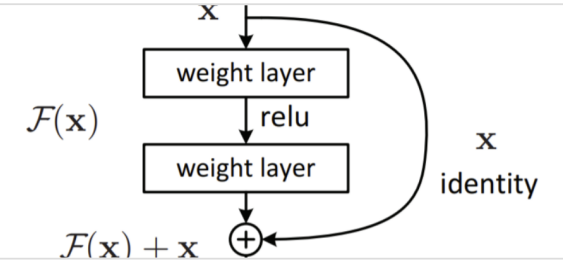


Ch14_합성곱 신경망을 이용한 컴퓨터 비전

Hands-On Machine Learning Chapter 14 합성곱 신경망을 사용한 컴퓨터 비전


<https://velog.io/@wlsn404/Hands-On-Machine-Learning-Chapter-14-합성곱-신경망을-사용한-컴퓨터-비전>



핸즈온 머신러닝 2 복습하기(챕터 14: 합성곱 신경망을 사용한 컴퓨터 비전)

14.1 시각 피질 구조 합성곱 신경망은 시각 피질 안의 많은 뉴런이 작은 국부 수용장(local receptive field)에서 아이디어를 얻었다. 뉴런들이 시야의 일부 범위 안에 있는 시각 자극에만 반응한다는 것이다. 14.2 합성곱 층 CNN의 가장 중요한 구성 요소는 합성곱 층(convolutional layer)이다. 첫 번째 합성곱 층의 뉴런은 입력 이미지의 모든 픽셀에 연결되는 것이


<https://moondol-ai.tistory.com/470>



[핸즈온 머신러닝] chapter 14. 합성곱 신경망

```
In [ ]: # 파이썬 ≥3.5 필수 import sys assert sys.version_info >= (3, 5) # 사이킷런 ≥0.20 필수 import sklearn assert sklearn.__version__ >= "0.20" # 텐서플로 ≥2.0 필수 import tensorflow as tf from tensorflow import keras assert tf.__version__ >= "2.0" # 공통 모듈 임포트 import numpy as np import os # 노트북 실행 결과를 동일하게 유지하기
```

<https://hesh-lumineux.tistory.com/60>



합성곱 신경망은 대뇌의 시각 피질 연구에서 시작되었고, 이미지 인식 분야에서 사용되기 시작했다. 최근 몇 년 동안 컴퓨터 성능의 향상과 많은 양의 훈련데이터, 심층 신경망을 훈련시키기 위한 기술들을 바탕으로 CNN이 일부 복잡한 이미지 처리 문제에서 사람을 능가하는 성능을 달성하기 시작했고, 이 기술은 이미지 검색 서비스, 자율주행 자동차, 영상 자동 분류 시스템 등에 큰 기여를 했다.

▼ 14.1 시각 피질 구조

| 합성곱 신경망(CNN, Convolutional neural network)

합성곱 신경망은 시각 피질 안의 많은 뉴런이 작은 국부 수용장을 가진다는 것에서 아이디어를 얻었다. 뉴런들이 시야의 일부 범위 안에 있는 시각 자극에만 반응한다는 것이다.

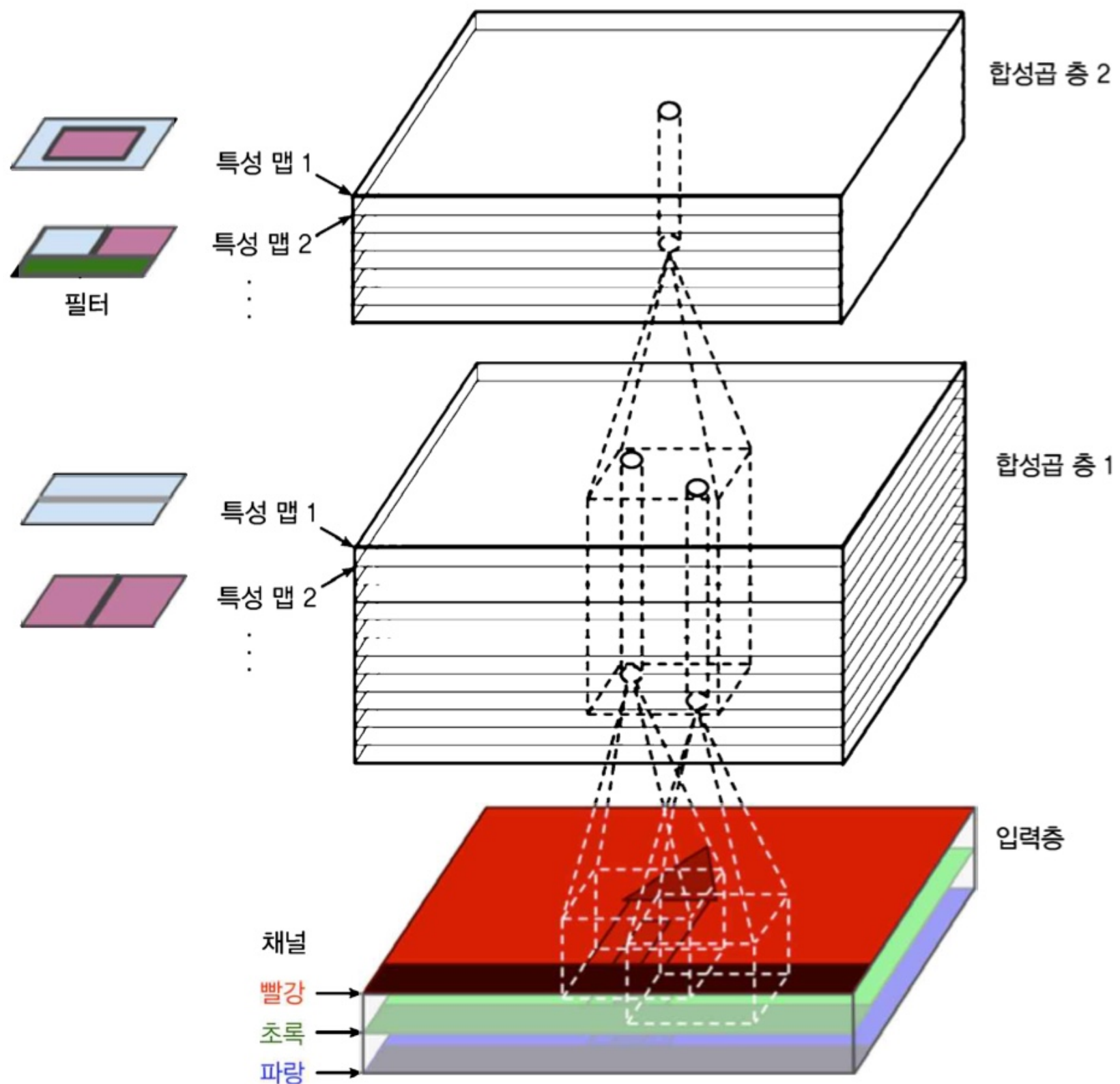
어떤 뉴런은 **수평성의 이미지에만** 반응하거나 반면 다른 뉴런은 **다른 각도의 선분에** 반응한다는 점을 보였다. 또한 어떤 뉴런은 **큰 수용장을 가져서 저수준 패턴이 조합된 더 복잡한 패턴에** 반응한다는 것을 알았고, 이러한 관찰은 **고수준 뉴런이 이웃한 저수준 뉴런의 출력에** 기반한다는 아이디어를 이끌어냈다.

이러한 발견은 이후 합성곱 신경망으로 점진적으로 진화되었고, 1998년 얀 르쿤의 논문에 의해 전환점을 맞게 되었다. 여기서 **합성곱 층(Convolution layer)**와 **풀링 층(Pooling layer)** 라는 새로운 구성 요소가 등장하게 된다.

▼ 14.2 합성곱 층

| 합성곱 층

CNN의 가장 중요한 구성 요소이다.



첫 번째 합성곱 층의 뉴런은 입력 이미지의 모든 픽셀에 연결되는 것이 아니라 합성곱 층 뉴런의 수용장 안에 있는 픽셀에만 연결되고 반응한다.

두 번째 합성곱 층에 있는 각 뉴런은 첫 번째 층의 작은 사각 영역 안에 위치한 뉴런에 연결된다.

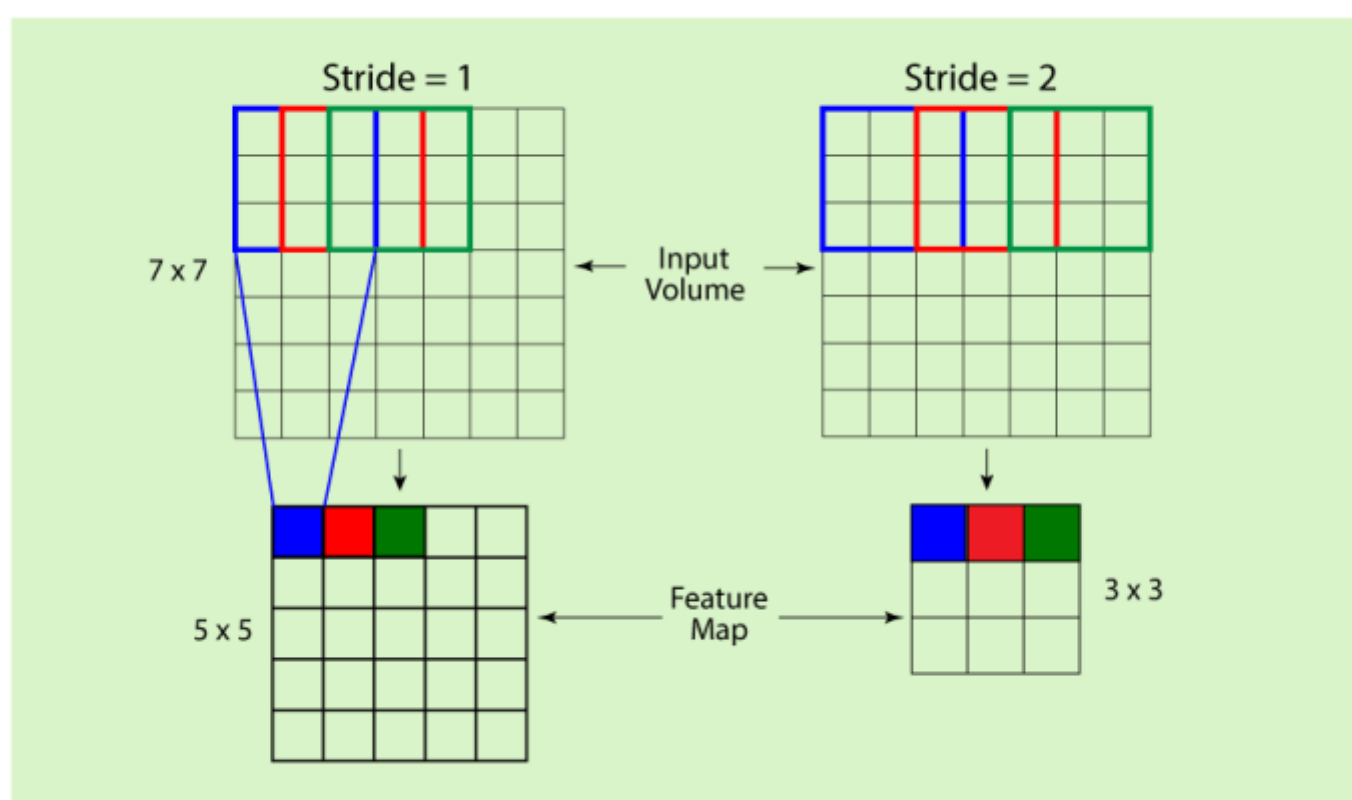
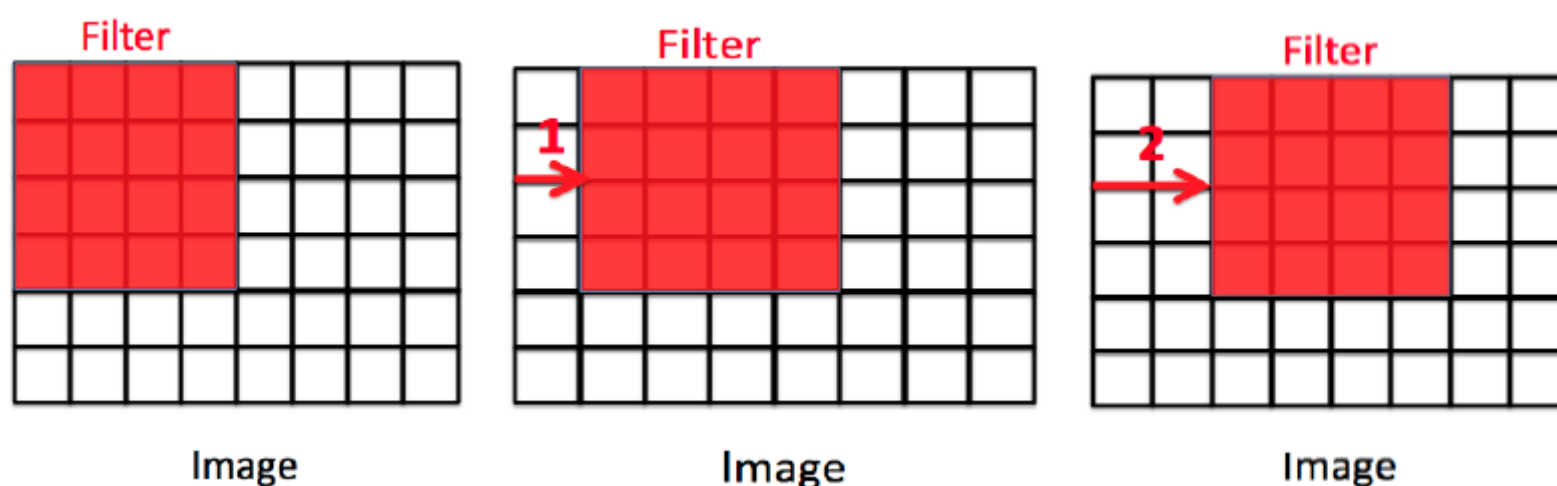
합성곱 신경망에서 해당 수용장을 **필터(filter)**라고 한다. CNN에서 필터와 커널(Kernel)은 혼용되어 쓰이는데, 엄격히 구분하면 필터는 여러 개의 커널로 구성되어 있다. 채널 수가 곧 커널 수가 되고, 커널이 모여 하나의 필터를 구성한다.

이런 뉴런들이 계속해서 이어지며 실제 시각 피질구조와 유사한 구조를 가지게 된다. 이러한 구조는 네트워크가 첫 번째 은닉층에서는 저 수준 특성에 집중하고, 그 다음 은닉층에서는 더 큰 고수준 특성으로 조합해나가도록 도와준다. 이런 계층적 구조가 CNN이 이미지 인식에 잘 작동하는 이유 중 하나이다.

스트라이드(stride)

- 입력 데이터(원본 이미지 or 특성 맵)에 필터를 적용할 때 **sliding window**가 이동하는 간격
- 한 수용장과 다음 수용장 사이의 간격

즉, 필터를 움직일 때 몇 만큼의 간격을 두고 이동(shift)해서 입력 데이터를 추출할 것인지 정하는 것이다.

