

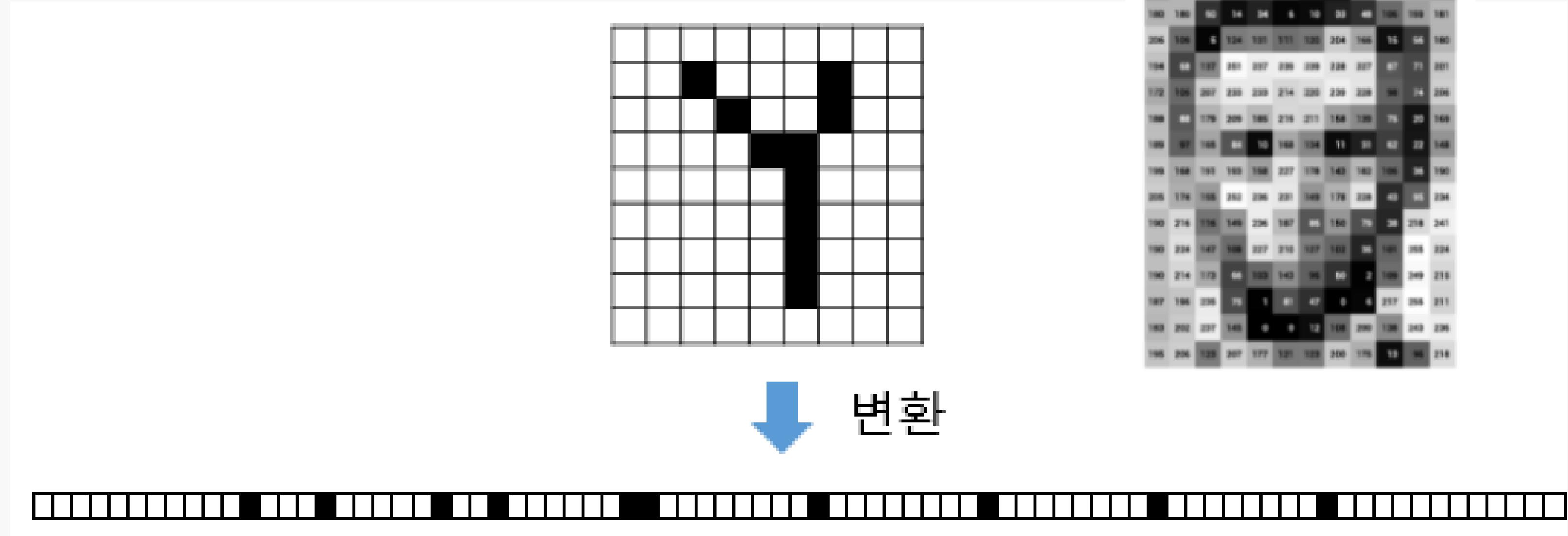
Convolutional neural network

Table of Contents

	Page
I 전체 구조	3
II 합성곱 계층	8
III 풀링 계층	30
IV 풀링 계층 구현	36

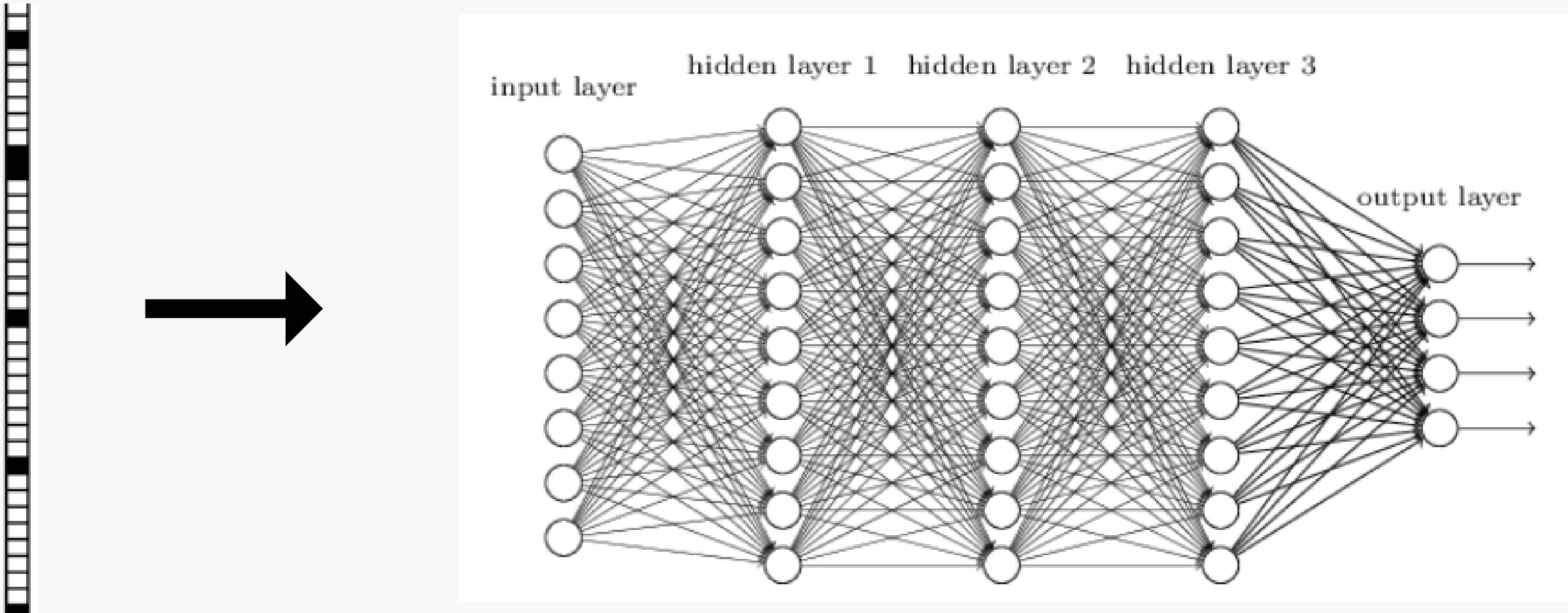
1. 전체 구조

II 완전연결 계층의 문제점



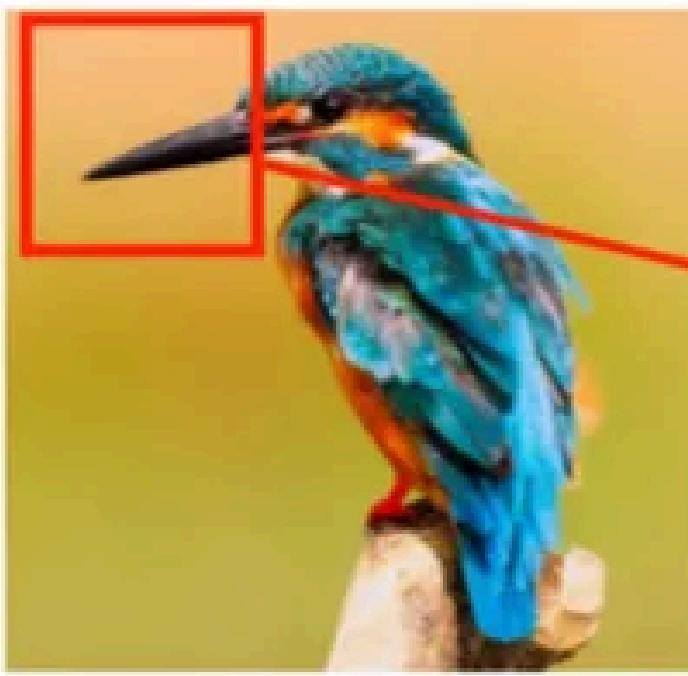
데이터의 형상 무시
공간적인 구조 정보 유실

II 완전연결 계층의 문제점

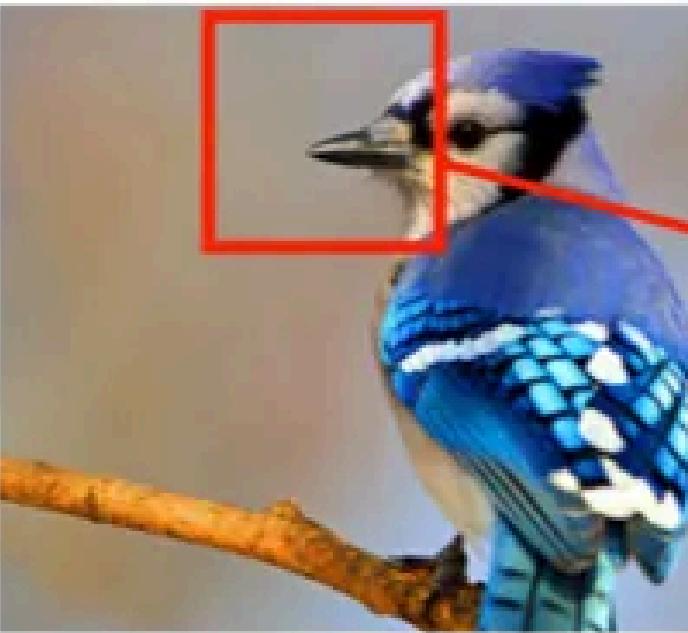
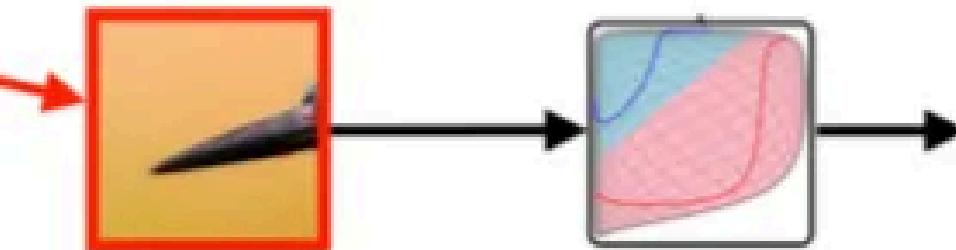


데이터의 형상 무시
공간적인 구조 정보 유실

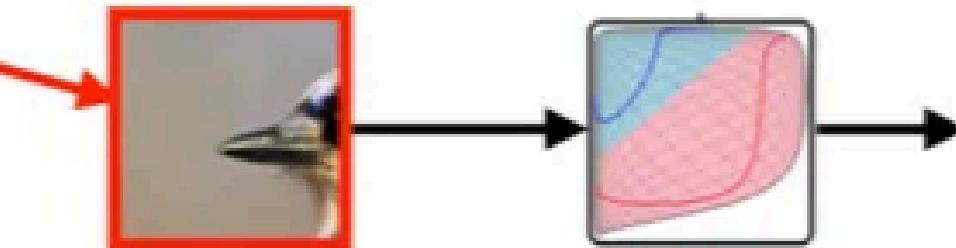
I CNN



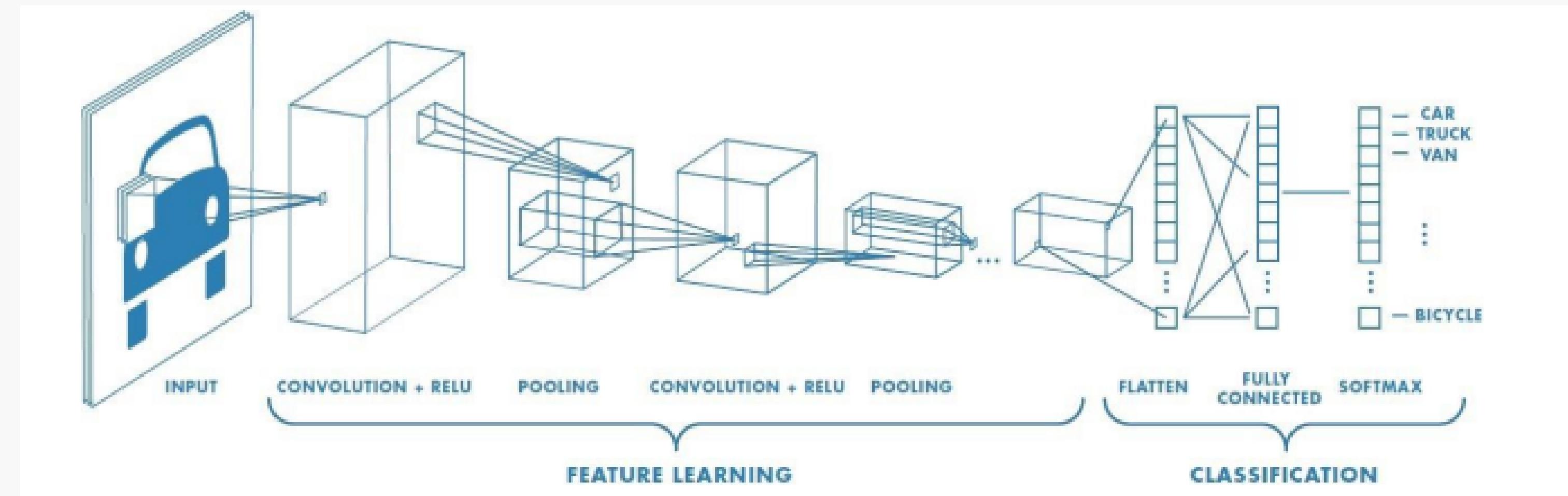
Upper left beak detector



Middle left beak detector



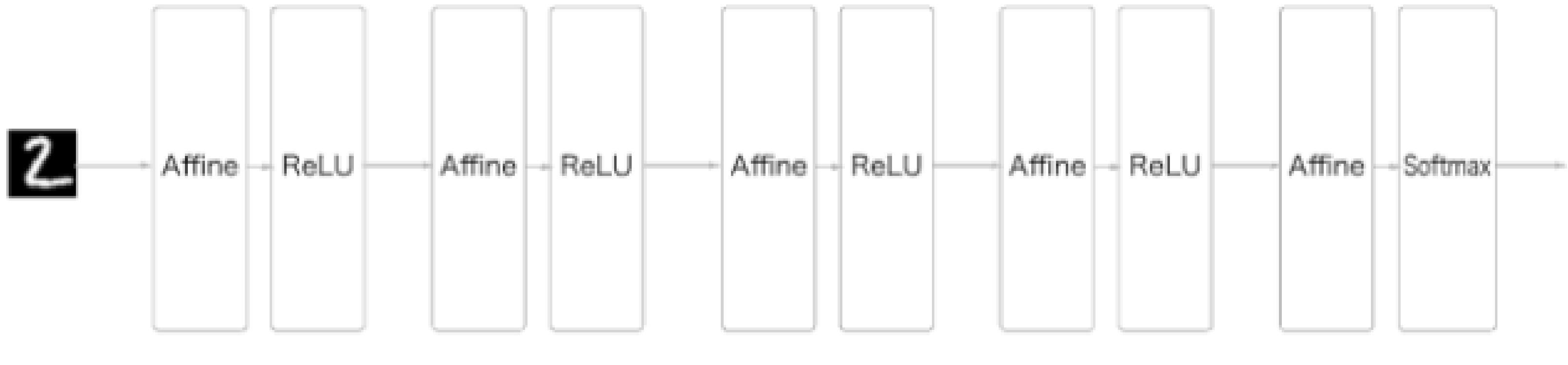
I CNN



Feature Learning : 실제로 사진을 학습하는 과정, 사진의 특징을 추출
Classification : Feature Learning의 결과를 도출

