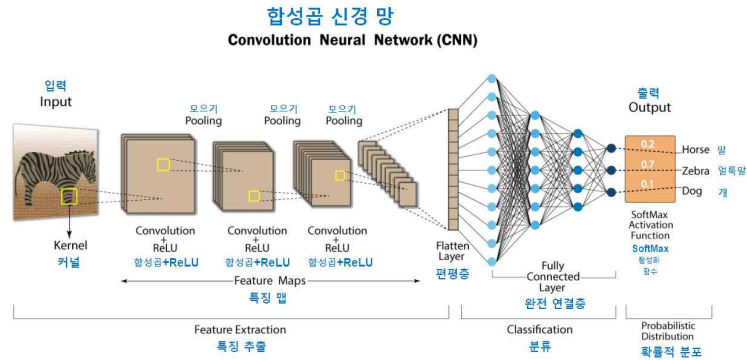


Chapter 05

CNN 알고리즘의 이해와 구현

01 CNN의 순전파 이해와 구현

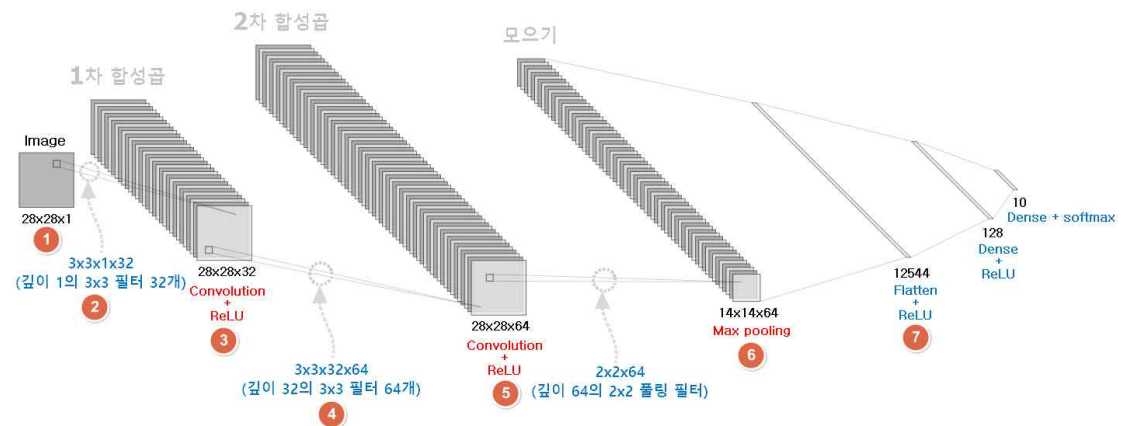
다음은 CNN 형태의 인공 신경망입니다. CNN은 이미지 인식에 뛰어난 인공 신경망으로 이미지의 특징을 뽑아내는 인공 신경망과 분류를 위한 인공 신경망으로 구성됩니다.



이미지의 특징을 뽑아내는 인공 신경망은 전통적인 영상 인식 방법을 접목한 신경망입니다. 전통적인 영상 인식 방법에서는 여러 가지 필터를 이용하여 이미지의 특징을 추출합니다. 필터는 커널이라고도 하며, CNN에서는 학습의 대상이 되는 가중치가 됩니다. 합성곱(Convolution)은 필터를 이용하여 이미지의 특징을 추출하는 연산을 말하며, 모으기(Pooling)는 합성 곱을 이용하여 얻어낸 이미지를 단순화하는 연산을 말합니다. 분류를 위한 인공 신경망은 앞에서 살펴봤던 신경망으로 완전 연결 층 신경망이라고도 합니다. 여기서는 CNN에서 합성 곱(Convolution)과 모으기(Pooling) 연산의 동작을 살펴보고 구현해 봅니다.

01 CNN 신경망 살펴보기

CNN에서는 입력 이미지가 필터를 거쳐 특징을 가진 이미지로 출력되는 합성 곱(Convolution) 과정, 이미지를 단순화하는 모으기(Pooling) 과정, 앞에서 배웠던 완전 연결 층을 거치는 과정으로 구성됩니다. 예를 들어, 다음은 28x28x1 이미지에 대해 합성 곱 1차, 합성 곱 2차, 모으기, 완전 연결 층으로 구성된 CNN 신경망을 나타냅니다.

