CHAPTER 08 이미지를 위한 인공 신경망

혼자 공부하는 머신러닝+딥러닝(개정판)

한국공학대학교 게임공학과
이재영

시작하기전에

지은이 / 박해선

기계공학을 전공했으나 졸업 후엔 줄곧 코드를 읽고 쓰는 일을 했다. 머신러닝과 딥러닝에 관한 책을 집필하고 번역하면서 소프트웨어와 과학의 경계를 흥미롭게 탐험하고 있다.

『핸즈온 머신러닝 2판』(한빛미디어, 2020)을 포함해서 여러 권의 머신러닝, 딥러닝 책을 우리말로 옮겼고 『Do it! 딥러닝 입문』(이지스퍼블리싱, 2019)을 집필했다.

- **교재의 모든 코드는 웹 브라우저에서 파이썬 코드를 실행할 수 있는 구글 코랩(Colab)을 사용하여 작성했습니다.**
- **사용할 실습 환경은 네트워크에 연결된 컴퓨터와 구글 계정입니다.**

학습 로드맵

머신러닝편 **01~06장**

딥러닝만 먼저 배우고
싶다면 01~04장을 읽은 후
07장으로 건너뛰어도 좋습니다.

딥러닝편 **07~10장**

07장을 읽은 후 08장과 09장은
순서대로 읽지 않아도 괜찮습니다.
10장을 읽기 전에 07장과 09장을
읽는 것이 좋습니다.

난이도 ●●●●●

START

01 나의 첫 머신러닝 ●○○○○

02 데이터 다루기 ●●○○○

03 회귀 알고리즘과 모델 규제

2번 보기

04 다양한 분류 알고리즘 ●●○○○

05 트리 알고리즘 ●●●○○

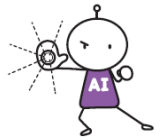
06 비지도 학습 ●●●○○

07 딥러닝을 시작합니다 ●●●●○

08 이미지를 위한 인공 신경망 ●●●●●

09 텍스트를 위한 인공 신경망 ●●●●●

10 언어 모델을 위한 신경망 ●●●●●



GOAL

이 책의 학습 목표

- **CHAPTER 01: 나의 첫 머신러닝**
  - 인공지능, 머신러닝, 딥러닝의 차이점을 이해합니다.
  - 구글 코랩 사용법을 배웁니다.
  - 첫 번째 머신러닝 프로그램을 만들고 머신러닝의 기본 작동 원리를 이해합니다.
- **CHAPTER 02: 데이터 다루기**
  - 머신러닝 알고리즘에 주입할 데이터를 준비하는 방법을 배웁니다.
  - 데이터 형태가 알고리즘에 미치는 영향을 이해합니다.
- **CHAPTER 03: 회귀 알고리즘과 모델 규제**
  - 지도 학습 알고리즘의 한 종류인 회귀 알고리즘에 대해 배웁니다.
  - 다양한 선형 회귀 알고리즘의 장단점을 이해합니다.
- **CHAPTER 04: 다양한 분류 알고리즘**
  - 로지스틱 회귀, 확률적 경사 하강법과 같은 분류 알고리즘을 배웁니다.
  - 이진 분류와 다중 분류의 차이를 이해하고 클래스별 확률을 예측합니다.
- **CHAPTER 05: 트리 알고리즘**
  - 성능이 좋고 이해하기 쉬운 트리 알고리즘에 대해 배웁니다.
  - 알고리즘의 성능을 최대화하기 위한 하이퍼파라미터 튜닝을 실습합니다.
  - 여러 트리를 합쳐 일반화 성능을 높일 수 있는 앙상블 모델을 배웁니다.

이 책의 학습 목표

- **CHAPTER 06: 비지도 학습**
  - 타깃이 없는 데이터를 사용하는 비지도 학습과 대표적인 알고리즘을 소개합니다.
  - 대표적인 군집 알고리즘인 k-평균과 DBSCAN을 배웁니다.
  - 대표적인 차원 축소 알고리즘인 주성분 분석(PCA)을 배웁니다.
- **CHAPTER 07: 딥러닝을 시작합니다**
  - 딥러닝의 핵심 알고리즘인 인공 신경망을 배웁니다.
  - 대표적인 인공 신경망 라이브러리인 텐서플로와 케라스를 소개합니다.
  - 인공 신경망 모델의 훈련을 돕는 도구를 익힙니다.
- **CHAPTER 08: 이미지를 위한 인공 신경망**
  - 이미지 분류 문제에 뛰어난 성능을 발휘하는 합성곱 신경망의 개념과 구성 요소에 대해 배웁니다.
  - 케라스 API로 합성곱 신경망을 만들어 패션 MNIST 데이터에서 성능을 평가해 봅니다.
  - 합성곱 층의 필터와 활성화 출력을 시각화하여 합성곱 신경망이 학습한 내용을 고찰해 봅니다.
- **CHAPTER 09: 텍스트를 위한 인공 신경망**
  - 텍스트와 시계열 데이터 같은 순차 데이터에 잘 맞는 순환 신경망의 개념과 구성 요소에 대해 배웁니다.
  - 케라스 API로 기본적인 순환 신경망에서 고급 순환 신경망을 만들어 영화 감상평을 분류하는 작업에 적용해 봅니다.
  - 순환 신경망에서 발생하는 문제점과 이를 극복하기 위한 해결책을 살펴봅니다.
- **CHAPTER 10: 언어 모델을 위한 신경망**
  - 어텐션 메커니즘과 트랜스포머에 대해 배웁니다.
  - 트랜스포머로 상품 설명 요약하기를 학습합니다.
  - 대규모 언어 모델로 텍스트 생성하기를 익힙니다.

Contents

- CHAPTER 08: 이미지를 위한 인공 신경망

CHAPTER 08 이미지를 위한 인공 신경망

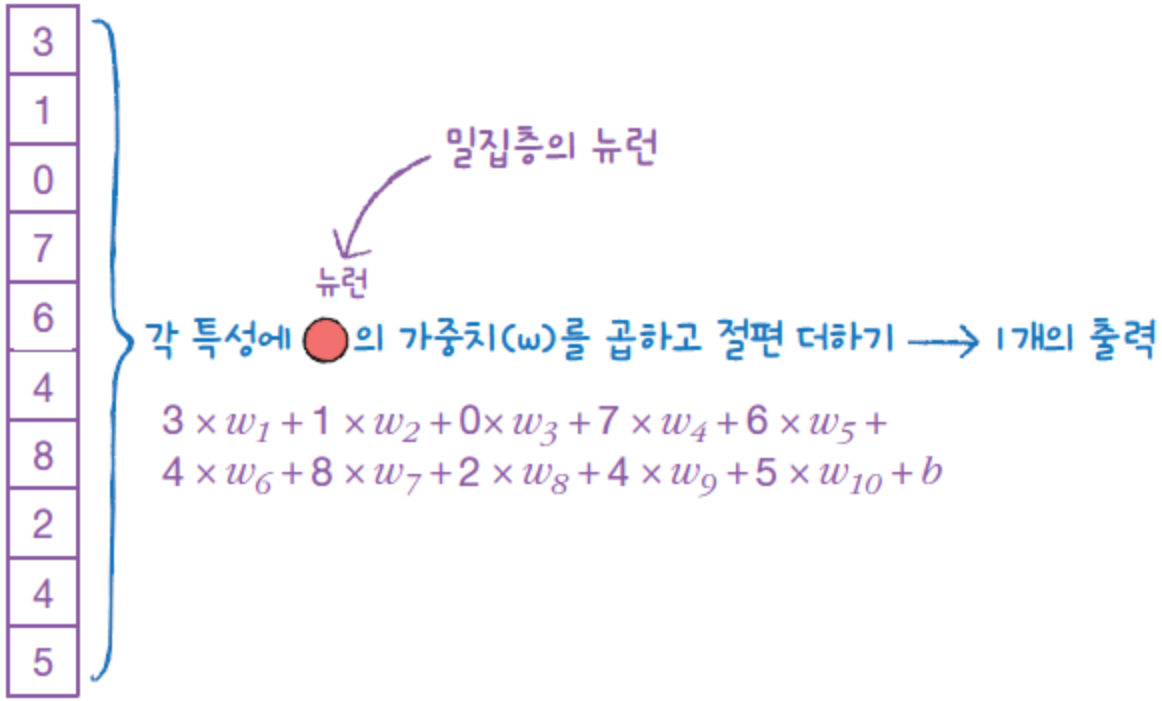
패션 럭키백의 정확도를 높입니다!

- 이미지 분류 문제에 뛰어난 성능을 발휘하는 합성곱 신경망의 개념과 구성 요소를 이해합니다.
- 케라스 API로 합성곱 신경망을 만들어 패션 MNIST 데이터에서 성능을 평가합니다.
- 합성곱 층의 필터와 활성화 출력을 시각화하여 합성곱 신경망이 학습한 내용을 고찰합니다.

SECTION 8-1 합성곱 신경망의 구성 요소(1)

- 합성곱(convolution)
  - 합성곱은 마치 입력 데이터에 마법의 도장을 찍어서 유용한 특성만 드러나게 하는 것과 같음
  - 합성곱의 동작 원리

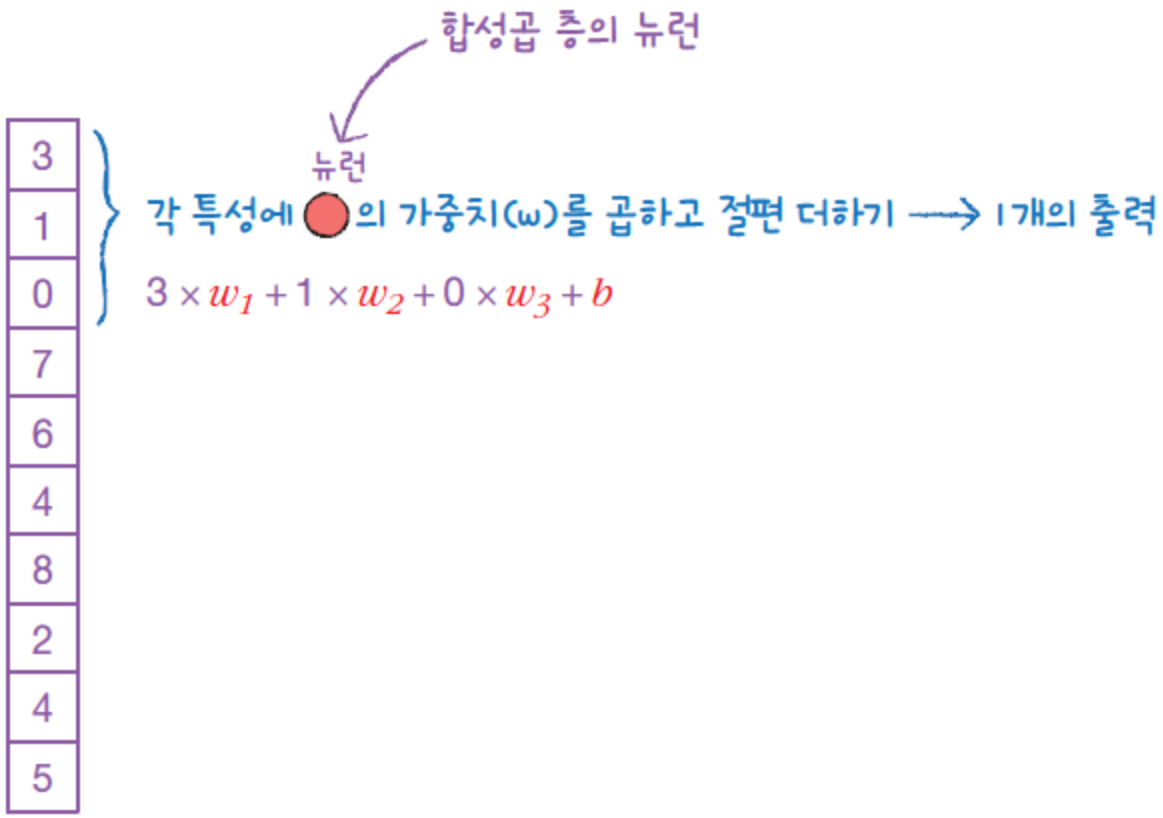
1) 7장에서 사용한 밀집층에는 뉴런마다 입력 개수만큼의 가중치가 존재. 즉 모든 입력에 가중치를 곱함



SECTION 8-1 합성곱 신경망의 구성 요소(2)

- 합성곱(convolution)
  - 합성곱의 동작 원리

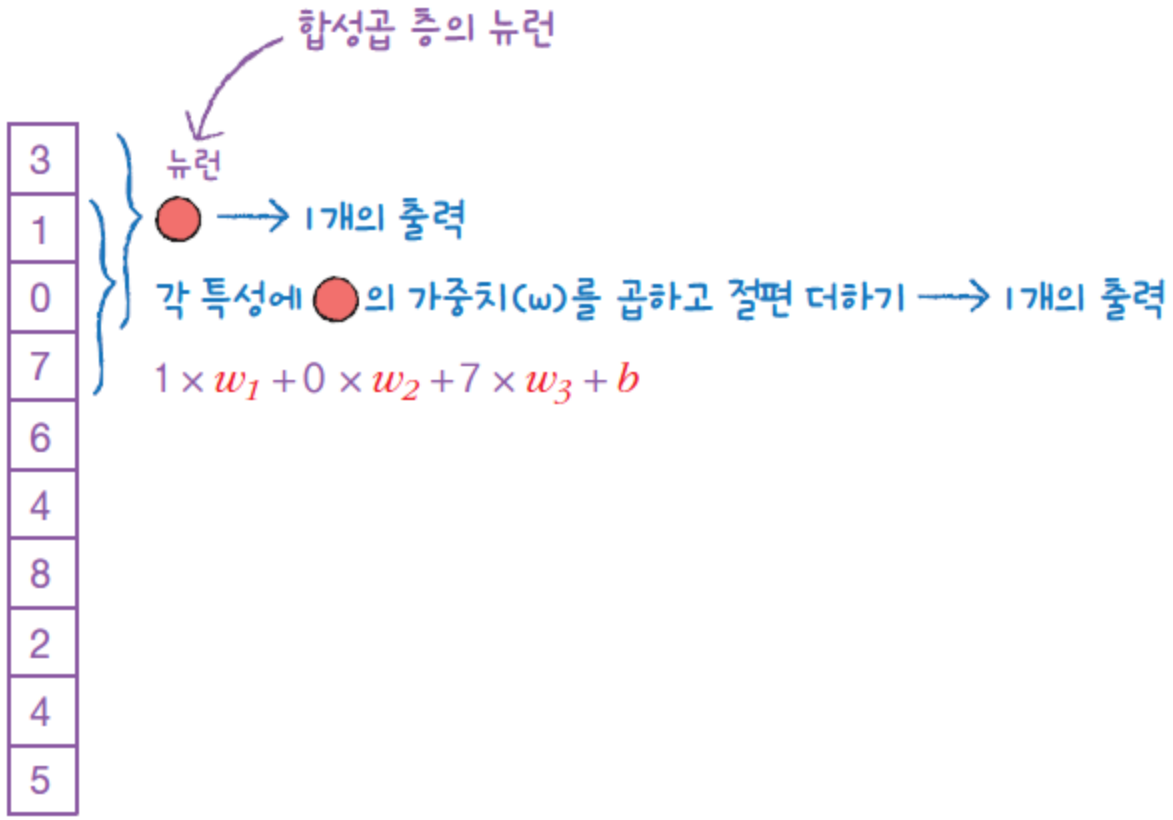
2) 합성곱은 밀집층의 계산과 다르게, 입력 데이터 전체에 가중치를 적용하지 않고 일부에 가중치를 곱함
   - 여기에서는 이 뉴런이 3개의 가중치를 가진다고 가정



SECTION 8-1 합성곱 신경망의 구성 요소(3)

- 합성곱(convolution)
  - 합성곱의 동작 원리

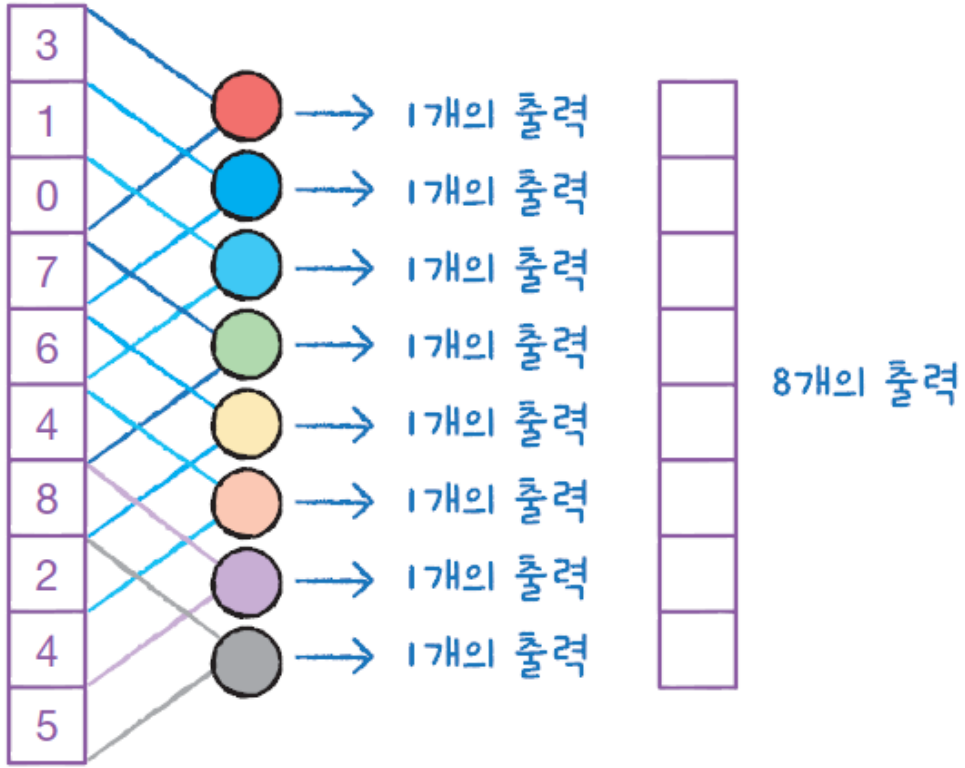
3) 가중치 w1~w3이 입력의 처음 3개 특성과 곱해져 1개의 출력을 만들고, 이 뉴런이 한 칸 아래로 이동해 두 번째부터 네 번째 특성과 곱해져 새로운 출력을 만듦



SECTION 8-1 합성곱 신경망의 구성 요소(4)

- 합성곱(convolution)
  - 합성곱의 동작 원리

4) 첫 번째 합성곱에 사용된 가중치 w1~w3과 절편 b가 두 번째 합성곱에도 동일하게 사용됨
이렇게 한 칸씩 아래로 이동하면서 출력을 만드는 것이 합성곱



▲ 뉴런의 가중치가 3개이기 때문에 모두 8개의 출력이 만들어짐

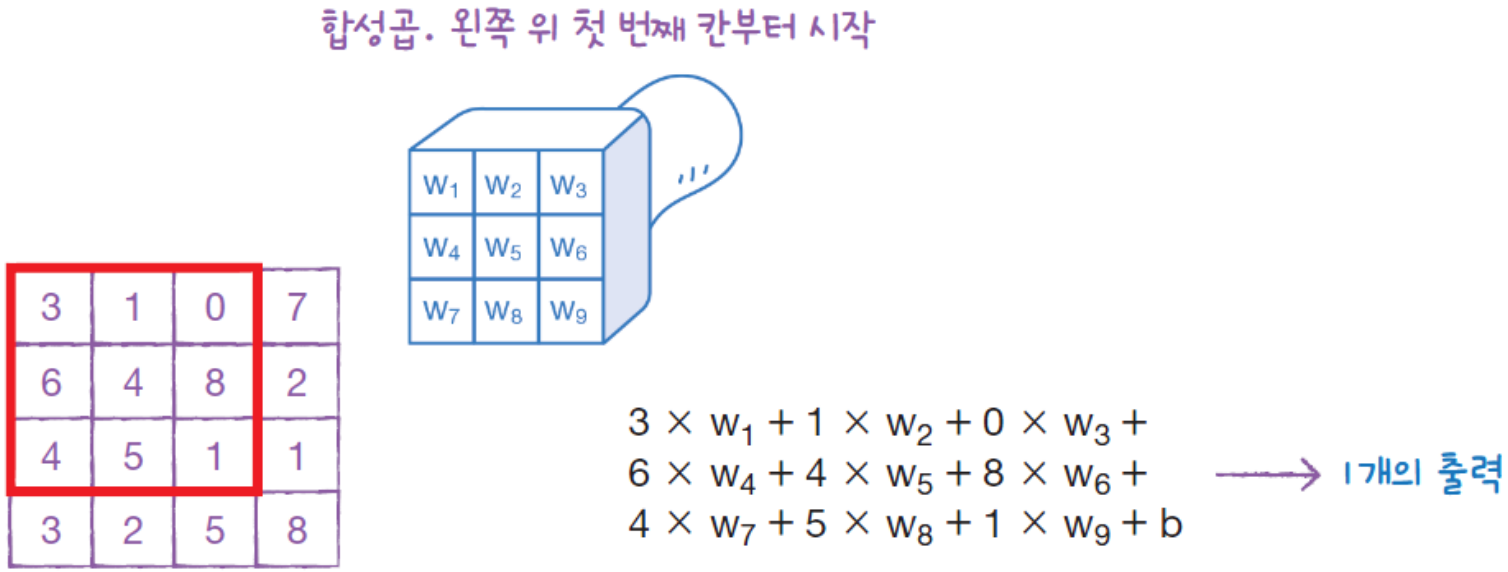
SECTION 8-1 합성곱 신경망의 구성 요소(5)

- 합성곱(convolution)
  - 필터(filter) 혹은 커널(kernel)

5) 밀집층의 뉴런은 입력 개수만큼 10개의 가중치를 가지고 1개의 출력을 만들지만, 합성곱 층의 뉴런은 3개의 가중치를 가지고 8개의 출력을 만듦
합성곱 신경망(convolutional neural network, CNN)에서는 완전 연결 신경망과 달리 뉴런을 필터(filter) 혹은 커널(kernel)이라고 부름

SECTION 8-1 합성곱 신경망의 구성 요소(6)

