Convolution 구현

최석재 *lingua@naver.com*

3차원 합성곱

3D Convolution

3차원 데이터 합성곱 연산

- 앞에서는 2차원 데이터에 대하여 합성곱 연산을 알아보았다
- 그러나 이미지 데이터는 R, G, B의 3채널로 이루어져 있는 3차원 데이터
- 3차원 데이터에서 합성곱 연산은 어떻게 이루어지는지 알아본다
- 입력 데이터는 다음과 같이 구성되어 있다

R

1	2	3	0
0	1	2	3
3	0	1	2
2	3	0	1

G

1	3	2	0
0	1	3	2
2	0	3	1
3	2	1	0

B

2	3	1	0
0	2	3	1
1	0	2	3
3	1	0	2

작성하세요!

- import numpy as np
- input_data = np.array([

```
    [[1, 2, 3, 0],
     [0, 1, 2, 3],
     [3, 0, 1, 2],
     [2, 3, 0, 1]],
    [[1, 3, 2, 0],
     [0, 1, 3, 2],
     [2, 0, 3, 1],
     [3, 2, 1, 0]],
    [[2, 3, 1, 0],
     [0, 2, 3, 1],
     [1, 0, 2, 3],
     [3, 1, 0, 2]]
])
```

3x4x4 크기의
3채널, 4x4 이미지

3차원 데이터의 합성곱 연산

- 하나의 필터도 입력 데이터와 동일한 채널 수를 가져야 한다

1

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

2

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

3

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

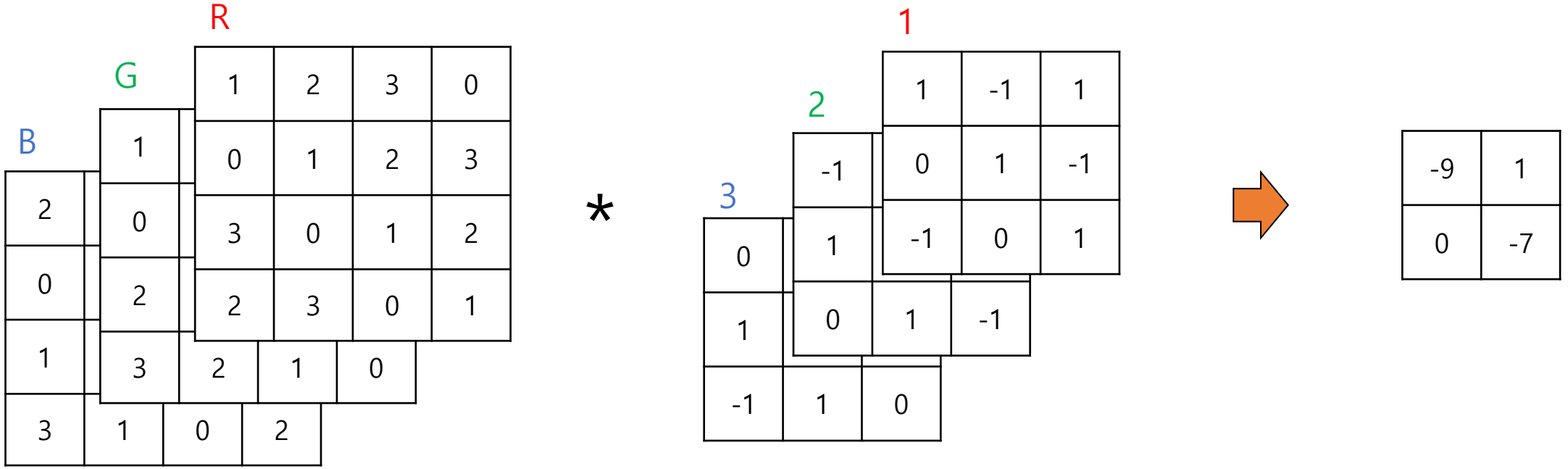
작성하세요!

- kernel = np.array([
  [[1, -1, 1],
  [0, 1, -1],
  [-1, 0, 1]],
  [[-1, 1, 0],
  [1, -1, 0],
  [0, 1, -1]],
  [[0, -1, 1],
  [1, 0, -1],
  [-1, 1, 0]]
])

