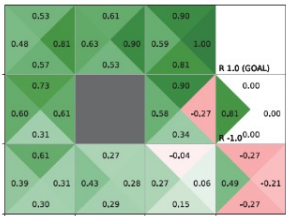
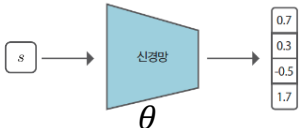


# Deep Q-Network (DQN)

Prof. Tae-Hyoung Park  
Dept. of Intelligent Systems & Robotics, CBNU

# Q-learning vs Q-Network vs DQN

cf. Policy based

항목	Q-Learning	Q-Network	DQN
방법론	Value based	←	←
Q 함수 표현 $Q(s, a)$ 가치	Table 	Neural networks 	←
Q 함수 학습 (update)	<u>Bellman optimality eq.</u> $Q'(S_t, A_t) = Q(S_t, A_t) + \alpha \{ R_t + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t) \}$	Error backpropagation (Gradient descent) - 신경망 출력값 = $Q(S_t, A_t)$ - 신경망 목표값 = $\gamma \max_a Q(S_{t+1}, a) + R_t$	←
		신경망 학습의 효율 및 안정성 문제	- Experience Replay 도입 - Target Network 도입
Action 선택 (policy)	$\epsilon$ -greedy	←	←
적용 환경	- Discrete state space - State space 크기가 작음	- Continuous state space 도 가능 - State space 크기가 큰 경우 도 가능	←

# OpenAI Gym

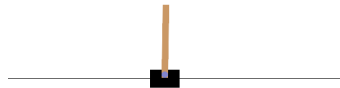
- Basics

- Open source Python library for developing and comparing reinforcement algorithms
- Public beta version, 2016.04
- Documentation: <https://www.gymnasium.dev/>

- Environments

```
pip install gym[classic_control]
```

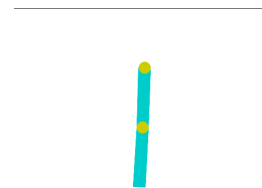
- Classic Control



Cart pole



Mountain car



Acrobot



Pendulum

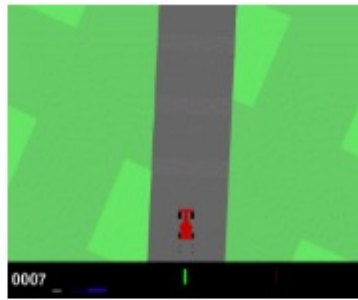
# OpenAI Gym

- Environments

- Box2D `pip install gym[box2d]`



Bipedal Walker



Car Racing

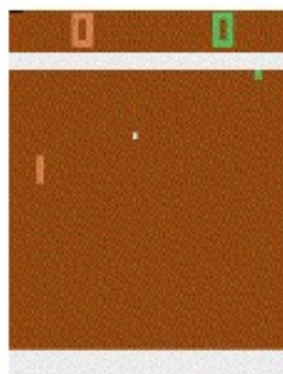


Lunar Lander

- Atari



Breakout



Pong

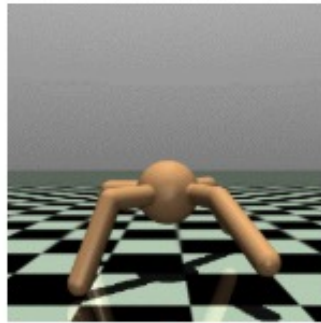


Space Invaders

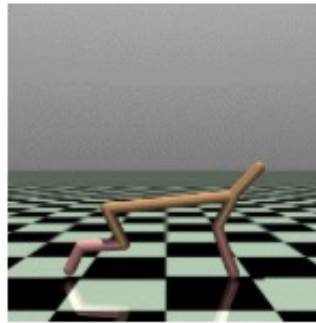
# OpenAI Gym

- Environments

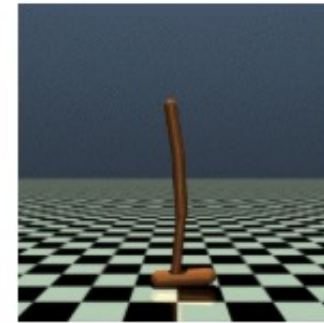
- MuJoCo `pip install gym[mujoco]`



Ant



Half Cheetah

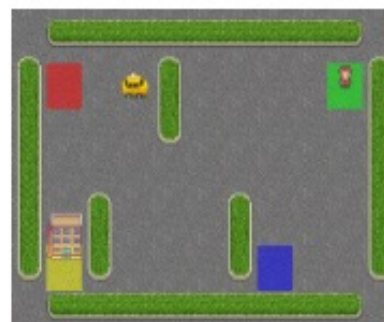


Hopper

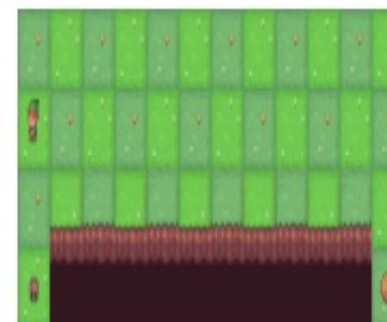
- Toy Text



Blackjack



Taxi



Cliff Walking

# Cart Pole

- Environment

```
import gym  
  
env = gym.make('CartPole-v0', render_mode='human')
```

- State (observation)

- Cart position:  $-4.8 \sim 4.8$
    - Cart velocity:  $-\infty \sim \infty$
    - Pole angle:  $-24(\text{deg}) \sim 24(\text{deg})$
    - Pole angular velocity:  $-\infty \sim \infty$

- Action

- 0: Push cart to the left
    - 1: Push cart to the right

- Rewards

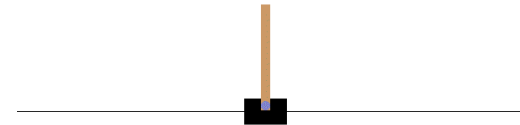
- +1 for every step (goal is to keep the pole upright)

- Starting state

- Uniformly random value in  $(-0.05, 0.05)$

- Episode ends if any one of the following occurs

- Pole angle is greater than  $\pm 12$  deg ( $\pm 0.209$  rad)
    - Cart position is greater than  $\pm 2.4$  (center of cart reaches the edge of display)
    - Episode length is greater than 500 (200 for v0)



