



HAJ Lecture

Chapter 14. 합성곱 신경망을 사용한 컴퓨터 비전

Hands-On Machine Learning
with Scikit-Learn, Keras & TensorFlow

14.1 시각 피질 구조

14.2 합성곱 층

14.3 풀링 층

14.4 CNN 구조

14.5 케라스를 사용해 ResNet-34 CNN 구현하기

14.6 케라스에서 제공하는 사전훈련된 모델 사용하기

14.7 사전훈련된 모델을 사용한 전이 학습

14.8 분류와 위치 추정

14.9 객체 탐지

14.10 시맨틱 분할

INDEX

| 14.1 시각 피질 구조

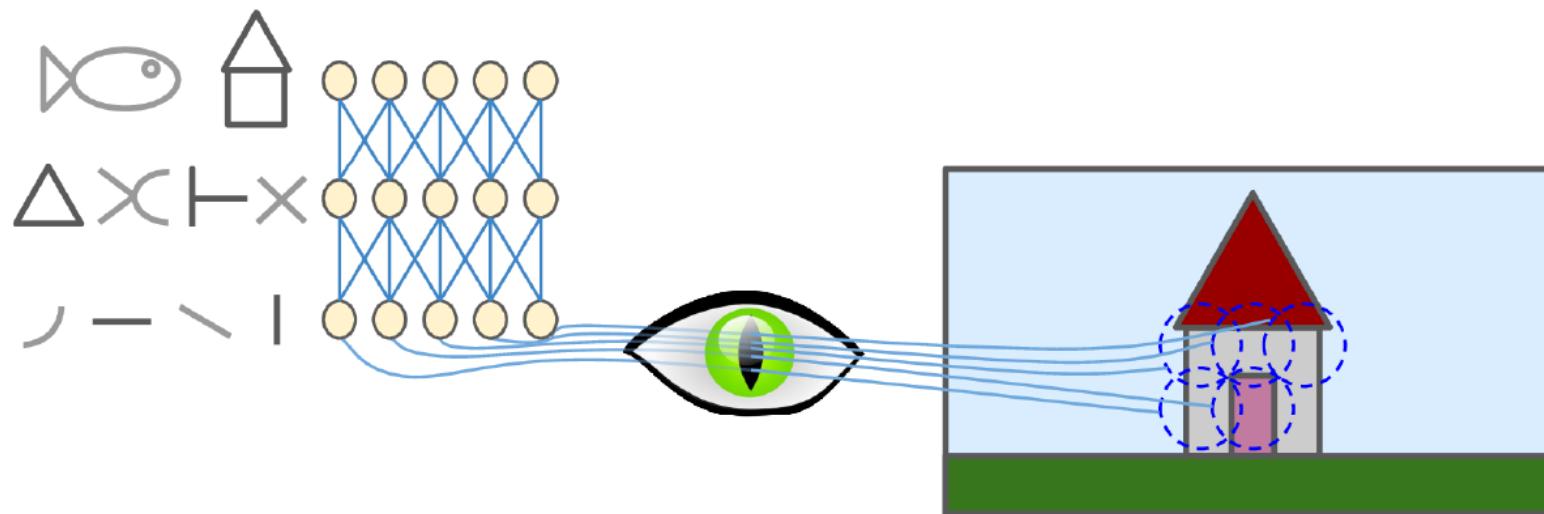
in 1958,

시각 피질 안의 많은 뉴런들이 작은 국부 수용장을 가진다는 것이 밝혀짐

→ 어떤 뉴런은 수평선 이미지에만, 어떤 것은 다른 각도 선분에 반응함

→ 또 어떤 뉴런은 큰 수용장을 가져 저수준 패턴이 조합된 더 복잡한 패턴에 반응

→ 고수준 뉴런이 이웃한 저수준 뉴런의 출력에 기반한다.



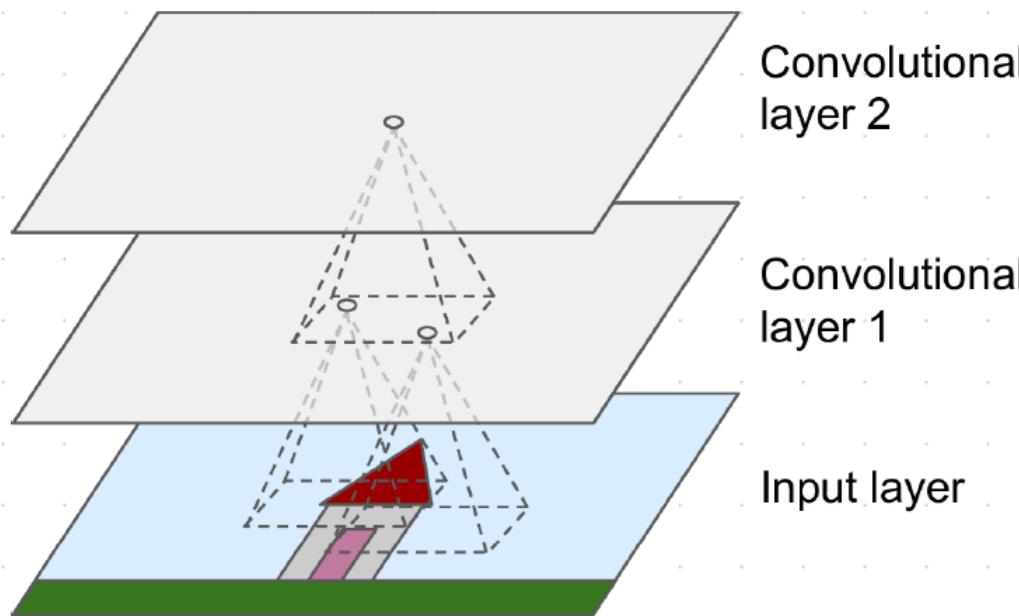
14.2 합성곱 층

합성곱 층(convolutional layer)

- 합성곱(convolution) 연산

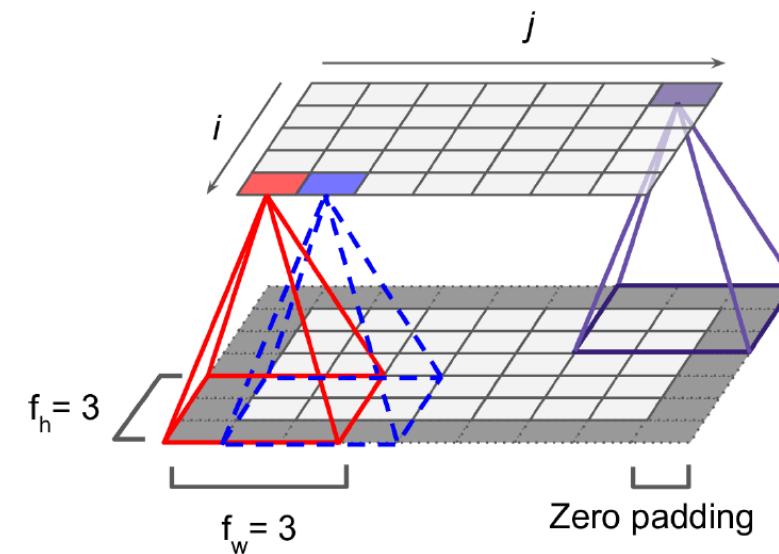
다른 함수 위를 이동하면서 원소별 곱셈의 적분을 계산하는 수학 연산

→ 라플라스 변환(laplace transform), 푸리에 변환(Fourier transform)



$$f(x) * g(x) = \int_{-\infty}^{\infty} f(\tau)g(t - \tau)d\tau$$

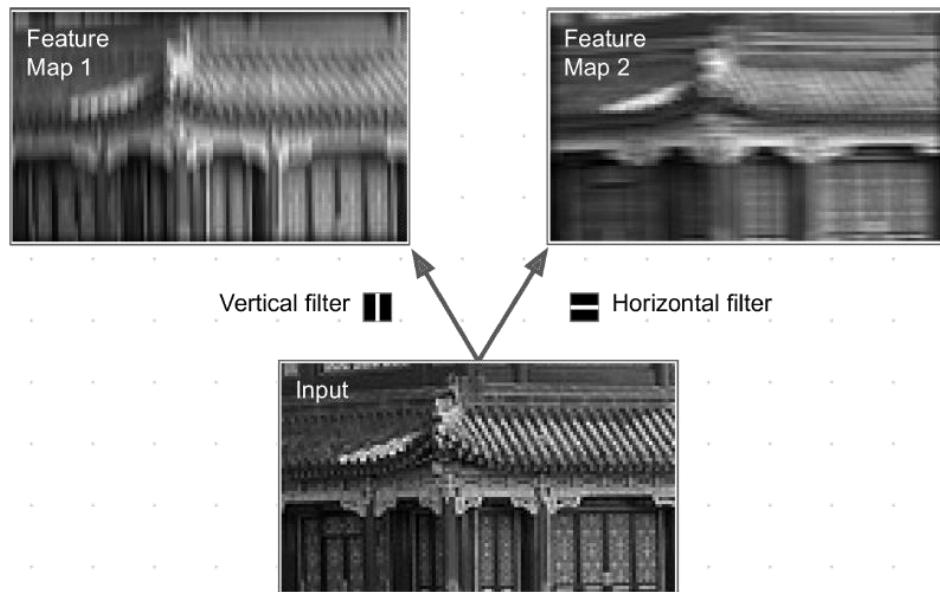
스트라이드(stride)
한 수용장과 다음 수용장 사이의 간격



| 14.2 합성곱 층

필터(filter)

