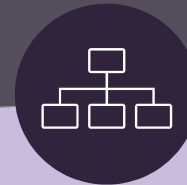
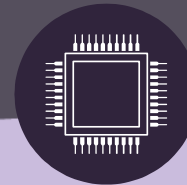


딥러닝 Ⅲ

경남대학교 창의융합대학



담당교수 : 유 현 주

comjoo@uok.ac.kr



Deep Learning



Review Practice

- Image Classification using CNN
- Yolo object detection
- Weather Prediction using RNN
- BBC Articles LSTM Model



Image Classification using CNN



- 채소 이미지 분류







- 15가지 종류의 일반 채소를 사용하여 수행
- 15개의 클래스에서 총 21,000개의 이미지가 사용
 - 콩, 쓴박, 병박, 브린잘, 브로콜리, 양배추, 캡시쿰, 당근, 콜리플라워, 오이, 파파야, 감자, 호박, 무, 토마토
- 각 클래스에는 크기 224×224의 이미지 1400개가 *.jpg 형식
- 데이터셋은 학습 목적으로 70%, 검증 목적으로 15%, 테스트 목적으로 15% 분할
 - 훈련 : 15,000장의 이미지 / 테스트 : 3000개 이미지 / 검증 : 3000개 이미지
 - 위의 각 폴더에는 각 채소의 이미지가 있는 다양한 채소의 하위 폴더가 포함

Yolo object detection



- Yolo는 객체 감지를 위해 합성곱 신경망을 사용하는 알고리즘
- Yolo를 구축
 - 텐서플로우(딥러닝), NumPy(수치 계산) 및 베개(이미지 처리) 라이브러리가 필요
 - 박스 색상을 바운딩하기 위해 Seaborn의 색상 팔레트를 사용
 - IPython display() 이미지를 표시

- Dataset

-  coco.names
 -  detections.gif
 -  dog.jpg
 -  futur.ttf
 -  office.jpg
 -  yolov3.weights



- Weather Prediction using RNN
 - 시계열 예측
 - 기상 데이터를 기반으로 일일 최고 기온을 예측하기 위해 RNN을 사용
 - 사용된 데이터셋은 시애틀 날씨 데이터셋
- BBC Articles|LSTM Model
 - **LSTM**과 텐서플로를 사용한 **BBC** 뉴스 분류



Contents

- 1장: 케라스를 사용한 인공 신경망 소개
- 2장: 심층 신경망 훈련
- 3장: 텐서플로를 사용한 사용자 정의 모델과 훈련
- 4장: 텐서플로를 사용한 데이터 적재와 전처리
- 5장: 합성곱 신경망을 사용한 컴퓨터 비전
- 6장: RNN과 CNN을 사용한 시퀀스 처리
- 7장: RNN과 어텐션을 사용한 자연어 처리
- 8장: 오토인코더, GAN 그리고 확산 모델
- 9장: 강화 학습
- 10장: 대규모 텐서플로 모델 훈련과 배포

Deep Learning



강화 학습

- 보상을 최적화하기 위한 학습
- 정책 탐색
- OpenAI Gym
- 신경망 정책
- 행동 평가: 신용 할당 문제
- 정책 그레디언트





보상을 최적화하기 위한 학습(1)

- 강화 학습

- 관측(에이전트) → 행동(환경) → 보상: 에이전트의 목적은 보상의 장기간 기대치를 최대화하는 행동을 학습

- 강화 학습 사례

- 에이전트(로봇을 제어하는 프로그램) → 카메라나 터치 센서 같은 여러 센서를 통해 환경을 관찰하고 모터를 구동하기 위해 시그널을 전송 → 목적지에 도착할 때 양의 보상을 받고, 시간을 낭비하거나 잘못된 방향으로 향할 때 음의 보상을 받도록 프로그램
- 에이전트는(미스 팩맨 제어 프로그램) → 아타리 게임의 아홉 가지 조이스틱 위치 → 관측은 스크린샷이 되고 보상은 게임의 점수
- 바둑 같은 보드 게임을 플레이하는 프로그램
- 스마트 온도조절기
- 주식시장의 가격을 관찰하고 매초 얼마나 사고팔아야 할지 결정

보상을 최적화하기 위한 학습(2)



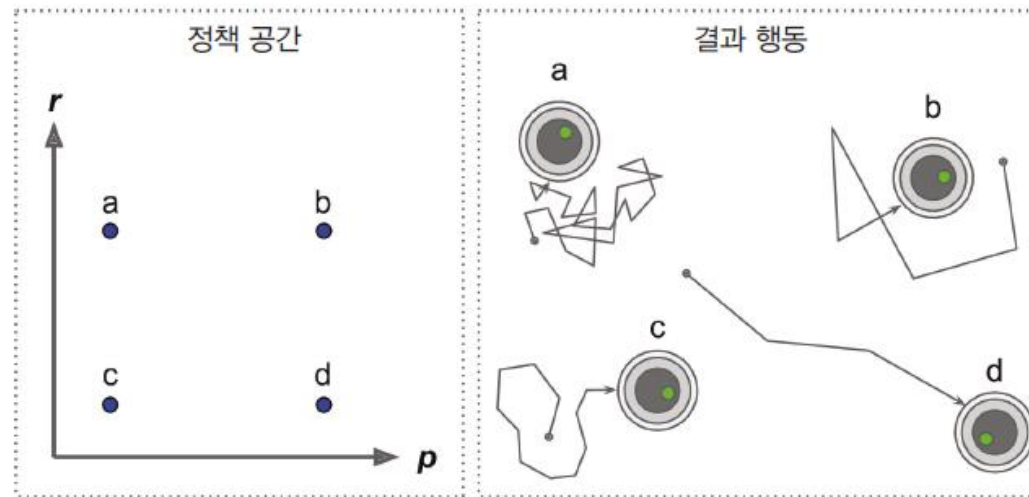
강화 학습의 예: (a) 로봇, (b) 미스 팩맨, (c) 바둑 게임,
(d) 온도 조절기, (e) 자동 매매 프로그램



정책 탐색(1)