# Cart Pole

- Env.reset()

```
state = env.reset()[0] # 상태 초기화
print('상태:', state)

action_space = env.action_space
print('행동의 차원 수:', action_space)
```

position, velocity, angle, angular velocity

```
상태: [ 0.03454657 -0.01361909 -0.02143636  0.02152179]
행동의 차원 수: Discrete(2)
```

- Env.step(action)

```
action = 0 # 혹은 1
next_state, reward, terminated, truncated, info = env.step(action)
print(next_state)
```

```
[ 0.03454657 -0.01361909 -0.02143636  0.02152179]
```

- next\_state: 다음 상태
- reward: 보상
- terminated: 목표 상태 도달 여부
- truncated: MDP 범위 밖의 종료 조건 충족 여부(시간 초과 등)
- info: 추가 정보

각도 초과, 위치 벗어남

action 500회 (v0: 200회 초과)

# Cart Pole

- Random Agent
  - 무작위로 행동 (action) 하는 agent

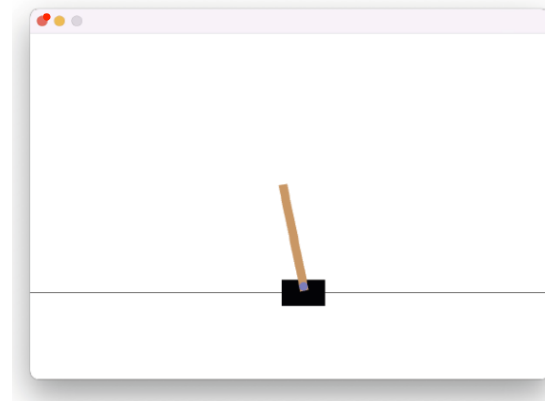
```
import numpy as np
import gym

env = gym.make('CartPole-v0', render_mode='human')
state = env.reset()[0]
done = False

while not done: # 에피소드가 끝날 때까지 반복
    env.render() # 진행 과정 시각화
    action = np.random.choice([0, 1]) # 행동 선택(무작위)
    next_state, reward, terminated, truncated, info = env.step(action)
    done = terminated | truncated # 둘 중 하나만 True면 에피소드 종료
env.close()
```

C.position C.velocity P.angle P.velocity

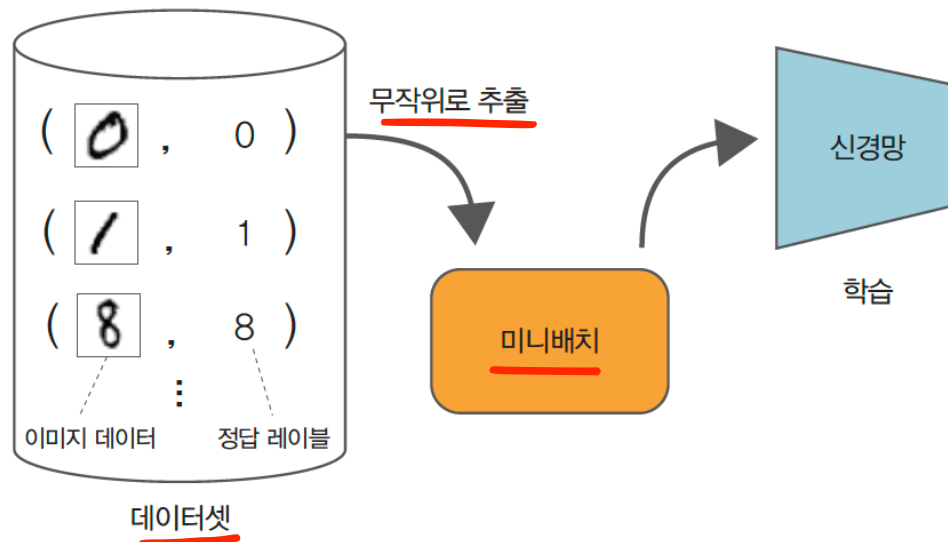
```
[ 0.01510794 -0.15462452  0.00201281  0.24891098]
[ 0.01201545 -0.34977517  0.00699103  0.5422281 ]
[ 0.00501994 -0.54499465  0.01783559  0.8371056 ]
[-0.00587995 -0.35012078  0.0345777  0.55008465]
[-0.01288236 -0.5457109  0.04557939  0.8534583 ]
[-0.02379658 -0.74142355  0.06264856  1.160118 ]
[-0.03862505 -0.54717124  0.08585092  0.8877178 ]
[-0.04956848 -0.7433469  0.10360527  1.2061068 ]
[-0.06443541 -0.9396437  0.1277274  1.5293785 ]
[-0.08322829 -1.1360537  0.15831497  1.8590435 ]
[-0.10594936 -0.94298404  0.19549584  1.619411 ]
[-0.12480905 -1.1397984  0.22788407  1.9661194 ]
```





# Experience Replay

- 신경망 학습 방법: Training Data Set
  - Supervised learning
    - Training data set 에서 일부 데이터 무작위 추출  $\Rightarrow$  mini-batch
    - Mini-batch 를 이용해 신경망 학습



# Experience Replay

- Experience Replay

- ~~Q Learning~~ **Q-network**

- Agent 가 환경 속에서 어떤 행동을 취할 때 마다 경험데이터  $E_t$  를 생성하고

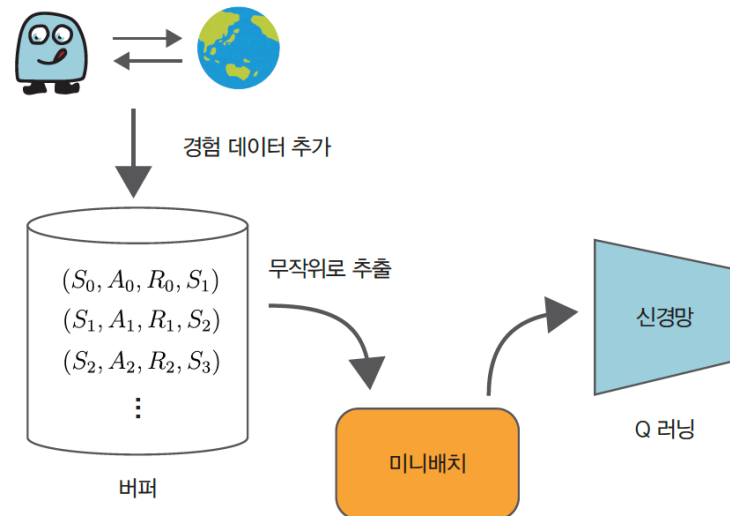
$$E_t = (S_t, A_t, R_t, S_{t+1})$$

buffer 에 저장

- Q 함수를 갱신할 때 buffer 로 부터 일부 경험 데이터 무작위 추출  $\Rightarrow$  mini-batch

- Mini-batch 를 사용하여 신경망 학습  
 $\Rightarrow$  Experience Replay (경험 재생)

