강원지역혁신플랫폼

기계학습

Machine Learning

풀링 계층, im2col, col2im 데이터 전개의 개념

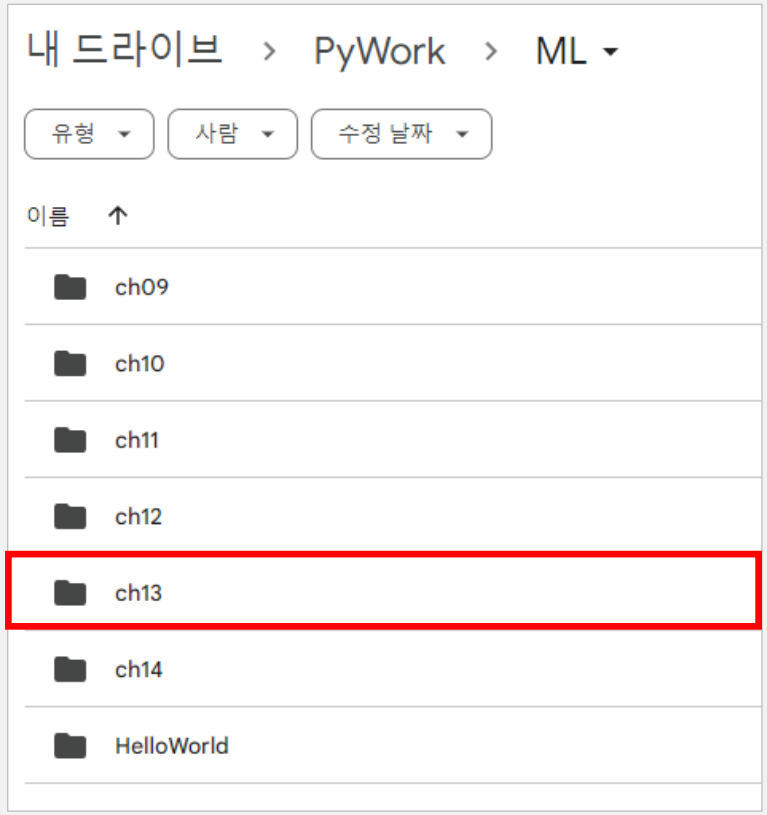
학습목표

- 풀링 계층, im2col, col2im 데이터 전개의 개념을 이해할 수 있습니다.

01 | 실습

(권장) 아래와 같은 경로에 실행 소스가 존재하면 환경 구축 완료

- 구글 드라이브 "PyWork > ML" 폴더로 이동함
  - 아래의 [ch13] 폴더를 클릭하면 됨



01 | 실습

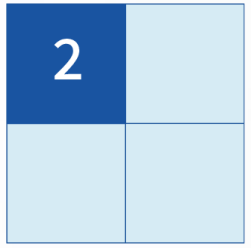
- “ML > ch13 >” 폴더를 클릭함
  - 아래의 [ch13_03_풀링계층, im2col, col2im.ipynb] 스크립트를 클릭함

풀링 계층

- 풀링은 세로(height) · 가로(width)방향의 공간을 줄이는 연산임
  - 예를 들어 아래 그림과 같이 2 x 2 영역을 원소 하나로 집약하여 공간 크기를 줄임

1	2	1	0
0	1	2	3
3	0	1	2
2	4	0	1

Max 풀링



02 | 풀링 계층

아래 그림은 2 x 2 최대 풀링(Max pooling)을 스트라이드 2로 처리하는 순서임

✦ 최대 풀링은 최댓값(Max)을 구하는 연산으로, '2 x 2'는 대상 영역의 크기를 뜻함

› 즉, 2 x 2 최대 풀링은 아래 그림과 같이 2 x 2 크기의 영역에서 가장 큰 원소 하나를 꺼냄

최대 풀링의 처리 순서

02 | 풀링 계층

- 아래 그림에서 스트라이드는 2로 설정했으므로 2 x 2 윈도우가 원소 두 칸 간격으로 이동함
  - 풀링의 윈도우 크기와 스트라이드는 같은 값으로 설정하는 것이 보통임

최대 풀링의 처리 순서

02 | 풀링 계층

- 예를 들어 윈도우가 3 X 3이면 스트라이드는 3으로 설정함
  - 만약, 윈도우가 4 X 4이면 스트라이드를 4로 설정함

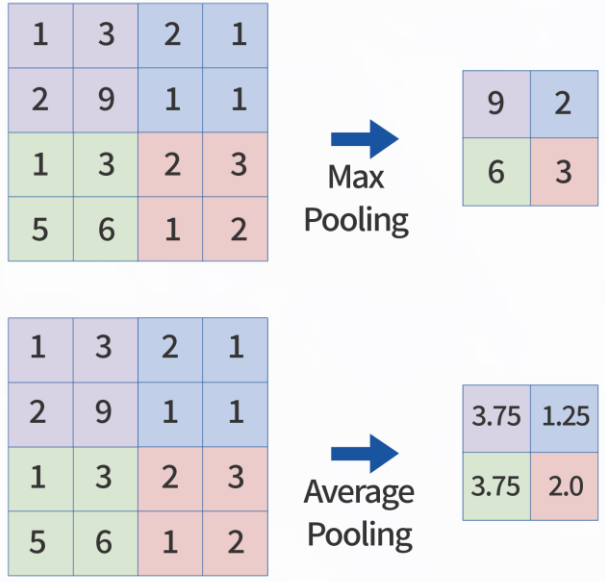
1	2	1	0	1	0
0	1	2	3	2	4
3	0	1	2	1	2
2	4	0	1	0	1
3	0	1	2	1	6
2	5	0	1	0	1