- 정책: 소프트웨어 에이전트가 행동을 결정하기 위해 사용하는 알고리즘
- 확률적 정책
 - 30분 동안 수집한 먼지의 양을 보상으로 받는 로봇 진공청소기
 - 로봇 훈련을 위해 변경이 가능한 두 개의 정책 파라미터: 확률 p , 각도의 범위 r
- 유전 알고리즘
 - 1세대 정책 100개를 랜덤하게 생성해서 시도 → 성능이 낮은 정책 80개는 버리고 20개를 살려 각각 자식 정책 4개를 생산(이 자식 정책은 부모를 복사한 것에 약간의 무작위성을 더한 것) → 살아남은 정책과 그 자식은 2세대를 구성 → 좋은 정책을 찾을 때까지 여러 세대에 걸쳐 반복
- 정책 그레이디언트(policy gradient, PG)
 - 정책 파라미터에 대한 보상의 그레이디언트를 평가해서 높은 보상의 방향을 따르는 그레이디언트로 파라미터를 수정하는 최적화 기법을 사용

정책 탐색(2)



정책 공간에 있는 지점 4개(왼쪽)와 이에 상응하는 에이전트의 행동(오른쪽)

OpenAI Gym(1)



- OpenAI Gym
 - 다양한 종류의 시뮬레이션 환경(아타리 게임, 보드 게임, 2D와 3D 물리 시뮬레이션 등)을 제공하는 툴킷
 - OpenAI는 일론 머스크가 공동 창업한 인공지능 연구 회사. 인류 우호적인 AI의 개발, 확산을 목적
- OpenAI Gym - 코랩에 사전 설치되어 있는 이전 버전을 최신 버전으로 교체

코랩을 사용하는 경우 이 명령을 실행하세요!

```
%pip install -q -U gymnasium
```

```
%pip install swig
```

```
%pip install -q -U gymnasium[classic_control,box2d,atari,accept-rom-license]
```



OpenAI Gym(2)

- OpenAI Gym을 임포트하고 환경 만들기
 - CartPole 환경
 - 카트 위에 놓인 막대가 넘어지지 않도록 왼쪽이나 오른쪽으로 가속할 수 있는 2D 시뮬레이션

```
import gymnasium as gym

env = gym.make("CartPole-v1", render_mode="rgb_array")
```

- 환경을 만든 후 reset() 메서드로 꼭 초기화

```
>>> obs, info = env.reset(seed=42)
>>> obs
array([ 0.0273956 , -0.00611216, 0.03585979, 0.0197368 ], dtype=float32)
>>> info
{}
```

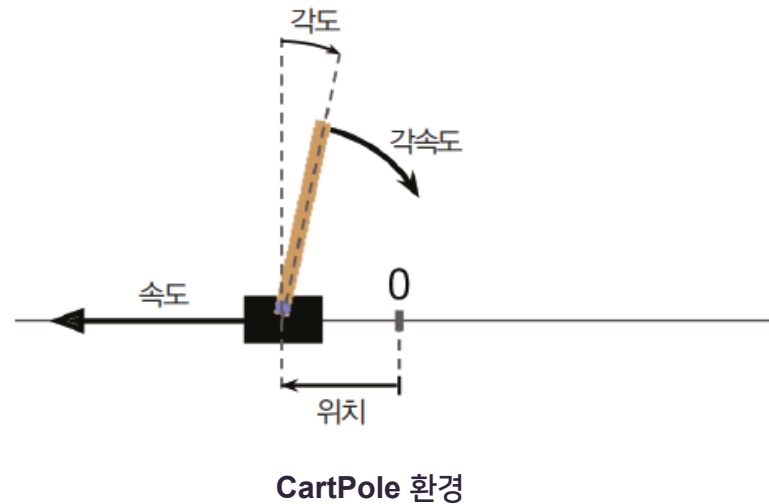


OpenAI Gym(3)

- render() 메서드를 호출해 이 환경을 이미지로 렌더링

```
>>> img = env.render()  
>>> img.shape # 높이, 너비, 채널(3 = 빨강, 초록, 파랑)  
(400, 600, 3)
```

- 맷플롯립의 imshow() 함수를 사용해 이미지를 화면에 그릴 수 있음



CartPole 환경

OpenAI Gym(4)



- 이 환경에서 어떤 행동이 가능한지 확인
 - Discrete(2)는 가능한 행동이 정수 0과 1이라는 것을 의미
 - 각각 왼쪽 가속과 오른쪽 가속

```
>>> env.action_space
Discrete(2)
```

- 막대가 오른쪽($obs[2] > 0$)으로 기울어져 있기 때문에 카트를 오른쪽으로 가속

```
>>> action = 1 # 오른쪽으로 가속
>>> obs, reward, done, truncated, info = env.step(action)
>>> obs
array([ 0.02727336, 0.18847767, 0.03625453, -0.26141977], dtype=float32)
>>> reward
1.0
>>> done
False
>>> truncated
False
>>> info
{}
```

OpenAI Gym(5)



- step() 메서드는 주어진 행동을 실행하고 네 가지 값을 반환
 - obs
 - 새로운 관측값. 이제 카트가 오른쪽 방향으로 움직임
 - reward
 - 이 환경에서는 어떤 행동을 실행해도 매 스텝마다 1.0의 보상을 받음
그러므로 시스템의 목적은 가능한 한 오랫동안 실행
 - done
 - 이 값이 True이면 이 에피소드 17가 끝난 것임
막대가 너무 기울어지거나 화면 밖으로 나가거나 200 스텝을 넘기면 에피소드가 끝남
 - truncated
 - 이 값은 에피소드가 조기에 중단되는 경우 True가 됨
 - info
 - reset() 메서드가 반환하는 값처럼 환경에 관련된 추가 정보를 담은 딕셔너리

OpenAI Gym(6)



- 막대가 왼쪽으로 기울어지면 카트를 왼쪽으로 가속하고 오른쪽으로 기울어지면 오른쪽으로 가속
- 이 정책으로 에피소드 500번 실행해서 얻은 평균 보상을 확인

```
def basic_policy(obs):
    angle = obs[2]
    return 0 if angle < 0 else 1

totals = []
for episode in range(500):
    episode_rewards = 0
    obs, info = env.reset(seed=episode)
    for step in range(200):
        action = basic_policy(obs)
        obs, reward, done, truncated, info = env.step(action)
        episode_rewards += reward
        if done or truncated:
            break

    totals.append(episode_rewards)
```

OpenAI Gym(7)



- 결과 확인

- 500번을 시도해도 이 정책은 막대를 쓰러뜨리지 않고 68번 이상의 스텝을 진행하지 못함

```
>>> import numpy as np
>>> np.mean(totals), np.std(totals), min(totals), max(totals)
(41.698, 8.389445512070509, 24.0, 63.0)
```



