


# **Team 4**

## **Reinforcement Learning**



# Index

1. Model Definition
2. Deep Q-networks

# Hyperparameter

---

```
import tensorflow as tf
import gym

import numpy as np
import random as ran
import datetime
import matplotlib.pyplot as plt

from collections import deque
from skimage.transform import resize
from skimage.color import rgb2gray
```

```
env = gym.make('BreakoutDeterministic-v4')
```

Gym에 여러 게임이 저장되어있음  
그 중 Breakout게임 불러오기

# Hyperparameter

# 하이퍼 파라미터

MINIBATCH\_SIZE = 32

HISTORY\_SIZE = 4

TRAIN\_START = 50000

FINAL\_EXPLORATION = 0.1

TARGET\_UPDATE = 10000

MEMORY\_SIZE = 400000

EXPLORATION = 1000000

START\_EXPLORATION = 1.

INPUT = env.observation\_space.shape

OUTPUT = env.action\_space.n

HEIGHT = 84

WIDTH = 84

LEARNING\_RATE = 0.00025

DISCOUNT = 0.99

EPSILON = 0.01

MOMENTUM = 0.95

## \*Mini batch

전체 학습 데이터를 batch size만큼 등분

-> 각 배치 셋을 순차적으로 수행

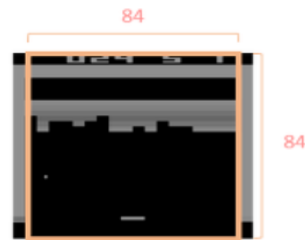
-> 전체 데이터를 최대한 반영하는 방법

## \*Target network

1만 번 마다 한 번씩 main network로부터 업데이트

## \*84x84

게임에 입력되는 그림 크기



## \*Discount

Q-learning update에 사용된 감마

$$Q(s, a) = r + \gamma \max_{a'} (Q(s', a'))$$

0.99: 미래에 대한 보상 < 현재에 대한 보상

# Hyperparameter

```
# 하이퍼 파라미터
MINIBATCH_SIZE = 32
HISTORY_SIZE = 4
TRAIN_START = 50000
FINAL_EXPLORATION = 0.1
TARGET_UPDATE = 10000
MEMORY_SIZE = 400000
EXPLORATION = 1000000
START_EXPLORATION = 1.
INPUT = env.observation_space.shape
OUTPUT = env.action_space.n
HEIGHT = 84
WIDTH = 84
LEARNING_RATE = 0.00025
DISCOUNT = 0.99
EPSILON = 0.01
MOMENTUM = 0.95
```

```
print(env.action_space)
print(env.observation_space)
```

Discrete(4)  
Box(210, 160, 3)

**\*Discrete(4)**  
No-op, fire, left, right

**\*Box(210, 160, 3)**  
210x160 크기

**Extended Data Table 1 | List of hyperparameters and their values**

Hyperparameter	Value	Description
minibatch size	32	Number of training cases over which each stochastic gradient descent (SGD) update is computed.
replay memory size	1000000	SGD updates are sampled from this number of most recent frames.
agent history length	4	The number of most recent frames experienced by the agent that are given as input to the Q network.
target network update frequency	10000	The frequency (measured in the number of parameter updates) with which the target network is updated (this corresponds to the parameter C from Algorithm 1).
discount factor	0.99	Discount factor gamma used in the Q-learning update.
action repeat	4	Repeat each action selected by the agent this many times. Using a value of 4 results in the agent seeing only every 4th input frame.
update frequency	4	The number of actions selected by the agent between successive SGD updates. Using a value of 4 results in the agent selecting 4 actions between each pair of successive updates.
learning rate	0.00025	The learning rate used by RMSProp.
gradient momentum	0.95	Gradient momentum used by RMSProp.
squared gradient momentum	0.95	Squared gradient (denominator) momentum used by RMSProp.
min squared gradient	0.01	Constant added to the squared gradient in the denominator of the RMSProp update.
initial exploration	1	Initial value of $\epsilon$ in $\epsilon$ -greedy exploration.
final exploration	0.1	Final value of $\epsilon$ in $\epsilon$ -greedy exploration.
final exploration frame	1000000	The number of frames over which the initial value of $\epsilon$ is linearly annealed to its final value.
replay start size	50000	A uniform random policy is run for this number of frames before learning starts and the resulting experience is used to populate the replay memory.
no-op max	30	Maximum number of “do nothing” actions to be performed by the agent at the start of an episode.