Max 풀링

3	4
5	6

02 | 풀링 계층

- 풀링은 최대 풀링(Max pooling, 맥스 풀링) 외에도 평균 풀링(Average pooling) 등이 있음
  - 최대 풀링은 대상 영역에서 최댓값을 취하는 연산임
  - 평균 풀링은 대상 영역의 평균을 계산함
  - 이미지 인식 분야에서는 주로 최대 풀링을 사용함



최대 풀링과 평균 풀링의 예

02 | 풀링 계층

- 다음은 아래 그림과 같이 처음에 크기가 (4, 4)인 입력 데이터에 스트라이드 2를 적용하여 최대 풀링을 수행하는 파이썬 코드로 구현해 보자.
  - 스트라이드가 2이므로 2x2의 필터를 사용함
  - 아래의 그림에서 윈도우는 파란색 2 x 2 부분임
    - 여기서 윈도우는 두 칸씩 이동함

1	2	1	0
0	1	2	3
3	0	1	2
2	4	0	1

Max 풀링

2	3
4	2

02 | 풀링 계층

다음은 앞의 최대 풀링을 구현한 파이썬 코드이다.

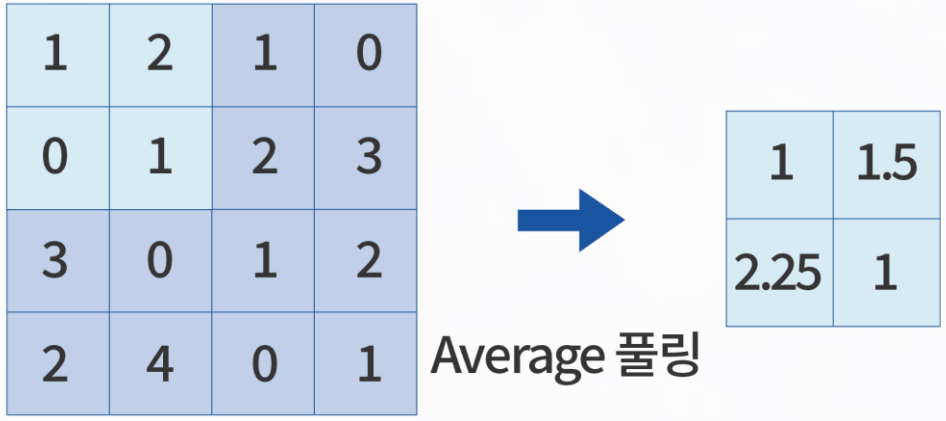
- 실행결과 아래와 같이 최대 풀링이 잘 적용된 것을 볼 수 있음

```
# 입력 데이터 (4x4)
input_matrix = np.array([
    [1, 2, 1, 0],
    [0, 1, 2, 3],
    [3, 0, 1, 2],
    [2, 4, 0, 1] ])

# 필터 크기 및 스트라이드
filter_size = 2
stride = 2

# 출력 크기 계산
output_height = (input_matrix.shape[0] - filter_size) // stride + 1
output_width = (input_matrix.shape[1] - filter_size) // stride + 1

# 출력 행렬 초기화
output_matrix = np.zeros((output_height, output_width))

# 최대 풀링 연산 수행
for i in range(output_height):
    for j in range(output_width):
        # 입력 행렬의 부분 행렬 추출
        sub_matrix = input_matrix[i*stride:i*stride+filter_size, j*stride:j*stride+filter_size]
        # 최대값을 출력 행렬에 저장
        output_matrix[i, j] = np.max(sub_matrix)

print("입력 행렬:\n", input_matrix)
print("출력 행렬 (Max Pooling 결과):\n", output_matrix)
```

```
입력 행렬:
 [[1 2 1 0]
 [0 1 2 3]
 [3 0 1 2]
 [2 4 0 1]]
출력 행렬 (Max Pooling 결과):
 [[2. 3.]
 [4. 2.]]
```

02 | 풀링 계층

- 다음은 아래 그림과 같이 처음에 크기가 (4, 4)인 입력 데이터에 스트라이드 2를 적용하여 평균 풀링을 수행하는 파이썬 코드로 구현해 보자.
  - 스트라이드가 2이므로 2x2의 필터를 사용함
  - 아래의 그림에서 윈도우는 파란색 2 x 2 부분임
    - 여기서 윈도우는 두 칸씩 이동함



02 | 풀링 계층

다음은 앞의 평균 풀링을 구현한 파이썬 코드이다.

