernel_size=3, stride=1, padding=0),
        nn.ReLU(),
        nn.Flatten(),
        nn.Linear(64*7*7,512),
        nn.ReLU(),
        nn.Linear(512, num_actions)
    ]

    self.network = nn.Sequential(*network)

def forward(self, x):
    actions = self.network(x)
    return actions
```

- Input data가 게임 screenshot(이미지)이기 때문에 Convolution Layer를 통해 input state를 해석한다. Convolution과 Activation Layer(ReLU)를 3layer씩 쌓는데, 이미지의 Spatial information이 중요하기 때문에 Padding과 Pooling은 사용하지 않는다.

Convolution Layer를 통해 특징을 추출하고 나면, 벡터를 Flatten시킨 후에 FC layer를 통과시켜 Output layer의 neuron의 개수만큼의 결과를 추출한다. 이 때 output의 개수는 가능한 전체 action의 개수와 동일해서, 해당 state에서 각 action의 Q-value값을 표현하게 된다.

- class Agent

Agent의 행동을 선택하고 epsilon값 조절

```
def e_greedy(self, x):

    actions = self.forward(x)

    greedy = torch.rand(1)
    if self.eps < greedy:
        return torch.argmax(actions)
    else:
        return (torch.rand(1) * self.num_actions).type('torch.LongTensor')[0]

def set_epsilon(self, epsilon):
    self.eps = epsilon
```

- Agent가 k-frame만큼 수행할 행동을 선택할 때 사용하는 epsilon-greedy를 구현한다. epsilon의 확률이 선택되면 random으로 선택하여 exploration을 수행하고, 1-epsilon의 확률이 선택되면 가장 가치가 높은 action을 선택한다. epsilon의 값은 set_epsilon을 통해 점점 감소하도록 구현한다.

- class Experience_Relay

DQN-based 알고리즘은 게임이 time series으로 진행되면서 각 state마다 correlation이 높기 때문에 Experience_Replay(buffer)를 사용해서 해결한다.

```
def __init__(self, capacity):
    self.capacity = capacity
    self.memory = []
    self.position = 0

def insert(self, transitions):

    for i in range(len(transitions)):
        if len(self.memory) < self.capacity:
            self.memory.append(None)
        self.memory[self.position] = transitions[i]
        self.position = (self.position + 1) % self.capacity

def get(self, batch_size):
    indexes = (np.random.rand(batch_size) * (len(self.memory)-1)).astype(int)
    return [self.memory[i] for i in indexes]
```

- 빈 memory를 하나 생성한 뒤에, 다음 state로의 transition이 일어날때마다 insert()를 통해 replay buffer에 저장해준다. replay buffer는 용량이 정해져있고, 새로운 값이 들어오면 기존의 값이 삭제되는 Queue에 가깝기 때문에 지정된 capacity를 넘지 않는지 매번 확인할 수 있도록 한다.

한번에 한 state만 보는 게 아니라 batch_size만큼씩 볼 수 있도록 했는데, correlation을 없애기 위해서는 buffer내에서 순서대로 추출하면 안되고 random으로 선택할 수 있도록 해야한다. get()을 통해 random index를 추출할 수 있도록 한다.

- class Env_Runner

```
def run(self, steps):

    obs = []
    actions = []
    rewards = []
    dones = []

    for step in range(steps):

        self.ob = torch.tensor(self.ob)
        action = self.agent.e_greedy(
            self.ob.to(device).to(dtype).unsqueeze(0) / 255)
        action = action.detach().cpu().numpy()

        obs.append(self.ob)
        actions.append(action)

        self.ob, r, done, info = self.env.step(action)

        if done:
            self.ob = self.env.reset()
            if "return" in info:
                self.logger.log(f'{self.total_steps+step},{info["return"]}')

        rewards.append(r)
        dones.append(done)

    self.total_steps += steps

    return obs, actions, rewards, dones
```

- 함수들을 종합해서 실제 transition이 일어나도록 하는 함수. Transition 정보를 return한다.

- Training

hyperparameter들을 정의하고(replay buffer의 크기, learning rate, discount factor, batch size, epsilon값 등등) breakout-v0환경을 env로 불러온 후에 위의 class들과 parameter들을 활용하여 학습을 진행한다.

```
Qs = agent(torch.cat([obs, next_obs]))
obs_Q, next_obs_Q = torch.split(Qs, minibatch_size ,dim=0)

obs_Q = obs_Q[range(minibatch_size), actions]

next_obs_Q_max = torch.max(next_obs_Q,1)[1].detach()
target_Q = target_agent(next_obs)[range(minibatch_size), next_obs_Q_max].detach()

target = rewards + gamma * target_Q * dones
```

***** Double DQN ***** 에서 추가된 코드이다.

기존 DQN에서는 single estimator가 사용되었는데, Double-DQN에서는 obs_Q와 next_obs_Q 두 개의 function으로 분리한다.

$$Y_t^{DDQN} \equiv R_{t+1} + \gamma Q(S_{t+1}, \arg\max_a Q(S_{t+1}, a; \theta_t); \theta_t')$$

위에서 봤던 DDQN target식처럼 분리된 Q-value function을 사용하고, 분리된 next_obs_Q를 maximum하는 action을 선택하여 기존 DQN식에 대입한다. 이후 target 식을 구하는 연산은 동일하다.

```
num_model_updates += 1

if num_model_updates%target_net_update == 0:
    target_agent.load_state_dict(agent.state_dict())
```

target network는 매 반복마다 update하지 않고 미리 지정한 target_net_update만큼의 update가 이루어지면 기존(selection에 사용하는) network로부터 복사해와서 update한다.

- Test (Display)

```
display = Display(visible=False, size=(2800, 1800))
display.start()

torch.save(agent,"agent.pt")
agent = torch.load("./drive/MyDrive/DDQN/agent_trained.pt", map_location=device)

raw_env = gym.make(env_name)
env = Atari_Wrapper(raw_env, env_name, num_stacked_frames)

steps = 5000
ob = env.reset()
agent.set_epsilon(0.025)
agent.eval()
imgs = []
for step in range(steps):

    action = agent.e_greedy(torch.tensor(ob, dtype=dtype).unsqueeze(0) / 255)
    action = action.detach().cpu().numpy()

    plt.imshow(env.render(mode='rgb_array'))
    ipythondisplay.clear_output(wait=True)
    ipythondisplay.display(plt.gcf())

    ob, _, done, info = env.step(action)

    time.sleep(0.005)
    if done:
        ob = env.reset()
        print(info)

    imgs.append(ob)

env.close()
```

학습이 끝난 이후에는 pt파일을 저장해두고 (agent_trained.pt) Test할 때 불러와서 게임을 실행한다. google colab에서 학습과 테스트를 진행했는데, 기존에 사용하던 Display방식이 잘 동작하지 않아서 matplotlib과 pythondisplay를 사용하여 출력하였다.