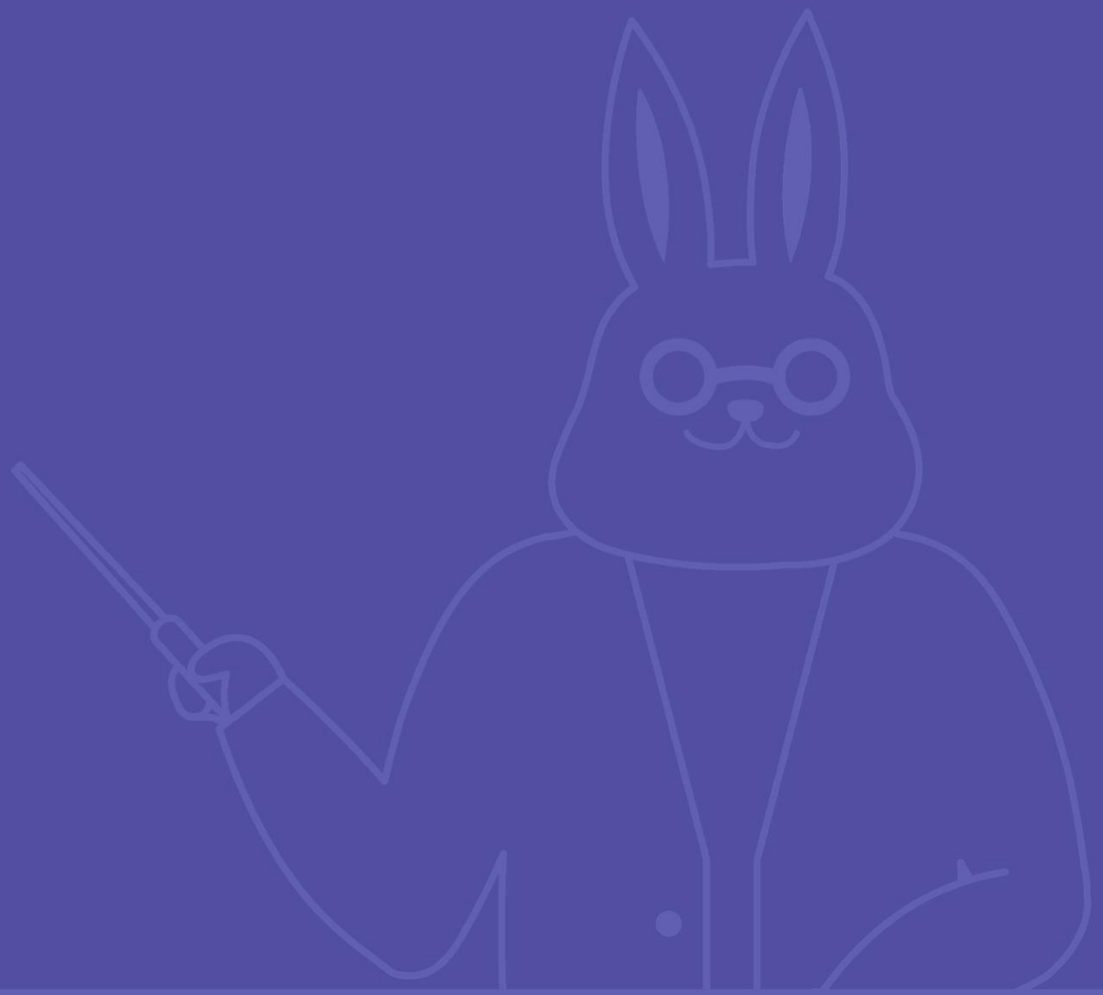
딥러닝 모델 활용

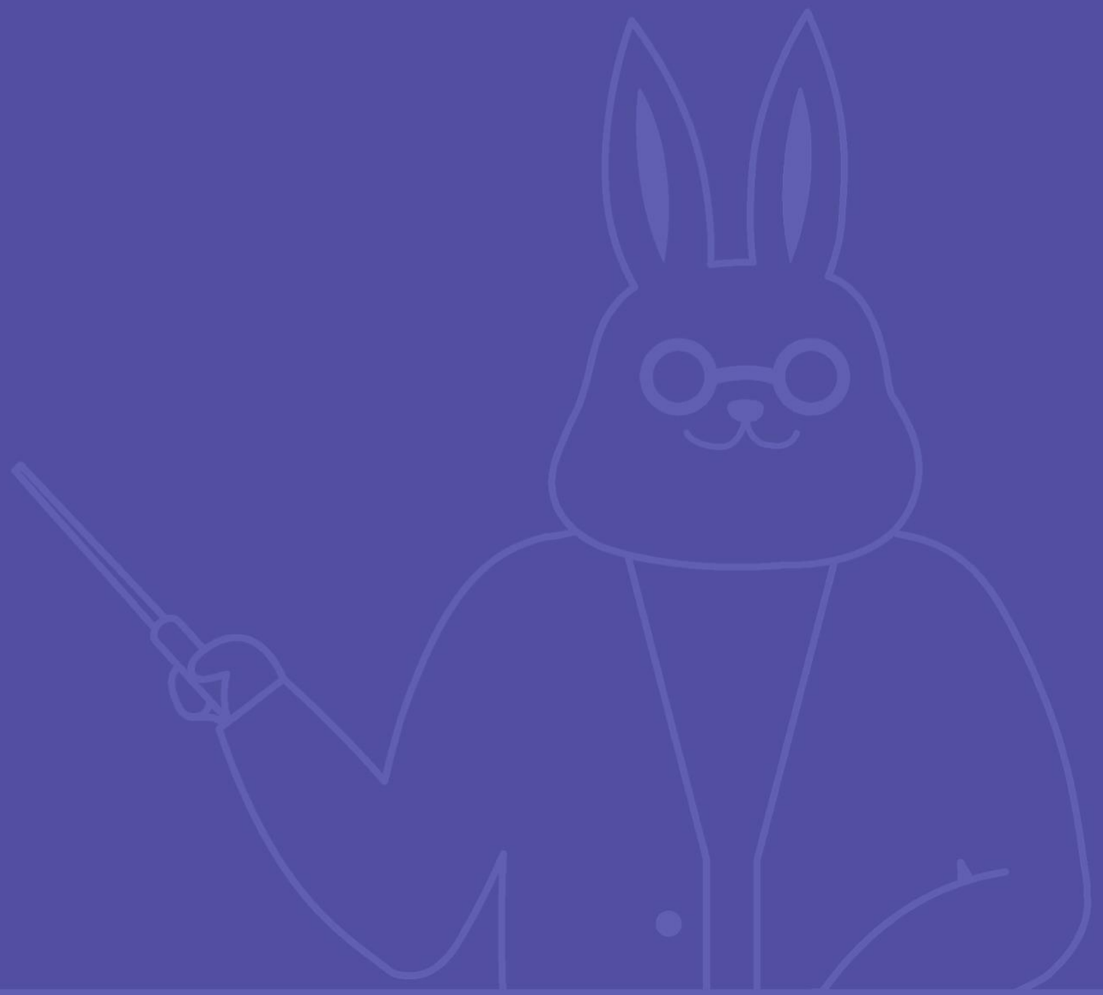
02 모델 서비스하기



목차

01

모델 저장하고 불러오기



모델의 구성요소

모델의 구조

- 레이어의 종류와 형태
- 입력 값의 형태

가중치 값

- 각 레이어의 행렬에 저장된 실제 float32 실수 값들
- 모델의 학습 = Loss값이 낮아지도록 가중치의 값을 수정하는 과정의 연속
- 같은 모델도 가중치 값에 따라 성능이 달라짐

Compile 정보

- Optimizer의 종류, Learning Rate(lr), 사용한 Loss Function 정보

모델의 저장 형식

H5 Format

- 과거 Keras에서 사용하던 저장 방식
- 모델의 구조와 가중치를 포함한 정보들을 저장
- 사용자 정의 레이어와 손실함수는 저장하지 않음

SavedModel

- 최근에 사용하는 Tensorflow 표준 저장 형식
- 모델의 구조와 가중치를 포함한 정보들을 저장
- 사용자 정의 레이어와 손실함수까지 모두 저장