I 완전연결신경망

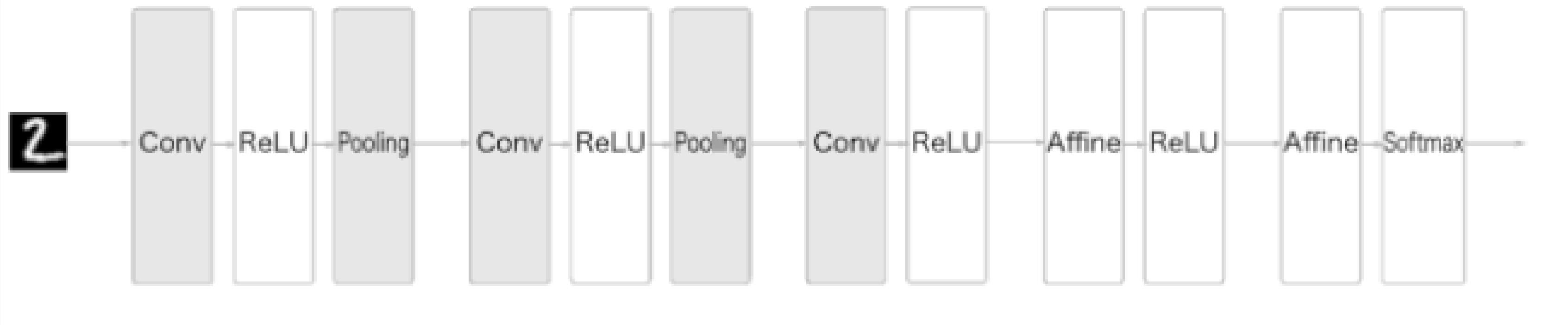
그림 7-1 완전연결 계층(Affine 계층)으로 이루어진 네트워크의 예



- Affine 계층(완전연결계층) : 인접하는 계층의 모든 뉴런과 결합
- ReLU 계층 : 활성화 함수
- Softmax 계층 : 최종 결과 출력

I CNN

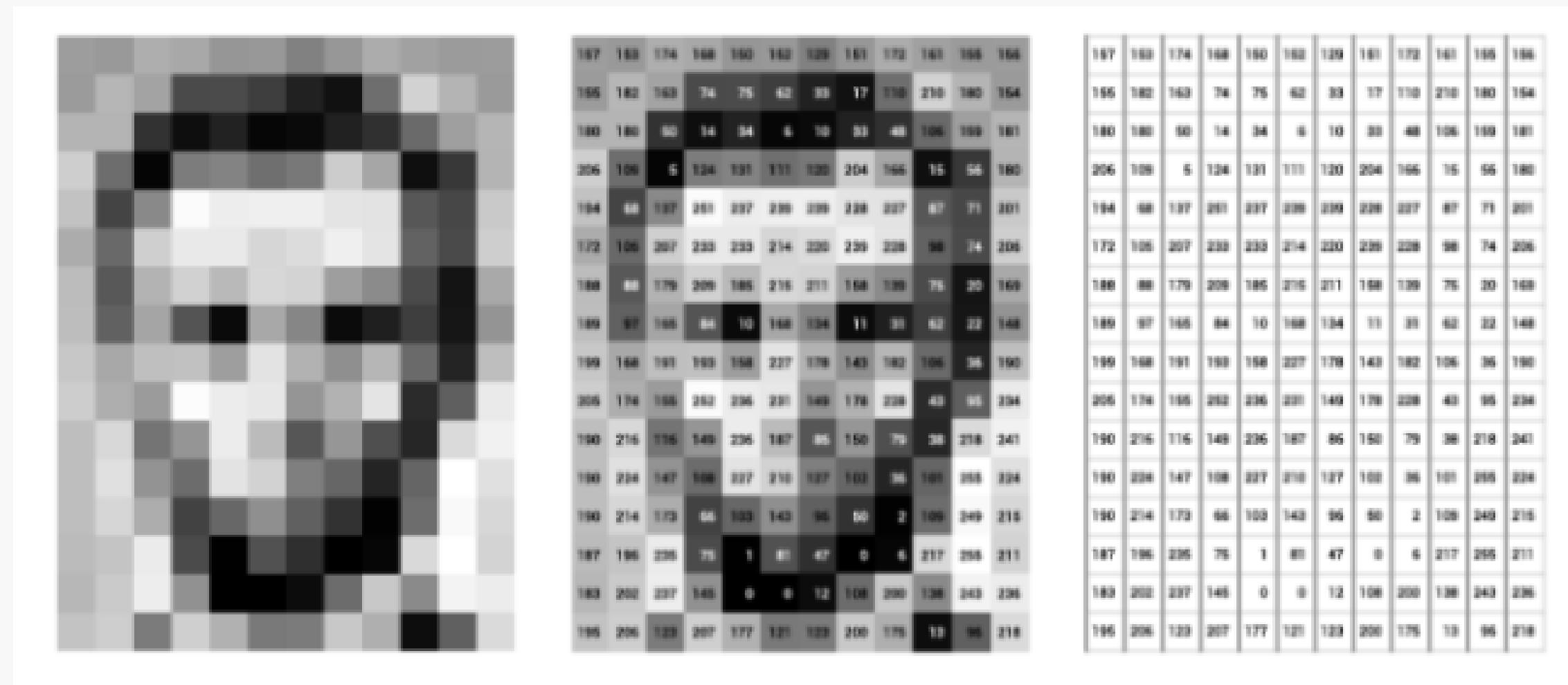
그림 7-2 CNN으로 이뤄진 네트워크의 예 : 합성곱 계층과 풀링 계층이 새로 추가(회색)



- 합성곱계층, 풀링계층 추가
- Conv - ReLU - Pooling 연결
- 출력과 가까운 층에서는 완전연결신경망과 유사

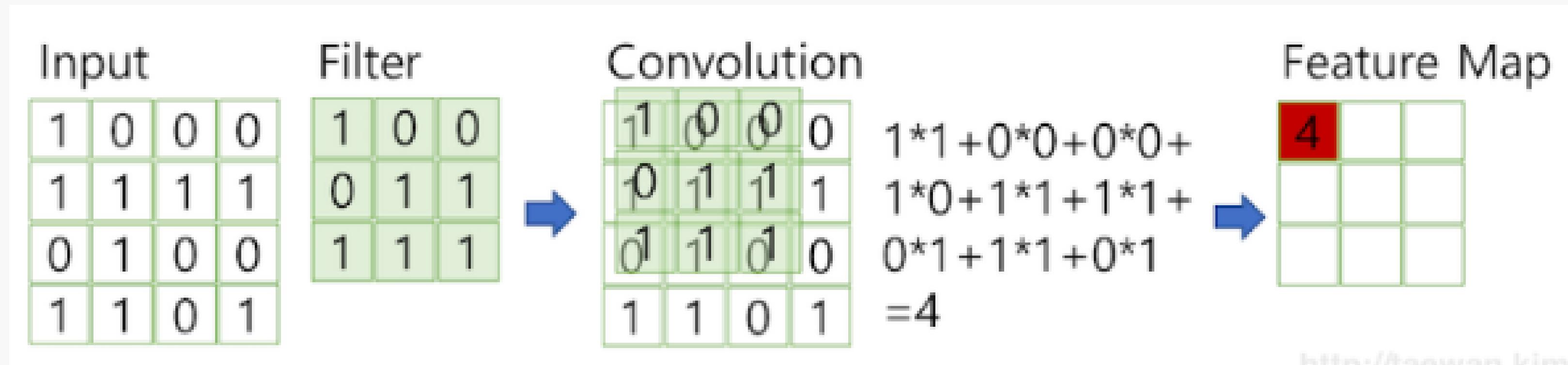
2. 합성곱 계층

II 합성곱계층



피처맵(feature map) : 합성곱계층의 입출력 데이터

II 합성곱계층

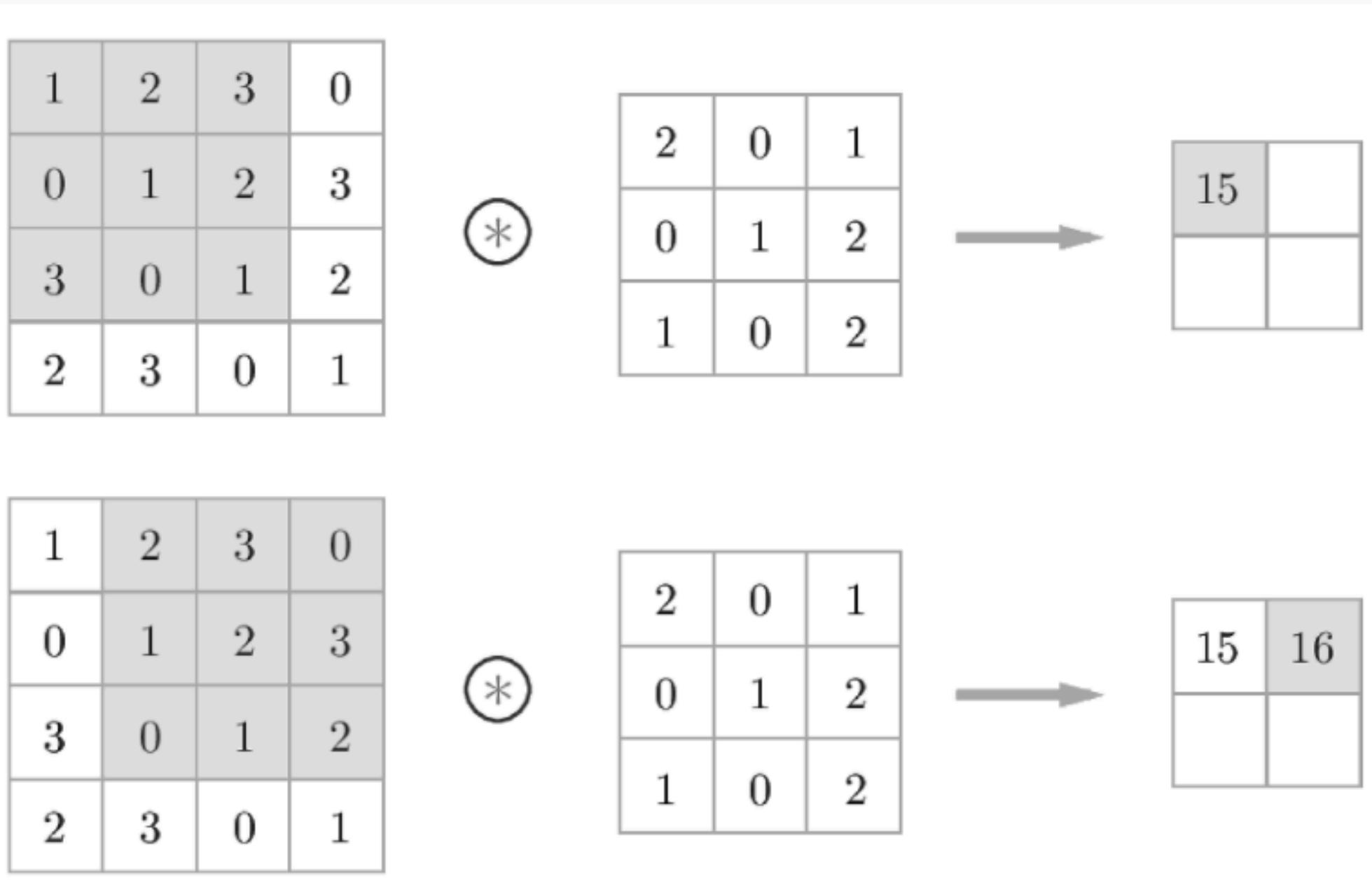


<https://krasnov.com/>

합성곱계층은 데이터의 형상을 유지

ex) 2차원 입력 \rightarrow 2차원 출력

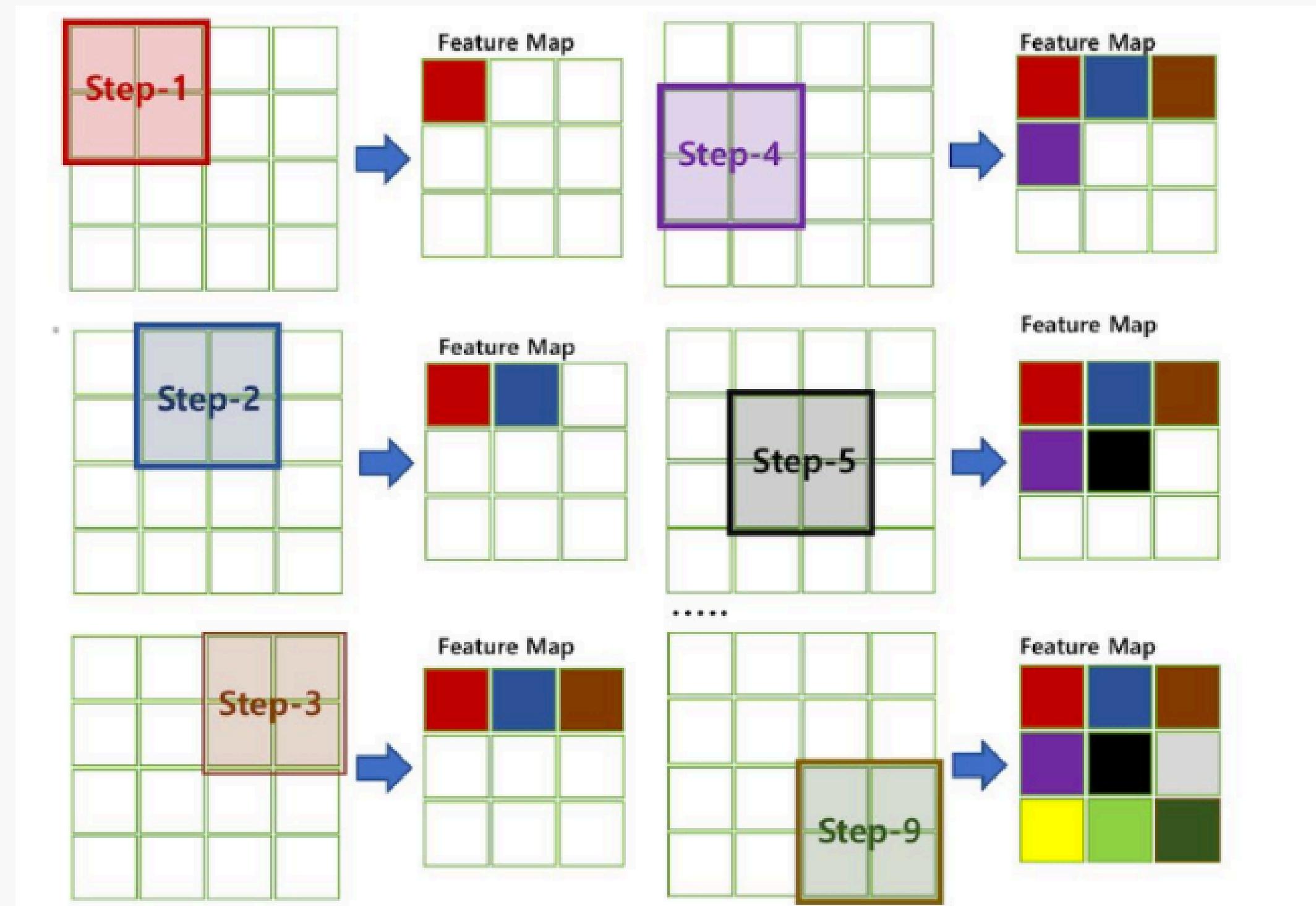
II 합성곱 연산



필터의 윈도우(회색 3×3 부분)를 일정간격으로 이동해가며 입력 데이터에 적용

입력과 필터에서 대응하는 원소끼리 곱한 후 총합을 피처맵으로 출력

II 합성곱 연산

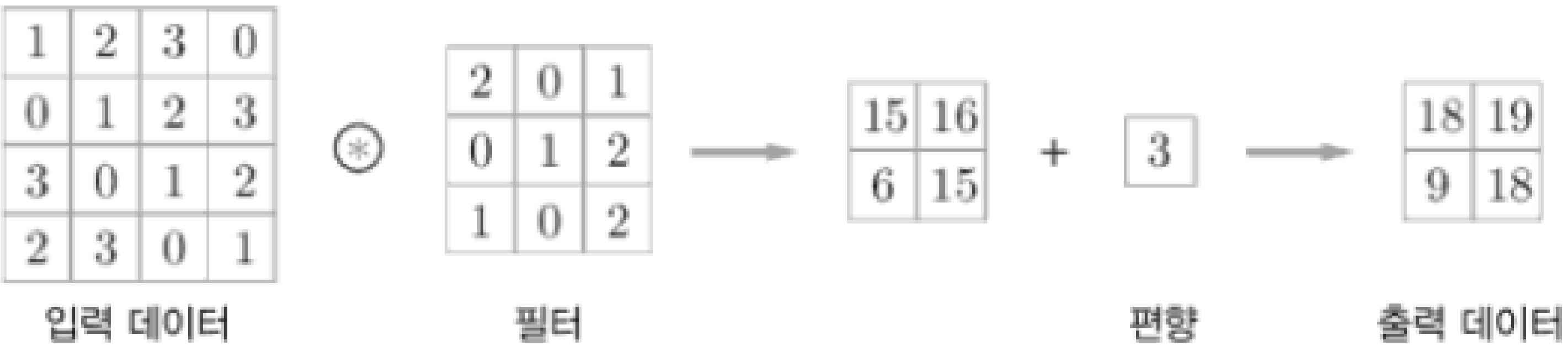


필터

	Convolution output	Test against threshold
	 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9	 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9
	 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9	 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9
	 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9	 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9
Kernels	 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9	 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9

II 가중치

그림 7-5 합성곱 연산의 편향 : 필터를 적용한 원소에 고정값(편향)을 더한다.