The values of all the hyperparameters were selected by performing an informal search on the games Pong, Breakout, Seaquest, Space Invaders and Beam Rider. We did not perform a systematic grid search owing to the high computational cost, although it is conceivable that even better results could be obtained by systematically tuning the hyperparameter values.

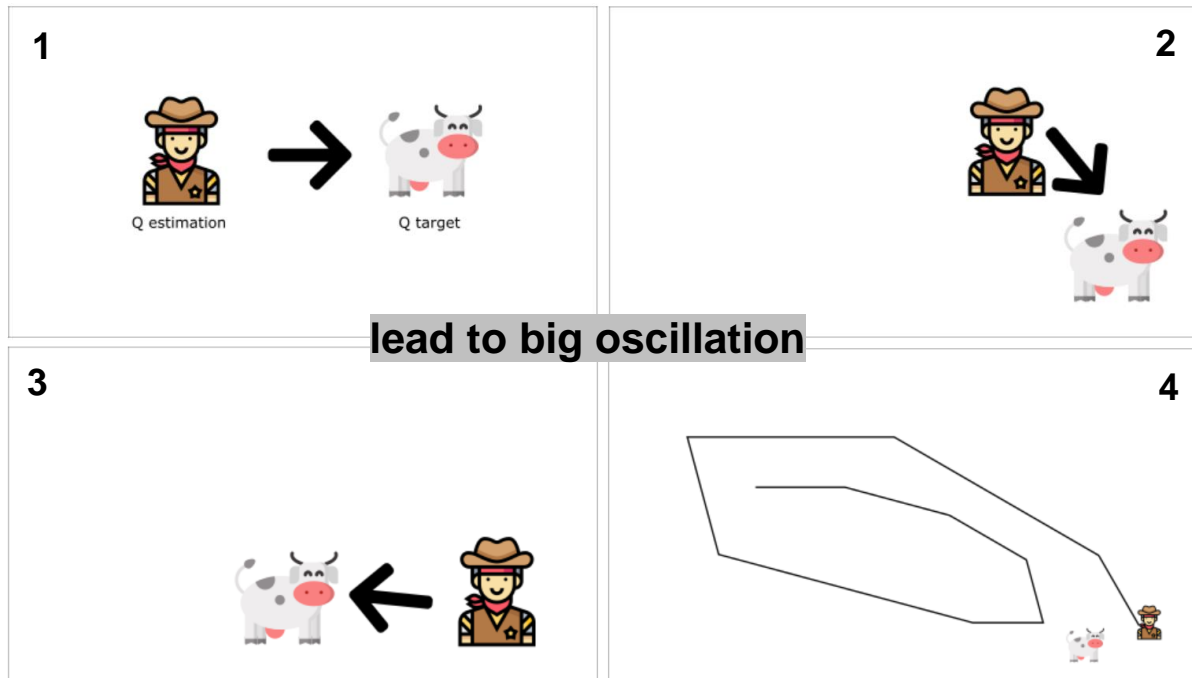
# Separate Target Network

$$\Delta w = \alpha \left[ \underbrace{(R + \gamma \max_a \hat{Q}(s', a, w))}_{\text{Maximum possible Qvalue for the next\_state (= Q\_target)}} - \underbrace{\hat{Q}(s, a, w)}_{\text{Current predicted Q-val}} \right] \underbrace{\nabla_w \hat{Q}(s, a, w)}_{\text{Gradient of our current predicted Q-value}}$$

Change in weights      learning rate      TD Error

- 목표 : **TD Error(Loss)**를 줄이는 것. 즉, target에 가까워지는 것
- 그러나 Q\_target과 Q\_value를 구할 때 같은 **weight**을 쓰기 때문에 문제가 발생!
  - target 의 움직임
- “Chasing a moving target”

# Separate Target Network



## <Fixed Q-targets>

- 학습의 불안정함을 줄이기 위해 같은 구조이지만 다른 parameter를 가진 target network를 만든다.
- Target network parameters는 매 C step마다 Q network parameters로 업데이트 된다.



# Separate Target Network

---

$$y_i = r + \gamma \max_{a'} \hat{Q}(s', a'; \theta_i^-).$$

$$L_i(\theta_i) = \mathbb{E}_{(s,a,r,s') \sim U(D)} \left[ \left( r + \gamma \max_{a'} \hat{Q}(s', a'; \theta_i^-) - Q(s, a; \theta_i) \right)^2 \right]$$

- C번의 iteration 동안 Q-learning update할 때 target이 움직이는 현상을 방지할 수 있다.
- 결과적으로, 좀 더 안정되게 훈련시킬 수 있다.