- 합성곱 커널(convolution kernel)
- 가중치 세트
- 층의 전체 뉴런에 적용된 하나의 필터는 **특성 맵(feature map)**을 만든다.
→ 필터를 가장 크게 활성화시키는 이미지의 영역을 강조



필터에서 흰색 부분을 제외하고는 수용장의 모든 것을 무시한다.
(흰색 부분을 제외하고는 입력에 모두 0이 곱해짐)

수동으로 필터를 정의할 필요X
→ 훈련 도중 합성곱 층이 자동으로 해당 문제에 유용한
필터를 찾고 상위층은 이들을 연결하기 때문

| 14.2 합성곱 층

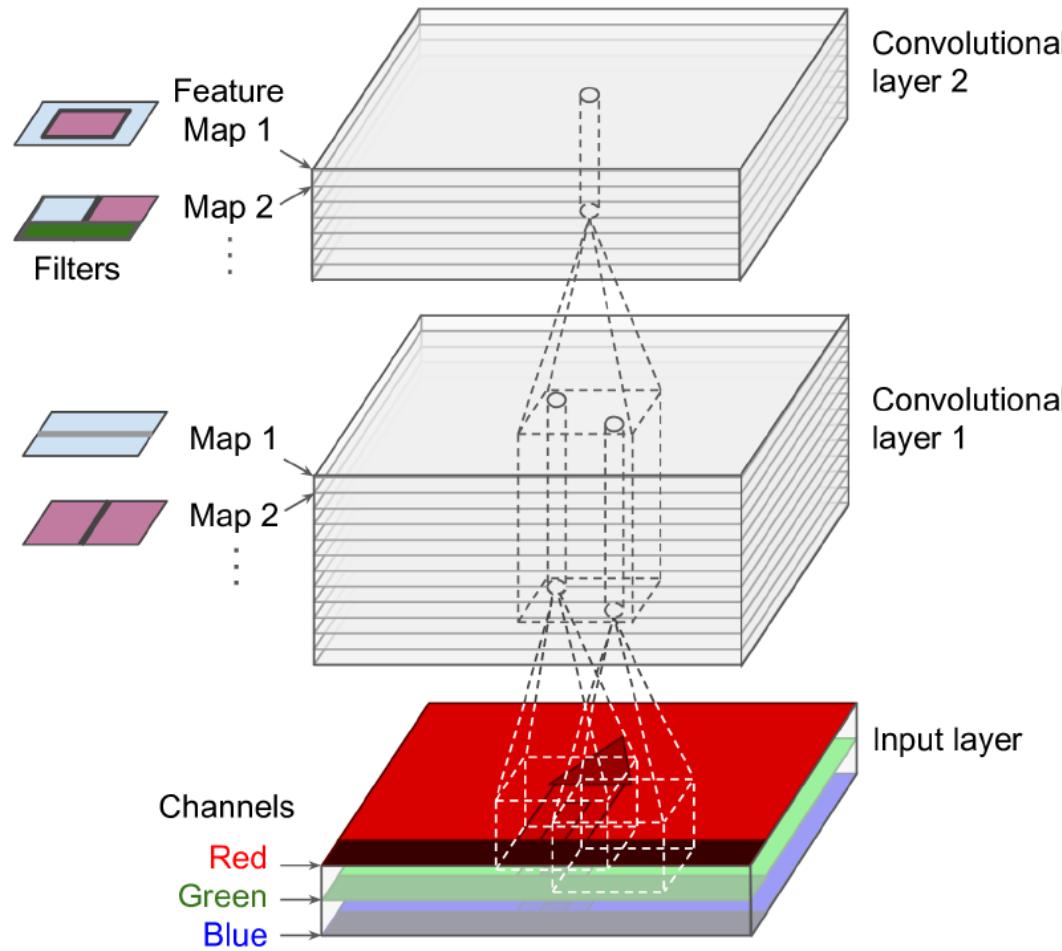
여러 가지 특성 맵 쌓기

실제 합성곱은 여러 가지 필터를 가지고, 필터마다 하나의 특성맵을 출력
→ 3D로 표현하는 것이 더 정확하다.

→ 각 특성 맵의 픽셀은 하나의 뉴런에 대응되고,
하나의 특성 맵 안에서는 모든 뉴런이 같은 파라미터를 공유
다른 특성 맵에 있는 뉴런은 다른 파라미터를 사용

14.2 합성곱 층

여러 가지 특성 맵 쌓기



여러 가지 특성 맵으로 이루어진
합성곱 층과 3개의 컬러 채널을 가진 이미지

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

with $\begin{cases} i' = i \times s_h + u \\ j' = j \times s_w + v \end{cases}$

| 14.2 합성곱 층

여러 가지 특성 맵 쌓기

- 각 입력 이미지는 [높이, 너비, 채널] 형태의 3D 텐서로 표현된다.
- 하나의 미니배치는 [미니배치 크기, 높이, 너비, 채널] 형태의 4D 텐서로 표현된다.

```
from sklearn.datasets import load_sample_image

# Load sample images
china = load_sample_image("china.jpg") / 255
flower = load_sample_image("flower.jpg") / 255
images = np.array([china, flower])
batch_size, height, width, channels = images.shape

# Create 2 filters
filters = np.zeros(shape=(7, 7, channels, 2), dtype=np.float32)
filters[:, :, :, 0] = 1 # vertical line
filters[3, :, :, 1] = 1 # horizontal line

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

plt.imshow(outputs[0, :, :, 1], cmap="gray") # plot 1st image's 2nd feature map
plt.show()
```

14.2 합성곱 층

여러 가지 특성 맵 쌓기

`tf.nn.conv2d()`

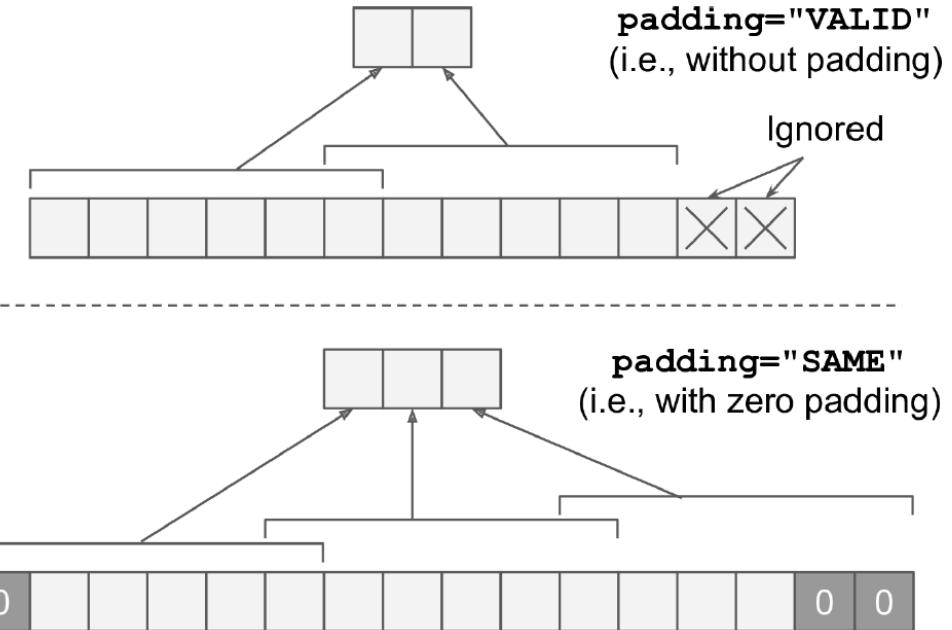
- **images**: 입력의 미니배치(\rightarrow 4D 텐서)
- **filters**: 적용된 일련의 필터(\rightarrow 4D 텐서)
- **strides**: 10이나 4개의 원소를 갖는 1D 배열

(수직 스트라이드(s_h), 수평 스트라이드(s_w), 배치 스트라이드, 채널

스트라이드

- **padding**: “VALID”- 합성곱 층에 제로 패딩을 사용하지 X
“SAME”- 필요한 경우 제로 패딩을 사용한다

e.g. `conv = keras.layers.Conv2D(filters=32, kernel_size=3, strides=1, padding="SAME", activation="relu")`

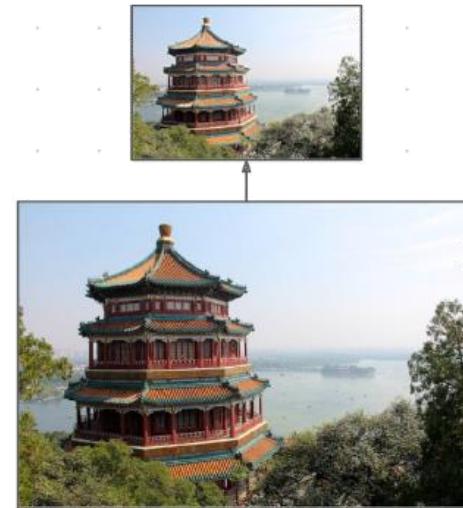
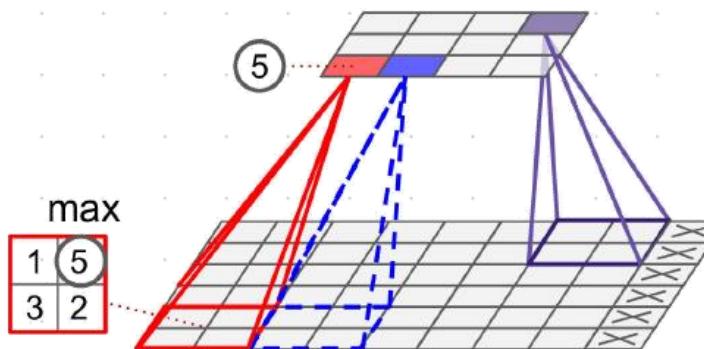


14.3 풀링 층

풀링 층(Pooling layer)

목적: 계산량/메모리 사용량/파라미터 수를 줄이기 위해 입력 이미지의 부표본(subsample)을 만들자.