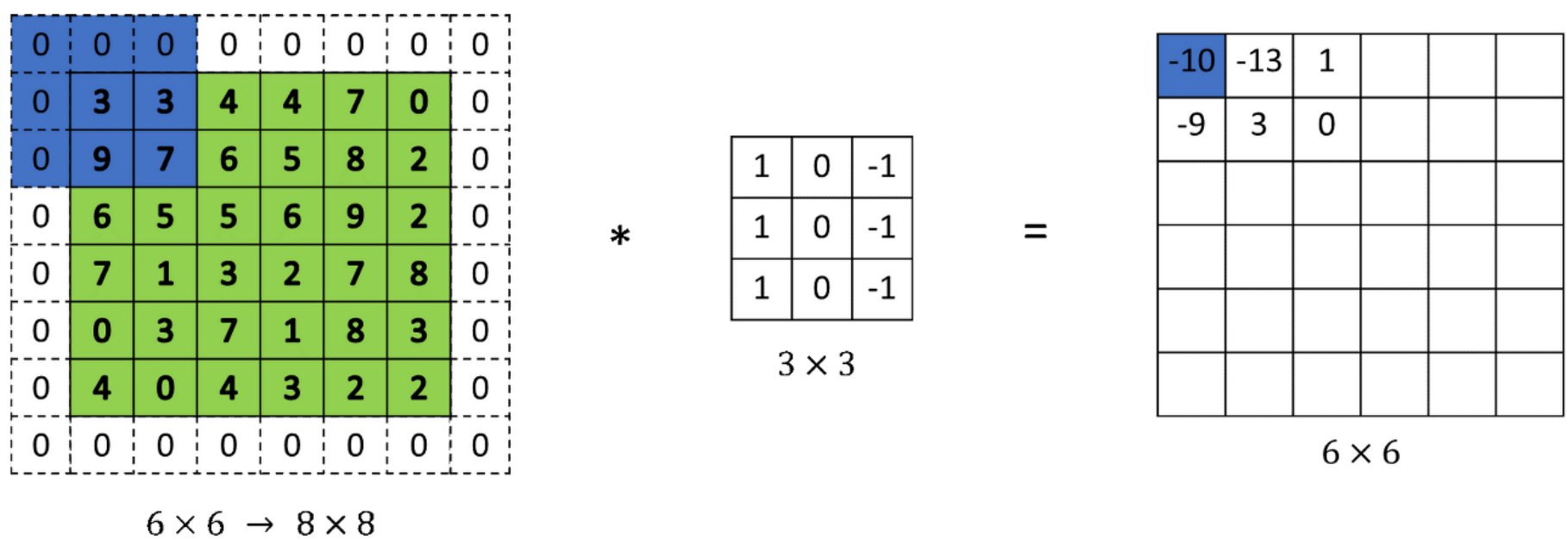
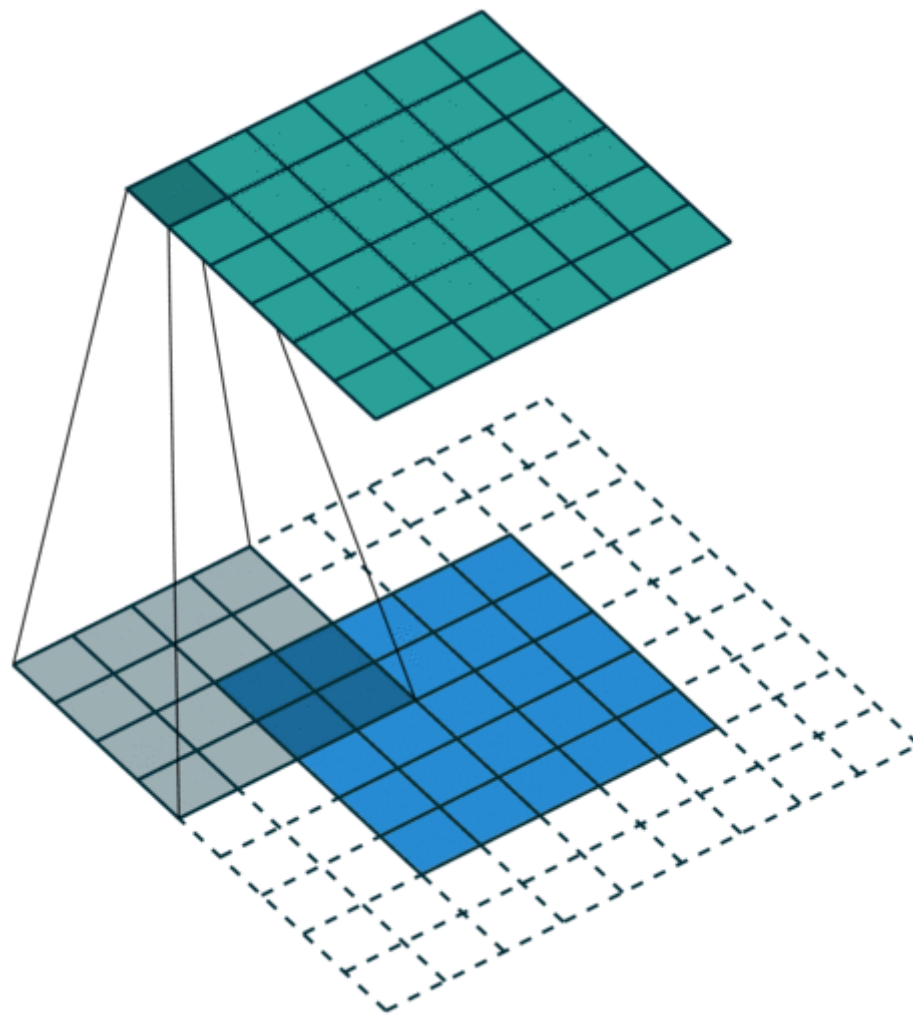


slide=1인 경우는 한 칸씩 움직이고, slide=2인 경우는 두 칸씩 이동한다. 이렇게 하면 공간적인 특성을 손실할 가능성이 높아지지만, **모델의 계산 복잡도를 크게 낮추고 연산 속도를 향상시키는** 효과가 있다.

제로 패딩(zero padding)

합성곱 연산 수행 시 출력 특성 맵(feature map)이 특성 맵 대비 계속 작아지는 것을 막기 위한 기법이다.

합성곱 층에서 아래의 그림과 같이 높이와 너비를 이전 층과 같게 하기 위해, 필터 적용 전에 보존하고자 하는 특성 맵 크기 만큼 입력 특성 맵의 주위(상하좌우 끝)에 0을 추가(공백 부분을 0으로 채워두는 것) 하는 것이다.



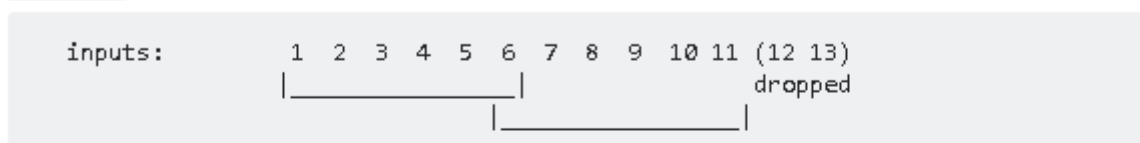
ex. 원본 입력 특성 맵 크기가 6X6인데, 상하좌우에 0을 각각 채움으로써 8X8 크기로 만든다. 커널 크기는 3X3으로 패딩을 하지 않았다면 출력 크기가 4X4로 되었겠지만, 패딩을 함으로써 원본 특성 맵 크기를 유지한다.

- padding 옵션

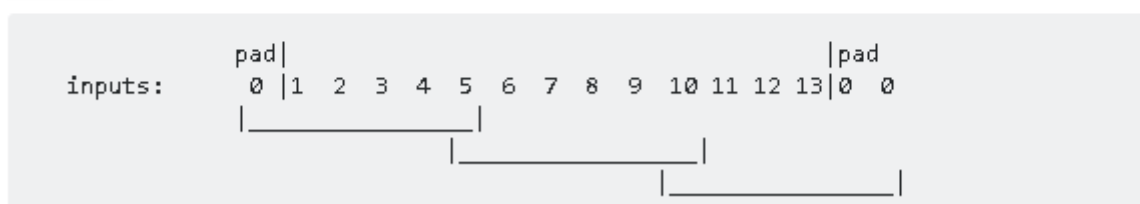
`padding='VALID'` : 패딩이 없다. 즉, 수용장이 입력을 안벗어나서 dropped value가 생긴다

`padding='SAME'` : 필요한 경우 패딩이 추가되고 스트라이드가 1이라면 입력과 출력의 크기가 같아진다.

`"VALID"` = without padding:



`"SAME"` = with zero padding:



- 입력 특성 맵, 필터 크기, 패딩, 스트라이드를 알면 출력 특성 맵의 크기를 계산할 수 있다.

$$O = (1 - F + 2P) / S + 1 \text{ (채널 수는 고려하지 않는다)}$$

I : 입력 특성 맵 크기

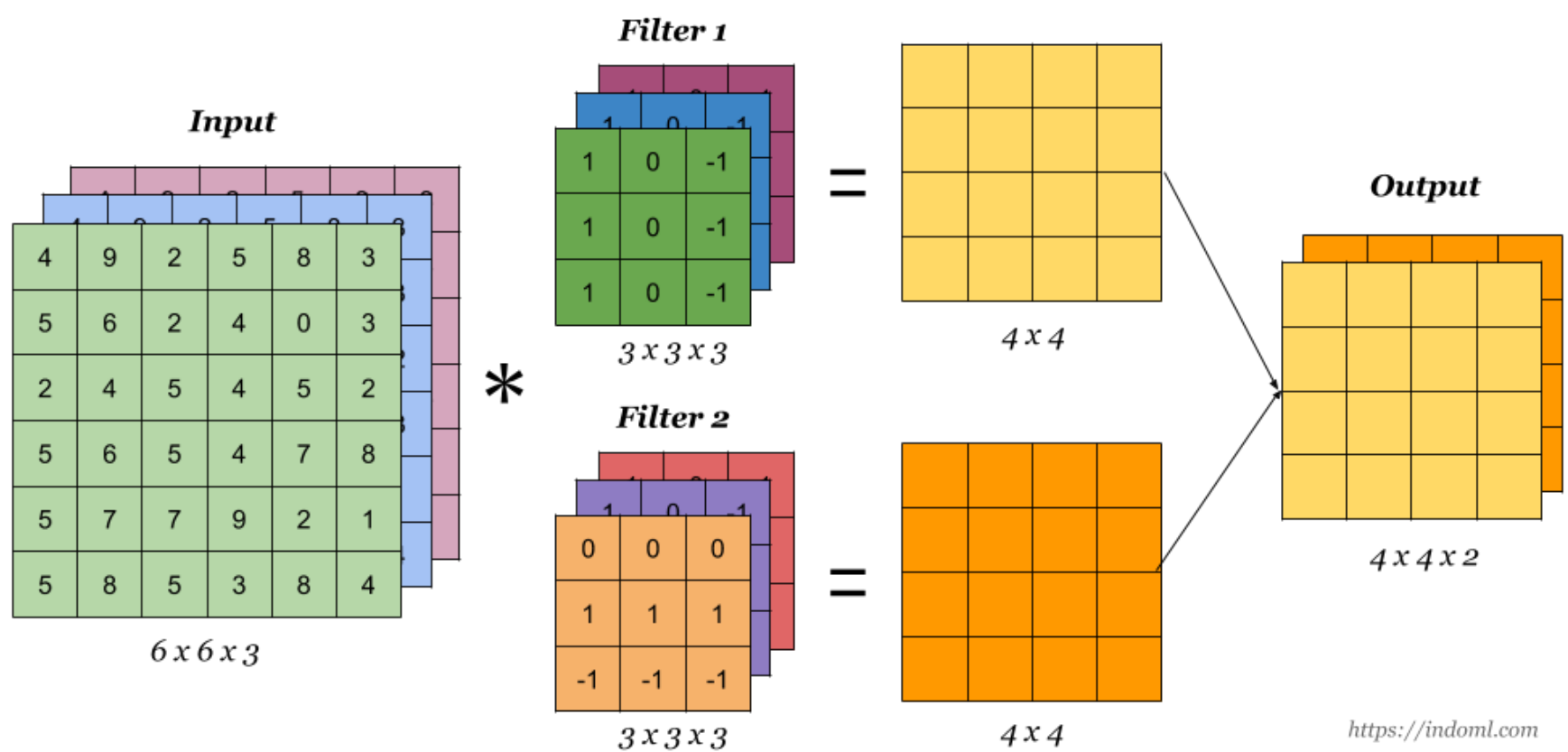
F : 필터 크기

P : 패딩 ($P = 1$: 상하좌우에 1개씩)

S : 스트라이드

▼ 14.2.1 필터

출력 특성 맵의 채널 수는 필터 개수로 결정된다. 필터는 여러 개의 커널(채널)로 이루어져 있고, 이런 필터가 몇 개 있는지가 곧 출력 특성 맵의 채널 수를 결정한다.



커널 크기는 보통 홀수(3x3, 5x5 등)로 설정한다. 만약 소수점이 나오면 소수점은 버려진다. 필터값은 수동으로 정의할 필요 없이 훈련하는 동안 합성곱 층이 자동으로 최적의 값을 찾는다.

▼ 14.2.2 여러 가지 특성 맵 쌓기

출력 특성 맵은 여러 채널(필터 갯수)을 가질 수 있다. 즉, 하나의 합성곱 층이 입력에 따라 여러 필터를 동시에 적용하여 입력에 있는 여러 특성을 감지할 수 있다는 것을 의미한다. 각 특성 맵의 픽셀은 하나의 뉴런에 해당하는데, 하나의 특성 맵 안에선 모든 뉴런이 같은 파라미터(가중치, 편향)를 공유하지만 다른 특성 맵에 있는 뉴런은 다른 파라미터를 사용한다. 한 특성 맵에 있는 모든 뉴런이 같은 파라미터를 공유하기 때문에 모델의 전체 파라미터 수를 급격하게 줄여준다는 장점이 있다.

합성곱 층에 있는 뉴런의 출력을 계산하는 수식은 다음과 같다.

$$z_{i,j,k} = b_k + \sum_{u=0}^{f_h-1} \sum_{v=0}^{f_w-1} \sum_{k'=0}^{f'_n-1} x_{i',j',k'} \times w_{u,v,k',k}$$

,where $i' : i \times s_h + u$

$j' : j \times s_w + v$

$z_{i,j,k}$: 합성곱 층(l 층)의 k 특성 맵에서 i 행, j 열에 위치한 뉴런의 출력

s_h, s_w : 수직과 수평 스트라이드

f_h, f_w : 필터의 높이와 너비

f'_n : 이전 층($l-1$ 층)에 있는 특성 맵 수

$x_{i',j',k'}$: $l-1$ 층의 i' 행, j' 열, k' 특성 맵(혹은 $l-1$ 층이 입력층이면 k' 채널)에 있는 뉴런의 출력

b_k 는 (l 레이어에 있는) k 특성 맵의 편향, 이를 k 특성 맵의 전체 밝기를 조정하는 다이얼로 생각할 수 있음

$w_{u,v,k',k}$: l 층의 k 특성 맵에 있는 모든 뉴런과 (뉴런의 필터에 연관된) u 행, v 열, k' 특성 맵에 위치한 입력 사이의 연결 가중치

▼ 14.2.3 텐서플로 구현

텐서플로에서의 표현을 다음과 같다.