# Separate Target Network

---

$$y_i = r + \gamma \max_{a'} \hat{Q}(s', a'; \theta_i^-).$$

$$L_i(\theta_i) = \mathbb{E}_{(s,a,r,s') \sim U(D)} \left[ \left( r + \gamma \max_{a'} \hat{Q}(s', a'; \theta_i^-) - Q(s, a; \theta_i) \right)^2 \right]$$

➤ **But 단점**

- Q network에서 변경된 값은 바로 적용되지 않음
- target network 상당한 시간 이후에 업데이트
- 학습속도 저하

# Target Network vs Main network(Q-network)

```
def get_copy_var_ops(*, dest_scope_name="target", src_scope_name="main"):
    """타겟네트워크에 메인네트워크의 Weight값을 복사.
```

Args:

dest\_scope\_name="target"(DQN): 'target'이라는 이름을 가진 객체를 가져옴  
src\_scope\_name="main"(DQN): 'main'이라는 이름을 가진 객체를 가져옴

Returns:

list: main의 trainable한 값들이 target의 값으로 복사된 값

"""

```
op_holder = []
```

```
src_vars = tf.get_collection(
    tf.GraphKeys.TRAINABLE_VARIABLES, scope=src_scope_name)
```

```
dest_vars = tf.get_collection(
    tf.GraphKeys.TRAINABLE_VARIABLES, scope=dest_scope_name)
```

```
for src_var, dest_var in zip(src_vars, dest_vars):
    op_holder.append(dest_var.assign(src_var.value()))
```

```
return op_holder
```

타겟네트워크에다가  
메인네트워크를 복사하는 과정

# Separate Target Network

Extended Data Table 3 | The effects of replay and separating the target Q-network

Game	With replay, with target Q	With replay, without target Q	Without replay, with target Q	Without replay, without target Q
Breakout	316.8	240.7	10.2	3.2
Enduro	1006.3	831.4	141.9	29.1
River Raid	7446.6	4102.8	2867.7	1453.0
Seaquest	2894.4	822.6	1003.0	275.8
Space Invaders	1088.9	826.3	373.2	302.0

DQN agents were trained for 10 million frames using standard hyperparameters for all possible combinations of turning replay on or off, using or not using a separate target Q-network, and three different learning rates. Each agent was evaluated every 250,000 training frames for 135,000 validation frames and the highest average episode score is reported. Note that these evaluation episodes were not truncated at 5 min leading to higher scores on Enduro than the ones reported in Extended Data Table 2. Note also that the number of training frames was shorter (10 million frames) as compared to the main results presented in Extended Data Table 2 (50 million frames).

# 라이프 게임



\* 라이프게임이란?  
잘못 타깃 했을 때마다  
라이프가 하나씩 감소하는 경우.

\* 이 경우는 벽돌을 튕긴다음에 막대에 부딪히지 않고 땅속으로 추락하는 경우에 라이프가 하나씩 줄어들음. 처음에 라이프 5개가 주어지고 0이 되면 게임 오버.

# 라이프 게임

```
def get_game_type(count, l, no_life_game, start_live):
    """라이프가 있는 게임인지 판별

    Args:
        count(int): 에피소드 시작 후 첫 프레임인지 확인하기 위한 arg
        l(dict): 라이프 값들이 저장되어있는 dict ex) l['ale.lives']
        no_life_game(bool): 라이프가 있는 게임일 경우, bool 값을 반환해줌
        start_live(int): 라이프가 있는 경우 라이프값을 초기화 하기 위한 arg

    Returns:
        list:
            no_life_game(bool): 라이프가 없는 게임이면 True, 있으면 False
            start_live(int): 라이프가 있는 게임이면 초기화된 라이프
    """
    if count == 1:
        start_live = l['ale.lives']
        # 시작 라이프가 0일 경우, 라이프 없는 게임
        if start_live == 0:
            no_life_game = True
        else:
            no_life_game = False
    return [no_life_game, start_live]

# 라이프가 있는 게임.라이프를 5개 주고 다 쓰면 게임 종료
```

\* 라이프게임이란?

실패할때마다 라이프가 하나씩 감소하는 경우.

\* 시작할때 라이프가 = 0

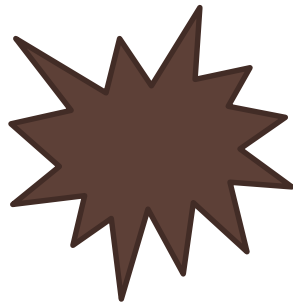
-> no\_life\_game

# Terminal 처리

---

- 벽돌깨기는 5개의 라이프가 있는 **라이프 게임**이다.