3x3x3 크기의
3채널, 3x3 필터

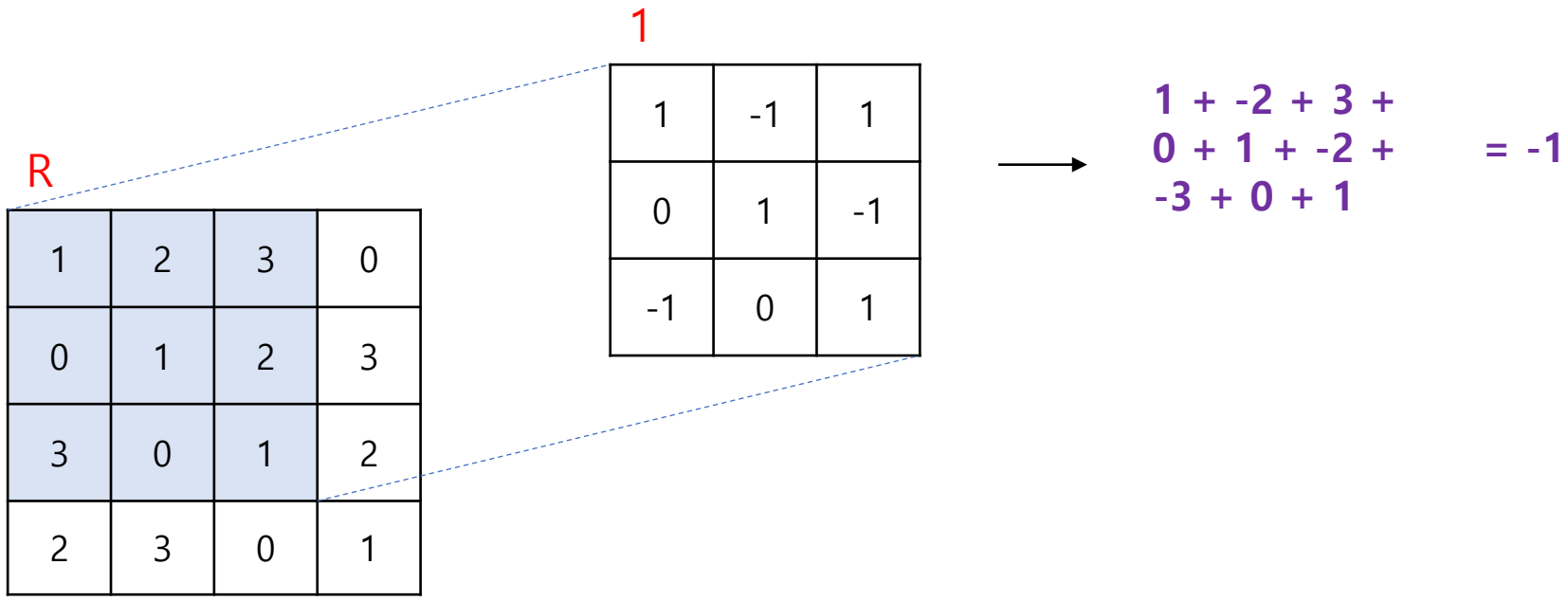
3차원 데이터의 합성곱 연산

- 3개의 채널을 갖는 입력 데이터의 역시 3개의 채널을 갖는 필터를 합성곱 연산하면
- 1개의 채널을 갖는 Feature Map을 얻는다
- 입력 데이터는 4x4x3, 필터는 3x3x3이며 stride=1 일 때, Feature Map은 2x2x1이 된다