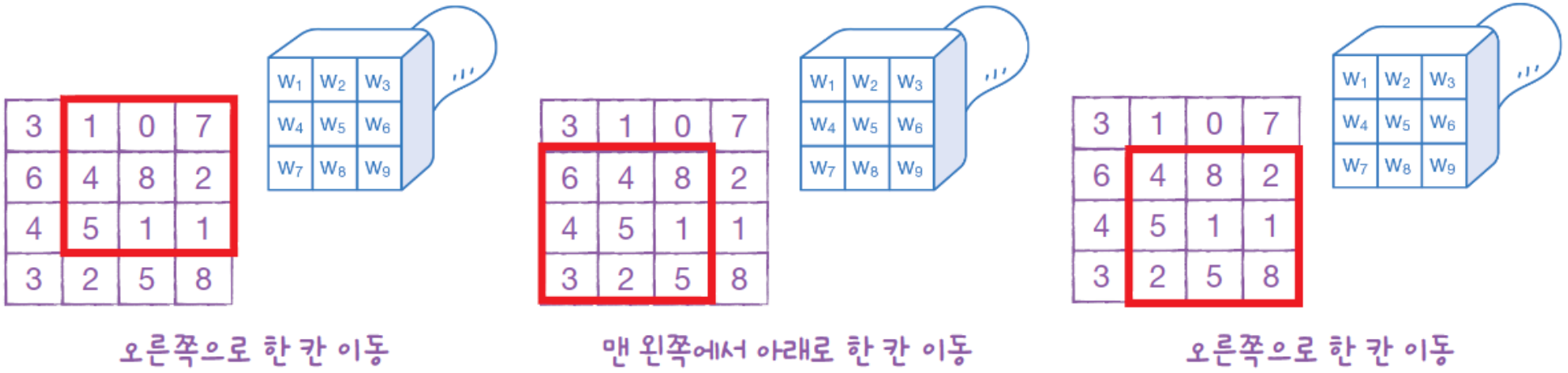
- 합성곱(convolution)
  - 합성곱의 장점: 1차원이 아니라 2차원 입력에도 적용 가능



※ 여기서는 케라스 API와 이름을 맞추어 다음과 같이 사용
- 커널: 입력에 곱하는 가중치
- 필터: 뉴런 개수를 표현

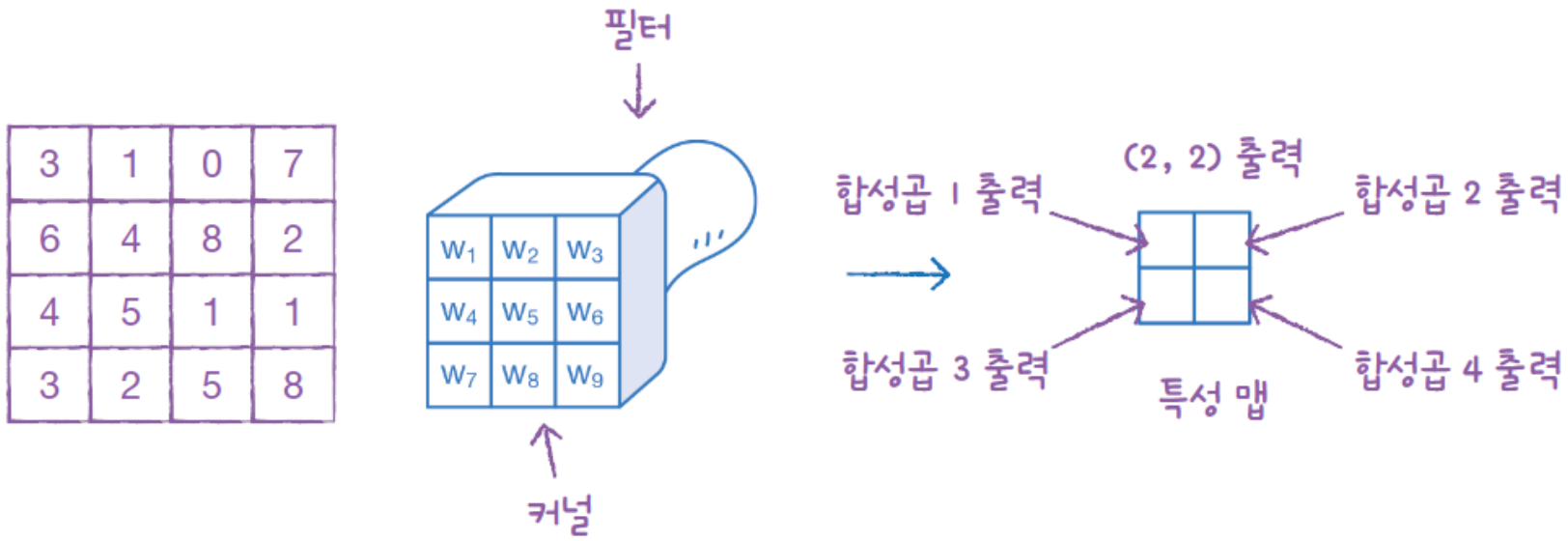
SECTION 8-1 합성곱 신경망의 구성 요소(7)

- 합성곱(convolution)
  1) 왼쪽 위 모서리에서부터 합성곱을 시작, 입력의 9개 원소와 커널의 9개 가중치를 곱한 후 (물론 여기에서도 절편을 더함) 1개의 출력을 생성
  2) 필터가 오른쪽으로 한 칸 이동하여 합성곱을 또 수행. 입력의 너비가 4이므로 더 이상 오른쪽으로 한 칸 이동할 수 없음
  3) 아래로 한 칸 이동한 다음 다시 왼쪽에서부터 합성곱을 수행
  4) 다시 오른쪽으로 한 칸 이동
  5) 필터는 모두 4번 이동할 수 있기 때문에 4개의 출력을 생성



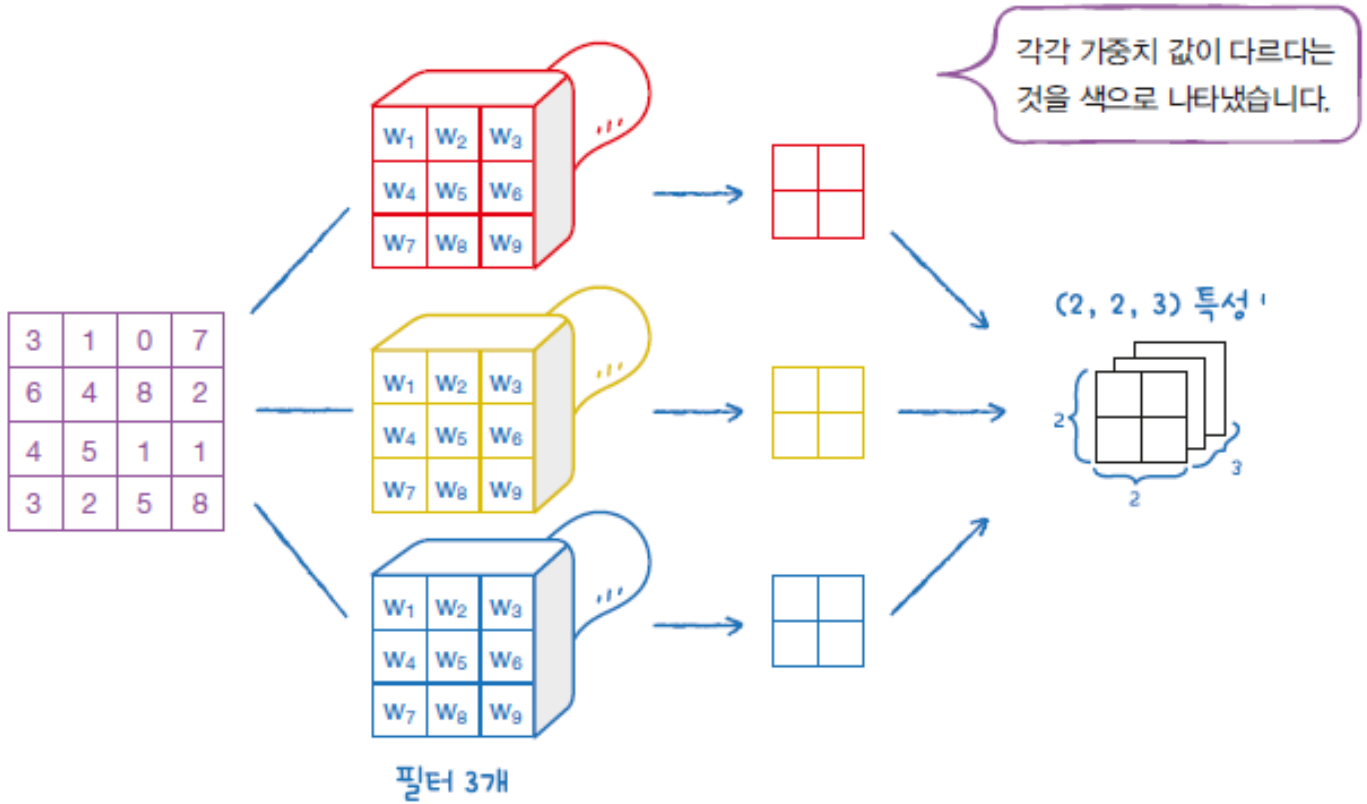
SECTION 8-1 합성곱 신경망의 구성 요소(8)

- 합성곱(convolution)
  - 특성 맵(feature map): 합성곱 계산을 통해 얻은 출력
  - 앞에서 4개의 출력을 필터가 입력에 놓인 위치에 맞게 2차원으로 배치
    - 즉 왼쪽 위, 오른쪽 위, 왼쪽 아래, 오른쪽 아래 모두 4개의 위치



SECTION 8-1 합성곱 신경망의 구성 요소(9)

- 합성곱(convolution)
  - 여러 개의 필터를 사용하면 만들어진 특성 맵은 순서대로 차곡차곡 쌓임
  - (2, 2) 크기의 특성 맵을 쌓으면 3차원 배열이 되고, 다음 그림에서는 3개의 필터를 사용했기 때문에 (2, 2, 3) 크기의 3차원 배열이 됨



SECTION 8-1 합성곱 신경망의 구성 요소(10)

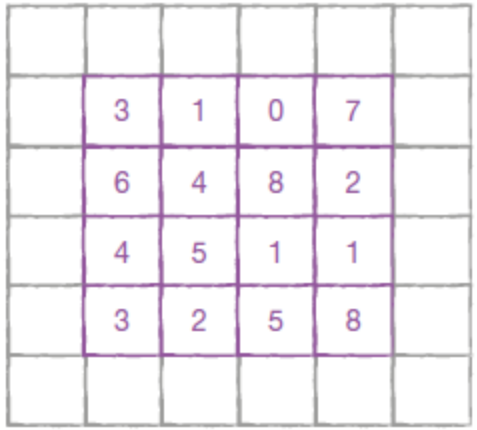
◦ 케라스 합성곱 층

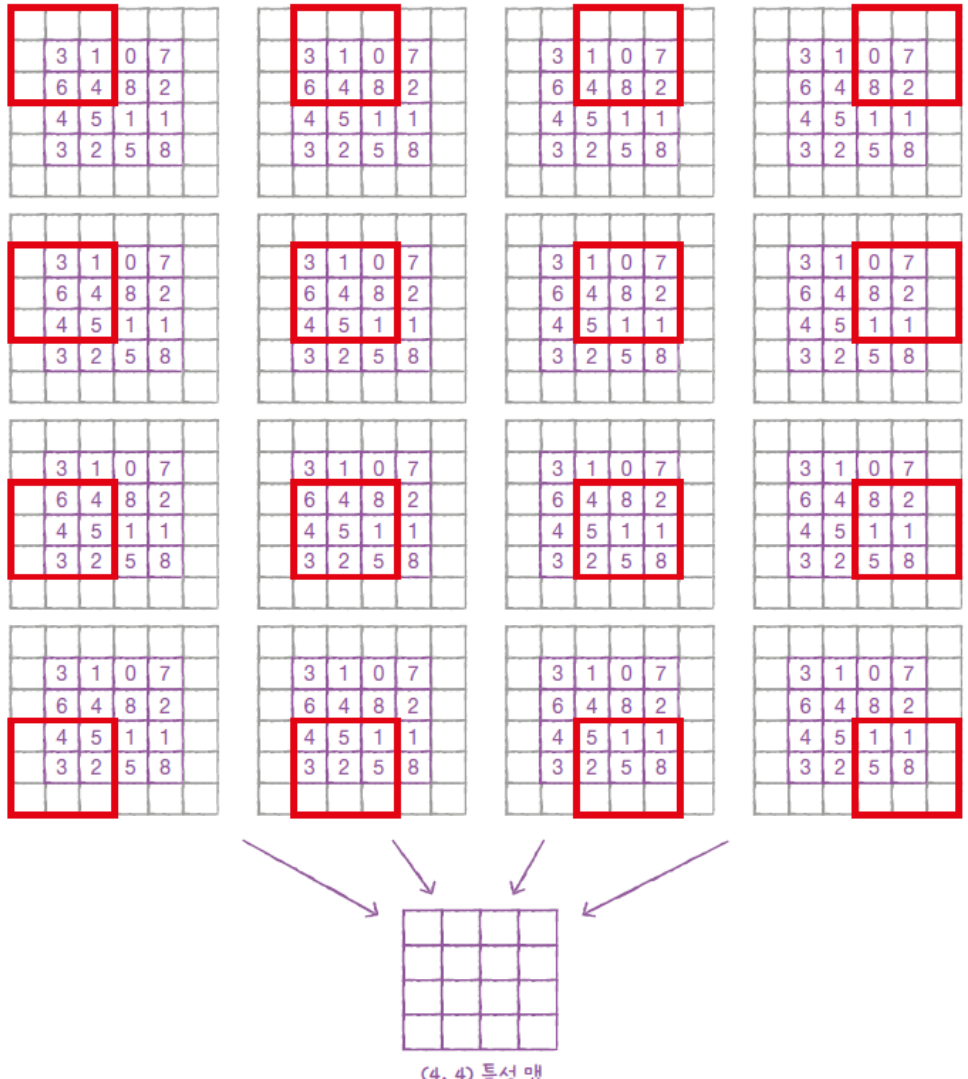
- 케라스의 합성곱 층은 keras.layers 패키지 아래 클래스로 구현
- 입력 위를 (왼쪽에서 오른쪽으로, 위에서 아래로) 이동하는 합성곱은 Conv2D 클래스로 제공
- 필터의 개수와 커널의 크기는 반드시 지정해야 하는 매개변수

```
import keras
keras.layers.Conv2D(10, kernel_size=(3, 3), activation='relu')
```

SECTION 8-1 합성곱 신경망의 구성 요소(11)

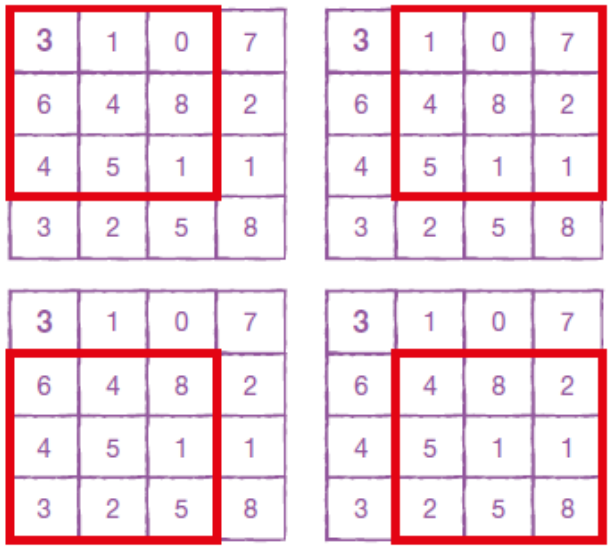
- 패딩(padding)과 스트라이드(stride)
  - 커널 크기는 (3, 3)으로 그대로 두고 출력의 크기를 입력과 동일하게 (4, 4)로 만들려면?

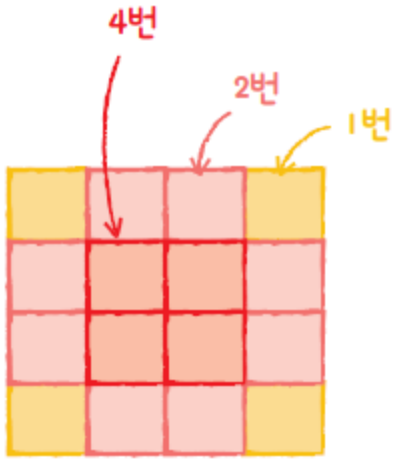


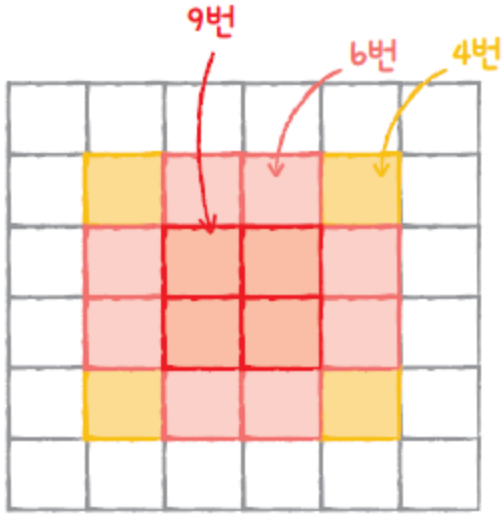


SECTION 8-1 합성곱 신경망의 구성 요소(12)

- 패딩(padding)과 스트라이드(stride)
  - 패딩: 입력 배열의 주위를 가상의 원소로 채우는 것. 실제 입력값이 아니기 때문에 패딩은 0으로 채움
  - 세임 패딩(same padding): 입력과 특성 맵의 크기를 동일하게 만들기 위해 입력 주위에 0으로 패딩 하는 것
  - 밸리드 패딩(valid padding): 패딩 없이 순수한 입력 배열에서만 합성곱을 하여 특성 맵을 만드는 경우
  - 패딩을 하지 않을 경우 중앙부와 모서리 픽셀이 합성곱에 참여하는 비율은 크게 차이남(4:1)
    - 1픽셀을 패딩 하면 이 차이는 크게 줄어듬(9:4)
    - 만약 2픽셀을 패딩 하면 중앙부와 모서리 픽셀이 합성곱에 참여하는 비율이 동일해짐(1:1).

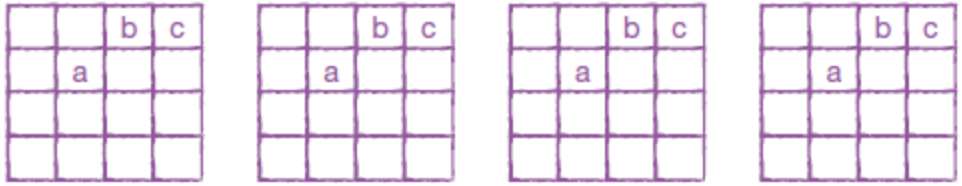




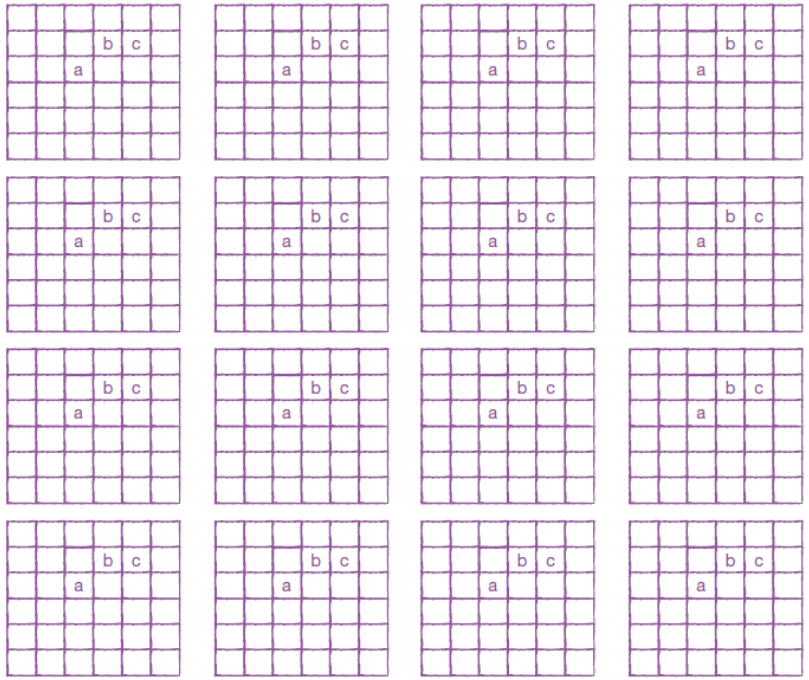


SECTION 8-1 합성곱 신경망의 구성 요소(13)

- 패딩(padding)과 스트라이드(stride)
  - (4, 4)에서 커널 크기가 (3, 3)일 때 a, b, c가 각각 몇 번 합성곱에 참여하나? 각각 4, 2, 1번씩 참여



  - 패딩을 준 (6, 6)에서 커널 크기가 (3, 3)일 때 a, b, c는 합성곱에 몇 번 참여하나? 9, 6, 4번씩 참여



SECTION 8-1 합성곱 신경망의 구성 요소(14)

◦ 패딩(padding)과 스트라이드(stride)

- 케라스 Conv2D 클래스에서는 padding 매개변수로 패딩을 지정
  - 기본값은 'valid'로 밸리드 패딩. 세임 패딩을 사용하려면 'same'으로 지정

```
keras.layers.Conv2D(10, kernel_size=(3,3), activation='relu', padding='same')
```