SavedModel 활용 방법

- 모델 저장

```
model.fit(x, y)
#'my_model' 이라는 이름의 SavedModel 폴더를 생성
model.save("my_model")
```

- 저장된 모델 사용

```
# 'my_model' 이라는 이름의 SavedModel을 불러옴
loaded_model = keras.models.load_model("my_model")
```

이어서 학습하기

- SavedModel형식은 모델의 모든 정보를 저장하고 불러오는 방식
- 불러온 모델을 그대로 학습을 마저 진행
  - initial_epoch과 epoch을 조절하여 이어서 학습을 진행

```
model.compile(...)
model.fit(x, y)
# 모델을 compile하고 학습까지 진행하고 저장
model.save("my_model")

loaded_model = keras.models.load_model("my_model")
# 저장한 시점의 compile 정보와 학습 상태까지 불러옴
# 다시 compile하지 않아도 바로 이어서 학습 가능
# 21 epoch부터 40 epoch까지 계속 학습
loaded_model.fit(x, y, initial_epoch = 20, epochs = 40)
```

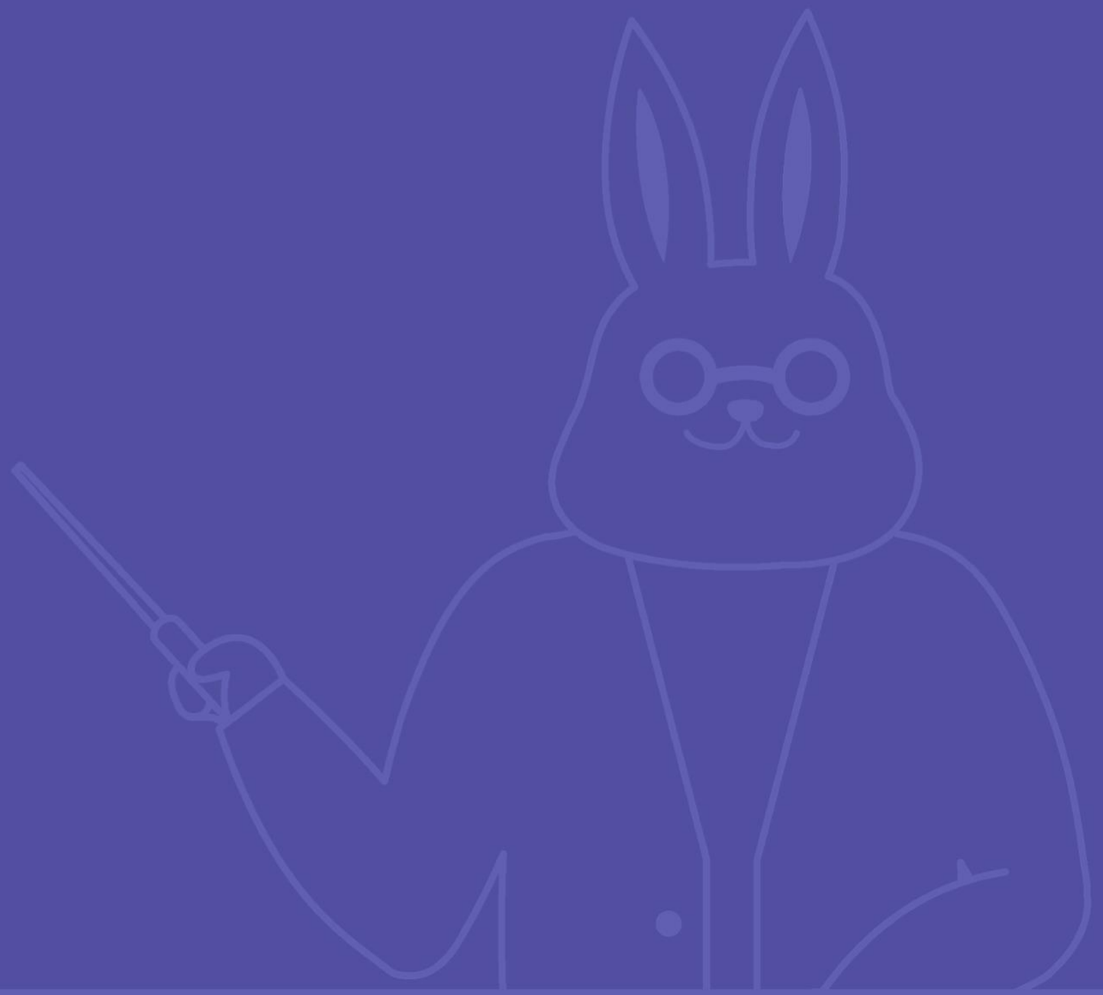
Checkpoint 불러오기

- Checkpoint 콜백함수의 인수 중 save_weights_only를 False로 설정
- 원하는 epoch의 체크포인트 경로를 전달
- initial_epoch과 epoch을 조절하여 이어서 학습을 진행

```
# 모델을 학습하면서 Checkpoint를 저장
# 20epoch 이후 저장된 체크포인트를 불러옴
loaded_model = keras.models.load_model("checkpoints/cp-0020.ckpt")
# 21 epoch부터 40 epoch까지 계속 학습
loaded_model.fit(x, y, initial_epoch = 20, epochs = 40)
```

02

모델 서비스 방법



자바스크립트를 이용한 모델 서비스

