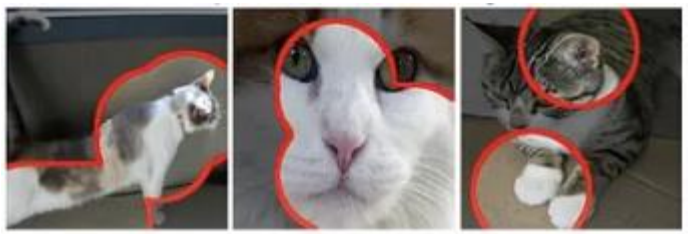
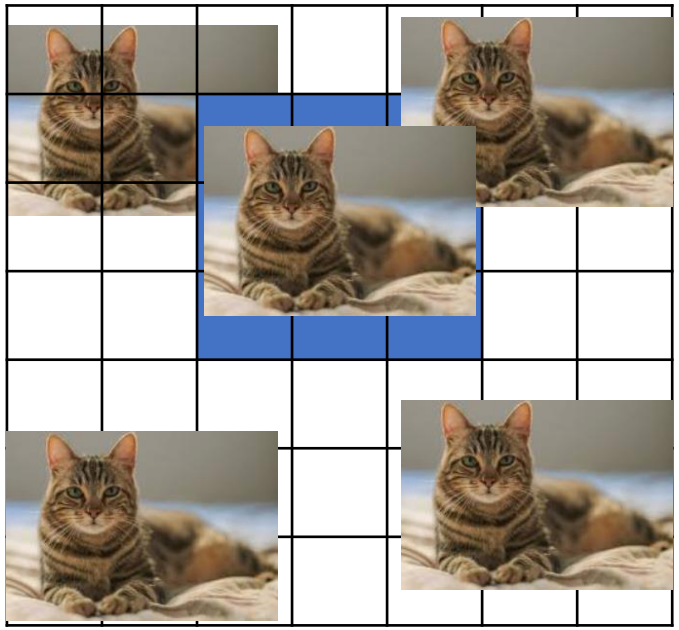
이미지 합성곱과 필터

최석재 *lingua@naver.com*

이미지 합성곱

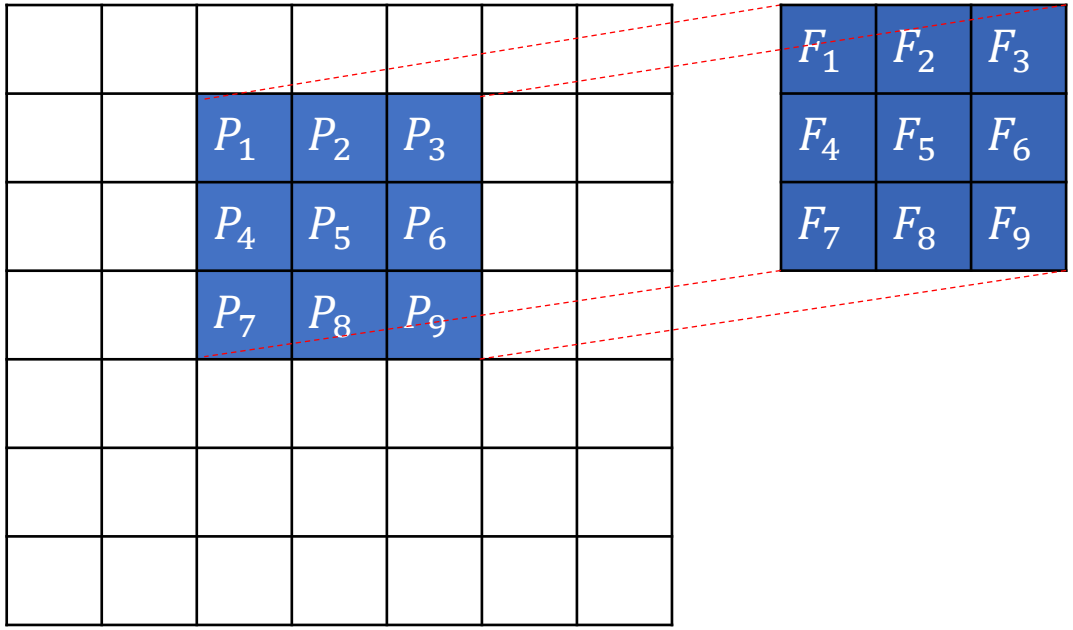
- 합성곱 계층은 필터를 사용해 데이터의 특징을 추출하므로
- 이미지 데이터가 가진 지역적인 패턴을 인식할 수 있다
- 특정 객체의 특징을 파악할 수 있기 때문에 이미지의 어느 위치에 있든지
- 또는 다른 포즈를 취해도 인식할 수 있다