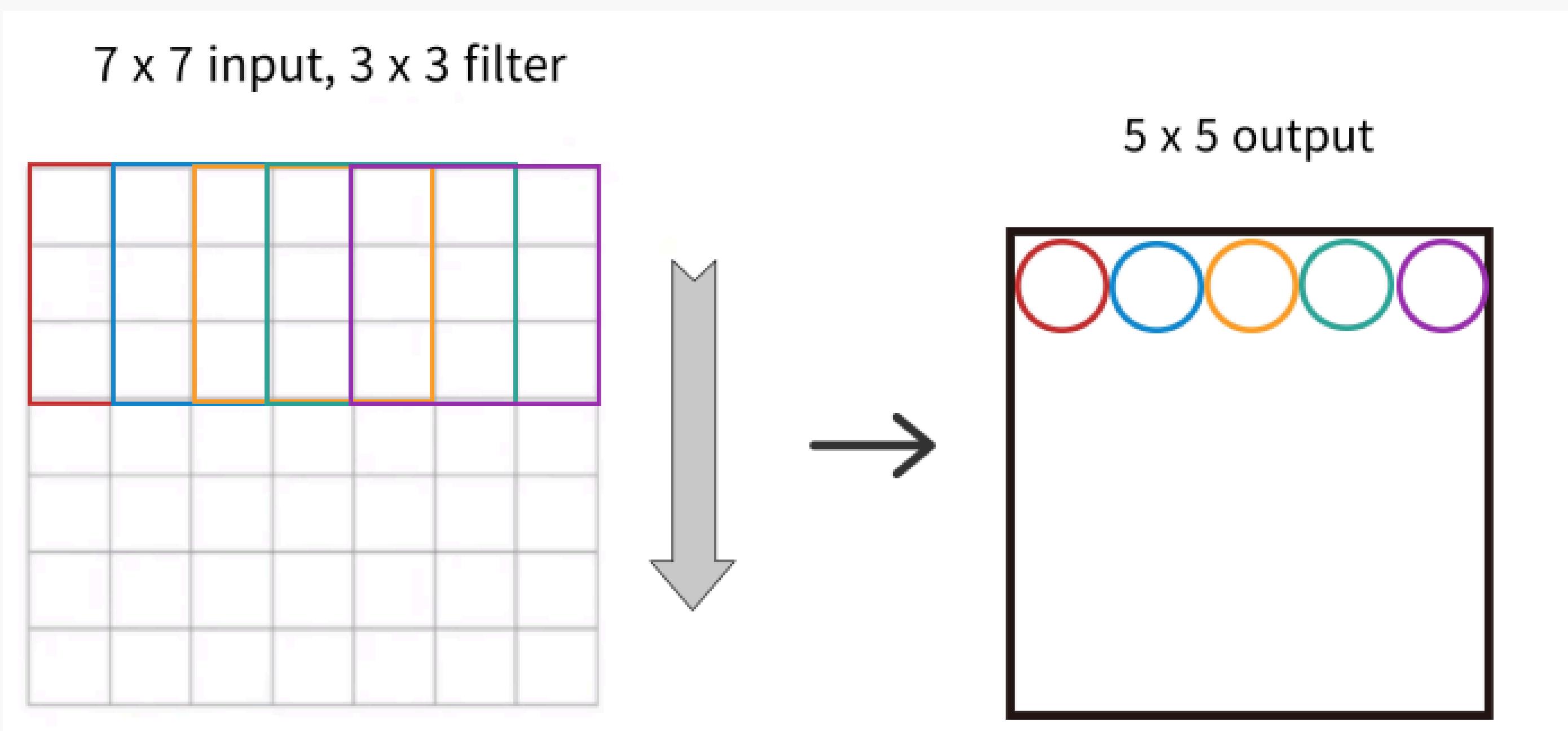
필터 : CNN에서의 학습 파라미터, 가중치
편향 : 하나(1×1)만 존재, 모든 원소에 더함

학습을 통해 필터와 편향 조정

II 스트라이드 & 패딩

1. 만약 Input 이미지가 크기가 매우 클 경우 (e.g, 3840 X 2160), CNN을 적용하기에 너무 계산량이 많다. 크기가 큰 이미지를 다루는 방법?
2. 필터의 크기가 클 경우, 계속 필터를 적용하다 보면 activation map의 크기가 줄어들 텐데, 크기 감소를 피하는 방법?

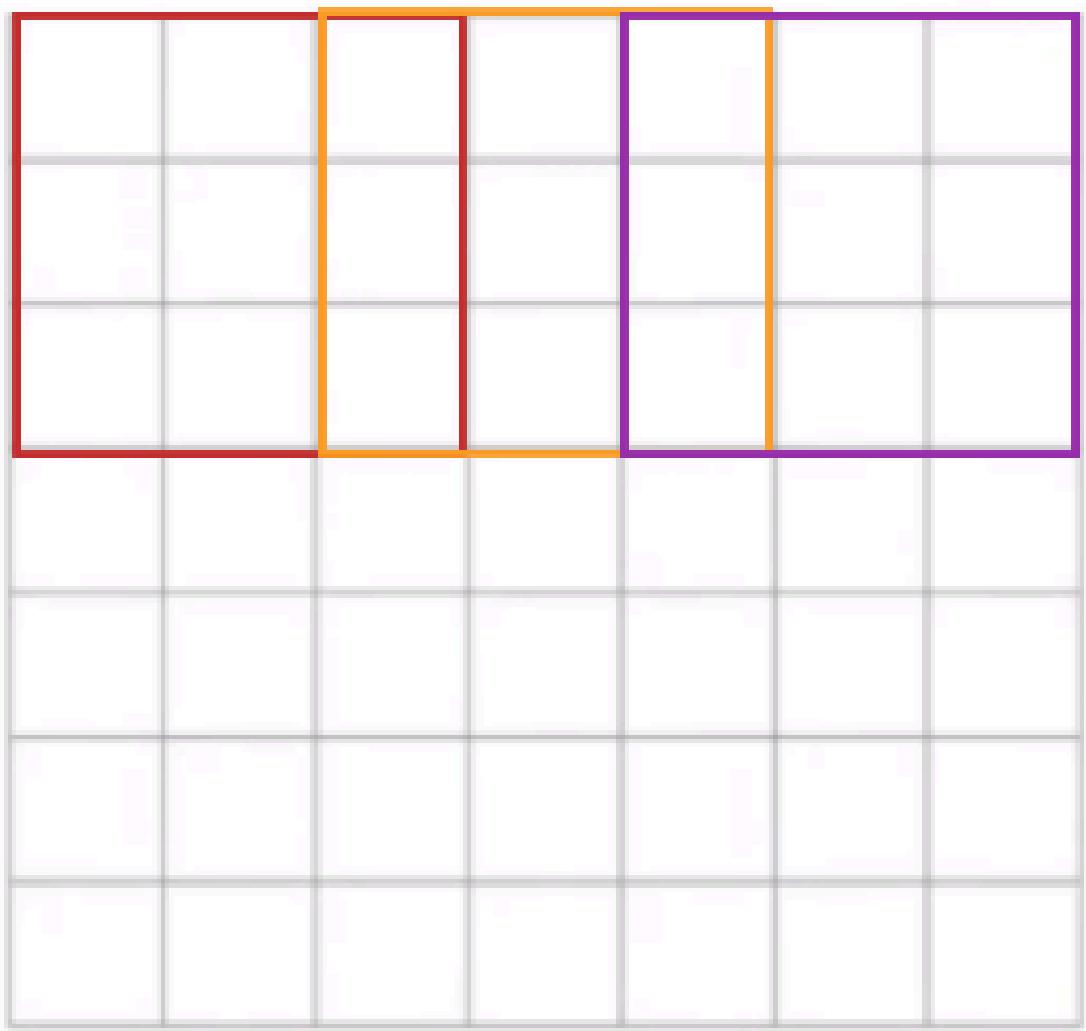
II 스트라이드 (Stride)