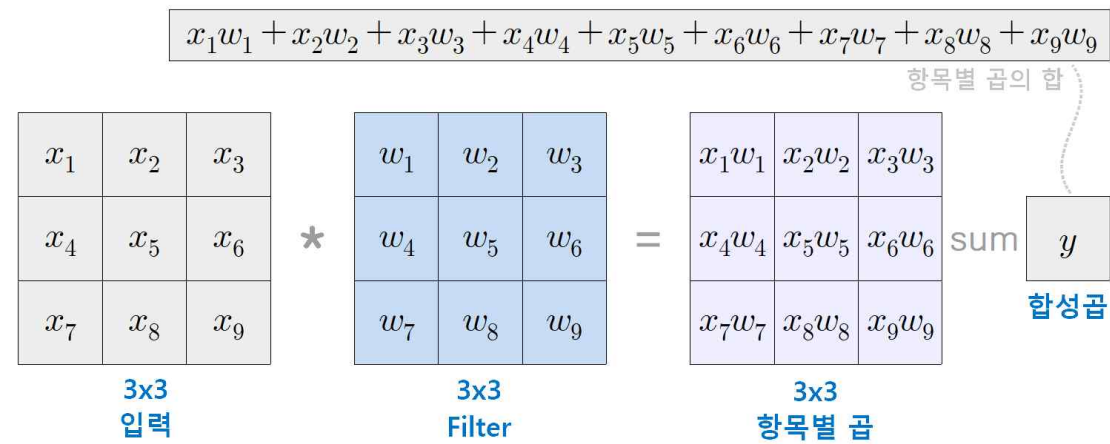


❶ 이 신경망에서 입력 층은 28x28x1의 이미지입니다. CNN에서 입력 이미지는 (세로 픽셀 개수x가로 픽셀 개수x깊이)로 구성됩니다. 여기서는 세로 28픽셀, 가로 28픽셀, 깊이 1의 이미지를 나타냅니다. 깊이는 채널(channel)이라고도 합니다. ❷ 입력 이미지에 필터 1개가 적용되면 특징을 가진 출력 이미지 1개가 나옵니다. 필터의 크기는 1x1, 3x3, 5x5와 같이 같은 크기의 홀수의 곱으로 표현됩니다. 일반적으로 3x3 크기의 필터가 사용됩니다. 입력 이미지에 필터가 적용될 때 필터의 깊이는 반드시 입력 이미지의 깊이와 같아야 합니다. 즉, 합성 곱에 사용하는 필터의 깊이는 입력 이미지의 깊이와 같아야 합니다. 예를 들어, 여기서 사용되는 필터는 3x3x1 크기의 필터가 되어야 합니다. 위 그림에서 28x28x1 이미지는 3x3x1 필터와 합성 곱이 수행된 후, 28x28x1의 특징을 가진 출력 이미지가 됩니다. 즉, 세로 28픽셀, 가로 28픽셀, 깊이 1의 특징을 가진 이미지가 출력됩니다. ❸ 또, 필터의 개수는 출력 이미지의 깊이와 같아야 합니다. 즉, 1차 합성 곱을 거친 출력 이미지의 깊이가 32이므로 1차 합성 곱에 필요한 필터의 개수는 32개가 되어야 합니다. 그래서 1차 합성 곱에 필요한 필터는 3x3x1x32의 크기가 됩니다. 1차 합성 곱을 거친 28x28x32 이미지는 세로 28픽셀, 가로 28픽셀, 깊이 32인 하나의 특징 이미지가 됩니다. ❹ 2차 합성 곱에서도 입력 이미지에 필터 1개가 적용되면 특징을 가진 출력 이미지 1개가 나옵니다. 입력 이미지의 깊이가 32이므로 필터의 크기는 3x3x32가 되어야 합니다. ❺ 2차 합성 곱을 거친 출력 이미지의 깊이가 64이므로 2차 합성 곱에 필요한 필터의 개수는 64개가 되어야 합니다. 그래서 2차 합성 곱에 필요한 필터는 3x3x32x64의 크기가 됩니다. 2차 합성 곱을 거친 28x28x64 이미지는 세로 28픽셀, 가로 28픽셀, 깊이 64인 하나의 특징 이미지가 됩니다. ❻ 모으기 과정은 입력 이미지의 가로, 세로 크기를 줄이는 과정으로 일반적으로 가로, 세로 크기를 각각 1/2로 줄이게 됩니다. 그래서 가로 14, 세로 14, 깊이 64가 됩니다. ❼ Flatten 과정에서는 모으기 과정을 거친 입체 이미지를 1차원 형태로 변형하는 작업을 수행합니다. 즉, 모으기 과정의 입체 이미지를 구성하는 모든 픽셀은 1차원 형태로 나열됩니다. 그래서 14x14x64 = 12544가 됩니다. 이후의 구성은 앞에서 배웠던 신경망의 구성과 같습니다.

02 3x3 입력 : filter size

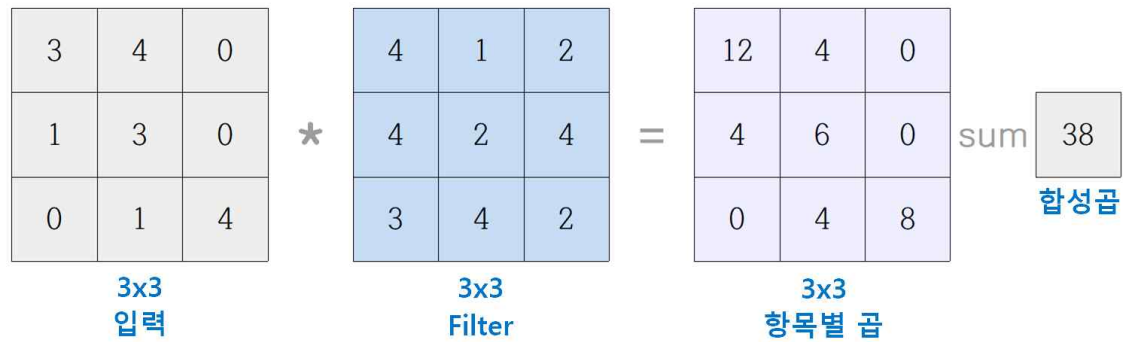
여기서는 3x3 입력 이미지와 3x3 필터에 대해 합성 곱이 수행되는 과정을 살펴봅니다. 이미지의 깊이는 1로 설명의 편의상 생략합니다. 깊이가 고려되는 과정은 뒤에서 구체적으로 살펴봅니다.

다음은 3x3 입력 이미지와 3x3 필터에 대해 합성 곱이 수행되는 과정을 보여줍니다.



3x3 입력 이미지와 3x3 필터는 각각의 대응되는 항목이 개별적으로 곱해져 항목별 곱을 얻은 후에 9개의 항목이 모두 더해져 최종 결과 값 하나를 얻게 됩니다. 합성 곱을 수행할 때 입력의 크기와 필터의 크기는 같아야 합니다. 일반적으로 필터의 크기는 1x1, 3x3, 5x5와 같이 홀수의 곱을 사용합니다.

다음은 3x3 입력 이미지에 대한 합성 곱의 구체적인 예를 보여줍니다.



이 과정을 예제를 통해 살펴봅니다.

1. 다음과 같이 예제를 작성합니다.

512_1.py

```

01 import numpy as np
02
03 np.random.seed(1)
```

```

04
05 image = np.random.randint(5, size=(3,3))
06 print('image =\n', image)
07
08 filter = np.random.randint(5, size=(3,3))
09 print('filter =\n', filter)
10
11 image_x_filter = image * filter
12 print('image_x_filter =\n', image_x_filter)
13
14 convolution = np.sum(image_x_filter)
15 print('convolution =\n', convolution)

```

01 : numpy 모듈을 np라는 이름으로 불러옵니다. numpy 모듈은 행렬이나 다차원 배열을 쉽게 처리 할 수 있도록 지원하는 파이썬의 라이브러리입니다.

03 : np.random.seed 함수를 호출하여 임의 숫자 생성기 모듈을 초기화합니다. 인자로 1을 주어 생성되는 숫자를 일정하게 합니다. 숫자를 변경하면 생성되는 숫자가 달라질 수 있습니다.

05 : 0이상 5미만의 정수를 가진 3x3 행렬을 생성하여 image 변수에 할당합니다.