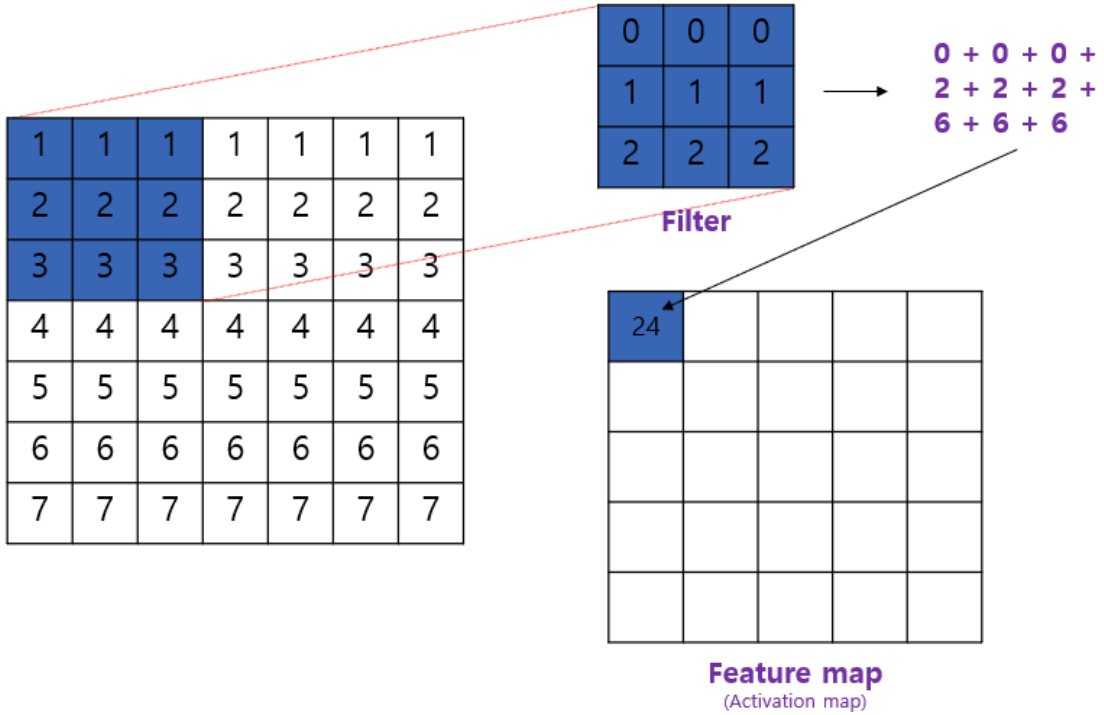
이미지 필터

- 필터는 입력 데이터에서 특징을 추출하는 주요 요소
- 이미지의 1개 채널은 2차원 데이터이므로, 필터도 2차원이어야 한다
- 입력 데이터의 모든 위치에서 동일한 필터를 사용한다(가중치 공유)
- 따라서 학습에 필요한 가중치 수가 감소해 과대적합을 방지한다