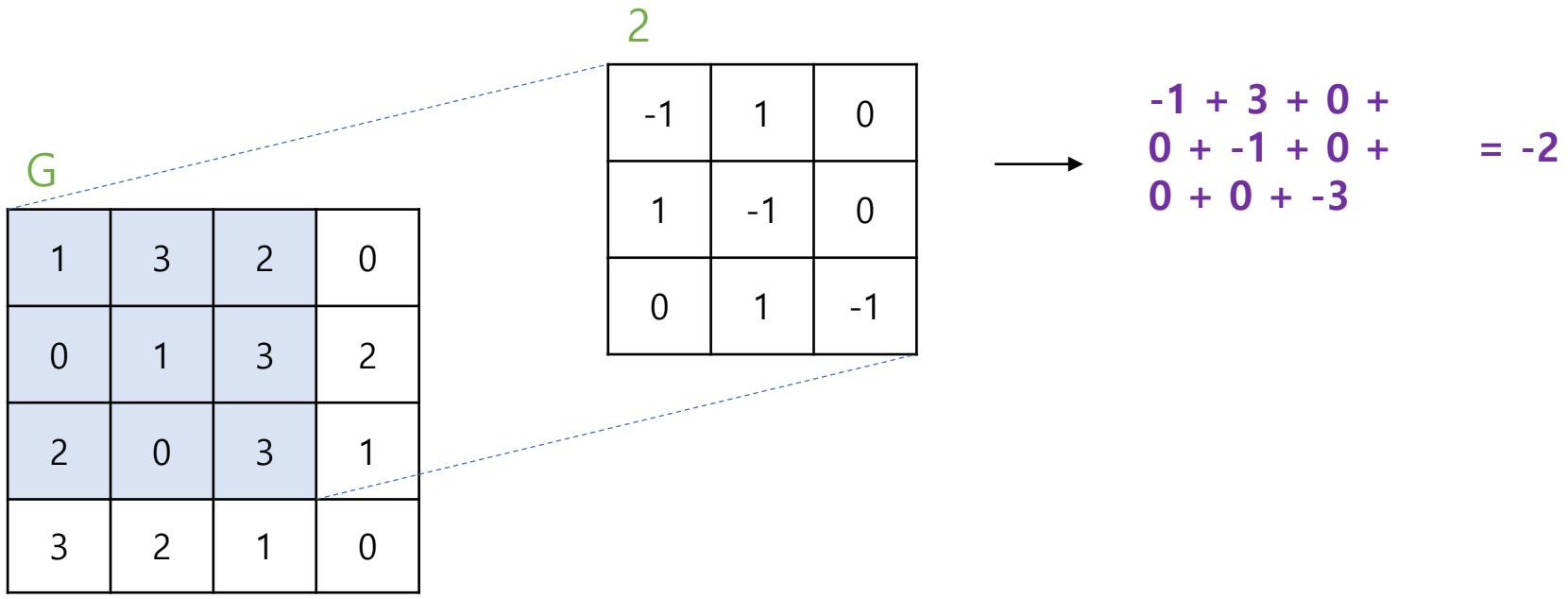
첫 번째 채널 연산

- 첫 번째 채널의 첫 번째 포지션에서 연산은 다음과 같이 이루어진다



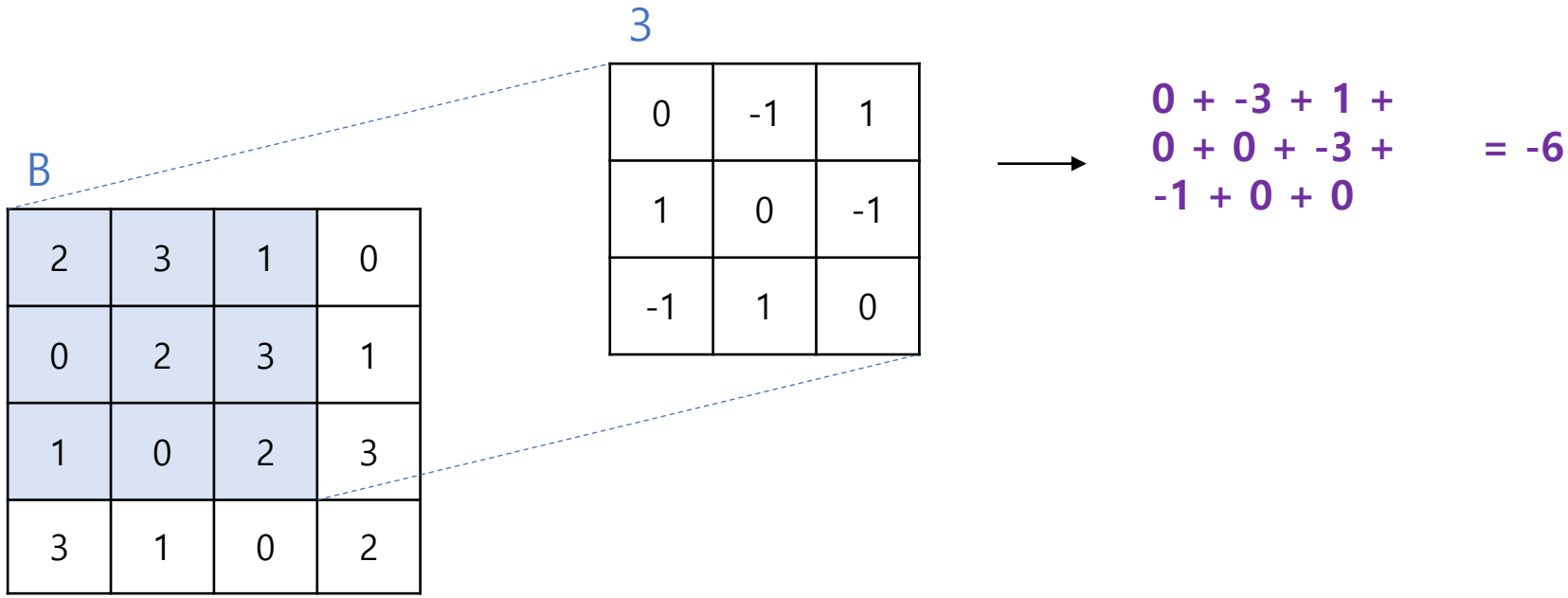
두 번째 채널 연산

- 커널 이동 없이 두 번째 채널의 첫 번째 포지션에서 연산이 이루어진다



세 번째 채널 연산

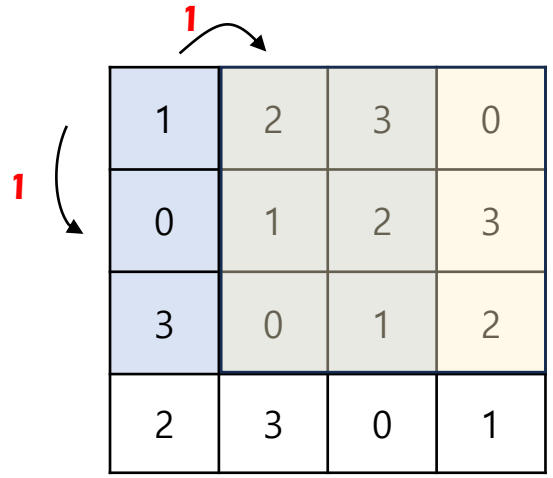
- 커널 이동 없이 세 번째 채널의 첫 번째 포지션에서 연산이 이루어진다



커널 이동 연산

- 세 채널에서의 값을 더하면 -1 + -2 + -6 = -9 가 된다
- 이것을 Feature Map의 첫 번째 셀에 채우고,
- 이와 같은 식으로 stride=1 커널 이동을 하면
- 총 4번의 커널 이동이 있게 된다. 이 값들을 채워 Feature Map을 완성한다

-9			



-9	1
0	-7

1 + -2 + 3 + 0 + 1 + -2 + -3 + 0 + 1 + -1 + 3 + 0 + 0 + -1 + 0 + 0 + 0 + -3 + 0 + -3 + 1 + 0 + 0 + -3 + -1 + 0 + 0 = -9

즉, 요소별 곱셈 후 모든 결과들을 더한다

3차원 데이터의 합성곱 구현

- 3차원 데이터의 합성곱 연산을 구현하면 다음과 같다 (앞에서 작성한 input_data와 kernel은 생략)

```
# 출력 데이터 초기화
output_data = np.zeros((2, 2), dtype=np.int16)  # 음수 표현을 위해 -32768 ~ 32767까지 표현할 수 있는 int16을 사용

# 합성곱 연산 수행
for y in range(2):                                  # 출력 데이터의 높이 (4-3+1=2)
    for x in range(2):                              # 출력 데이터의 너비 (4-3+1=2)
        roi = input_data[:, y:y+3, x:x+3]           # 현재 위치에서 3x3 영역 추출
        filtered = roi * kernel                     # (3x3x3) * (3x3x3)의 합성곱 (모든 채널 연산)
        conv_value = np.sum(filtered)               # 요소별 곱셈 후 모든 결과들을 더함
        output_data[y, x] = np.int16(conv_value)    # 결과를 int16으로 변환
```

- print("Feature Map:₩n", output_data)

```
Feature Map:
 [[-9  1]
 [ 0 -7]]
```

np.pad

np.pad 함수 연습

- Stride & Zero Padding에서 제로 패딩을 직접 구현하였으나
- 넘파이에는 패딩을 쉽게 구현할 수 있게 하는 함수가 구현되어 있다

- 기본 사용법은 np.pad(array, (before, after)) 와 같다

- array = np.array([1, 2])
- padded_array = np.pad(array, [(1, 1)], constant_values=0)
- print(padded_array)

배열의 앞과 뒤에 각각 하나씩의 패딩을 삽입

패딩 값은 0으로 설정

```
[0 1 2 0]
```

np.pad

- import numpy as np
- array = np.array([[1, 2], [3, 4]]) # 초기 배열 생성

- print('--------패딩이 없는 원본 -------')
- padded_array = np.pad(array, [(0, 0), (0, 0)], constant_values=0)
- print(padded_array)

행과 열 모두 패딩 설정 없음

- print('--------첫번째 차원인 행 차원의 앞과 뒤에 1개씩 패딩 -------')
- padded_array = np.pad(array, [(1, 1), (0, 0)], constant_values=0)
- print(padded_array)