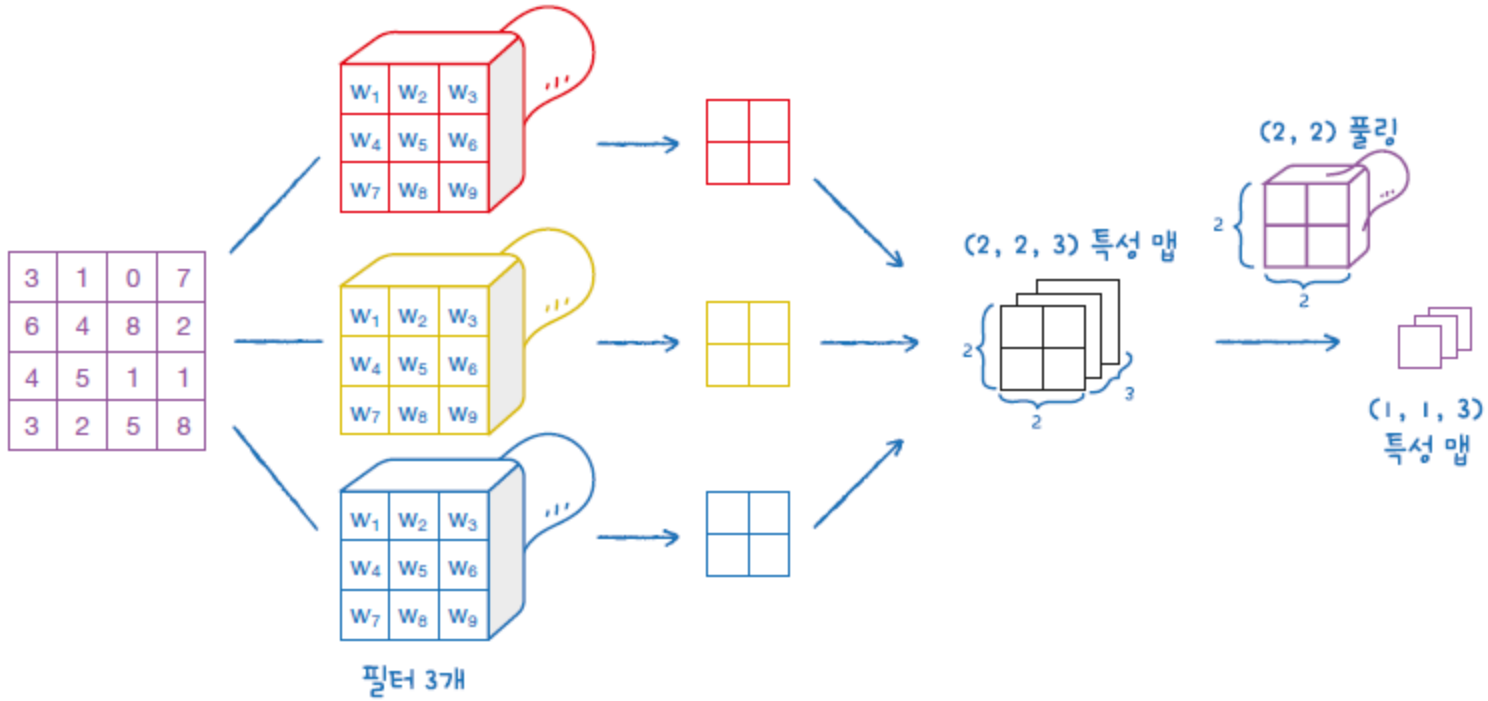
- 스트라이드: 합성곱 연산에서 이동의 크기

```
keras.layers.Conv2D(10, kernel_size=(3,3), activation='relu',
                    padding='same', strides=1)
```

SECTION 8-1 합성곱 신경망의 구성 요소(15)

◦ 풀링(pooling)

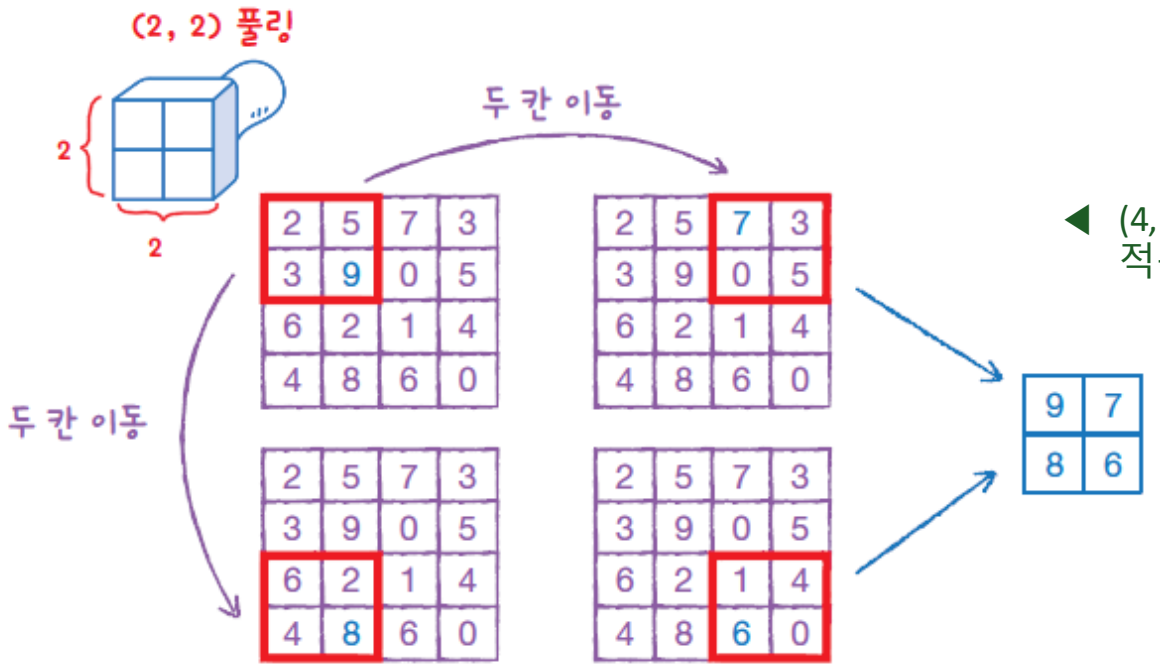
- 합성곱 층에서 만든 특성 맵의 가로세로 크기를 줄이는 역할을 수행하며, 특성맵의 개수는 줄이지 않음



▲ (2, 2, 3) 크기의 특성 맵에 풀링을 적용하면 마지막 차원인 개수는 그대로 유지하고 너비와 높이만 줄어들어 (1, 1, 3) 크기의 특성 맵이 됨

SECTION 8-1 합성곱 신경망의 구성 요소(16)

- 풀링(pooling)
  - 풀링에는 가중치가 없음
  - 최대 풀링(max pooling)과 평균 풀링(average pooling)
    - 도장을 찍은 영역에서 가장 큰 값을 고르거나 평균값을 계산



◀ (4, 4) 크기의 특성 맵에 (2, 2) 최대 풀링을 적용하면 절반으로 크기가 줄어듬

SECTION 8-1 합성곱 신경망의 구성 요소(17)

- 풀링(pooling)
  - 케라스에서는 MaxPooling2D 클래스로 최대 풀링을 수행

```
keras.layers.MaxPooling2D(2)
```

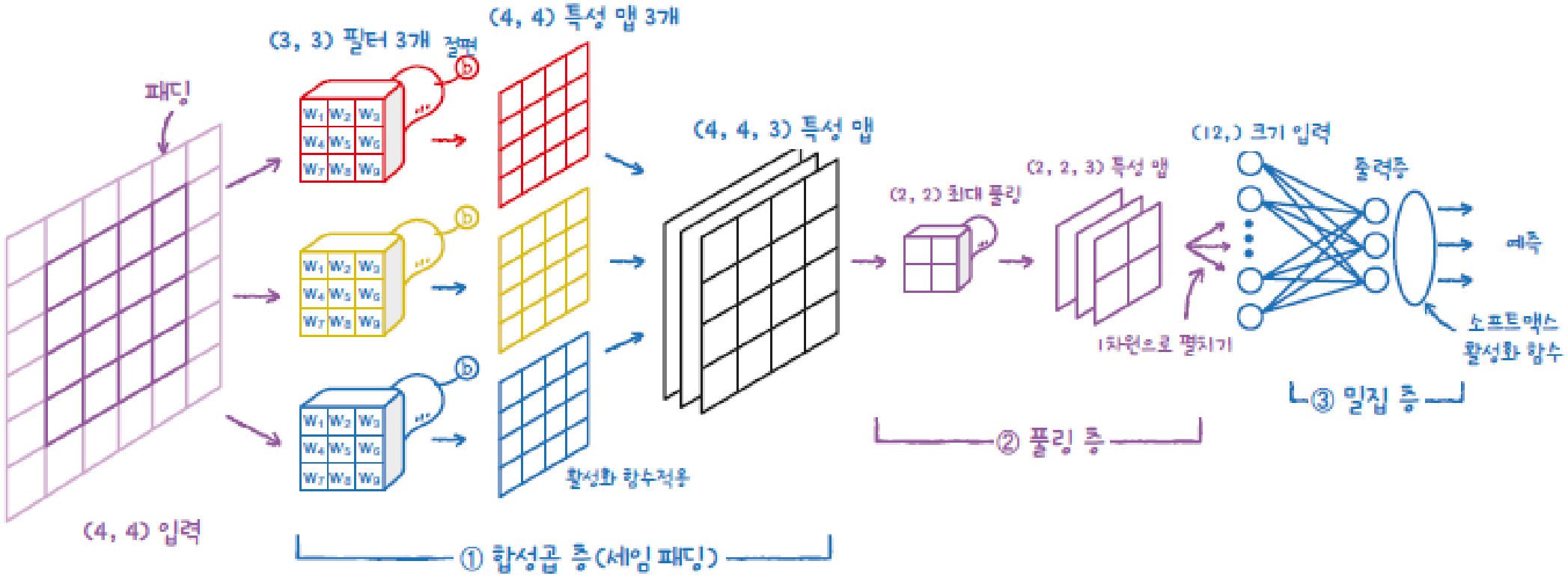
  - strides의 기본값은 자동으로 풀링의 크기이므로 따로 지정할 필요가 없음
    예) 바로 위에 쓴 최대 풀링과 같은 코드는 다음과 같음

```
keras.layers.MaxPooling2D(2, strides=2, padding='valid')
```

  - 평균 풀링을 제공하는 클래스는 AveragePooling2D
    - 최댓값 대신 평균을 계산하는 것만 빼면 MaxPooling2D와 동일하며 제공하는 매개변수도 같음

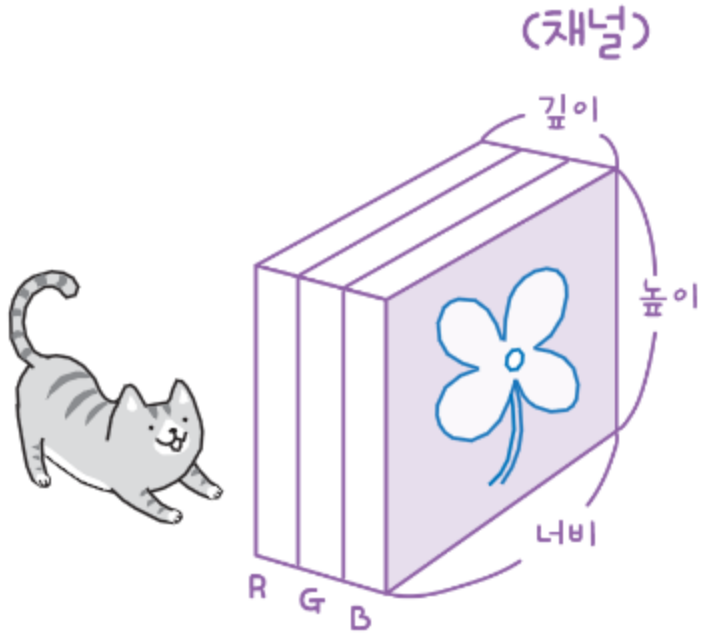
SECTION 8-1 합성곱 신경망의 구성 요소(18)

◦ 합성곱 신경망의 전체 구조



SECTION 8-1 합성곱 신경망의 구성 요소(19)

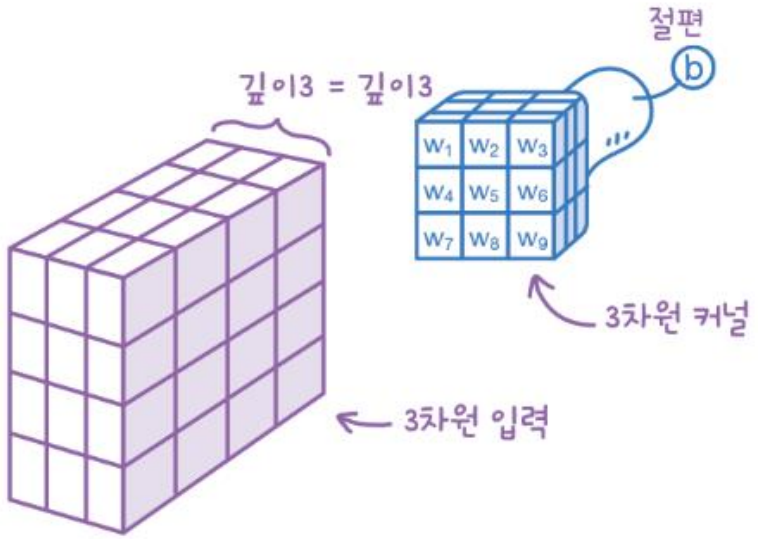
- 컬러 이미지를 사용한 합성곱
  - 컬러 이미지는 RGB(빨강, 초록, 파랑) 채널로 구성되어 있기 때문에 컴퓨터는 이를 3차원 배열로 표시

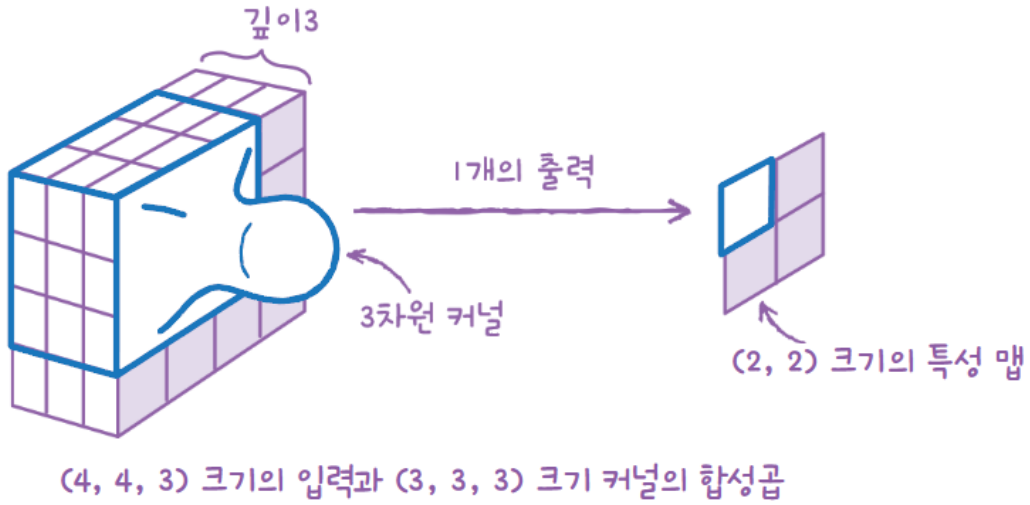


SECTION 8-1 합성곱 신경망의 구성 요소(20)

◦ 컬러 이미지를 사용한 합성곱

- 커널 배열의 깊이는 항상 입력의 깊이와 같음
- 합성곱의 계산은 (3, 3, 3) 영역에 해당하는 27개의 원소에 27개의 가중치를 곱하고 절편을 더함



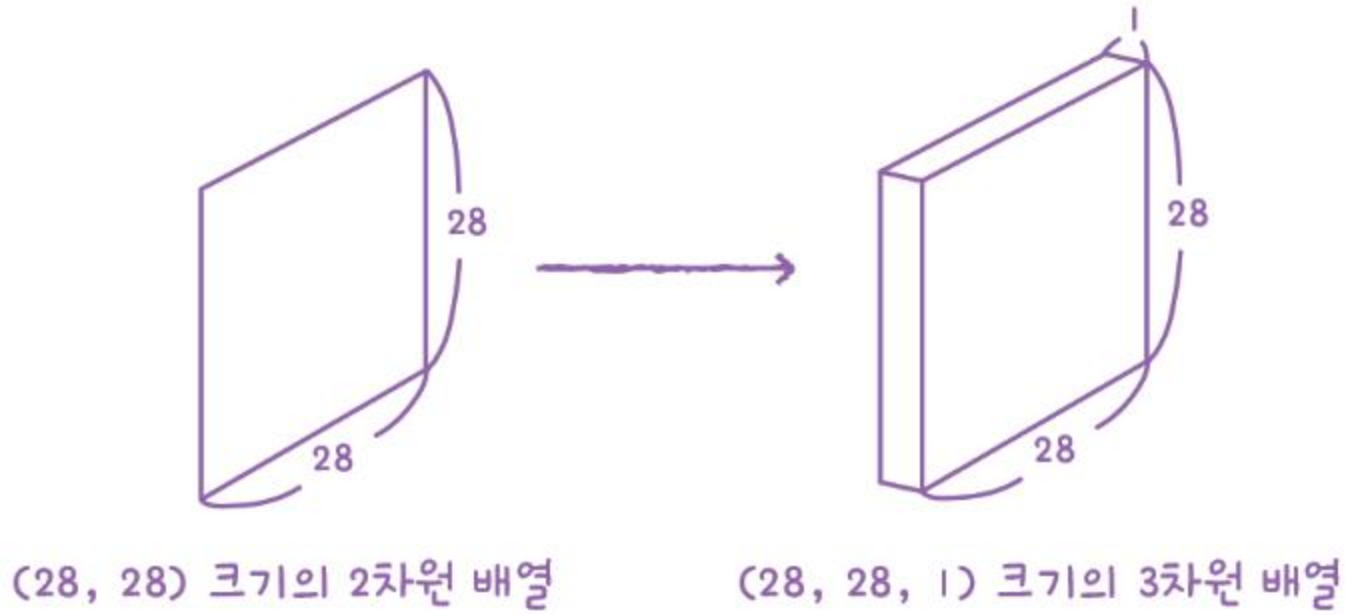


(4, 4, 3) 크기의 입력과 (3, 3, 3) 크기 커널의 합성곱

SECTION 8-1 합성곱 신경망의 구성 요소(21)

◦ 컬러 이미지를 사용한 합성곱

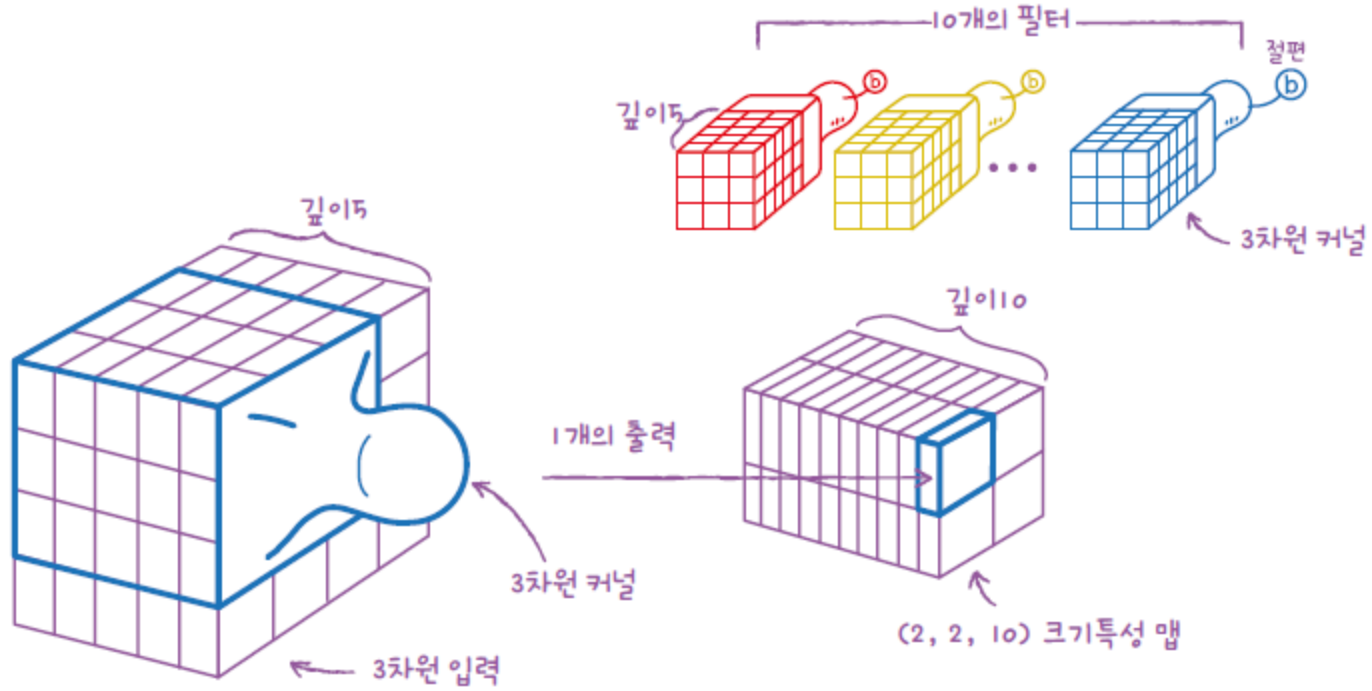
- 패션 MNIST 데이터처럼 흑백 이미지일 경우에는 깊이 차원이 1인 3차원 배열로 변환하여 전달
  - 예를 들어 (28, 28) 크기의 2차원 배열을 (28, 28, 1) 크기의 3차원 배열로 변환



SECTION 8-1 합성곱 신경망의 구성 요소(22)

◦ 컬러 이미지를 사용한 합성곱

- 패션합성곱 층-풀링 층 다음에 다시 또 합성곱 층이 올 경우
  - 첫 번째 합성곱 층의 필터 개수가 5개라고 가정하여 첫 번째 풀링 층을 통과한 특성 맵의 크기가 (4, 4, 5)
  - 두 번째 합성곱 층에서 필터의 너비와 높이가 각각 3이라면 이 필터의 커널 크기는 (3, 3, 5)



(4, 4, 5)크기 입력과 (3,3,5) 크기 커널의 합성곱

SECTION 8-1 합성곱 신경망의 구성 요소(23)

- 합성곱 층과 풀링 층 이해하기(문제해결 과정)
  - 학습
    - 합성곱 신경망을 구성하는 핵심 개념
    - 합성곱, 필터, 패딩, 스트라이드, 풀링 등
    - 합성곱 신경망은 직관적으로 이해하기 쉽지 않지만 이미지 처리에서 뛰어난 성능을 발휘
    - 합성곱 층과 풀링 층은 거의 항상 함께 사용
    - 합성곱 층에서 입력의 크기를 유지하며 각 필터가 추출한 특성 맵을 출력하면 풀링 층에서 특성 맵의 가로세로를 줄임 - 일반적으로 최대 풀링을 즐겨 사용하며 특성 맵을 절반으로 줄임
    - 마지막에는 특성 맵을 1차원 배열로 펼쳐서 1개 이상의 밀집층에 통과시켜 클래스에 대한 확률을 만듦

SECTION 8-1 마무리(1)

- 키워드로 끝나는 핵심 포인트
  - 합성곱은 밀집층과 비슷하게 입력과 가중치를 곱하고 절편을 더하는 선형 계산
    - 하지만 밀집층과 달리 각 합성곱은 입력 전체가 아니라 일부만 사용하여 선형 계산을 수행
  - 합성곱 층의 필터는 밀집층의 뉴런에 해당
    - 필터의 가중치와 절편을 종종 커널이라고 부름
    - 자주 사용되는 커널의 크기는 (3, 3) 또는 (5, 5)이며, 커널의 깊이는 입력의 깊이와 같음
  - 특성 맵은 합성곱 층이나 풀링 층의 출력 배열을 의미
    - 필터 하나가 하나의 특성 맵을 만듦. 합성곱 층에서 5개의 필터를 적용하면 5개의 특성 맵
  - 패딩은 합성곱 층의 입력 주위에 추가한 0으로 채워진 픽셀
    - 밸리드 패딩: 패딩을 사용하지 않는 것
    - 세임 패딩: 합성곱 층의 출력 크기를 입력과 동일하게 만들기 위해 입력에 패딩을 추가하는 것
  - 스트라이드는 합성곱 층에서 필터가 입력 위를 이동하는 크기
    - 일반적으로 스트라이드는 1픽셀을 사용
  - 풀링은 가중치가 없고 특성 맵의 가로세로 크기를 줄이는 역할을 수행
    - 대표적으로 최대 풀링과 평균 풀링이 있으며 (2, 2) 풀링으로 입력을 절반으로 줄임

SECTION 8-1 확인 문제(1)

1. 다음 중 합성곱 층에 대해 잘못 설명한 것은 무엇인가?