- 크기, 스트라이드, 패딩 유형 등을 지정해야 한다.
- 가중치가 없다.(합성곱 층과 비교되는 점)



최대 풀링 층(max pooling layer)
2x2 풀링 커널, 스트라이드 = 2
패딩 없음
→ 각 수용장에서 가장 큰 입력값이
다음 층으로 전달되고 다른 값은 버려짐

14.3 풀링 층

- 일정 수준의 **불변성(invariance)**을 만들어준다.

e.g. 3개의 이미지는 우측으로 한칸씩 이동한 것

→ A, B에서는 최대 풀링 층의 출력이 동일하다.

(이동 불편성 – translation invariance)

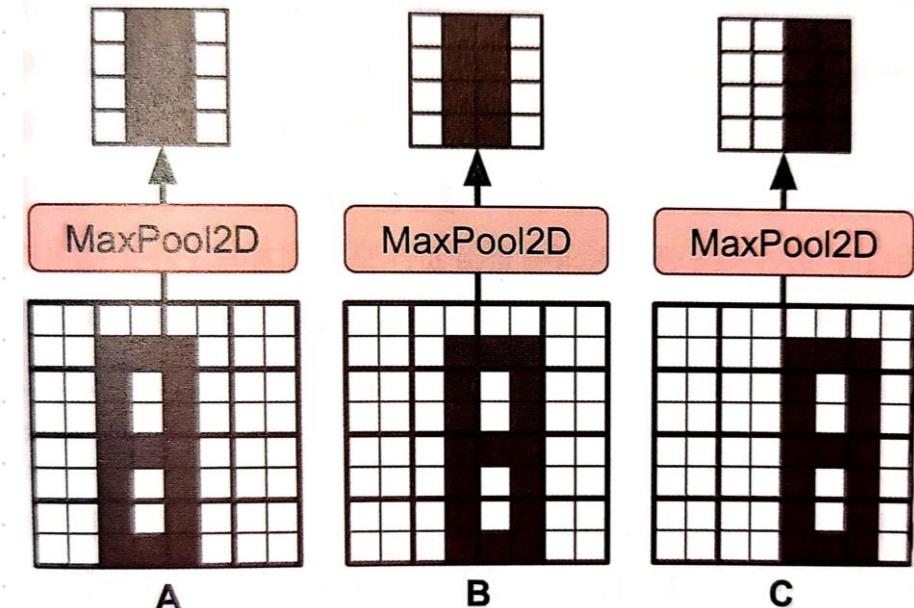
- 단점: 매우 파괴적인 특성

e.g. 2x2 커널과 스트라이드 2

→ 출력은 양방향으로 절반이 줄어들고

면적은 $1/4$ 이 되어 입력값의 75%를 소실

→ 시맨틱 분할 등에서는 이러한 특징이 바람직하지 X(→ **등변성(equivariance)**)



14.3 풀링 층

TensorFlow로 구현

e.g. 2x2 커널을 사용, 스트라이드 2

기본적으로 “valid” 패딩을 사용 → 패딩 X

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

- 평균 풀링 층 (average pooling layer)

AvgPool2D

→ 최댓값이 아닌 평균을 계산하는 것을 제외하고는 동일하게 작동

- 일반적으로 최대 풀링 층이 성능이 더 우월하여 이를 사용함
- 평균을 계산하면 최댓값을 계산하는 것보다 정보 손실이 적다.
- 최대 풀링은 의미 없는 것을 모두 제거하고 가장 큰 특징만을 유지

14.3 풀링 층

최대 풀링과 평균 풀링은 공간 차원이 아니라, 깊이 차원으로도 수행 가능

`tf.nn.max_pool()`

→ 커널 크기와 스트라이드를 4개의 원소를 가진 튜플로 지정

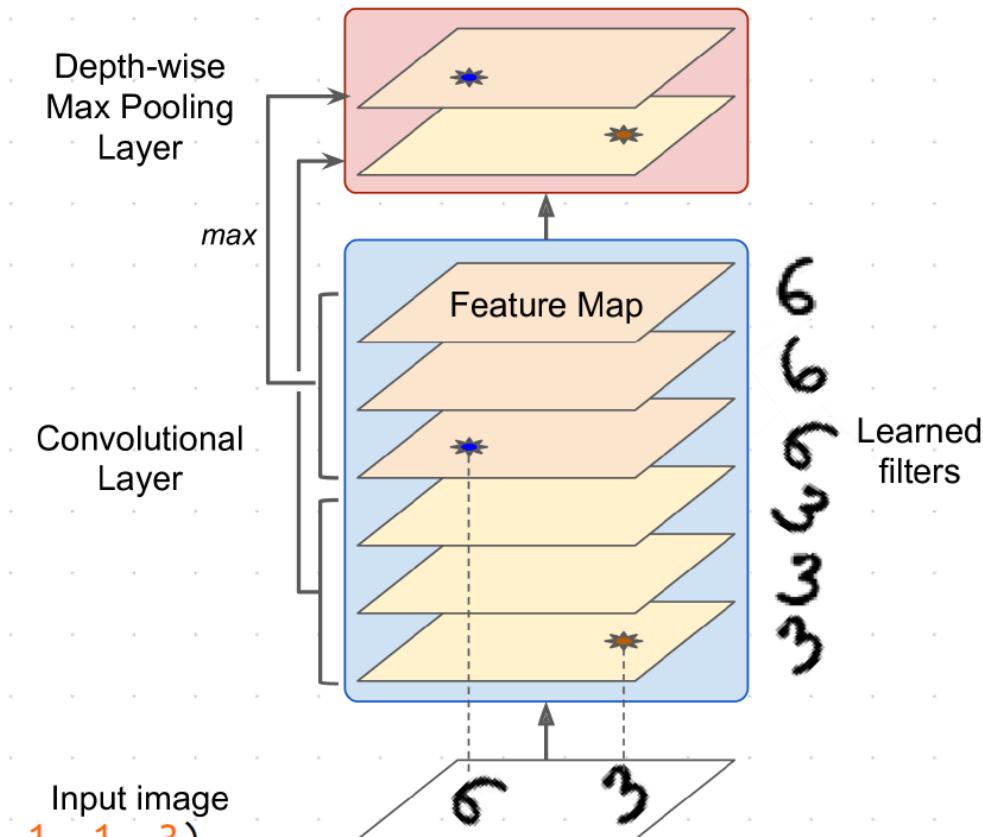
→ 첫번째 3개 값은 모두 1

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



케라스 모델의 층으로 사용하고 싶다면?

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



| 14.3 풀링 층

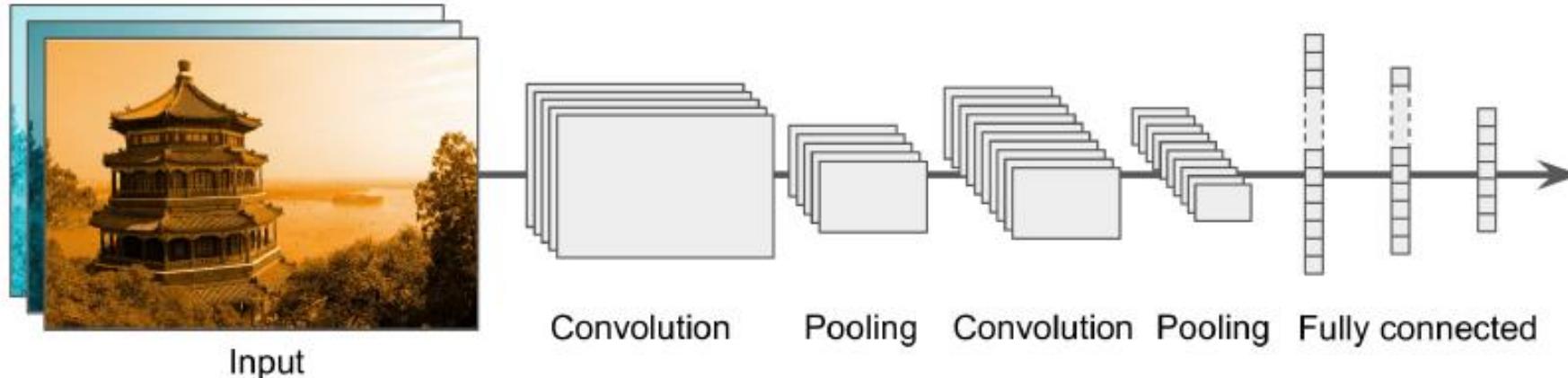
전역 평균 풀링 층(global average pooling layer)