스트라이드 : 필터를 적용하는 위치의 간격

II 스트라이드

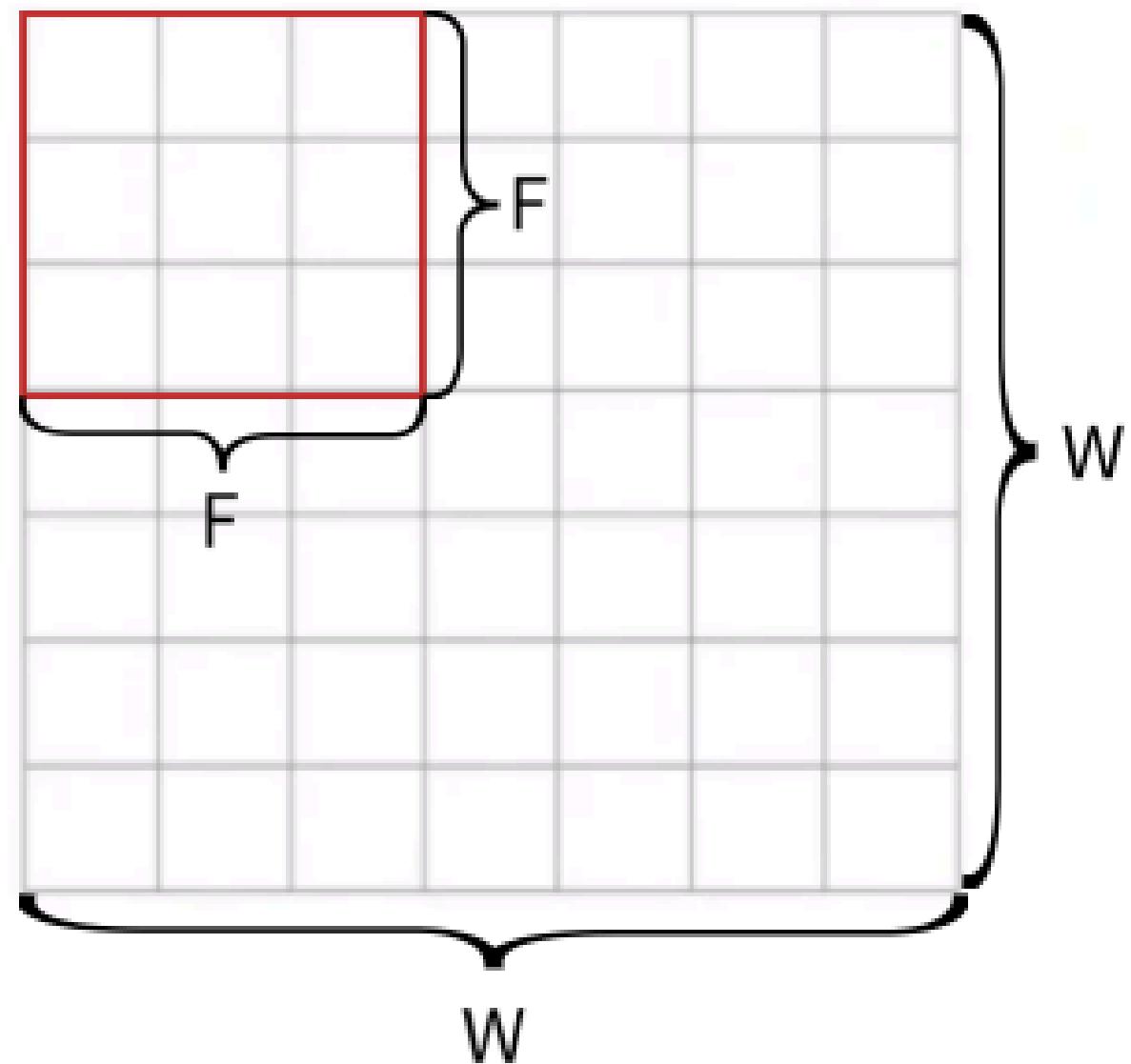
7 x 7 input, 3 x 3 filter, stride 2



3 x 3 output

II 스트라이드

7 x 7 input, 3 x 3 filter



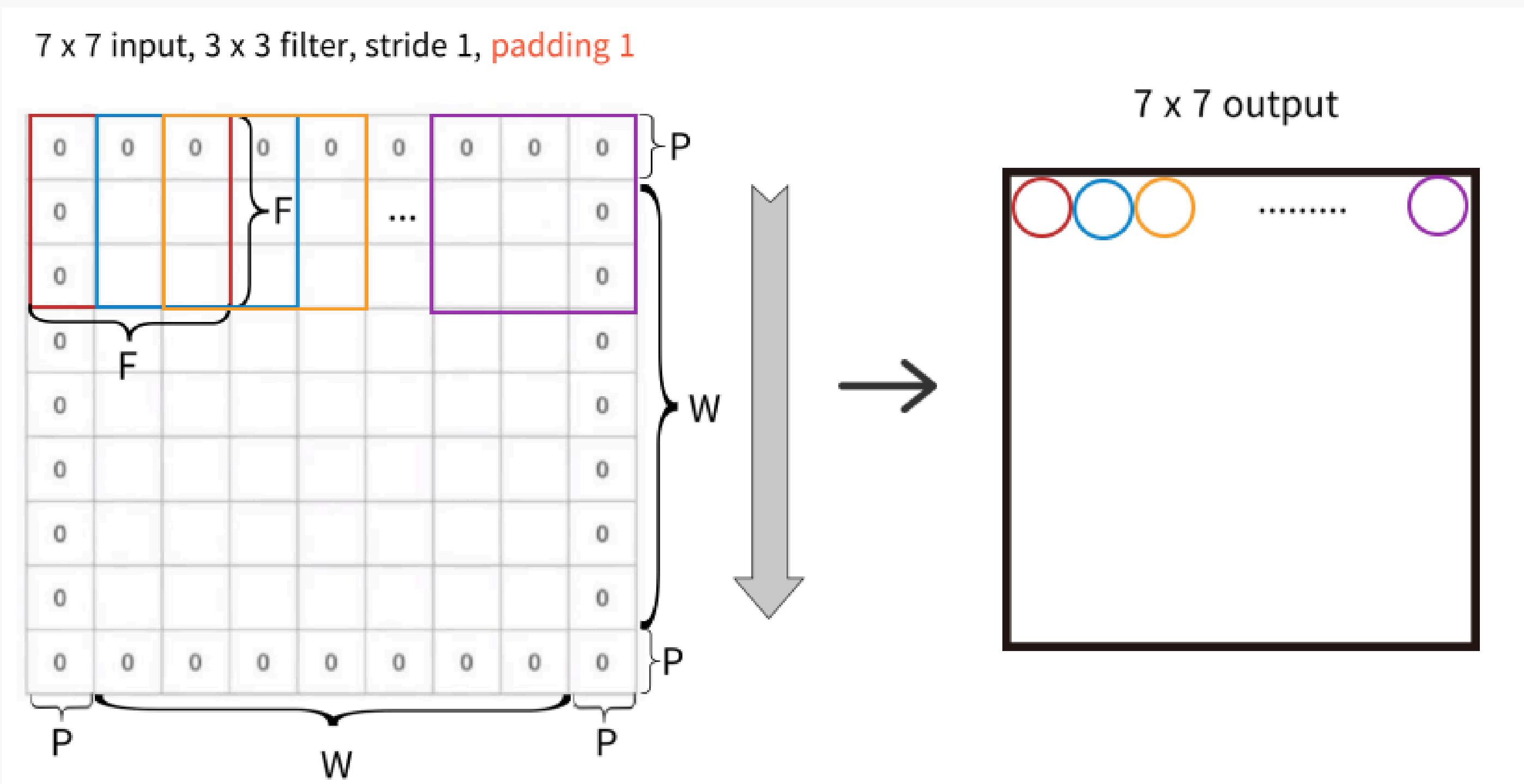
Output Size(W'):

$$W' = (W-F) / S + 1 \quad (W : \text{Input} , F: \text{Filter} , S: \text{Stride})$$

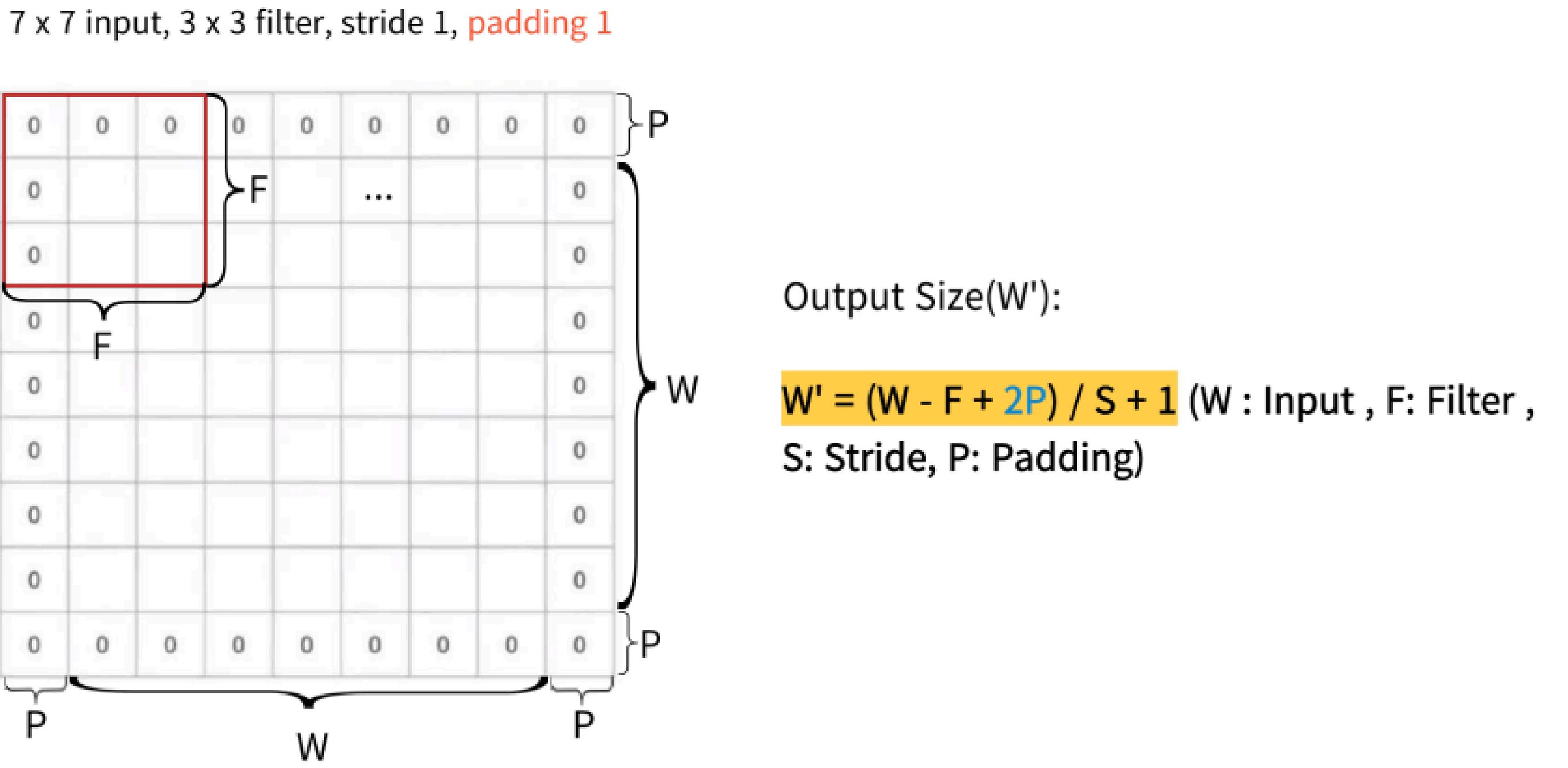
- Stride 1: $W' = (7 - 3) / 1 + 1 = 5$
- Stride 2: $W' = (7 - 3) / 2 + 1 = 3$
- Stride 3: $W' = (7 - 3) / 3 + 1 = 2.33$

출력 크기가 정수이어야 함에 주의

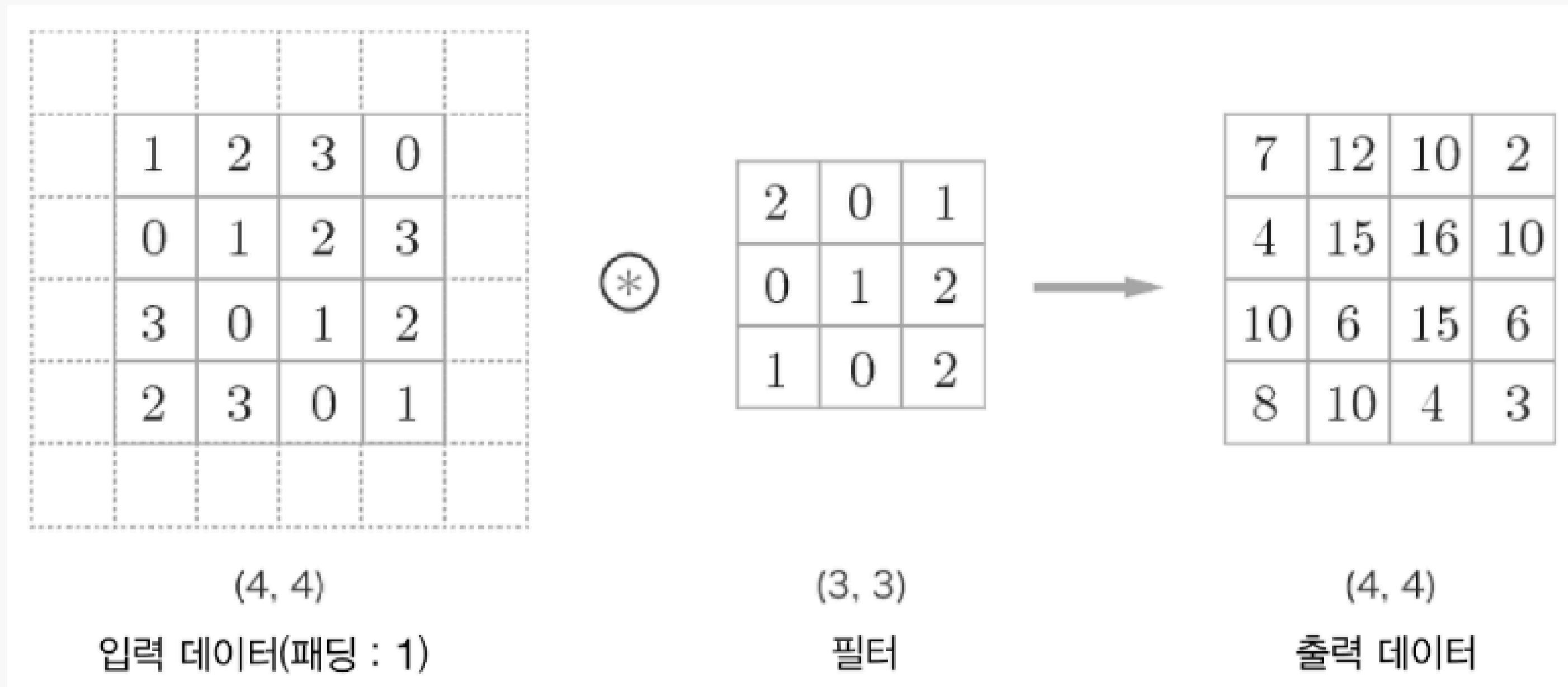
|| 패딩(Padding)



II 패딩(Padding)

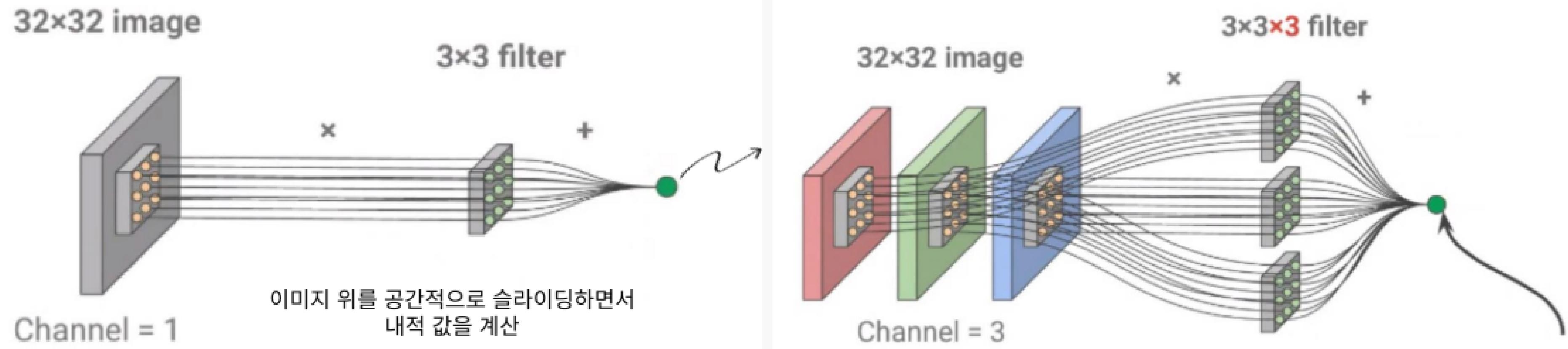


II 패딩(Padding)



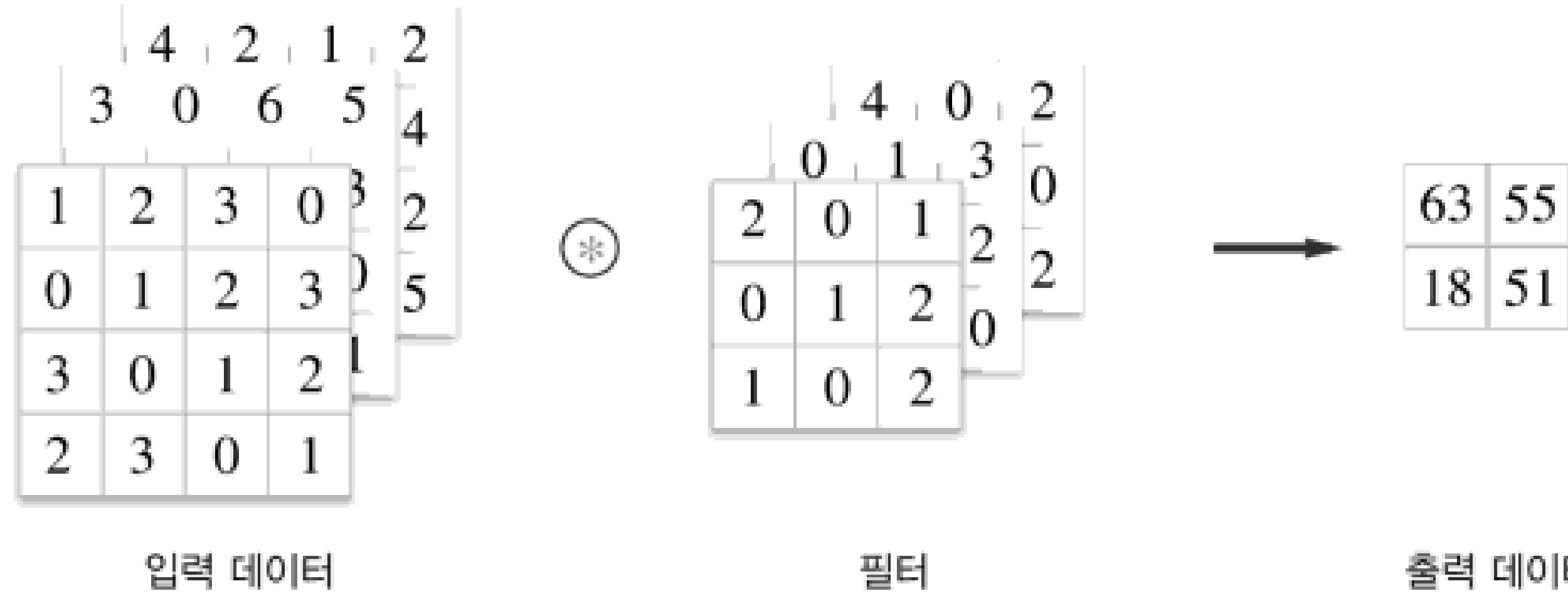
주로 출력 데이터의 크기를 조정할 목적으로 사용
공간적 크기를 고정한 채로 다음 계층 전달