행 쪽(첫번째 축)에만 앞과 뒤에 패딩 설정

```
--------패딩이 없는 원본 -------
[[1 2]
 [3 4]]
--------첫번째 차원인 행 차원의 앞과 뒤에 1개씩 패딩 -------
[[0 0]
 [1 2]
 [3 4]
 [0 0]]
```

constant_values=0 은 기본값이므로 생략 가능

np.pad

- print('--------두번째 차원인 열 차원의 앞과 뒤에 1개씩 패딩-------')
- padded_array = np.pad(array, [(0, 0), (1, 1)], constant_values=0)
- print(padded_array)

- print('--------첫번째 차원, 두번째 차원 모두 1개씩 패딩-------')
- padded_array = np.pad(array, [(1, 1), (1, 1)], constant_values=0)
- print(padded_array)

- print('--------두번째 차원인 열 차원의 앞과 뒤에 2개씩 패딩 -------')
- padded_array = np.pad(array, [(0, 0), (2, 2)], constant_values=0)
- print(padded_array)

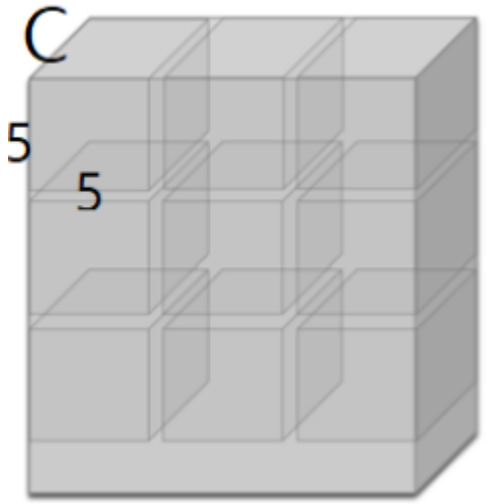
```
--------두번째 차원인 열 차원의 앞과 뒤에 1개씩 패딩-------
[[0 1 2 0]
 [0 3 4 0]]
--------첫번째 차원, 두번째 차원 모두 1개씩 패딩-------
[[0 0 0 0]
 [0 1 2 0]
 [0 3 4 0]
 [0 0 0 0]]
--------두번째 차원인 열 차원의 앞과 뒤에 2개씩 패딩 -------
[[0 0 1 2 0 0]
 [0 0 3 4 0 0]]
```

im2col

4차원 데이터

- 실제 딥러닝에서 데이터를 처리할 때는
- (채널, 높이, 너비)의 3차원 데이터에 한 번에 몇 개의 데이터를 묶어 처리하는
- 배치 차원까지 더해져 (배치, 채널, 높이, 너비)의 4차원 데이터를 처리하게 된다

- 4차원 데이터는 코드로는 다음과 같이 표현할 수 있다



5x5의 높이와 너비에 C(3)만큼의 채널을 가진 데이터가 9개 묶여 있는 경우

- import numpy as np
- x = np.random.randint(low=0, high=101, size=(9, 3, 5, 5))
- print(x.shape)
- print(x)

```
(9, 3, 5, 5)
[[[[ 68  38   9  84  31]
   [ 82  89  33  64  21]
   [ 86  65  55  35  37]
   [ 24   6   1  45  68]
   [  3  60  77  87  32]]

  [[ 88  73  99  93  27]
   [ 16   4  66  35  33]
   [ 45  12  53  78  76]
   [ 96   5  89  71  65]
   [ 27  85   9  28  37]]

  [[ 68  51  10  47  35]
   [ 63  98  93  16  22]
   [ 13  45  16  98  85]
   [ 19  33  60  22  88]
   [ 83  10  63  83  76]]]


 [[[ 40  18  93  52  15]
   [ 10  40  45  77  22]
   [ 88  94  63  71  54]
   [ 62   1  31  41  93]
   [ 95  88  86  33  42]]

  [[ 19  98  93  36  79]
   [ 34  94  21  84  74]
   [ 61  23  90  87  14]
   [ 46  82  88  10  61]
   [ 54  49  80  57  45]]

  [[ 28  78  50  98  87]
   [ 71  73  62  88  26]
   [  3  86  18  86  42]
   [ 49  63  77  89  33]
   [ 30   5  87  25  77]]]


 [[[ 45  84  99  12  63]
```

4차원 데이터

배치 1 데이터

- print(x[0])

배치 1의 채널 1 (2차원 이미지)

- print(x[0][0])

```
# 배치 1 데이터
print(x[0])
```

```
[[[68 38  9 84 31]
  [82 89 33 64 21]
  [86 65 55 35 37]
  [24  6  1 45 68]
  [ 3 60 77 87 32]]

 [[88 73 99 93 27]
  [16  4 66 35 33]
  [45 12 53 78 76]
  [96  5 89 71 65]
  [27 85  9 28 37]]

 [[68 51 10 47 35]
  [63 98 93 16 22]
  [13 45 16 98 85]
  [19 33 60 22 88]
  [83 10 63 83 76]]]
```

3

```
# 배치 1의 채널 1 (2차원 이미지)
print(x[0][0])
```

```
[[68 38  9 84 31]
 [82 89 33 64 21]
 [86 65 55 35 37]
 [24  6  1 45 68]
 [ 3 60 77 87 32]]
```

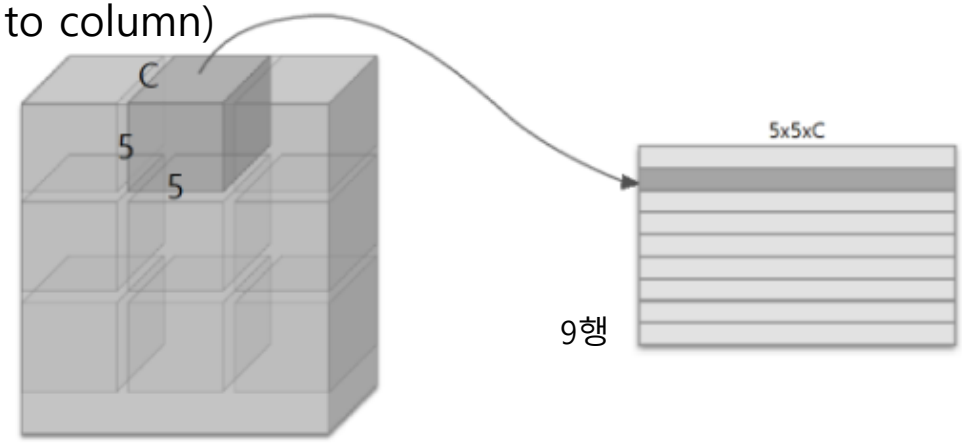
5x5

4차원 데이터의 합성곱

- 4차원 데이터를 앞에서 본 바와 같이 직접 구현하려면 for 문을 반복 중첩해야 함
- 다루기 어려울 뿐만 아니라, 넘파이 배열에 for 문을 사용하면 성능 하락의 문제가 있음