- 각 특성 맵의 평균을 계산하는 것
- 각 샘플의 특성 맵마다 하나의 숫자를 출력한다. → 매우 파괴적인 연산
→ 특성 맵에 있는 대부분의 정보를 잃게되기 때문
- keras.layers.GlobalAvgPool2D
`global_avg_pool = keras.layers.GlobalAvgPool2D()`
- 공간 방향(높이와 너비)를 따라 평균을 계산하는 Lambda 층과 동등
`global_avg_pool = keras.layers.Lambda(lambda X: tf.reduce_mean(X, axis=[1, 2]))`

14.4 CNN 구조

입력 → 합성곱 → 풀링 → 합성곱 → 풀링 → ⋯

네트워크를 통과하여 진행할수록 이미지는 점점 작아지지만,
합성곱 층 때문에 일반적으로 점점 더 깊어진다.(→ 더 많은 특성 맵을 가짐)



14.4 CNN 구조

```
from functools import partial

DefaultConv2D = partial(keras.layers.Conv2D,
                      kernel_size=3, activation='relu', padding="SAME")

model = keras.models.Sequential([
    DefaultConv2D(filters=64, kernel_size=7, input_shape=[28, 28, 1]),
    keras.layers.MaxPooling2D(pool_size=2),
    DefaultConv2D(filters=128),
    DefaultConv2D(filters=128),
    keras.layers.MaxPooling2D(pool_size=2),
    DefaultConv2D(filters=256),
    DefaultConv2D(filters=256),
    keras.layers.MaxPooling2D(pool_size=2),
    keras.layers.Flatten(),
    keras.layers.Dense(units=128, activation='relu'),
    keras.layers.Dropout(0.5),
    keras.layers.Dense(units=64, activation='relu'),
    keras.layers.Dropout(0.5),
    keras.layers.Dense(units=10, activation='softmax'),
])
```

14.4 CNN 구조

LeNet-5

출력층 – 입력과 가중치 벡터를 행렬곱하는 대신, 각 뉴런에서
입력 벡터와 가중치 벡터 사이의 유클리드 거리를 출력
→ 각 출력은 얼마나 특정 숫자 클래스에 속하는지를 측정해줌

Layer	Type	Maps	Size	Kernel size	Stride	Activation
Out	Fully Connected	–	10	–	–	RBF
F6	Fully Connected	–	84	–	–	tanh
C5	Convolution	120	1×1	5×5	1	tanh
S4	Avg Pooling	16	5×5	2×2	2	tanh
C3	Convolution	16	10×10	5×5	1	tanh
S2	Avg Pooling	6	14×14	2×2	2	tanh
C1	Convolution	6	28×28	5×5	1	tanh
In	Input	1	32×32	–	–	–

14.4 CNN 구조

처음으로 합성곱 층 위에 풀링 층을 쌓지 않고
바로 합성곱 층끼리 쌓는 구조를 선보임

AlexNet

Layer	Type	Maps	Size	Kernel size	Stride	Padding	Activation
Out	Fully Connected	–	1,000	–	–	–	Softmax
F9	Fully Connected	–	4,096	–	–	–	ReLU
F8	Fully Connected	–	4,096	–	–	–	ReLU
C7	Convolution	256	13×13	3×3	1	SAME	ReLU
C6	Convolution	384	13×13	3×3	1	SAME	ReLU
C5	Convolution	384	13×13	3×3	1	SAME	ReLU
S4	Max Pooling	256	13×13	3×3	2	VALID	–
C3	Convolution	256	27×27	5×5	1	SAME	ReLU
S2	Max Pooling	96	27×27	3×3	2	VALID	–
C1	Convolution	96	55×55	11×11	4	VALID	ReLU
In	Input	3 (RGB)	227×227	–	–	–	–

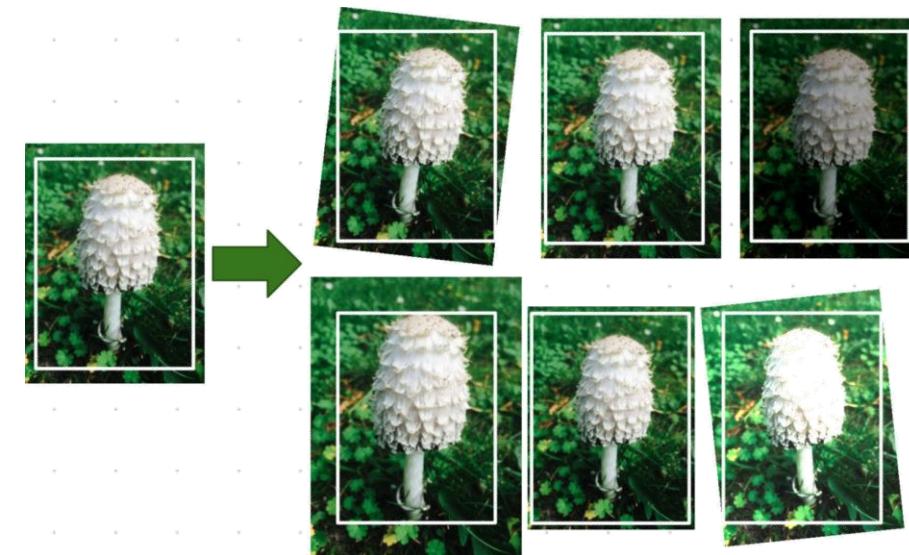
| 14.4 CNN 구조

AlexNet

- 과대적합을 줄이기 위해 두 가지 규제 기법을 도입함
 - 훈련하는 동안 F9, F10 출력에 드롭아웃을 50% 비율로 적용함
 - 데이터 증식(data augmentation)

훈련 이미지를 랜덤하게 여러 간격으로 이동하거나

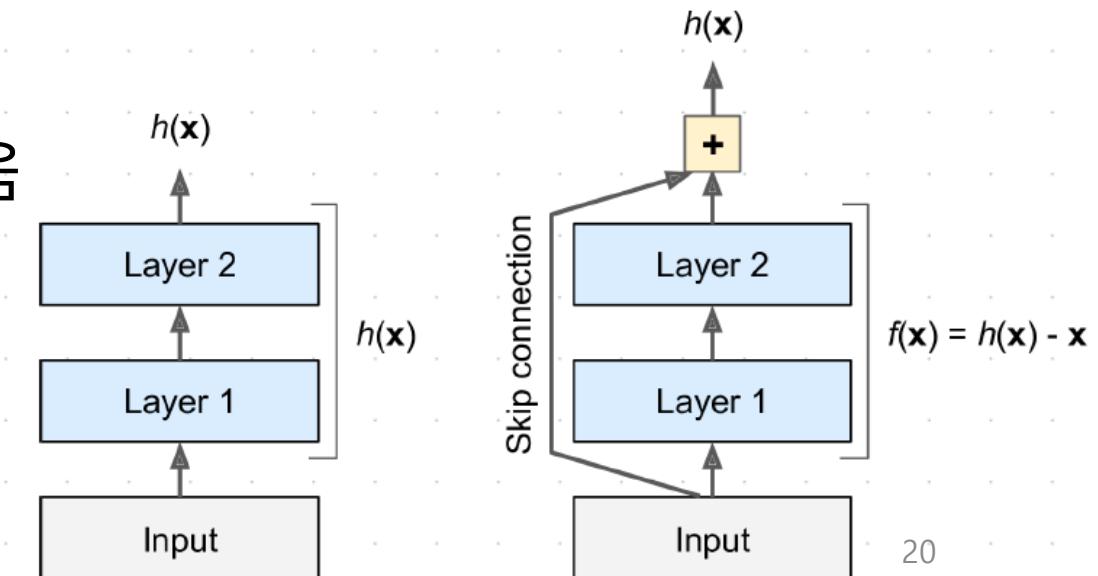
수평으로 뒤집고 조명을 바꾸는 식



14.4 CNN 구조

ResNet

- 더 적은 파라미터를 이용해 점점 더 깊은 네트워크로 모델을 구성하는 일반적인 트렌드 → **스킵 연결**(skip connection, shortcut connection)
- 신경망 훈련의 목적? 목적 함수 $h(x)$ 모델링
→ x 를 네트워크 출력에 더하면(스킵연결 추가), $f(x) = h(x) - x$ 를 학습하게 됨
(잔차 학습)
- 일반적인 신경망에서는 가중치가 0에 가까움
→ 초기에는 항등 함수