$E_t$  와  $E_{t+1}$  사이의 상관관계 높음  
 $\rightarrow$  상관관계를 약화시켜 편향이 작은 학습데이터 생성



# 실습

## • Experience Replay 구현

### 실습 #1 Replay\_buffer.py

```
from collections import deque
import random
import numpy as np
import gym

class ReplayBuffer:
    def __init__(self, buffer_size, batch_size):
        self.buffer = deque(maxlen=buffer_size)
        self.batch_size = batch_size

    def add(self, state, action, reward, next_state, done):
        data = (state, action, reward, next_state, done)
        self.buffer.append(data)

    def __len__(self):
        return len(self.buffer)

    def get_batch(self):
        data = random.sample(self.buffer, self.batch_size)

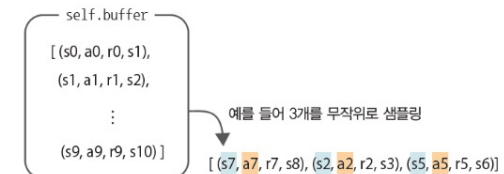
        state = np.stack([x[0] for x in data])
        action = np.array([x[1] for x in data])
        reward = np.array([x[2] for x in data])
        next_state = np.stack([x[3] for x in data])
        done = np.array([x[4] for x in data]).astype(np.int32)
        return state, action, reward, next_state, done
```

**buffer\_size:** buffer 최대 크기  
**batch\_size:** mini-batch 크기  
**deque:** 먼저 삽입된 데이터부터 삭제 (선입선출)

버퍼에 경험데이터 추가  
 $E_t = (S_t, A_t, R_t, S_{t+1})$

Buffer 의 크기

Buffer 에 담긴 데이터에서 mini-batch 생성



state는 형상이 (4, )인 np.ndarray  
np.stack([s7, s2, s5])로  
형상이 (3, 4)인 np.ndarray로

action은 int 타입  
np.stack([a7, a2, a5])로  
형상이 (3)인 np.ndarray로

# 실습

- Experience Replay 구현

실습 #1 Replay\_buffer.py

```
env = gym.make('CartPole-v0', render_mode='human')
replay_buffer = ReplayBuffer(buffer_size=10000, batch_size=32)

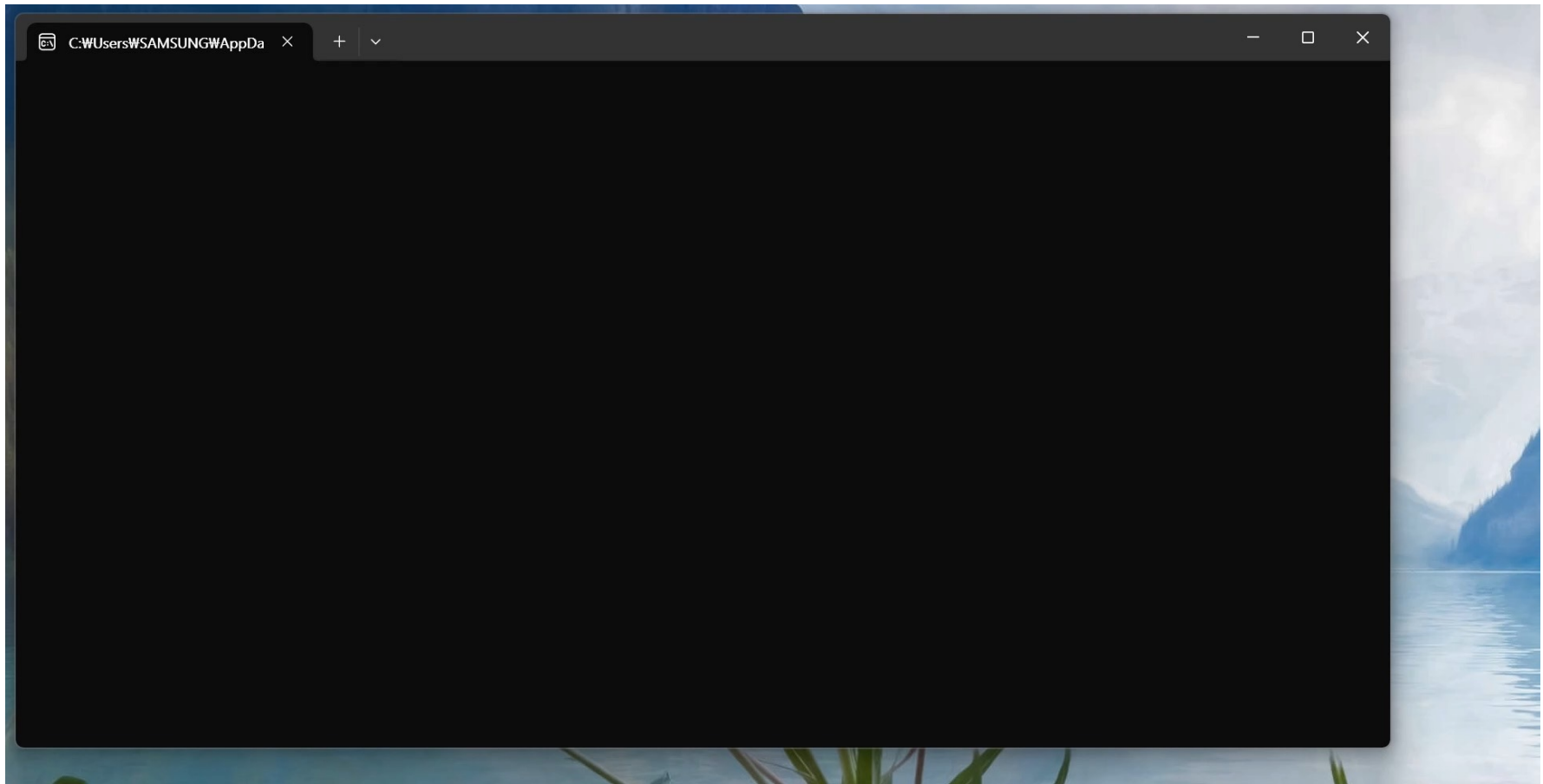
for episode in range(10): # 에피소드 10회 수행
    state = env.reset()[0]
    done = False

    while not done:
        action = 0 # 항상 0번째 행동만 수행
        next_state, reward, terminated, truncated, info = env.step(action) # 경험 데이터 획득
        done = terminated | truncated

        replay_buffer.add(state, action, reward, next_state, done) # 버퍼에 추가
        state = next_state

# 경험 데이터 버퍼로부터 미니배치 생성
state, action, reward, next_state, done = replay_buffer.get_batch()
print(state.shape) # (32, 4)
print(action.shape) # (32,)
print(reward.shape) # (32,)
print(next_state.shape) # (32, 4)
print(done.shape) # (32,)
```