06 : print 함수를 호출하여 image 변수가 가리키는 행렬 값을 출력해 봅니다.

08 : 0이상 5미만의 정수를 가진 3x3 행렬을 생성하여 filter 변수에 할당합니다.


09 : print 함수를 호출하여 filter 변수가 가리키는 행렬 값을 출력해 봅니다.

11 : image와 filter를 곱한 후, 결과 행렬을 image_x_filter에 할당합니다. 파이썬에서 같은 크기를 갖는 두 개의 NumPy 행렬을 곱하면 같은 위치에 있는 항목끼리 곱셈이 수행됩니다.

12 : print 함수를 호출하여 image_x_filter 변수가 가리키는 행렬 값을 출력해 봅니다.

14 : np.sum 함수를 호출하여 image_x_filter 행렬의 모든 항목 값을 더하여 convolution 변수에 할당합니다.

15 : print 함수를 호출하여 convolution 값을 출력해 봅니다.

2.  버튼을 눌러 프로그램을 실행시킵니다. 다음은 실행 결과 화면입니다.

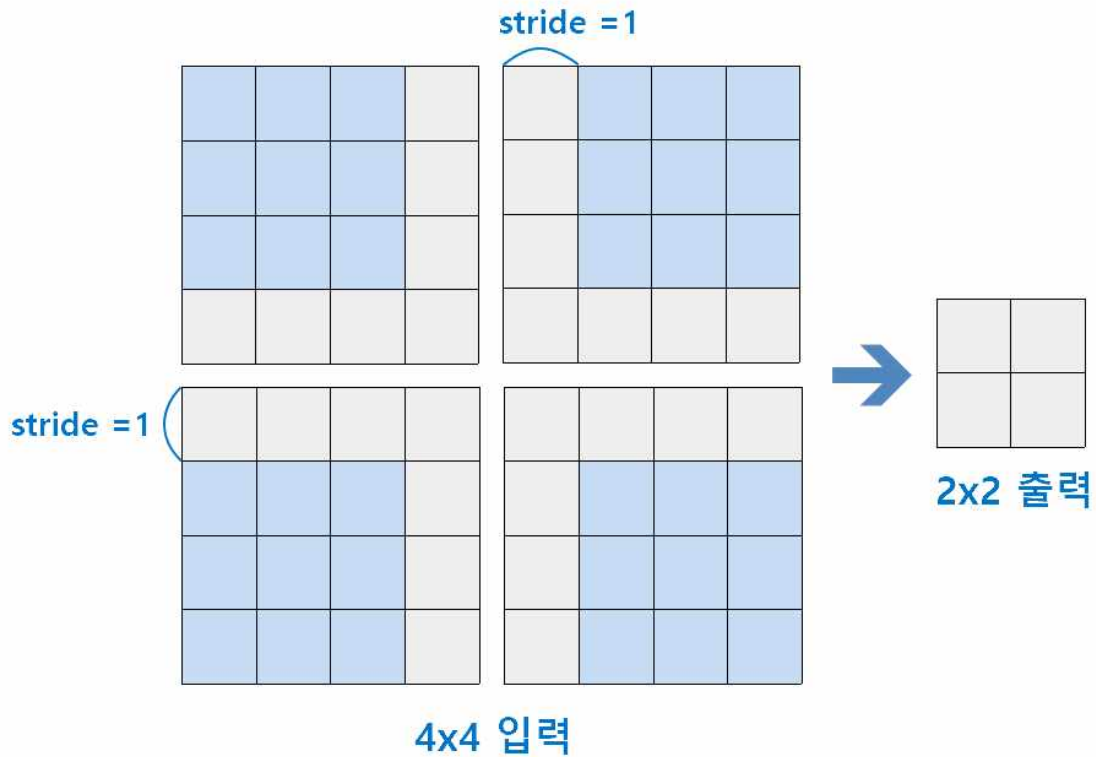


출력 결과를 앞의 그림과 비교하면서 살펴봅니다.

*** 독자의 이해를 돕기 위해 결과 화면을 편집하였습니다.

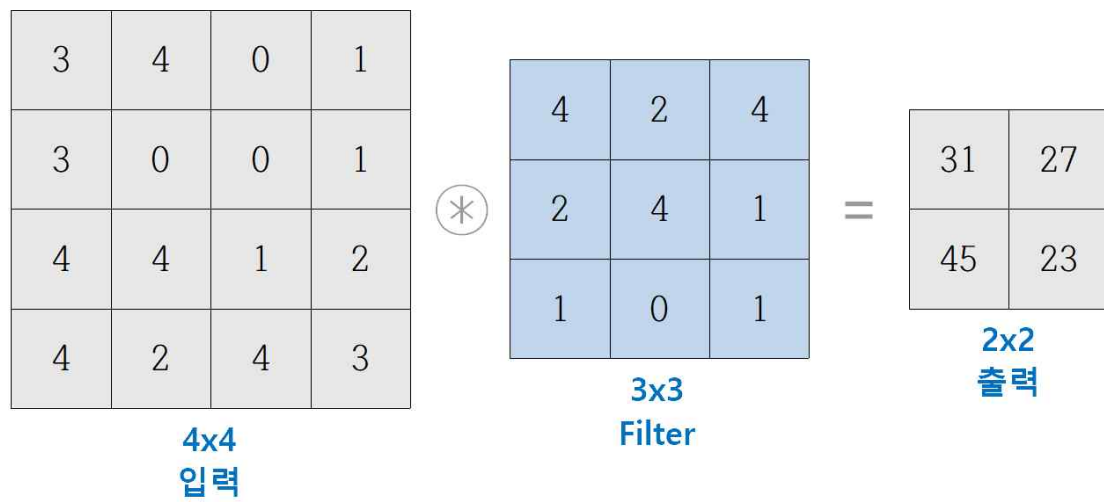
03 4x4 입력 : stride

여기서는 4x4 입력 이미지와 3x3 필터에 대해 합성 곱이 수행되는 과정을 살펴봅니다. 다음은 4x4 입력 이미지와 3x3 필터에 대해 합성 곱이 수행되는 과정을 보여줍니다.