- 따라서 for 문의 사용을 최소화하기 위해 4차원 데이터를 2차원으로 reshape 함
- 2차원 데이터는 행렬 연산에 더욱 최적화되어 있음

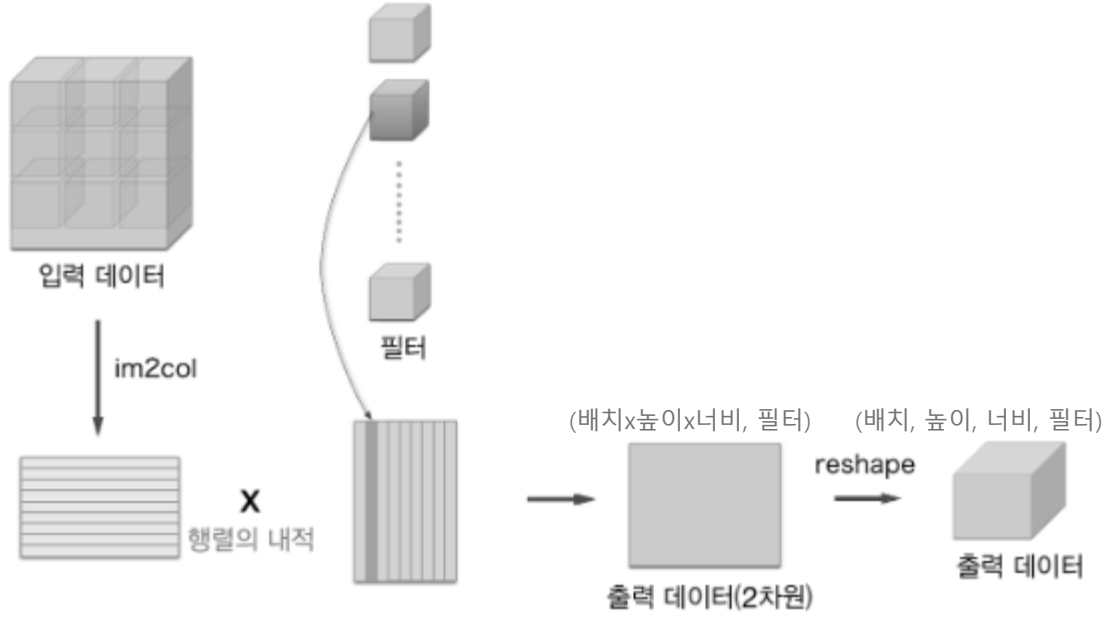
- 4차원 데이터를 배치 단위 별로 쪼개어 2차원 데이터로 만듦
- 즉, 3차원 이미지 데이터를 컬럼 형태로 만드는 것 (image to column)



필터 reshape

- 필터도 reshape해야 한다
- 필터의 채널 수는 입력 데이터의 채널 수와 같아야 하므로, 3차원이며
- 여기에 배치 차원이 더해져 4차원 데이터
- 앞에서 본 바와 같은 방식으로 정리하여 역시 2차원 데이터로 만든다
- 그리고 입력 데이터와 행렬 연산으로 합성곱을 수행한다

- 출력 결과는 2차원으로 나오게 되고,
- 각 이미지 데이터를 개별적으로 처리할 수 있도록
- 데이터를 다시 4차원으로 reshape한다



im2col 구현

- def im2col(img, kh, kw, stride=1, pad=0): # 필터높이(kh), 필터너비(kw), 필터 이동 간격(stride), 패딩 크기(pad)
  N, C, H, W = img.shape # N은 배치 크기, C는 채널 수, H는 이미지의 높이, W는 이미지의 너비
  out_h = (H + 2*pad - kh)//stride + 1 # 출력 데이터의 높이 계산
  out_w = (W + 2*pad - kw)//stride + 1 # 출력 데이터의 너비 계산

  (배치, 채널, 높이, 너비)
  padded_img = np.pad(array=img, pad_width=[(0,0), (0,0), (pad, pad), (pad, pad)]) # 높이와 너비 차원에만 pad값으로 패딩 적용
  col = np.zeros((N, C, kh, kw, out_h, out_w)) # (배치 크기, 채널 수, 필터 높이, 필터 너비, 출력 높이, 출력 너비)로 결과 배열 초기화

  for y in range(kh): # 필터 커널이 가질 수 있는 모든 높이 위치에 대하여 탐색
    y_max = y + stride*out_h # 해당 필터의 현재 높이 위치에서 이미지 데이터에 접근할 범위의 끝 계산
    for x in range(kw): # 필터 커널이 가질 수 있는 모든 너비 위치에 대하여 탐색
      x_max = x + stride*out_w # 해당 필터의 현재 너비 위치에서 이미지 데이터에 접근할 범위의 끝 계산
      col[:, :, y, x, :, :] = padded_img[:, :, y:y_max:stride, x:x_max:stride] # 현재 필터 위치(y, x)에 해당하는 데이터 추출하여 결과 배열에 입력
                                                (start:stop:step)

  col = col.transpose(0, 4, 5, 1, 2, 3).reshape(N*out_h*out_w, -1) # 합성곱을 단순한 행렬곱셈으로 처리할 수 있도록 구조 변경
  return col