# 실습

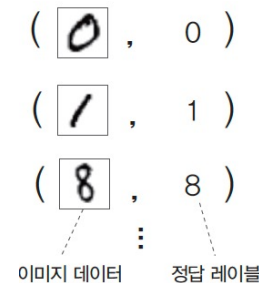


# Target Network

- 신경망 학습 방법 : 정답 Labelling

- Supervised learning

- 학습 데이터에 정답 레이블 부여
- 정답 레이블은 영구적으로 고정됨



- Q Network

- $Q(S_t, A_t)$  의 목표값 =  $R_t + \gamma \max_a Q(S_{t+1}, a)$  : 정답 레이블
- $Q$  값이 갱신될 때마다 정답 레이블이 변경됨
  - 신경망 학습 어려움
  - ⇒ Target Network (목표 신경망) 을 사용하여 목표값을 고정

- ❖ Target Network

- 원본 신경망 (qnet) 과 구조가 같은 신경망 (qnet\_target)
- qnet 은 일반적인 Q Network 으로 가중치 갱신
- qnet\_target 은 주기적으로 qnet 의 가중치와 동기화시키고, 그 외에는 가중치 고정
- 이후 qnet\_target 을 이용하여 목표값 계산
- ⇒ 정답 레이블이 계속 바뀌는 것을 억제하여 신경망 학습 안정화

# Target Network

- Target Network 구현

```
import copy
from dezero import Model
from dezero import optimizers
import dezero.functions as F
import dezero.layers as L

class QNet(Model): # ❶ 신경망 클래스
    def __init__(self, action_size):
        super().__init__()
        self.l1 = L.Linear(128)
        self.l2 = L.Linear(128)
        self.l3 = L.Linear(action_size)

    def forward(self, x):
        x = F.relu(self.l1(x))
        x = F.relu(self.l2(x))
        x = self.l3(x)
        return x

class DQNAgent: # 에이전트 클래스
    def __init__(self):
        self.gamma = 0.98
        self.lr = 0.0005
        self.epsilon = 0.1
        self.buffer_size = 10000 # 경험 재생 버퍼 크기
```

```
self.batch_size = 32 # 미니배치 크기
self.action_size = 2

self.replay_buffer = ReplayBuffer(self.buffer_size, self.batch_size)
self.qnet = QNet(self.action_size) # ❷ 원본 신경망
self.qnet_target = QNet(self.action_size) # ❸ 목표 신경망
self.optimizer = optimizers.Adam(self.lr)
self.optimizer.setup(self.qnet) # ❹ 옵티마이저에 qnet 등록

# 가중치 업데이트는 qnet 만

def sync_qnet(self): # ❺ 두 신경망 동기화
    self.qnet_target = copy.deepcopy(self.qnet) # 깊은복사

def get_action(self, state):
    if np.random.rand() < self.epsilon:
        return np.random.choice(self.action_size)
    else:
        state = state[np.newaxis, :] # 배치 처리용 차원 추가
        qs = self.qnet(state)
        return qs.data.argmax()
```

# Target Network

- Target Network 구현

```
class DQNAgent:
    ...

    def update(self, state, action, reward, next_state, done):
        # ❶ 경험 재생 버퍼에 경험 데이터 추가
        self.replay_buffer.add(state, action, reward, next_state, done)
        if len(self.replay_buffer) < self.batch_size:
            return # 데이터가 미니배치 크기만큼 쌓이지 않았다면 여기서 끝

        # ❷ 미니배치 크기 이상이 쌓이면 미니배치 생성
        state, action, reward, next_state, done = self.replay_buffer.get_batch()

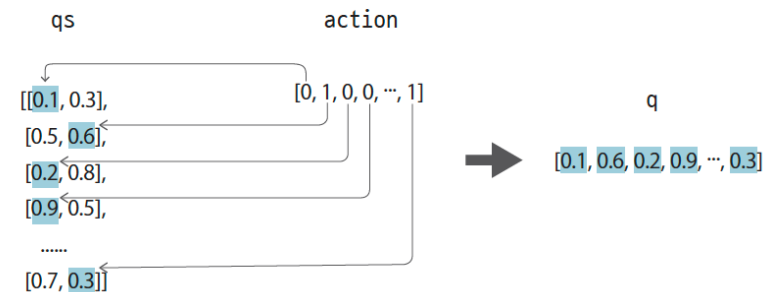
        qs = self.qnet(state) # ❸
        q = qs[np.arange(self.batch_size), action] # ❹

        next_qs = self.qnet_target(next_state) # ❺
        next_q = next_qs.max(axis=1)
        next_q.unchain()
        target = reward + (1 - done) * self.gamma * next_q # ❻

        loss = F.mean_squared_error(q, target)

        self.qnet.cleargrads()
        loss.backward()
        self.optimizer.update()
```

batch\_size=32  
state: (32, 4), action: (32, )  
qs : (32, 2), q: (32, )