   ① 합성곱 층에서 널리 사용되는 커널 크기는 (3, 3)이나 (5, 5)

   ② 합성곱 층이 출력한 특성 맵의 채널 크기는 사용한 필터의 개수와 같음

   ③ 합성곱의 필터마다 서로 다른 가중치와 절편을 사용하여 선형 계산을 수행함

   ④ 일반적으로 입력에 풀링을 먼저 수행한 후 합성곱 층이 적용됨

2. 어떤 합성곱 층이 컬러 이미지에 대해 5개의 필터를 사용해 세임 패딩으로 합성곱을 수행하고 그다음 (2, 2) 풀링 층을 통과한 특성 맵의 크기가 (4, 4, 5). 이 경우 합성곱 층에 주입된 입력의 크기는?

   ① (4, 4, 3)　　② (4, 4, 5)

   ③ (8, 8, 3)　　④ (8, 8, 5)

SECTION 8-1 확인 문제(2)

3. 다음과 같은 입력에서 (3, 3) 커널과 밸리드 패딩으로 합성곱을 수행합니다. 필터의 개수는 1개이고 입력의 깊이(채널)도 1개, 절편은 0이라고 가정. 이 합성곱의 결과를 계산하면?

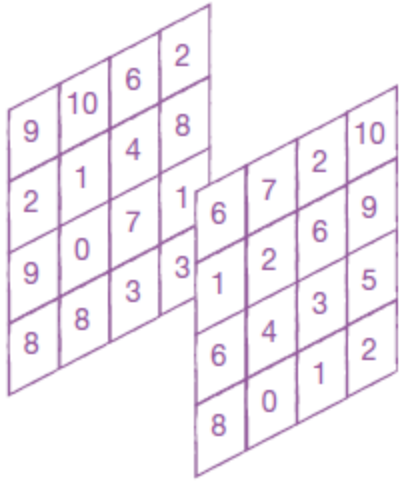
(5, 5) 입력

3	0	9	1	2
5	1	2	0	7
8	2	4	1	3
2	1	5	3	6
4	1	6	2	7

(3, 3) 커널

2	0	1
2	0	1
2	0	1

4. 다음과 같은 (4, 4, 2) 크기의 특성 맵에서 (2, 2) 최대 풀링의 결과를 계산



SECTION 8-2 합성곱 신경망을 사용한 이미지 분류(1)

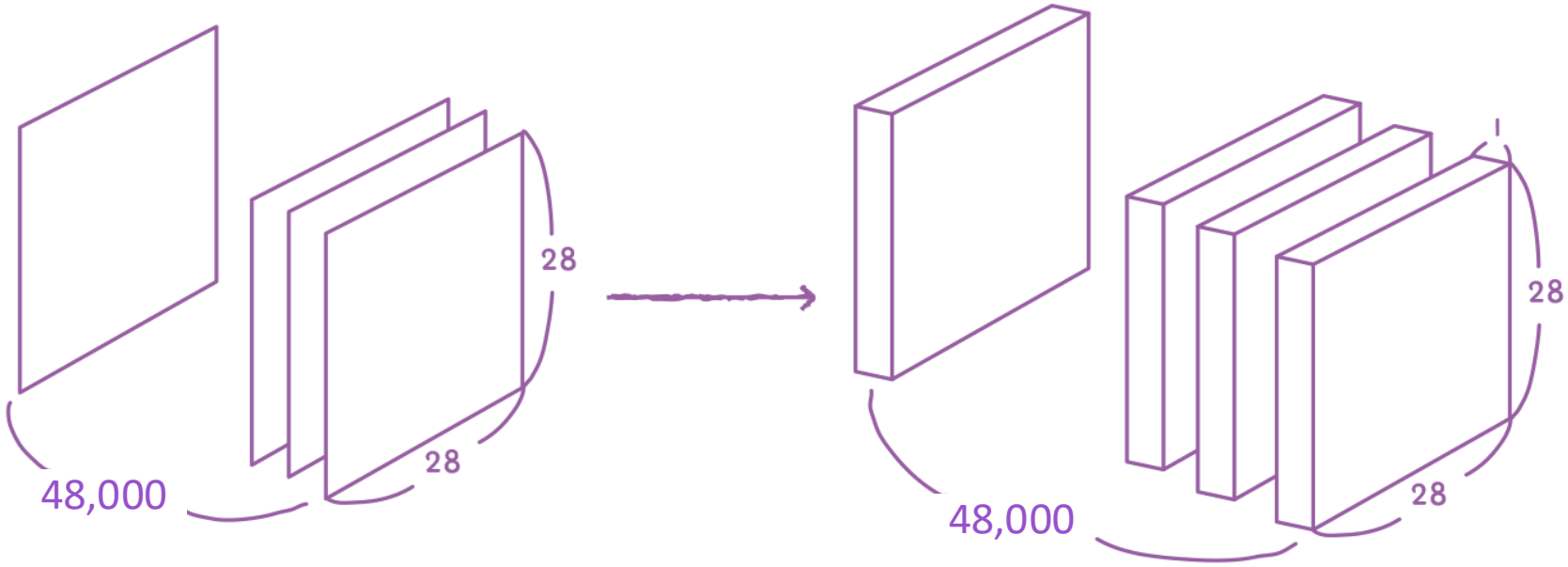
◦ 패션 MNIST 데이터 불러오기

- 주피터 노트북에서 케라스 API를 사용해 패션 MNIST 데이터를 불러오고 적절히 전처리
  - 데이터 스케일을 0~255 사이에서 0~1 사이로 바꾸고 훈련 세트와 검증 세트로 나누기
  - 완전 연결 신경망에서는 입력 이미지를 밀집층에 연결하기 위해 일렬로 펼쳐야 함
  - 넘파이 reshape( ) 메서드를 사용해 전체 배열 차원을 그대로 유지하면서 마지막에 차원을 추가

```
import keras
from sklearn.model_selection import train_test_split
(train_input, train_target), (test_input, test_target) =\
    keras.datasets.fashion_mnist.load_data()
train_scaled = train_input.reshape(-1, 28, 28, 1) / 255.0
train_scaled, val_scaled, train_target, val_target = train_test_split(
    train_scaled, train_target, test_size=0.2, random_state=42)
```

SECTION 8-2 합성곱 신경망을 사용한 이미지 분류(2)

- 패션 MNIST 데이터 불러오기
  - 이제 (48000, 28, 28) 크기인 train_input이 (48000, 28, 28, 1) 크기인 train_scaled가 됨



(48000, 28, 28) 크기의 3차원 배열

(48000, 28, 28, 1) 크기의 4차원 배열

SECTION 8-2 합성곱 신경망을 사용한 이미지 분류(3)

◦ 합성곱 신경망 만들기

- 전형적인 합성곱 신경망의 구조는 합성곱 층으로 이미지에서 특징을 감지한 후 밀집층으로 클래스에 따른 분류 확률을 계산
- 케라스의 Sequential 클래스를 사용해 순서대로 이 구조를 정의
  - Sequential 클래스의 객체를 만들고 첫 번째 합성곱 층인 Conv2D를 추가 - add( ) 메서드를 사용

```
model = keras.Sequential()
model.add(keras.layers.Input(shape=(28,28,1)))
model.add(keras.layers.Conv2D(32, kernel_size=3, activation='relu',
                              padding='same'))
```

  - 풀링 층을 추가 - MaxPooling2D와 AveragePooling2D 클래스

```
model.add(keras.layers.MaxPooling2D(2))
```

  - 첫 번째 합성곱-풀링 층 다음에 두 번째 합성곱-풀링 층을 추가

```
model.add(keras.layers.Conv2D(64, kernel_size=3, activation='relu',
                              padding='same'))
model.add(keras.layers.MaxPooling2D(2))
```

SECTION 8-2 합성곱 신경망을 사용한 이미지 분류(4)

◦ 합성곱 신경망 만들기

- 3차원 특성 맵을 일렬로 펼침
  - 마지막에1 0개의 뉴런을 가진 (밀집) 출력층에서 확률을 계산하기 때문임
  - 특성 맵을 일렬로 펼쳐서 바로 출력층에 전달하지 않고 중간에 하나의 밀집 은닉층을 하나 더 구성
    - 즉 Flatten 클래스 다음에 Dense 은닉층, 마지막으로 Dense 출력층의 순서대로 구성
    - 은닉층과 출력층 사이에 드롭아웃을 넣어 은닉층의 과대적합을 막아 성능을 조금 더 개선

```
model.add(keras.layers.Flatten())
model.add(keras.layers.Dense(100, activation='relu'))
model.add(keras.layers.Dropout(0.4))
model.add(keras.layers.Dense(10, activation='softmax'))
```

SECTION 8-2 합성곱 신경망을 사용한 이미지 분류(5)

- 합성곱 신경망 만들기
  - summary( ) 메서드로 모델 구조를 출력

```
model.summary()
```

→

Model: "sequential"

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 28, 28, 32)	320
max_pooling2d (MaxPooling2D)	(None, 14, 14, 32)	0
conv2d_1 (Conv2D)	(None, 14, 14, 64)	18,496
max_pooling2d_1 (MaxPooling2D)	(None, 7, 7, 64)	0
flatten (Flatten)	(None, 3136)	0
dense (Dense)	(None, 100)	313,700
dropout (Dropout)	(None, 100)	0
dense_1 (Dense)	(None, 10)	1,010

Total params: 333,526 (1.27 MB)

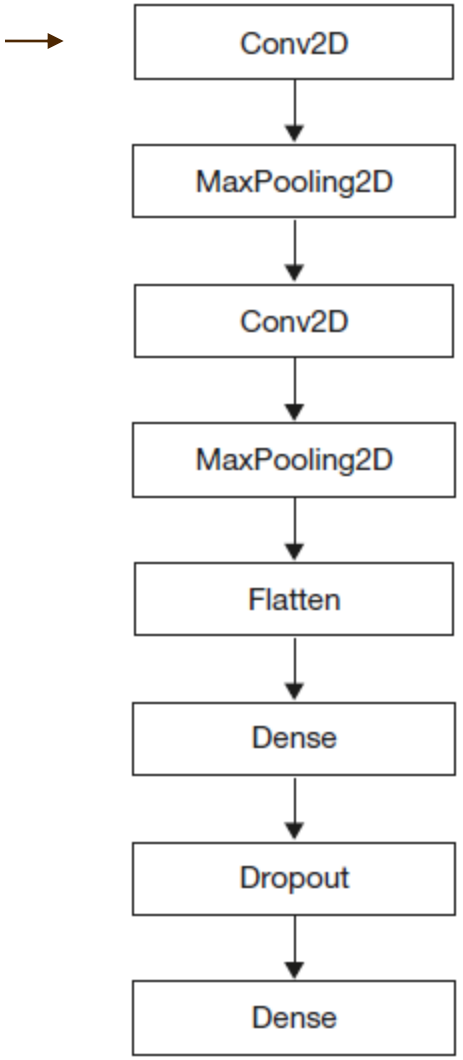
Trainable params: 333,526 (1.27 MB)

Non-trainable params: 0 (0.00 B)

SECTION 8-2 합성곱 신경망을 사용한 이미지 분류(6)

- 합성곱 신경망 만들기
  - plot_model( ) 함수로 층의 구성을 그림으로 확인

```
keras.utils.plot_model(model)
```

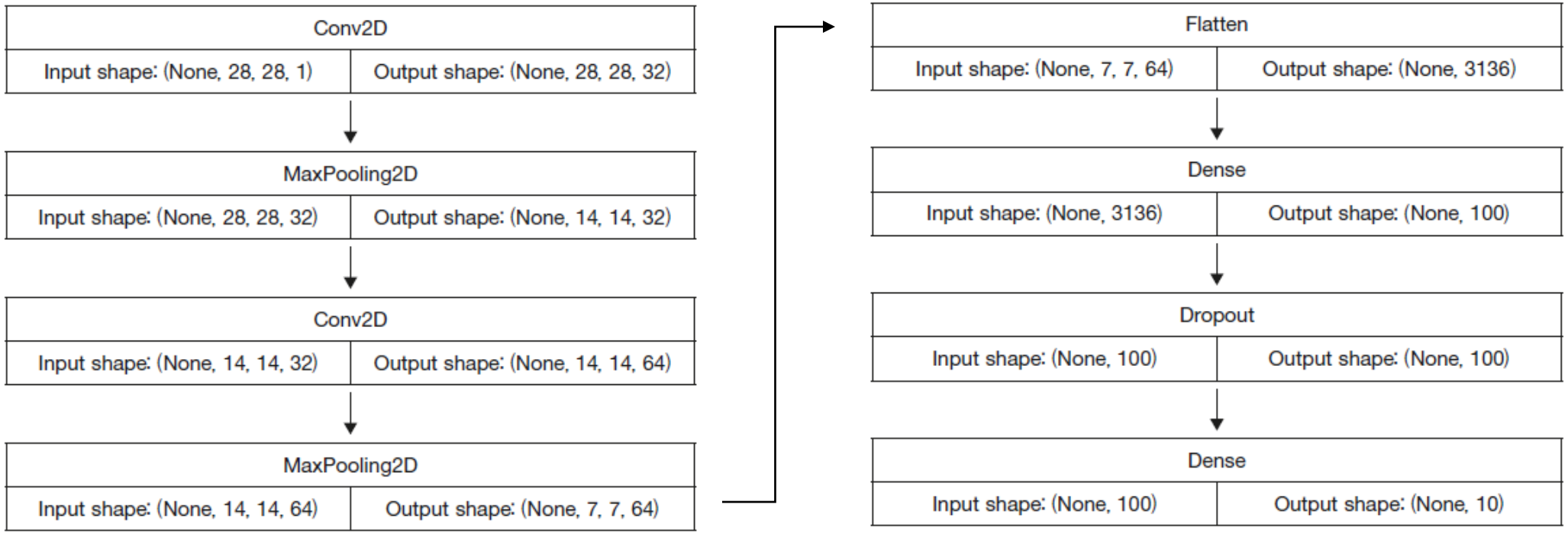


SECTION 8-2 합성곱 신경망을 사용한 이미지 분류(7)

◦ 합성곱 신경망 만들기

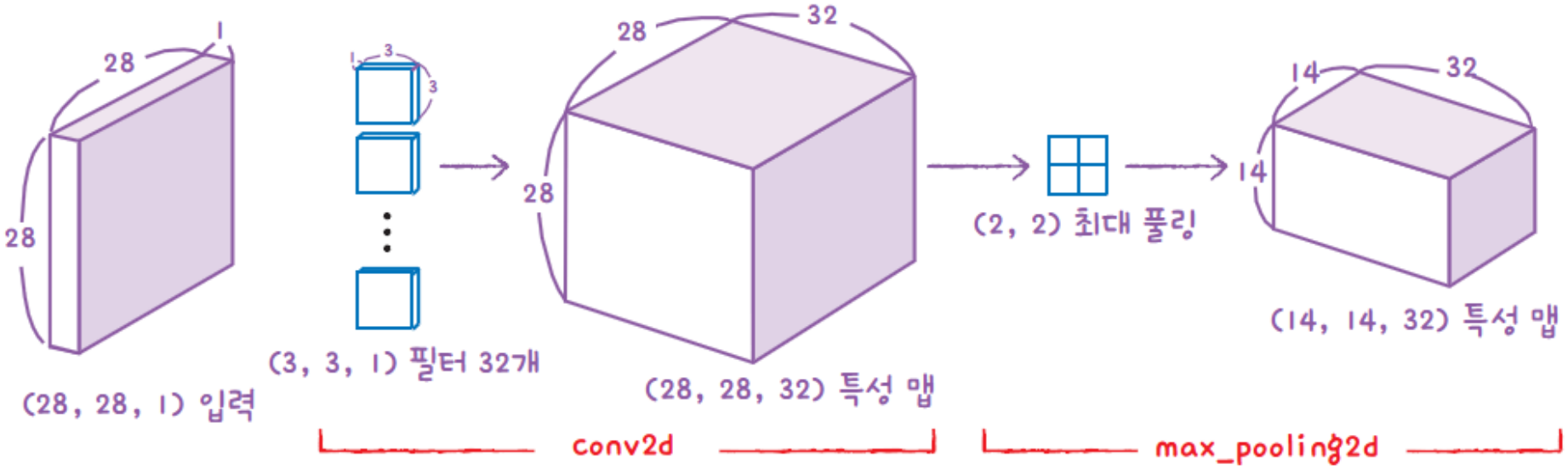
- plot_model( ) 함수의 show_shapes 매개변수를 True로 설정하면 이 그림에 입력과 출력의 크기를 표시 to_file 매개변수에 파일 이름을 지정하면 출력한 이미지를 파일로 저장 - dpi 기본값은 200

```
keras.utils.plot_model(model, show_shapes=True)
```



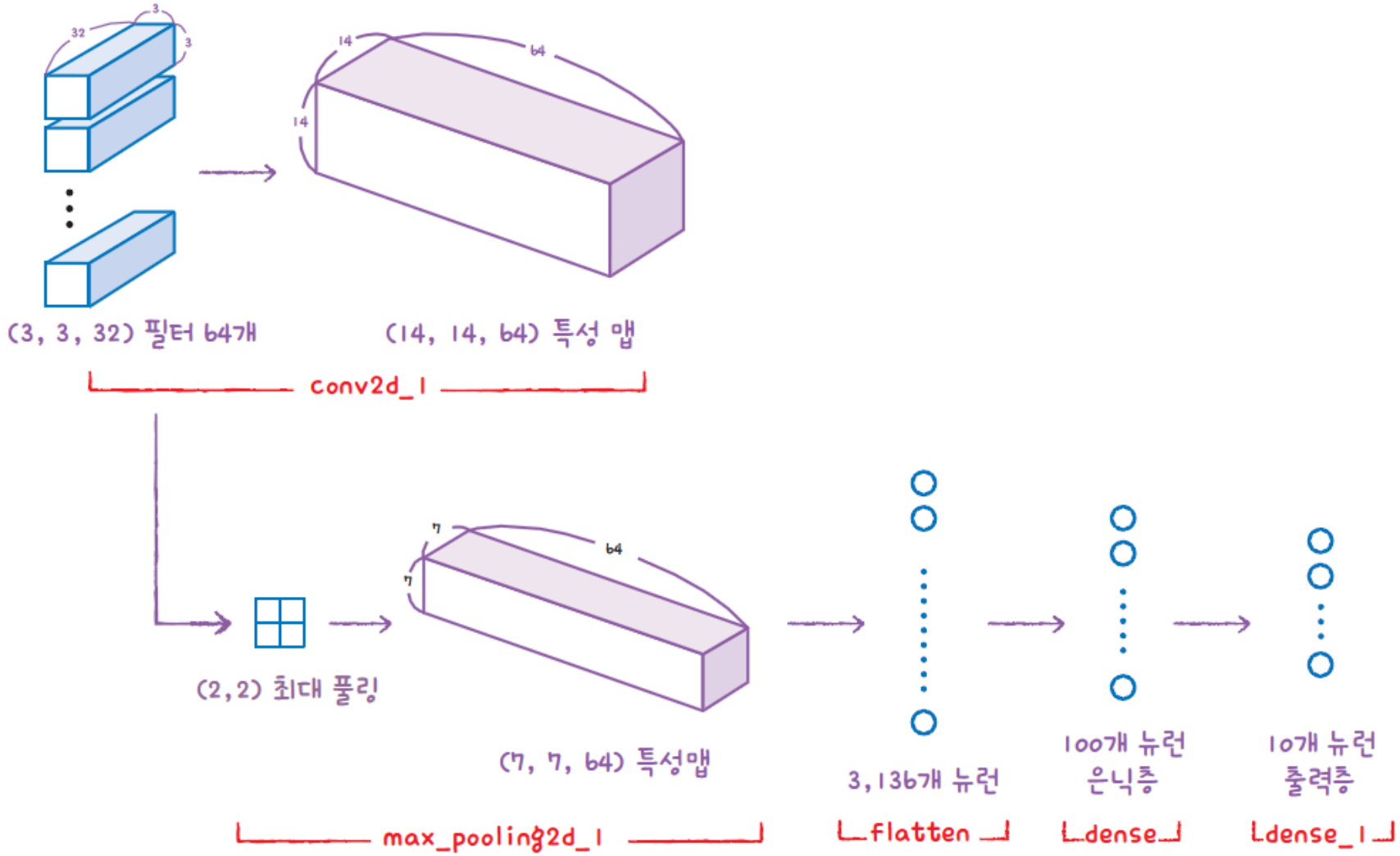
SECTION 8-2 합성곱 신경망을 사용한 이미지 분류(8)