이미지 필터 적용

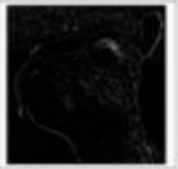
- 필터의 사이즈는 정해진 것은 없으나 3x3 또는 5x5 필터를 많이 사용한다
- 필터를 적용할 수 있는 이미지 영역에 대해서 각 요소끼리 곱한 후 더한다
- 이 과정을 일정 간격으로 이동하면서 수행해 특징 맵(Feature map)을 생성한다



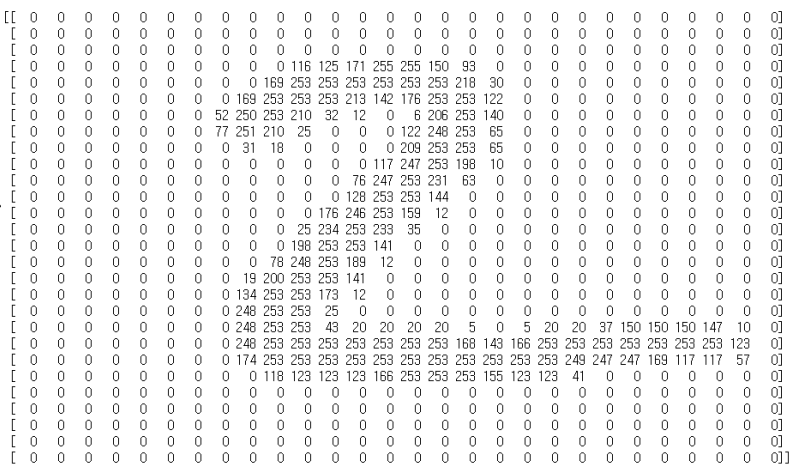
다양한 필터 사용

- 하나의 계층에 여러 개의 필터를 사용한다
- 이미지가 갖는 다양한 특징이 추출되어
- 이미지의 여러 특징을 인식하고 분류할 수 있다

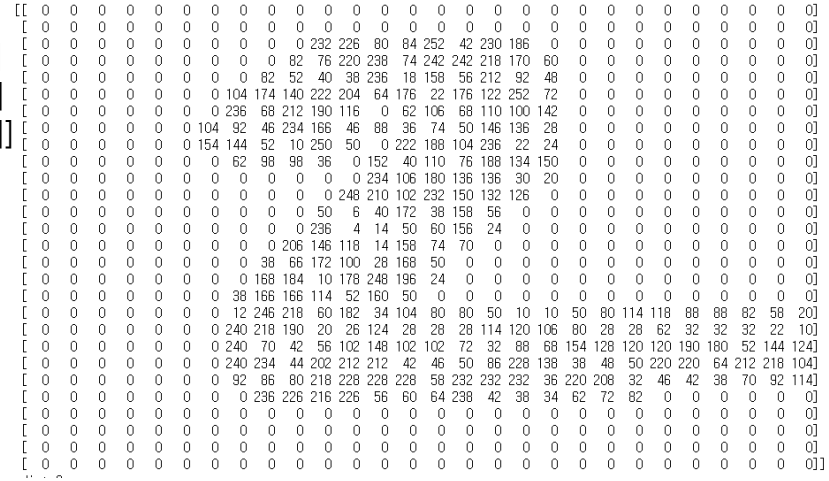
X

Operation	Filter	Convolved Image
Identity	$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$	
Edge detection	$\begin{bmatrix} 1 & 0 & -1 \\ 0 & 0 & 0 \\ -1 & 0 & 1 \end{bmatrix}$	
	$\begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$	
	$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$	
Sharpen	$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$	
Box blur (normalized)	$\frac{1}{9}\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$	
Gaussian blur (approximation)	$\frac{1}{16}\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$	