- 실행결과 아래와 같이 평균 풀링이 잘 적용된 것을 볼 수 있음

```
# 입력 데이터 (4x4)
input_matrix = np.array([
    [1, 2, 1, 0],
    [0, 1, 2, 3],
    [3, 0, 1, 2],
    [2, 4, 0, 1] ])

# 필터 크기 및 스트라이드
filter_size = 2
stride = 2

# 출력 크기 계산
output_height = (input_matrix.shape[0] - filter_size) // stride + 1
output_width = (input_matrix.shape[1] - filter_size) // stride + 1

# 출력 행렬 초기화
output_matrix = np.zeros((output_height, output_width))

# 평균 풀링 연산 수행
for i in range(output_height):
    for j in range(output_width):
        # 입력 행렬의 부분 행렬 추출
        sub_matrix = input_matrix[i*stride:i*stride+filter_size, j*stride:j*stride+filter_size]
        # 부분 행렬의 평균값을 출력 행렬에 저장
        output_matrix[i, j] = np.mean(sub_matrix)

print("입력 행렬:\n", input_matrix)
print("출력 행렬 (Average Pooling 결과):\n", output_matrix)
```

```
입력 행렬:
 [[1 2 1 0]
 [0 1 2 3]
 [3 0 1 2]
 [2 4 0 1]]
출력 행렬 (Average Pooling 결과):
 [[1.   1.5 ]
 [2.25 1.  ]]
```

03 | 풀링 계층의 특징

풀링 계층의 특징

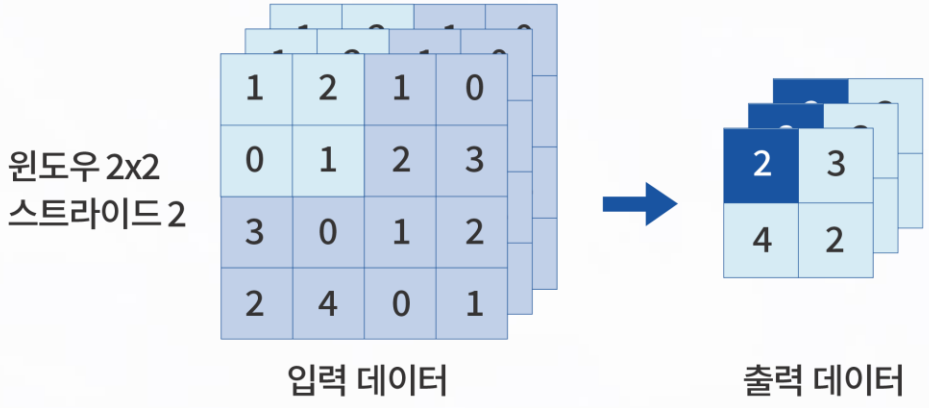
- 풀링 계층의 특징은 다음과 같음
  1. 학습해야 할 매개변수가 없음
  2. 채널 수가 변하지 않음
  3. 입력의 변화에 영향을 적게 받음(강건함)

03 | 풀링 계층의 특징

1 학습해야 할 매개변수가 없음

✦ 풀링 계층은 합성곱 계층과 달리 **학습해야 할 매개변수**가 **없음**

› 풀링은 **대상 영역**에서 **최댓값**(Max)이나 **평균**(Average)을 **취하는 명확한 처리**이므로 **특별히 학습할 것**이 **없음**