**바닥에 떨어질 때마다 라이프가 감소.**  
남은 라이프 수가 0이 되면 게임 끝.



- **Terminal state** : 각 episode가 끝나는 상태  
새로운 episode는 이전 episode의 승패와 상관없이 독립적으로 다시 시작.

# Terminal 처리

- 목숨이 줄어들거나, negative reward를 받았을 경우 terminal 처리

```
def get_terminal(start_live, I, reward, no_life_game, ter):
```

\***start\_live**(int) : 라이프가 있는 게임일 경우, 현재 라이프 수

\***I**(dict) : 다음 상태에서 라이프가 줄었는지 확인하기 위한 다음 frame의 라이프 정보

```
if no_life_game:
    # 목숨이 없는 게임일 경우 Terminal 처리
    if reward < 0:
        ter = True
else:
    # 목숨 있는 게임일 경우 Terminal 처리
    if start_live > I['ale.lives']:
        ter = True
        start_live = I['ale.lives']

return [ter, start_live] -> update!
```



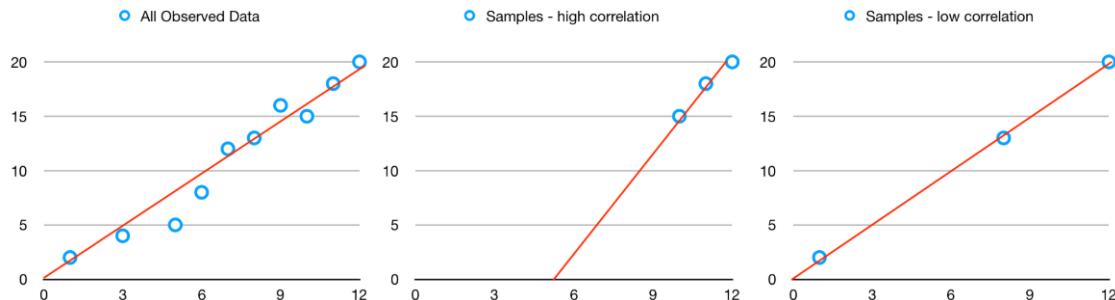
# Experience Replay

## ➤ Correlation between samples

강화학습에서의 학습데이터는 시간의 흐름에 따라 순차적으로 수집.  
순차적인 데이터는 근접한 것들끼리 **높은 상관관계**를 띄게 된다.



**제대로 된  
학습 어려움**



**Solution : experience replay!**

# Experience Replay

## ➤ Replay Memory

강화학습에서 학습의 재료가 되는 중요한 sample을 저장해두는 저장소

## ➤ Experience Replay

게임을 진행하면서 모든 과정이 들어오는 즉시 훈련시키지 않고, 일단 메모리에 저장해 두었다가 나중에 일정 수의 샘플을 랜덤으로 꺼내서 학습시키는 방식

① Agent의 경험(experience)을 각 time-step마다 튜플 형태로 **메모리 D에 저장**해 둔다.

$$e_t = (s_t, a_t, r_t, s_{t+1}) \quad \mathcal{D} = e_1, \dots, e_N$$

② D에서 uniform random sampling을 통해 **minibatch**를 구성해서 학습한다.

*\*data set은 무한히 저장할 수 없으므로 N으로 고정하고, FIFO(first in first out)방식으로 저장한다.*

# Experience Replay

---

## ➤ Experience Replay의 이점

- ① minibatch가 순차적인 데이터로 구성되지 않으므로 입력 데이터 사이의 상관관계를 줄일 수 있다.
- ② 각각의 experience가 업데이트할 때 재사용되기 때문에 experience를 한 번만 사용하고 버리는 기존 방법보다 훨씬 **data efficiency**하다.