- 입력 이미지 : [높이,너비,채널] 형태의 3D 텐서로 표현
- 하나의 미니배치 : [미니배치 크기, 높이, 너비, 채널]형태의 4D 텐서로 표현
- 합성곱 층의 가중치 : [f_h,f_w,f_n',f_n] 형태의 4D 텐서로 표현

- 사이킷런의 `load_sample_images()` 를 사용하여 두 개의 샘플 이미지 로드

```
import numpy as np
from sklearn.datasets import load_sample_image

# 샘플 이미지 로드

china=load_sample_image("china.jpg")/225
flower=load_sample_image("flower.jpg")/225
images=np.array([china,flower])

batch_size,height,width,channels= images.shape

#2 개의 필터 생성
filters=np.zeros(shape=(7,7,channels,2), dtype=np.float32)
filters[:,3:,0]=1 #수직선
filters[3, :, :, 1]=1 #수평선

outputs=tf.nn.conv2d(images,filters,strides=1,padding='SAME')

plt.imshow(outputs[0, :, :, 1], cmap="gray") #첫 번째 이미지의 두 번째 특성맵 그리기
plt.axis("off") #축에 없음
plt.show()
```




- 각 컬러 채널의 픽셀 강도는 0에서 255 사이의 값을 가진 바이트 → 하나로 표현. 이 특성을 255로 나누어 0에서 1 사이의 값으로 바꾼다.
- 두 개의 7X7 필터를 만든다. 하나는 가운데 흰 수직선, 하나는 가운데 흰 수평선이 존재한다.
- `tf.nn.conv2d()` 함수를 이용해 제로 패딩 및 스트라이드 1 사용

- `tf.nn.conv2d()`

: `images` 는 입력의 미니배치 (4D 텐서)

: `Filters` 는 적용될 일련의 필터 (4D 텐서)

: `strides` 는 1이나 4개의 원소를 갖는 1D배열로 지정할 수 있다. 이때 1D배열의 가운데 두 개의 원소는 수직, 수평 스트라이드이고, 현재는 첫 번째와 마지막 원소가 1이어야 한다.

`padding = "VALID"` : 제로 패딩을 사용X

`padding="SAME"` : 제로 패딩 사용 O

→ 만들어진 특성맵 중 하나를 그래프로 그린다.

- `keras.layers.Conv2D`
- 위의 예시는 필터를 직접 지정하였지만, 실제로 CNN에서는 보통 훈련 가능한 변수로 필터를 정의하여 신경망이 가장 잘 맞는 필터를 학습한다. 이때 `keras.layers.Conv2D` 를 사용한다.

```
conv = keras.layers.Conv2D(filters=32, kernel_size=3, strides=1,
                             padding='same', activation='relu')
```

- 3X3 크기의 32개 필터, 수평과 수직 방향의 스트라이드 1을 사용한다.
- 제로 패딩을 사용하는 Conv2D를 만들고, 출력을 위해 ReLU 활성화 함수를 적용한다.

▼ 14.2.4 메모리 요구 사항

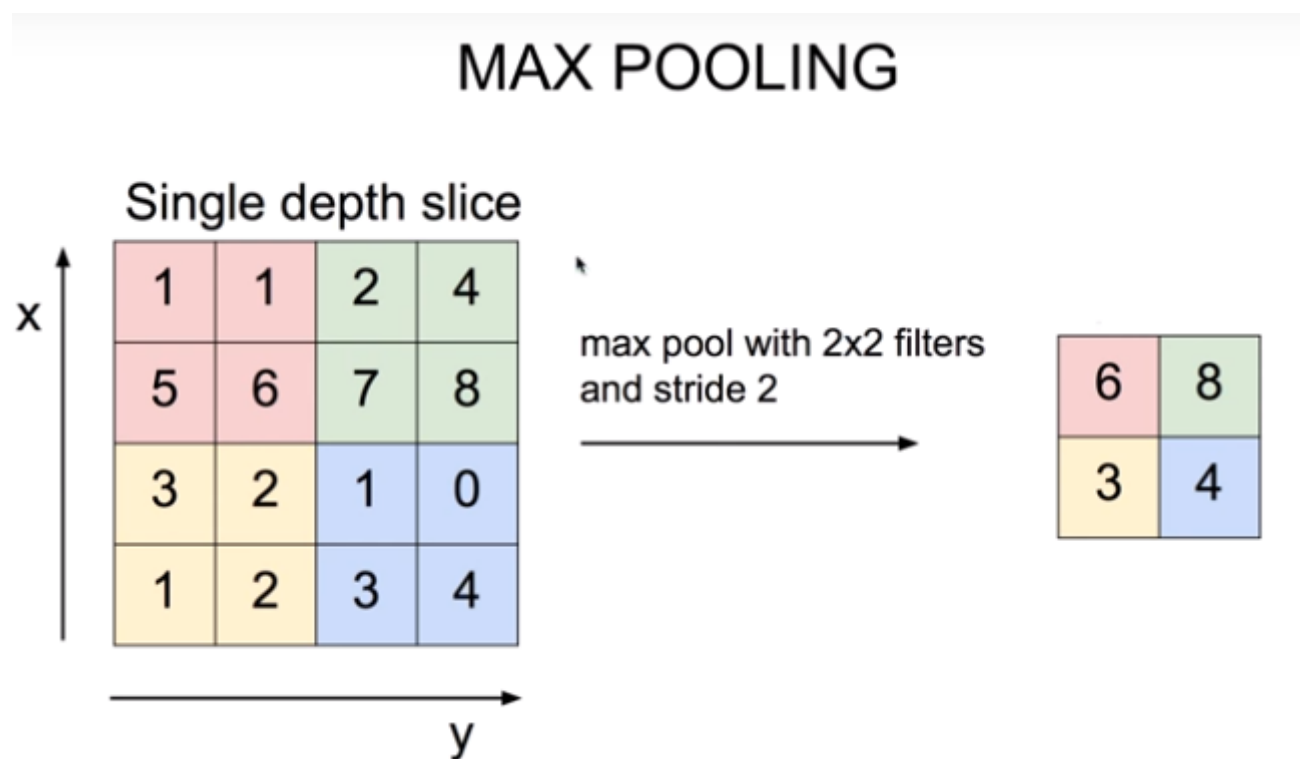
CNN의 문제 중 하나로, 합성곱 층이 많은 양의 RAM을 필요로 한다.
메모리 부족으로 훈련이 실패하다면 미니배치 크기를 줄이거나 스트라이드를 사용해 차원을 줄이는 방법 등을 사용할 수 있다.

▼ 14.3 풀링 층

풀링 층의 목적은 **계산량과 메모리 사용량, 파라미터의 수(결과적으로 과대적합을 줄여주는)**를 줄이기 위해 입력 이미지의 부표본 (subsample)을 만드는 것이다.

풀링 층의 각 뉴런은 이전 층의 작은 사각 영역의 수용장 안에 있는 뉴런의 출력과 연결되어 있어 이전과 동일하게 크기, 스트라이드, 패딩 유형을 지정해야 하지만 가중치는 없다. 최대나 평균과 같은 합산 함수를 사용해 입력값을 더하는 것이 전부다.

MAX POOLING



- 합성곱 층을 통해 출력된 특성 맵에 2X2 크기의 풀링 층(스트라이드 2)을 적용한 결과.
- 가장 널리 이용되는 최대 풀링 층(max pooling layer)으로 해당 영역에서 가장 큰 값을 추출한다.

• 최대 풀링의 장점

최대 풀링은 작은 변화에도 일정 수분의 불변성을 만들어준다. 불변성은 분류 작업처럼 예측이 이런 작은 부분에서 영향을 받지 않는 경우 유용할 수 있다.

• 최대 풀링의 단점

최대 풀링은 파괴적이기 때문에 2X2 필터와 스트라이드 2를 사용해도 출력은 양방향으로 절반이 줄어들어 입력값의 75%를 잃는다. 시맨틱 분할의 경우 불변성이 필요하지 않고 동변성이 목표가 되므로 입력의 작은 변화가 출력에서 그에 상응되는 작은 변화로 이어져야 한다.

▼ 14.3.1 텐서플로 구현

- 텐서플로로 최대 풀링 층 구현

```
max_pool = keras.layers.MaxPool2D(pool_size=2)
```

이 코드를 통해 2X2 커널을 사용하여 최대 풀링 층을 만든다. 스트라이드의 기본값은 커널 크기이므로, 이 층은 모두 스트라이드 2를 사용하고 기본적으로 padding='valid'(패딩 사용 X)을 사용한다.

평균 풀링 층(Average pooling layer)

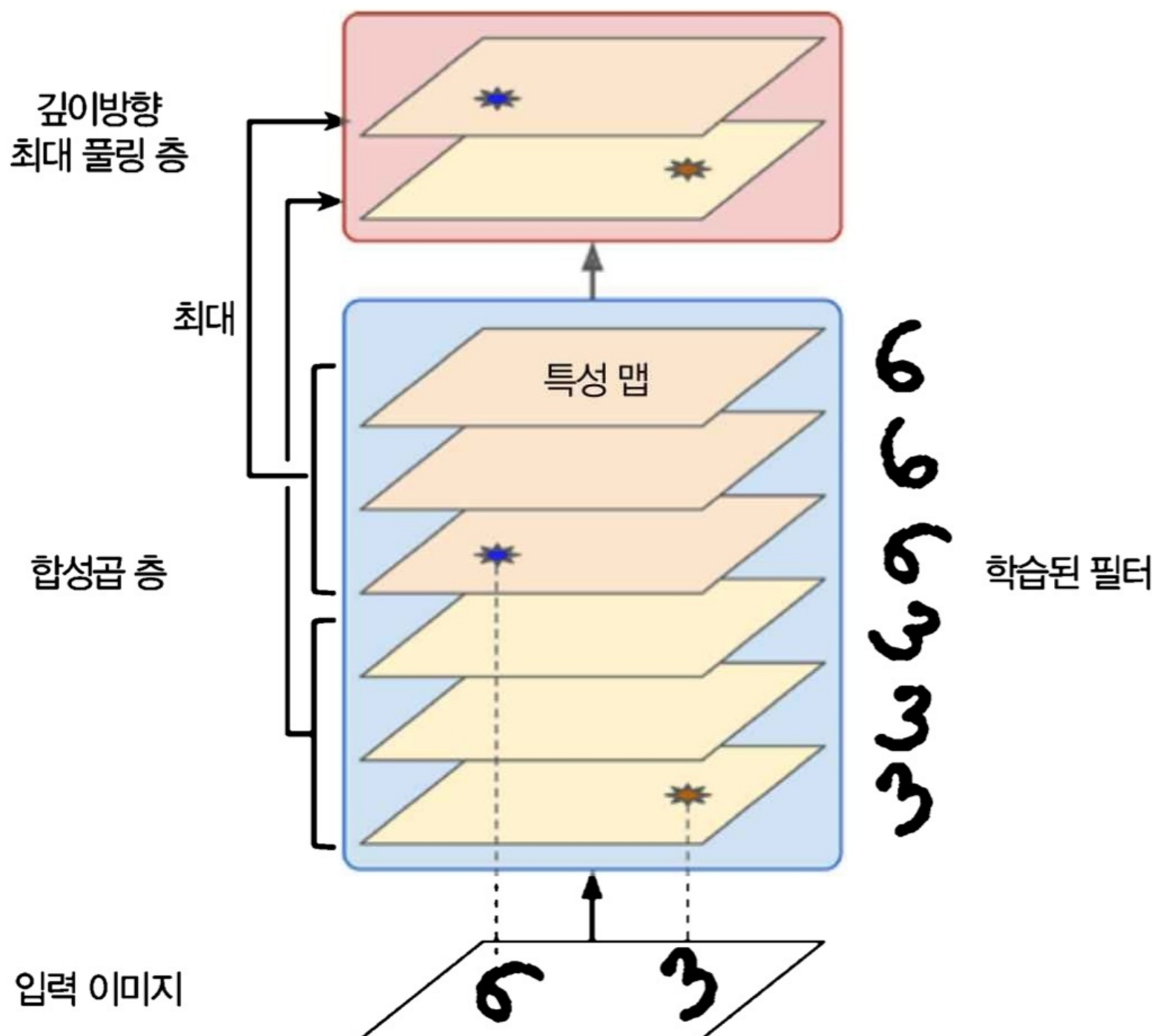
평균 풀링 층(Average pooling layer)을 만드려면 `MaxPool2D` 대신 `AvePool2D` 을 사용한다. 최대값이 아닌 평균을 계산한다.

일반적으로 성능은 최대 풀링이 더 좋지만, 평균을 이용하게 되면 정보 손실이 더 적다는 장점이 있다.

그러나 최대 풀링은 의미 없는 것은 모두 제거하고 가장 큰 특징만 유지하므로 다음 층이 조금 더 명확한 신호로 작업할 수도 있으며, 최대 풀링은 평균 풀링보다 강력한 이동 불변성을 제공하고 연산 비용이 조금 덜 든다.

최대 풀링과 평균 풀링은 공간 차원이 아니라 깊이 차원으로 수행될 수 있다. 이를 통해 CNN은 다양한 특성에 대한 불변성을 학습할 수 있다.

그림과 같이 입력 이미지가 회전된다 하더라도, 깊이방향(depthwise) 최대 풀링 층은 그와 상관없이 동일 출력을 만든다. (두께, 밝기, 왜곡, 색상 등 가능)



케라스에서는 깊이 방향 풀링 층을 제공하지 않지만, 텐서플로 저수준 딥러닝 API를 이용할 수 있다.

이는 `tf.nn.max_pool()` 함수를 사용하는데, 커널 크기와 스트라이드를 4개 원소를 가진 튜플로 지정한다. 이때, 첫 번째 세값은 1이어야 하는데, 이는 배치, 높이, 너비 차원을 따라 커널 크기와 스트라이드가 1이라는 뜻이다.