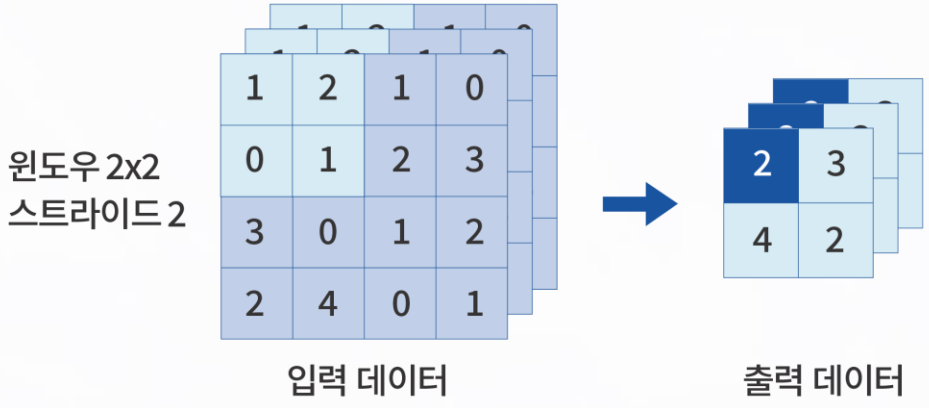


최대 풀링 (윈도우 2x2, 스트라이드 2)의 예

2 채널 수가 변하지 않음

✦ 풀링 연산은 입력 데이터의 채널 수 그대로 출력 데이터로 내보냄

› 아래의 그림처럼 채널마다 독립적으로 계산하기 때문임

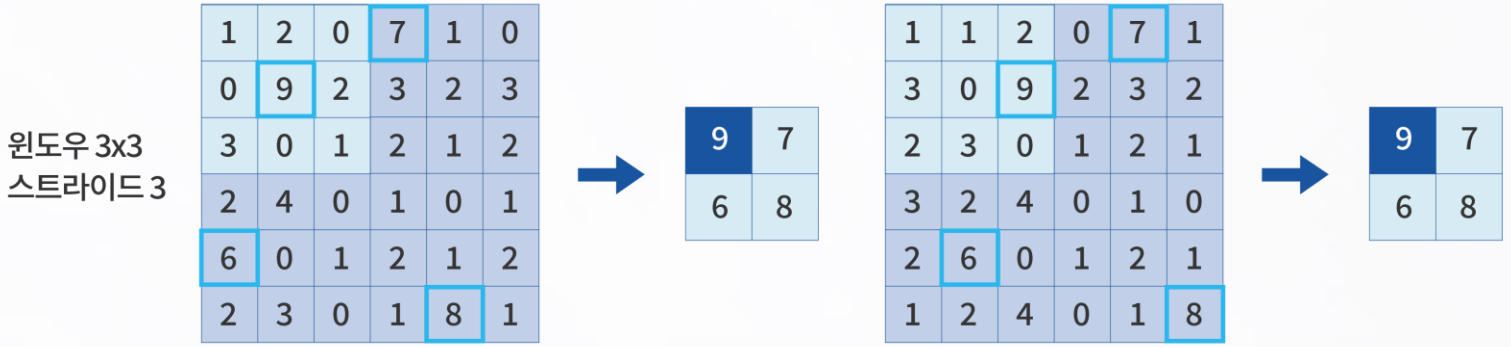


최대 풀링 (윈도우 2x2, 스트라이드 2)의 예

03 | 풀링 계층의 특징

3 입력의 변화에 영향을 적게 받음(강건함)

- 입력 데이터가 조금 변해도 풀링의 결과는 잘 변하지 않음
  - 아래의 그림처럼 입력 데이터의 차이를 풀링이 흡수해 사라지게 하는 모습을 보여줌
    - 아래의 경우 데이터가 오른쪽으로 한 칸씩 이동함

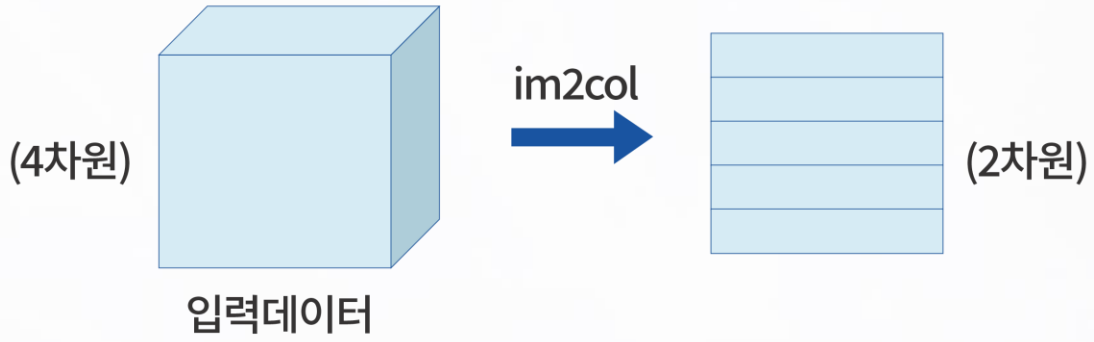


입력 데이터가 가로로 1원소만큼 어긋나도 출력은 같음 (데이터에 따라서는 다를 수도 있음)

04 | im2col (Image to column) 데이터 전개

im2col 메서드로 데이터 전개(4차원 데이터를 2차원으로 전개)

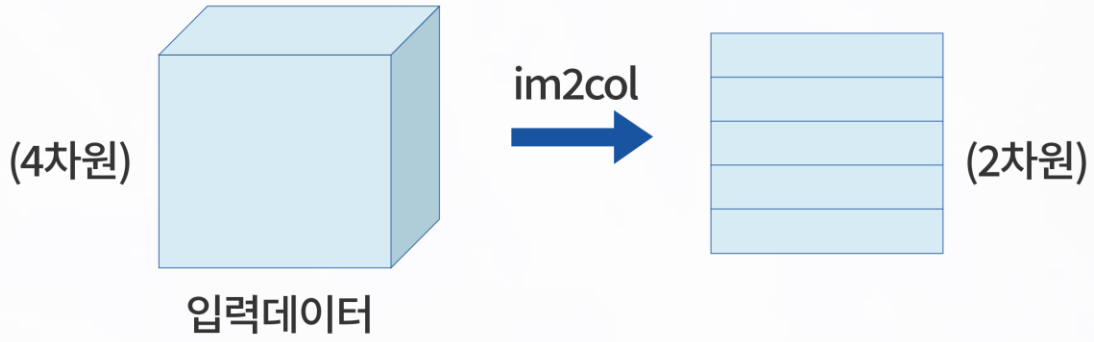
- 합성곱 연산을 곧이곧대로 구현하려면 for 문을 겹겹이 써야 함
  - 또한, 넘파이에 for 문을 사용하여 접근하면 성능이 떨어진다는 단점이 있음
    - 넘파이 원소에 접근할 때 for 문을 사용하지 않는 것이 바람직함