II 3차원 데이터의 합성곱 연산



II 3차원 데이터의 합성곱 연산

그림 7-8 3차원 데이터 합성곱 연산의 예



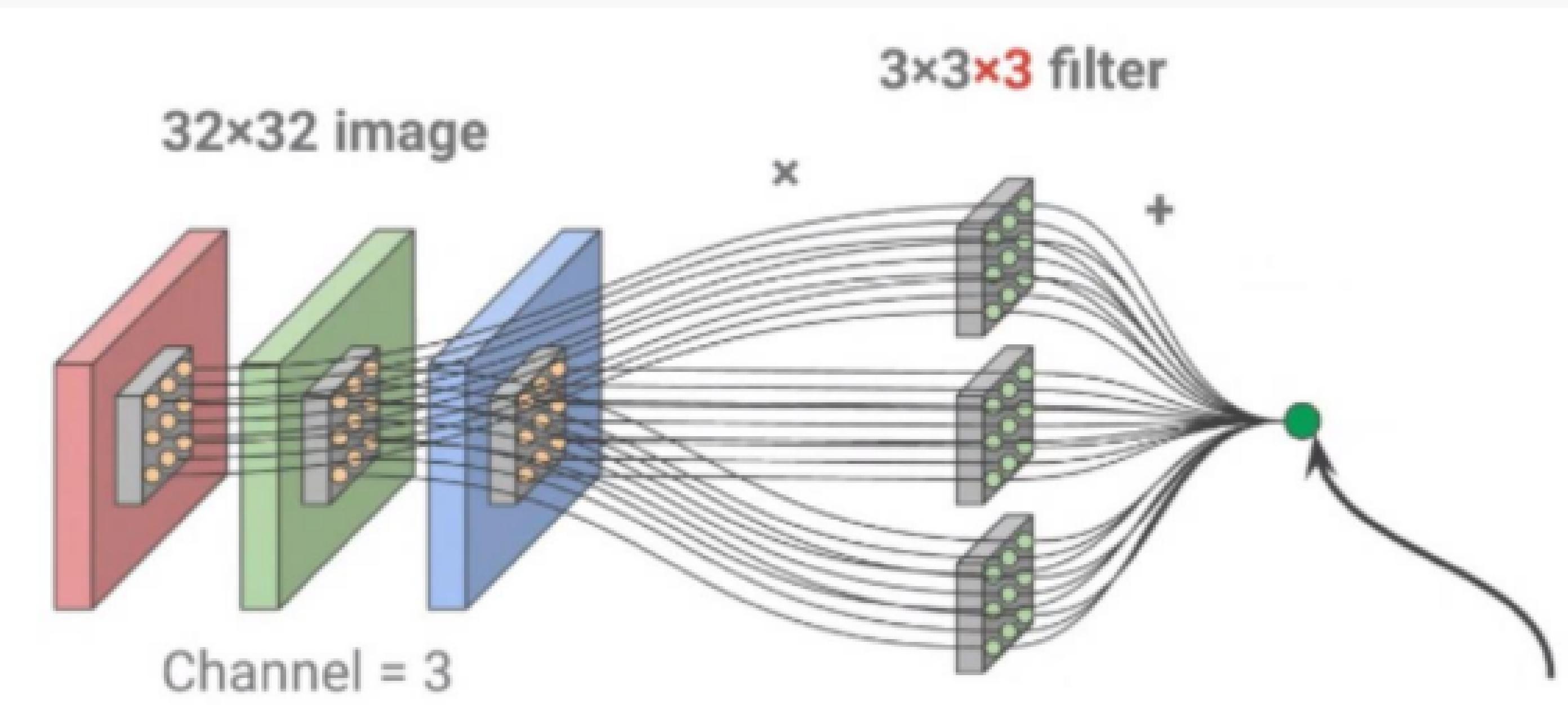
입력 데이터

필터

출력 데이터

채널마다 합성곱연산 수행 후 더하여 하나의 출력 얻음
입력데이터의 채널 수 = 필터의 채널 수

II 3차원 데이터의 합성곱 연산



학습 파라미터 수 (필터, 편향) = Quiz 2

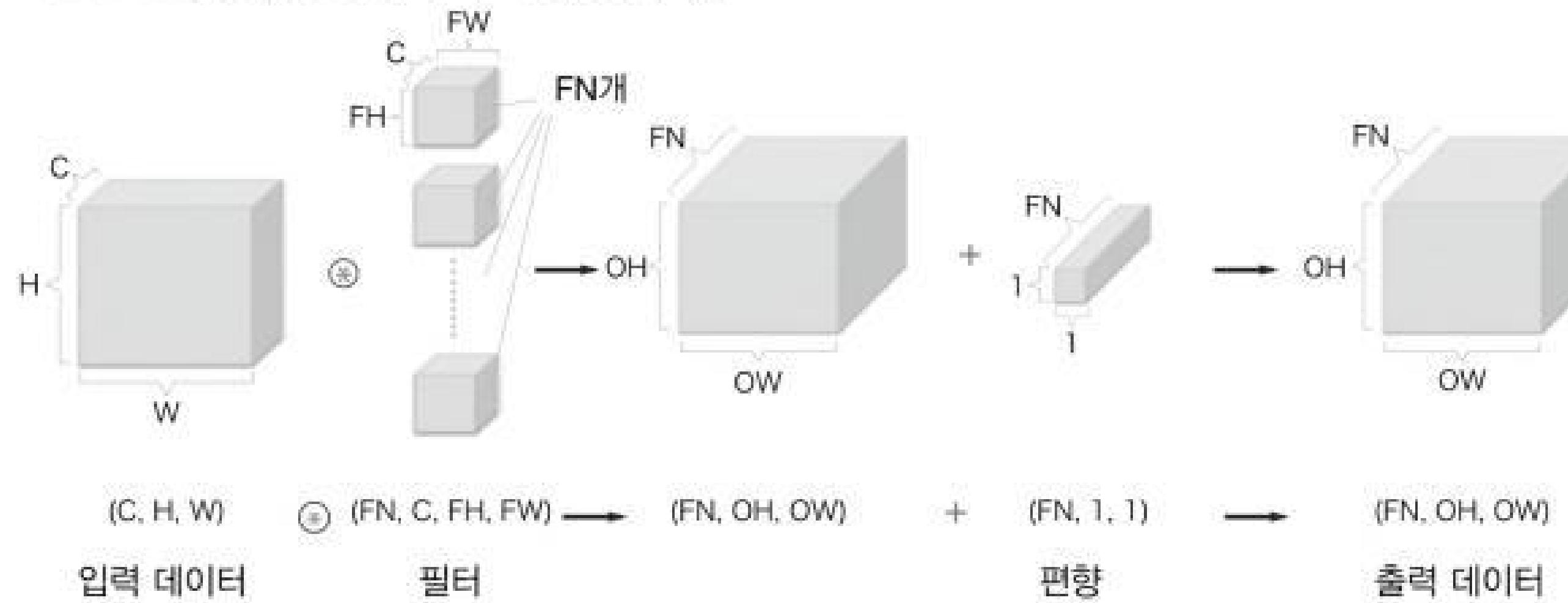
II 패딩(Padding)

Quiz 1.

32 * 32 input 이미지에 2개의 5*5*3 필터를 적용할 경우,
출력 피처맵의 사이즈는?

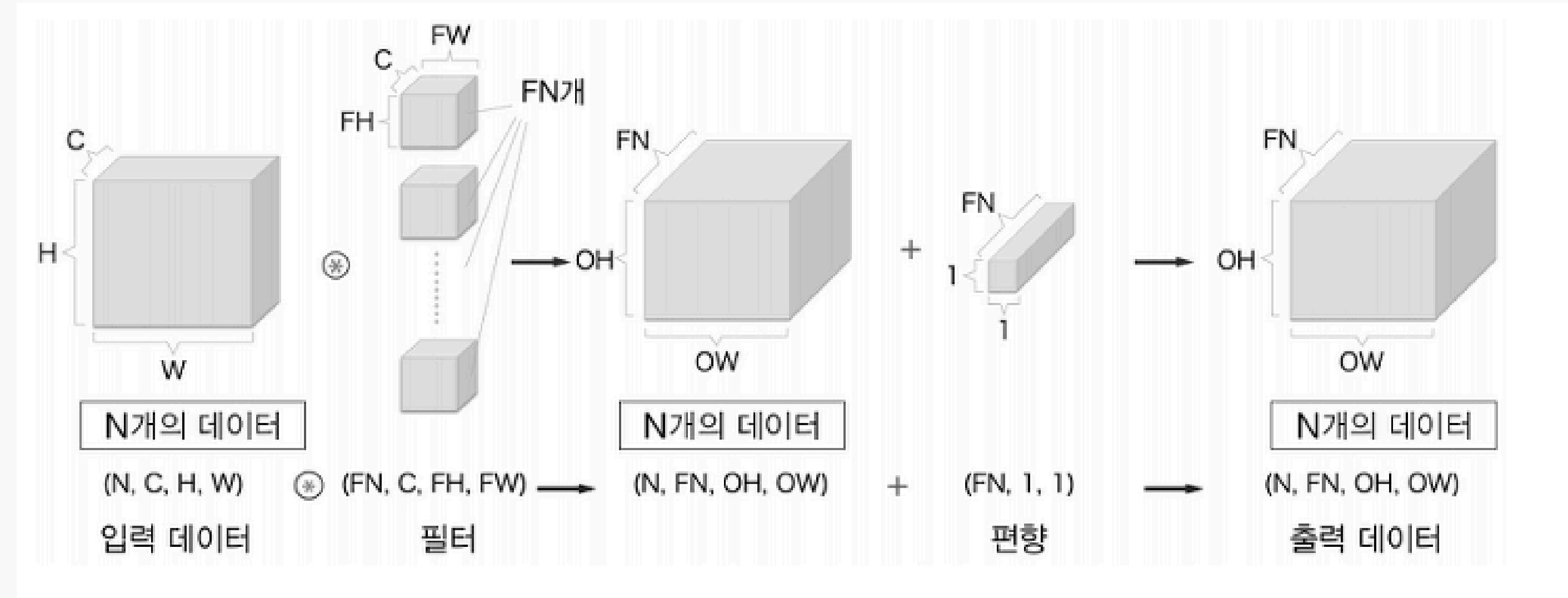
II 다수의 채널 출력

그림 7-12 합성곱 연산의 처리 흐름(편향 추가)