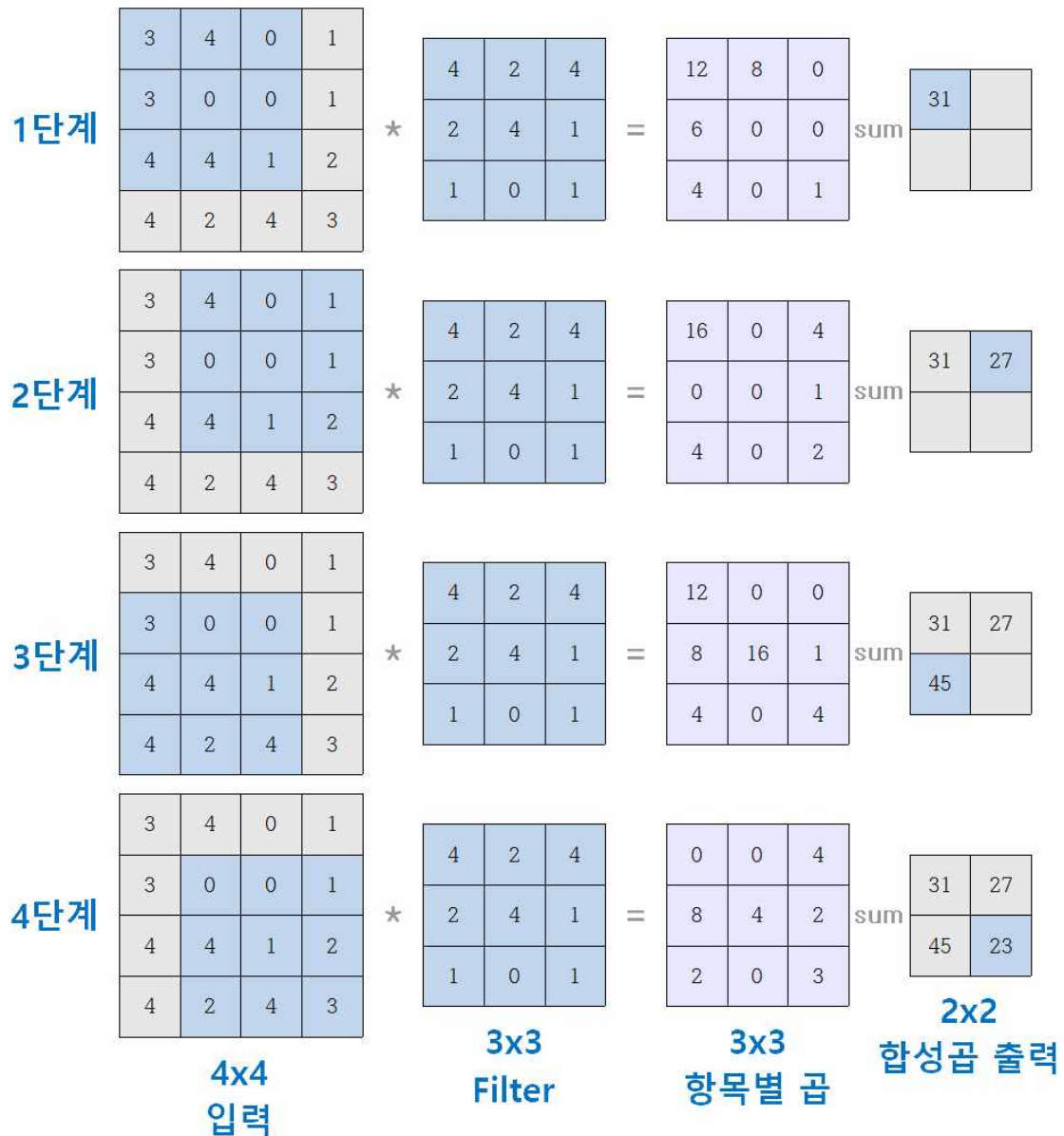


일반적으로 입력 이미지는 필터의 크기보다 큰데, 이때는 필터를 입력 이미지 위에서 일정 간격만큼 왼쪽에서 오른쪽으로 위에서 아래로 옮겨 다니면서 합성 곱을 수행합니다. 이 때 옮겨 다니는 크기를 stride(걸음)라고 합니다. stride 값은 1 이상의 정수 값이 되며 여기서는 1을 사용하고 있습니다. 위 그림은 4x4 입력 이미지로 4개의 3x3 하위 이미지를 생성한 후, 각각의 이미지에 대해 3x3 필터와 합성 곱을 수행하여 2x2 출력 이미지를 생성하는 과정을 나타냅니다.

다음은 4x4 입력 이미지에 대한 합성 곱의 구체적인 예를 보여줍니다.



구체적으로 다음과 같은 과정을 거쳐 합성 곱을 수행합니다.



이 과정을 예제를 통해 살펴봅니다.

1. 다음과 같이 예제를 작성합니다.

513_1.py

```
01 import numpy as np
02
03 np.random.seed(1)
04
05 image = np.random.randint(5, size=(4,4))
06 print('image =\n', image)
```



```

07
08 filter = np.random.randint(5, size=(3,3))
09 print('filter =\n', filter)
10
11 convolution = np.zeros((2,2))
12
13 for row in range(2):
14     for col in range(2):
15         window = image[row:row+3, col:col+3]
16         print('window(%d, %d) =\n' %(row, col), window)
17         print('window(%d, %d)*filter =\n' %(row, col), window*filter)
18         convolution[row, col] = np.sum(window*filter)
19
20 print('convolution =\n', convolution)

```

05 : 0이상 5미만의 임의 정수를 가진 4x4 행렬을 생성하여 image 변수에 할당합니다.

11 : np.zeros 함수를 호출하여 0값으로 채워진 2x2 행렬을 생성하여 convolution 변수에 할당합니다.

13 : row값을 0에서 2 미만까지 바꾸어가며 14~18줄을 2회 수행합니다.

14 : col값을 0에서 2 미만까지 바꾸어가며 15~18줄을 2회 수행합니다.


15 : image 행렬의 row 이상 (row+3) 미만까지, col 이상 (col+3) 미만까지의 행렬을 window 변수가 가리키게 합니다. 예를 들어, (row=0, col=0)일 경우 window 변수는 image[0:3, 0:3] 행렬을 가리키게 됩니다.

16 : print 함수를 호출하여 row, col값과 window가 가리키는 행렬 값을 출력합니다.

17 : print 함수를 호출하여 row, col값과 window*filter 행렬 값을 출력합니다.

18 : np.sum 함수를 호출하여 window*filter 행렬의 모든 항목의 값을 더해서 convolution 행렬의 (row, col) 항목에 할당합니다.