◦ 합성곱 신경망 만들기



SECTION 8-2 합성곱 신경망을 사용한 이미지 분류(9)

- 합성곱 신경망 만들기



SECTION 8-2 합성곱 신경망을 사용한 이미지 분류(10)

◦ 모델 컴파일과 훈련

• Adam 옵티마이저, ModelCheckpoint 콜백과 EarlyStopping 콜백을 함께 사용해 조기 종료 기법을 구현

```
model.compile(optimizer='adam', loss='sparse_categorical_crossentropy',
                        metrics=['accuracy'])
checkpoint_cb = keras.callbacks.ModelCheckpoint('best-cnn-model.keras',
                                                save_best_only=True)
early_stopping_cb = keras.callbacks.EarlyStopping(patience=2,
                                                  restore_best_weights=True)
history = model.fit(train_scaled, train_target, epochs=20,
                            validation_data=(val_scaled, val_target),
                            callbacks=[checkpoint_cb, early_stopping_cb])
```

▶ 출력 결과는 다음 쪽에 이어짐

SECTION 8-2 합성곱 신경망을 사용한 이미지 분류(11)

◦ 모델 컴파일과 훈련

- Adam 옵티마이저, ModelCheckpoint 콜백과 EarlyStopping 콜백을 함께 사용해 조기 종료 기법을 구현

◀ 앞쪽 코드 출력 결과

```
Epoch 1/20
1500/1500 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 13s 5ms/step - accuracy:
0.7401 - loss: 0.7269 - val_accuracy: 0.8774 - val_loss: 0.3317
Epoch 2/20
1500/1500 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 7s 5ms/step - accuracy:
0.8664 - loss: 0.3699 - val_accuracy: 0.8928 - val_loss: 0.2956
Epoch 3/20
1500/1500 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 10s 4ms/step - accuracy:
0.8858 - loss: 0.3108 - val_accuracy: 0.9027 - val_loss: 0.2621
Epoch 4/20
1500/1500 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 10s 4ms/step - accuracy:
0.8966 - loss: 0.2785 - val_accuracy: 0.9102 - val_loss: 0.2471
Epoch 5/20
1500/1500 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 10s 4ms/step - accuracy:
0.9067 - loss: 0.2539 - val_accuracy: 0.9087 - val_loss: 0.2466
Epoch 6/20
1500/1500 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 7s 5ms/step - accuracy:
0.9124 - loss: 0.2320 - val_accuracy: 0.9149 - val_loss: 0.2350
Epoch 7/20
1500/1500 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 6s 4ms/step - accuracy:
0.9197 - loss: 0.2169 - val_accuracy: 0.9195 - val_loss: 0.2370
Epoch 8/20
1500/1500 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 8s 5ms/step - accuracy:
0.9267 - loss: 0.1947 - val_accuracy: 0.9191 - val_loss: 0.2369
```

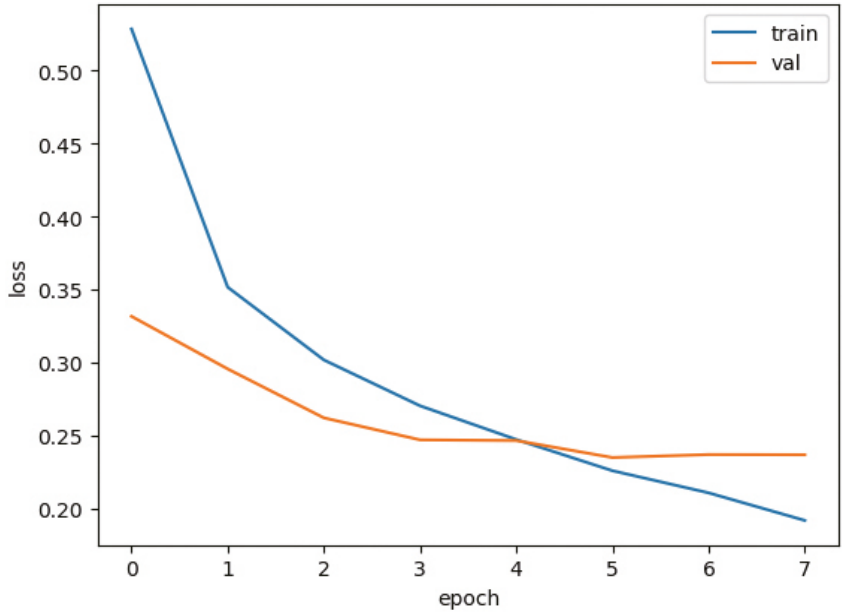
SECTION 8-2 합성곱 신경망을 사용한 이미지 분류(12)

◦ 모델 컴파일과 훈련

- 손실 그래프를 그려서 조기 종료가 잘 이루어졌는지 확인

```
import matplotlib.pyplot as plt

plt.plot(history.history['loss'], label='train')
plt.plot(history.history['val_loss'], label='val')
plt.xlabel('epoch')
plt.ylabel('loss')
plt.legend()
plt.show()
```

→



- EarlyStopping 클래스에서 restore_best_weights 매개변수를 True로 지정했으므로 현재 model 객체가 최적의 모델 파라미터로 복원

▲ 검증 세트에 대한 손실이 점차 감소하다가 정체되기 시작하고 훈련 세트에 대한 손실은 점점 더 낮아지는 추세. 이 그래프를 기반으로 여섯 번째 에포크를 최적으로 판단할 수 있음

SECTION 8-2 합성곱 신경망을 사용한 이미지 분류(13)

◦ 모델 컴파일과 훈련

- 세트에 대한 성능 평가

```
model.evaluate(val_scaled, val_target)
```

→

```
375/375 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy:
0.9153 - loss: 0.2300
[0.2349757254123687, 0.9149166941642761]
```

- 결과는 fit ( ) 메서드의 출력 중 여섯 번째 에포크의 출력과 동일
- EarlyStopping 콜백이 model 객체를 최상의 모델 파라미터로 잘 복원한 것으로 이해됨

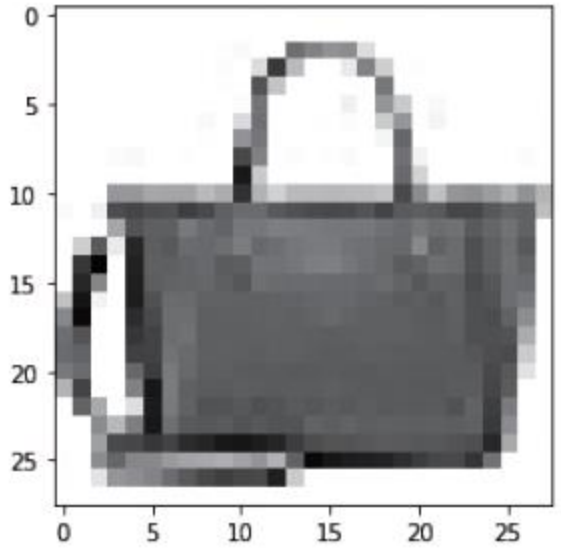
SECTION 8-2 합성곱 신경망을 사용한 이미지 분류(14)

◦ 모델 컴파일과 훈련

- 첫 번째 샘플 이미지 확인
  -맷플롯립에서는 흑백 이미지에 깊이 차원은 없으므로 (28, 28, 1) 크기를 (28,28)로 바꾸어 출력

```
plt.imshow(val_scaled[0].reshape(28, 28), cmap='gray_r')
plt.show()
```

→



SECTION 8-2 합성곱 신경망을 사용한 이미지 분류(15)

◦ 모델 컴파일과 훈련

- predict ( ) 메서드는 10개의 클래스에 대한 예측 확률을 출력

```
preds = model.predict(val_scaled[0:1])
print(preds)
```

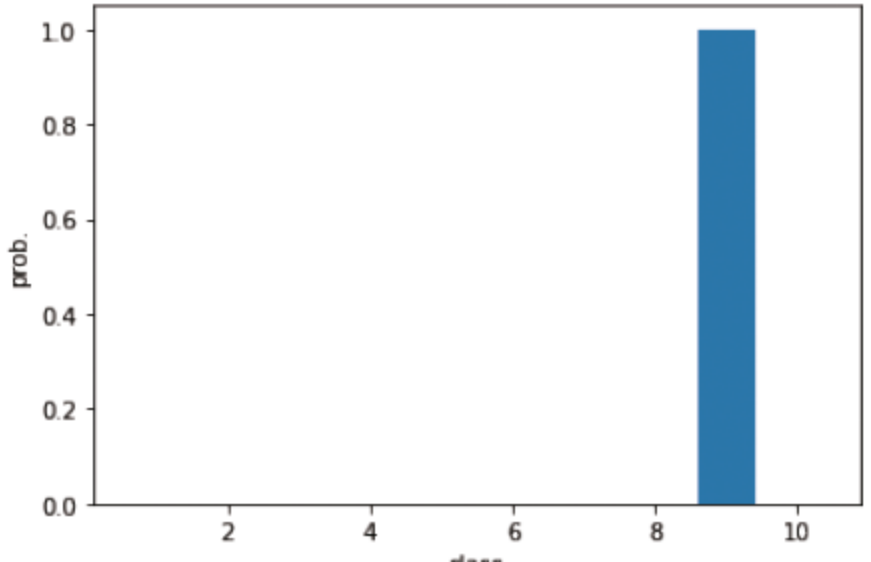
→

```
1/1 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 0s 133ms/step
[[2.1421800e-18 8.7024404e-25 3.2938590e-20 3.9525481e-20 4.2847604e-19
5.3240785e-18 6.1603418e-19 4.9034962e-19 1.0000000e+00 1.5801817e-20]]
```

- 출력 결과를 보면 아홉 번째 값이 1이고 다른 값은 거의 0에 가까움
  막대그래프로 확인

```
plt.bar(range(1, 11), preds[0])
plt.xlabel('class')
plt.ylabel('prob.')
plt.show()
```

→



SECTION 8-2 합성곱 신경망을 사용한 이미지 분류(16)

◦ 모델 컴파일과 훈련

- 아홉 번째 클래스가 실제로 무엇인지는 패션 MNIST 데이터셋의 정의를 참고
- 패션 MNIST 데이터셋의 레이블을 리스트로 저장

```
classes = ['티셔츠', '바지', '스웨터', '드레스', '코트', '샌달', '셔츠', '스니커즈',
           '가방', '앵클 부츠']
```

- preds 배열에서 가장 큰 인덱스를 찾아 classes 리스트의 인덱스로 사용

```
import numpy as np

print(classes[np.argmax(preds)])
```

→ 가방

SECTION 8-2 합성곱 신경망을 사용한 이미지 분류(17)

◦ 모델 컴파일과 훈련

- 맨 처음에 떼어 놓았던 테스트 세트로 합성곱 신경망의 일반화 성능 측정

  • 훈련 세트와 검증 세트에서 했던 것처럼 픽셀값의 범위를 0~1 사이로 바꾸고 이미지 크기를 (28,28)에서 (28, 28, 1)로 변환

```
test_scaled = test_input.reshape(-1, 28, 28, 1) / 255.0
```

  • evaluate ( ) 메서드로 테스트 세트에 대한 성능을 측정

```
model.evaluate(test_scaled, test_target)
```

→

```
313/313 [==============================] - 1s 4ms/step - loss: 0.2457 -
accuracy: 0.9124
[0.26197266578674316, 0.9088000059127808]
```

SECTION 8-2 합성곱 신경망을 사용한 이미지 분류(18)

◦ 케라스 API로 합성곱 신경망 구현(문제해결 과정)

- 학습

  • 케라스 API를 사용해 합성곱 신경망 만들기

  • 케라스의 Conv2D 클래스를 사용해 32개의 필터와 64개의 필터를 둔 2개의 합성곱 층을 추가

  • 두 합성곱 층 다음에는 모두 (2, 2) 크기의 최대 풀링 층을 배치

  • 두 번째 풀링 층을 통과한 특성 맵을 펼친 다음 밀집 은닉층에 연결하고 최종적으로 10개의 뉴런을 가진 출력층에서 샘플에 대한 확률을 출력

  • 조기 종료 기법을 사용해 모델을 훈련한 다음 검증 세트로 최적의 에포크에서 성능을 평가

  • 샘플 데이터 하나를 선택해 예측 클래스를 출력하는 방법

  • 이제까지 사용하지 않았던 테스트 세트를 사용해 최종 모델의 일반화 성능을 평가
    - 항상 테스트 세트는 모델을 출시하기 직전 딱 한 번만 사용
    - 그렇지 않다면 모델을 실전에 투입했을 때 성능을 올바르게 예측하지 못함

  • 합성곱 신경망은 이미지를 주로 다루기 때문에 각 층의 출력을 시각화하기 좋음

SECTION 8-2 마무리(1)

- 키워드로 끝나는 핵심 포인트
  - 케라스의 Conv2D, MaxPooling2D, plot_model를 활용한 실습

- 핵심 패키지와 함수
  - Keras
    - Conv2D: 입력의 너비와 높이 방향의 합성곱 연산을 구현한 클래스
      - 첫 번째 매개변수는 합성곱 필터의 개수
      - kernel_size 매개변수 - 필터의 커널 크기를 지정
      - strides 매개변수 - 필터의 이동 간격을 지정
      - padding 매개변수 - 입력의 패딩 타입을 지정
      - activation 매개변수 - 합성곱 층에 적용할 활성화 함수를 지정

SECTION 8-2 마무리(2)

◦ 핵심 패키지와 함수

- Keras

- MaxPooling2D: 입력의 너비와 높이를 줄이는 풀링 연산을 구현한 클래스
  - 첫 번째 매개변수 - 풀링의 크기를 지정
  - strides 매개변수 - 풀링의 이동 간격을 지정
  - padding 매개변수 - 입력의 패딩 타입을 지정
- plot_model ( ): 케라스 모델 구조를 주피터 노트북에 그리거나 파일로 저장
  - 첫 번째 매개변수 - 케라스 모델 객체를 전달
  - to_file 매개변수 - 파일 이름을 지정하면 그림을 파일로 저장
  - show_shapes 매개변수 - True로 지정하면 층의 입력, 출력 크기를 표시
  - show_layer_names 매개변수 - True로 지정하면 층 이름을 출력

- matplotlib

- bar ( )는 막대그래프를 출력
  - 첫 번째 매개변수 - x축의 값을 리스트나 넘파이 배열로 전달
  - 두 번째 매개변수 - 막대의 y축 값을 리스트나 넘파이 배열로 전달
  - width 매개변수 - 막대의 두께를 지정(기본값은 0.8)

SECTION 8-2 확인 문제(1)

1. 합성곱 층의 필터가 입력 위를 이동하는 간격을 조절하는 Conv2D 클래스의 매개변수는?

   ① kernel_size
   ② strides
   ③ padding
   ④ activation

2. Conv2D 클래스의 padding 매개변수의 올바른 설명은?

   ① 'valid'는 입력의 크기가 커널의 크기의 배수가 되도록 패딩
   ② 'valid'는 입력과 출력의 가로세로 크기를 동일하게 만들도록 패딩
   ③ 'same'은 입력의 크기가 커널의 크기의 배수가 되도록 패딩
   ④ 'same'은 입력과 출력의 가로세로 크기를 동일하게 만들도록 패딩

SECTION 8-2 확인 문제(2)

3. 다음 중 최대 풀링 층의 풀링 매개변수를 잘못 설정한 것은?

① MaxPooling2D(2)
② MaxPooling2D((2, 2))
③ MaxPooling2D(2, 2)
④ MaxPooling2D((2, 2, 2))

Conv2D 층의 입력에 대해 올바르게 설명한 것은?

① Conv2D 층은 3차원 입력을 기대함
② Conv2D 층을 사용하는 합성곱 신경망의 입력 크기는 배치 차원을 포함해 3차원임
③ 기본적으로 케라스 신경망 모델에서 이미지 입력의 크기는 (배치, 채널, 높이, 너비)
④ 흑백 이미지의 경우 채널이 없어도 3차원 배열로 간주

SECTION 8-2 파이토치 버전 살펴보기(1)

- 파이토치로 합성곱 신경망 모델 훈련하기
  - 합성곱 층과, Sequential 클래스에 층을 추가하는 방법
  - 수동으로 배치 데이터셋을 준비하는 대신 파이토치에 포함된 데이터 로더(data loader)를 활용하여 모델을 훈련

SECTION 8-2 파이토치 버전 살펴보기(2)

- torchvision 패키지를 사용해 패션 MNIST 데이터셋을 로드하고 훈련 세트와 검증 세트로 나누기
  - 주의 - 파이토치는 이미지의 채널 차원이 배치 차원 바로 다음에 올 것이라 기대한다는 점
  - 따라서 기존 (28, 28) 크기의 이미지를 (1, 28, 28) 크기로 변환해야 함
  - train_input의 reshape( ) 메서드를 사용해 첫 번째 배치 차원 다음에 크기가 1인 두 번째 차원을 추가

```
from torchvision.datasets import FashionMNIST

fm_train = FashionMNIST(root='.', train=True, download=True)
fm_test = FashionMNIST(root='.', train=False, download=True)

train_input = fm_train.data
train_target = fm_train.targets
train_scaled = train_input.reshape(-1, 1, 28, 28) / 255.0

from sklearn.model_selection import train_test_split

train_scaled, val_scaled, train_target, val_target = train_test_split(
        train_scaled, train_target, test_size=0.2, random_state=42)
```

SECTION 8-2 파이토치 버전 살펴보기(3)

- Sequential클래스로 모델 객체를 먼저 생성한 후, add_module( ) 메서드로 층을 하나씩 추가
  - add_module( ) 메서드를 사용할 때는 첫 번째 매개변수로 층 이름을, 두 번째 매개변수로 층 객체를 전달
  - 본문에서와 같이 32개의 필터를 가진 합성곱 층과 렐루 활성화 함수, 풀링 층을 추가

```
import torch.nn as nn

model = nn.Sequential()
model.add_module('conv1', nn.Conv2d(1, 32, kernel_size=3, padding='same'))
model.add_module('relu1', nn.ReLU())
model.add_module('pool1', nn.MaxPool2d(2))
```

    - 파이토치의 합성곱 층 클래스는 Conv2d(마지막 d가 소문자)
    - 첫 번째 매개변수로 입력 채널 개수를 지정하고 두번째 매개변수에 출력 채널 개수, 즉 필터 개수를 지정
    - kernel_size로 커널 크기를 정수 하나 또는 정수의 튜플로 지정
    - 패딩도 본문의 예와 동일하게 'same' 패딩을 사용
    - 렐루 활성화 함수를 추가하고, 풀링 층을 추가
    - 파이토치의 최대 풀링 클래스는 MaxPool2d
    - 첫 번째 매개변수에 풀링 크기를 지정
    - 합성곱 층과 마찬가지로 정수 하나 또는 (높이, 너비)를 나타내는 정수 튜플을 지정할 수 있음

SECTION 8-2 파이토치 버전 살펴보기(4)

- 두 번째 합성곱 층과 렐루 활성화 함수, 최대 풀링 층을 또 추가
- 마지막에 Flatten 층을 추가
- 두 번째 합성곱 층의 입력 채널 크기는 첫 번째 합성곱 층의 출력 채널 개수와 같음
  - 렐루 활성화 함수와 최대 풀링 층은 채널 크기를 변경시키지 않기 때문임

```
model.add_module('conv2', nn.Conv2d(32, 64, kernel_size=3, padding='same'))
model.add_module('relu2', nn.ReLU())
model.add_module('pool2', nn.MaxPool2d(2))
model.add_module('flatten', nn.Flatten())
```

SECTION 8-2 파이토치 버전 살펴보기(5)

- Linear 층 추가를 위해 Flatten 층으로 펼친 입력의 크기를 먼저 확인
  - 첫 번째 합성곱 층은 세임 패딩을 사용하므로, 입력과 출력의 높이와 너비가 동일하게 유지
  - 하지만 그다음에 오는 풀링 층 때문에 높이와 너비가 절반으로 줄어 들어 14×14가 됨
  - 따라서 두 번째 합성곱 층도 세임 패딩을 사용하므로, 출력 크기가 동일
  - 두 번째 최대 풀링 때문에 특성 맵의 높이와 너비는 절반으로 다시 줄어 들어 7×7가 됨
  - 두 번째 합성곱 층이 64개의 채널을 만들기 때문에 이 특성 맵을 Flatten 층으로 펼친 크기는 7×7×64 = 3136

  - 간단한 방법 - 지금까지 만든 모델에 가짜 입력을 만들어 통과시켜서 어떤 크기의 출력이 만들어지는지 확인
    예) torch.ones( ) 함수로 값이 모두 1로 채워진 (1, 1, 28, 28) 크기 배열을 하나 만들어 모델에 전달

```
outputs = model(torch.ones(1, 1, 28, 28))
print(outputs.shape)
```

→ torch.Size([1, 3136])

SECTION 8-2 파이토치 버전 살펴보기(6)

- Linear 층, 렐루 활성화 함수, 드롭아웃 층, 출력 층을 추가하여 모델 구성을 완료

```
model.add_module('dense1', nn.Linear(3136, 100))
model.add_module('relu3', nn.ReLU())
model.add_module('dropout', nn.Dropout(0.3))
model.add_module('dense2', nn.Linear(100, 10))
```

SECTION 8-2 파이토치 버전 살펴보기(7)

- GPU로 모델을 전달하고 손실 함수와 옵티마이저를 준비

```
import torch

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model.to(device)

import torch.optim as optim

criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters())
```

SECTION 8-2 파이토치 버전 살펴보기(8)

- 데이터셋을 변환하고 배치 데이터셋을 만들기 위한 여러 도구
  - TensorDataset 클래스 - 여러 텐서를 결합하여 하나의 데이터셋으로 관리
  - 데이터를 섞을 때 입력과 타깃이 같은 순서로 섞여야 함
    - 따라서 배치를 만들기 전에 TensorDataset으로 입력과 타깃을 묶어 주는 것이 좋음
  - DataLoader 클래스 - TensorDataset으로 만든 데이터셋으로 배치를 생성
    - 이때 배치 크기와 데이터를 섞을지 여부를 결정