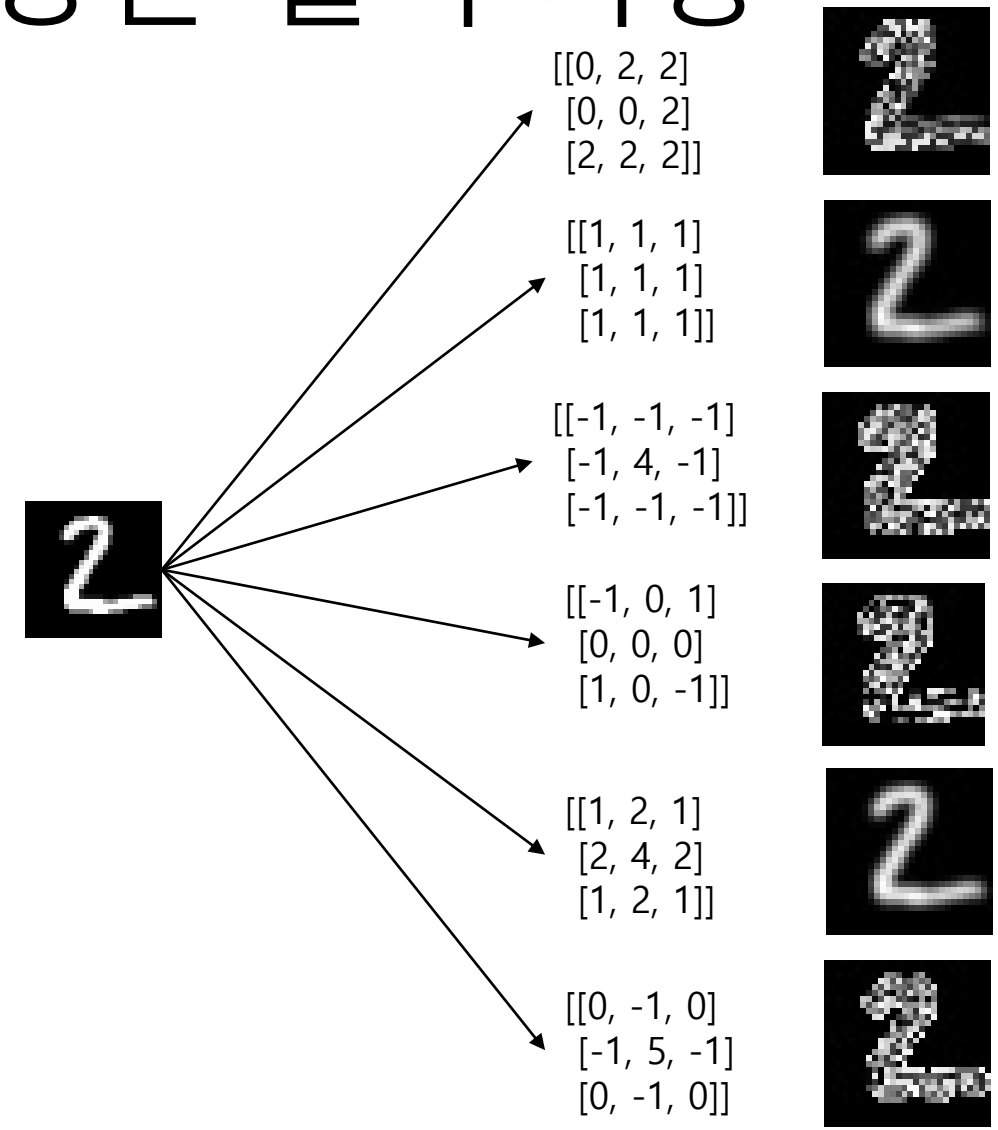
다양한 필터 사용



[[0, 2, 2]
[0, 0, 2]
[2, 2, 2]]