20 : print 함수를 호출하여 convolution 값을 출력합니다.

2.  버튼을 눌러 프로그램을 실행시킵니다. 다음은 실행 결과 화면입니다.

1단계	image = $\begin{bmatrix} 3 & 4 & 0 & 1 \\ 3 & 0 & 0 & 1 \\ 4 & 4 & 1 & 2 \\ 4 & 2 & 4 & 3 \end{bmatrix}$	window(0, 0) = $\begin{bmatrix} 3 & 4 & 0 \\ 3 & 0 & 0 \\ 4 & 4 & 1 \end{bmatrix}$	filter = $\begin{bmatrix} 4 & 2 & 4 \\ 2 & 4 & 1 \\ 1 & 0 & 1 \end{bmatrix}$	window(0, 0)*filter = $\begin{bmatrix} 12 & 8 & 0 \\ 6 & 0 & 0 \\ 4 & 0 & 1 \end{bmatrix}$	convolution = $\begin{bmatrix} 31. & 27. \\ 45. & 23. \end{bmatrix}$
2단계	image = $\begin{bmatrix} 3 & 4 & 0 & 1 \\ 3 & 0 & 0 & 1 \\ 4 & 4 & 1 & 2 \\ 4 & 2 & 4 & 3 \end{bmatrix}$	window(0, 1) = $\begin{bmatrix} 4 & 0 & 1 \\ 0 & 0 & 1 \\ 4 & 1 & 2 \end{bmatrix}$	filter = $\begin{bmatrix} 4 & 2 & 4 \\ 2 & 4 & 1 \\ 1 & 0 & 1 \end{bmatrix}$	window(0, 1)*filter = $\begin{bmatrix} 16 & 0 & 4 \\ 0 & 0 & 1 \\ 4 & 0 & 2 \end{bmatrix}$	convolution = $\begin{bmatrix} 31. & 27. \\ 45. & 23. \end{bmatrix}$
3단계	image = $\begin{bmatrix} 3 & 4 & 0 & 1 \\ 3 & 0 & 0 & 1 \\ 4 & 4 & 1 & 2 \\ 4 & 2 & 4 & 3 \end{bmatrix}$	window(1, 0) = $\begin{bmatrix} 3 & 0 & 0 \\ 4 & 4 & 1 \\ 4 & 2 & 4 \end{bmatrix}$	filter = $\begin{bmatrix} 4 & 2 & 4 \\ 2 & 4 & 1 \\ 1 & 0 & 1 \end{bmatrix}$	window(1, 0)*filter = $\begin{bmatrix} 12 & 0 & 0 \\ 8 & 16 & 1 \\ 4 & 0 & 4 \end{bmatrix}$	convolution = $\begin{bmatrix} 31. & 27. \\ 45. & 23. \end{bmatrix}$
4단계	image = $\begin{bmatrix} 3 & 4 & 0 & 1 \\ 3 & 0 & 0 & 1 \\ 4 & 4 & 1 & 2 \\ 4 & 2 & 4 & 3 \end{bmatrix}$	window(1, 1) = $\begin{bmatrix} 0 & 0 & 1 \\ 4 & 1 & 2 \\ 2 & 4 & 3 \end{bmatrix}$	filter = $\begin{bmatrix} 4 & 2 & 4 \\ 2 & 4 & 1 \\ 1 & 0 & 1 \end{bmatrix}$	window(1, 1)*filter = $\begin{bmatrix} 0 & 0 & 4 \\ 8 & 4 & 2 \\ 2 & 0 & 3 \end{bmatrix}$	convolution = $\begin{bmatrix} 31. & 27. \\ 45. & 23. \end{bmatrix}$
	4x4 입력	3x3 원도우	3x3 Filter	3x3 항목별 곱	2x2 합성곱 출력

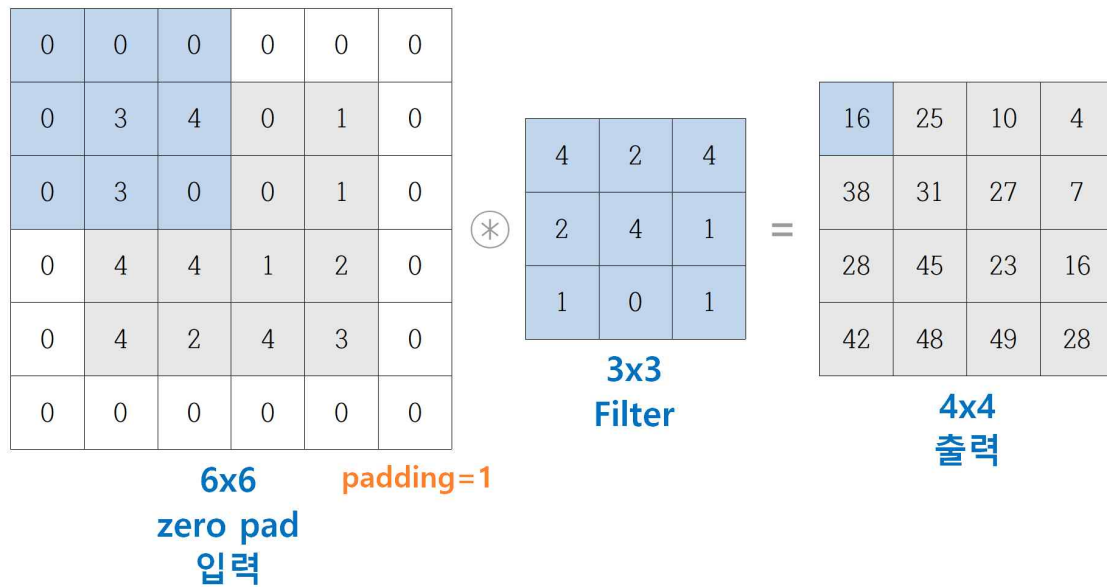
출력 결과를 앞의 그림과 비교하면서 살펴봅니다.

*** 독자의 이해를 돕기 위해 결과 화면을 편집하였습니다.

04 6x6 입력 : padding

앞에서 우리는 4x4 입력 이미지에 대해 3x3 필터와 합성 곱을 수행해 보았습니다. 그 결과 2x2 크기의 출력 이미지를 얻게 되었습니다. 합성 곱을 수행하면 출력 이미지는 입력 이미지 보다 작아지게 됩니다. 반복적으로 합성 곱을 수행하면 이미지의 크기는 점점 더 작아지게 됩니다. 입력 이미지의 크기와 출력 이미지의 크기를 같게 하기 위해서는 입력 이미지의 크기를 늘려 주어야 합니다. 이 과정을 padding(붙이기)라고 합니다.