신경망 정책(1)

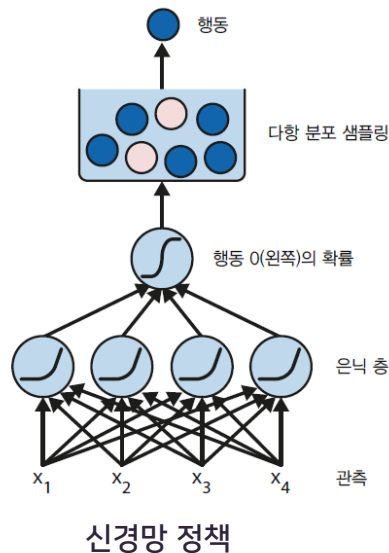
- 신경망 정책 학습

- 관측을 입력으로 받고 실행할 행동을 출력: 각 행동에 대한 확률을 추정하고, 추정된 확률에 따라 랜덤하게 행동을 선택

- CartPole 환경의 경우

- 가능한 행동이 두 개(왼쪽과 오른쪽) 있으므로 하나의 출력 뉴런만 있으면 됨
- 이 뉴런은 행동 0(왼쪽)의 확률을 출력합니다. 당연히 행동 1(오른쪽)의 확률은 $1-p$

- 새로운 행동을 탐험(exploring)하는 것과 잘 할 수 있는 행동을 활용(exploiting)하는 것 사이의 균형





신경망 정책(2)

- 케라스를 사용하여 신경망 정책을 구현하는 코드

```
import tensorflow as tf

model = tf.keras.Sequential([
    tf.keras.layers.Dense(5, activation="relu"),
    tf.keras.layers.Dense(1, activation="sigmoid"),
])
```



행동 평가: 신용 할당 문제

● 신용 할당 문제

● 보상은 일반적으로 드물고 지연되어 나타남

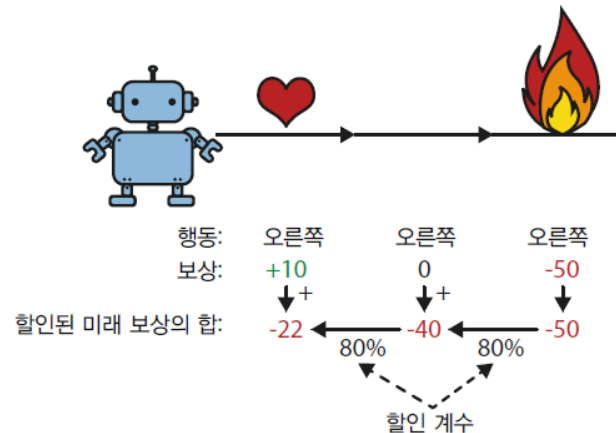
- 예를 들어 에이전트가 100스텝 동안 막대의 균형을 유지했다면 이 100번의 행동 중 어떤 것이 좋고, 어떤 것이 나쁜지 알 수 있을까?

● 에이전트가 보상을 받았을 때 어떤 행동 덕분인지(혹은 탓인지) 알기 어려운 문제 해결

- 흔히 사용하는 전략은 행동이 일어난 후 각 단계마다 할인 계수 γ (감마)를 적용한 보상을 모두 합하여 행동을 평가

● 행동 이익

- 평균적으로 다른 가능한 행동과 비교해서 각 행동이 얼마나 좋은지 혹은 나쁜지를 추정



행동의 이득 계산하기: 할인된 미래 보상의 합



정책 그레이디언트(1)

- REINFORCE 알고리즘

- PG 알고리즘은 높은 보상을 얻는 방향의 그레이디언트를 따르도록 정책의 파라미터를 최적화하는 알고리즘입니다. REINFORCE 알고리즘은 인기 있는 PG 알고리즘 중 하나

- REINFORCE 알고리즘 방식

1. 먼저 신경망 정책이 여러 번에 걸쳐 게임을 플레이하고 매 스텝마다 선택된 행동이 더 높은 가능성을 가지도록 만드는 그레이디언트를 계산(이 그레이디언트를 적용 전)
2. 에피소드를 몇 번 실행한 다음, 각 행동의 이익을 계산
3. 한 행동의 이익이 양수 또는 음수에 따라 각각에 맞는 그레이디언트를 적용
4. 모든 결과 그레이디언트 벡터를 평균 내어 경사 하강법 스텝을 수행



정책 그레디디언트(2)

- 케라스를 사용하여 이 알고리즘을 구현

```
def play_one_step(env, obs, model, loss_fn):  
    with tf.GradientTape() as tape:  
        left_proba = model(obs[np.newaxis])           ❶  
        action = (tf.random.uniform([1, 1]) > left_proba)  ❷  
        y_target = tf.constant([[1.]]) - tf.cast(action, tf.float32)  ❸  
        loss = tf.reduce_mean(loss_fn(y_target, left_proba))  ❹  
  
        grads = tape.gradient(loss, model.trainable_variables)  ❹  
        obs, reward, done, truncated, info = env.step(int(action))  
        return obs, reward, done, truncated, grads           ❺
```



정책 그레디디언트(3)

- `play_one_step()` 함수를 사용해 여러 에피소드를 플레이하고, 전체 보상 및 각 에피소드와 스텝의 그레디디언트를 반환하는 또 다른 함수 만들기

```
def play_multiple_episodes(env, n_episodes, n_max_steps, model, loss_fn):
    all_rewards = []
    all_grads = []
    for episode in range(n_episodes):
        current_rewards = []
        current_grads = []
        obs, info = env.reset()
        for step in range(n_max_steps):
            obs, reward, done, truncated, grads = play_one_step(
                env, obs, model, loss_fn)
            current_rewards.append(reward)
            current_grads.append(grads)
            if done or truncated:
                break

        all_rewards.append(current_rewards)
        all_grads.append(current_grads)

    return all_rewards, all_grads
```




정책 그레디디언트(4)

- 첫 번째 함수는 각 스텝에서 할인된 미래 보상의 합을 계산
 - 두 번째 함수는 여러 에피소드에 걸쳐 계산된 할인된 모든 보상(대가)에서 평균을 빼고 표준 편차로 나누어 정규화

```
def discount_rewards(rewards, discount_factor):
    discounted = np.array(rewards)
    for step in range(len(rewards) - 2, -1, -1):
        discounted[step] += discounted[step + 1] * discount_factor
    return discounted

def discount_and_normalize_rewards(all_rewards, discount_factor):
    all_discounted_rewards = [discount_rewards(rewards, discount_factor)
                              for rewards in all_rewards]
    flat_rewards = np.concatenate(all_discounted_rewards)
    reward_mean = flat_rewards.mean()
    reward_std = flat_rewards.std()
    return [(discounted_rewards - reward_mean) / reward_std
            for discounted_rewards in all_discounted_rewards]
```