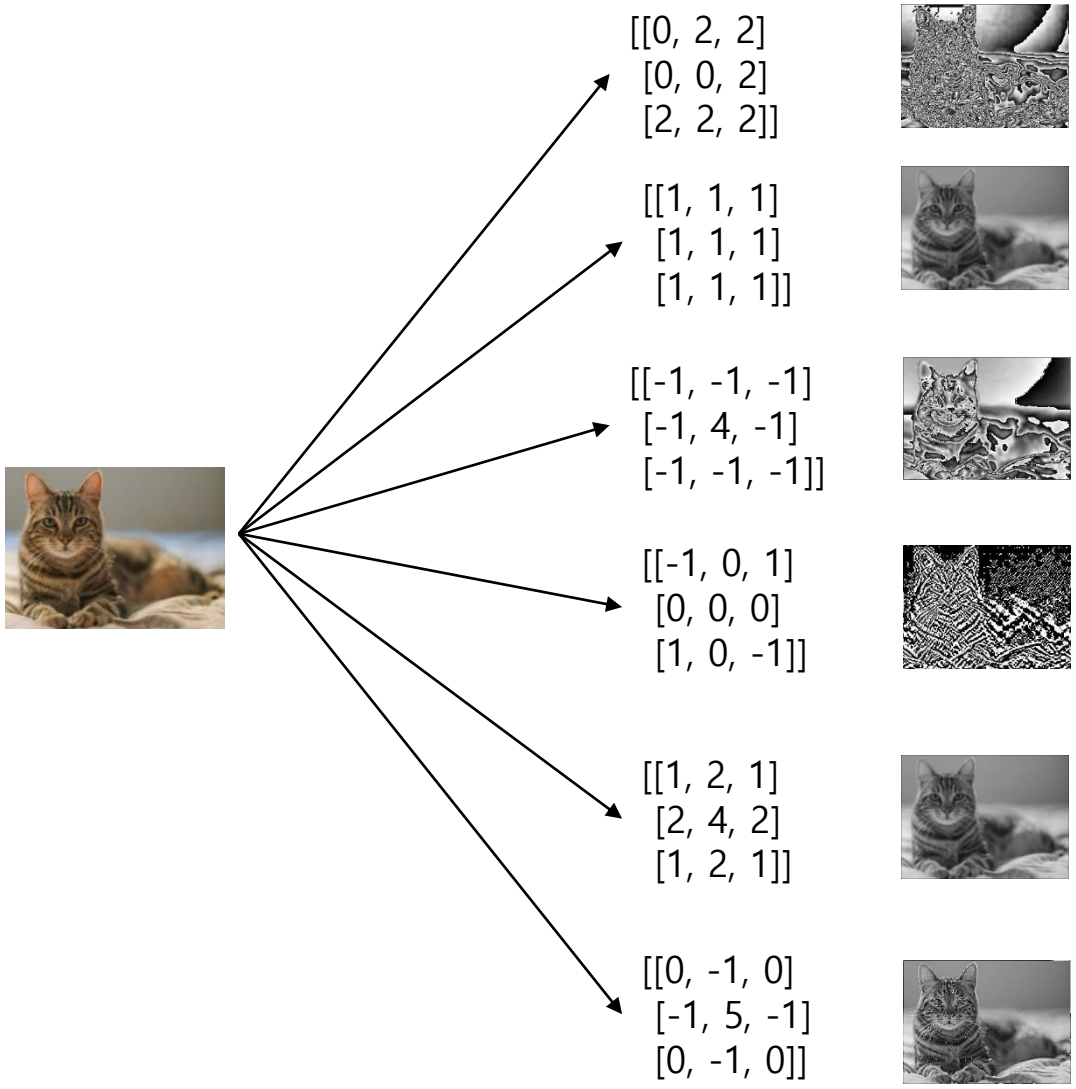


다양한 필터 사용



- 다양한 필터를 사용하여
  이미지의 형상을 다양한 측면으로 파악한다

다양한 필터 사용



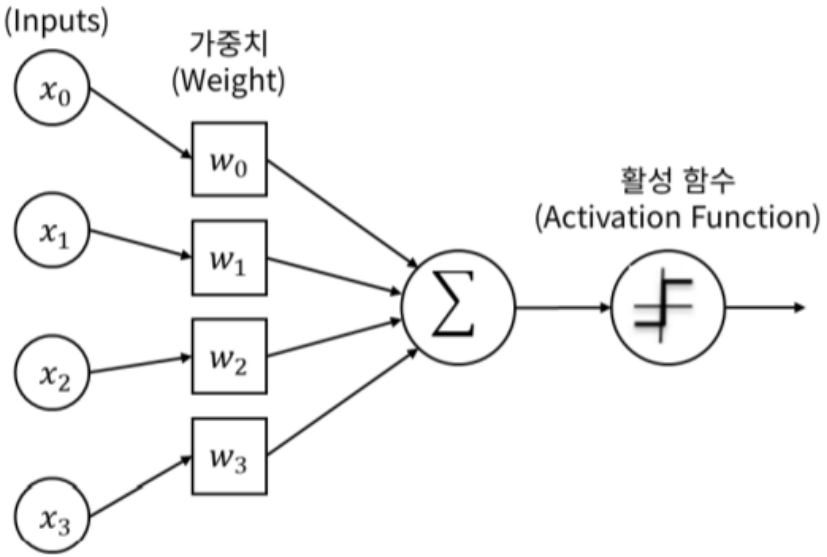
- 다양한 필터를 사용하여
  이미지의 형상을 다양한 측면으로 파악한다

필터의 초기값은 랜덤으로 다양하게 생성하게 된다