다음은 4x4 입력 이미지를 상하 좌우로 1칸씩 zero padding을 하여 6x6 이미지로 변경한 후, 합성 곱을 수행한 결과를 보여줍니다.



여기서는 padding=1, stride=1로 수행할 경우 출력 이미지의 크기는 입력 이미지의 크기와 같게 됩니다. 일반적으로 입력 이미지의 크기가 N*N, 필터의 크기가 K*K, padding의 크기가 P, stride의 크기가 S라면 출력 이미지의 크기는 다음과 같습니다.

$$\left\lfloor \frac{N+2P-K}{S} + 1 \right\rfloor \times \left\lfloor \frac{N+2P-K}{S} + 1 \right\rfloor$$

예를 들어, 이미지의 크기가 4x4, 필터의 크기가 3x3, padding의 크기가 1, stride의 크기가 1이라면 출력 이미지의 크기는 다음과 같습니다.

$$\left\lfloor \frac{4+2 \times 1-3}{1} + 1 \right\rfloor \times \left\lfloor \frac{4+2 \times 1-3}{1} + 1 \right\rfloor = 4 \times 4$$

위의 과정을 예제를 통해 살펴봅시다.

1. 다음과 같이 예제를 작성합니다.

514_1.py

```
01 import numpy as np
02
03 np.random.seed(1)
```

```

04
05 image = np.random.randint(5, size=(4,4))
06 print('image =\n', image)
07
08 filter = np.random.randint(5, size=(3,3))
09 print('filter =\n', filter)
10
11 image_pad = np.pad(image,((1,1), (1,1)))
12 print('image_pad =\n', image_pad)
13
14 convolution = np.zeros((4,4))
15
16 for row in range(4):
17     for col in range(4):
18         window = image_pad[row:row+3, col:col+3]
19         convolution[row, col] = np.sum(window*filter)
20
21 print('convolution =\n', convolution)

```

11 : np.pad 함수를 호출하여 image 행렬과 같은 모양의 행렬을 내부적으로 생성한 후, 행을 상하로 한 칸씩, 열을 좌우로 한 칸씩 늘려 0으로 채운 후, image_pad 변수에 할당합니다.

12 : print 함수를 호출하여 image_pad 변수가 가리키는 행렬 값을 출력해 봅니다.

14 : np.zeros 함수를 호출하여 0값으로 채워진 4x4 행렬을 생성하여 convolution 변수에 할당합니다.


16 : row값을 0에서 4 미만까지 바꾸어가며 17~19줄을 4회 수행합니다.

17 : col값을 0에서 4 미만까지 바꾸어가며 18~19줄을 4회 수행합니다.

18 : image_pad 행렬의 row 이상 (row+3) 미만까지, col 이상 (col+3) 미만까지의 행렬을 window 변수가 가리키게 합니다. 예를 들어, (row=0, col=0)일 경우 window 변수는 image_pad[0:3, 0:3] 행렬을 가리키게 됩니다.

19 : np.sum 함수를 호출하여 window*filter 행렬의 모든 항목의 값을 더해서 convolution 행렬의 (row, col) 항목에 할당합니다.

21 : print 함수를 호출하여 convolution 행렬 값을 출력합니다.

2.  버튼을 눌러 프로그램을 실행시킵니다. 다음은 실행 결과 화면입니다.

```

image =
[[3 4 0 1]
 [3 0 0 1]
 [4 4 1 2]
 [4 2 4 3]]
4x4
입력

image_pad =
[[0 0 0 0 0 0]
 [0 3 4 0 1 0]
 [0 3 0 0 1 0]
 [0 4 4 1 2 0]
 [0 4 2 4 3 0]
 [0 0 0 0 0 0]]
6x6
zero pad
입력

filter =
[[4 2 4]
 [2 4 1]
 [1 0 1]]
3x3
Filter

convolution =
[[16. 25. 10.  4.]
 [38. 31. 27.  7.]
 [28. 45. 23. 16.]
 [42. 48. 49. 28.]]
4x4
출력

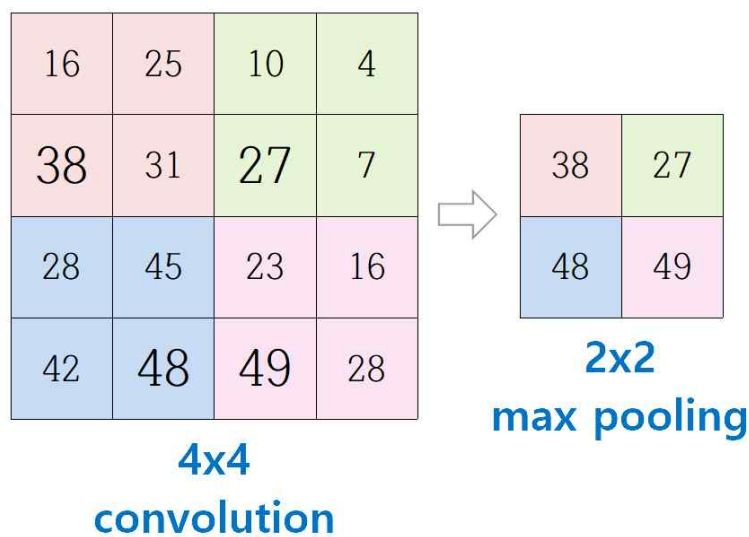
```

출력 결과를 앞의 그림과 비교하면서 살펴봅니다.

*** 독자의 이해를 돕기 위해 결과 화면을 편집하였습니다.

05 4x4 합성 곱 : pooling

다음은 4x4 합성 곱 이미지에 대해 max pooling을 수행한 결과를 보여줍니다. pooling은 특별한 종류의 필터로 생각할 수 있습니다. max pooling은 pooling 필터가 적용되는 영역에서 최대값을 골라냅니다.



일반적으로 pooling 필터의 크기는 2x2가 사용되며, stride는 2가 됩니다.

이 과정을 예제를 통해 살펴봅니다.

1. 다음과 같이 예제를 작성합니다.