데이터 입력 부분

※ 데이터 입력 과정을 살펴보기 위해 코드를 약간 수정한다

- def im2col_test(img, kh, kw, stride=1, pad=0):

```
    N, C, H, W = img.shape
    out_h = (H + 2*pad - kh) // stride + 1                    # 2
    out_w = (W + 2*pad - kw) // stride + 1                    # 2


    padded_img = np.pad(array=img, pad_width=[(0, 0), (0, 0), (pad, pad), (pad, pad)])
    col = np.zeros((N, C, kh, kw, out_h, out_w))
    print("입력 전:₩n", col, '₩n--------------')

    for y in range(kh):
        y_max = y + stride * out_h
        for x in range(kw):
            x_max = x + stride * out_w
            col[:, :, y, x, :, :] = padded_img[:, :, y:y_max:stride, x:x_max:stride]
            print(f'col[:, :, {y}, {x}, :, :]:')
            print(col[:, :, y, x, :, :], '₩n--------------')

    return col
```

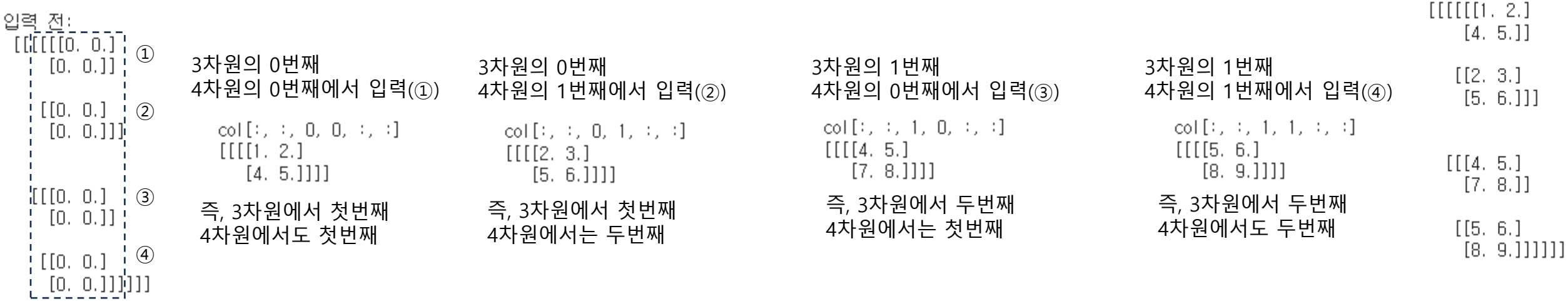
← 마지막의 col = col.transpose(0, 4, 5, 1, 2, 3).reshape(N*out_h*out_w, -1) 부분 삭제

데이터 입력 부분

작은 크기의 입력 데이터로 테스트

- x1 = np.array([[[[1, 2, 3],
  [4, 5, 6],
  [7, 8, 9]]]]) # 1개의 배치, 1개의 채널, 3x3 이미지

- result = im2col_test(x1, kh=2, kw=2, stride=1, pad=0)
- print("Final result:₩n", result)



차원 변경 부분

- col = col.transpose(0, 4, 5, 1, 2, 3).reshape(N*out_h*out_w, -1)
- 이 부분은 합성곱을 단순한 행렬 곱셈으로 처리할 수 있도록 구조를 변경한다

- 먼저 tranpose(0, 4, 5, 1, 2, 3)의 순으로 배열의 축 순서를 변경한 뒤, reshape()로 2차원 데이터를 만든다
- transpose 이전 : (배치크기, 채널수, 필터높이, 필터너비, 출력높이, 출력너비)
- transpose 이후 : (배치크기, 출력높이, 출력너비, 채널수, 필터높이, 필터너비)

- 다음으로 각 행이 필터의 크기와 채널 수에 따라 독립적인 연산을 수행할 수 있도록 2D로 reshape 한다
- 즉, 각 위치에서의 필터링을 위한 입력 벡터를 행으로 가지게 된다
- reshape 이후 : (배치크기x출력높이x출력너비, 채널수x필터높이x필터너비)
- reshape(..., -1) 의 -1은 나머지 모든 차원을 자동으로 계산하여 하나의 차원으로 만든다

차원 변경 확인

- x1 = np.random.rand(9, 3, 5, 5) # (9, 3, 5, 5) shape로 난수 생성 (배치크기, 채널수, 필터높이, 필터너비)
- col1 = im2col(x1, kh=5, kw=5, stride=1, pad=0)
- print(col1.shape) # (9, 75) (배치크기x출력높이x출력너비, 채널수x필터높이x필터너비)

- 4차원 데이터가 2차원 데이터로 잘 변형되었다
- 동일한 필터를 가지는 경우 배치 크기를 늘려보면 그만큼 행이 늘어난다

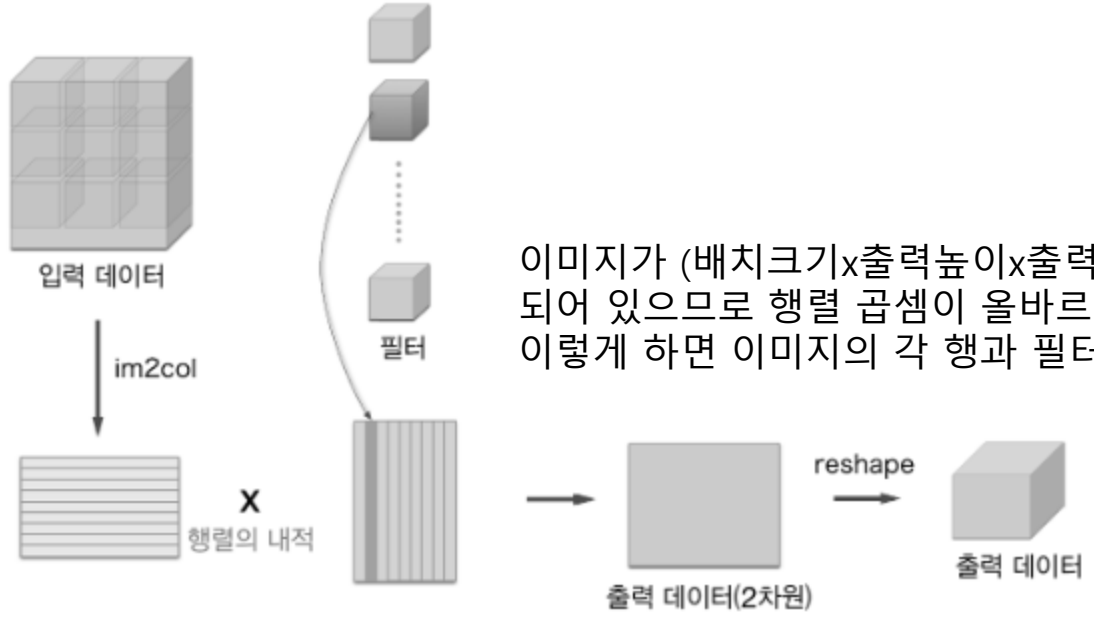
- x2 = np.random.rand(90, 3, 5, 5) # (90, 3, 5, 5)
- col2 = im2col(x2, 5, 5, stride=1, pad=0)
- print(col2.shape) # (90, 75) (배치크기x출력높이x출력너비, 채널수x필터높이x필터너비)