마지막 값은 깊이 차원을 따라 원하는 커널 크기와 스트라이드를 가리킨다. 이때, 입력 깊이를 나누었을 때 떨어지는 값이어야 한다. 이전 층에서 20개의 특성 맵이 출력된다면 3의 배수가 아니므로 작동하지 않는다.

```
output=tf.nn.max_pool(images.ksize=(1,1,1,3),padding="valid")
```

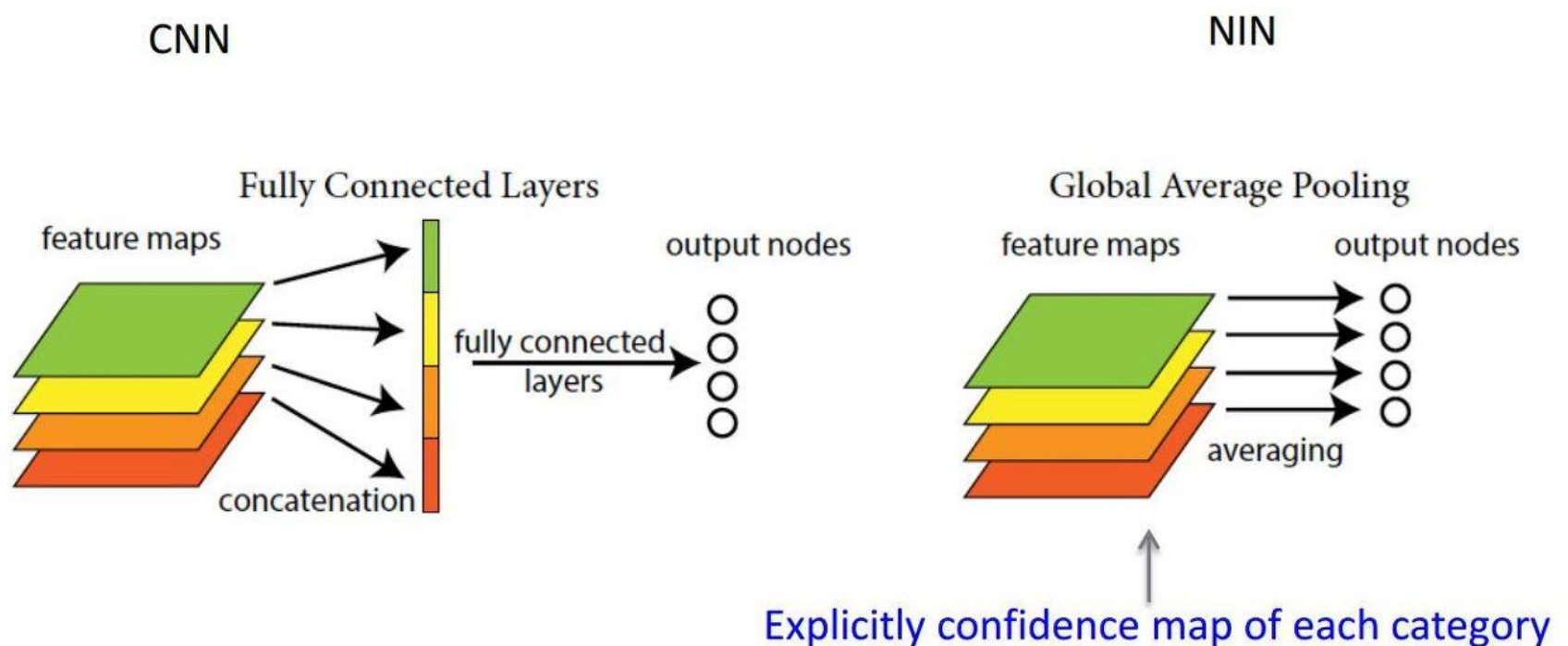
이를 케라스 모델의 층으로 만들고 싶으면 `Lambda` 층으로 감싸면 된다.

```
depth_pool = keras.layers.Lambda(lambda X:tf.nn.max_pool(X, ksize=(1,1,1,3), strides=(1,1,1,3), padding="valid"))
```

전역 평균 풀링 층(GAP, global average pooling layer)

전역 평균 풀링 층의 작동 방식은 각 특성 맵의 평균을 계산하는 것이다. 이는 입력과 공간 방향 차원이 같은 커널을 사용하는 풀링 층과 같다. 각 샘플의 특성 맵마다 하나의 숫자를 출력한다는 의미이다.

특성 맵의 대부분 정보를 잃지만 효과적으로 노드와 파라미터를 줄여 출력층에는 유용할 수 있다. 충분히 특이 맵의 채널 수가 많으면 적용해도 되지만, 채널 수가 적으면 Flatten이 유용하다. 명확한 채널 수 기준을 없지만 경험상 512차원 이상이 되면 GAP를 사용하면 좋다.



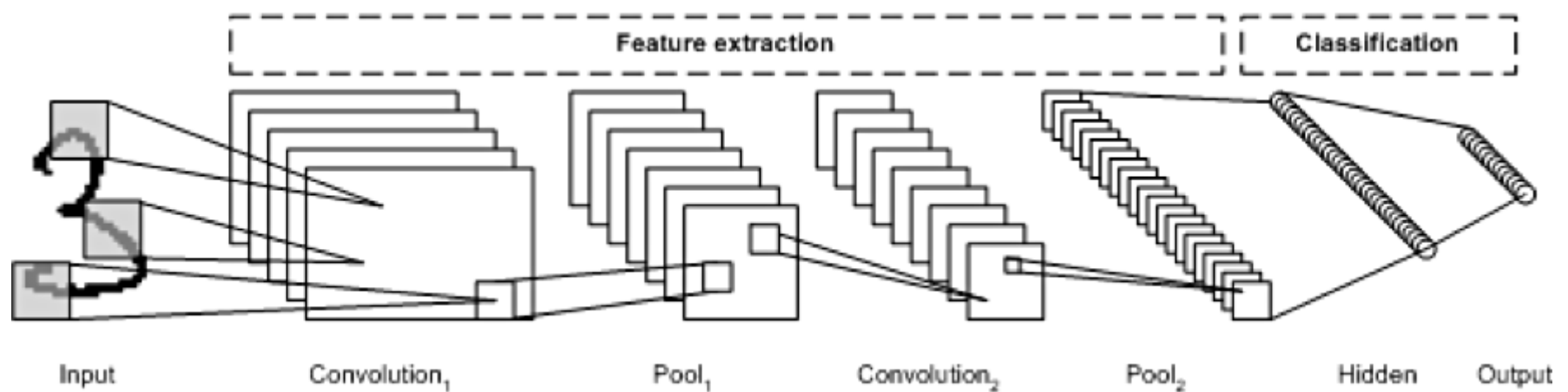
구현을 위해 `keras.layers.GlobalAvgPool2D` 를 사용한다.

```
global_avg_pool=keras.layers.GlobalAvgPool2D()
```

```
output_global_avg2 = keras.layers.Lambda(lambda X: tf.reduce_mean(X, axis=[1, 2]))
```

▼ 14.4 CNN 구조

전형적인 CNN구조는 합성곱 층을 몇 개 쌓고(각 층마다 ReLU 층을 쌓는다), 그 다음 풀링층을 쌓고, 그 다음에 또 합성곱 층을 몇 개 더 쌓고, 그 다음 다시 풀링 층을 쌓는 식이다.



CNN은 네트워크를 통과하여 진행할수록 이미지는 점점 작아지지만, 합성곱 층 때문에 일반적으로 점점 더 깊어지며 더 많은 특성 맵을 의미한다.

맨 위층에서는 완전 연결 층과 ReLU로 구성된 일반적인 FNN이 추가되고, 마지막 층에는 소프트 맥스 층 등을 통해 예측을 출력한다.

- tips

- 합성곱에 무 큰 커널을 사용하는 것 보다 작은 커널을 여러 겹 쌓는 것이 더 좋은 성능을 낸다. 보통 3X3 크기의 커널을 쌓지만 첫번째 합성곱 층은 예외로 5X5를 많이 사용한다.
- 풀링 층 다음에 필터 개수를 2배로 늘리는 것이 일반적인 방법이다. 밀집 네트워크는 샘플의 특성으로 1D 배열을 기대하므로 입력을 일괄로 펼쳐야 한다.

- 패션MNIST 데이터셋 문제를 해결하기 위한 간단한 CNN 코드

```
model = keras.models.Sequential([
    keras.layers.Conv2D(filters=64, kernel_size=7, activation='relu', padding='same',
                        input_shape=[28, 28, 1]),
    keras.layers.MaxPooling2D(2),
    keras.layers.Conv2D(filters=128, kernel_size=3, activation='relu', padding='same'),
    keras.layers.Conv2D(filters=128, kernel_size=3, activation='relu', padding='same'),
    keras.layers.MaxPooling2D(2),
    keras.layers.Conv2D(filters=256, kernel_size=3, activation='relu', padding='same'),
    keras.layers.Conv2D(filters=256, kernel_size=3, activation='relu', padding='same'),
    keras.layers.MaxPooling2D(2),
    keras.layers.Flatten(),
    keras.layers.Dense(128, activation='relu'),
    keras.layers.Dropout(0.5),
    keras.layers.Dense(64, activation='relu'),
    keras.layers.Dropout(0.5),
    keras.layers.Dense(10, activation='softmax')
])
```

→ 첫번째 층은 64개의 큰 필터(7X7)와 스트라이드 1을 사용한다. 이미지가 28X28 픽셀 크기이고, 하나의 컬러 채널이므로 `input_shape=[28, 28, 1]` 로 지정한다.

→ 풀링 크기가 2인 최대 풀링 층을 추가하여 차원을 절반으로 줄인다.

→ 이와 동일한 구조를 두 번 반복한다.

→ CNN이 출력층에 다다를수록 필터 개수가 늘어난다. 저수준 특성(ex. 작은 동심원, 수평성)의 개수는 적지만 이를 연결하여 고수준 특성을 만들 수 있는 방법이 많기 때문에 이런 구조가 합리적이다. 풀링 층 다음에 필터 개수를 두 배로 늘리는 것이 일반적인 방법, 풀링 층이 차원을 절반으로 줄이므로 이어지는 층에서는 파라미터 개수, 메모리 사용량, 계산 비용을 크게 늘리지 않고 특성 맵 개수만 두배로 늘릴 수 있다.

→ 다음은 두 개의 은닉층과 하나의 출력층으로 구성된 완전 네트워크이다. 이때 밀집 네트워크는 샘플의 특성으로 1D 배열을 필요로 하므로 일괄로 펼쳐야 한다. 또한 과대적합을 줄이기 위해 드롭아웃 층을 추가한다.

- CNN의 장점

- 1) 파라미터 공유(parameter sharing)

: CNN은 동일한 필터를 입력 특성 맵 각 부분에 적용하기 때문에 ‘공유’라고 표현한다.

- 2) 희소 연결(sparsity connection)

: 각 결과값은 오직 입력 특성 맵의 특정 부분에만 의존하기 때문에 ‘희소’라고 표현한다.

이렇게 되면 몇 픽셀 이동한 이미지도 유사한 속성을 갖게 되며(이동 불변성, translation invariance), 파라미터 수가 비교적 적기 때문에 오버피팅을 방지할 수 있게 된다.

ex. vertical egde detector의 역할을 하는 필터가 있다고 가정하자.

1	0	-1
1	0	-1
1	0	-1

해당 필터는 입력 이미지의 어느 부분에 적용해도 이미지의 수직 부분을 강조하는 역할을 수행한다. 또한 모든 뉴런을 연결해야 하는 일반 신경망(Fully Connected Layer)보다 파라미터 수가 적어 오버피팅의 위험이 줄어든다.

▼ 14.4.1 LeNet-5

LeNet-5 구조¹는 1998년 안 르쿤이 만들었으며 가장 널리 알려진 CNN 구조이다.

층	종류	특성 맵	크기	커널 크기	스트라이드	활성화 함수
입력	입력	1	32 x 32	-	-	-
C1	합성곱	6	28 x 28	5 x 5	1	tanh
S2	평균 풀링	6	14 x 14	2 x 2	2	tanh
C3	합성곱	16	10 x 10	5 x 5	1	tanh
S4	평균 풀링	16	5 x 5	2 x 2	2	tanh
C5	합성곱	120	1 x 1	5 x 5	1	tanh
F6	완전 연결	-	84	-	-	tanh
출력	완전 연결	-	10	-	-	RBF

- MNIST 이미지는 28 x 28 픽셀이지만 제로 패딩되어 32 x 32가 되고 네트워크에 주입되기 전 정규화 진행(네트워크의 나머지 부분은 패딩을 사용하지 않음)
- 평균 풀링 층에선 각 뉴런이 입력의 평균을 계산한 뒤 그 값에 학습되는 계숫값(특성 맵마다 하나씩 존재)을 곱함 -> 학습되는 값인 편향을 더함 -> 마지막으로 활성화 함수 적용
- C3에 있는 대부분의 뉴런은 S2의 (6개 맵 전체가 아닌) 3개 또는 4개 맵에 있는 뉴런에만 연결
- 출력층에선 입력과 가중치 벡터를 행렬 곱셈하는 대신 각 뉴런에서 입력 벡터와 가중치 벡터 사이의 유클리드 거리를 출력 -> 각 출력은 이미지가 얼마나 특정 숫자 클래스에 속하는지 측정 (요즘엔 잘못된 예측을 많이 줄여주고 그레이디언트 값이 크고 빠르게 수렴되기 때문에 크로스 엔트로피 비용 함수 선호)

▼ 14.4.2 AlexNet

이 구조는 더 크고 싶을 뿐 LeNet-5와 비슷하며, 처음으로 합성곱 층 위에 풀링 층을 쌓지 않고 바로 합성곱 층끼리 쌓았다.

층	종류	특성 맵	크기	커널 크기	스트라이드	패딩	활성화 함수
입력	입력	3(RGB)	227 x 227	-	-	-	-
C1	합성곱	96	55 x 55	11 x 11	4	valid	ReLU
S2	최대 풀링	96	27 x 27	3 x 3	2	valid	-
C3	합성곱	256	27 x 27	5 x 5	1	same	ReLU
S4	최대 풀링	256	13 x 13	3 x 3	2	valid	-
C5	합성곱	384	13 x 13	3 x 3	1	same	ReLU
C6	합성곱	384	13 x 13	3 x 3	1	same	ReLU
C7	합성곱	256	13 x 13	3 x 3	1	same	ReLU
F8	최대 풀링	256	6 x 6	3 x 3	2	valid	-
F9	완전 연결	-	4,096	-	-	-	ReLU
F10	완전 연결	-	4,096	-	-	-	ReLU
출력	완전 연결	-	1,000	-	-	-	Softmax

이 구조는 과대적합을 줄이기 위해 2가지 규제 기법을 사용한다.

1. 훈련하는 동안 F9와 F10의 출력에 드롭아웃을 50% 비율로 적용한다.
2. 훈련 이미지를 랜덤으로 여러 간격으로 이동하거나 수평으로 뒤집고 조명을 바꾸는 식으로 데이터 증식(data augmentation)을 수행한다.

• 데이터 증식

데이터 증식은 진짜 같은 훈련 샘플을 인공적으로 생성하여 훈련 세트의 크기를 늘리는 것이다. 오버피팅을 줄이므로 규제 기법으로 사용된다.

원본 이미지에 다양한 변형(크기 변경, 이동, 회전 등)을 가하면 모델이 그림에 있는 물체의 위치, 방향, 크기 변화에 덜 민감해진다. 주의할 점은 원본 학습 이미지의 개수를 늘리는 것이 아니라 학습시마다 개별 원본 이미지를 변형해서 학습을 수행한다는 것이다.

데이터 증식 유형

1) 공간(Spatial) 레벨 변형

Flip: Vertical(up/down), Horizontal(left/right)

Crop: Center, Random

Affine: Rotate, Translate, Shear, Scale(Zoom)

2) 픽셀(Pixel) 레벨 변형

Bright, Saturation, Hue, GrayScale, ColorJitter

Contrast

Blur, Gaussian Blur, Median Blur

Noise, Cutout

Histogram

Gamma

RGBShift

Sharpen

케라스에서는 `ImageDataGenerator` 라는 패키지를 제공한다. 비교적 쉽게 데이터 증식을 수행하고 케라스와 통합되어 있어 편리한 데이터 전처리 파이프라인을 제공한다는 장점이 있다.

대표적인 `ImageDataGenerator` 변환 유형은 다음과 같다. `Rotation` 이나 `Shift`, `Zoom`의 경우 엣지에 빈 공간이 생기기도 하는데 이때 `fill_mode` 라는 파라미터를 통해 빈 공간을 채운다.