515_1.py

```
01 import numpy as np
02
03 np.random.seed(1)
04
05 image = np.random.randint(5, size=(4,4))
06 print('image =\n', image)
07
08 filter = np.random.randint(5, size=(3,3))
09 print('filter =\n', filter)
10
11 image_pad = np.pad(image,((1,1), (1,1)))
12 print('image_pad =\n', image_pad)
13
14 convolution = np.zeros((4,4))
15
16 for row in range(4):
17     for col in range(4):
18         window = image_pad[row:row+3, col:col+3]
19         convolution[row, col] = np.sum(window*filter)
20
21 print('convolution =\n', convolution)
22
23 max_pooled = np.zeros((2,2))
24
25 for row in range(0, 2):
26     for col in range(0, 2):
27         window = convolution[2*row:2*row+2, 2*col:2*col+2]
28         max_pooled[row, col] = np.max(window)
29
30 print('max_pooled =\n', max_pooled)
```

23 : np.zeros 함수를 호출하여 0값으로 채워진 2x2 행렬을 생성하여 max_pooled 변수에 할당합니다.

25 : row값을 0에서 2 미만까지 26~28줄을 2회 수행합니다.

26 : col값을 0에서 2 미만까지 27~28줄을 2회 수행합니다.

27 : convolution 행렬의 2*row 이상 (2*row+2) 미만까지, 2*col 이상 (2*col+2) 미만까지의 행렬을 window 변수가 가리키게 합니다. 예를 들어, (row=0, col=0)일 경우 window 변수는 convolution[0:2, 0:2] 행렬을 가리키게 됩니다.

28 : np.max 함수를 호출하여 window가 가리키는 행렬의 항목 중에 최대값을 갖는 항목을 max_pooled 행렬의 (row, col) 항목에 할당합니다.

30 : print 함수를 호출하여 max_pooled 행렬 값을 출력합니다.

2. ▶ 버튼을 눌러 프로그램을 실행시킵니다. 다음은 실행 결과 화면입니다.

```

image =      image_pad =
[[3 4 0 1]   [[0 0 0 0 0 0]
[3 0 0 1]    [0 3 4 0 1 0]
[4 4 1 2]    [0 3 0 0 1 0]
[4 2 4 3]]   [0 4 4 1 2 0]
              [0 4 2 4 3 0]
              [0 0 0 0 0 0]]

              6x6
              zero pad
              입력

filter =      convolution =
[[4 2 4]     [[16. 25. 10. 4.]
[2 4 1]      [38. 31. 27. 7.]
[1 0 1]]     [28. 45. 23. 16.]
              [42. 48. 49. 28.]]

              3x3
              Filter

              4x4
              convolution

max_pooled =
[[38. 27.]
[48. 49.]]

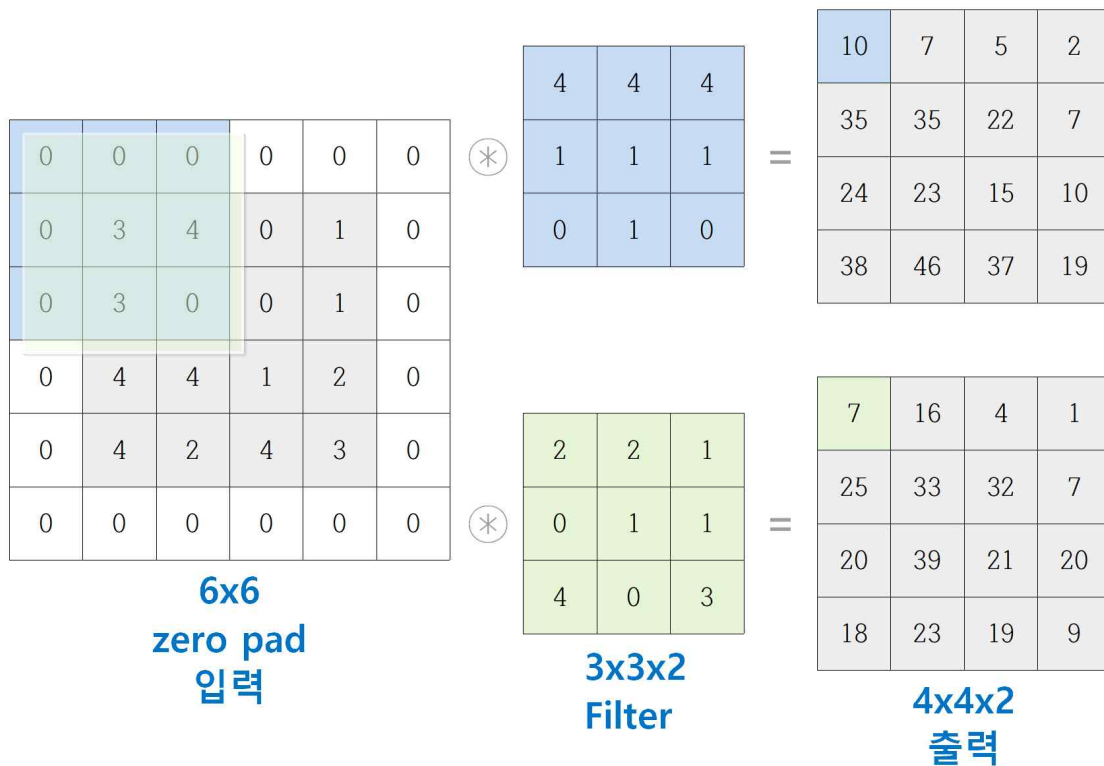
              2x2
              max pooling
  
```

출력 결과를 앞의 그림과 비교하면서 살펴봅니다.