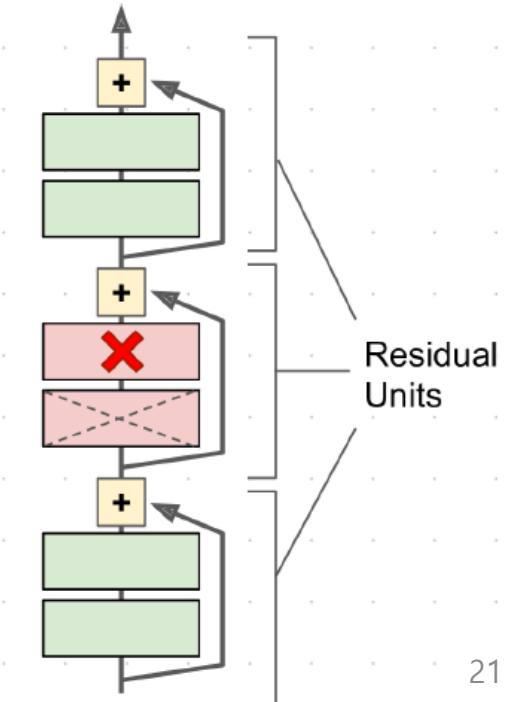
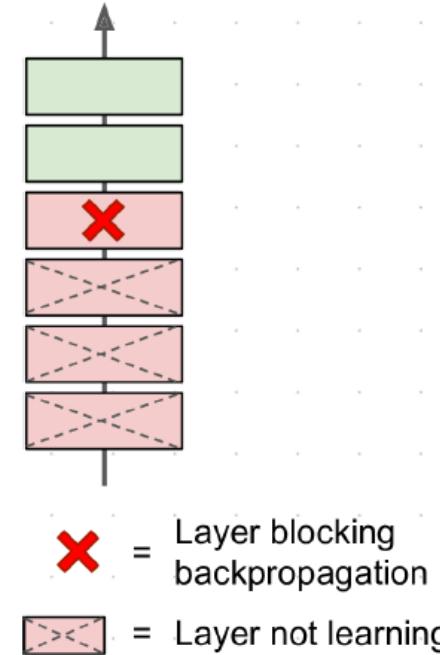


14.4 CNN 구조

ResNet

- 스킵 연결을 많이 추가하면 일부 층이 아직 학습되지 않았더라도 네트워크는 훈련을 시작할 수 있다.
- 스킵 연결로 인해 입력 신호가 전체 네트워크에 손쉽게 영향을 미치게 된다.
- **잔차 유닛(RU; residual unit)**

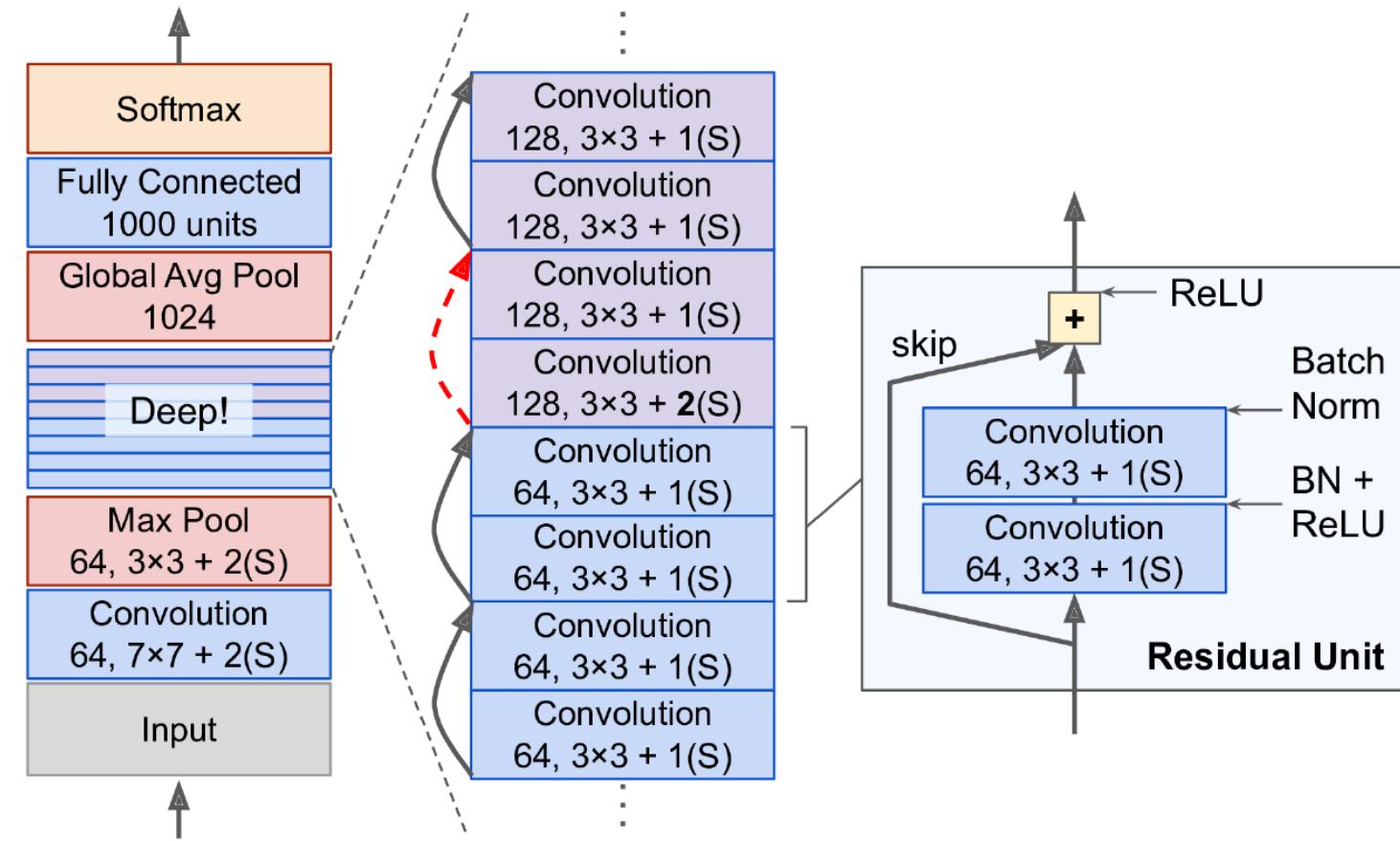
스킵 연결을 가진 작은 신경망



14.4 CNN 구조

ResNet

- 64개의 특성맵을 출력하는 3개 RU
- 128개 맵의 4개 RU
- 256개 맵의 6개 RU
- 512개 맵의 3개 RU

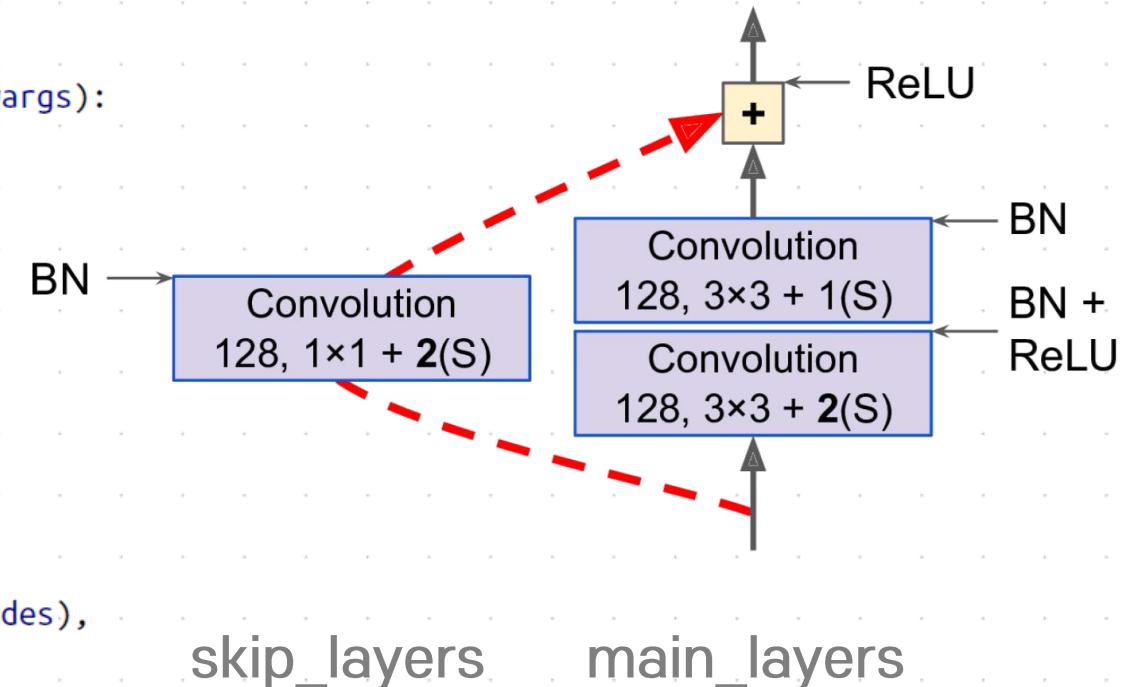


14.5 케라스를 사용해 ResNet-34 CNN 구현하기

```
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)
```



14.5 케라스를 사용해 ResNet-34 CNN 구현하기

```
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:
    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"))
```

처음 3개 RU는 64개의 필터를
그다음 4개 RU는 128개의 필터를 가진다.
→ 필터개수가 이전 RU와 동일하면 스트라이드 = 1
→ 아니면 스트라이드 = 2

| 14.6 케라스에서 제공하는 사전훈련된 모델 사용하기

표준모델은 직접 구현할 필요X → keras.application 패키지 사용

e.g. ResNet-50 모델 불러오기

기본적으로 224 x 224 pixel를 기대함

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

Q. 이미지가 적절한 크기가 아니라면? → 크기를 바꿔야 한다.

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

경우에 따라서는 [0, 1] 또는 [-1, 1] 범위의 입력이 필요할 때도 있다.