보통 32개나 64개의 필터를 사용하고,
이 결과들은 출력의 깊이(즉, 채널)와 함께 3차원을 형성한다
학습이 진행됨에 따라 이 값들은 최적 결과를 내도록 조정된다
(필터 가중치)

필터 결과 더하기

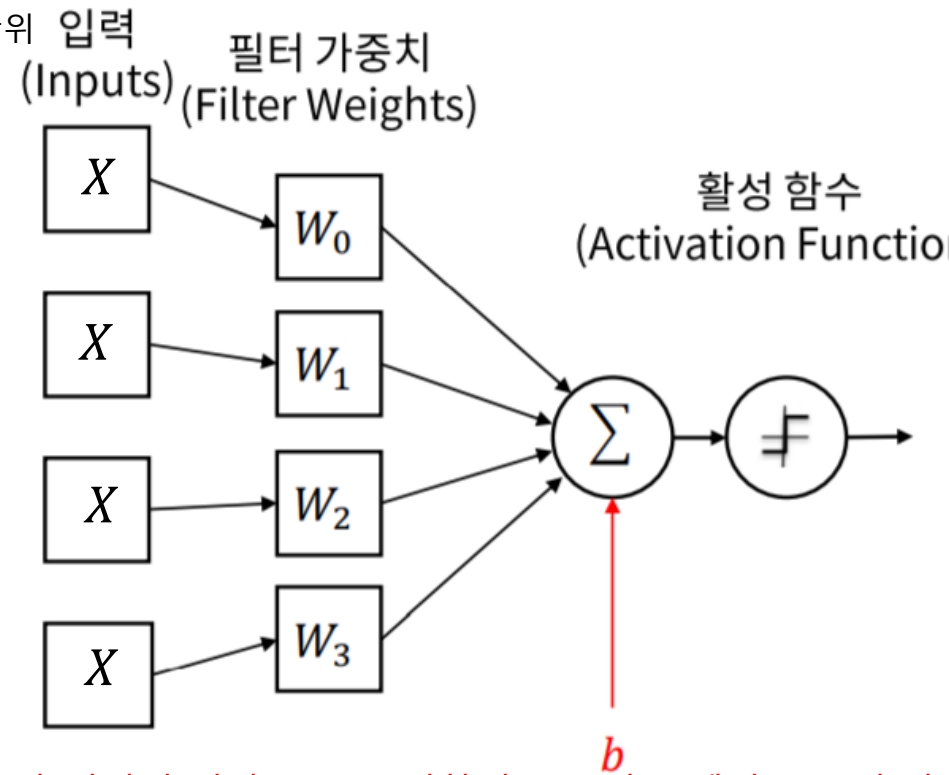
- 각 필터의 결과는 모두 더한 뒤, 활성화 함수에 보낸다
- 입력 노드별이 아니라, 이미지별로 가중치가 계산되므로 연산량이 줄어든다