Filp

좌우 반전 : `horizontal_flip = True`

상하 반전 : `Vertical_flip = True`

Rotation

`rotation_range = 45`

임의의 -45 ~ +45 사이 회전

Shift

좌우 이동 : `width_shift_range = 0.2`

상하 이동 : `height_shift_range = 0.2`

0~1 사이값 설정하여 좌우, 상하를 이동

Zoom

`zoom_range = [0.5, 1.5]`

1보다 작은 값은 확장, 큰 값은 축소

shear

`shear_range = 45`

x축 또는 y축을 중심으로 0~45도 사이 변환

Bright

`brightness_range = (0.1, 0.9)`

`brightness_range`로 밝기 조절: 0에 가까울수록 어둡고, 1에 가까울수록 밝음

Channel Shift

`channel_shift_range = 120`

R, G, B 픽셀값을 -120 ~ +120 사이의 임의값을 더하여 변환

Normalization

`featurewise_center = True`: 각 R, G, B 픽셀값에서 개별 채널별 평균 픽셀값을 빼서 평균이 0이 되도록 유지

`featurewise_std_normalization = True`: 각 R, G, B 픽셀값에서 개별 채널별 표준편차 픽셀값을 나눔

`.rescale = 1/255.0`: 딥러닝 입력은 비교적 작은 값은 선호하므로 픽셀값을 0 ~ 1 사이값으로 변환

`fill_mode`

1. nearest: 빈 공간에 가장 근접한 픽셀로 채움
2. reflect: 빈 공간만큼의 영역을 근처 공간으로 채우되 마치 거울로 반사되는 이미지를 보듯이 채움
3. wrap: 빈 공간을 이동으로 잘려나간 이미지로 채움
4. constant: 특정 픽셀값으로 채움(cval이란 파라미터 채우며 cval=0일 때 검은색 픽셀)

AlexNet 구조는 , C1과 C3 층의 ReLU 단계 후에 바로 **LRN**(local response normalization)이라 부르는 경쟁적인 **정규화 단계**를 사용했다. **가장 강하게 활성화된 뉴런이 다른 특성 맵에 있는 같은 위치의 뉴런을 억제하는 구조**이다. 이는 특성 맵을 각기 특별하게 다른 것과 구분되게 하고, 더 넓은 시각에서 특징을 탐색하도록 만들어 결국 일반화 성능을 향상시킨다.

$$b_i = a_i(k + \alpha \sum_{j=j_{low}}^{j_{high}} a_j^2)^{-\beta} \text{ where}$$

$$j_{high} = \min(i + \frac{r}{2}, f_n - 1)$$

$$j_{low} = \max(0, i - \frac{r}{2})$$

b_i : i 특성 맵, u 행, v 열에 위치한 뉴런의 정규화된 출력(이 식에선 현재 행과 열에 위치한 뉴런만 고려하므로 u 와 v 는 없음)
 a_i : ReLU 단계를 지나고 정규화 단계는 거치기 전인 뉴런의 활성화 값
 k, α, β, r : 하이퍼파라미터로 k 는 편향, r 은 깊이 반경(depth radius)
 f_n : 특성 맵 수

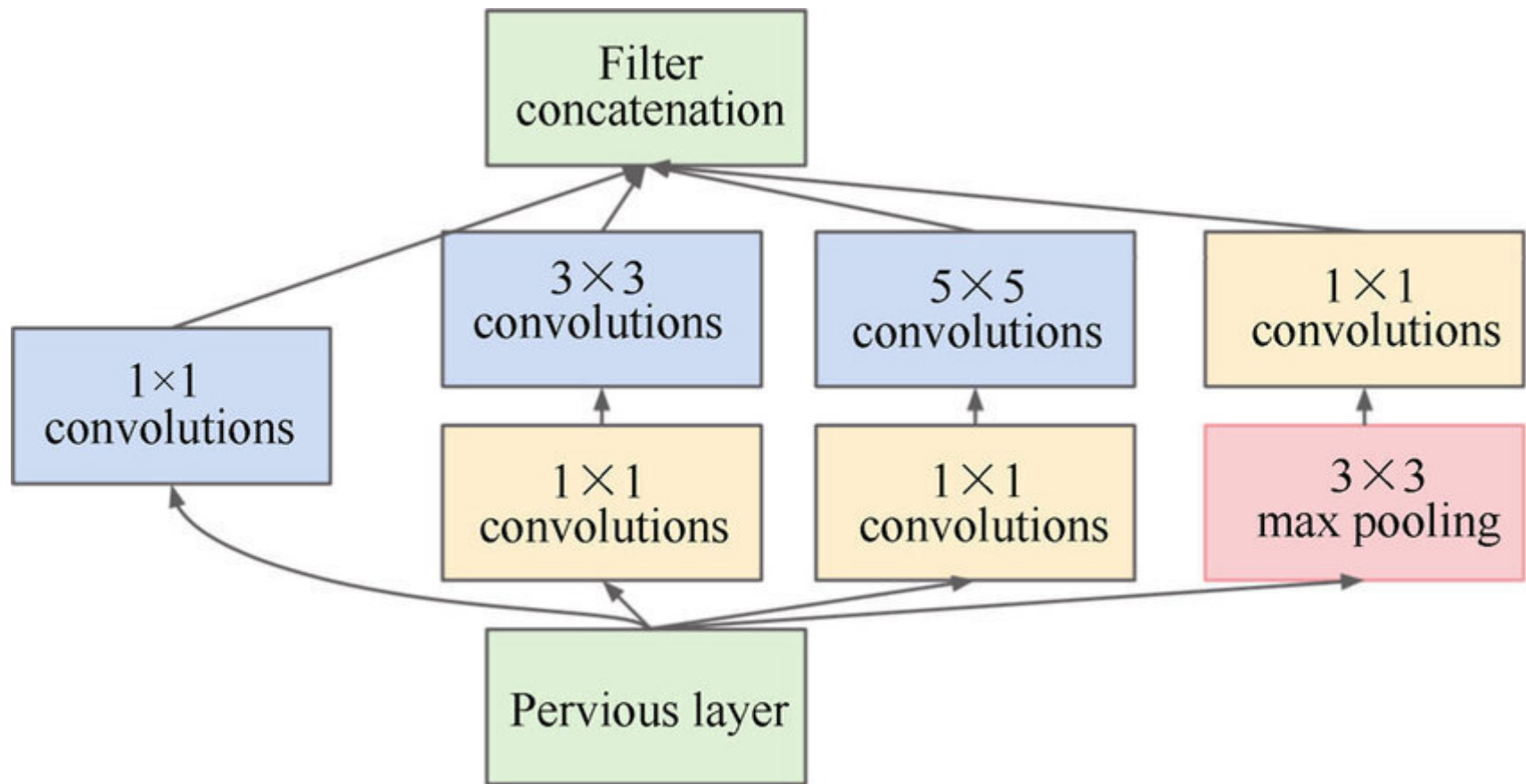
예를 들어, $r = 2$ 이고 한 뉴런이 강하게 활성화 되었다면 자신의 위와 아래의 특성 맵에 위치한 뉴런의 활성화를 억제한다. AlexNet에서 하이퍼파라미터는 $r = 2, \alpha = 0.00002, \beta = 0.75, k = 1$ 로 설정되어있다.

▼ 14.4.3 GoogLeNet

GoogLeNet 구조는 이전 CNN보다 훨씬 더 깊은 네트워크 구조를 가진다. 또한 인셉션 모듈(inception module)이라는 서브 네트워크를 가지고 있어 GoogLeNet이 이전 구조보다 훨씬 효과적으로 파라미터를 사용한다.

- 인셉션 모듈의 구조는 다음과 같다.

모든 층마다 스트라이드 1 및 same 패딩을 사용하고 있다.



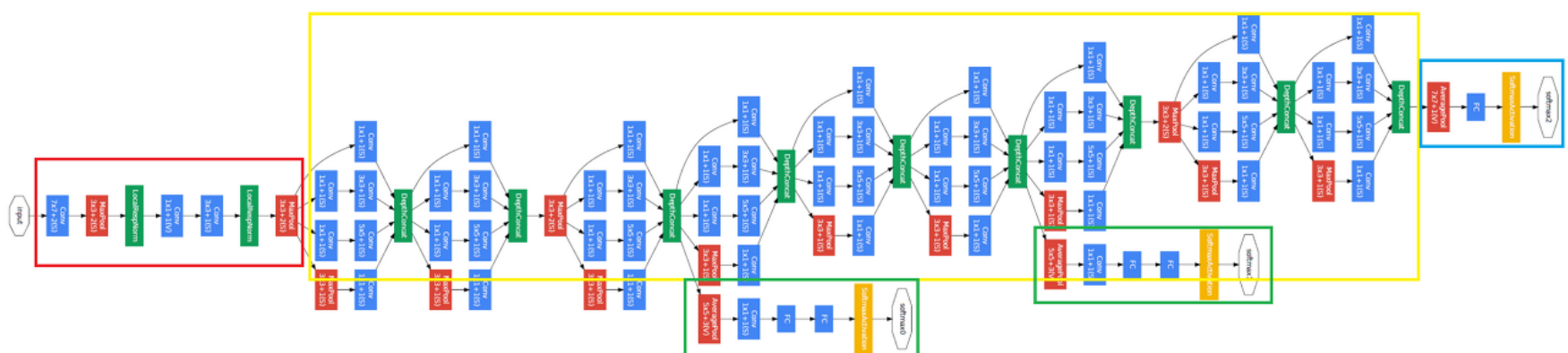
처음에 입력한 신호가 복사되어 4개의 다른 층에 주입된다. 모든 합성곱 층은 ReLU 활성화 함수를 사용하고, 두 번째 합성곱 층은 각기 다른 커널 크기(1X1, 3X3, 5X5)를 사용해 다른 크기의 패턴을 잡는다. 이때 모든 층은 스트라이드 1과 same 패딩을 사용하므로 모든 층마다 출력의 높이와 너비가 모두 입력과 같기 때문에 모든 출력을 깊이 연결 층(depth concatenation layer)에서 깊이 방향으로 연결할 수 있다. 텐서플로로 구현하기 위해서는 `axis=3` (3은 깊이 방향 축) 매개변수로 `tf.concat()` 연산을 사용하면 된다.

- 인셉션 모듈이 1X1 커널의 합성곱 층을 가지는 목적은 다음과 같다.

1. 공간상의 패턴을 잡을 수는 없지만 깊이 차원을 따라 높은 패턴을 잡을 수 있다.
2. 입력보다 더 작은 특성 맵을 출력하므로(예를 들어, 필터가 1개라면 기존 채널 값들이 하나의 수로 출력된다는 의미) 차원을 줄인다는 의미인 병목 층(bottleneck layer) 역할 담당 → 연산 비용과 파라미터 개수를 줄여 훈련 속도를 높이고 일반화 성능 향상
3. 합성곱 층의 쌍([1X1, 5X5])이 더 복잡한 패턴을 감지할 수 있는 한 개의 강력한 합성곱 층처럼 작동

- GoogLeNet의 CNN 구조를 살펴보면 다음과 같다.

네트워크를 하나로 길게 쌓은 구조이고 9개의 인셉션 모듈을 포함하고 있다. 모든 합성곱 층은 ReLU 활성화 함수를 사용한다.



빨간색 사각형 : pooling

초록색 사각형 : LRN

파란색 사각형 : 합성곱

주황색 사각형 : 소프트맥스

- 네트워크 로직은 다음과 같다.

- 처음 두 층은 계산의 양을 줄이기 위해 이미지의 높이와 너비를 4배로 줄인다(same 패딩에 스트라이드가 2이므로). 많은 정보를 유지하기 위해 첫 번째 층은 큰 크기의 커널(7X7)을 사용한다.
- LRN 층은 이전 층이 다양한 특성을 학습하도록 한다.
- 이어지는 두 개의 합성곱 층 중에서 첫 번째 층이 앞서 설명했듯 병목 층처럼 작동한다. 이 합성곱 쌍을 하나의 똑똑한 합성곱 층으로 생각할 수 있다.
- 다시 한 번 LRN 층이 이전 층으로 하여금 다양한 패턴을 학습하도록 한다.
- 9개의 인셉션 모듈이 이어지고 차원 감소와 속도 향상을 위해 몇 개의 최대 풀링 층을 끼워 넣는다.
- 전역 평균 풀링 층이 각 특성 맵의 평균을 출력한다. 여기서 공간 방향 정보를 모두 잃지만 남아 있는 공간 정보가 많지 않아 괜찮다. 이 층에서 차원 축소로 인해 (AlexNet 처럼) CNN 위에 몇 개의 완전 연결 층을 둘 필요가 없다.
- 규제를 위한 드롭아웃 층 다음에 1000개의 유닛과 소프트 맥스 활성화 함수를 적용한 완전 연결 층으로 클래스 확률 추정 값을 출력한다.

위 도식에서 3번째 및 6번째 인셉션 모듈 위에 연결된 2개의 부가적인 분류기를 볼 수 있다. 이들은 모두 평균 풀링 - 합성곱 - FC - FC - 소프트 맥스로 구성되어 있다. 훈련하는 동안 여기에서의 손실이 (70% 정도 감해서) 전체 손실에 더해지며, 이는 그레디언트 소실 문제를 줄이고 네트워크를 규제하기 위함이다. 하지만 효과는 비교적 적은 것으로 알려져 있다.

▼ 14.4.4 VGGNet

VGGNet은 단순하고 고전적인 구조로, 2개 또는 3개의 합성곱 층 뒤에 풀링 층이 나오고 다시 2개 또는 3개의 합성곱 층과 풀링 층이 등장하는 식이다. 마지막 밀집 네트워크는 2개의 은닉층과 출력층으로 이루어진다. VGGNet은 많은 개수의 필터를 사용하지만 크기는 3X3 필터만 사용한다.

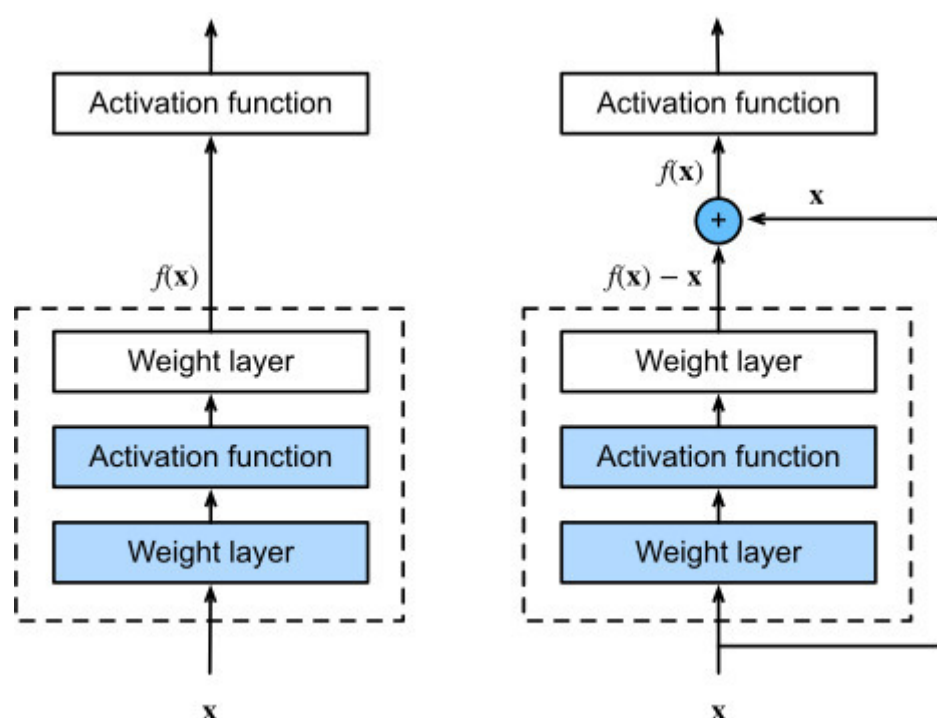
▼ 14.4.5 ResNet

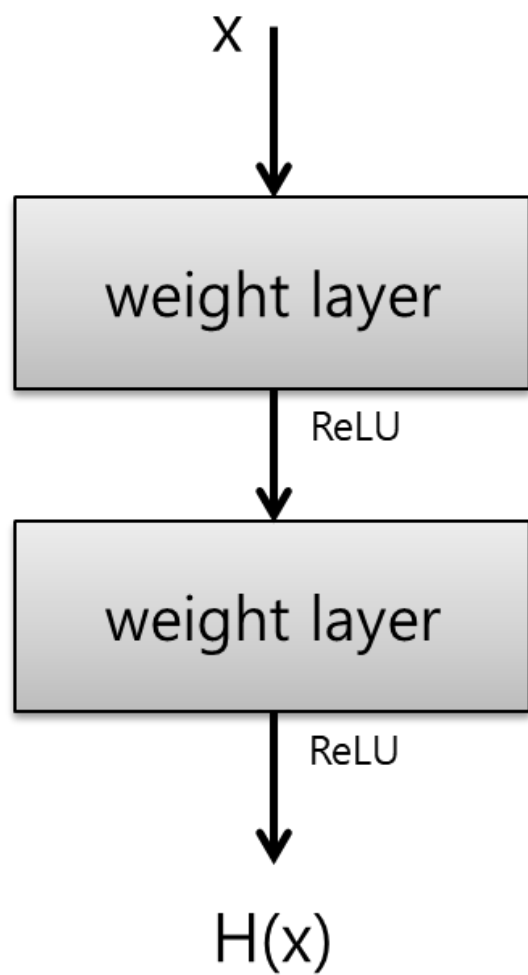
잔차 네트워크(residual network, ResNet)는 파라미터는 적어지고 네트워크는 더 깊어지는 일반적인 트렌드를 만들었다.