(대략적인) im2col의 동작

04 | im2col (Image to column) 데이터 전개

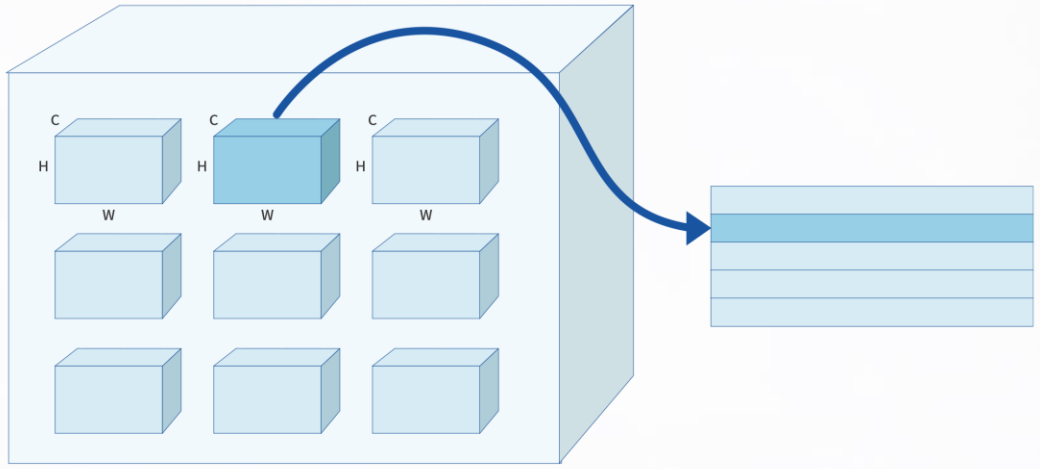
- im2col(images to column, 이미지에서 행렬로)함수로 이 문제를 해결할 수 있음
  - im2col은 입력 데이터를 필터링(가중치 계산)하기 좋게 전개하는(펼치는)함수임
    - 아래의 그림은 대략적인 im2col 함수의 동작임
      - 4차원 입력 데이터에 im2col을 적용하면 2차원 행렬로 바꿈



(대략적인) im2col의 동작

04 | im2col (Image to column) 데이터 전개

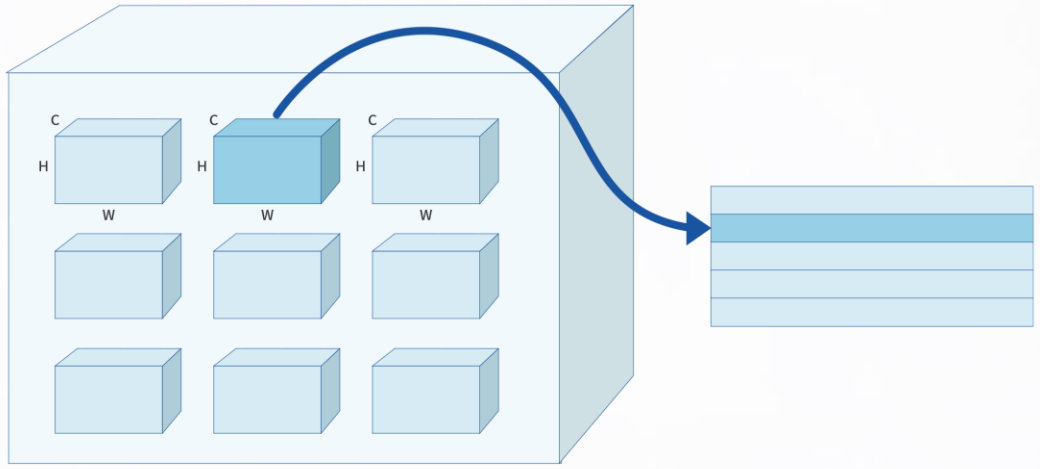
- 아래의 그림처럼 im2col은 필터링하기 좋게 입력 데이터를 전개함
  - 아래의 그림은 입력 데이터에서 필터를 적용하는 영역(3차원 블록)을 한 줄로 늘어놓음
    - 이 전개를 입력 데이터에서 필터를 적용하는 모든 영역에서 수행하는 것이 im2col 메서드임