# Target Network

- DQN 학습

```
episodes = 300      # 에피소드 수
sync_interval = 20  # 신경망 동기화 주기(20번째 에피소드마다 동기화)
env = gym.make('CartPole-v0', render_mode='rgb_array')
agent = DQNAgent()
reward_history = [] # 에피소드별 보상 기록

for episode in range(episodes):
    state = env.reset()[0]
    done = False
    total_reward = 0

    while not done:
        action = agent.get_action(state)  $\epsilon$  greedy
        next_state, reward, terminated, truncated, info = env.step(action)
        done = terminated | truncated

        agent.update(state, action, reward, next_state, done)
        state = next_state
        total_reward += reward

    if episode % sync_interval == 0:
        agent.sync_qnet() # Target network 동기화

    reward_history.append(total_reward)
```

그림 8-8 <카트 폴>에서 에피소드별 보상 총합의 추이

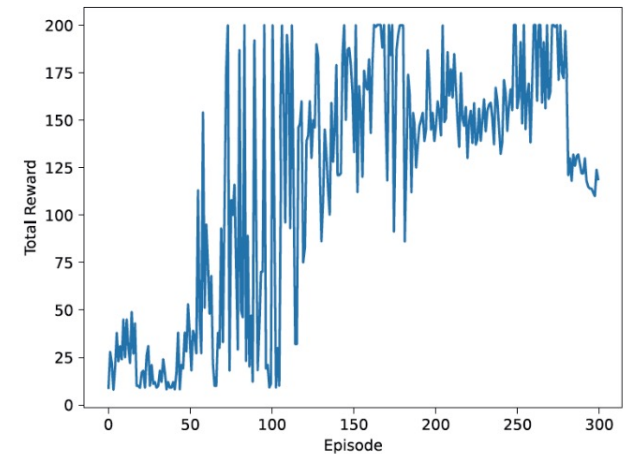
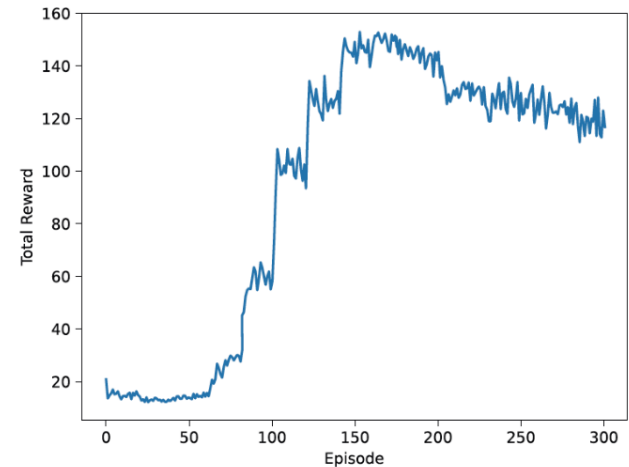


그림 8-9 100번 실험 후 평균한 결과



# DQN

- DQN 실행

```
agent.epsilon = 0 # 탐욕 정책(무작위로 행동할 확률  $\epsilon$ 을 0으로 설정)
state = env.reset()[0]
done = False
total_reward = 0

while not done:
    action = agent.get_action(state)
    next_state, reward, terminated, truncated, info = env.step(action)
    done = terminated | truncated
    state = next_state
    total_reward += reward
    env.render()
print('Total Reward:', total_reward)
```

Total Reward: 116

# 실습

## 실습 #2 dqn2.py

```
import copy
from collections import deque
import random
import matplotlib.pyplot as plt
import numpy as np
import gym
from dezero import Model
from dezero import optimizers
import dezero.functions as F
import dezero.layers as L

class ReplayBuffer:
    def __init__(self, buffer_size, batch_size):
        self.buffer = deque(maxlen=buffer_size)
        self.batch_size = batch_size

    def add(self, state, action, reward, next_state, done):
        data = (state, action, reward, next_state, done)
        self.buffer.append(data)

    def __len__(self):
        return len(self.buffer)

    def get_batch(self):
        data = random.sample(self.buffer, self.batch_size)

        state = np.stack([x[0] for x in data])
        action = np.array([x[1] for x in data])
        reward = np.array([x[2] for x in data])
        next_state = np.stack([x[3] for x in data])
        done = np.array([x[4] for x in data]).astype(np.int32)
        return state, action, reward, next_state, done
```

```
class QNet(Model): # 신경망 클래스
    def __init__(self, action_size):
        super().__init__()
        self.l1 = L.Linear(128)
        self.l2 = L.Linear(128)
        self.l3 = L.Linear(action_size)

    def forward(self, x):
        x = F.relu(self.l1(x))
        x = F.relu(self.l2(x))
        x = self.l3(x)
        return x

class DQNAgent: # 에이전트 클래스
    def __init__(self):
        self.gamma = 0.98
        self.lr = 0.0005
        self.epsilon = 0.1
        self.buffer_size = 10000 # 경험 재생 버퍼 크기
        self.batch_size = 32 # 미니배치 크기
        self.action_size = 2

        self.replay_buffer = ReplayBuffer(self.buffer_size, self.batch_size)
        self.qnet = QNet(self.action_size) # 원본 신경망
        self.qnet_target = QNet(self.action_size) # 목표 신경망
        self.optimizer = optimizers.Adam(self.lr)
        self.optimizer.setup(self.qnet) # 옵티마이저에 qnet 등록

    def get_action(self, state):
        if np.random.rand() < self.epsilon:
            return np.random.choice(self.action_size)
        else:
            state = state[np.newaxis, :] # 배치 처리용 차원 추가
            qs = self.qnet(state)
            return qs.data.argmax()
```

```

def update(self, state, action, reward, next_state, done):
    # 경험 재생 버퍼에 경험 데이터 추가
    self.replay_buffer.add(state, action, reward, next_state, done)
    if len(self.replay_buffer) < self.batch_size:
        return # 데이터가 미니배치 크기만큼 쌓이지 않았다면 여기서 끝

    # 미니배치 크기 이상이 쌓이면 미니배치 생성
    state, action, reward, next_state, done = self.replay_buffer.get_batch()
    qs = self.qnet(state)
    q = qs[np.arange(self.batch_size), action]

    next_qs = self.qnet_target(next_state)
    next_q = next_qs.max(axis=1)
    next_q.unchain()
    target = reward + (1 - done) * self.gamma * next_q

    loss = F.mean_squared_error(q, target)

    self.qnet.cleargrads()
    loss.backward()
    self.optimizer.update()

def sync_qnet(self): # 두 신경망 동기화
    self.qnet_target = copy.deepcopy(self.qnet)

episodes = 300 # 에피소드 수
sync_interval = 20 # 신경망 동기화 주기(20번째 에피소드마다 동기화)
env = gym.make('CartPole-v0', render_mode='rgb_array')
agent = DQNAgent()
reward_history = [] # 에피소드별 보상 기록