정책 그레이디언트(5)

- 함수 확인

- `discount_rewards()` 함수를 호출하면 정확히 기대한 값이 반환

```
>>> discount_rewards([10, 0, -50], discount_factor=0.8)
array([-22, -40, -50])
>>> discount_and_normalize_rewards([[10, 0, -50], [10, 20]],
...                               discount_factor=0.8)
...
...
[array([-0.28435071, -0.86597718, -1.18910299]),
 array([1.26665318, 1.0727777 ])]
```



정책 그레디디언트(6)

- 훈련 반복을 150번 실행. 각 반복은 에피소드 10개를 진행하고 각 에피소드는 스텝을 최대 200번 플레이 할인 계수는 0.95를 적용

```
n_iterations = 150
n_episodes_per_update = 10
n_max_steps = 200
discount_factor = 0.95
```

- 옵티마이저와 손실 함수
 - 학습률 0.01인 Adam 옵티마이저가 무난
 - 이진 분류기를 훈련하므로 이진 크로스 엔트로피 손실 함수를 사용

```
optimizer = tf.keras.optimizers.Nadam(learning_rate=0.01)
loss_fn = tf.keras.losses.binary_crossentropy
```



정책 그레디디언트(7)

- 훈련 반복을 만들어 실행

```
for iteration in range(n_iterations):  
    all_rewards, all_grads = play_multiple_episodes(  
        env, n_episodes_per_update, n_max_steps, model, loss_fn) ❶  
    all_final_rewards = discount_and_normalize_rewards(all_rewards,  
        discount_factor) ❷  
  
    all_mean_grads = []  
    for var_index in range(len(model.trainable_variables)):  
        mean_grads = tf.reduce_mean(  
            [final_reward * all_grads[episode_index][step][var_index]  
            for episode_index, final_rewards in enumerate(all_final_rewards) ❸  
            for step, final_reward in enumerate(final_rewards)], axis=0)  
        all_mean_grads.append(mean_grads)  
    optimizer.apply_gradients(zip(all_mean_grads, model.trainable_variables)) ❹
```



강화 학습

- 마르코프 결정 과정
- 시간차 학습
- Q-러닝
- 심층 Q-러닝 구현
- 심층 Q-러닝의 변형
- 다른 강화 학습 알고리즘





마르코프 결정 과정(1)

- 마르코프 연쇄
 - 메모리가 없는 확률 과정(stochastic process)
 - 다양한 역학 관계를 모델링할 수 있어서 열역학, 화학, 통계 등 많은 분야에서 사용

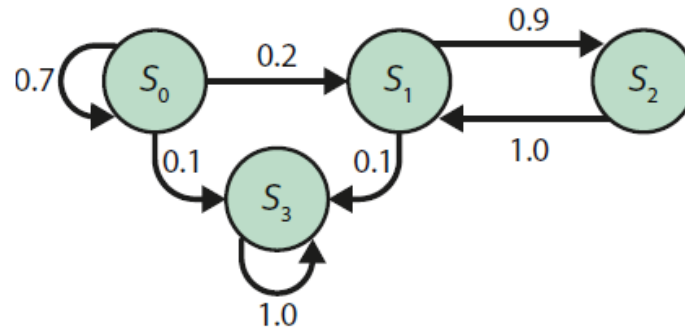


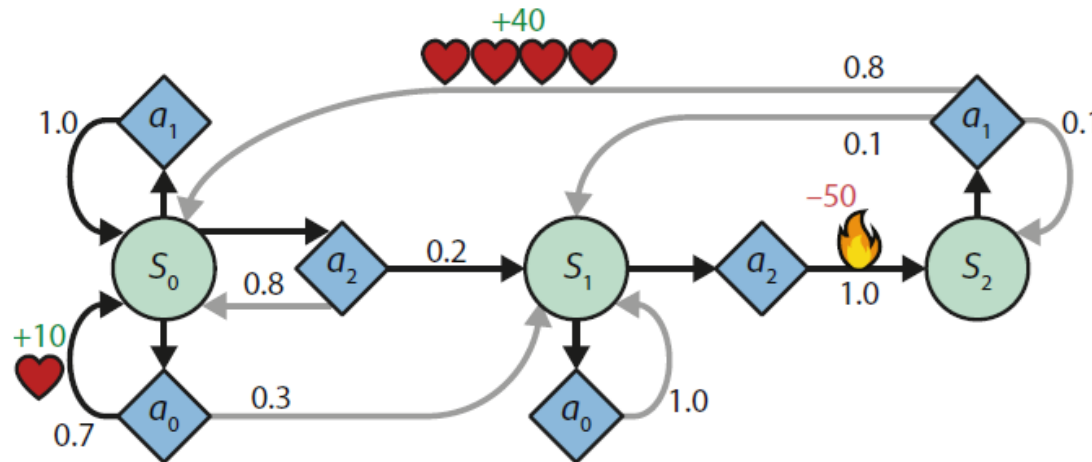
그림 18-7 마르코프 연쇄의 예



마르코프 결정 과정(2)

● 마르코프 결정 과정

- 마르코프 결정 과정은 1950년대 리처드 벨만(Richard Bellman)이 처음으로 논문에 기술
- 마르코프 연쇄와 비슷하지만 약간 다른 점이 있음
 - 각 스텝에서 에이전트는 여러 가능한 행동 중 하나를 선택할 수 있고, 전이 확률은 선택된 행동에 따라 달라짐
 - 어떤 상태 전이는 보상(음수 또는 양수)을 반환
 - 에이전트의 목적은 시간이 지남에 따라 보상을 최대화하기 위한 정책을 찾는 것



마르코프 결정 과정의 예



마르코프 결정 과정(3)

- 벨만 최적 방정식(Bellman optimality equation)

- 어떤 상태 s 의 최적의 상태 가치(optimal state value) $V^*(s)$ 를 추정하는 방법
- 이 값은 에이전트가 상태 s 에 도달한 후 최적으로 행동한다고 가정하고 평균적으로 기대할 수 있는 할인된 미래 보상의 합