Tensorflow.js

- 자바스크립트 Tensorflow 라이브러리
- 브라우저 또는 Node.js에서 학습된 모델을 사용가능

Tensorflow.js에서 사용하는 과정

- 모델의 학습을 Tensorflow에서 진행
- Python에서 학습된 모델을 Tensorflow.js에 맞는 형식으로 변환
- Tensorflow.js에서 모델의 동작을 js로 작성하여 서비스

Tensorflow.js 형식으로 변환하는 방법

- tensorflowjs를 사용하여 변환하여 저장
- target_dir에 json파일을 포함한 변환된 모델이 저장됨

tensorflowjs

```
import tensorflowjs as tfjs                    # pip를 통해 tensorflowjs 설치 필요
def train(...):
model = keras.models.Sequential()
...
model.compile(...)
model.fit(...)                                 # 모델 정의 및 학습 과정
                                               # model을 변환하여 target_dir에 저장
tfjs.converters.save_keras_model(model, target_dir)
```

Tensorflowjs로 변환한 모델 사용방법

- model.json 파일의 URL을 제공하여 TensorFlow.js에 모델을 로드

javascript

```
import * as tf from '@tensorflow/tfjs';
const model = await
tf.loadLayersModel('https://foo.bar/tfjs_artifacts/model.json');
```

- 모델을 학습, 추론하기

javascript

```
const example = tf.fromPixels(webcamElement); // 웹캠 Element를 사용한다고 가정
const prediction = model.predict(example);    // 예측에 사용
```