```

```

for episode in range(episodes):
    state = env.reset()[0]
    done = False
    total_reward = 0

    while not done:
        action = agent.get_action(state)
        next_state, reward, terminated, truncated, info = env.step(action)
        done = terminated | truncated

        agent.update(state, action, reward, next_state, done)
        state = next_state
        total_reward += reward

    if episode % sync_interval == 0:
        agent.sync_qnet()

    reward_history.append(total_reward)
    if episode % 10 == 0:
        print("episode : {}, total reward : {}".format(episode, total_reward))

# 카트 폴 에서 에피소드별 보상 총합의 추이
plt.xlabel('Episode')
plt.ylabel('Total Reward')
plt.plot(range(len(reward_history)), reward_history)
plt.show()

# 학습이 끝난 에이전트에 탐욕 행동을 선택하도록 하여 플레이
env2 = gym.make('CartPole-v0', render_mode='human')

agent.epsilon = 0 # 탐욕 정책(무작위로 행동할 확률 ε을 0로 설정)
state = env2.reset()[0]
done = False
total_reward = 0

while not done:
    action = agent.get_action(state)
    next_state, reward, terminated, truncated, info = env2.step(action)
    done = terminated | truncated
    state = next_state
    total_reward += reward
    env2.render()
print('Total Reward:', total_reward)

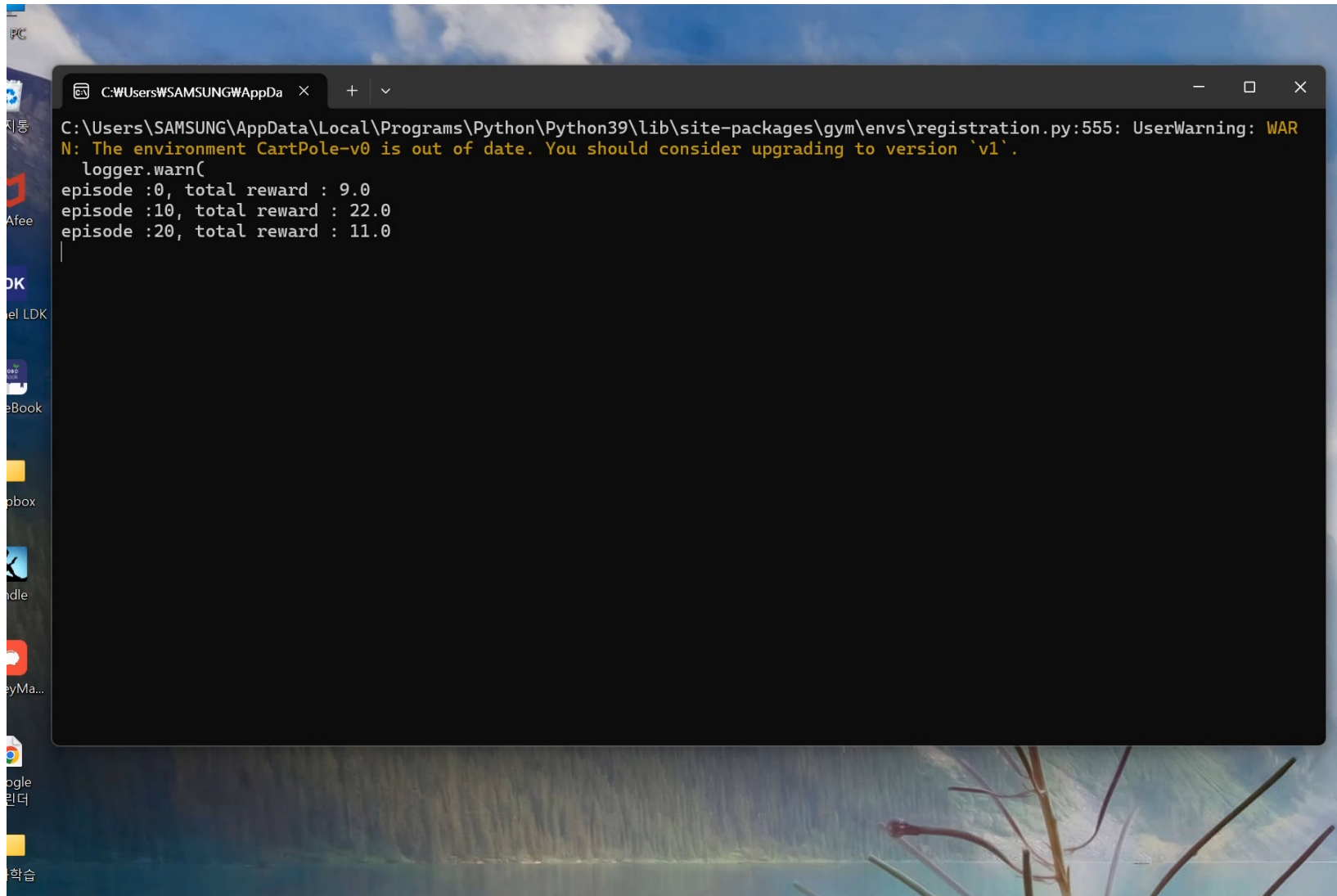
```

# 실습

- Hyper-Parameter

- 할인율 ( $\gamma = 0.98$ )
- 학습률 ( $lr = 0.0005$ )
- $\epsilon$ -탐욕 확률 ( $\epsilon = 0.05$ )
- 경험 재생 버퍼 크기 ( $buffer\_size = 100000$ )
- 미니배치 크기 ( $batch\_size = 32$ )
- 에피소드 수 ( $episodes = 300$ )
- 동기화 주기 ( $sync\_interval = 20$ )
- 신경망 구조 (계층 수, Linear 계층의 노드 수 등)