필터를 FN개 적용하면 FN개의 출력 II처맵 생성
편향은 채널 하나에 값 하나로 구성

II 배치 처리

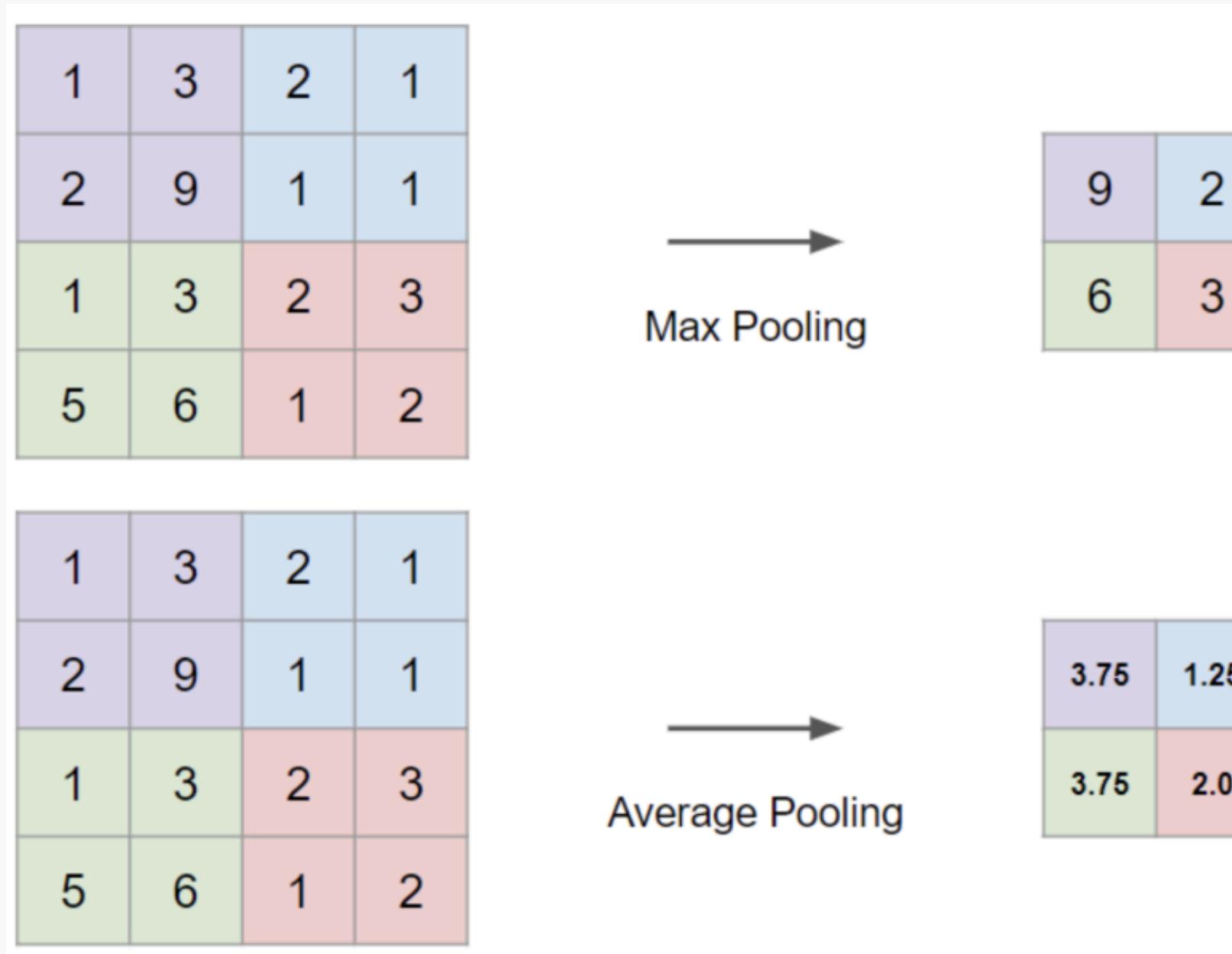


각 계층을 흐르는 데이터의 차원을 하나 늘려 4차원으로 저장
ex) 하나의 배치 당 N개의 데이터로 구성되는 경우

3. 풀링 계층

III 풀링

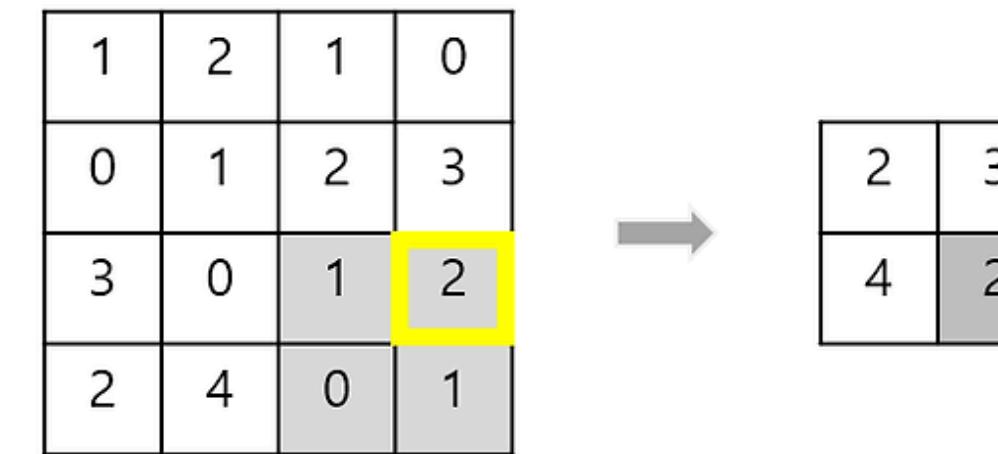
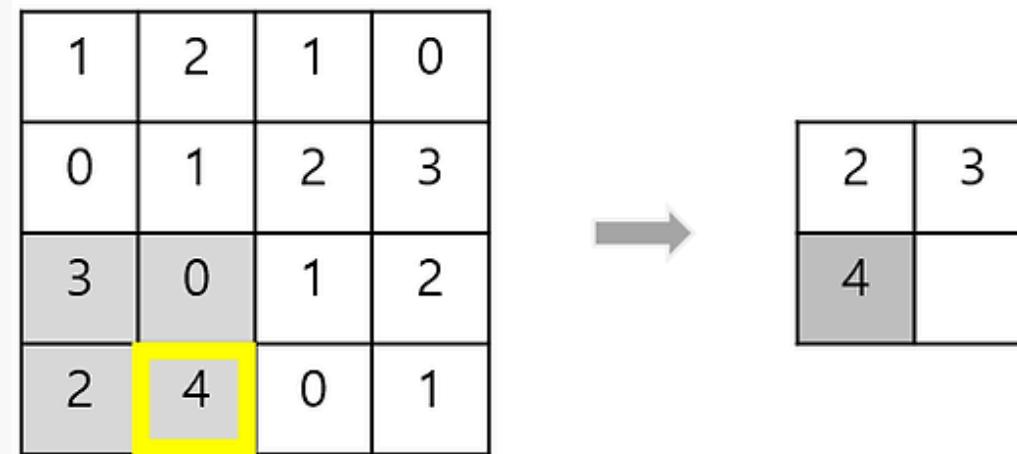
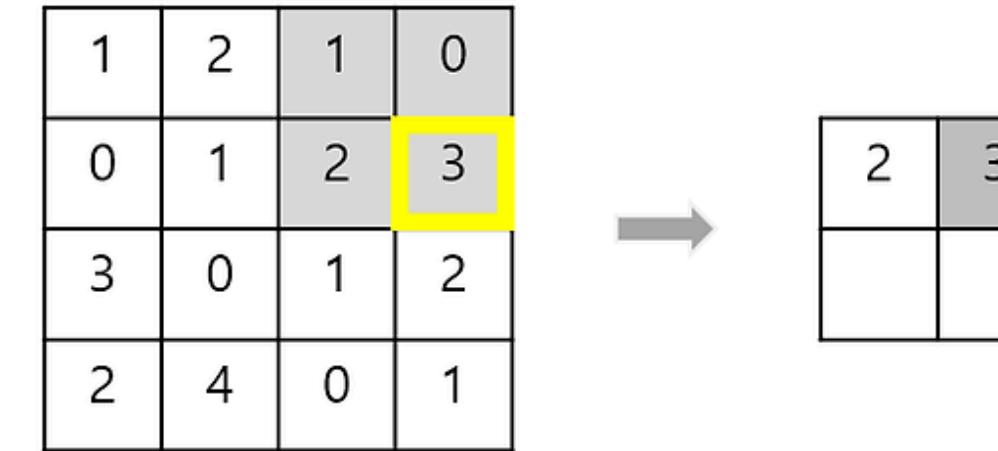
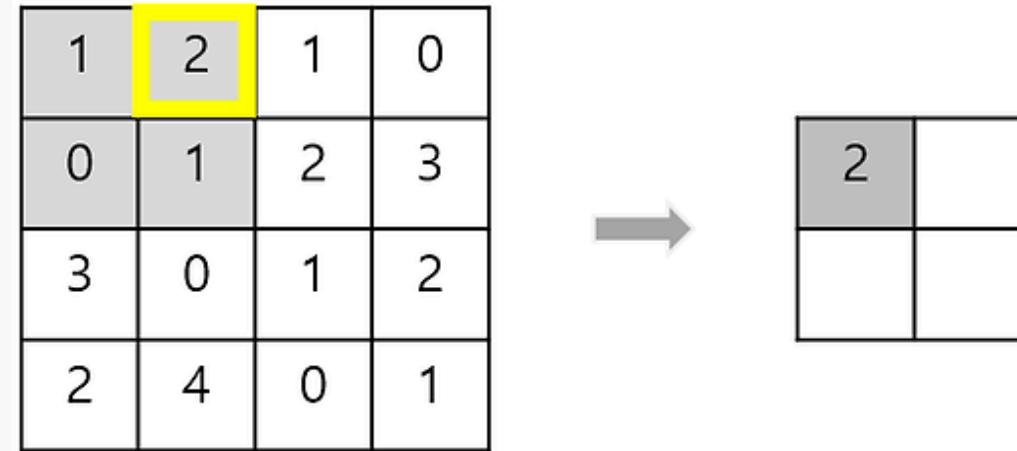
세로·가로 방향의 공간을 줄이는 연산



nxn Max Pooling

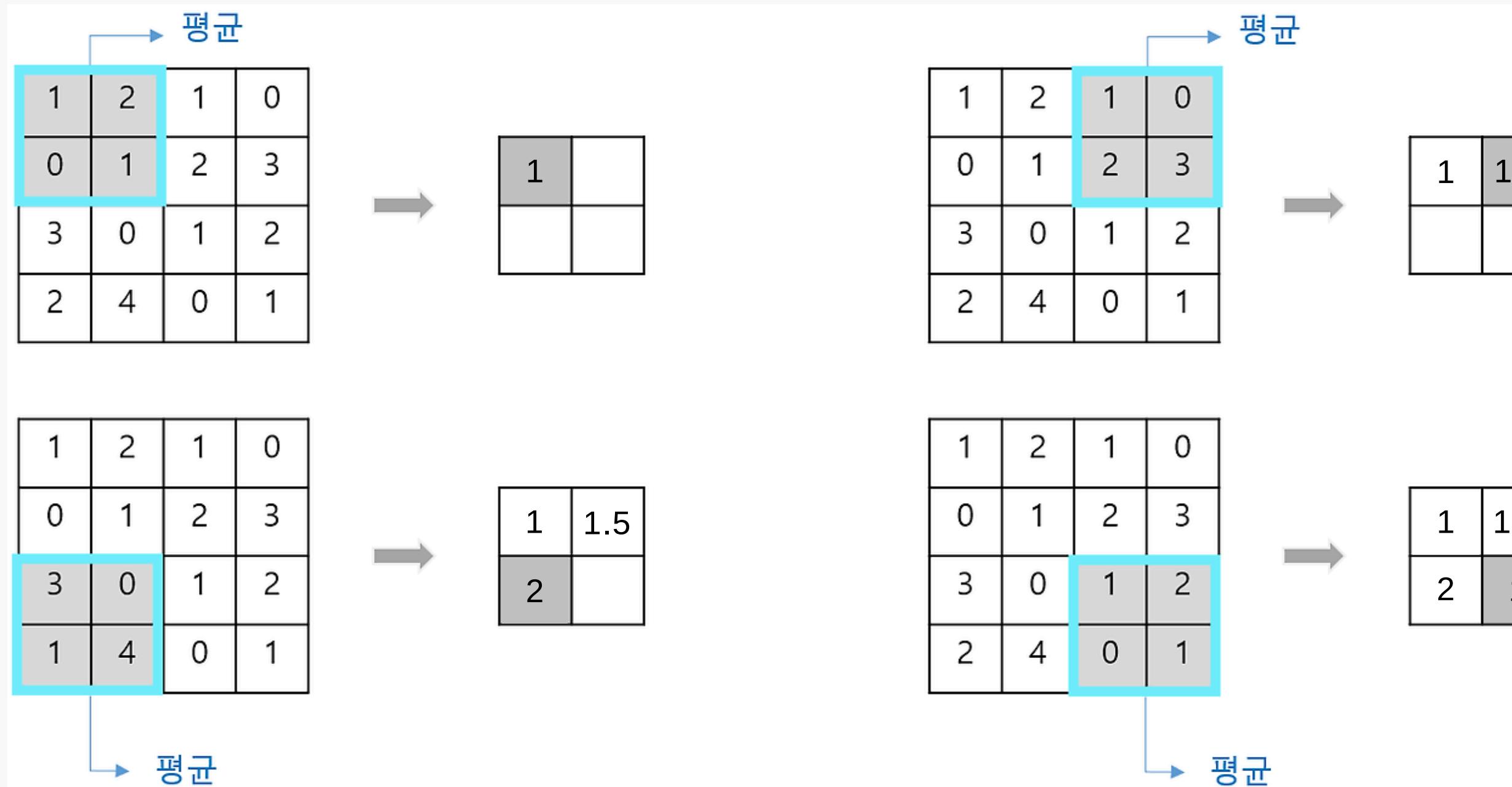
nxn Average Pooling

III Max Pooling



- $n \times n$ 크기에서 **가장 큰 값**을 선택하여 특성을 요약한다.
- 보편적으로 사용하는 방법이다.
- 중요한 정보를 강조하고, 특징을 뚜렷하게 추출할 수 있다.

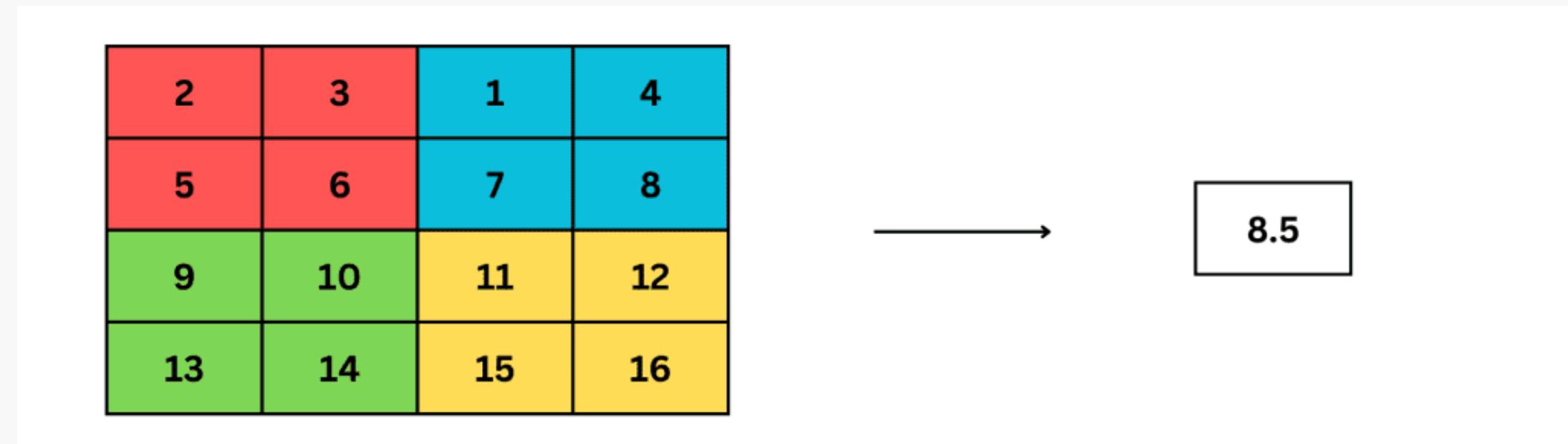
III Average Pooling



- $n \times n$ 크기에서 평균값을 추출하여 특성을 요약한다.
- LeNet에서 사용되었다.
- 평균화 과정을 통해 노이즈를 줄이고 안정적인 특징을 추출할 수 있다.

III Global Average Pooling

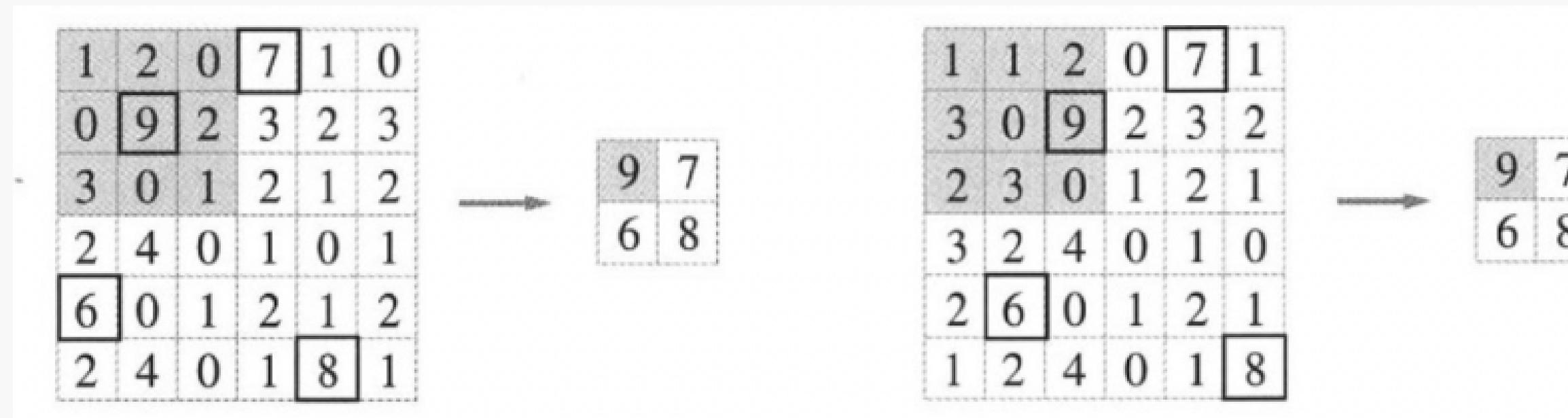
기존의 Fully Connected Layer의 단점을 극복하기 위해서 고안된 방법이다.



- Location 정보를 FC Layer보다 적게 잃는다.
- 파라미터를 차지하지 않아 계산 속도가 빠르다.
- 마찬가지로 파라미터가 많아지지 않기 때문에 오버피팅을 방지한다.
- feature map 안의 값들의 평균을 사용하기 때문에 global context 정보를 가진다.