필터 적용 영역을 앞에서부터 순서대로 한 줄로 펼침

04 | im2col (Image to column) 데이터 전개

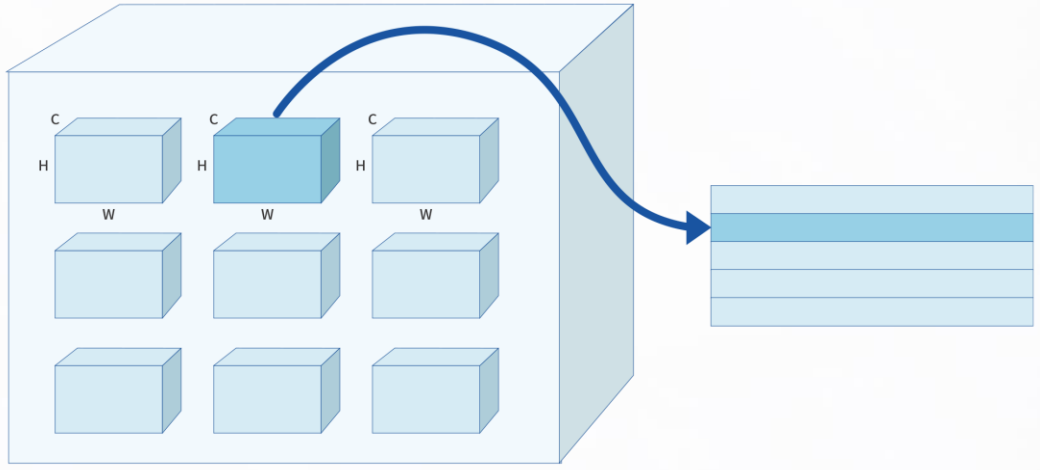
- 아래의 그림에서는 스트라이드를 크게 잡아 필터의 적용 영역이 겹치지 않도록 했음
  - 하지만, 실제 상황에서는 영역이 겹치는 경우가 대부분임



필터 적용 영역을 앞에서부터 순서대로 한 줄로 펼침

04 | im2col (Image to column) 데이터 전개

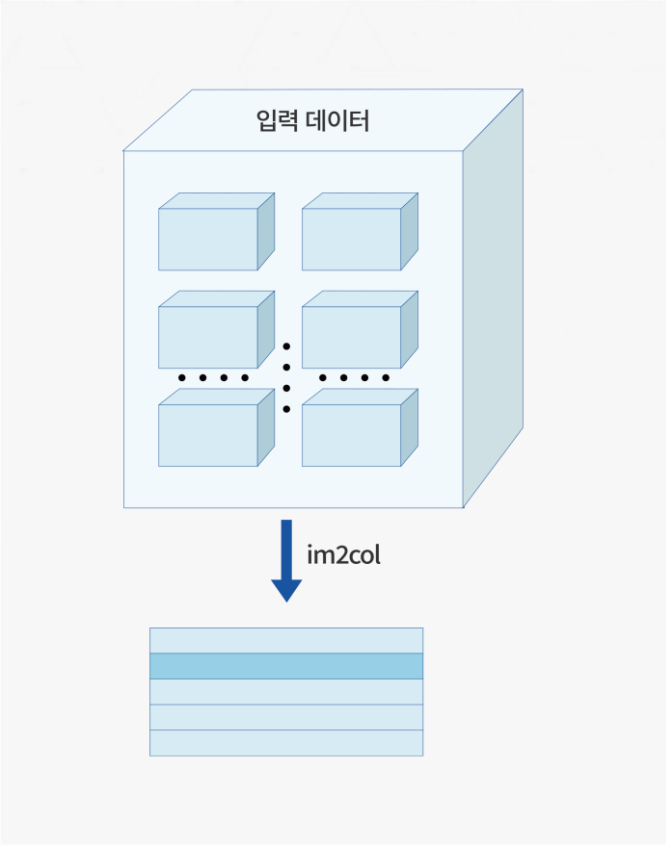
- 필터 영역이 겹치게 되면 im2col로 전개한 후의 원소 수가 블록의 원소 수보다 많아짐
  - 하여, im2col을 사용해 구현하면 메모리를 더 많이 소비하는 단점이 있음
    - 하지만 컴퓨터는 큰 행렬을 묶어서 계산하는 데 탁월함
    - 예를 들어 선형 대수 라이브러리 등은 행렬 계산에 고도로 최적화되어 큰 행렬의 곱셈을 빠르게 계산할 수 있음