# 실습



```
C:\Users\WSAMSUNG\AppData\Local\Programs\Python\Python39\lib\site-packages\gym\envs\registration.py:555: UserWarning: WARN: The environment CartPole-v0 is out of date. You should consider upgrading to version 'v1'.  
  logger.warn(  
episode :0, total reward : 9.0  
episode :10, total reward : 22.0  
episode :20, total reward : 11.0
```

# Mountain Car

- Environment

- State (observation)

Num	Observation	Min	Max	Unit
0	position of the car along the x-axis	-1.2	0.6	position (m)
1	velocity of the car	-0.07	0.07	velocity (v)

- Action

0: Accelerate to the left  
1: Don't accelerate  
2: Accelerate to the right

- Rewards

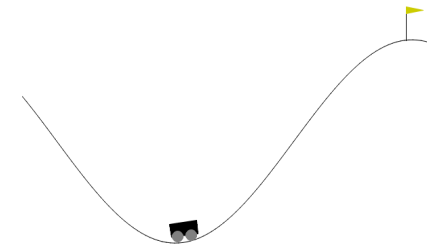
- -1 for each time step (goal is to reach the flag as quickly as possible)

- Starting state

- Position of the car: uniformly random value in  $[-0.6, -0.4]$

- Episode ends if any one of the following occurs

1. Termination: The position of the car is greater than or equal to 0.5 (the goal position on top of the right hill)
2. Truncation: The length of the episode is 200.

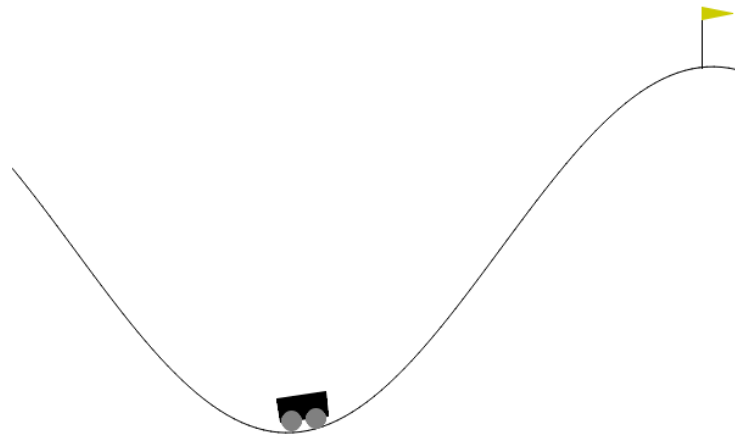


# Quiz

(Q1) DQN 을 Mountain Car 문제에 적용하되, Hyper-parameter 를 변경하여 최대의 total reward 를 갖는 policy 를 결정하라.

(제출물: PPT)

- 1) 최적 hyperparameter
- 2) Episode 별 total reward graph
- 3) 최대 total reward 값 및 해당 policy 적용 시의 동영상





# Double DQN

- DQN 의 Overfitting 문제

- TD Target

$$R_t + \gamma \max_a Q(S_{t+1}, a)$$

- Estimation of TD Target (Network)

- Original network:  $\theta \rightarrow R_t + \gamma \max_a Q_\theta(S_{t+1}, a)$

- Target network:  $\theta' \rightarrow R_t + \gamma \max_a Q_{\theta'}(S_{t+1}, a)$

- Overfitting (과대적합) 문제

- $\max_a Q_\theta(S_{t+1}, a)$  : training data 에 overfitting 가능

- (ex)  $q(s, a_0) = q(s, a_1) = q(s, a_2) = q(s, a_3) = 0$

- $\Rightarrow$  계산값:  $\mathbb{E} \left[ \max_a q(s, a) \right] = 0$

- $\Rightarrow$  추정값:  $\mathbb{E} \left[ \max_a Q(s, a) \right] > 0$  (Q: 정규분포 데이터)

# Double DQN

- DQN 의 Overfitting 문제

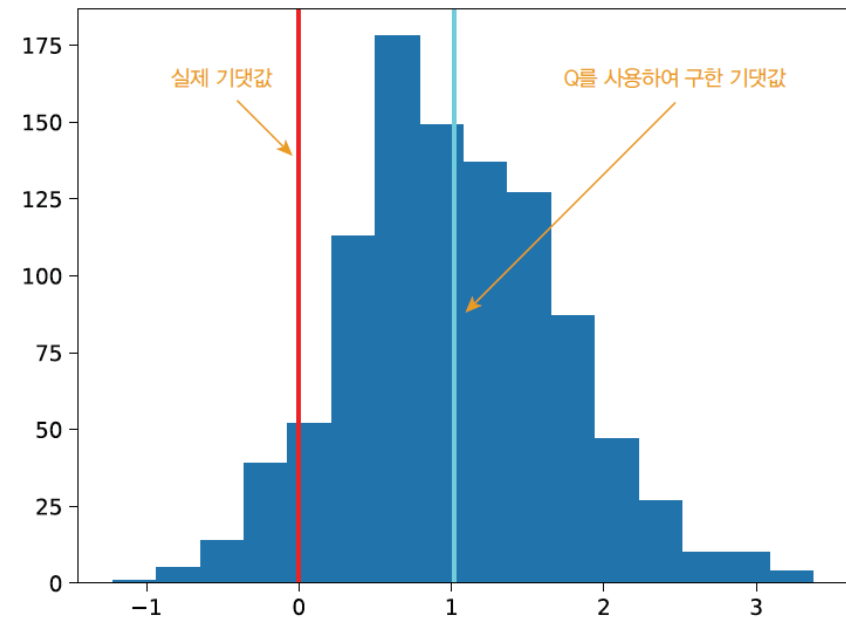
```
import numpy as np
import matplotlib.pyplot as plt

samples = 1000
action_size = 4
Qs = []

for _ in range(samples):
    # 정규분포에서 생성한 무작위 수를 노이즈로 추가
    Q = np.random.randn(action_size)
    Qs.append(Q.max())

# 히스토그램으로 시각화
plt.hist(Qs, bins=16)
plt.axvline(x=0, color='red')
plt.axvline(x=np.array(Qs).mean(), color='cyan')
plt.show()
```