SECTION 8-2 파이토치 버전 살펴보기(9)

- 훈련 세트와 검증 세트로 각각 데이터셋을 만들고 배치 크기 32인 데이터 로더를 만들기
  - 검증 세트는 모델을 훈련하는데 사용하는 것이 아니므로 배치마다 섞을 필요가 없음
    - 따라서 shuffle 매개변수를 False로 지정(False가 기본값이지만 작동 방식을 잘 드러내기 위해 지정)

```
from torch.utils.data import TensorDataset, DataLoader

train_dataset = TensorDataset(train_scaled, train_target)
val_dataset = TensorDataset(val_scaled, val_target)

train_loader = DataLoader(train_dataset, batch_size=32, shuffle=True)
val_loader = DataLoader(val_dataset, batch_size=32, shuffle=False)
```

SECTION 8-2 파이토치 버전 살펴보기(10)

- 모델 훈련 코드는 거의 동일하지만 batches 변수는 필요하지 않으며, 두 번째 for 문에서 데이터로더가 입력과 타깃을 자동으로 배치 단위로 전달해 주기 때문에 코드가 간결해짐

```
train_hist = []
val_hist = []
patience = 2
best_loss = -1
early_stopping_counter = 0

epochs = 20
for epoch in range(epochs):
    model.train()

train_loss = 0
for inputs, targets in train_loader:
    inputs, targets = inputs.to(device), targets.to(device)
    optimizer.zero_grad()
    outputs = model(inputs)
    loss = criterion(outputs, targets)
    loss.backward()
    optimizer.step()
    train_loss += loss.item()
```

SECTION 8-2 파이토치 버전 살펴보기(11)

- 검증 손실 계산
  - for 반복문과 검증 세트용 데이터로더를 사용
  - val_loss 변수에 검증 손실을 누적해서 기록

```
model.eval()
    val_loss = 0
    with torch.no_grad():
        for inputs, targets in val_loader:
            inputs, targets = inputs.to(device), targets.to(device)
            outputs = model(inputs)
            loss = criterion(outputs, targets)
            val_loss += loss.item()
```

SECTION 8-2 파이토치 버전 살펴보기(12)

- 평균 훈련 손실과 평균 검증 손실을 계산해 출력
  - 누적된 훈련 손실과 검증 손실 값은 배치 횟수로 나눔
  - 데이터로더에 파이썬 len ( ) 함수를 적용하면 배치 반복 횟수를 반환
  - 마지막의 조기 종료 코드는 이전 절과 동일

```
train_loss = train_loss/len(train_loader)
    val_loss = val_loss/len(val_loader)
    train_hist.append(train_loss)
    val_hist.append(val_loss)
    print(f"에포크:{epoch+1},",
        f"훈련 손실:{train_loss:.4f}, 검증 손실:{val_loss:.4f}")

if best_loss == -1 or val_loss < best_loss:
    best_loss = val_loss
    early_stopping_counter = 0
    torch.save(model.state_dict(), 'best_cnn_model.pt')
else:
    early_stopping_counter += 1
    if early_stopping_counter >= patience:
        print(f"{epoch+1}번째 에포크에서 조기 종료되었습니
다.")
        break
```

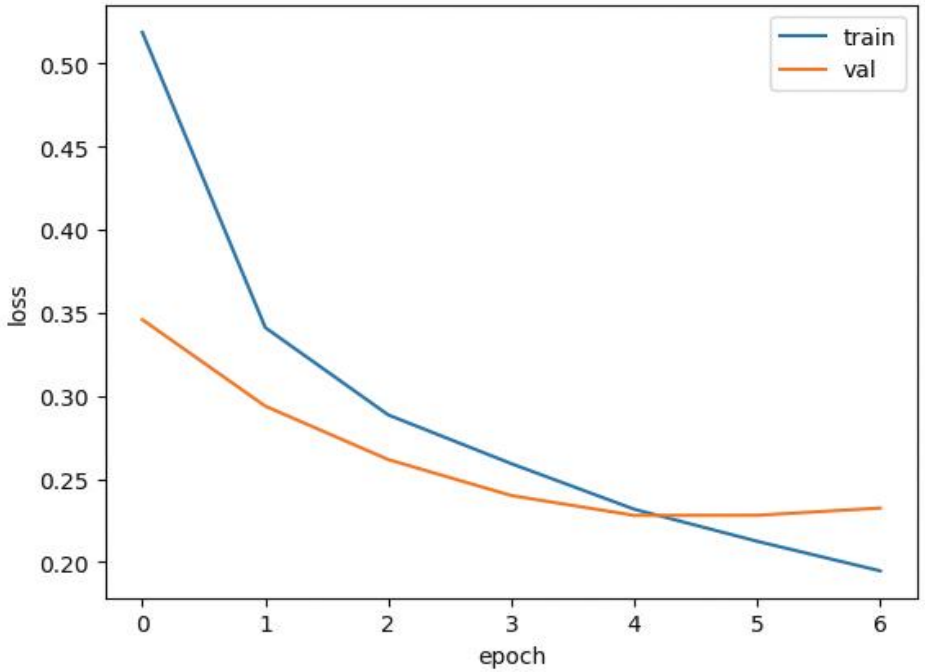
→

```
에포크:1, 훈련 손실:0.5187, 검증 손실:0.3458
에포크:2, 훈련 손실:0.3410, 검증 손실:0.2938
에포크:3, 훈련 손실:0.2885, 검증 손실:0.2616
에포크:4, 훈련 손실:0.2592, 검증 손실:0.2400
에포크:5, 훈련 손실:0.2317, 검증 손실:0.2280
에포크:6, 훈련 손실:0.2124, 검증 손실:0.2281
에포크:7, 훈련 손실:0.1946, 검증 손실:0.2324
7번째 에포크에서 조기 종료되었습니다.
```

SECTION 8-2 파이토치 버전 살펴보기(13)

- 훈련 손실과 검증 손실 그래프

```
import matplotlib.pyplot as plt

plt.plot(train_hist, label='train')
plt.plot(val_hist, label='val')
plt.xlabel('epoch')
plt.ylabel('loss')
plt.legend()
plt.show()
```



다섯 번째 에포크 이후부터 과대적합이 발생

SECTION 8-2 파이토치 버전 살펴보기(14)

- 검증 세트에 대한 정확도 확인
  - 훈련 과정에서 저장한 'best_cnn_model.pt' 파일을 다시 로드
  - 데이터로더를 사용하므로, corrects 변수에 값을 누적한 후 검증 세트 크기로 나누어 정확도를 계산

```
model.load_state_dict(torch.load('best_cnn_model.pt', weights_only=True))

model.eval()
corrects = 0
with torch.no_grad():
    for inputs, targets in val_loader:
        inputs, targets = inputs.to(device), targets.to(device)
        outputs = model(inputs)
        predicts = torch.argmax(outputs, 1)
        corrects += (predicts == targets).sum().item()

accuracy = corrects / len(val_dataset)
print(f"검증 정확도: {accuracy:.4f}")
```

→ 검증 정확도: 0.9155

SECTION 8-2 파이토치 버전 살펴보기(15)

- 훈련 세트에 대한 정확도 확인
  - fm_test 객체에 준비된 FashionMNIST 클래스를 사용한 테스트 세트
  - 이 객체의 data와 targets 속성을 사용해 앞서 훈련 세트에서 했던 것과 동일한 과정을 거쳐 데이터로더를 준비

```
test_scaled = fm_test.data.reshape(-1, 1, 28, 28) / 255.0
test_target = fm_test.targets
test_dataset = TensorDataset(test_scaled, test_target)
test_loader = DataLoader(test_dataset, batch_size=32, shuffle=False)
```

  - 테스트 세트의 정확도

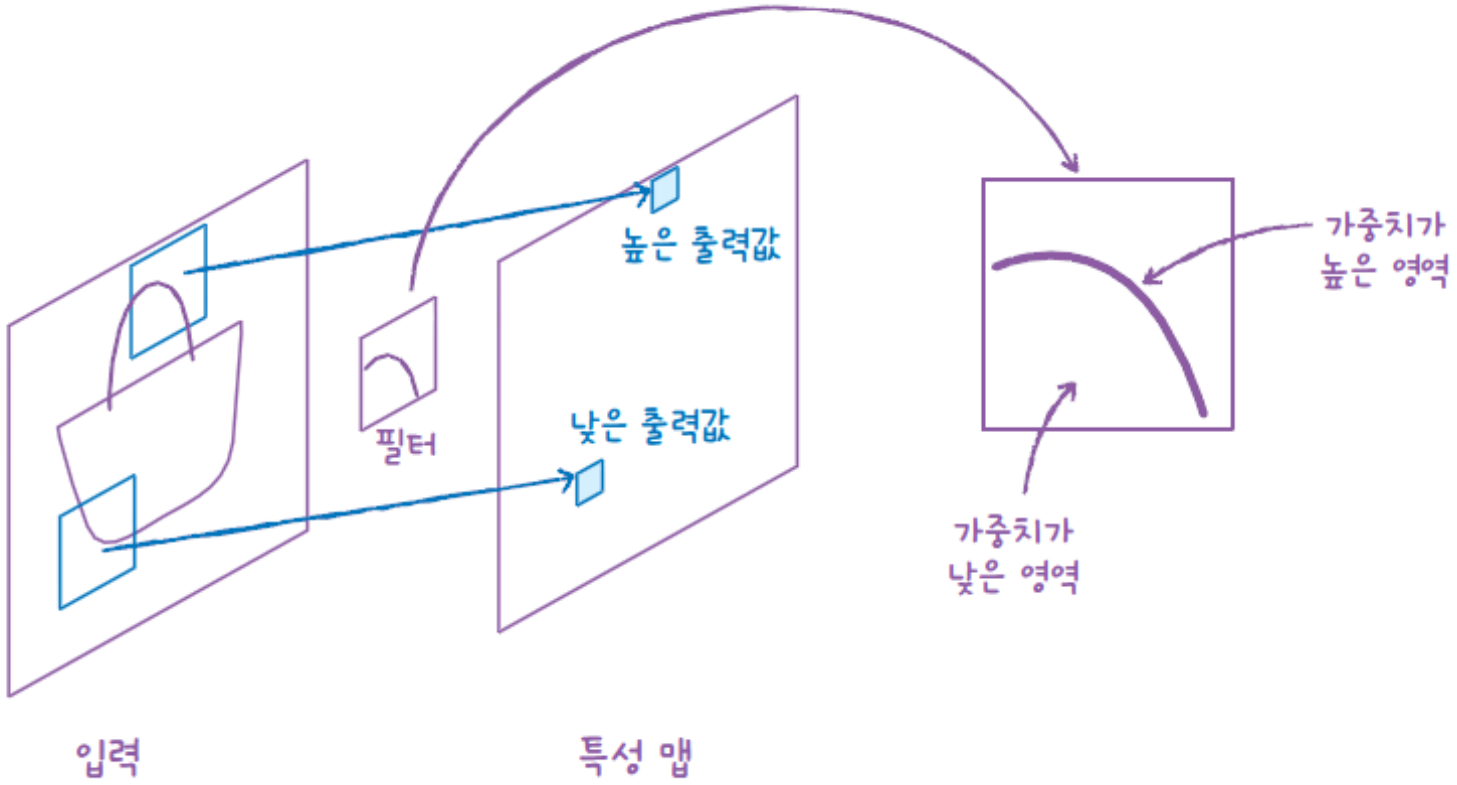
```
model.eval()
corrects = 0
with torch.no_grad():
    for inputs, targets in test_loader:
        inputs, targets = inputs.to(device), targets.to(device)
        outputs = model(inputs)
        predicts = torch.argmax(outputs, 1)
        corrects += (predicts == targets).sum().item()

accuracy = corrects / len(test_dataset)
print(f"테스트 정확도: {accuracy:.4f}")
```

→ 테스트 정확도: 0.9083

SECTION 8-3 합성곱 신경망의 시각화(1)

◦ 가중치 시각화

- 가중치는 입력 이미지의 2차원 영역에 적용되어 어떤 특징을 크게 두드러지게 표현하는 역할



SECTION 8-3 합성곱 신경망의 시각화(2)

- 가중치 시각화
  - 2절에서 만든 모델이 어떤 가중치를 학습했는지 확인하기 위해 체크포인트 파일 읽기

```
import keras
model = keras.models.load_model('best-cnn-model.keras')
```

  - model.layers 출력

```
model.layers
```

→

```
[<Conv2D name=conv2d, built=True>,
<MaxPooling2D name=max_pooling2d, built=True>,
<Conv2D name=conv2d_1, built=True>,
<MaxPooling2D name=max_pooling2d_1, built=True>,
<Flatten name=flatten, built=True>,
<Dense name=dense, built=True>,
<Dropout name=dropout, built=True>,
<Dense name=dense_1, built=True>]
```

SECTION 8-3 합성곱 신경망의 시각화(3)

- 가중치 시각화
  - 첫 번째 합성곱 층의 가중치를 조사 - layers 속성의 첫 번째 원소를 선택해 weights의 첫 번째 원소(가중치)와 두 번째 원소(절편)의 크기를 출력

```
conv = model.layers[0]
print(conv.weights[0].shape, conv.weights[1].shape)
```

→ (3, 3, 1, 32) (32,)

  - weights 속성을 넘파이 배열로 변환. 그다음 가중치 배열의 평균과 표준편차를 넘파이 mean( ) 메서드와 std ( ) 메서드로 계산

```
conv_weights = conv.weights[0].numpy()
print(conv_weights.mean(), conv_weights.std())
```

→ -0.014383553 0.23351653

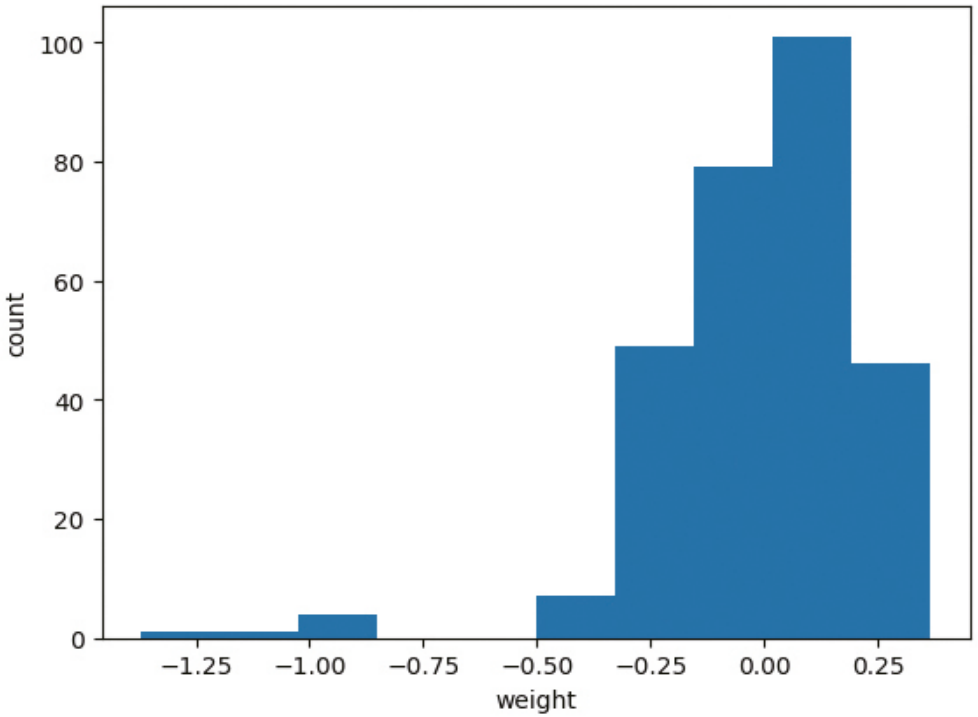
    - 이 가중치의 평균값은 0에 가깝고 표준편차는 0.23 정도

SECTION 8-3 합성곱 신경망의 시각화(4)

◦ 가중치 시각화

• 가중치 분포 히스토그램

```
import matplotlib.pyplot as plt

plt.hist(conv_weights.reshape(-1, 1))
plt.xlabel('weight')
plt.ylabel('count')
plt.show()
```



▲ 맷플롯립의 hist ( ) 함수에는 히스토그램을 그리기 위해 1차원 배열로 전달해야 함
이를 위해 넘파이 reshape 메서드로 conv_weights 배열을 1개의 열이 있는 배열로 변환

SECTION 8-3 합성곱 신경망의 시각화(5)

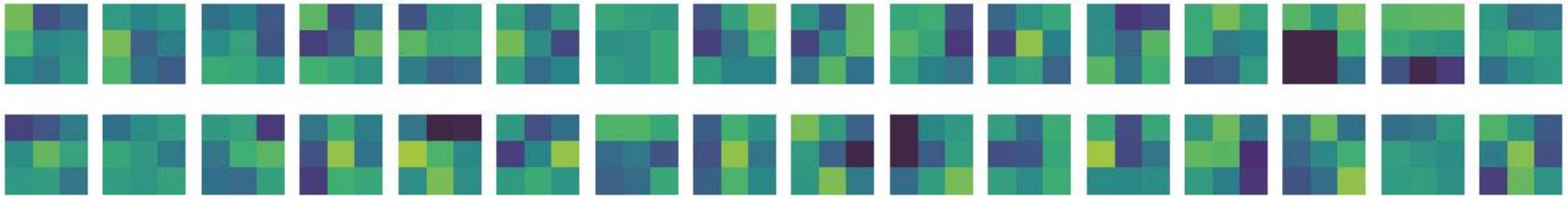
◦ 가중치 시각화

• 32개의 커널을 16개씩 두 줄에 출력

- 맷플롯립의 subplots( ) 함수를 사용해 32개의 그래프 영역을 만들고 순서대로 커널을 출력

```
fig, axs = plt.subplots(2, 16, figsize=(15,2))
for i in range(2):
    for j in range(16):
        axs[i, j].imshow(conv_weights[:,:,0,i*16 + j], vmin=-0.5, vmax=0.5)
        axs[i, j].axis('off')
plt.show()
```

→



SECTION 8-3 합성곱 신경망의 시각화(6)

◦ 가중치 시각화

- 훈련하지 않은 빈 합성곱 신경망을 만들어 이 합성곱 층의 가중치가 위에서 본 훈련한 가중치와 어떻게 다른지 그림으로 비교

• Sequential 클래스로 모델을 만들고 Conv2D 층을 하나 추가

```
no_training_model = keras.Sequential()
no_training_model.add(keras.layers.Input(shape=(28,28,1)))
no_training_model.add(keras.layers.Conv2D(32, kernel_size=3, activation=\
                                          'relu', padding='same'))
```

• 모델의 첫 번째 층(즉 Conv2D 층)의 가중치를 no_training_conv 변수에 저장

```
no_training_conv = no_training_model.layers[0]
print(no_training_conv.weights[0].shape)
```

→ (3, 3, 1, 32)

• 가중치의 평균과 표준편차를 확인

```
no_training_weights = no_training_conv.weights[0].numpy()
print(no_training_weights.mean(), no_training_weights.std())
```

→ 0.0053191613 0.08463709

SECTION 8-3 합성곱 신경망의 시각화(7)

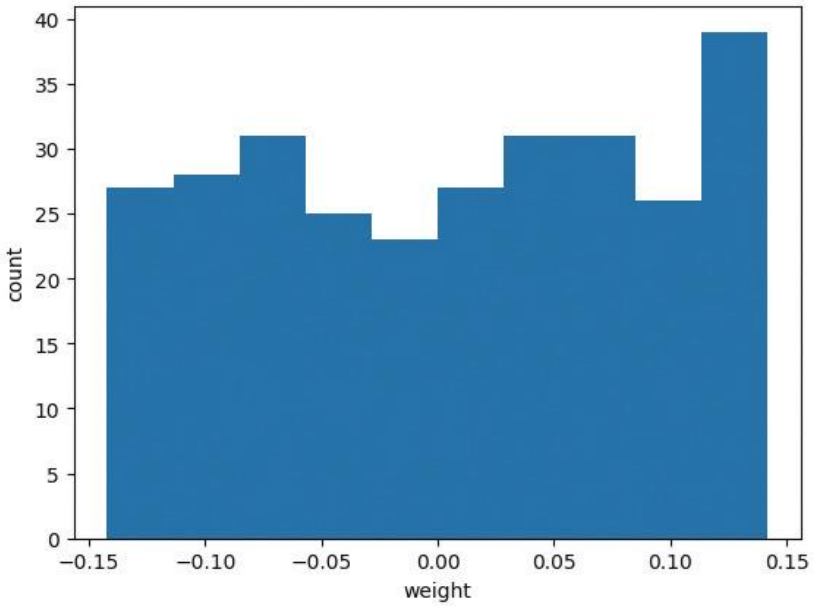
◦ 가중치 시각화

- 훈련하지 않은 빈 합성곱 신경망을 만들어 이 합성곱 층의 가중치가 위에서 본 훈련한 가중치와 어떻게 다른지 그림으로 비교

• 가중치 배열을 히스토그램으로 표현

```
plt.hist(no_training_weights.reshape(-1, 1))
plt.xlabel('weight')
plt.ylabel('count')
plt.show()
```

→



▲ 대부분의 가중치가 -0.15~0.15 사이에 있고 비교적 고른 분포
텐서플로가 신경망의 가중치를 처음 초기화할 때 균등 분포에서 랜덤하게 값을 선택하기 때문임

SECTION 8-3 합성곱 신경망의 시각화(8)