필터 적용 영역을 앞에서부터 순서대로 한 줄로 펼침

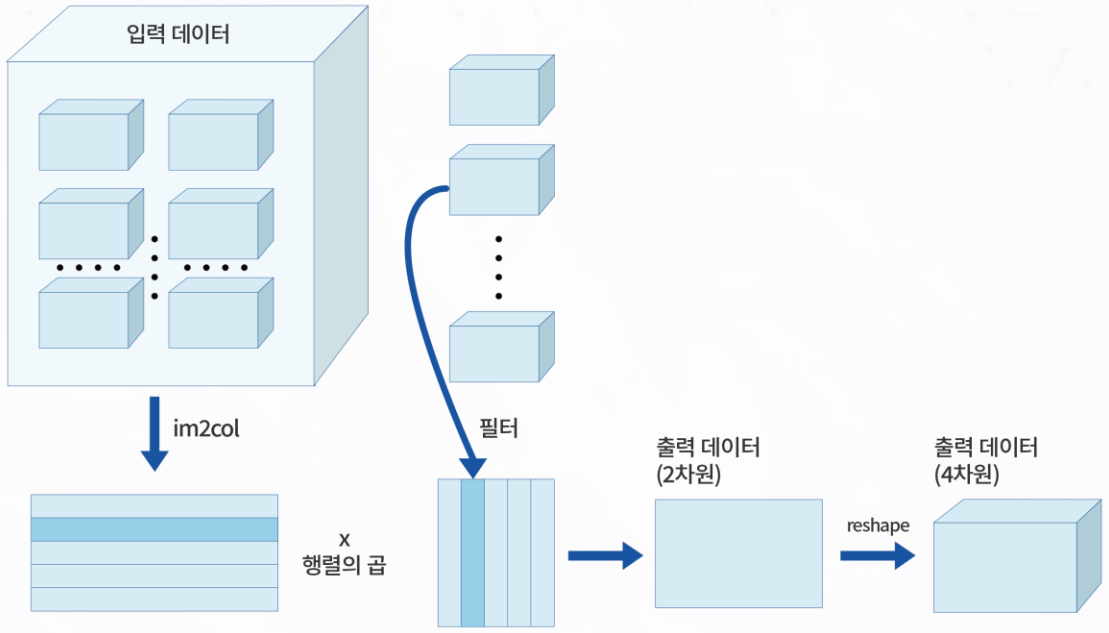
04 | im2col (Image to column) 데이터 전개

아래의 그림처럼 im2col은 'image to column' 즉 '이미지에서 행렬로' 라는 뜻임



04 | im2col (Image to column) 데이터 전개

- 아래의 그림은 **합성곱 계층**의 **구현 흐름**임
  - ✦ im2col로 **입력 데이터**를 **전개**한 **다음**에는 **합성곱 계층**의 **필터**(가중치)를 **1열**로 **전개**함
    - › 그리고, **두 행렬**의 **곱**을 **계산**하면 됨



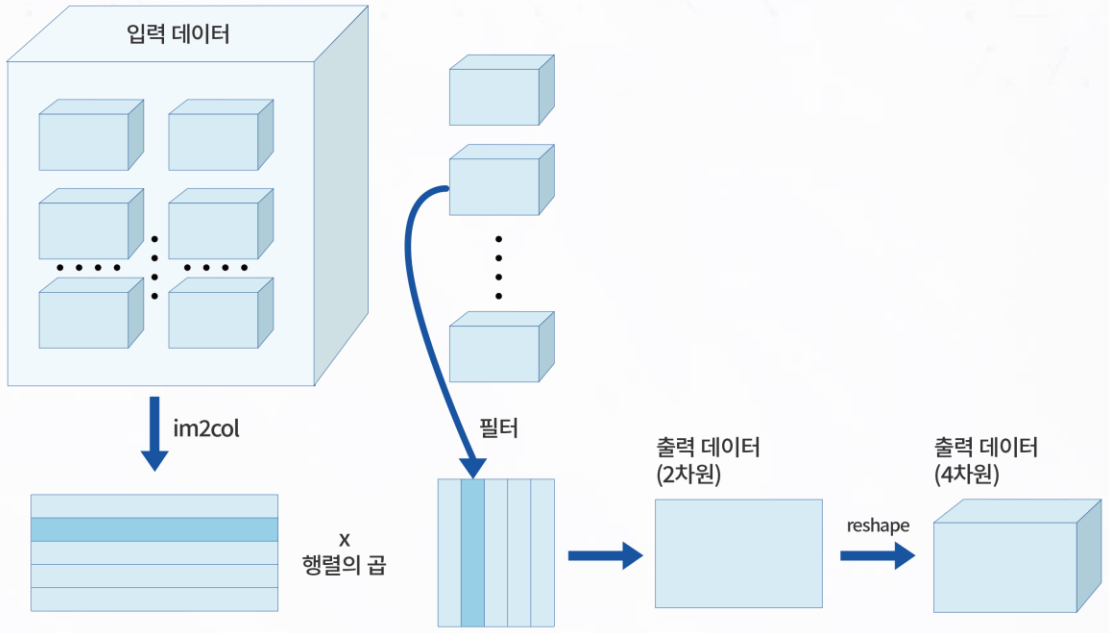
합성곱 계층의 구현 흐름

① 필터를 세로로 1열로 전개 ② im2col이 전개한 데이터와 행렬곱을 계산

③ 마지막으로 출력 데이터를 변형함

04 | im2col (Image to column) 데이터 전개

- im2col 방식으로 출력한 결과는 2차원 행렬임
  - ✦ CNN은 데이터를 4차원 배열로 저장하므로 2차원인 출력 데이터를 4차원으로 변형함



합성곱 계층의 구현 흐름

① 필터를 세로로 1열로 전개 ② im2col이 전개한 데이터와 행렬곱을 계산

③ 마지막으로 출력 데이터를 변형함

04 | im2col (Image to column) 데이터 전개

다음은 im2col 함수의 인터페이스이다.

- im2col() 함수는 '필터 크기', '스트라이드', '패딩'을 고려하여 입력 데이터를 2차원 배열로 전개함

```
im2col(input_data, filter_h, filter_w, stride=1, pad=0)
```

  - input_data – (데이터 수, 채널 수, 높이, 너비)의 4차원 배열로 이뤄진 입력 데이터
  - filter_h – 필터의 높이
  - filter_w – 필터의 너비
  - stride – 스트라이드
  - pad – 패딩