Flask에서 서비스하기

- 학습한 모델을 불러오기 : SavedModel 형식의 모델을 로드

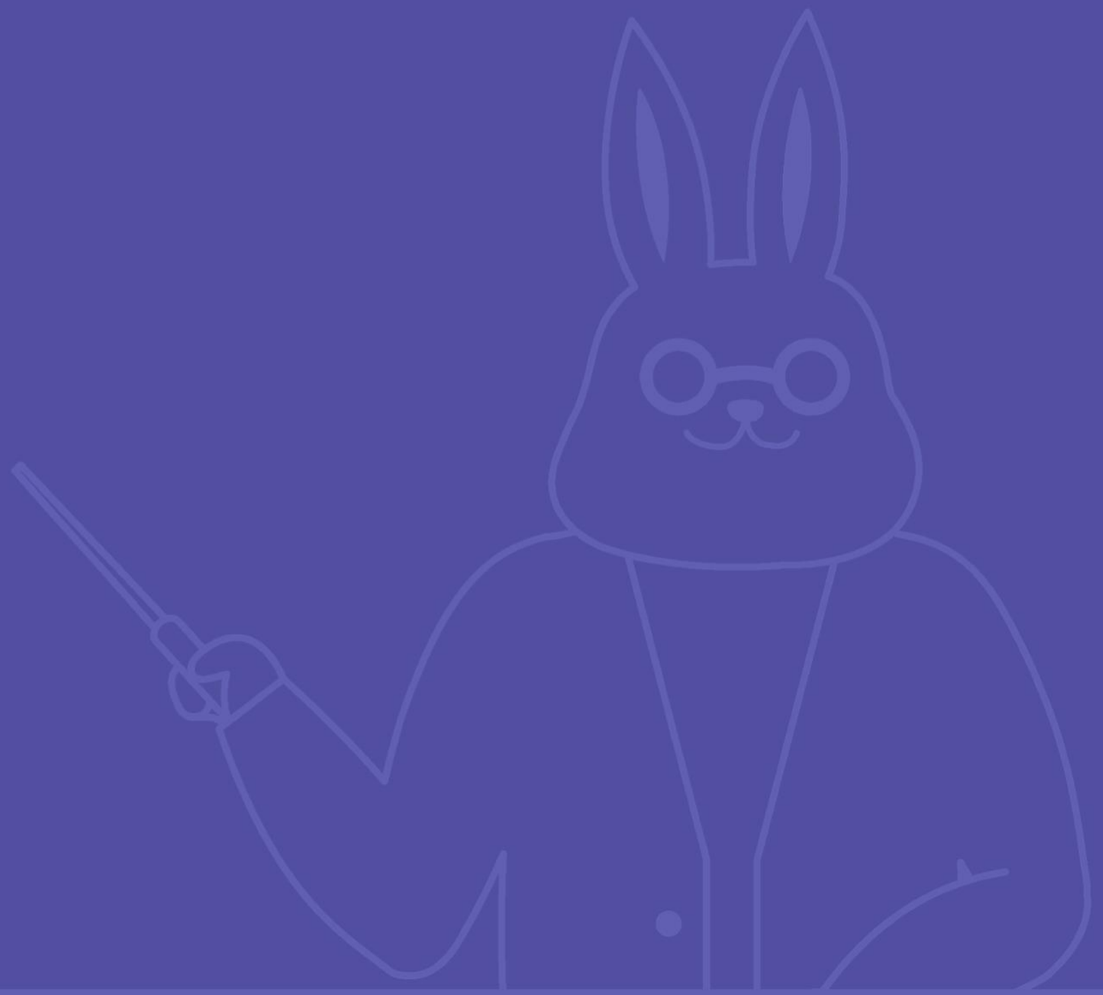
```
if __name__ == '__main__':
  # 모델 로드
    model = tf.keras.models.load_model("my_model")
    # Flask 서비스 시작
    app.run(host='localhost', port=8080)
```

- 모델의 사용은 그대로 predict를 사용

```
res = model.predict([inputdata])
# res에서 정보를 추출하여 서비스에 활용
```

03

서버 안정화 처리



서버 안정화 처리의 필요성

딥러닝 모델 서비스

- 모델의 추론과정은 일반적인 연산에 비해 처리 시간이 길고 자원도 많이 소모
- GPU자원을 사용하는 경우 GPU를 병렬적으로 사용하기 위한 처리가 필요
- 사용자의 요청에 따라 계속 연산을 처리하면 서비스가 종료될 수 있음

서비스 안정화

- 사용자의 요청을 거절하더라도 서비스가 종료되지 않도록 자원관리가 필요
- 서버의 연산 성능, 처리중인 작업의 수를 고려한 설계가 요구됨

실행 가능한 작업 제한

서버에서 동시에 진행가능한 작업의 수를 제한하여 안정화

구현 방법의 예시

- 동시에 처리 가능한 최대 작업의 수를 상수로 정의 (max_works)
- 전역변수를 이용하여 진행중인 작업의 수를 저장 (num_works)
- 사용자의 요청이 왔을 때, 진행중인 작업의 수 비교 (max_works> num_works)
  - 작업의 수가 최대일 때: 사용자에게 잠시 후 시도해달라는 안내 메시지를 출력하고 거절
  - 작업의 수가 최대가 아닐 때: 작업을 수락하고 num_works를 1증가 시킴
- 작업이 완료되면 작업 수를 1 감소 (num_works -= 1)