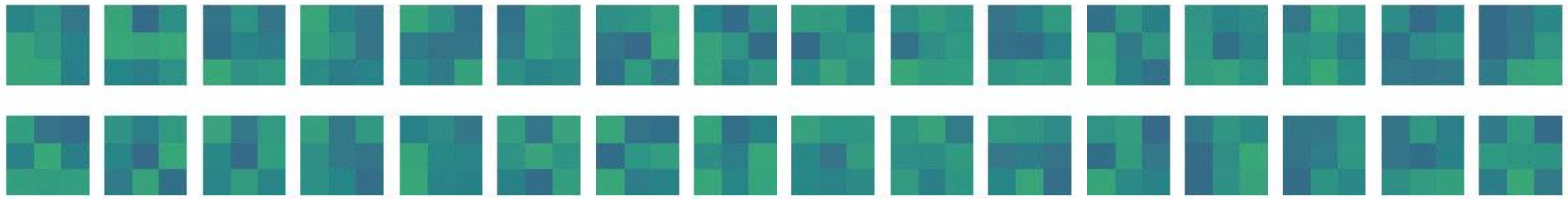
◦ 가중치 시각화

- 훈련하지 않은 빈 합성곱 신경망을 만들어 이 합성곱 층의 가중치가 위에서 본 훈련한 가중치와 어떻게 다른지 그림으로 비교

  • 가중치 값을 맷플롯립의 imshow( ) 함수를 사용해 이전처럼 그림으로 출력
  - 학습된 가중치와 비교하기 위해 동일하게 vmin과 vmax를 -0.5와 0.5로 설정

```
fig, axs = plt.subplots(2, 16, figsize=(15,2))
for i in range(2):
    for j in range(16):
        axs[i, j].imshow(no_training_weights[:,:,0,i*16 + j], vmin=-0.5, vmax=0.5)
        axs[i, j].axis('off')
plt.show()
```

→



SECTION 8-3 합성곱 신경망의 시각화(9)

- 함수형 API(functional API)
  - 케라스 Sequential 클래스는 층을 차례대로 쌓은 모델을 만듦
  - 딥러닝에서는 좀 더 복잡한 모델이 존재(입력이 2개, 출력이 2개 등) - 이런 경우는 Sequential 클래스를 사용하기 어려워 대신 함수형 API를 사용
  - Dense 층 2개로 이루어진 완전 연결 신경망을 함수형 API로 구현
    - 2개의 Dense 층 객체를 생성

```
inputs = keras.Input(shape=(784,))
dense1 = keras.layers.Dense(100, activation='relu')
dense2 = keras.layers.Dense(10, activation='softmax')
```

    - 객체를 함수처럼 호출

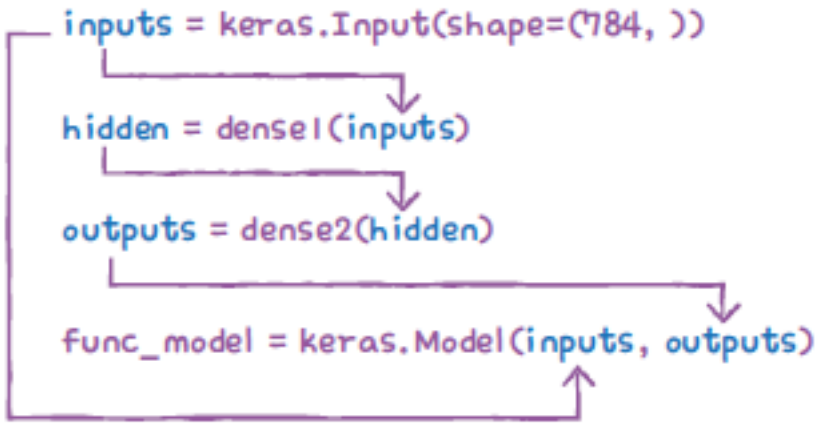
```
hidden = dense1(inputs)
```

    - 두 번째 층을 호출 – 첫 번째 층의 출력을 입력으로 사용

```
outputs = dense2(hidden)
```

    - inputs와 outputs을 Model 클래스로 연결

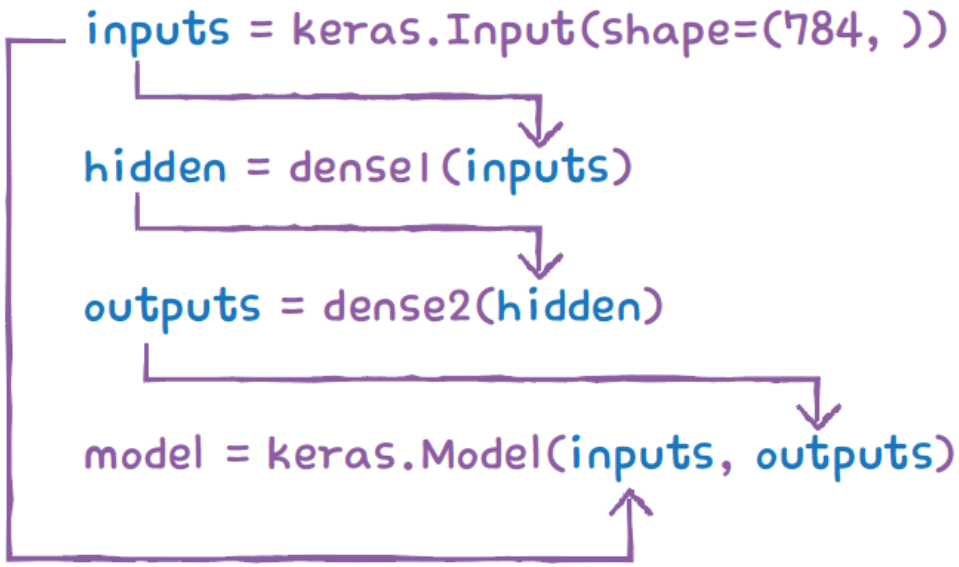
```
func_model = keras.Model(inputs, outputs)
```



SECTION 8-3 합성곱 신경망의 시각화(10)

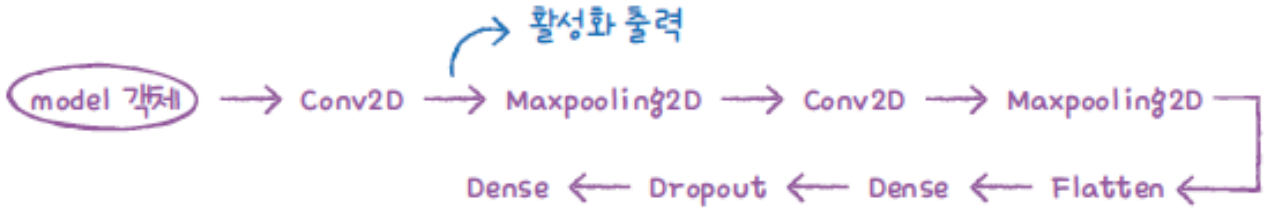
- 함수형 API(functional API)
  - Dense 층 2개로 이루어진 완전 연결 신경망을 함수형 API로 구현
    - 입력의 크기를 지정하는 shape 매개변수와 함께 Input( ) 함수를 호출하면 InputLayer 클래스 객체를 만들어 출력을 반환

```
inputs = keras.Input(shape=(784,))
```



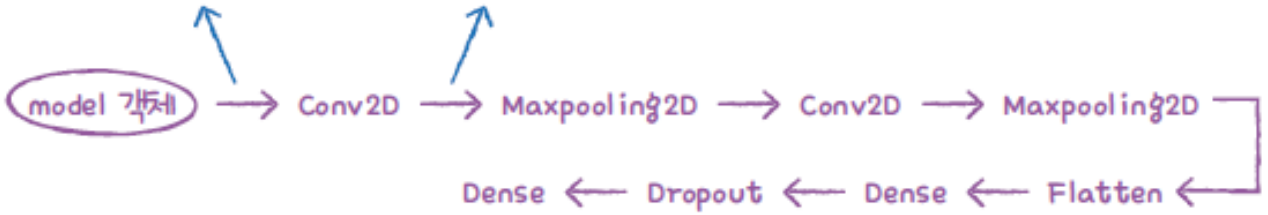
SECTION 8-3 합성곱 신경망의 시각화(11)

- 함수형 API(functional API)
  - 특성 맵 시각화를 만드는 데 함수형 API가 왜 필요한 것일까?
    - 2절에서 정의한 model 객체의 층을 순서대로 나열



    - 우리가 필요한 것은 첫 번째 Conv2D의 출력
      - model 객체의 입력과 Conv2D의 출력을 알수 있다면 이 둘을 연결하여 새로운 모델을 얻을 수 있음

```
conv_acti = keras.Model(model.inputs, model.layers[0].output)
```



SECTION 8-3 합성곱 신경망의 시각화(12)

- 함수형 API(functional API)
  - model 객체의 predict( ) 메서드를 호출하면 입력부터 마지막 층까지 모든 계산을 수행한 후 최종 출력을 반환
  - 첫 번째 층의 출력은 Conv2D 객체의 output 속성에서 획득 – model.layers[0].output
  - model 객체의 입력은 케라스 모델은 input 속성으로 입력을 참조

```
print(model.inputs)
```

→ [<KerasTensor shape=(None, 28, 28, 1), dtype=float32, sparse=False, name=input_layer>]

  - model.input과 model.layers[0].output을 연결하는 새로운 conv_acti 모델 생성

```
conv_acti = keras.Model(model.inputs, model.layers[0].output)
```

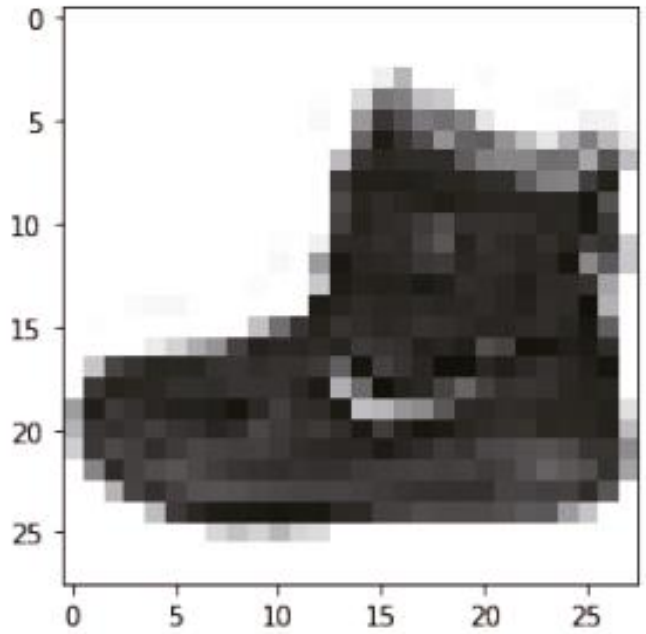
SECTION 8-3 합성곱 신경망의 시각화(13)

◦ 특성 맵 시각화

- 케라스로 패션 MNIST 데이터셋을 읽은 후 훈련 세트에 있는 첫 번째 샘플을 그리기

```
(train_input, train_target), (test_input, test_target) =\
      keras.datasets.fashion_mnist.load_data()
plt.imshow(train_input[0], cmap='gray_r')
plt.show()
```

→



SECTION 8-3 합성곱 신경망의 시각화(14)

◦ 특성 맵 시각화

- 이 샘플을 conv_acti 모델에 주입하여 Conv2D 층이 만드는 특성 맵을 출력

  • 슬라이싱 연산자를 사용해 첫 번째 샘플을 선택, 그다음에 (28, 28) 크기를 (28, 28, 1) 크기로 변경하고 255로 나누기

```
ankle_boot = train_input[0:1].reshape(-1, 28, 28, 1) / 255.0
feature_maps = conv_acti.predict(ankle_boot)
```

  • conv_acti.predict ( ) 메서드가 출력한 feature_maps의 크기를 확인

```
print(feature_maps.shape)
```

→ (1, 28, 28, 32)

▲ 세임 패딩과 32개의 필터를 사용한 합성곱 층의 출력이므로 (28, 28, 32)
첫 번째 차원은 배치 차원 - 샘플을 하나 입력했으므로 1

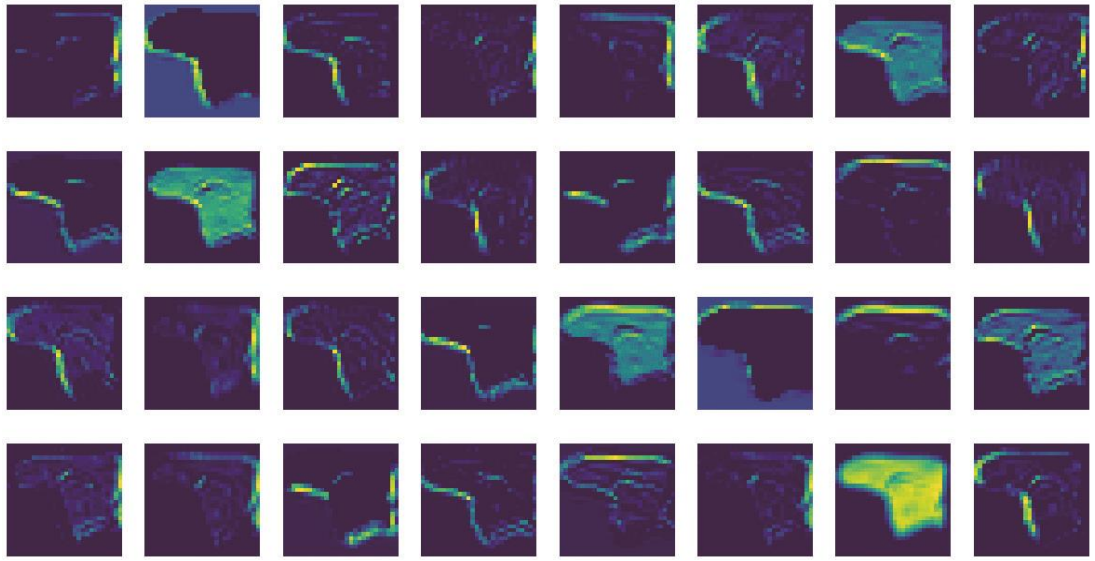
SECTION 8-3 합성곱 신경망의 시각화(15)

◦ 특성 맵 시각화

- 맷플롯립의 imshow 함수로 이 특성 맵을 그리기 – 총 32개의 특성 맵이 있으므로 4개의 행으로 나눔

```
fig, axs = plt.subplots(4, 8, figsize=(15,8))
for i in range(4):
    for j in range(8):
        axs[i, j].imshow(feature_maps[0,:,:,i*8 + j])
        axs[i, j].axis('off')
plt.show()
```

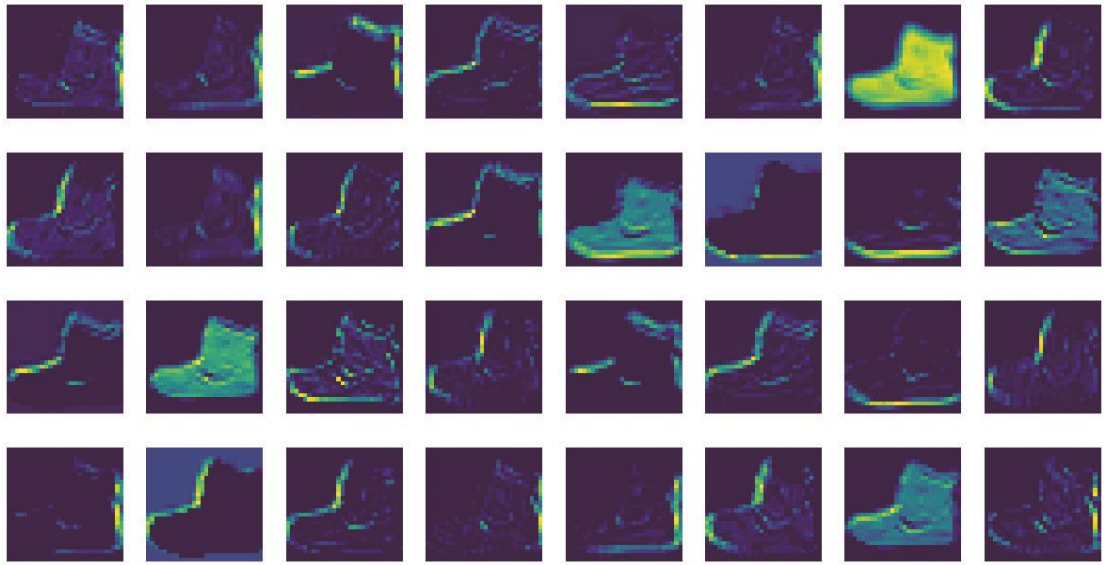
→



SECTION 8-3 합성곱 신경망의 시각화(16)

◦ 특성 맵 시각화

- 앞서 32개 필터의 가중치를 출력한 그림과 몇 개를 비교



- 일곱 번째 필터는 전체적으로 밝은색이므로 전면이 모두 칠해진 영역을 감지
- 일곱 번째 특성 맵에서 이를 잘 확인할 수 있음
- 흑백 부츠 이미지에서 검은 영역이 모두 잘 활성화되어 있음
- 스물 네 번째 필터는 수직선을 감지
- 세 번째 줄의 맨 오른쪽 특성 맵은 이 필터가 감지한 수직선이 강하게 활성화됨

SECTION 8-3 합성곱 신경망의 시각화(17)

◦ 특성 맵 시각화

- 두 번째 합성곱 층이 만든 특성 맵도 같은 방식으로 확인
- model 객체의 입력과 두 번째 합성곱 층인 model.layers[2]의 출력을 연결한 conv2_acti 모델을 생성

```
conv2_acti = keras.Model(model.inputs, model.layers[2].output)
```

- 앵클 부츠 샘플을 conv2_acti 모델의 predict ( ) 메서드에 전달

```
feature_maps = conv2_acti.predict(ankle_boot)
```

- feature_maps의 크기 확인

```
print(feature_maps.shape)
```

→ (1, 14, 14, 64)

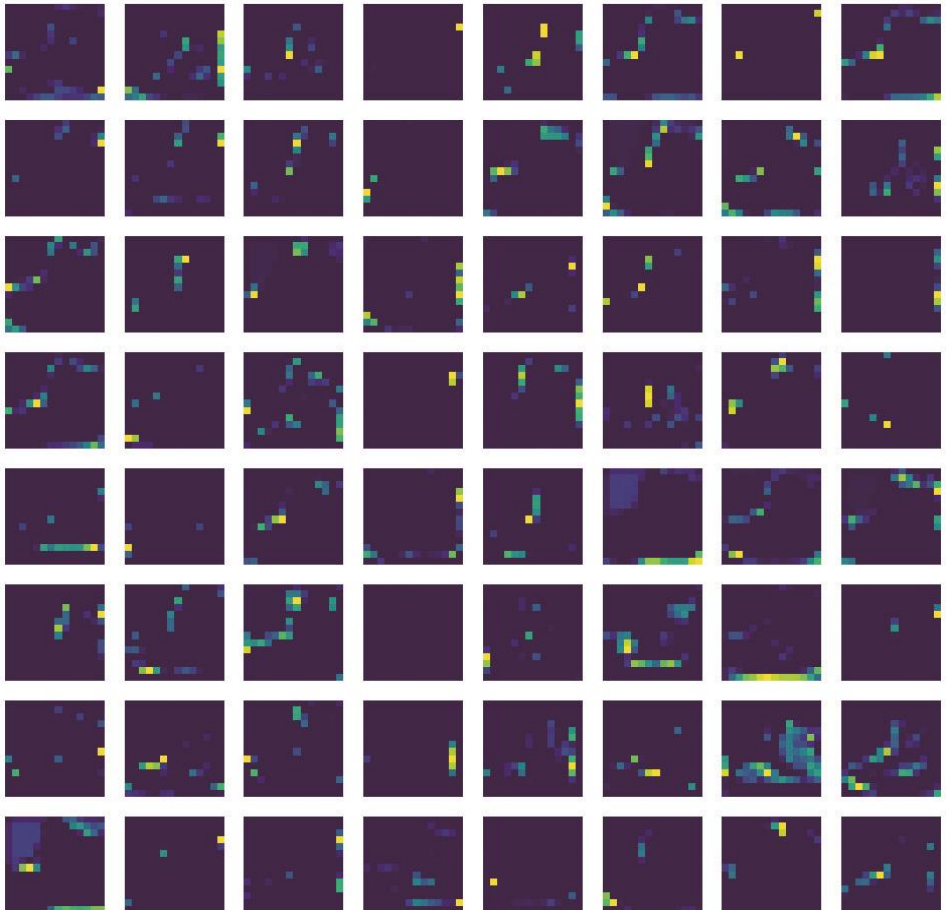
SECTION 8-3 합성곱 신경망의 시각화(18)

◦ 특성 맵 시각화

- 64개의 특성 맵을 8개씩 나누어 imshow( ) 함수로 시각화

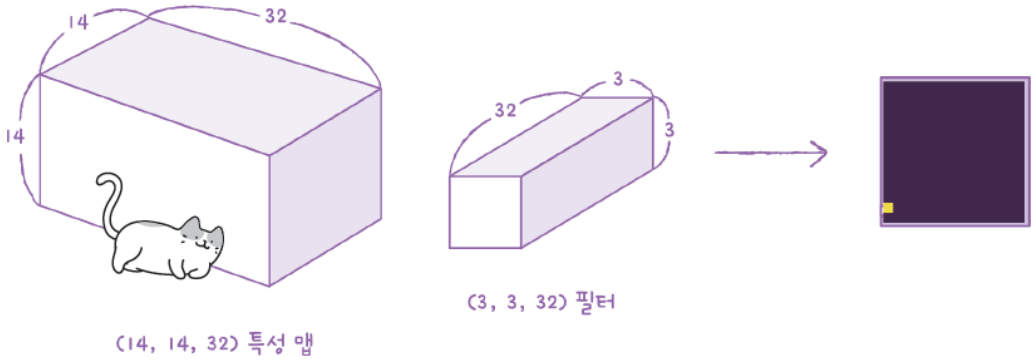
```
fig, axs = plt.subplots(8, 8, figsize=(12,12))
for i in range(8):
    for j in range(8):
        axs[i, j].imshow(feature_maps[0,:,:,i*8 + j])
        axs[i, j].axis('off')
plt.show()
```

→



SECTION 8-3 합성곱 신경망의 시각화(19)

- 특성 맵 시각화
  - 앞의 특성 맵은 시각적으로 이해하기 어려운 결과가 출력됨
    - 두 번째 합성곱 층의 필터 크기는 (3, 3, 32)
    - 두 번째 합성곱 층의 첫 번째 필터가 앞서 출력한 32개의 특성 맵과 곱해져 두 번째 합성곱 층의 첫 번째 특성 맵이 됨
    - 아래의 그림처럼 이렇게 계산된 출력은 (14, 14, 32) 특성 맵에서 어떤 부위를 감지하는지 직관적으로 이해하기가 어려움
    - 이런 현상은 합성곱 층을 많이 쌓을수록 심해짐 – 이를 바꾸어 생각하면 합성곱 신경망의 앞부분에 있는 합성곱 층은 이미지의 시각적인 정보를 감지하고 뒤쪽에 있는 합성곱 층은 앞쪽에서 감지한 시각적인 정보를 바탕으로 추상적인 정보를 학습한다고 볼 수 있음
    - 합성곱 신경망이 패션 MNIST 이미지를 인식하여 10개의 클래스를 찾아낼 수 있는 이유가 바로 여기 있음



SECTION 8-3 합성곱 신경망의 시각화(20)

◦ 시각화로 이해하는 합성곱 신경망(문제해결 과정)

- 학습

  • 훈련하며 저장한 합성곱 신경망 모델을 읽어 들인 후 이 모델의 가중치와 특성 맵을 시각화
  - 이를 통해 합성곱 층이 어떻게 입력에서 특성을 학습하는지 관찰

  • 입력에 가까운 합성곱 층은 이미지에서 시각적인 정보나 패턴을 감지하도록 훈련됨
  - 이어지는 합성곱 층은 이런 시각적인 정보를 활용해 조금 더 고차원적인 개념을 학습
  - 층이 추가될수록 이런 현상은 더욱 강화됨
  - 결국 주어진 이미지가 패션 MNIST 데이터셋에 있는 10개의 클래스 중 어떤 것인지를 판단

  • 특성 맵을 시각화하면서 케라스 API의 핵심 기능 중 하나인 함수형 API를 학습

  • 함수형 API를 사용하면 복잡한 조합의 모델을 자유롭게 구성 가능
  - 여기서는 입력과 합성곱 층의 출력을 연결하여 특성 맵을 시각화하기 위한 용도로 사용

SECTION 8-3 마무리(1)

- 키워드로 끝나는 핵심 포인트
  - 가중치 시각화는 합성곱 층의 가중치를 이미지로 출력하는 것
    - 합성곱 신경망은 주로 이미지를 다루기 때문에 가중치가 시각적인 패턴을 학습하는지 알아볼 수 있음
  - 특성 맵 시각화는 합성곱 층의 활성화 출력을 이미지로 그리는 것
    - 가중치 시각화와 함께 비교하여 각 필터가 이미지의 어느 부분을 활성화시키는지 확인할 수 있음
  - 함수형 API는 케라스에서 신경망 모델을 만드는 방법 중 하나
    - Model 클래스에 모델의 입력과 출력을 지정
    - 전형적으로 입력은 Input( ) 함수를 사용하여 정의하고 출력은 마지막 층의 출력으로 정의

SECTION 8-3 마무리(2)

- 핵심 패키지와 함수
  - Keras
    - Model: 케라스 모델을 만드는 클래스
      - 첫 번째 매개변수인 inputs - 모델의 입력 또는 입력의 리스트를 지정
      - 두 번째 매개변수인 outputs - 모델의 출력 또는 출력의 리스트을 지정
      - name 매개변수 - 모델의 이름을 지정

SECTION 8-3 확인 문제(1)

1. 합성곱 신경망의 첫 번째 합성곱 층에서 다음과 같은 필터가 학습되었다. 이 필터를 사용해 가장 높은 값의 특성 맵을 만들 수 있는 입력은 무엇일까?