4차원 합성곱

4D Convolution

4차원 데이터의 합성곱

- 이제 4차원 데이터의 합성곱을 구현해본다
- 먼저 이미지 데이터와의 행렬 연산을 위해 필터를 2D로 변환하는 함수를 정의한다
- 필터가 가지고 있는 필터 수, 채널 수, 필터 높이, 필터 너비를 받아
- (필터 수, 채널 수 x 필터 높이 x 필터 너비)의 2차원 데이터로 출력한다
- 이후 이것을 다시 전치하여 (채널 수 x 필터 높이 x 필터 너비, 필터 수)로 만들 것이다



이미지가 (배치크기x출력높이x출력너비, 채널수x필터높이x필터너비)로
되어 있으므로 행렬 곱셈이 올바르게 이루어지도록 필터를 전치한다
이렇게 하면 이미지의 각 행과 필터의 각 열이 곱셈 연산에 적절히 대응된다

필터 reshape 함수

- import numpy as np

- def fil2col(kernel):

```
    filter_num, channel_num, filter_height, filter_width = kernel.shape
    col = filters.reshape(filter_num, channel_num * filter_height * filter_width)
    return col
```

필터 reshape 확인

- 필터 reshape가 올바르게 되는지 확인해본다

6개의 필터가 있고 각 필터가 3개의 채널을 가지며, 필터의 크기는 5x5
- filters = np.random.rand(6, 3, 5, 5)

필터를 2차원 행렬로 변환
- fil_col = fil2col(filters)
- print(fil_col.shape) # (6, 75) (필터 수, 채널 수 x 필터 높이 x 필터 너비)

합성곱 준비

- 이미지 데이터와 필터를 모두 2차원 데이터로 만들어 합성곱(행렬연산)을 준비한다

- N, C, H, W = 1, 3, 5, 5 # 1개의 이미지, 3개의 채널, 5x5 크기의 이미지
- F, kh, kw, stride, pad = 6, 3, 3, 1, 1 # 6개의 필터, 3x3 필터 크기, 스트라이드 1, 패딩 1

- input_data = np.random.rand(N, C, H, W) # 이미지 데이터 생성
- filters = np.random.rand(F, C, kh, kw) # 필터 가중치 생성

- img_col = im2col(input_data, kh, kw, stride, pad) # im2col() 적용
- print("im2col 결과:₩n", img_col.shape) # (배치 크기x출력 높이x출력 너비, 채널 수x필터 높이x필터 너비)의 2D 출력

- fil_col = fil2col(filters) # fil2col() 적용
- print("fil2col 결과:₩n", fil_col.shape) # (필터 수, 채널 수x필터 높이x필터 너비)의 2D 출력

```
im2col 결과:
(25, 27)
fil2col 결과:
(6, 27)
```

합성곱 연산 수행

```
im2col 결과:
 (25, 27)
fil2col 결과:
 (6, 27)
```

fil_col의 두 번째 차원 (채널 수x필터 높이x필터 너비)는 img_col의 역시 두 번째 차원 (채널 수x필터 높이x필터 너비)와 일치
행렬 곱을 이루려면 앞쪽 행렬의 열 수와 뒷쪽 행렬의 행 수가 일치해야 하므로 fil_col을 전치시킨다

- conv_out = np.dot(img_col, fil_col.T)

결과는 (배치크기x출력높이x출력너비, 필터 수)

- print("합성곱 연산 결과:₩n", conv_out.shape)
- print(conv_out)

```
합성곱 연산 결과:
 (25, 6)
[[3.39464953 2.18736495 2.6029757  2.3434502  2.03916592 1.95158197]
 [5.50497627 3.2592215  4.55407015 4.18343031 3.66212278 3.89095355]
 [4.66325415 3.56739735 4.6151765  4.30365776 3.62596986 3.70793438]
 [5.38462655 3.11597174 4.35834804 3.90953246 3.63594099 3.86840129]
 [2.94935658 1.506678   2.19174942 2.02482642 2.56165211 2.04556065]
 [4.00597776 4.25583213 4.41297651 5.37095705 4.30765969 2.60767869]
 [7.95044949 5.94862698 7.27468588 6.64753164 5.99500404 5.92045358]
 [7.03550098 4.94551753 6.79088366 6.16481692 5.93939    5.33762836]
 [6.18707103 5.12162698 5.98952653 6.97490544 6.02848459 5.04666116]
 [4.69821129 3.12907611 3.93947747 3.77001805 4.28844448 3.40456111]
 [4.4662069  4.04153392 3.79332754 4.60978115 3.92104474 3.59110305]
 [7.54605732 5.77392862 7.43958071 7.21291342 5.89126122 4.97165358]
 [6.61349779 4.21480378 6.49540994 5.69526842 4.96276127 4.88829653]
 [7.88357746 5.45170006 6.2853355  6.55056856 5.64085509 5.50375818]
```