III 풀링 계층의 특징

- 학습해야 할 매개변수가 없다
 - 대상 영역에서 최댓값이나 평균을 취하는 명확한 처리이므로 별도의 학습이 필요하지 않음
- 채널 수가 변하지 않는다
 - 채널마다 독립적으로 계산하므로 입력 데이터의 채널 수 그대로 출력 데이터로 보냄
- 입력 변화에 영향을 적게 받는다
 - 입력 데이터가 변해도 풀링의 결과는 잘 변하지 않음

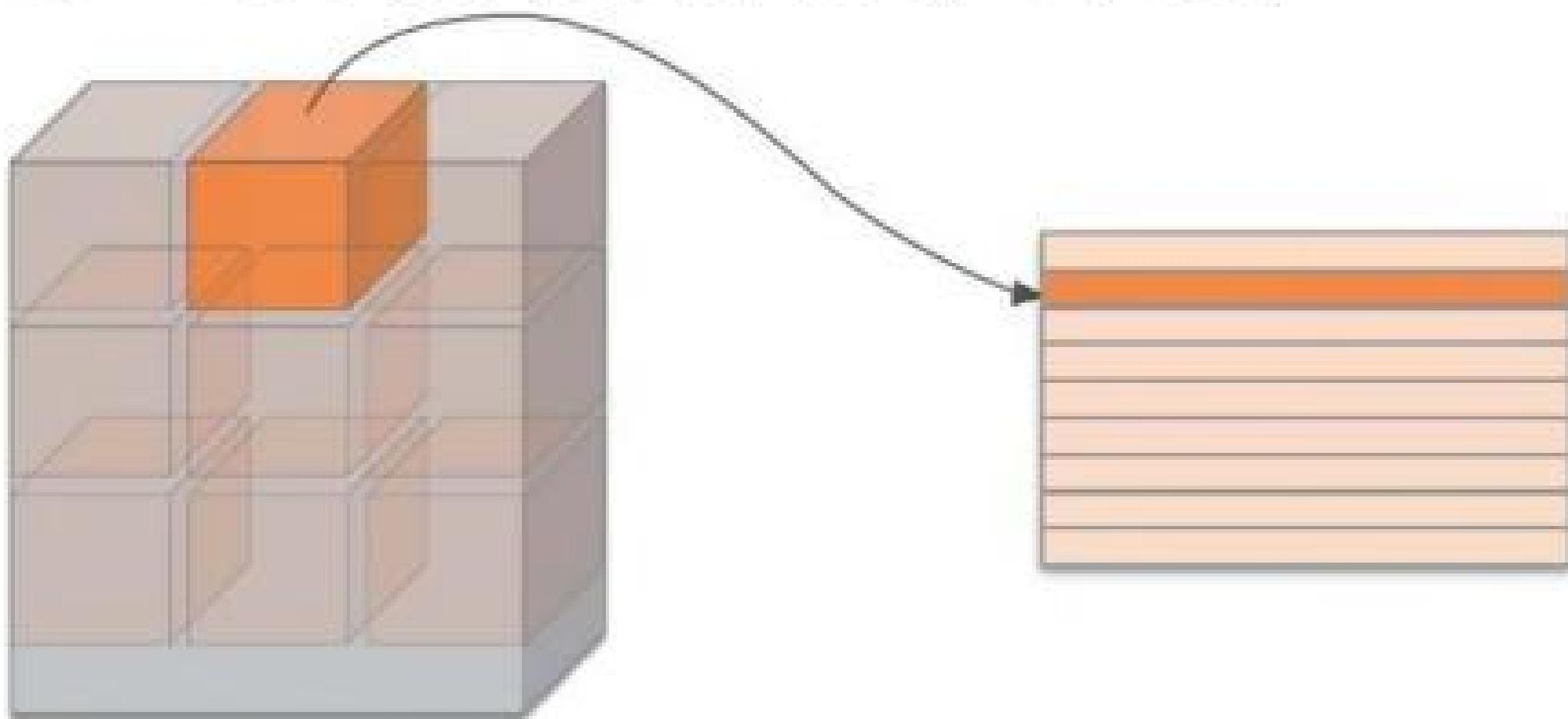


4. 블링 계층 구현

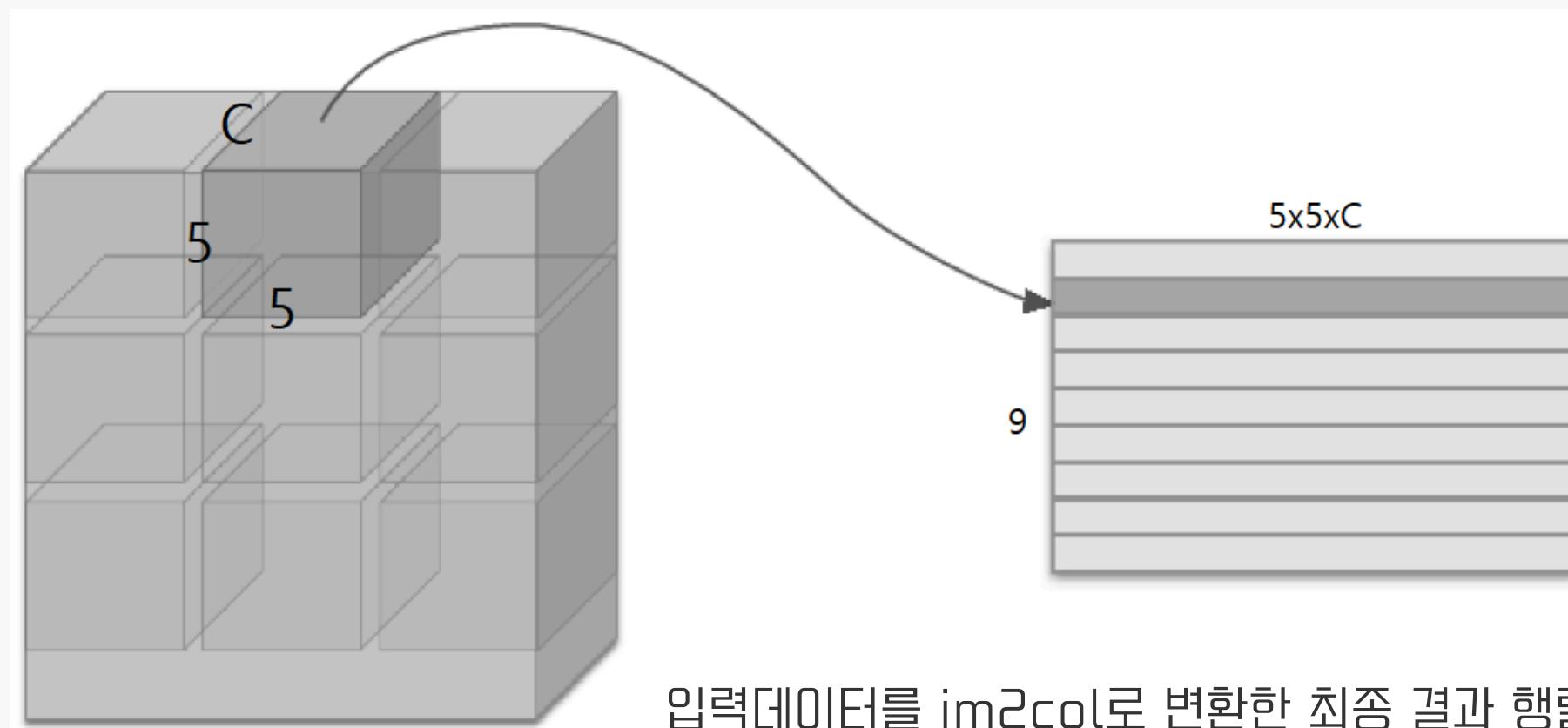
IV im2col로 데이터 전개하기

- im2col : 입력 데이터를 필터링(가중치 계산)하기 좋게 전개하는 함수
 - 3차원 행렬을 2차원 행렬로 변환
 - 3차원 행렬의 연산속도 저하 문제 방지
 - * 정확히는 배치 안의 데이터 수까지 4차원 데이터를 2차원으로 변환

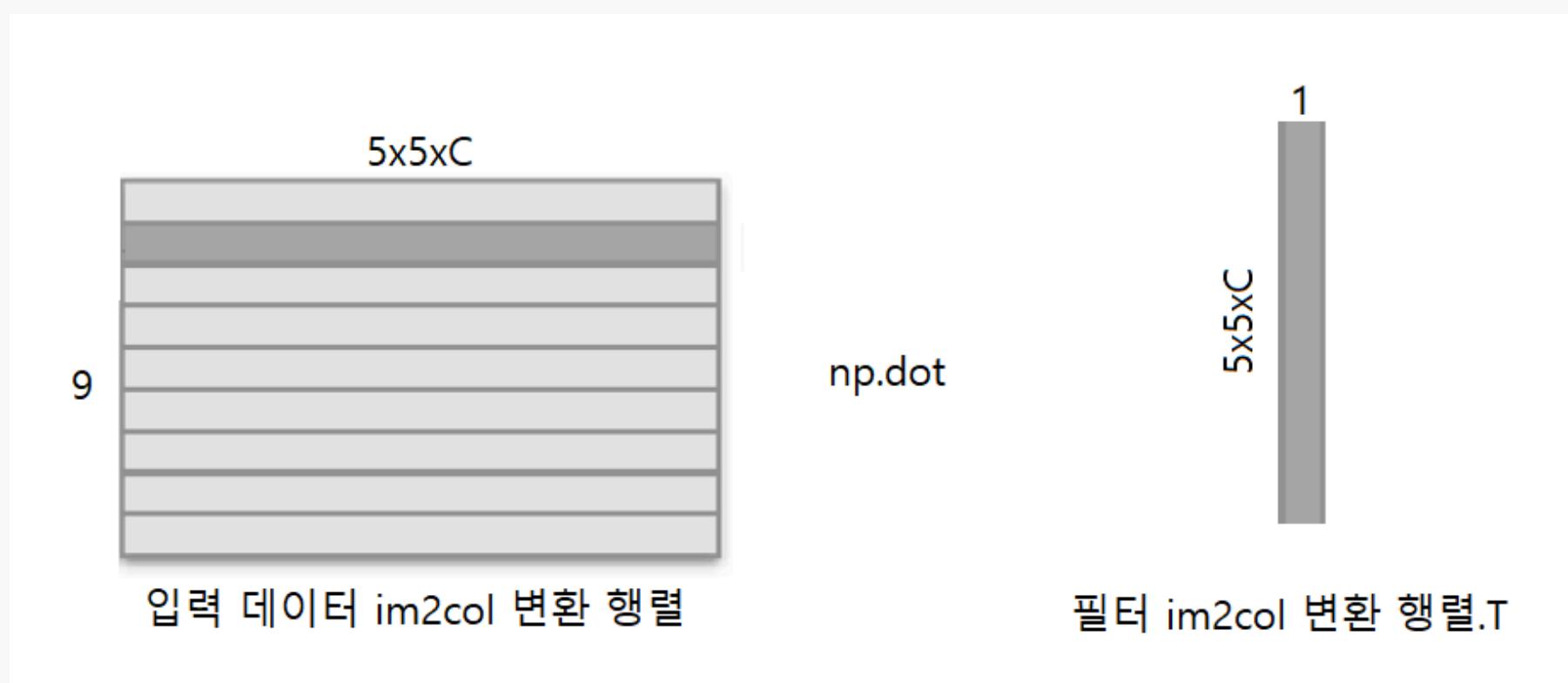
그림 7-18 필터 적용 영역을 앞에서부터 순서대로 1줄로 펼친다.



IV im2col로 데이터 전개하기

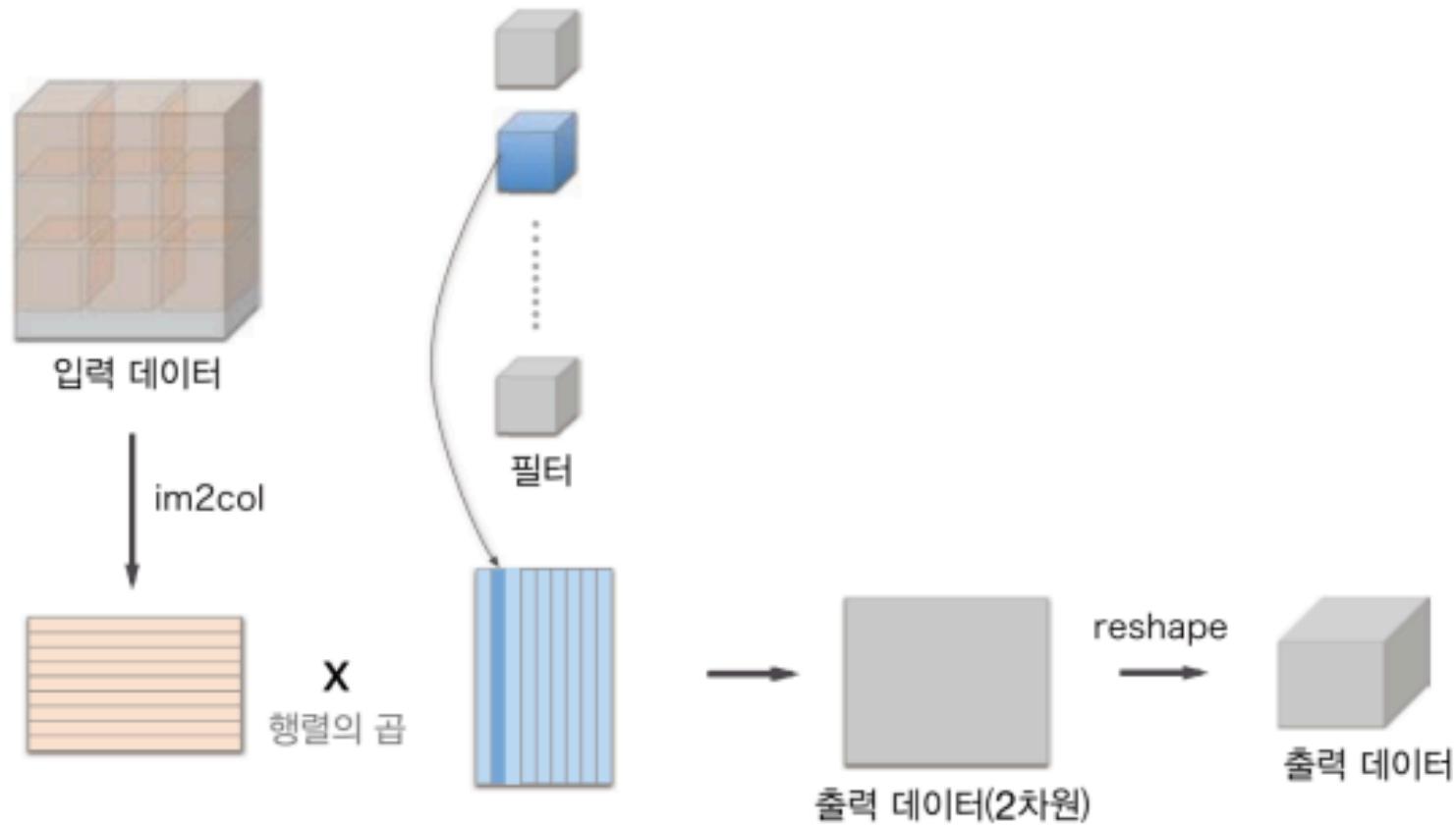


입력 데이터 : (7, 7, 3)
필터 사이즈 : (5, 5, 3)



IV im2col로 데이터 전개하기

그림 7-19 합성곱 연산의 필터 처리 상세 과정 : 필터를 세로로 1열로 전개하고, im2col이 전개한 데이터와 행렬 곱을 계산합니다. 마지막으로 출력 데이터를 변형(reshape)합니다.



```
def im2col(input_data, filter_h, filter_w, stride=1, pad=0):
    # 입력 데이터의 형태 추출
    N, C, H, W = input_data.shape

    # 출력 크기 계산
    out_h = (H + 2*pad - filter_h)//stride + 1
    out_w = (W + 2*pad - filter_w)//stride + 1

    # 패딩을 적용한 4차원 배열 생성
    img = np.pad(input_data, [(0,0), (0,0), (pad, pad), (pad, pad)], 'constant')

    # 각 필터 위치에 대한 열(column) 생성
    col = np.zeros((N, C, filter_h, filter_w, out_h, out_w))
    for y in range(filter_h):
        y_max = y + stride*out_h
        for x in range(filter_w):
            x_max = x + stride*out_w
            col[:, :, y, x, :, :] = img[:, :, y:y_max:stride, x:x_max:stride]

    # 2차원으로 변환
    col = col.transpose(0, 4, 5, 1, 2, 3).reshape(N*out_h*out_w, -1)
    return col
```