실행 가능한 작업 제한

장점

- 비교적 간단하게 구현이 가능
- 가벼운 모델만 사용할 때, 서비스가 다운되는 현상은 방지할 수 있음

단점

- 사용자가 작업을 예약하는 등의 처리는 어려움
- 작업이 매우 오래 걸리는 모델의 경우 응답까지 시간이 오래걸림
- 대규모 서비스에 적용하기는 부적절함

작업 큐를 이용한 비동기 처리

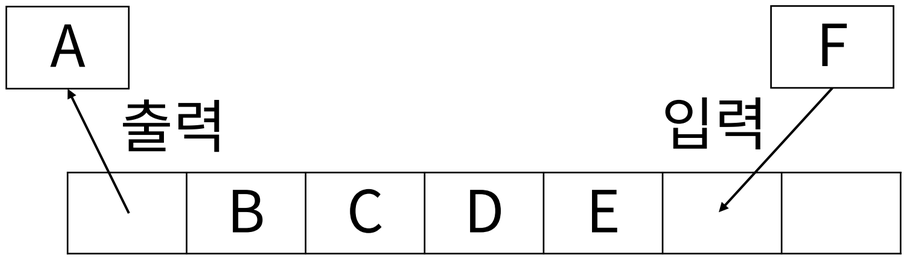
- 사용자와 상호작용을 관리하는 프로세스와 작업을 처리하는 Worker 프로세스를 분리
- 각 프로세스는 큐를 이용하여 상호작용함

- 유사한 방식이 활용되는 분야
  - 안드로이드 GUI의 이벤트시스템
  - Windows의 메시지 큐와 윈도우 프로시저
  - 그래픽 처리 분야의 렌더링 큐
  - 프린터의 스풀링작업
  - Tensorflow의 데이터를 불러오기 위한 workers 옵션

자료구조 큐 다시보기

큐(Queue)