이런 깊은 네트워크를 훈련시킬 수 있는 핵심 요소는 스킵 연결(skip connection) 또는 숏컷 연결(shortcut connection)이다. 즉, 어떤 층에 주입되는 신호가 상위 층의 출력에도 더해지는 것이다.

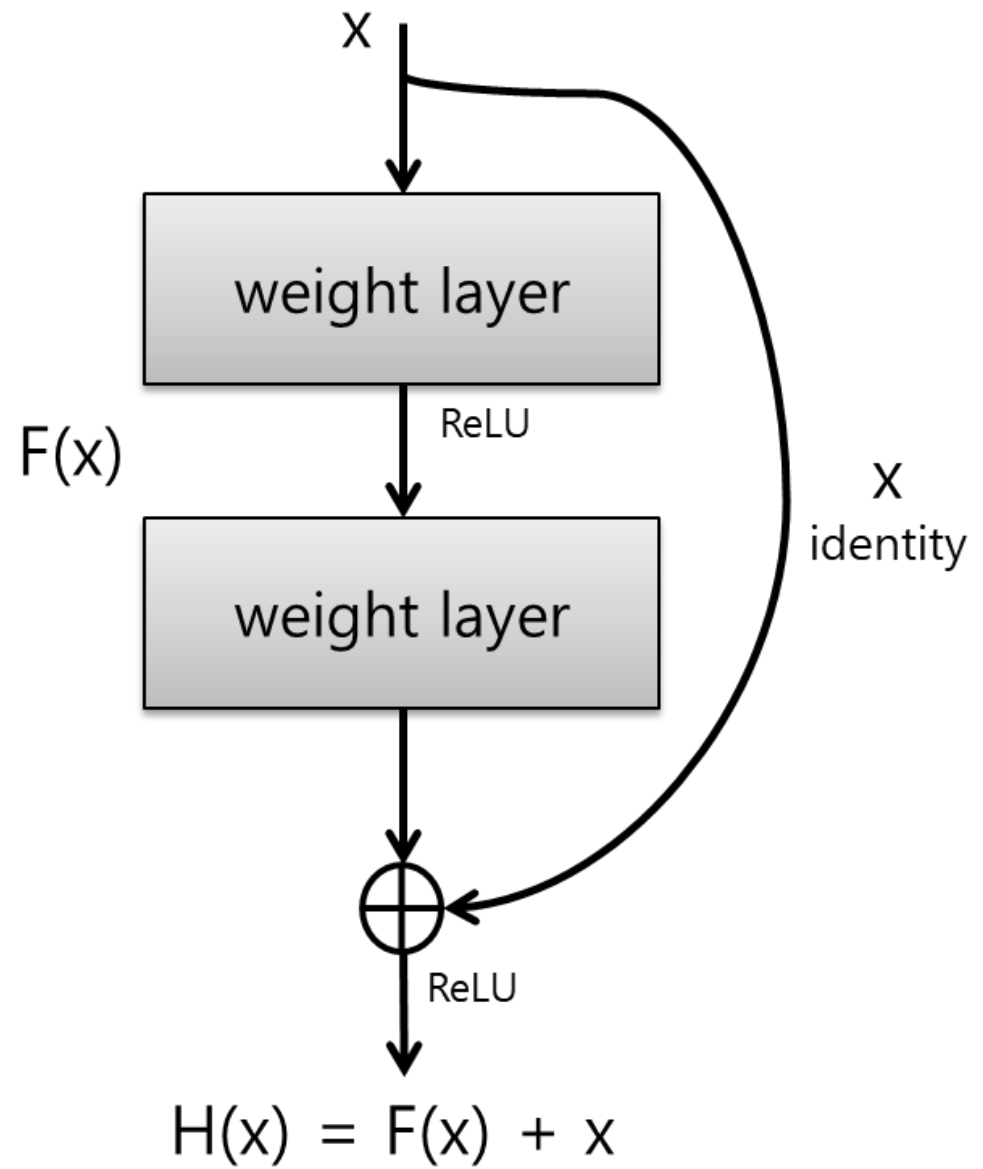
잔차 학습(residual learning)

신경망을 훈련시킬 때는 목적 함수 $f(x)$ 를 모델링 하는 것이 목표다. 만약 입력 x 를 네트워크 출력에 더한다면 네트워크는 $f(x) - x$ 를 학습하게 되며 이를 잔차학습이라고 한다.





기존 방식

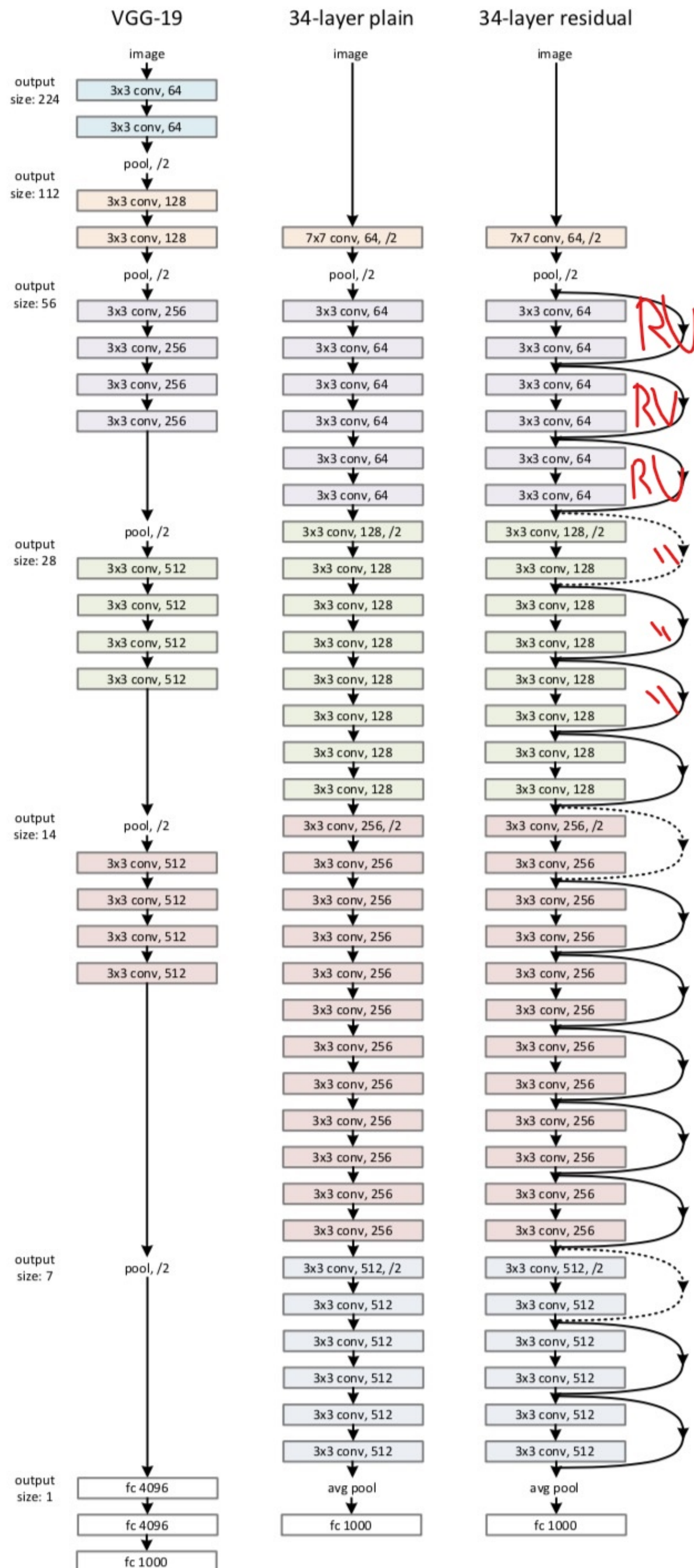


Residual block

일반적인 신경망을 초기화 할 때는 가중치가 0에 가깝기 때문에 네트워크도 0에 가까운 값을 출력한다. 스킵 연결을 추가하면 이 네트워크는 입력과 같은 값을 출력한다. 즉, 초기에는 항등 함수(identity function)를 모델링한다. 목적 함수가 항등 함수에 매우 가까우다면 훈련 속도가 매우 빨라질 것이다.

또한 스킵 연결을 많이 추가하면 일부 층이 학습되지 않았더라도 네트워크는 훈련을 시작할 수 있다. 즉, 스킵 연결 덕분에 입력 신호가 잔차 네트워크에 손쉽게 영향을 미치게 된다.

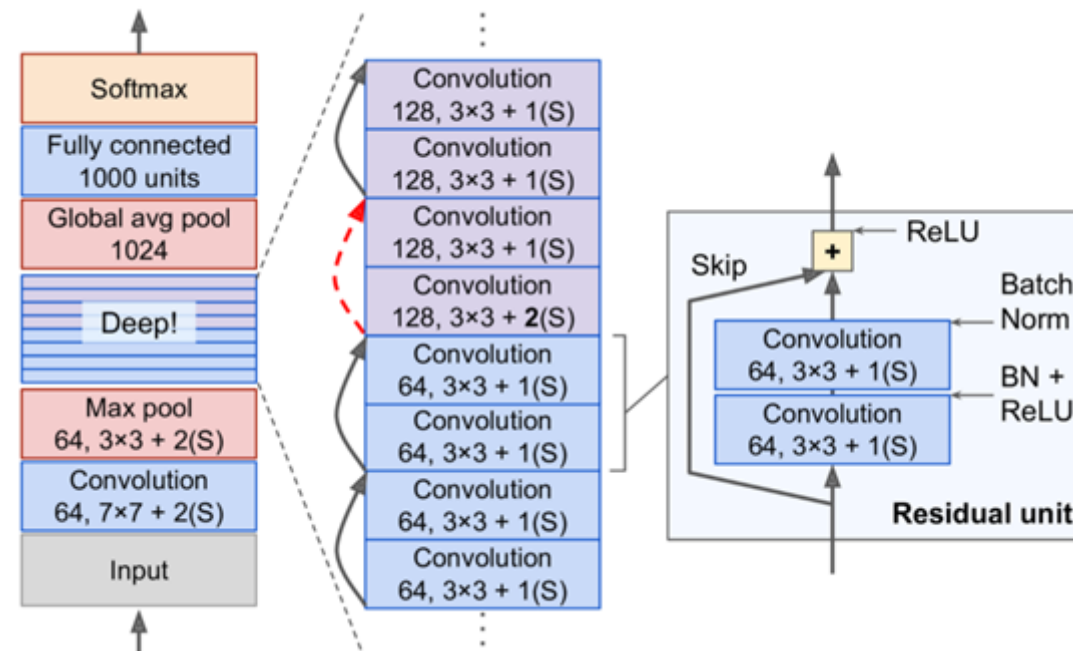
심층 잔차 네트워크는 스킵 연결을 가진 작은 신경망인 잔차 유닛(Residual unit, RU)을 쌓을 것으로 볼 수 있다.



- 구조는 다음과 같다.

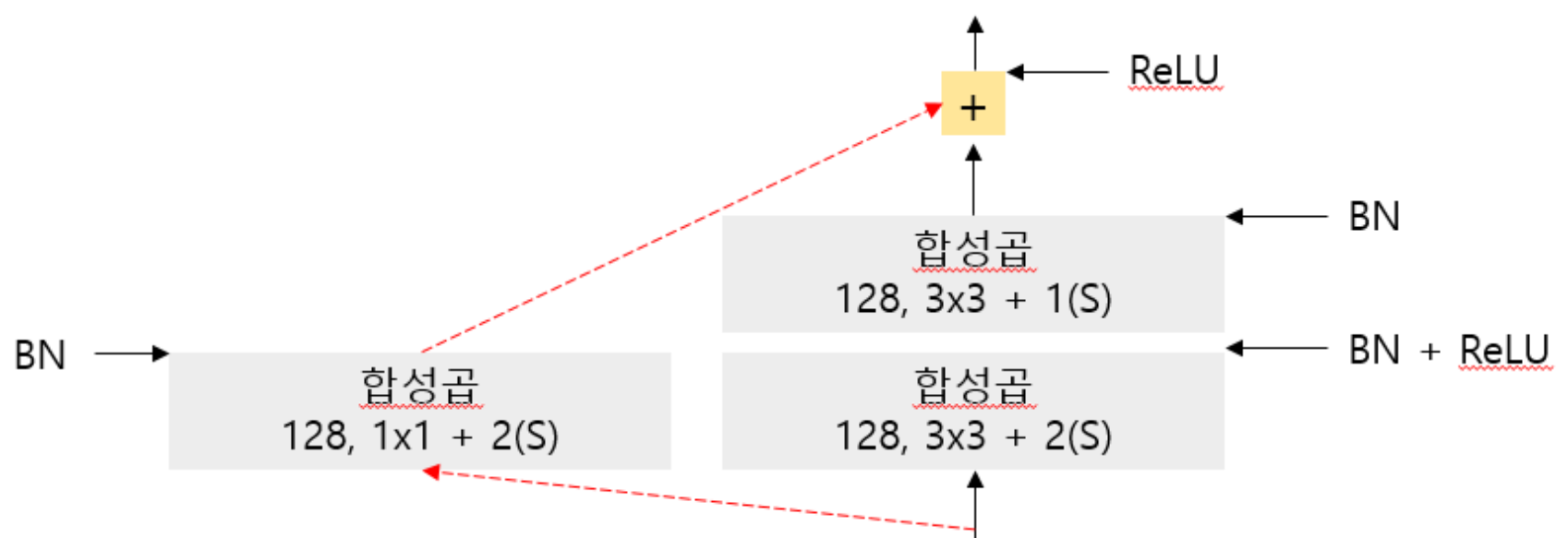
드롭 아웃 층을 제외하면, GoogLeNet과 똑같이 시작하고 종료한다. 그러나 중간에 단순한 잔차 유닛을 매우 깊게 쌓은 것이다.