벨만 최적 방정식

$$V^*(s) = \max_a \sum_{s'} T(s, a, s') [R(s, a, s') + \gamma \cdot V^*(s')] \quad \text{모든 } s \text{에 대해}$$

- 가치 반복(value iteration) 알고리즘을 사용하여 반복적으로 업데이트

가치 반복 알고리즘

$$V_{k+1}(s) \leftarrow \max_a \sum_{s'} T(s, a, s') [R(s, a, s') + \gamma \cdot V_k(s')] \quad \text{모든 } s \text{에 대해}$$

- Q-가치 반복(Q-value iteration) 알고리즘

Q-가치 반복 알고리즘

$$Q_{k+1}(s, a) \leftarrow \sum_{s'} T(s, a, s') [R(s, a, s') + \gamma \cdot \max_{a'} Q_k(s', a')] \quad \text{모든 } (s, a) \text{에 대해}$$



마르코프 결정 과정(4)

- 최적의 정책인 $\pi^*(s)$ 를 정의
 - 에이전트가 상태 s 에 도달했을 때 가장 높은 Q-가치를 가진 행동을 선택

$$\pi^*(s) = \underset{a}{\operatorname{argmax}} Q^*(s, a)$$

- 알고리즘을 MDP에 적용
 - MDP 정의

```
transition_probabilities = [ # 크기는 [s, a, s']
    [[0.7, 0.3, 0.0], [1.0, 0.0, 0.0], [0.8, 0.2, 0.0]],
    [[0.0, 1.0, 0.0], None, [0.0, 0.0, 1.0]],
    [None, [0.8, 0.1, 0.1], None]
]
rewards = [ # 크기는 [s, a, s']
    [[+10, 0, 0], [0, 0, 0], [0, 0, 0]],
    [[0, 0, 0], [0, 0, 0], [0, 0, -50]],
    [[0, 0, 0], [+40, 0, 0], [0, 0, 0]]
]
possible_actions = [[0, 1, 2], [0, 2], [1]]
```



마르코프 결정 과정(5)

- 모든 Q-가치를 0으로 초기화
 - 불가능한 행동은 제외. 이 행동의 Q-가치는 $-\infty$ 로 설정

```
Q_values = np.full((3, 3), -np.inf) # 불가능한 행동에 대해서는 -np.inf
for state, actions in enumerate(possible_actions):
    Q_values[state, actions] = 0.0 # 모든 가능한 행동에 대해서
```

- Q-가치 반복 알고리즘을 실행

```
gamma = 0.90 # 할인 계수

for iteration in range(50):
    Q_prev = Q_values.copy()
    for s in range(3):
        for a in possible_actions[s]:
            Q_values[s, a] = np.sum([
                transition_probabilities[s][a][sp]
                * (rewards[s][a][sp] + gamma * Q_prev[sp].max())
                for sp in range(3)])
```



마르코프 결정 과정(6)

- 결과 Q-가치

```
>>> Q_values
array([[18.91891892, 17.02702702, 13.62162162],
       [ 0.          ,        -inf, -4.87971488],
       [        -inf, 50.13365013,        -inf]])
```

- 각 상태에 대해 가장 높은 Q-가치를 갖는 행동을 확인

```
>>> np.argmax(Q_values, axis=1) # 각 상태에 대해 최적의 행동
array([0, 0, 1])
```



시간차 학습

- 시간차 학습(temporal difference learning, TD)
 - 가치 반복 알고리즘과 매우 비슷하지만, 에이전트가 MDP에 대해 일부 정보만 알고 있을 때를 다룰 수 있도록 변형
 - 에이전트는 탐험 정책(exploration policy)을 사용해 MDP를 탐험

TD 학습 알고리즘

$$V_{k+1}(s) \leftarrow (1-\alpha)V_k(s) + \alpha(r + \gamma \cdot V_k(s'))$$

$$V_{k+1}(s) \leftarrow V_k(s) + \alpha \delta_k(s, r, s') \quad \text{여기에서} \quad \delta_k(s, r, s') = r + \gamma \cdot V_k(s') - V_k(s)$$

$$V(s) \leftarrow_{\alpha} r + \gamma \cdot V(s')$$



Q-러닝(1)

- Q-러닝(Q-learning) 알고리즘

- 전이 확률과 보상을 초기에 알지 못한 상황에서 Q-가치 반복 알고리즘을 적용한 것

Q-러닝 알고리즘

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

- Q-러닝 알고리즘을 구현

- 먼저 에이전트가 환경을 탐색하게 만들기
- 에이전트가 한 행동을 실행하고 결과 상태와 보상을 받을 수 있는 스텝 함수 만들기

```
def step(state, action):  
    probas = transition_probabilities[state][action]  
    next_state = np.random.choice([0, 1, 2], p=probas)  
    reward = rewards[state][action][next_state]  
    return next_state, reward
```

Q-러닝(2)



- 에이전트의 탐색 정책을 구현

```
def exploration_policy(state):  
    return np.random.choice(possible_actions[state])
```

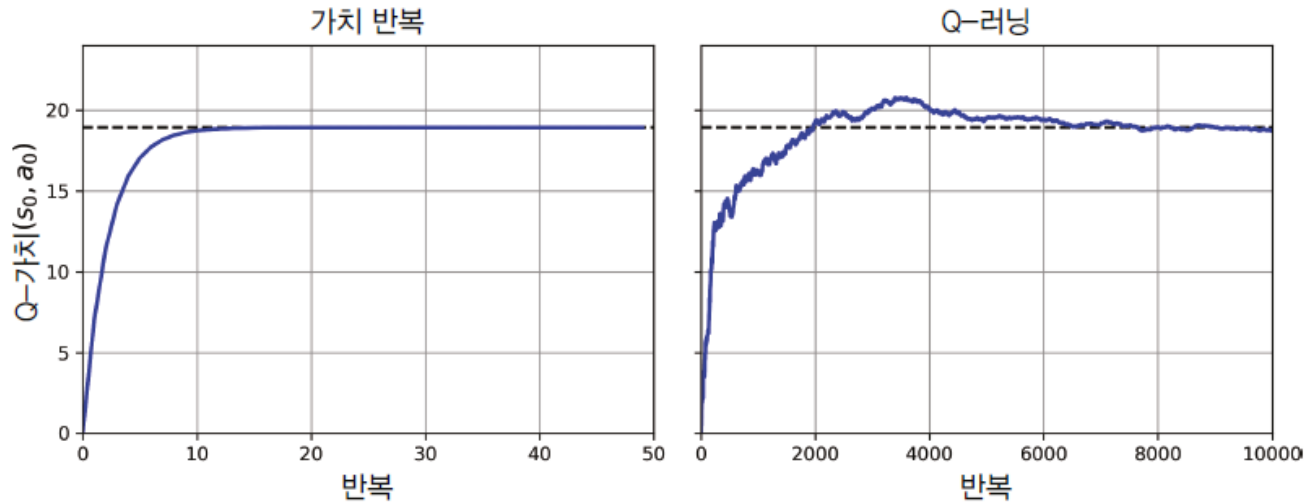
- Q-가치를 초기화한 후 학습률 감쇠(거듭제곱 기반 스케줄링(power scheduling))를 사용해 Q-러닝 알고리즘을 실행

```
alpha0 = 0.05    # 초기 학습률  
decay = 0.005    # 학습률 감쇠  
gamma = 0.90     # 할인 계수  
state = 0        # 초기 상태  
  
for iteration in range(10_000):  
    action = exploration_policy(state)  
    next_state, reward = step(state, action)  
    next_value = Q_values[next_state].max() # 다음 스텝에서 탐욕적 정책  
    alpha = alpha0 / (1 + iteration * decay)  
    Q_values[state, action] *= 1 - alpha  
    Q_values[state, action] += alpha * (reward + gamma * next_value)  
    state = next_state
```