픽셀 단위



기존 방식으로는 이미지가 가지고 있는 각 셀 별로 가중치가 필요하나,
(백만 화소면 백만 개의 개별적인 가중치)
필터 방식에서는 하나의 이미지 내 채널은 필터의 값(가중치)을 공유한다

이미지의 채널 단위



각 필터의 결과를 모두 더하면 공통되는 엣지 부분의 값이 매우 높아져
그 부분이 강조된 값이 나오게 된다

합성곱 함수 구현 (전체)

- import numpy as np
- def Conv2D(img, kernel=None):

결과 이미지 크기 계산 부분

```
    h, w = img.shape                                                    # 이미지의 높이와 너비
    kh, kw = kernel.shape                                               # 커널(필터)의 높이와 너비

    output_height = h - kh + 1                                          # 결과 이미지 높이 계산
    output_width = w - kw + 1                                           # 결과 이미지 너비 계산
    img_out = np.zeros((output_height, output_width), dtype=np.uint8)   # 결과 이미지 초기화
```

합성곱 연산 부분

```
    for y in range(output_height):                                      # 커널의 높이와 너비 수만큼 진행
        for x in range(output_width):
            roi = img[y:y+kh, x:x+kw]                                   # 관심 영역 추출
            filtered = roi * kernel                                     # 추출한 ROI와 커널 사이의 요소별 곱셈 수행
            conv_value = np.abs(np.sum(filtered))                       # 필터링된 값들의 합을 계산한 뒤, 음수 방지 위해 절대값
            img_out[y, x] = np.uint8(conv_value)                        # 0~255의 값을 갖는 uint8 타입으로 변환

    return img_out
```

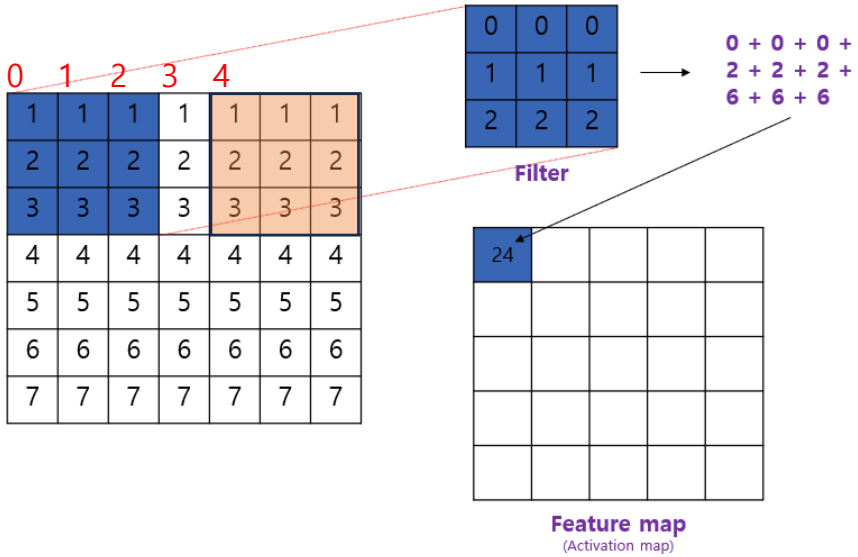
절대값은 사용하지 않아도 되나,
효과적인 엣지 검출을 위해서 사용되기도 한다

결과 이미지 크기 계산 부분

- def Conv2D(img, kernel=None):

```
    h, w = img.shape                                                    # 이미지의 높이와 너비
    kh, kw = kernel.shape                                               # 커널(필터)의 높이와 너비

    output_height = h - kh + 1                                          # 결과 이미지 높이 계산
    output_width = w - kw + 1                                           # 결과 이미지 너비 계산
    img_out = np.zeros((output_height, output_width), dtype=np.uint8)   # 결과 이미지 초기화
```