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

사전훈련된 모델을 사용해 예측을 수행

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

14.6 케라스에서 제공하는 사전훈련된 모델 사용하기

```
top_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()
```

Image #0

n03877845 - palace	42.87%
n02825657 - bell_cote	40.57%
n03781244 - monastery	14.56%

Image #1

n04522168 - vase	46.83%
n07930864 - cup	7.78%
n11939491 - daisy	4.87%

14.7 사전훈련된 모델을 사용한 전이 학습

충분하지 않은 훈련데이터로 이미지 분류기를 훈련하려면?

사전훈련된 모델의 하위층을 사용하는 것이 좋다.

```
import tensorflow_datasets as tfds

dataset, info = tfds.load("tf_flowers", as_supervised=True, with_info=True)
dataset_size = info.splits["train"].num_examples # 3670
class_names = info.features["label"].names # ["dandelion", "daisy", ...]
n_classes = info.features["label"].num_classes # 5

test_split, valid_split, train_split = tfds.Split.TRAIN.subsplit([10, 15, 75])

test_set = tfds.load("tf_flowers", split=test_split, as_supervised=True)
valid_set = tfds.load("tf_flowers", split=valid_split, as_supervised=True)
train_set = tfds.load("tf_flowers", split=train_split, as_supervised=True)
```

데이터셋에 대한 정보도
함께 load함

10%-테스트 세트
15%-검증 세트
75%-훈련 세트

14.7 사전훈련된 모델을 사용한 전이 학습

```
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
```

이미지 전처리 단계
244 × 244 크기

```
batch_size = 32
train_set = train_set.shuffle(1000).repeat()
train_set = train_set.map(preprocess).batch(batch_size).prefetch(1)
valid_set = valid_set.map(preprocess).batch(batch_size).prefetch(1)
test_set = test_set.map(preprocess).batch(batch_size).prefetch(1)
```

훈련세트를 골고루 섞음
전처리함수를 3개 데이터셋에 모두 적용

```
base_model = keras.applications.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)
```

사전훈련된 Xception 모델을 로드
include_top=False
→ 네트워크 최상층의 전역평균 풀링 층
과 밀집 출력 층을 제외시킴

14.7 사전훈련된 모델을 사용한 전이 학습

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

훈련 초기에 사전훈련된 층의 가중치를 잠시 동결

```
optimizer = keras.optimizers.SGD(lr=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)
```

모델 컴파일 후 훈련 시작

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

몇 번의 epoch를 거친 후
더 나아지지 않는 상황에 도달
→ 모든 층의 동결 해제
→ 사전훈련된 가중치의 훼손을 막기 위해
초기에는 작은 학습률을 사용

```
optimizer = keras.optimizers.SGD(lr=0.01, momentum=0.9, decay=0.001)  
model.compile(...)  
history = model.fit(...)
```

| 14.8 분류와 위치 추정

사진에서 물체의 위치를 추정하는 것

= 물체 주위의 바운딩(bounding box)을 예측하는 것

= 물체 중심의 수평좌표, 수직좌표, 높이, 너비를 예측하는 것

= 4개의 숫자를 예측하는 것 → 회귀(Regression)

```
base_model = keras.applications.Xception(weights="imagenet",
                                            include_top=False)
avg = keras.layers.GlobalAveragePooling2D()(base_model.output)
class_output = keras.layers.Dense(n_classes, activation="softmax")(avg)
loc_output = keras.layers.Dense(4)(avg)
model = keras.models.Model(inputs=base_model.input,
                           outputs=[class_output, loc_output])
model.compile(loss=["sparse_categorical_crossentropy", "mse"],
              loss_weights=[0.8, 0.2], # 어떤 것을 중요하게 생각하느냐에 따라
              optimizer=optimizer, metrics=["accuracy"])
```

| 14.8 분류와 위치 추정

바운딩 박스가 준비되었다면…

→ 클래스 레이블, 바운딩 박스와 함께 전처리된 이미의 배치가
하나의 원소인 데이터셋을 만들어야 한다.

→ (images, (class_labels, bounding_boxes)) 형태의 튜플

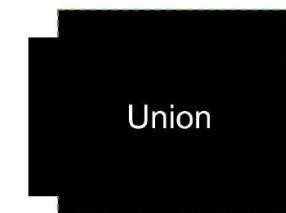
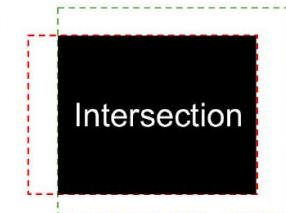
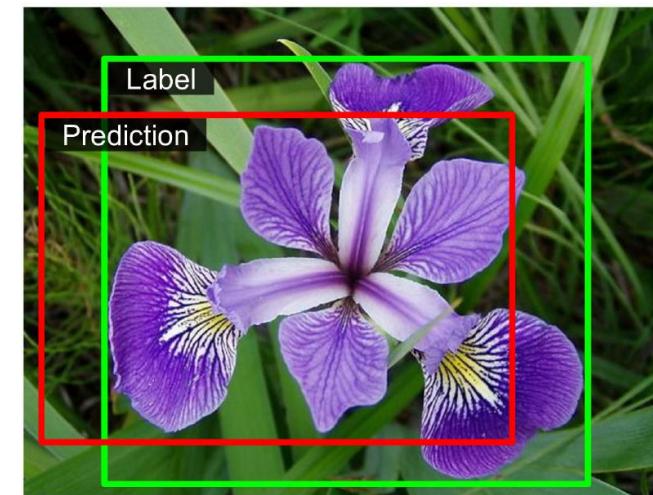
- MSE가 이 상황에서 좋은 평가 지표가 될 수 있을까?

IoU(intersection over union)

예측한 바운딩 박스와 타깃 바운딩 박스

사이에 중첩되는 영역을 전체영역으로 나눈 것

`tf.keras.metrics.MeanIoU`



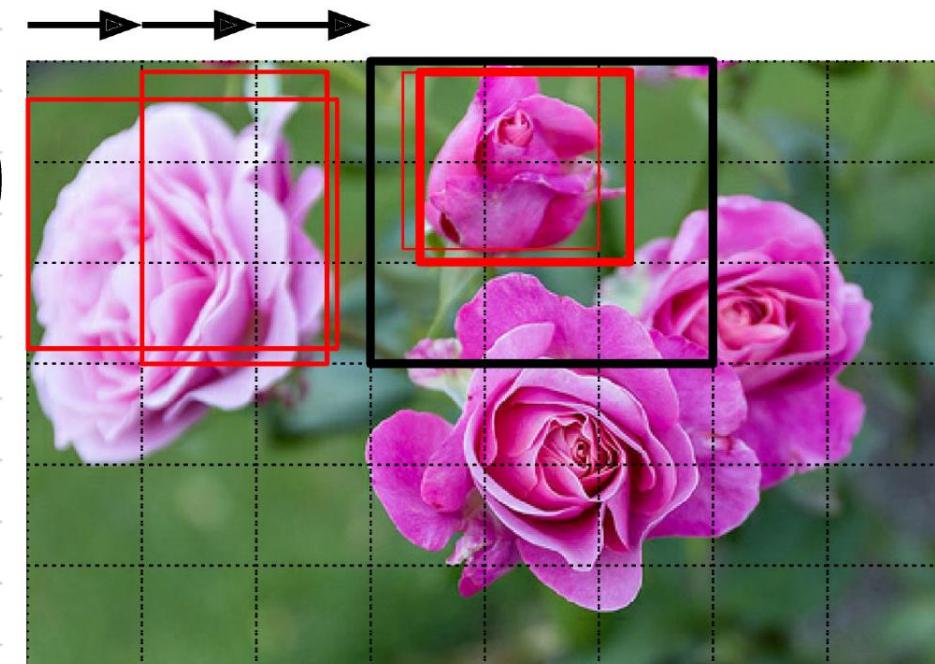
| 14.9 객체 탐지

Q. 하나의 물체를 분류하고 위치를 추정하는 것은 14.8절 내용

→ 하나의 이미지에 여러 물체가 들어있는 경우 각각을 인식해야 한다면?

객체 탐지(object detection)

하나의 이미지에서 여러 물체를 분류하고 위치를 추정하는 작업



가장 간단한 방법은? (→ 효율적이지는 않지만)

분류기를 훈련한 다음 이미지를 모두 훑는 것

윗부분부터 이미지를 차례로 훑는 것이기에,
동일한 물체를 여러 번 감지하게 된다.