Q-러닝(3)

- 최적의 Q-가치에 수렴하겠지만 많은 반복과 하이퍼파라미터 튜닝이 필요



Q-가치 반복 알고리즘(왼쪽)과 Q-러닝 알고리즘(오른쪽)

- Q-러닝 알고리즘 - 오프-폴리시(off-policy)
- 정책 그레이디언트 알고리즘 - 온-폴리시(on-policy)

Q-러닝(4)



탐험 정책

- ϵ - 그리디 정책(ϵ -greedy policy)을 사용
 - 각 스텝에서 ϵ 확률로 랜덤하게 행동하거나 $1-\epsilon$ 확률로 그 순간 가장 최선인 것으로(가장 높은 Q-가치를 선택하여) 행동
 - ϵ -그리디 정책의 장점은 (완전한 랜덤 정책에 비해) Q-가치 추정이 점점 더 향상되기 때문에 환경에서 관심 있는 부분을 살피는 데 점점 더 많은 시간을 사용한다는 점

탐험 함수를 사용한 Q-러닝

$$Q(s,a) \leftarrow r + \gamma \cdot \max_{a'} f(Q(s',a'), N(s',a'))$$

Q-러닝(5)



근사 Q-러닝과 심층 Q-러닝

- Q-러닝의 주요 문제는 많은 상태와 행동을 가진 대규모(또는 중간 규모)의 MDP에 적용하기 어렵다는 것
- 해결책은 어떤 상태-행동 (s, a) 쌍의 Q-가치를 근사하는 함수 $Q_{\theta}(s, a)$ 를 (파라미터 벡터 θ 로 주어진) 적절한 개수의 파라미터를 사용하여 찾는 것
- 근사 Q-러닝(approximate Q-learning)
- 심층 Q-네트워크(deep Q-network, DQN)
 - Q-가치를 추정하기 위해 사용하는 DNN
- 심층 Q-러닝(deep Q-learning)
 - 근사 Q-러닝을 위해 DQN을 사용하는 것

타깃 Q-가치

$$Q_{\text{target}}(s, a) = r + \gamma \cdot \max_{a'} Q_{\theta}(s', a')$$



심층 Q-러닝 구현(1)

- 심층 Q-네트워크 구현

- 상태-행동 쌍을 입력으로 받고 근사 Q-가치를 출력하는 신경망

```
input_shape = [4] # == env.observation_space.shape
n_outputs = 2     # == env.action_space.n

model = tf.keras.Sequential([
    tf.keras.layers.Dense(32, activation="elu", input_shape=input_shape),
    tf.keras.layers.Dense(32, activation="elu"),
    tf.keras.layers.Dense(n_outputs)
])
```

- 에이전트가 환경을 탐험하도록 만들기 위해 ϵ -그리디 정책을 사용

```
def epsilon_greedy_policy(state, epsilon=0):
    if np.random.rand() < epsilon:
        return np.random.randint(n_outputs) # 랜덤 행동
    else:
        Q_values = model.predict(state[np.newaxis], verbose=0)[0]
        return Q_values.argmax()           # DQN에 따른 최적의 행동
```



심층 Q-러닝 구현(2)

- 재생 버퍼 replay buffer(또는 재생 메모리 replay memory)에 모든 경험을 저장하고 훈련 반복마다 여기에서 랜덤한 훈련 배치를 샘플링
 - 경험과 훈련 배치 사이의 상관관계가 줄어들어 훈련에 큰 도움
 - 덱(double-ended queue, deque)을 사용

```
from collections import deque

replay_buffer = deque(maxlen=2000)
```

- 재생 버퍼에서 경험을 랜덤하게 샘플링하기 위한 함수
 - 이 함수는 경험 원소 6개에 상응하는 넘파이 배열 6개를 반환

```
def sample_experiences(batch_size):
    indices = np.random.randint(len(replay_buffer), size=batch_size)
    batch = [replay_buffer[index] for index in indices]
    return [
        np.array([experience[field_index] for experience in batch])
        for field_index in range(6)
    ] # [states, actions, rewards, next_states, dones, truncateds]
```



심층 Q-러닝 구현(3)

- ϵ -그리디 정책을 사용해 하나의 스텝을 플레이하고 반환된 경험을 재생 버퍼에 저장하는 함수

```
def play_one_step(env, state, epsilon):  
    action = epsilon_greedy_policy(state, epsilon)  
    next_state, reward, done, truncated, info = env.step(action)  
    replay_buffer.append((state, action, reward, next_state, done, truncated))  
    return next_state, reward, done, truncated, info
```



심층 Q-러닝 구현(4)