IV im2col로 데이터 전개하기

```
def im2col(input_data, filter_h, filter_w, stride=1, pad=0):
    # 입력 데이터의 형태 추출
    N, C, H, W = input_data.shape

    # 출력 크기 계산
    out_h = (H + 2*pad - filter_h)//stride + 1
    out_w = (W + 2*pad - filter_w)//stride + 1

    # 패딩을 적용한 4차원 배열 생성
    img = np.pad(input_data, [(0,0), (0,0), (pad, pad), (pad, pad)], 'constant')

    # 각 필터 위치에 대한 열(column) 생성
    col = np.zeros((N, C, filter_h, filter_w, out_h, out_w))
    for y in range(filter_h):
        y_max = y + stride*out_h
        for x in range(filter_w):
            x_max = x + stride*out_w
            col[:, :, y, x, :, :] = img[:, :, y:y_max:stride, x:x_max:stride]

    # 2차원으로 변환
    col = col.transpose(0, 4, 5, 1, 2, 3).reshape(N*out_h*out_w, -1)
    return col
```

```
x1 = np.random.rand(1, 3, 7, 7)
col1 = im2col(x1, 5, 5, stride=1,
               pad=0)
print(col1.shape)
```

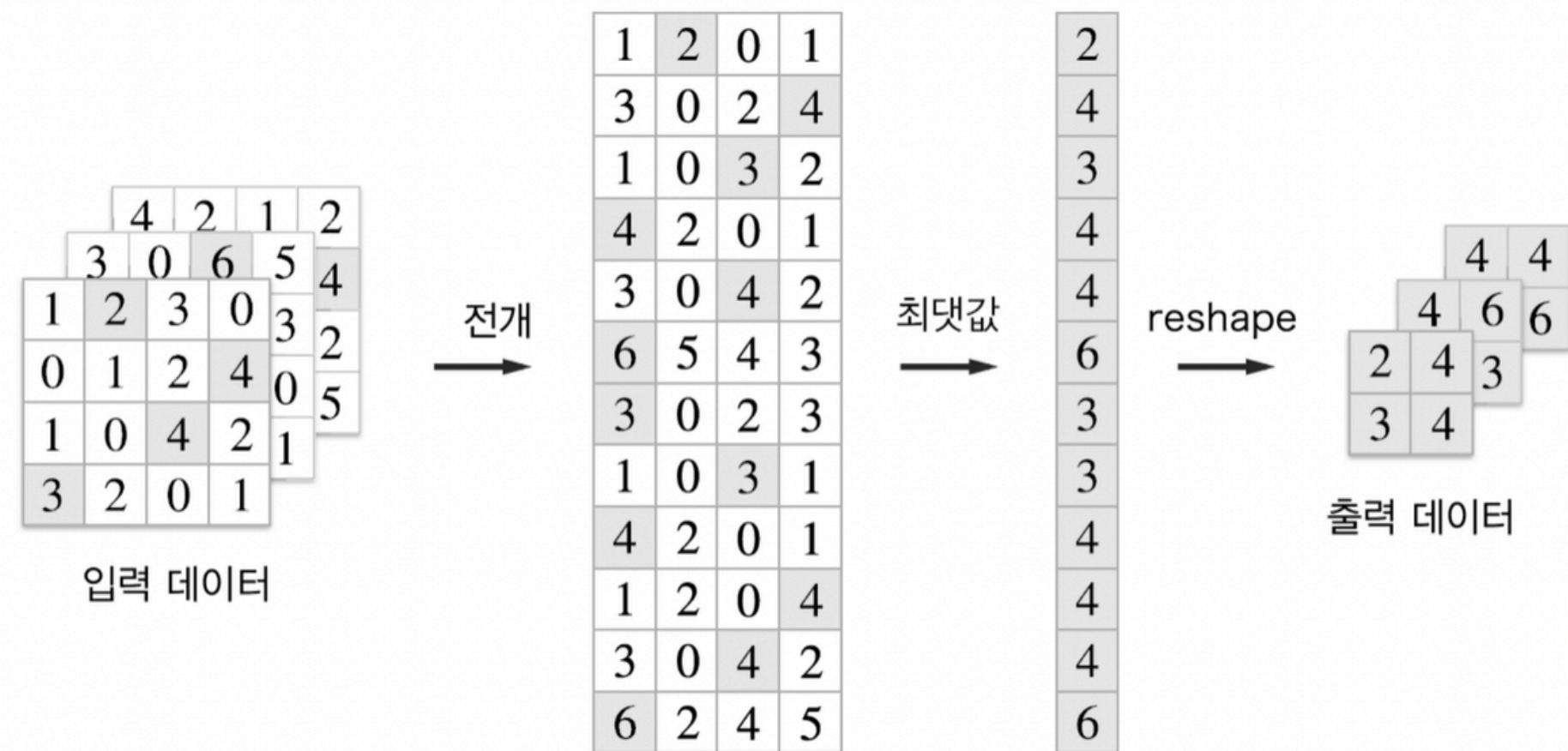
result : (9, 75)

```
x2 = np.random.rand(10, 3, 7, 7)
col2 = im2col(x2, 5, 5, stride=1,
               pad=0)
print(col2.shape)
```

result : QUIZ 3

IV 풀링 계층의 구현

- 풀링 계층도 합성곱 계층처럼 im2col을 사용해 입력 데이터를 전개
- 풀링의 경우 채널 쪽이 독립적이므로 채널마다 독립적으로 전개
- 전개 후 pooling 사이즈에 맞게 reshape
- 각 행마다 최댓값을 추출 (max pooling)
- 적절한 형상으로 다시 reshape



IV Max Pooling 구현

```
class MaxPooling:  
    def __init__(self, pool_h, pool_w, stride=1, pad=0):  
        self.pool_h = pool_h # 풀링 영역의 높이  
        self.pool_w = pool_w # 풀링 영역의 너비  
        self.stride = stride # 스트라이드 크기  
        self.pad = pad # 패딩 크기  
  
    def forward(self, x):  
        N, C, H, W = x.shape # 입력 데이터의 배치 크기, 채널 수, 높이, 너비  
  
        # 출력 데이터의 높이와 너비 계산  
        out_h = int(1 + (H - self.pool_h) / self.stride)  
        out_w = int(1 + (W - self.pool_w) / self.stride)  
  
        # 전개: 입력 데이터를 열(column) 형태로 변환  
        col = im2col(x, self.pool_h, self.pool_w, self.stride, self.pad)  
  
        # 풀링 영역의 크기만큼 열의 크기를 reshape  
        col = col.reshape(-1, self.pool_h * self.pool_w)  
  
        # 최댓값 풀링: 각 풀링 영역에서 최댓값을 계산  
        out = np.max(col, axis=1)  
  
        # 출력 데이터의 형태를 (배치 크기, 채널 수, 출력 높이, 출력 너비)로 변환  
        out = out.reshape(N, out_h, out_w, C).transpose(0, 3, 1, 2)  
  
    return out
```

예제 1)

```
x1 = np.random.rand(1, 3, 7, 7)  
max_pool = MaxPooling(pool_h=2,  
pool_w=2, stride=2)  
max_pool_output = max_pool.forward(x1)  
  
print("Max Pooling Output Shape:",  
max_pool_output.shape)  
print(max_pool_output)
```

```
Max Pooling Output Shape: (1, 3, 3, 3)  
[[[0.99544274 0.86993667 0.91689049]  
 [0.92573656 0.94737616 0.61802574]  
 [0.79493582 0.85787889 0.85644496]]]
```

```
[[0.70545092 0.87696007 0.99761803]  
 [0.63985597 0.91468909 0.59888182]  
 [0.91486933 0.96323366 0.52437713]]
```

```
[[0.49718874 0.88001152 0.59215277]  
 [0.67011632 0.85933699 0.8487361 ]  
 [0.86223669 0.99787501 0.42321258]]]]
```

IV Max Pooling 구현

```
class MaxPooling:  
    def __init__(self, pool_h, pool_w, stride=1, pad=0):  
        self.pool_h = pool_h # 풀링 영역의 높이  
        self.pool_w = pool_w # 풀링 영역의 너비  
        self.stride = stride # 스트라이드 크기  
        self.pad = pad # 패딩 크기  
  
    def forward(self, x):  
        N, C, H, W = x.shape # 입력 데이터의 배치 크기, 채널 수, 높이, 너비  
  
        # 출력 데이터의 높이와 너비 계산  
        out_h = int(1 + (H - self.pool_h) / self.stride)  
        out_w = int(1 + (W - self.pool_w) / self.stride)  
  
        # 전개: 입력 데이터를 열(column) 형태로 변환  
        col = im2col(x, self.pool_h, self.pool_w, self.stride, self.pad)  
  
        # 풀링 영역의 크기만큼 열의 크기를 reshape  
        col = col.reshape(-1, self.pool_h * self.pool_w)  
  
        # 최댓값 풀링: 각 풀링 영역에서 최댓값을 계산  
        out = np.max(col, axis=1)  
  
        # 출력 데이터의 형태를 (배치 크기, 채널 수, 출력 높이, 출력 너비)로 변환  
        out = out.reshape(N, out_h, out_w, C).transpose(0, 3, 1, 2)  
  
    return out
```

예제 2)

```
x2 = np.random.rand(10, 3, 7, 7)  
max_pool = MaxPooling(pool_h=2,  
pool_w=2, stride=2)  
max_pool_output = max_pool.forward(x2)  
  
print("Max Pooling Output Shape:",  
max_pool_output.shape)  
print(max_pool_output)
```

결과 일부

```
Max Pooling Output Shape: (10, 3, 3, 3)  
[[[ [0.98135577 0.98620439 0.40648507]  
    [0.70118042 0.71508876 0.77876965]  
    [0.82919228 0.84353838 0.9862894] ]  
  
    [[ 0.62523863 0.40855324 0.9201146 ]  
    [ 0.94828394 0.82550479 0.71399295 ]  
    [ 0.5999845 0.80901882 0.71158592 ] ]  
  
    [[ 0.79608064 0.35302635 0.94363122 ]  
    [ 0.81980819 0.90674514 0.87700248 ]  
    [ 0.65199228 0.7294771 0.89457442 ] ]  
  
    [[[ 0.97007271 0.93209391 0.52774553 ]  
    [ 0.78289261 0.72920278 0.85074521 ]
```

IV Average Pooling 구현

```
class AveragePooling:  
    def __init__(self, pool_h, pool_w, stride=1, pad=0):  
        self.pool_h = pool_h # 풀링 영역의 높이  
        self.pool_w = pool_w # 풀링 영역의 너비  
        self.stride = stride # 스트라이드 크기  
        self.pad = pad # 패딩 크기  
  
    def forward(self, x):  
        N, C, H, W = x.shape # 입력 데이터의 배치 크기, 채널 수, 높이, 너비  
  
        # 출력 데이터의 높이와 너비 계산  
        out_h = int(1 + (H - self.pool_h) / self.stride)  
        out_w = int(1 + (W - self.pool_w) / self.stride)  
  
        # 전개: 입력 데이터를 열(column) 형태로 변환  
        col = im2col(x, self.pool_h, self.pool_w, self.stride, self.pad)  
  
        # 풀링 영역의 크기만큼 열의 크기를 reshape  
        col = col.reshape(-1, self.pool_h * self.pool_w)  
  
        # 평균 풀링: 각 풀링 영역에서 평균값을 계산  
        out = np.  
            QUIZ 4  
            # 출력 데이터의 형태를 (배치 크기, 채널 수, 출력 높이, 출력 너비)로 변환  
            out = out.reshape(N, out_h, out_w, C).transpose(0, 3, 1, 2)  
  
    return out
```

예제 1)

```
x1 = np.random.rand(1, 3, 7, 7)  
avg_pool = AveragePooling(pool_h=2,  
                           pool_w=2, stride=2)  
avg_pool_output = avg_pool.forward(x1)  
  
print("Average Pooling Output Shape:",  
      avg_pool_output.shape)  
print(avg_pool_output)  
  
Average Pooling Output Shape: (1, 3, 3, 3)  
[[[[0.50148256 0.41663914 0.44549506]  
   [0.54387247 0.25273934 0.79525154]  
   [0.41384216 0.53178579 0.64914696]]  
  
   [[0.53129761 0.34539329 0.38818979]  
   [0.61190935 0.37166596 0.54845949]  
   [0.18023073 0.23936105 0.30891174]]  
  
   [[0.30304683 0.64823995 0.28043632]  
   [0.42492869 0.534159 0.60628092]  
   [0.55449051 0.44342996 0.44927928]]]]]
```

IV Average Pooling 구현

```
class AveragePooling:  
    def __init__(self, pool_h, pool_w, stride=1, pad=0):  
        self.pool_h = pool_h # 풀링 영역의 높이  
        self.pool_w = pool_w # 풀링 영역의 너비  
        self.stride = stride # 스트라이드 크기  
        self.pad = pad # 패딩 크기  
  
    def forward(self, x):  
        N, C, H, W = x.shape # 입력 데이터의 배치 크기, 채널 수, 높이, 너비  
  
        # 출력 데이터의 높이와 너비 계산  
        out_h = int(1 + (H - self.pool_h) / self.stride)  
        out_w = int(1 + (W - self.pool_w) / self.stride)  
  
        # 전개: 입력 데이터를 열(column) 형태로 변환  
        col = im2col(x, self.pool_h, self.pool_w, self.stride, self.pad)  
  
        # 풀링 영역의 크기만큼 열의 크기를 reshape  
        col = col.reshape(-1, self.pool_h * self.pool_w)  
  
        # 평균 풀링: 각 풀링 영역에서 평균값을 계산  
        out = np.  
            QUIZ 4  
        # 출력 데이터의 형태를 (배치 크기, 채널 수, 출력 높이, 출력 너비)로 변환  
        out = out.reshape(N, out_h, out_w, C).transpose(0, 3, 1, 2)  
  
    return out
```

예제 2)

```
x2 = np.random.rand(10, 3, 7, 7)  
avg_pool = AveragePooling(pool_h=2,  
                           pool_w=2, stride=2)  
avg_pool_output = avg_pool.forward(x2)  
  
print("Average Pooling Output Shape:",  
      avg_pool_output.shape)  
print(avg_pool_output)
```

결과 일부

```
Average Pooling Output Shape: (10, 3, 3, 3)  
[[[0.57719912 0.42447491 0.28164485]  
 [0.50609054 0.51523989 0.58585696]  
 [0.60264207 0.3777563 0.54884701]]  
  
 [[0.34670436 0.2606746 0.384535 ]  
 [0.60261855 0.58026608 0.43201184]  
 [0.41482189 0.68765784 0.56571204]]  
  
 [[0.59613937 0.22973067 0.37073831]  
 [0.51873355 0.62615109 0.52551361]  
 [0.38959481 0.28088083 0.56704952]]  
  
 [[0.5160699 0.46559569 0.36944478]
```

IV Global Average Pooling 실습(pytorch)

```
# NumPy 배열을 PyTorch 텐서로 변환
x1 = np.random.rand(1, 3, 7, 7)
x1_tensor = torch.tensor(x1, dtype=torch.float32)

# Global Average Pooling 적용
# 방법 1: F.avg_pool2d 사용
gap_output_1 = F.avg_pool2d(x1_tensor, x1_tensor.size()[2:])

# 방법 2: F.adaptive_avg_pool2d 사용
gap_output_2 = F.adaptive_avg_pool2d(x1_tensor, (1, 1))

# 방법 3: torch.mean 사용
gap_output_3 = torch.mean(x1_tensor.view(x1_tensor.size(0),
                                         x1_tensor.size(1), -1), dim=2)

# 결과 출력
print(gap_output_1.shape) # (1, 3, 1, 1)
print(gap_output_2.shape) # (1, 3, 1, 1)
print(gap_output_3.shape) # (1, 3)
```

**Thank you
for listening!**