ResNet의 각 잔차 유닛은 배치정규화(BN)와 ReLU, 3x3 커널을 사용하고 공간 정보를 유지하는 (스트라이드 1, same 패딩) 2개의 합성곱 층으로 이루어져 있다.



특성 맵의 수는 몇 개의 잔차 유닛마다 2배로 늘어나고 높이와 너비는 절반이 된다. (스트라이드 2인 합성곱 층을 통해). 이때 입력과 출력의 크기가 다르기 때문에 입력이 잔차 유닛의 출력에 바로 더해질 수 없다. (빨간색 파선)

이 문제를 해결하기 위해 스트라이드 2이고 출력 특성 맵이 수가 같은 1x1 합성곱 층으로 입력을 통과시킨다.



RestNet-34: 64개 특성 맵을 출력하는 3개 RU / 128개 맵의 4개 RU / 256개 맵의 6개 RU / 512개 맵의 3개 RU
 RestNet-152: 256개 맵을 출력하는 3개 RU / 512개 맵의 8개 RU / 1024개 맵의 36개 RU / 2048개 맵의 3개 RU

▼ 14.4.6 Xception

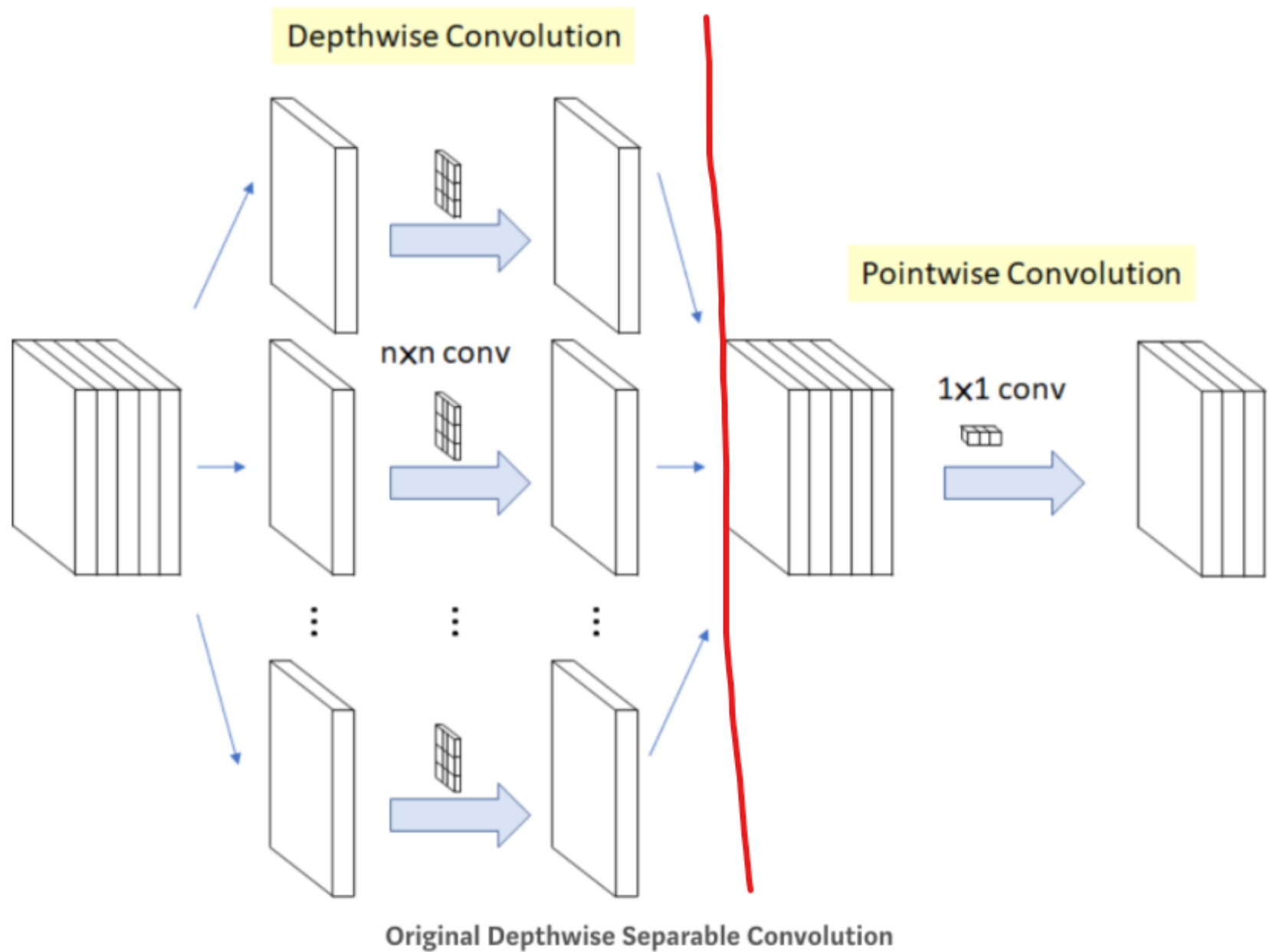
Xception

GoogLeNet 구조의 또 다른 변종이다. GoogLeNet과 ResNet의 아이디어를 합쳤지만 인셉션 모듈은 깊이별 분리 합성곱 층 (depthwise separable convolution layer) 이라는 특별한 층으로 대체했다.

즉, Xception은 수정된 깊이별 분리 합성곱 층(modified depthwise seperable convolution layer)을 사용했다.

일반적인 합성곱 층은 공간상의 패턴(타원 형태)와 패널 사이의 패턴(입+코+눈=얼굴)을 동시에 잡기 위해 필터를 사용한다. 분리 합성곱 층은 공간 패턴과 채널 사이 패턴을 분리하여 모델링 할 수 있다고 가정한다.

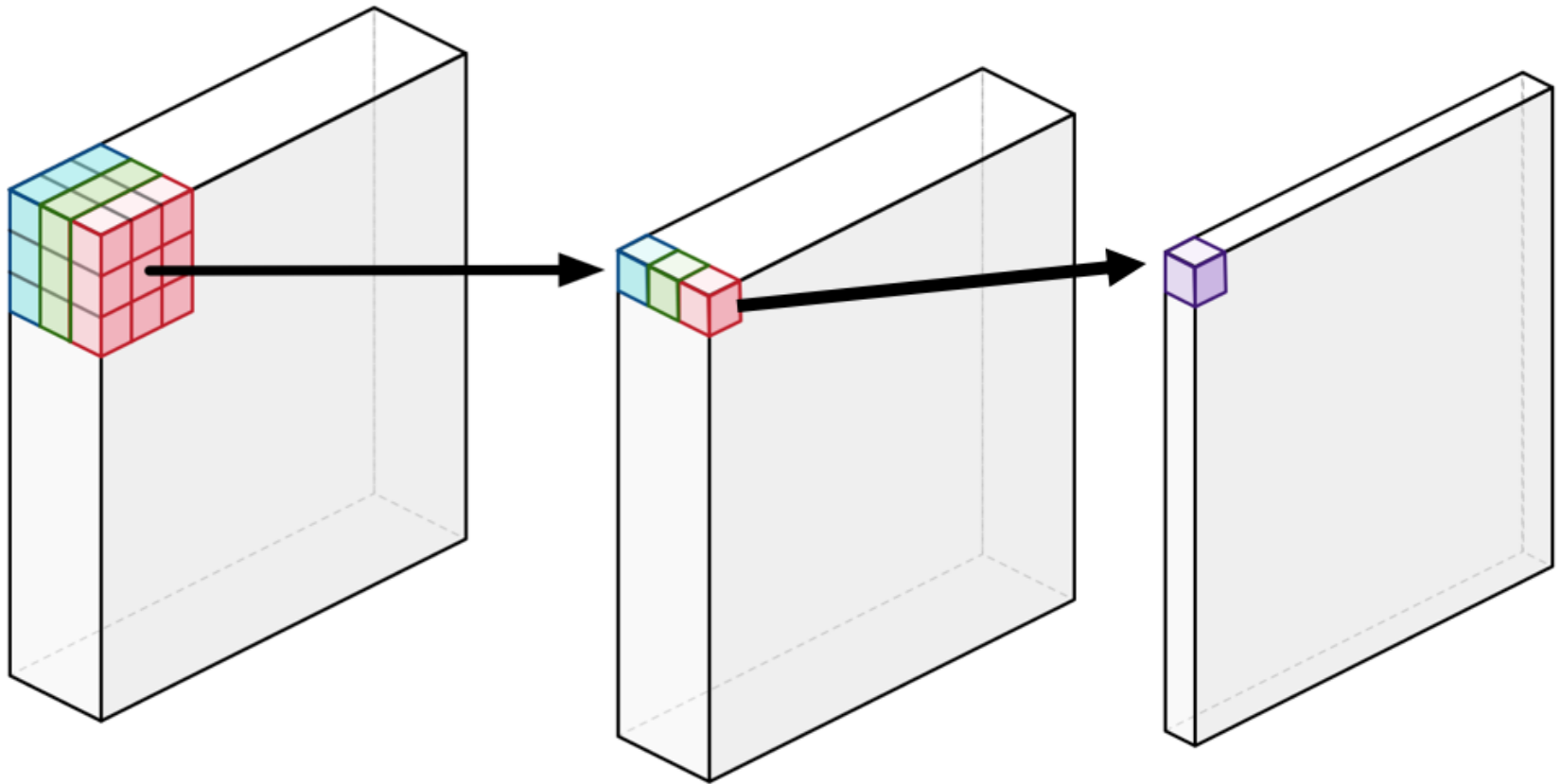
오리지널 깊이별 분리 합성곱 층을 살펴보면 다음과 같다.



해당 층은 위처럼 2개의 부분으로 구성된다.

1) Depthwise Convolution : 하나의 공간 필터를 각 입력 특성에 적용한다.

2) Pointwise Convolution : 깊이 필터 (1X1)를 사용해 채널 사이 패턴만 조사한다. 분리 합성곱층은 입력 채널마다 하나의 공간 필터만 가지기 때문에 입력층과 같이 채널이 너무 적은 층 다음에 사용하는 것을 피해야 한다. 이런 이유로 Xception 구조는 2개의 일반 합성곱 층으로 시작하며(Featuer 2), 나머지는 분리 합성곱만 사용한다(총 34개)



ex. $256 \times 256 \times 3$ 의 input 이미지가 있을 때,

- 1) $256 \times 256 \times 1$ 을 3번 진행해서 concat 진행
- 2) pointwise convolution을 이용해 채널의 갯수를 1로 줄인다. (단순히 weighted sum 계산)

이 과정으로 convolution을 하면 약 9배 정도 빠르다고 한다.

인셉션 모듈이 전혀 없는데 Xception을 GoogLeNet의 변종으로 간주하는 이유가 있다. 위에서 언급한 것처럼 인셉션 모듈은 1×1 필터를 사용한 합성곱 층을 포함한다. 이 층은 채널 사이의 패턴만 감지한다. 하지만 위에 놓인 합성곱 층은 공간과 채널 패턴을 모두 감지하는 일반적인 합성곱 층이다. 따라서 인셉션 모듈을(공간 패턴과 채널 패턴을 함께 고려하는) 일반 합성곱 층과 (따로 고려하는) 분리 합성곱 층의 중간 형태로 생각할 수 있다.

분리 합성곱 층은 일반 합성곱 층보다 파라미터, 연산, 메모리를 더 적게 사용하고 일반적으로 성능은 더 높다. 그러나 분리 합성곱 층은 입력 채널마다 하나의 공간 필터만 가지기 때문에 입력층과 같이 채널이 너무 적은 층 다음에 사용하는 것은 피해야 한다.

▼ 14.4.7 SENet

| SENet

SENet은 인셉션 네트워크와 ResNet 같은 기존 구조를 확장하여 성능을 높였다. 인셉션 네트워크와 ResNet을 확장한 버전을 각각 SE-Inception과 SE-ResNet이라고 한다. SENet은 원래 구조에 있는 모든 유닛(모든 인셉션 모듈 및 잔차 유닛)에 SE 블록이라는 작은 신경망을 추가한 것이다.

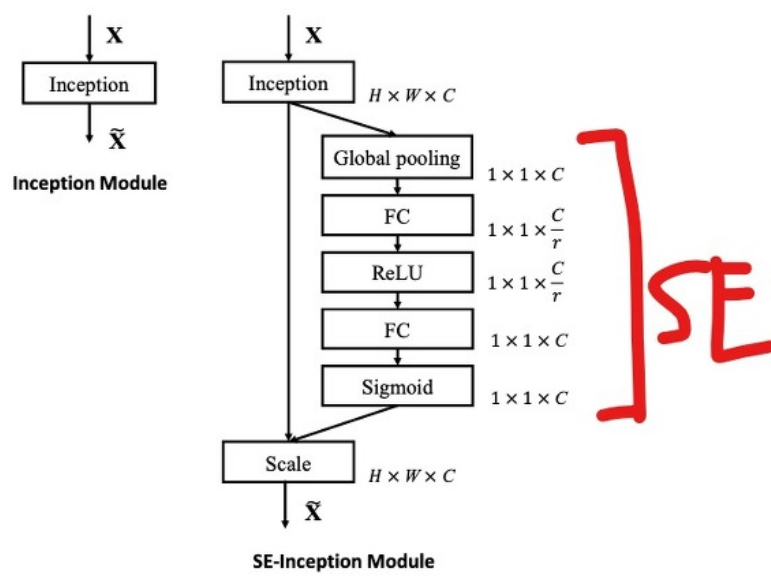


Fig. 2. The schema of the original Inception module (left) and the SE-Inception module (right).

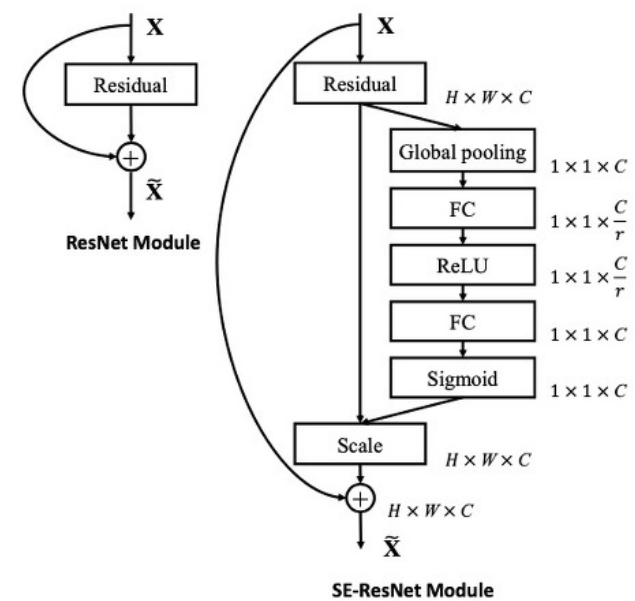


Fig. 3. The schema of the original ResNet module (left) and the SE-ResNet module (right).

SE 블록이 추가된 부분의 유닛의 출력을 깊이 차원에 초점을 맞추어 분석한다. (공간 패턴은 신경 X). 어떤 특성이 일반적으로 동시에 가장 크게 활성화되는지 학습한다. 그 다음 이 정보를 사용하여 특성 맵을 보정하게 된다.

ex. SE 블록이 그림에서 함께 등장하는 입,코,눈을 학습할 수 있고, 우리가 사진에서 입과 코를 보았다면 눈도 볼 수 있다고 기대한다. 따라서 입과 코 특성 맵이 강하게 활성화되고 눈 특성 맵만 크게 활성화되지 않았다면 이 블록이 눈 특성 맵의 출력을 높인다.(관련 없는 특성 맵의 값을 줄인다.)