그림 C-1 Q.max()의 데이터 분포



최대값을 구하는 것이므로 noise 영향 큼

# Double DQN

- DQN 의 Overfitting 해결방법

- 두 개의 독립적 Q 함수 사용: Q & Q\_prime ⇒ double DQN
  - 최대 행동 선택 시: Q 사용
  - 선택된 행동에 대한 Q 값 : Q\_prime 사용

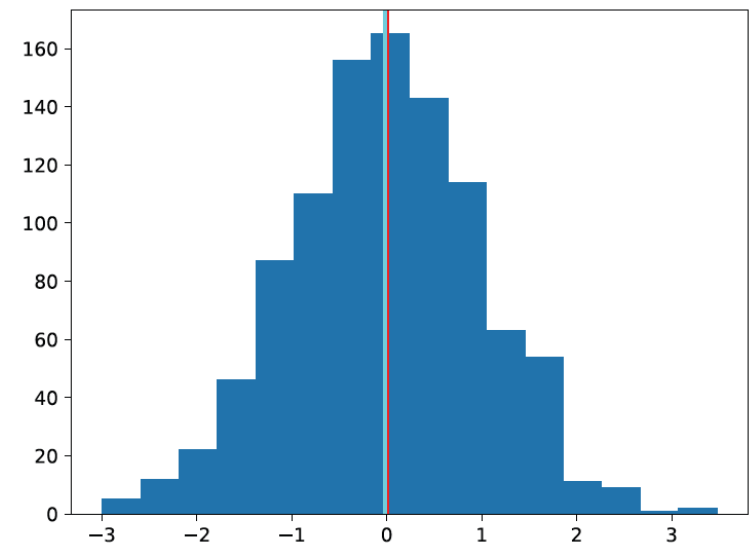
```
import numpy as np
import matplotlib.pyplot as plt

samples = 1000
action_size = 4
Qs = []

for _ in range(samples):
    Q = np.random.randn(action_size)
    Q_prime = np.random.randn(action_size) # 또 다른 Q 함수
    idx = np.argmax(Q) # Q에서 최대 행동 선택
    Qs.append(Q_prime[idx]) # 선택된 행동에 대한 값을 Q_prime에서 구함

# 히스토그램으로 시각화
plt.hist(Qs, bins=16)
plt.axvline(x=0, color='red')
plt.axvline(x=np.array(Qs).mean(), color='cyan')
plt.show()
```

그림 C-2 Double DQN 이용 시의 데이터 분포



# Prioritized Experience Replay

- Experience Replay

- 경험 데이터  $E_t = (S_t, A_t, R_t, S_{t+1})$  를 buffer 에 저장

- => 학습 시 무작위로 추출하여 사용

- Prioritized Experience Replay (PER)

- 경험 데이터에 학습 Loss 값 포함

- $\delta_t = \left| R_t + \gamma \max_a Q_{\theta'}(S_{t+1}, a) - Q_{\theta}(S_t, a) \right|$  : loss

- $\delta_t$  가 큰 경우 : 수정할 것이 많음 -> 학습 시킬 것이 많음

- $\delta_t$  가 작은 경우 : 수정할 것이 적음 -> 학습 시킬 것이 적음

- 경험 데이터  $E_t = (S_t, A_t, R_t, S_{t+1}, \delta_t)$

- 경험데이터 선택 시 우선순위를 고려하여 추출 => 학습속도 향상

- 선택확률

$$p_i = \frac{\delta_i}{\sum_{k=0}^N \delta_k}$$

현 시세 값.

# Dueling DQN

- Advantage Function

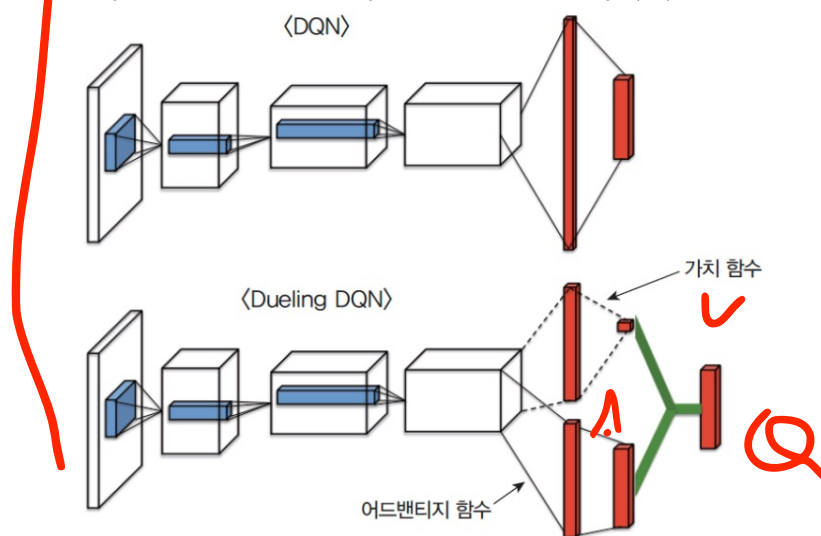
Dueling = competing

- Q 함수 (action value) 와 V 함수 (state value) 의 차이
- $A_{\pi}(s, a) = Q_{\pi}(s, a) - V_{\pi}(s)$

- $Q_{\pi}(s, a)$ : 상태  $s$ 에서 '특정 행동  $a$ '를 취하고 그 이후에는  $\pi$ 에 따라 행동했을 때 얻을 수 있는 기대 수익
- $V_{\pi}(s)$ : 상태  $s$ 에서 이후의 모든 행동을 정책  $\pi$ 에 따라 했을 때 얻을 수 있는 기대 수익

- Dueling DQN

$$Q_{\pi}(s, a) = A_{\pi}(s, a) + V_{\pi}(s)$$



어떤 행동(action)을 선택해도 결과가 달라지지 않는 상황

- DQN : 학습 진행되지 않음
- Dueling DQN: V 함수로 Q 근사 학습 진행  
학습속도 향상 기대