행렬 곱셈은 앞쪽 행렬의 행(row)과 뒷쪽 행렬의 열(column)을 상호 관계시키는 연산.
합성곱 연산에서는 행렬 곱셈을 이용하여 이미지 데이터와 필터를 상호 관계시켜,
필터가 이미지의 모든 위치에서 효율적으로 적용되도록 한다

오른쪽에서 6개의 컬럼은 각각의 필터에 대한 적용 결과이다

4차원 데이터로 변환

- 신경망 구조가 각 이미지를 개별적으로 파악하고 처리할 수 있도록 하기 위해서는
- (배치크기x출력높이x출력너비, 필터 수)로 재구성해야 한다

- 배치 크기: 각각의 이미지를 개별적으로 처리할 수 있도록 배치 차원을 유지한다
- 출력 높이와 너비: 공간적인 특징을 포착하기 위하여 출력 높이와 출력 너비를 유지한다
- 필터 수: 각 필터는 입력 데이터에 대해 독립적인 특징을 추출하므로 유지한다

```
- out_h = (H + 2*pad - kh) // stride + 1          # 출력 데이터의 높이 계산
- out_w = (W + 2*pad - kw) // stride + 1          # 출력 데이터의 너비 계산
- conv_out = conv_out.reshape(N, out_h, out_w, F) # (배치크기, 출력높이, 출력너비, 필터개수)
```

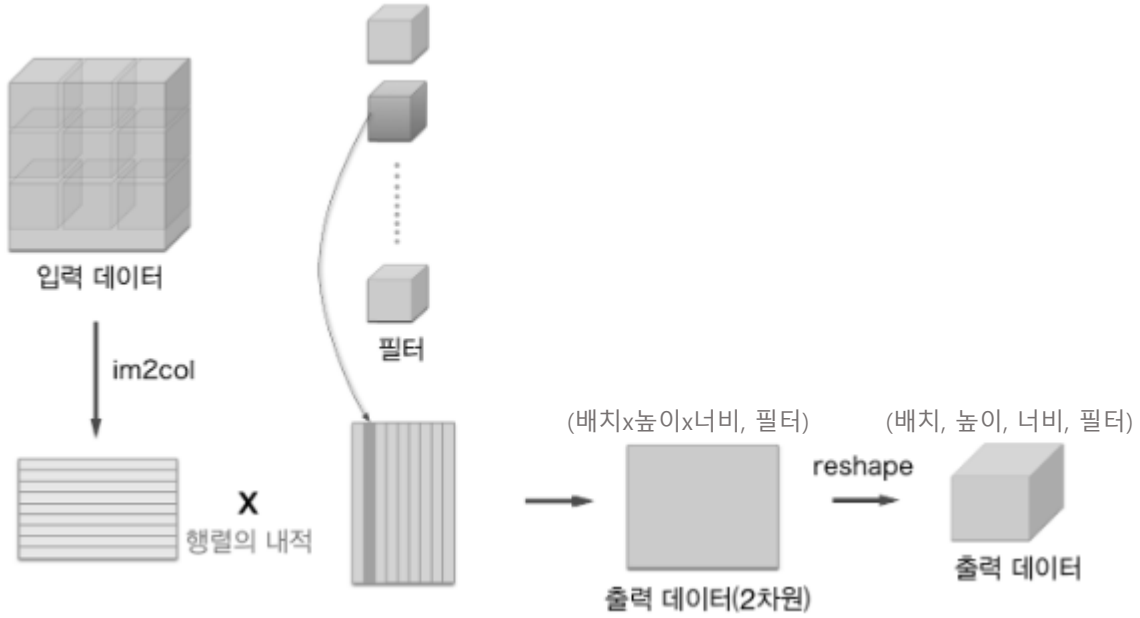
```
- print(conv_out.shape)
- print(conv_out)
```

```
(1, 5, 5, 6)
[[[[3.49173024 3.31086458 3.75857766 3.33879456 3.17262291 4.11346509]
   [3.56939673 5.03026587 4.75123785 3.92601933 4.17798047 5.27678807]
   [3.77303838 5.15126526 4.80435078 3.61129042 4.00872591 5.6429784 ]
   [3.94984694 4.3637585  4.87034207 3.95167282 3.78825109 4.57290978]
   [2.39734615 2.74659033 2.29896866 1.99326475 2.02521136 2.32636141]]

  [[5.01320758 4.65649684 5.16195139 4.21242487 3.59058036 5.14783786]
   [6.63213036 6.48295966 7.31015375 6.28463148 5.33571209 7.62450745]
   [5.53116807 5.62935354 6.68716665 5.06360893 5.07468064 8.09372295]
```

정리

- 4차원 이미지 데이터와 필터 가중치를 행렬 연산을 이루기 위하여
- 각각 2차원 데이터로 만들고,
- 이들을 합성곱 연산(행렬 연산)하면 2차원 데이터가 출력된다
- 그리고 신경망의 다음 계층에서 이미지에 대한 처리가 적절히 이루어지도록
- 다시 4차원 데이터로 reshape한다



이후 단계

- 이후에는 풀링 레이어를 적용하는 것은 물론,
- 일반적인 신경망에서와 같이 활성화 함수의 적용,
- 그리고 추가 Convolution 레이어 적용을 통해 보다 추상적인 특징을 학습할 수 있게 한다
- 또한 네트워크의 마지막 부분에 Dense 층을 두어 분류 클래스의 확률을 계산하여
- 전체 CNN 아키텍처를 구성한다

- 텐서플로, 케라스, 파이토치의 Convolution 레이어를 구현한 Conv2D와 같은 클래스는
- 이미지 데이터의 처리에 최적화되어 설계되어 있으므로
- 이들을 활용하여 이미지 분류, 객체 탐지 등의 다양한 이미지 처리를 수행한다

- 단, 여기서의 구현은 한 번의 처리를 위한 핵심적인 부분을 보여주는 것이고
- 실제로 사용하는 합성곱 레이어는 추가적인 설정과 최적화가 필요하다