# Experience Replay

```
def train_minibatch(mainDQN, targetDQN, minibatch):
    '''미니배치로 가져온 sample데이터로 메인네트워크 학습

    Args:
        mainDQN(object): 메인 네트워크
        targetDQN(object): 타겟 네트워크
        minibatch: replay_memory에서 MINIBATCH 개수만큼 랜덤 sampling 해온 값

    Note:
        replay_memory에서 꺼내온 값으로 메인 네트워크를 학습
        ...

    s_stack = []
    a_stack = []
    r_stack = []
    s1_stack = []
    d_stack = []

    for s_r, a_r, r_r, d_r in minibatch:
        s_stack.append(s_r[:, :, :4])
        a_stack.append(a_r)
        r_stack.append(r_r)
        s1_stack.append(s_r[:, :, 1:])
        d_stack.append(d_r)
```

```
# True, False 값을 1과 0으로 변환
```

```
d_stack = np.array(d_stack) + 0
```

```
Q1 = targetDQN.get_q(np.array(s1_stack))
```

```
y = r_stack + (1 - d_stack) * DISCOUNT * np.max(Q1, axis=1)
```

**\*d(done)**

게임이 끝나면 True=1

그렇지 않으면 False=0

-> 업데이트된 Q값으로 메인네트워크 학습

# Deep Q-networks

## ➤ CNN(Convolutional Neural Network)

```
def build_network(self):
    with tf.variable_scope(self.name):
        self.X = tf.placeholder('float', [None, self.height, self.width, self.history_size])
        self.Y = tf.placeholder('float', [None])
        self.a = tf.placeholder('int64', [None])

        f1 = tf.get_variable("f1", shape=[8, 8, 4, 32], initializer=tf.contrib.layers.xavier_initializer_conv2d())
        f2 = tf.get_variable("f2", shape=[4, 4, 32, 64], initializer=tf.contrib.layers.xavier_initializer_conv2d())
        f3 = tf.get_variable("f3", shape=[3, 3, 64, 64], initializer=tf.contrib.layers.xavier_initializer_conv2d())
        w1 = tf.get_variable("w1", shape=[7 * 7 * 64, 512], initializer=tf.contrib.layers.xavier_initializer())
        w2 = tf.get_variable("w2", shape=[512, OUTPUT], initializer=tf.contrib.layers.xavier_initializer())

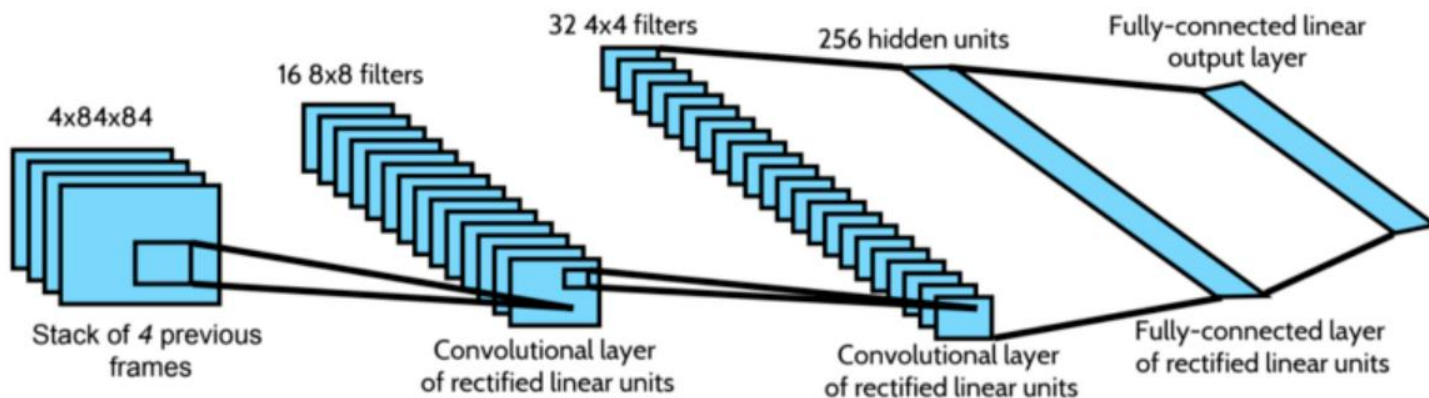
        c1 = tf.nn.relu(tf.nn.conv2d(self.X, f1, strides=[1, 4, 4, 1], padding="VALID"))
        c2 = tf.nn.relu(tf.nn.conv2d(c1, f2, strides=[1, 2, 2, 1], padding="VALID"))
        c3 = tf.nn.relu(tf.nn.conv2d(c2, f3, strides=[1, 1, 1, 1], padding="VALID"))

        l1 = tf.reshape(c3, [-1, w1.get_shape().as_list()[0]])
        l2 = tf.nn.relu(tf.matmul(l1, w1))

        self.Q_pre = tf.matmul(l2, w2)
```

# Deep Q-networks

## ➤ CNN(Convolutional Neural Network)



<Fig 3. DQN model architecture>

**Thank you**