- 재생 버퍼에서 경험 배치를 샘플링하고 이 배치에서 경사 하강법 한 스텝을 수행하여 DQN을 훈련하는 함수

```
batch_size = 32
discount_factor = 0.95
optimizer = tf.keras.optimizers.Nadam(learning_rate=1e-2)
loss_fn = tf.keras.losses.mean_squared_error

def training_step(batch_size):
    experiences = sample_experiences(batch_size)
    states, actions, rewards, next_states, done, truncated = experiences
    next_Q_values = model.predict(next_states, verbose=0)
    max_next_Q_values = next_Q_values.max(axis=1)
    runs = 1.0 - (done | truncated) # 에피소드가 중지되거나 종료되지 않음
    target_Q_values = rewards + runs * discount_factor * max_next_Q_values
    target_Q_values = target_Q_values.reshape(-1, 1)
    mask = tf.one_hot(actions, n_outputs)
    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(loss_fn(target_Q_values, Q_values))

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

1

2

3

4

5



심층 Q-러닝 구현(5)

- 모델 훈련
 - 최대 스텝 200번으로 이루어진 에피소드 600개를 실행

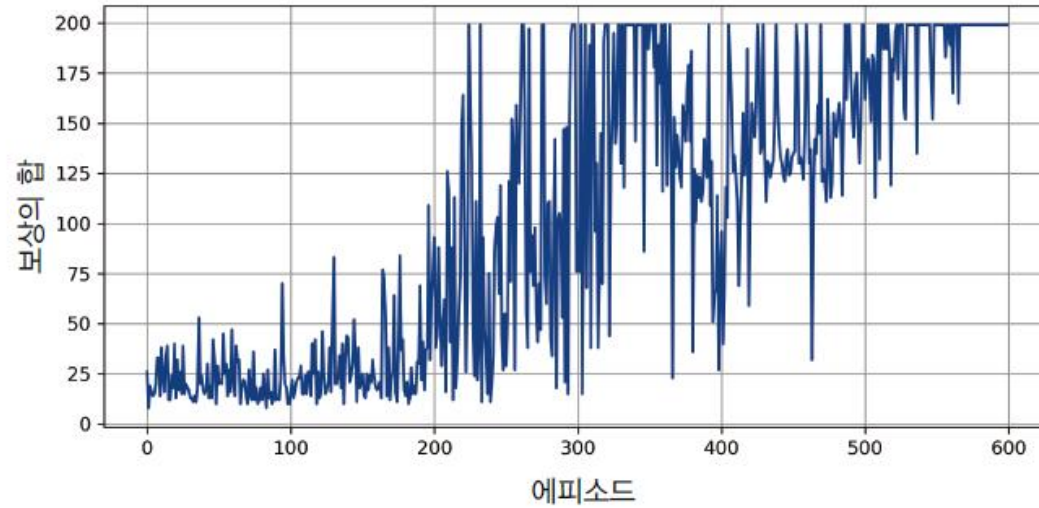
```
for episode in range(600):
    obs, info = env.reset()
    for step in range(200):
        epsilon = max(1 - episode / 500, 0.01)
        obs, reward, done, truncated, info = play_one_step(env, obs, epsilon)
        if done or truncated:
            break

    if episode > 50:
        training_step(batch_size)
```

심층 Q-러닝 구현(6)



- 각 에피소드에서 에이전트가 얻은 총 보상



심층 Q-러닝 알고리즘의 학습 곡선



심층 Q-러닝의 변형(1)

고정 Q-가치 타깃

- 피드백 순환 과정은 네트워크를 불안정하게 만들어 발산, 진동, 동결 등의 문제가 발생
 - 이 문제를 해결하기 위해 딥마인드 연구자들은 2013년 논문에서 한 개가 아닌 두 개의 DQN을 사용
 - 첫 번째 DQN은 각 스텝에서 학습하고 에이전트를 움직이는 데 사용하는 온라인 모델(online model)
 - 두 번째는 타깃을 정의하기 위해서만 사용하는 타깃 모델(target model)
 - 타깃 모델은 온라인 모델의 단순한 복사본

```
target = tf.keras.models.clone_model(model) # 모델 구조 복사
target.set_weights(model.get_weights()) # 가중치 복사
```

- training_step() 함수에서 다음 상태의 Q-가치를 계산할 때 온라인 모델 대신 타깃 모델을 사용하도록 한 줄을 변경

```
next_Q_values = target.predict(next_states, verbose=0)
```

- 훈련 반복에서 일정한 간격(예 – 50 에피소드)으로 온라인 모델의 가중치를 타깃 모델로 복사

```
if episode % 50 == 0:
    target.set_weights(model.get_weights())
```



심층 Q-러닝의 변형(2)

더블 DQN

- 더블 DQN(double DQN)
 - DQN 알고리즘을 개선하여 성능과 훈련의 안정성을 향상
 - 다음 상태에서 최선의 행동을 선택할 때 타깃 모델 대신 온라인 모델을 사용하도록 제안
 - 타깃 모델은 최선의 행동에 대한 Q-가치를 추정할 때만 사용
 - 수정된 training_step() 함수

```
def training_step(batch_size):
    experiences = sample_experiences(batch_size)
    states, actions, rewards, next_states, done, truncateds = experiences
    next_Q_values = model.predict(next_states, verbose=0) # ≠ target.predict()
    best_next_actions = next_Q_values.argmax(axis=1)
    next_mask = tf.one_hot(best_next_actions, n_outputs).numpy()
    max_next_Q_values = (target.predict(next_states, verbose=0) * next_mask
                        ).sum(axis=1)
    [...] # 나머지 코드는 이전과 동일합니다.
```



심층 Q-러닝의 변형(3)

우선 순위 기반 경험 재생

- 중요도 샘플링(importance sampling, IS) 또는 우선 순위 기반 경험 재생(prioritized experience replay, PER)
 - 재생 버퍼에서 경험을 균일하게 샘플링하는 것이 아니라 중요한 경험을 더 자주 샘플링
 - 샘플이 중요한 경험에 편향되어 있으므로 훈련하는 동안 중요도에 따라 경험의 가중치를 낮춰서 이 편향을 보상



심층 Q-러닝의 변형(4)

듀얼링 DQN

- 듀얼링 DQN(dueling DQN, DDQN)
 - 모델이 상태의 가치와 가능한 각 행동의 이익을 모두 추정
 - 최선의 행동은 이익이 0이기 때문에 모델이 예측한 모든 이익에서 모든 최대 이익을 뺌
 - 함수형 API로 구현한 간단한 듀얼링 DQN 모델