① 

② 

③ 

④ 

2. 다음 중 케라스 모델을 만드는 올바른 방법이 아닌 것은?

① model = Sequential ( )
② model = Model(ins, outs)
③ model = Model(inputs=ins, outputs=outs)
④ model = Model( ) (ins, outs)

SECTION 8-3 확인 문제(2)

3. 다음 중 케라스 모델의 입력을 올바르게 참조하는 것은?

① model.input
② model.inputs
③ model.Input
④ model.Inputs

4. 다음 중 합성곱 신경망에 대해 올바르게 설명한 것은?

① 입력에 가까운 합성곱 층은 경계, 색깔 같은 저수준의 특징을 감지하고, 출력에 가까운 합성곱 층은 추상적인 특징을 감지
② 합성곱 신경망의 특성 맵은 층이 거듭될수록 일반적으로 높이와 너비가 커지고 채널이 얕아짐
③ 합성곱의 커널 크기는 클수록 좋음
④ 합성곱 신경망에서는 일반적으로 합성곱 층의 스트라이드를 사용하여 특성맵의 크기를 줄임

SECTION 8-3 파이토치 버전 살펴보기(1)

- 파이토치로 합성곱 신경망 시각화하기
  - 훈련된 가중치와 특성 맵 시각화
    - 08-2절에서 파이토치로 훈련한 모델을 사용
    - 이전 절에서 훈련한 best_cnn_model.pt 파일을 준비
    - 코랩을 사용하는 경우라면 다음 명령으로 깃허브에 저장된 파일을 다운로드

```
!wget https://bit.ly/3DQeEH8 -O best_cnn_model.pt
```

SECTION 8-3 파이토치 버전 살펴보기(2)

- 파이토치로 합성곱 신경망 시각화하기
  - best_cnn_model.pt 파일에는 가중치만 저장되어 있기 때문에,
    1) 이 전에 만든 것과 동일한 모델을 생성한 후
       - Sequential 클래스를 사용해서 이전 절에서 만든 것과 동일한 합성곱 신경망 모델 생성
    2) 이 가중치를 로드
       - best_cnn_model.pt에 저장한 가중치를 로드

```
import torch.nn as nn

model = nn.Sequential()
model.add_module('conv1', nn.Conv2d(1, 32, kernel_size=3,
padding='same'))
model.add_module('relu1', nn.ReLU())
model.add_module('pool1', nn.MaxPool2d(2))
model.add_module('conv2', nn.Conv2d(32, 64, kernel_size=3,
padding='same'))
model.add_module('relu2', nn.ReLU())
model.add_module('pool2', nn.MaxPool2d(2))
model.add_module('flatten', nn.Flatten())
model.add_module('dense1', nn.Linear(3136, 100))
model.add_module('relu3', nn.ReLU())
model.add_module('dropout', nn.Dropout(0.3))
model.add_module('dense2', nn.Linear(100, 10))
```

```
model.load_state_dict(torch.load('best_cnn_model.pt',
                      weights_only=True))
```

SECTION 8-3 파이토치 버전 살펴보기(3)

- 파이토치 모델의 층을 참조하는 방법
  - 리스트 내포 구문을 사용하여 model 객체의 모든 층을 가져오기
    - model 객체의 children ( ) 메서드는 모델에 추가된 층을 반환하는 파이썬 제너레이터 객체를 반환

```
layers = [layer for layer in model.children()]
```

1) layers 리스트를 사용해 model의 층을 참조

예) 이 리스트의 첫 번째 원소는 모델에 가장 처음 추가한 합성곱 층

```
print(layers[0])
```

→ Conv2d(1, 32, kernel_size=(3, 3), stride=(1, 1), padding=same)

2) Sequential 클래스로 만든 모델은 정수 인덱스로 하위 층을 참조

```
model[0]
```

→ Conv2d(1, 32, kernel_size=(3, 3), stride=(1, 1), padding=same)

SECTION 8-3 파이토치 버전 살펴보기(4)

- 파이토치 모델의 층을 참조하는 방법

3) 모델 객체의 named_children ( ) 메서드를 사용하면 층의 이름과 층 객체를 함께 참조

```
for name, layer in model.named_children():
    print(f"{name:10s}", layer)
```

→

```
conv1 Conv2d(1, 32, kernel_size=(3, 3), stride=(1, 1), padding=same)
relu1 ReLU()
pool1 MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_
mode=False)
conv2 Conv2d(32, 64, kernel_size=(3, 3), stride=(1, 1), padding=same)
relu2 ReLU()
pool2 MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_
mode=False)
flatten Flatten(start_dim=1, end_dim=-1)
dense1 Linear(in_features=3136, out_features=100, bias=True)
relu3 ReLU()
dropout Dropout(p=0.3, inplace=False)
dense2 Linear(in_features=100, out_features=10, bias=True)
```

```
model.conv1
```

→ Conv2d(1, 32, kernel_size=(3, 3), stride=(1, 1), padding=same)

SECTION 8-3 파이토치 버전 살펴보기(5)

- 가중치의 평균과 표준편차 계산
  - 파이토치 층의 가중치와 절편은 각각 weight와 bias 속성에 저장
  - 속성은 파이토치의 Parameter 클래스의 객체 - 실제 가중치 텐서는 weight와 bias의 data 속성에 들어 있음
  - 평균과 표준 편차

```
conv_weights = model.conv1.weight.data
print(conv_weights.mean(), conv_weights.std())
```

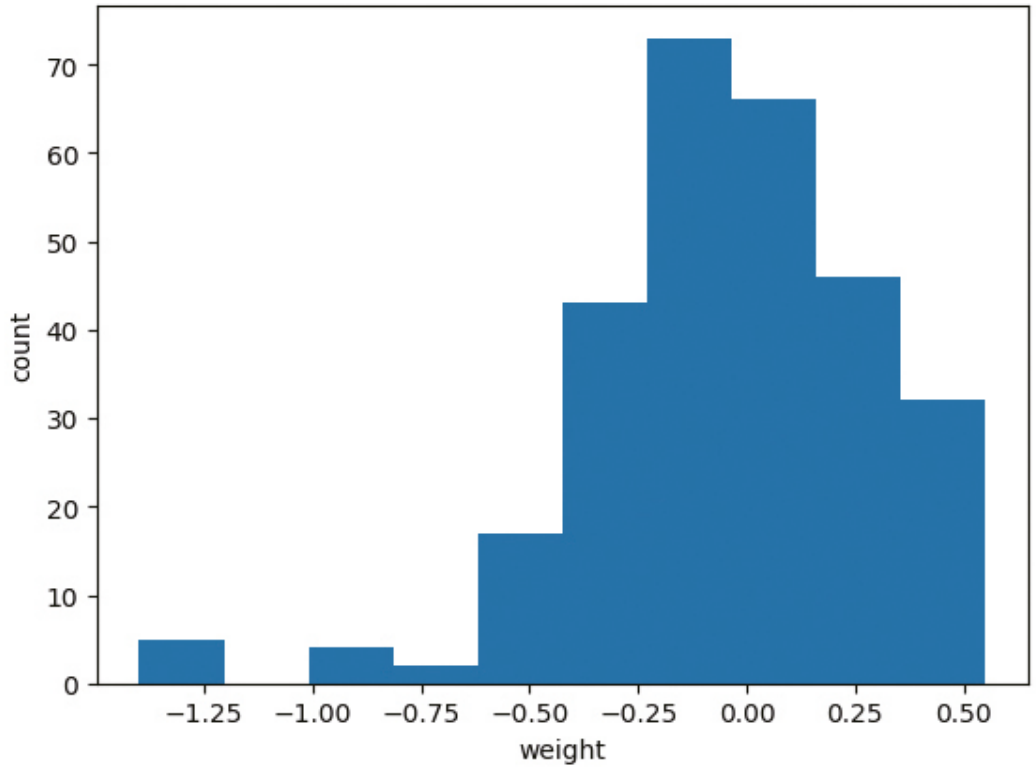
→ tensor(-0.0505) tensor(0.3335)

평균은 0에 가깝고, 표준편차는 약 0.3 정도

SECTION 8-3 파이토치 버전 살펴보기(6)

- 히스토그램으로 가중치 값의 분포 확인

```
import matplotlib.pyplot as plt

plt.hist(conv_weights.reshape(-1, 1))
plt.xlabel('weight')
plt.ylabel('count')
plt.show()
```



SECTION 8-3 파이토치 버전 살펴보기(7)

- 가중치를 그림으로 그리기
  - 주의 - 파이토치는 채널 차원이 가장 먼저 나옴
  - conv_weight 텐서의 크기를 확인해보면, (필터 개수, 채널, 높이, 너비)로 가중치를 나타냄
    -케라스는 기본적으로 가중치를 (높이, 너비, 채널, 필터 개수) 순서임

```
print(conv_weights.shape)
```

→ torch.Size([32, 1, 3, 3])

  - 가중치를 그리는 본문의 코드에서 conv_weights 첨자의 순서를 수정

```
fig, axs = plt.subplots(2, 16, figsize=(15,2))
    for i in range(2):
        for j in range(16):
            axs[i, j].imshow(conv_weights[i*16 + j,0,:,:], vmin=-0.5, vmax=0.5)
            axs[i, j].axis('off')
plt.show()
```

→

SECTION 8-3 파이토치 버전 살펴보기(8)

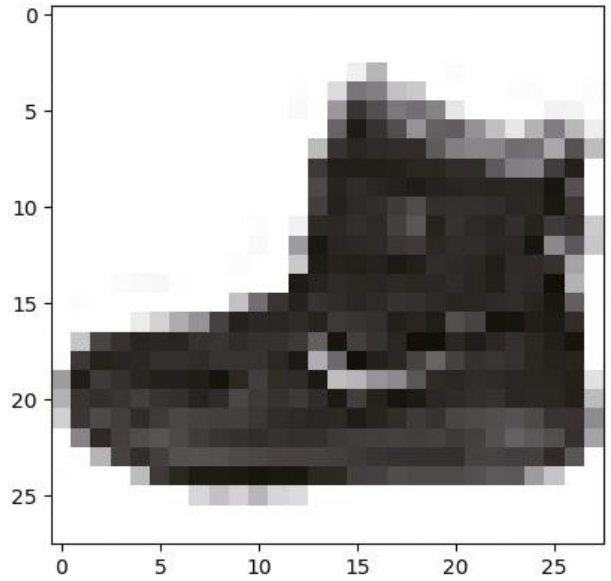
- 패션 MNIST 데이터 중 하나를 모델에 통과시키고 합성곱 층이 어떤 활성화 출력을 만드는지 확인
  - 패션 MNIST 데이터 중에서 훈련 세트만 다운로드

```
from torchvision.datasets import FashionMNIST

fm_train = FashionMNIST(root='.', train=True, download=True)
train_input = fm_train.data
```

  - 본문과 동일하게 훈련 세트에 있는 첫 번째 샘플 그리기

```
plt.imshow(train_input[0], cmap='gray_r')
plt.show()
```

→



SECTION 8-3 파이토치 버전 살펴보기(9)

- 훈련 세트를 배치 차원과 채널 차원을 추가하고 255로 나눈 다음 model.conv1 층에 전달
  - 케라스 모델은 활성화 함수가 층에 포함되어 있지만,
  - 파이토치의 경우 렐루 함수가 별도의 층으로 분리되어 있음
    - model.conv1 층이 반환한 결과를 다시 model.relu1에 전달

```
ankle_boot = train_input[0:1].reshape(1, 1, 28, 28) / 255.0

model.eval()
with torch.no_grad():
    feature_maps = model.conv1(ankle_boot)
    feature_maps = model.relu1(feature_maps)
```

SECTION 8-3 파이토치 버전 살펴보기(10)

- feature_maps의 크기 확인

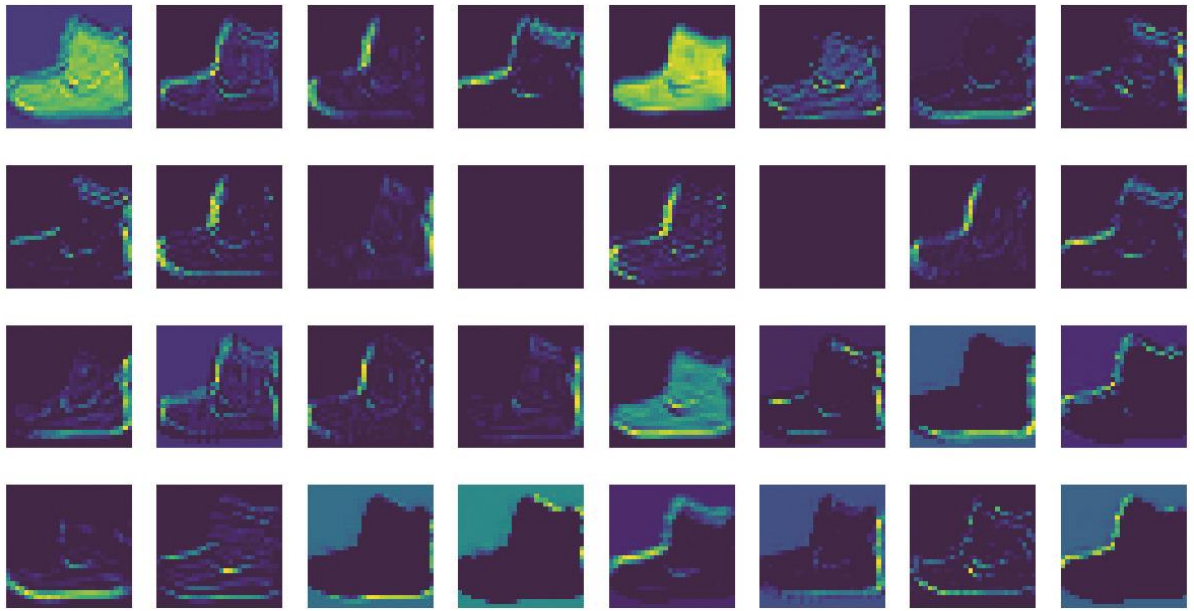
```
print(feature_maps.shape)
```

→ torch.Size([1, 32, 28, 28])

- 특성 맵 그리기 - 맷플롯립을 사용

```
fig, axs = plt.subplots(4, 8, figsize=(15,8))
for i in range(4):
    for j in range(8):
        axs[i, j].imshow(feature_maps[0,i*8 + j,:,:])
        axs[i, j].axis('off')
plt.show()
```

→



SECTION 8-3 파이토치 버전 살펴보기(11)

- 두 번째 합성곱 층이 만드는 특성 맵 그리기
  - 합성곱 층, 렐루 함수, 풀링 층, 합성곱 층, 렐루 함수를 이어서 호출

```
model.eval()
with torch.no_grad():
    feature_maps = model.conv1(ankle_boot)
    feature_maps = model.relu1(feature_maps)
    feature_maps = model.pool1(feature_maps)
    feature_maps = model.conv2(feature_maps)
   feature_maps = model.relu2(feature_maps)
```

SECTION 8-3 파이토치 버전 살펴보기(12)

- 두 번째 합성곱 층이 만드는 특성 맵 그리기
  - 층이 매우 깊다면 named_children ( ) 메서드를 사용해 특정 이름을 가진 층까지만 호출하도록 코드를 작성
  - 다음 코드는 모델의 층을 반복하여 호출하면서 층 이름이 r'elu2'인 경우 반복을 중지

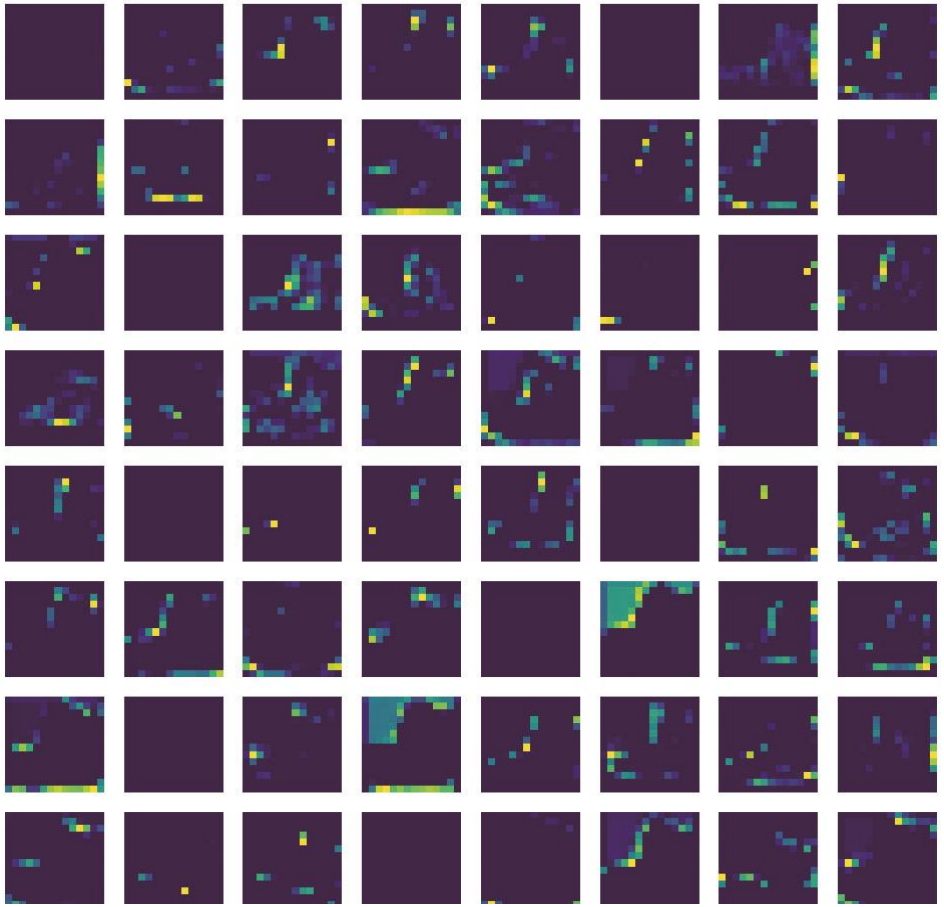
```
model.eval()
x = ankle_boot
with torch.no_grad():
    for name, layer in model.named_children():
        x = layer(x)
        if name == 'relu2':
            break
feature_maps = x
```

SECTION 8-3 파이토치 버전 살펴보기(12)

- 두 번째 합성곱 층이 만드는 특성 맵 그리기
  - feature_maps를 사용해 두 번째 합성곱 층이 만든 특성 맵 그리기

```
fig, axs = plt.subplots(8, 8, figsize=(12,12))
for i in range(8):
    for j in range(8):
        axs[i, j].imshow(feature_maps[0,i*8 + j,:,:])
        axs[i, j].axis('off')
plt.show()
```

→



SECTION 8-3 자주하는 질문

- 이미지 분류 문제를 해결하기 위해 합성곱 신경망을 만들려고 할 때, 합성곱 층을 몇 개나 쌓아야 할까?
- 08-1절의 그림처럼 1차원 데이터에 적용할 수 있는 합성곱 층도 있나?
- 08-2절에서 흑백 이미지에 차원을 추가한다는게 이해가 잘 안됨
  - 그냥 28×28 이미지를 사용하면 안되나?