04 | im2col (Image to column) 데이터 전개

다음은 im2col() 함수의 코드이다.

```
def im2col(input_data, filter_h, filter_w, stride=1, pad=0):
    """다수의 이미지를 입력받아 2차원 배열로 변환한다(평탄화).

    Parameters
    ----------
    input_data : 4차원 배열 형태의 입력 데이터(이미지 수, 채널 수, 높이, 너비)
    filter_h : 필터의 높이
    filter_w : 필터의 너비
    stride : 스트라이드
    pad : 패딩

    Returns
    -------
    col : 2차원 배열
    """

    N, C, H, W = input_data.shape
    out_h = (H + 2 * pad - filter_h) // stride + 1
    out_w = (W + 2 * pad - filter_w) // stride + 1

    img = np.pad(input_data, [(0, 0), (0, 0), (pad, pad), (pad, pad)], 'constant')
    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]

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

04 | im2col (Image to column) 데이터 전개

- 다음은 im2col() 함수를 실제로 사용해 테스트한 결과이다.
  - 위 코드에서 두 가지 예를 보여주고 있음
    - 첫 번째는 배치 크기가 1(데이터 1개), 채널은 3개, 높이·너비가 7x7의 데이터임
    - 두 번째는 배치 크기가 10(데이터 10개), 채널은 3개, 높이·너비가 7x7의 데이터임
    - 예제에서는 im2col 함수를 적용한 두 경우 모두 두 번째 차원의 원소는 75개임
      - 이 값은 필터의 원소 수와 같음(=채널 3 * 필터 크기(5x5))
      - 배치 크기가 1일 때는 im2col의 결과의 크기가 (9, 75) 크기의 데이터가 저장됨
      - 배치 크기가 10일 때는 그 10배인 (90, 75) 크기의 데이터가 저장됨

```
# im2col()로 필터 크기, 스트라이드, 패딩을 고려하여 입력 데이터를 2차원 배열로 전개
x1 = np.random.rand(1, 3, 7, 7)              # (데이터 수, 채널 수, 높이, 너비)
col1 = im2col(x1, 5, 5, stride=1, pad=0) # (입력 데이터, 필터 높이, 필터 너비, 스트라이드, 패딩)
print(col1.shape)  # (9, 75)

x2 = np.random.rand(10, 3, 7, 7)             # (데이터 수, 채널 수, 높이, 너비)
col2 = im2col(x2, 5, 5, stride=1, pad=0)
print(col2.shape)  # (90, 75)
```

05 | col2im (Column to Image) 데이터 전개

col2im 메서드로 데이터 전개 (2차원 데이터를 4차원 데이터로 전개)

- im2col 메서드의 반대
  - ✦ 2차원 배열을 입력 받아 다수의 이미지 묶음으로 변환함

05 | col2im (Column to Image) 데이터 전개

다음은 col2im() 함수의 코드이다.

```
def col2im(col, input_shape, filter_h, filter_w, stride=1, pad=0):
    """(im2col과 반대) 2차원 배열을 입력받아 다수의 이미지 묶음으로 변환한다.

    Parameters
    ----------
    col : 2차원 배열(입력 데이터)
    input_shape : 원래 이미지 데이터의 형상（예 : (10, 1, 28, 28)）
    filter_h : 필터의 높이
    filter_w : 필터의 너비
    stride : 스트라이드
    pad : 패딩

    Returns
    -------
    img : 변환된 이미지들
    """
    N, C, H, W = input_shape
    out_h = (H + 2 * pad - filter_h) // stride + 1
    out_w = (W + 2 * pad - filter_w) // stride + 1
    col = col.reshape(N, out_h, out_w, C, filter_h, filter_w).transpose(0, 3, 4, 5, 1, 2)
    img = np.zeros((N, C, H + 2 * pad + stride - 1, W + 2 * pad + stride - 1))
    for y in range(filter_h):
        y_max = y + stride * out_h
        for x in range(filter_w):
            x_max = x + stride * out_w
            img[:, :, y:y_max:stride, x:x_max:stride] += col[:, :, y, x, :, :]

    return img[:, :, pad:H + pad, pad:W + pad]
```

05 | col2im (Column to Image) 데이터 전개

- 다음은 im2col(), col2im() 함수를 실제로 사용해 테스트한 결과이다.
  - 입력 데이터는 배치 크기가 1(데이터 1개), 채널은 3개, 높이·너비가 7x7의 데이터임
    - 예제에서 im2col 함수를 적용한 경우 두 번째 차원의 원소는 75개임
      - 이 값은 필터의 원소 수와 같음(=채널 3 * 필터 크기(5x5))
      - 이 값들을 col2im 함수에 적용한 경우 입력 데이터의 형상 (1, 3, 7, 7)과 동일한 것을 알 수 있음

```
# 2차원 배열을 입력받아 다수의 이미지 묶음으로 변환한다.
x1 = np.random.rand(1, 3, 7, 7)             # (데이터 수, 채널 수, 높이, 너비)
col1 = im2col(x1, 5, 5, stride=1, pad=0)  # (입력 데이터, 필터 높이, 필터 너비, 스트라이드, 패딩)
print(col1.shape)  # (9, 75)

# (2차원 배열, 원래 이미지 데이터의 형상, 필터의 높이, 필터의 너비, 스트라이드, 패딩)
img1 = col2im(col1, x1.shape, 5, 5, stride=1, pad=0)
print(img1.shape) # (1, 3, 7, 7)
```