하나의 SE블록은 3개 층으로 구성된다.

전역 풀링 층, ReLU 활성화 함수를 사용하는 밀집 은닉층, 시그모이드 활성화 함수를 사용하는 밀집 출력 층이다.

처음에 전역 풀링 층이 각 특성 맵에 대한 평균 활성화 값을 계산한다. 256개 특성 맵을 가진 입력이라면 각 필터의 전반적인 응답 수준을 나타내는 256개의 숫자가 출력된다.

다음 층은 256개보다 훨씬 적은 뉴런(일반적으로 특성 맵 개수보다 16배 적음)을 가지며 압축된다. 이 저차원 벡터(=하나의 임베딩)는 특성 응답의 분포를 표현한다. 해당 병목 층을 통해 SE블록이 특성 조합에 대한 일반적인 표현을 학습하게 된다.

마지막으로 출력층은 이 임베딩을 받아 특성 맵마다 0과 1 사이의 하나의 숫자를 담은 보정된 벡터를 출력한다. 그 다음 특성 맵과 이 보정된 벡터를 곱해 관련 없는 특성값을 낮추고 관련 있는 특성값은 그대로 유지한다.

▼ 14.5 케라스를 사용해 ResNet-34 CNN 구현하기

ResNet-34 모델

ResNet-34 모델 구현을 위해서는 먼저 ResidualUnit층을 만들어야 한다.

```
DefaultConv2D = partial(keras.layers.Conv2D, kernel_size=3, strides=1,
                        padding="SAME", use_bias=False)

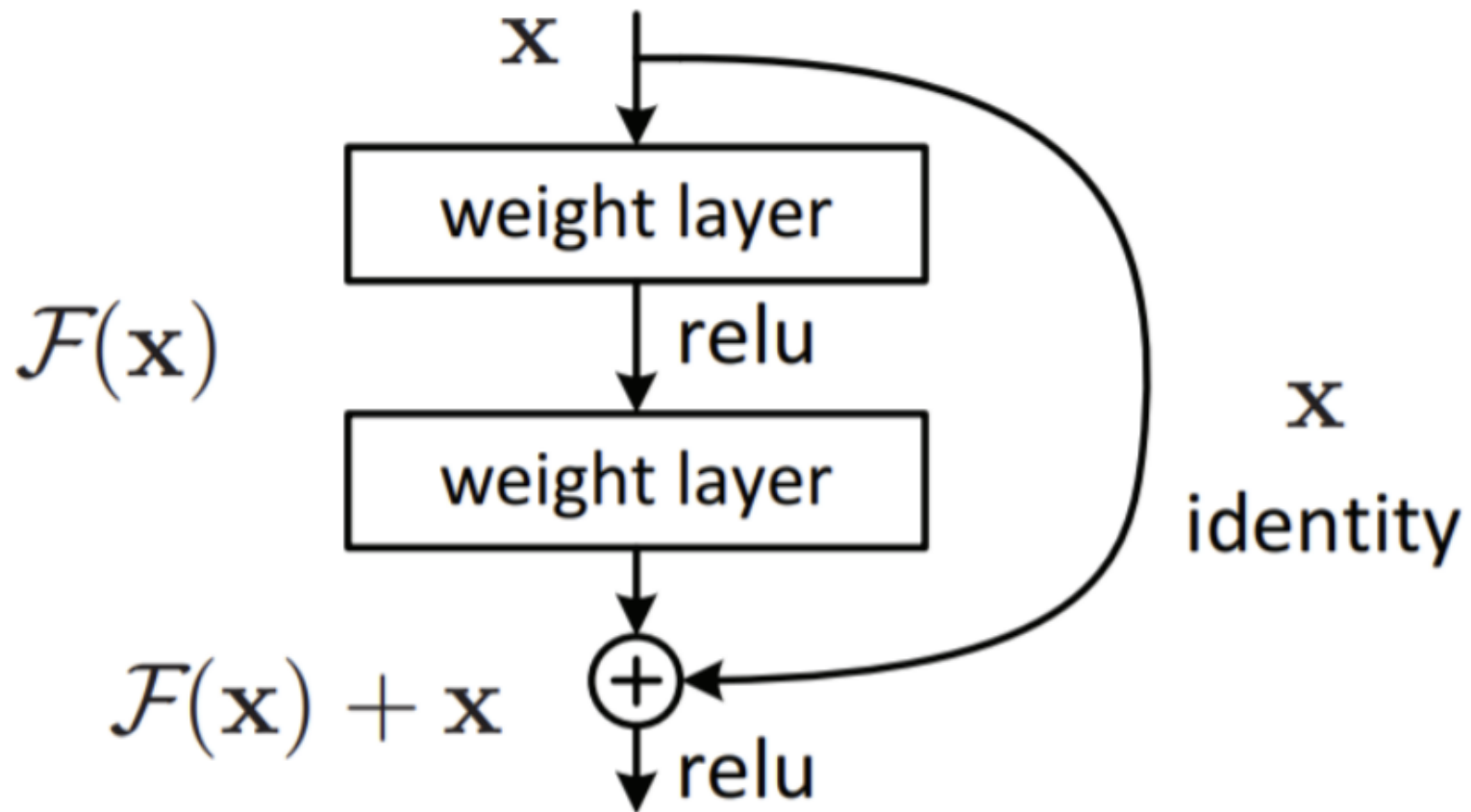
class ResidualUnit(keras.layers.Layer):
    def __init__(self, filters, strides=1, activation="relu", **kwargs):
        super().__init__(**kwargs)
        self.activation = keras.activations.get(activation)
        self.main_layers = [
            DefaultConv2D(filters, strides=strides),
            keras.layers.BatchNormalization(),
            self.activation,
            DefaultConv2D(filters),
            keras.layers.BatchNormalization()]
        self.skip_layers = []
```

```

    if strides > 1:
        self.skip_layers = [
            DefaultConv2D(filters, kernel_size=1, strides=strides),
            keras.layers.BatchNormalization()]

    def call(self, inputs):
        Z = inputs
        for layer in self.main_layers:
            Z = layer(Z)
        skip_Z = inputs
        for layer in self.skip_layers:
            skip_Z = layer(skip_Z)
        return self.activation(Z + skip_Z)

```



먼저 생성자에서 필요한 층을 만든다.

main_layers가 이 그림의 왼쪽 모듈이며, Skip_layers는 오른쪽 모듈이다. call() 메서드에서 입력은 main_layers와 skip_layers에 통과시킨 후 두 출력을 더하여 활성화 함수를 적용한다. 이때, skip_layers는 스트라이드가 1보다 큰 경우에만 필요하다. 이 네트워크는 연속되어 길게 연결된 층이기 때문에 Sequential 클래스를 사용하여 모델을 만든다. (ResidualUnit 클래스를 미리 준비해두었으니 잔차 유닛을 하나의 층처럼 취급 가능하다)

```

model = keras.models.Sequential()

model.add(DefaultConv2D(64, kernel_size=7, strides=2,
                        input_shape=[224, 224, 3]))

model.add(keras.layers.BatchNormalization())

model.add(keras.layers.Activation("relu"))

model.add(keras.layers.MaxPool2D(pool_size=3, strides=2, padding="SAME"))
prev_filters = 64

for filters in [64] * 3 + [128] * 4 + [256] * 6 + [512] * 3:
    # 필터 개수가 이전 RU와 동일할 경우 스트라이드를 1로, 아니면 2로 설정
    strides = 1 if filters == prev_filters else 2
    model.add(ResidualUnit(filters, strides=strides))
    prev_filters = filters

model.add(keras.layers.GlobalAvgPool2D())
model.add(keras.layers.Flatten())
model.add(keras.layers.Dense(10, activation="softmax"))

```

for문은 모델에 ResidualUnit층을 더하는 반복루프이다.

3개 RU는 64개의 필터를가지고,그 다음 4개는 RU는 128개의 필터를 가지는 식이다. 필터 개수가 이전 RU와 동일할 경우는 스트라이드를 1로 설정하거나 2로 설정하게 되고, 이후 마지막에 prev_filters를 업데이트한다.

▼ 14.6 케라스에서 제공하는 사전훈련된 모델 사용하기

`keras.applications` 패키지에 준비되어 있는 사전훈련된 모델을 불러올 수 있다.

다음은 이미지넷 데이터셋에서 사전훈련된 ResNet-50 모델을 로드하는 과정이다.

```
model = keras.applications.resnet50.ResNet50(weights="imagenet")
```

이때, 사전훈련된 모델을 사용하려면 이미지가 적절한 크기인지 확인해야 한다. 이때 이미지의 크기가 다르면 `tf.image.resize()` 함수로 적재한 이미지의 크기를 바꿀 수 있다.

ResNet-50 모델은 224 X 224 픽셀 크기의 이미지를 기대하기 때문에 텐서플로의 `tf.image.resize()` 함수로 앞서 적재한 이미지의 크기를 바꿔야 한다.

```
images_resized = tf.image.resize(images, [224, 224])
```

사전 훈련된 모델은 이미지가 적절한 방식으로 전처리되었다고 가정한다. 경우에 따라 (0,1) 또는 (-1,1)사이의 입력을 기대하므로, 이를 위해 모델마다 이미지를 전처리 해주는 `preprocess_input()` 함수를 제공한다. 이 함수는 픽셀값이 0에서 255사이라고 가정하기에 앞에서 0과 1사이로 바꾼 이미지 입력을 다시 바꾸어야 하기 때문에 `image_resized` 에 255를 곱한다.

```
inputs=keras.applications.resnet50.prprocess_input(images_resized*255)
```

이제 사전 훈련된 모델을 사용하여 예측을 수행한다.

```
Y_proba = model.predict(inputs)
```

통상적인 구조대로 Y_proba는 행이 하나의 이미지고 열이 하나의 클래스인 행렬이다.

각 이미지에 대해 최상위 K개의 예측에 대해 클래스 이름, 예측 클래스의 추정 확률을 출력하려면 `decode_predictions()` 함수를 사용한다. 각

이미지에 대해 최상위 K개의 예측을 담은 리스트를 반환한다. 이때, 각 예측은 클래스 아이디, 이름, 확률을 포함하는 튜플이다.

```
op_K = keras.applications.resnet50.decode_predictions(Y_proba, top=3)

for image_index in range(len(images)):
    print("Image #{}".format(image_index))
    for class_id, name, y_proba in top_K[image_index]:
        print(" {} - {:12s} {:.2f}%".format(class_id, name, y_proba * 100))
    print()
```

▼ 14.7 사전훈련된 모델을 사용한 전이 학습

충분하지 않은 훈련 데이터로 이미지 분류기를 훈련하려면 사전훈련된 모델의 하위층을 사용하는 것이 좋다

ex. 사전훈련된 Xception 모델을 사용해 꽃 이미지를 분류하는 모델을 훈련해보자.

먼저 데이터를 적재한 후, `tfds.Split.TRAIN.subsplit` 함수를 이용하여 데이터셋을 나눈다.

```
import tensorflow_datasets as tfds

dataset, info = tfds.load("tf_flowers", as_supervised=True, with_info=True)
dataset_size = info.splits["train"].num_examples
class_names = info.features["label"].names
n_classes = info.features["label"].num_classes

test_set_raw, valid_set_raw, train_set_raw = tfds.load(
    "tf_flowers",
    split=["train[:10%]", "train[10%:25%]", "train[25%:]"],
    as_supervised=True)
```

이후 이미지를 전처리해야 한다.

이CNN 모델은 224X224 크기 이미지를 기대하므로 크기를 조정해야 한다.

```
def preprocess(image, label):
    resized_image = tf.image.resize(image, [224, 224])
    final_image = keras.applications.xception.preprocess_input(resized_image)
    return final_image, label
```

훈련 세트를 섞은 후, 이 전처리 함수들을 데이터셋에 모두 적용하고 배치 크기를 지정하고 프리페치를 적용한다. 이때

`tf.image.random.crop()` 등을 이용해 데이터 증식이 가능하다.

```
def central_crop(image):
    shape = tf.shape(image)
    min_dim = tf.reduce_min([shape[0], shape[1]])
    top_crop = (shape[0] - min_dim) // 4
    bottom_crop = shape[0] - top_crop
    left_crop = (shape[1] - min_dim) // 4
    right_crop = shape[1] - left_crop
    return image[top_crop:bottom_crop, left_crop:right_crop]

def random_crop(image):
    shape = tf.shape(image)
    min_dim = tf.reduce_min([shape[0], shape[1]]) * 90 // 100
    return tf.image.random_crop(image, [min_dim, min_dim, 3])

def preprocess(image, label, randomize=False):
    if randomize:
        cropped_image = random_crop(image)
        cropped_image = tf.image.random_flip_left_right(cropped_image)
    else:
        cropped_image = central_crop(image)
    resized_image = tf.image.resize(cropped_image, [224, 224])
    final_image = keras.applications.xception.preprocess_input(resized_image)
    return final_image, label

batch_size = 32
train_set = train_set_raw.shuffle(1000).repeat()
train_set = train_set.map(partial(preprocess, randomize=True)).batch(batch_size).prefetch(1)
valid_set = valid_set_raw.map(preprocess).batch(batch_size).prefetch(1)
test_set = test_set_raw.map(preprocess).batch(batch_size).prefetch(1)
```


그 다음 이미지넷에서 사전훈련된 Xception 모델 로드한다. 이때 `include_top=False` 로 지정하여 네트워크의 최상층에 해당하는 전역 평균 풀링 층과 밀집 출력 층은 제외시킨다. 이때, 훈련 초기에는 사전훈련된 층의 가중치를 동결한다.

```
base_model = keras.applications.xception.Xception(weights="imagenet",
                                                    include_top=False)
avg = keras.layers.GlobalAveragePooling2D()(base_model.output)
output = keras.layers.Dense(n_classes, activation="softmax")(avg)
model = keras.models.Model(inputs=base_model.input, outputs=output)

for layer in base_model.layers:
    layer.trainable = False

optimizer = keras.optimizers.SGD(learning_rate=0.2, momentum=0.9, decay=0.01)
model.compile(loss="sparse_categorical_crossentropy", optimizer=optimizer,
              metrics=["accuracy"])
history = model.fit(train_set,
                    steps_per_epoch=int(0.75 * dataset_size / batch_size),
                    validation_data=valid_set,
                    validation_steps=int(0.15 * dataset_size / batch_size),
                    epochs=5)
```

이때, 검증 정확도가 어느정도 선에 머물러서 더 나아지지 않는다면 모든 층의 가중치 동결을 해제하고 훈련한다. 이때는 사전훈련된 가중치가 훼손되는 것을 피하기 위해 훨씬 작은 학습률을 사용한다.

```
for layer in base_model.layers:
    layer.trainable = True

optimizer = keras.optimizers.SGD(learning_rate=0.01, momentum=0.9,
                                   nesterov=True, decay=0.001)
model.compile(loss="sparse_categorical_crossentropy", optimizer=optimizer,
              metrics=["accuracy"])
history = model.fit(train_set,
                    steps_per_epoch=int(0.75 * dataset_size / batch_size),
                    validation_data=valid_set,
                    validation_steps=int(0.15 * dataset_size / batch_size),
                    epochs=40)
```