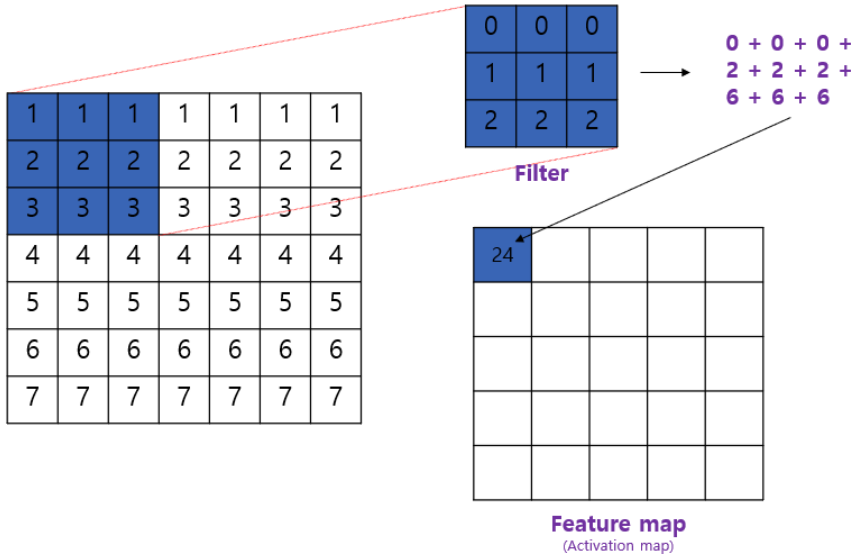
왼쪽과 같이 이미지의 너비 사이즈가 7, 커널의 너비 사이즈가 3인 경우
커널이 이동될 수 있는 위치는 총 5개이다

시작 위치는 (0, **0**)이고,
커널이 마지막 위치할 수 있는 곳은 이미지 사이즈 – 커널 사이즈 위치인 (0, **4**)이므로
커널이 이동될 수 있는 위치를 계산하려면 이미지 사이즈 – 커널 사이즈 + 1을 해준다

높이에 대해서도 동일한 계산을 한다

합성곱 연산 부분

```
for y in range(output_height):                          # 커널이 가능한 모든 위치에서 진행
    for x in range(output_width):
        roi = img[y:y+kh, x:x+kw]                       # 관심 영역 추출
        filtered = roi * kernel                         # 추출한 ROI와 커널 사이의 요소별 곱셈 수행
        conv_value = np.abs(np.sum(filtered))           # 필터링된 값들의 합을 계산한 뒤, 음수 방지 위해 절대값
        img_out[y, x] = np.uint8(conv_value)            # 0~255의 값을 갖는 uint8 타입으로 변환
```



커널이 현재 바라보는 이미지 영역이 관심 영역(Region of Interest).
커널의 높이와 너비만큼 움직인다

'*' 연산자를 이용하면 관심 영역과 커널 사이에 요소별 연산을 수행한다
결과를 모두 합산한 뒤, 현재 (y, x) 위치에 0~255의 양수 값으로 입력한다

구글 드라이브 접속

- 구글 드라이브 접속

- from google.colab import drive
- drive.mount('/content/gdrive')

- 경로 변경
- %cd /content/gdrive/MyDrive/pytest_img/opencv/

시각화

- import cv2
- from google.colab.patches import cv2_imshow # Colab에서 cv2 출력하기 위하여

- img = cv2.imread("mountain.jpg", cv2.IMREAD_GRAYSCALE) # 단채널 이미지로 읽기 위해
- kernel = np.array([[0,0,0], [-1,2,-1], [0,0,0]]) # 커널
- output = Conv2D(img, kernel=kernel) # Conv2D 함수 실행
- cv2_imshow(img)
- cv2_imshow(output)
- cv2.waitKey(0) # 사용자의 반응을 기다리는 함수
- cv2.destroyAllWindows() # 모든 윈도우를 닫는다

시각화

경계 부분이 잘 드러나게 되었다
다른 커널로도 그려본다