```
input_states = tf.keras.layers.Input(shape=[4])
hidden1 = tf.keras.layers.Dense(32, activation="elu")(input_states)
hidden2 = tf.keras.layers.Dense(32, activation="elu")(hidden1)
state_values = tf.keras.layers.Dense(1)(hidden2)
raw_advantages = tf.keras.layers.Dense(n_outputs)(hidden2)
advantages = raw_advantages - tf.reduce_max(raw_advantages, axis=1, keepdims=True)
Q_values = state_values + advantages
model = tf.keras.Model(inputs=[input_states], outputs=[Q_values])
```



다른 강화 학습 알고리즘(1)

알파고

- 알파고(AlphaGo)는 심층 신경망에 기반한 몬테 카를로 트리 검색(Monte Carlo tree search, MCTS)의 변형을 사용하여 바둑 게임에서 인간 챔피언을 이김

액터-크리틱

- 액터-크리틱(actor-critic)은 정책 그레디언트와 심층 Q-네트워크를 결합한 강화 학습 알고리즘

A3C

- 복사된 다른 환경을 탐색하면서 병렬로 여러 에이전트가 학습하는 중요한 액터-크리틱 변형

A2C

- 비동기성을 제거한 A3C 알고리즘의 변형

SAC

- 액터-크리틱 변형. 이 모델은 보상뿐만 아니라 행동의 엔트로피를 최대화하도록 훈련

PPO

- 큰 가중치 업데이트를 피하기 위해 손실 함수를 클리핑하는 A2C 기반의 알고리즘



다른 강화 학습 알고리즘(2)

호기심 기반 탐색

- 강화 학습에서 계속 일어나는 문제는 보상의 희소성
- 보상없이 호기심만으로 환경을 탐색

개방형 학습

- 개방형 학습(Open-ended learning, OEL)의 목표는 보통 차례대로 생성되는 새롭고 흥미로운 작업을 끊임없이 학습할 수 있는 에이전트를 훈련
- POET 알고리즘
- 커리큘럼 학습(curriculum learning)



연습문제(1)

1. 강화 학습을 어떻게 정의할 수 있나? 지도 학습이나 비지도 학습과 어떻게 다른가?
2. 이 장에서 언급하지 않은 가능한 RL 애플리케이션을 세 가지 생각하기. 각 애플리케이션의 환경은 무엇인가? 에이전트는 무엇인가? 가능한 행동은 무엇인가? 보상은 무엇인가?
3. 할인 계수는 무엇인가? 할인 계수를 바꾸면 최적의 정책이 바뀔 수 있나?
4. 강화 학습 에이전트의 성능은 어떻게 측정할 수 있나?
5. 신용 할당 문제가 무엇인가요? 언제 이런 문제가 발생하나? 어떻게 이를 감소시킬 수 있나?
6. 재생 메모리를 사용하는 이유는 무엇인가?
7. 오프-폴리시 RL 알고리즘이 무엇인가?
8. 정책 그래디언트를 사용해 OpenAI Gym의 LunarLander-v2 환경을 해결해보기

연습문제(2)

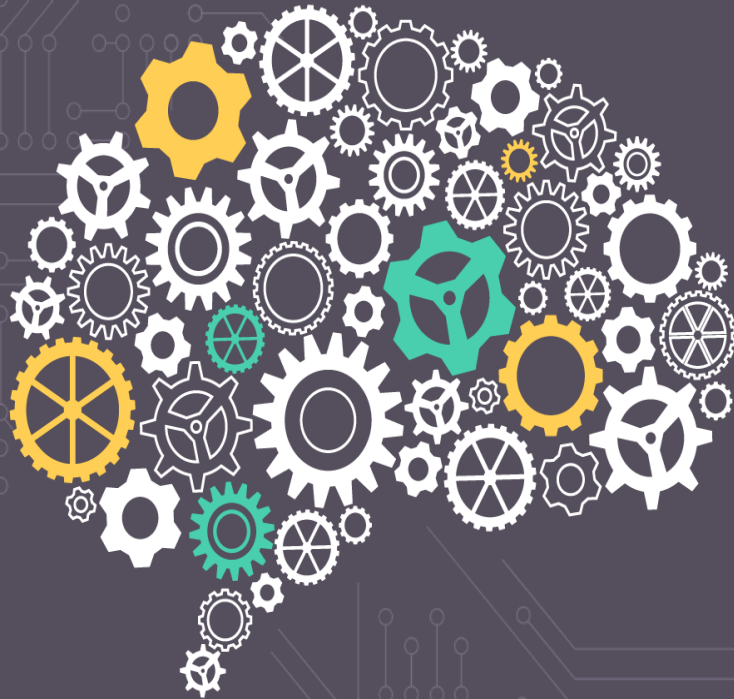


9. 더블 듀얼링 DQN을 사용하여 유명한 아타리 <브레이크아웃> 게임("ALE/Breakout-v5")에서 사람의 수준을 뛰어 넘을 수 있는 에이전트를 훈련. 관측은 이미지. 작업을 단순화하기 위해 흑백으로 변환(채널 축을 따라 평균).

그다음 재생할 수 있을만큼 크지만 그 이상이 되지 않도록 자르고 다운샘플링. 개별 이미지만으로는 공과 패들 (paddle)이 어느 방향으로 가고 있는지 알 수 없으므로 두세 개의 연속된 이미지를 병합하여 각 상태를 만들어야 함. 마지막으로 DQN은 합성곱 층으로 대부분 구성되어야 함

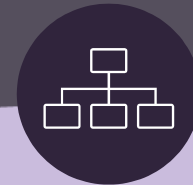
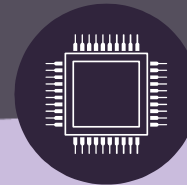
10. 10만 원 정도 여유가 있다면 라즈베리 파이3와 저렴한 로봇틱스 구성품을 구입해 텐서플로를 설치하고 실행할 수 있음!

예를 들어 루카스 비월드의 재미있는 포스트(<https://homl.info/2>)를 참고하거나, GoPiGo42나 BrickPi43를 둘러보기. 간단한 작업부터 시작. 예를 들어 (조도 센서가 있다면) 로봇이 밝은 쪽으로 회전하거나 (초음파 센서가 있다면) 가까운 물체가 있는 쪽으로 움직이도록 해보기. 그다음 딥러닝을 사용해보기. 예를 들어 로봇에 카메라가 있다면 객체 탐지 알고리즘을 구현해 사람을 감지하고 가까이 다가가게 만들 수 있음. 강화 학습을 사용해 목표를 달성하기 위해 모터 사용법을 스스로 학습할 수도 있음



Thank you!

See you next time.



담당교수 : 유 현 주

comjoo@uok.ac.kr