- 먼저 입력된 값이 먼저 출력되는 자료구조
- 입력된 시간 순서대로 처리해야 할 필요가 있는 상황에 이용

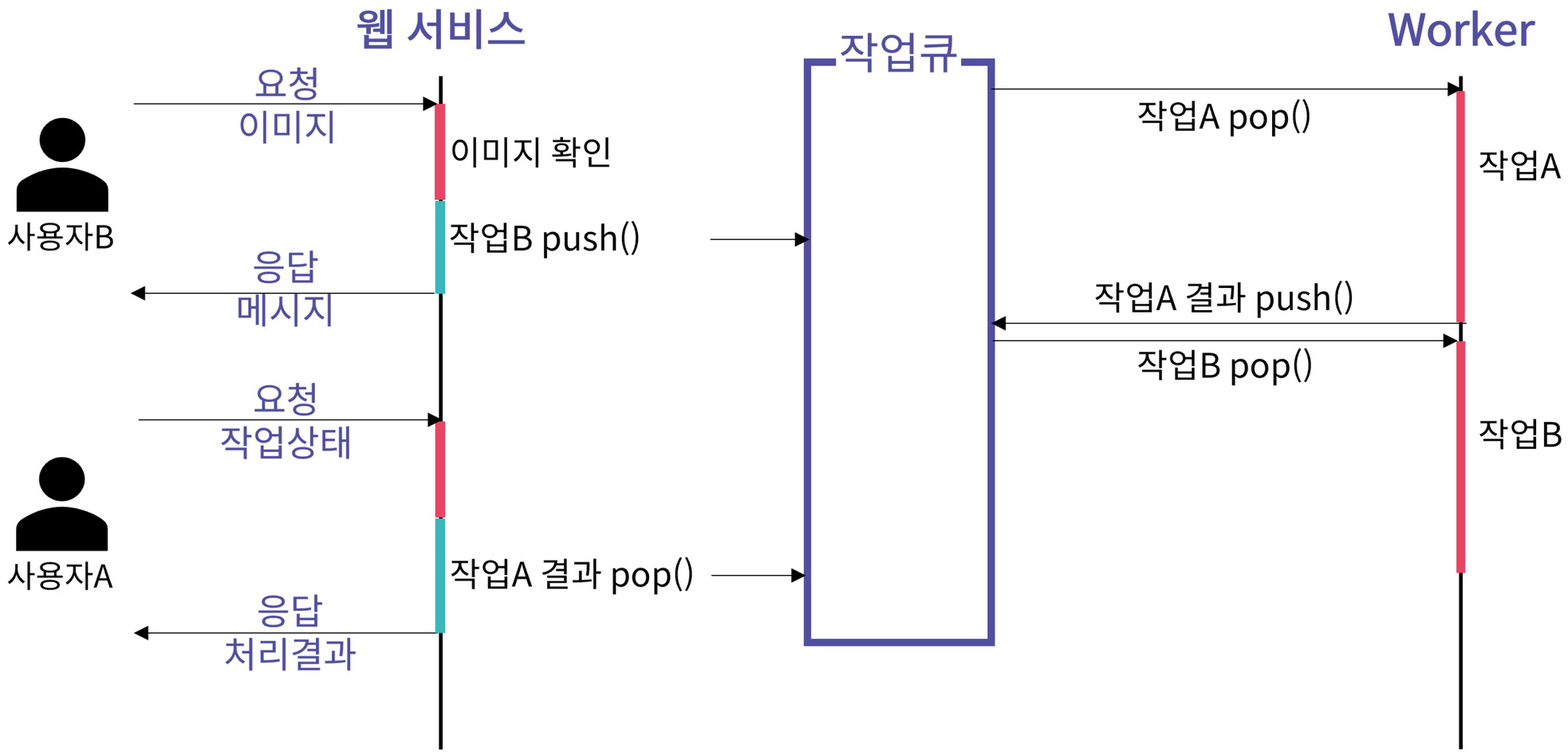


큐를 이용한 처리

- 메인 프로세스
  - 사용자의 요청을 처리하는 프로세스
  - 처리 시간이 짧은 처리만 담당하여 빠르게 사용자의 요청에 반응
  - 이 프로세스는 절대로 종료되지 않도록 주의하면서 처리
  - 사용자의 요청이 들어오면 큐에 작업을 추가하고 사용자에게 응답

- Worker 프로세스
  - 시간이 오래걸리거나 복잡한 작업을 단순하게 계속 처리
  - 큐에 있는 작업을 불러와 처리하는 과정만 계속 반복
  - 연산에 필요한 자원은 미리 할당한 상태로 작업
  - 이 프로세스가 종료되더라도 메인 서비스는 계속 동작

작업 큐를 이용한 비동기 처리 구현의 예



작업 큐와 비동기 처리

장점

- Worker 프로세스가 오류로 종료되어도 웹 서비스는 계속 동작
- 딥러닝 모델의 교체, 가중치 업데이트는 Worker 프로세스만 수정하면 반영
- 웹 서비스는 교체 작업 도중에도 계속 동작하여 사용자에게 상황을 안내할 수 있음
- 딥러닝 모델 담당자는 Worker 프로세스만 관리하여 분업이 쉬움

사용자의 로그인 정보와 작업을 매칭

- 사용자가 처리를 요청하고 브라우저를 닫아도 진행상황이 유지되고 결과를 받을 수 있음
- 요금을 부담하면 우선순위를 조정하거나 더 큰 이미지를 요청할 수 있도록 서비스 가능
- 대부분 무거운 딥러닝 모델을 사용하는 서비스가 사용하는 방식

크레딧

/* elice */

코스 매니저

-

콘텐츠 제작자

김승환

강사

김승환

감수자

-

디자이너

강혜정

연락처

TEL

070-4633-2015

WEB

https://elice.io

E-MAIL

contact@elice.io