→ 불필요한 바운딩 박스를 제거하는 작업이 소요됨

14.9 객체 탐지

NMS(non-max suppression)

- 또 다른 객체가 존재하는지에 대한 확률을 추정하기 위해 존재여부 출력을 추가
with 시그모이드 활성화 함수, 이진 크로스 엔트로피 손실
- 존재여부가 임곗값 이하인 바운딩 박스를 모두 삭제
- 존재여부가 가장 높은 바운딩 박스를 찾고 이와 많이 중첩된(IoU)
다른 바운딩 박스를 모두 제거
- 제거할 바운딩 박스가 더 없는 상황에 다다르기 전까지 이를 계속 반복

| 14.9 객체 탐지

완전 합성곱 신경망

IDEA] 맨 위의 밀집 층을 합성곱 층으로 바꾸자.

e.g. 7×7 맵 크기 100개의 특성 맵 출력하는 합성곱 층

뉴런이 200개 있는 밀집층

→ 각 뉴런은 $100 \times 100 \times 7$ 크기의 활성화 값에 대한 가중치 합 계산

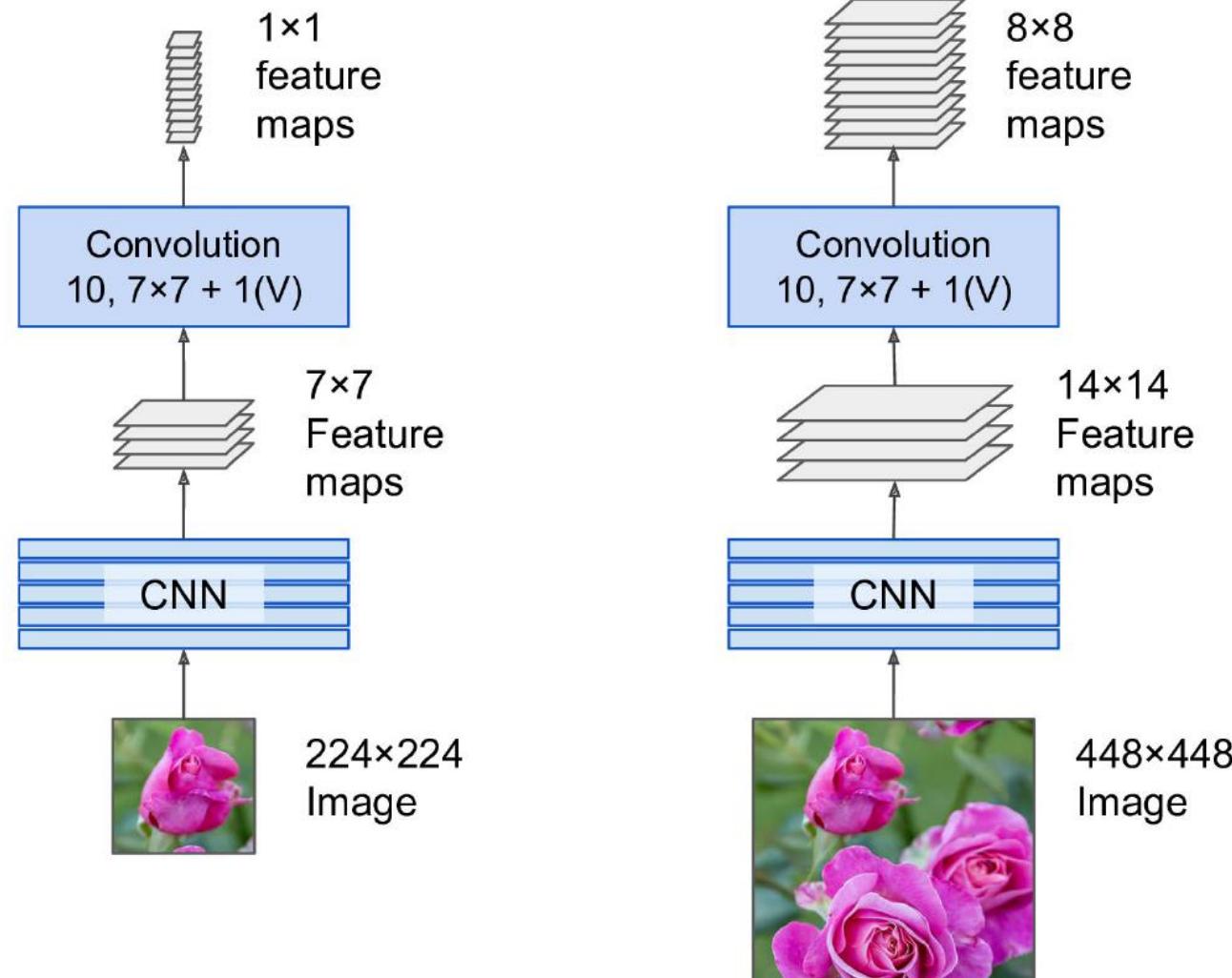
이 밀집 층을 7×7 크기의 필터 200개 + “valid” 패딩 합성곱 층으로 바꾼다면?

→ 1×1 크기의 특성 맵 200개 출력

∴ [배치 크기, 200] vs. [배치 크기, 1, 1, 200]

14.9 객체 탐지

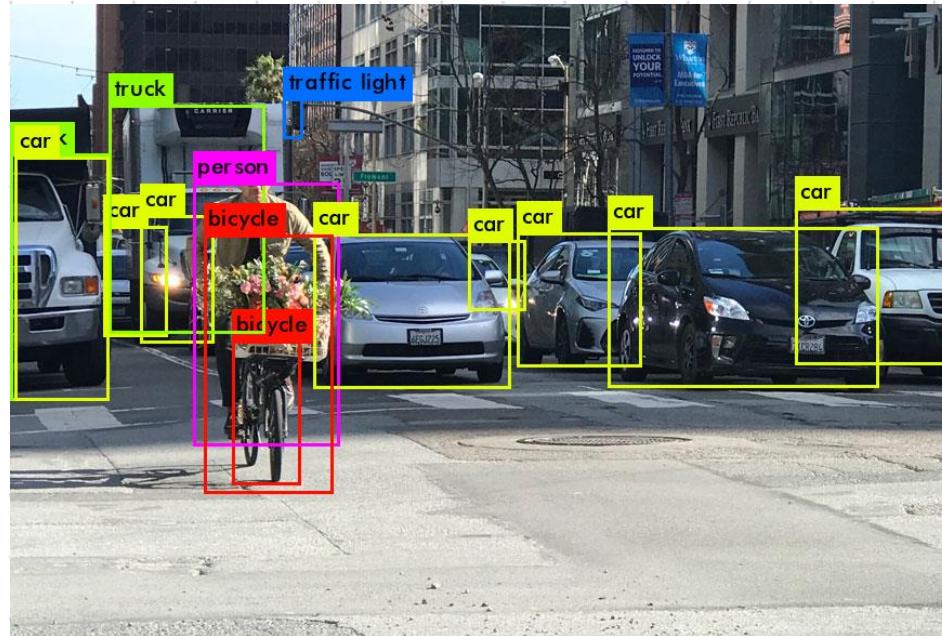
완전 합성곱 신경망



14.9 객체 탐지

YOLO

“you only look once” → 이미지를 딱 한 번만 처리하는 효율적인 방법



<https://zhanghanduo.github.io/post/yolo1/>

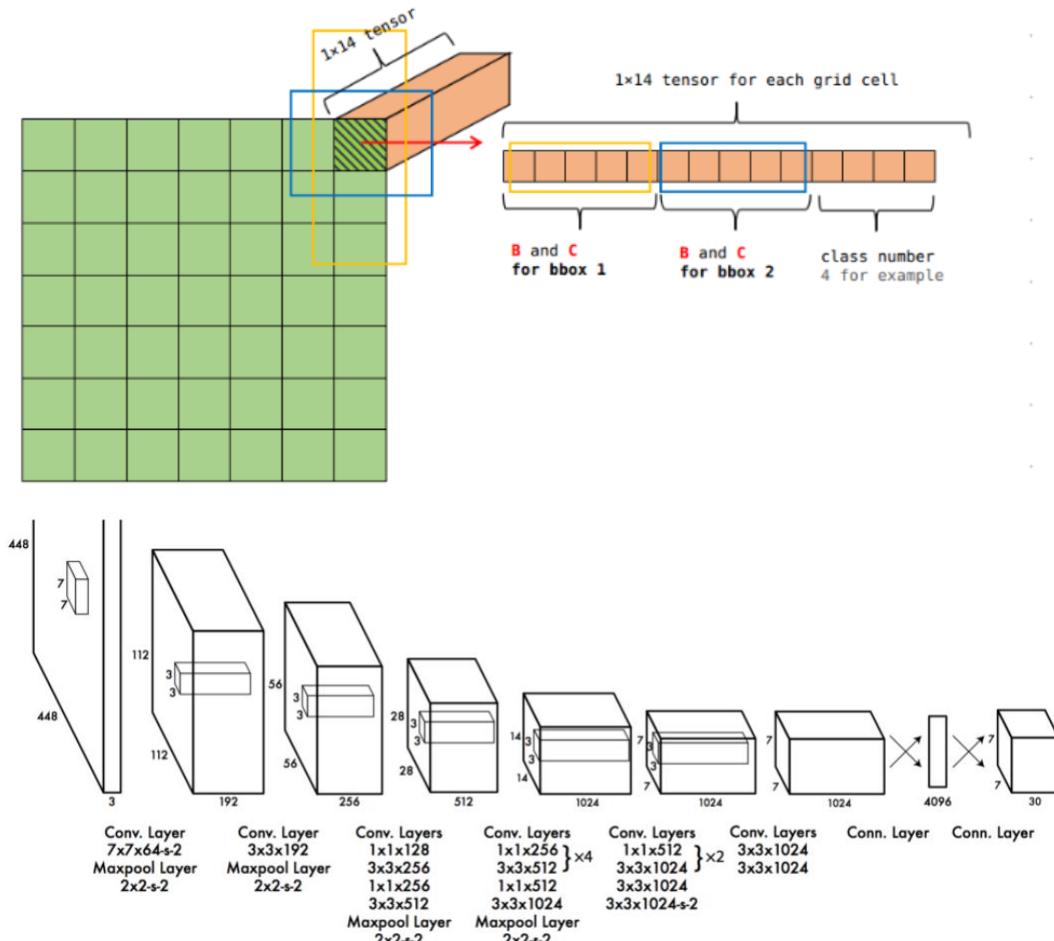
YOLOv3

각 격자 셀마다 5개의 바운딩 박스를 출력

→ 각 바운딩 박스마다 하나의 존재여부 점수가 부여됨

→ 결과: 4개의 좌표를 가진 5개의 바운딩 박스

5개의 존재여부 점수, 20개의 클래스 확률



14.9 객체 탐지

YOLO



<https://zhanghanduo.github.io/post/yolov2/>

14.10 시맨틱 분할

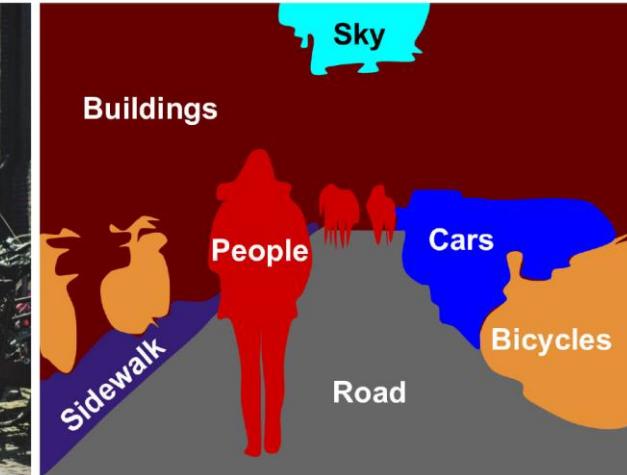
시맨틱 분할(semantic segmentation)

각 픽셀을 그것이 속한 객체의 클래스로 분류하는 것

난점) 이미지가 일반적인 CNN을 통과할 때

점진적으로 위치 정보를 잃는다.

(1 이상의 스트라이드를 사용하는 층)



해결) 사전훈련된 CNN을 FCN으로 변환

→ CNN이 입력 이미지에 적용하는 전체 스트라이드 = 32 (1보다 큰것을 모두 더함)

→ 마지막 층이 입력 이미지보다 32배 작은 특성 맵을 만든다.

14.10 시맨틱 분할

업샘플링 층(upsampling layer)

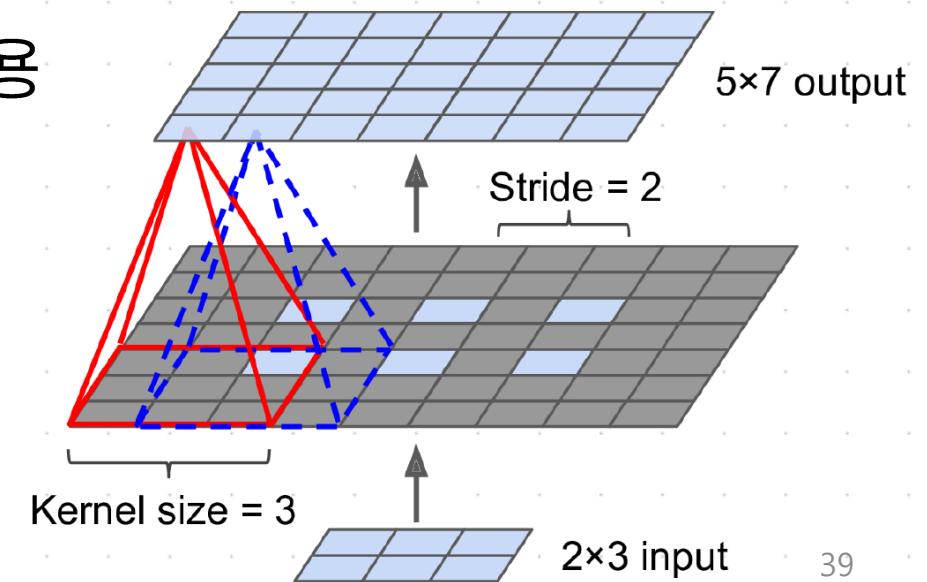
해상도를 32배 늘리기 위해 추가된 층

- i) 선형 보간법(bi-linear interpolation) → 4~8배 늘리는 데에 적합
- ii) 전치 합성곱(transposed convolutional layer)
 - 이미지에 0으로 채워진 빈 행/열을 삽입하여 늘린 후 일반적 합성곱 수행
 - in tf.Keras → Conv2DTranspose 층을 사용

in 전치 합성곱 층

스트라이드는 필터의 스텝 크기가 아니라, 입력이 얼마나
늘어나는지로 정의한다.

→ 스트라이드가 클수록 출력이 커진다.(일반적 층과는 다름)



| 14.10 시맨틱 분할

- keras.layers.Conv1D

1D 입력에 대한 합성곱 층을 만든다. e.g. 시계열, 텍스트 등

- keras.layers.Conv3D

3D 입력에 대한 합성곱 층을 만든다. e.g. 3D PET 스캔 등

- dilation_rate

tf.keras 합성곱 층에 있는 dilation_rate를 2 이상으로 바꾸면 아트루스 합성곱 층이 됨

- tf.nn.depthwise_conv2d()

깊이방향 합성곱 층(depthwise convolutional layer)

→ 모든 필터를 개개의 입력 채널에 독립적으로 적용

| 14.10 시맨틱 분할

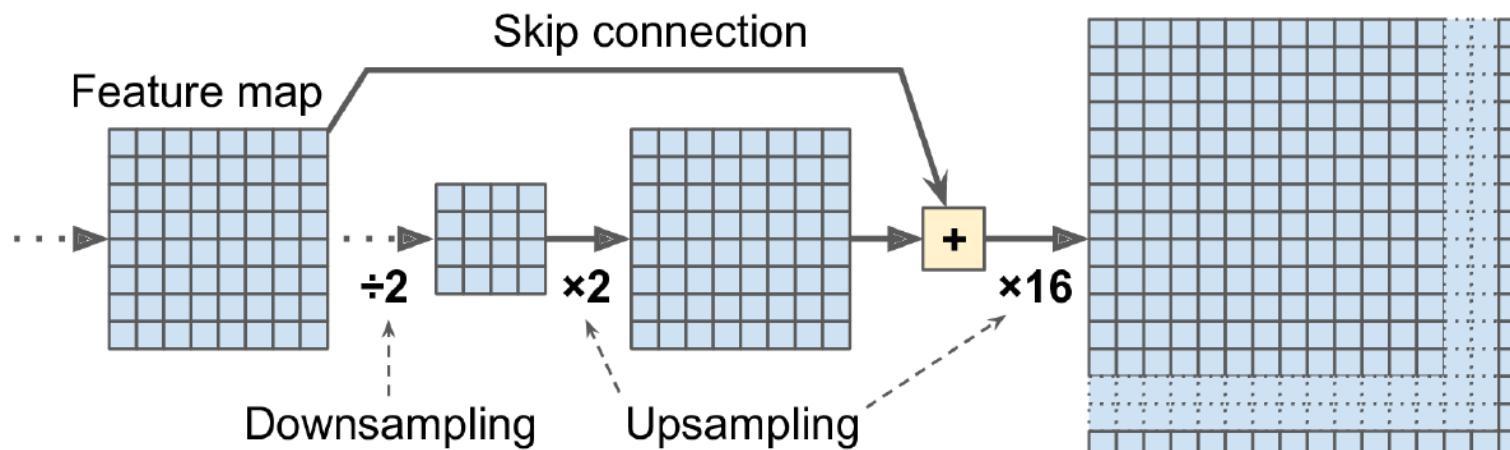
(기존의 방법들은 여전히 해상도가 떨어짐)

초해상도(super-resolution)

아래쪽 층에서부터 스킵 연결을 추가

→ 32배가 아니라 2배로 출력이미지를 업샘플링, 아래층의 출력을 더하여
해상도를 2배로 키움

→ 그다음 이것을 16배로 늘려 업샘플링 → 최종적으로 $2 \times 16 = 32$ 배





Thank You