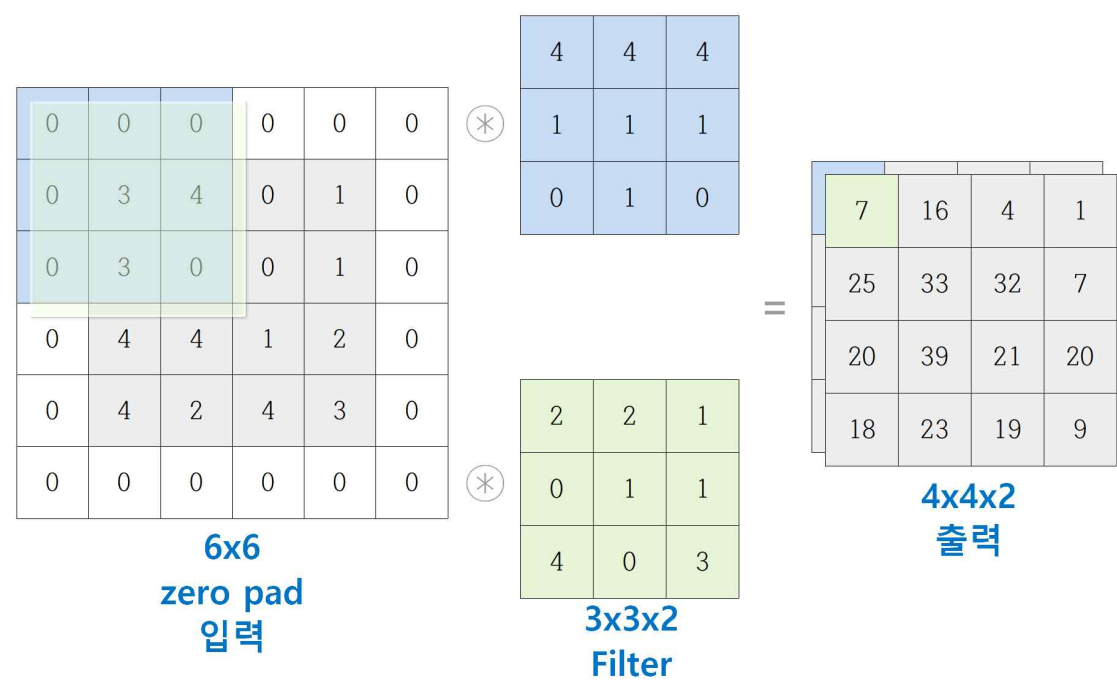
*** 독자의 이해를 돕기 위해 결과 화면을 편집하였습니다.

06 6x6 입력 필터 늘리기

다음은 6x6 입력 이미지에 대해 2개의 필터를 이용하여 2개의 출력 이미지를 얻는 과정을 보여줍니다. 필터는 이미지의 특징을 추출하는 역할을 하므로 여기서는 입력 이미지에서 2가지 특징을 추출하게 됩니다. 일반적으로 필터의 개수만큼 이미지의 특징이 추출됩니다.



추출된 2장의 이미지는 다음과 같이 겹친 상태로 2층으로 쌓여진 하나의 입체 이미지를 구성합니다. 즉, 겹쳐진 개수만큼의 깊이를 갖는 하나의 이미지를 구성합니다.



일반적으로 필터의 개수만큼 출력 이미지가 생성되며, 출력 이미지는 겹쳐진 상태로 하나의 입체 이미지로 출력됩니다. CNN에서는 최초에 입력된 이미지의 경우도 1층짜리 하나의 입체 이미지로 처리됩니다.

다음은 4x4x2 출력 이미지에 대해 max pooling을 수행한 결과를 보여줍니다.

10	7	5	2
35	35	22	7
24	23	15	10
38	46	37	19

⇒

35	22
46	37

7	16	4	1
25	33	32	7
20	39	21	20
18	23	19	9

⇒

33	32
39	21

4x4x2
convolution

2x2x2
max pooling

지금까지의 과정을 예제를 통해 정리해 봅니다.

1. 다음과 같이 예제를 작성합니다.

516_1.py

```

01 import numpy as np
02
03 np.random.seed(1)
04
05 image = np.random.randint(5, size=(4,4))
06 print('image =\n', image)
07
08 filter = np.random.randint(5, size=(3,3,2))
09 print('filter_0 =\n', filter[:, :, 0])
10 print('filter_1 =\n', filter[:, :, 1])
11

```

```

12 image_pad = np.pad(image,((1,1), (1,1)))
13 print('image_pad =\n', image_pad)
14
15 convolution = np.zeros((4,4,2))
16
17 for fn in range(2):
18     for row in range(4):
19         for col in range(4):
20             window = image_pad[row:row+3, col:col+3]
21             convolution[row, col, fn] = np.sum(window*filter[:, :, fn])
22
23 print('convolution_0 =\n', convolution[:, :, 0])
24 print('convolution_1 =\n', convolution[:, :, 1])
25
26 max_pooled = np.zeros((2,2,2))
27
28 for fn in range(2):
29     for row in range(0, 2):
30         for col in range(0, 2):
31             window = convolution[2*row:2*row+2, 2*col:2*col+2, fn]
32             max_pooled[row, col, fn] = np.max(window)
33
34 print('max_pooled_0 =\n', max_pooled[:, :, 0])
35 print('max_pooled_1 =\n', max_pooled[:, :, 1])

```

08 : 0이상 5미만의 정수를 가진 3x3x2 행렬을 생성하여 filter 변수에 할당합니다. 이렇게 하면 3x3 행렬 2개가 생성됩니다. 이 경우에는 3x3 크기의 필터 2개로 해석합니다.

09 : print 함수를 호출하여 첫 번째 filter 행렬 값을 출력해 봅니다.

10 : print 함수를 호출하여 두 번째 filter 행렬 값을 출력해 봅니다.

15 : np.zeros 함수를 호출하여 0값으로 채워진 4x4x2 행렬을 생성하여 convolution 변수에 할당합니다. 이 경우에는 2의 depth(깊이)를 갖는 4x4 행렬로 해석합니다.

17 : fn값을 0에서 2 미만까지 바꾸어가며 18~21줄을 2회 수행합니다. fn은 filter number를 의미합니다.

18 : row값을 0에서 4 미만까지 바꾸어가며 19~21줄을 4회 수행합니다.

19 : col값을 0에서 4 미만까지 바꾸어가며 20~21줄을 4회 수행합니다.

20 : image_pad 행렬의 row 이상 (row+3) 미만까지, col 이상 (col+3) 미만까지의 행렬을 window 변수가 가리키게 합니다. 예를 들어, (row=0, col=0)일 경우 window 변수는 image_pad[0:3, 0:3] 행렬을 가리키게 됩니다.

21 : np.sum 함수를 호출하여 window*filter[:, :, fn] 행렬의 모든 항목의 값을 더해 convolution 행렬의 (row, col, fn) 항목에 할당합니다.

23 : print 함수를 호출하여 convolution[:, :, 0] 행렬 값을 출력합니다.

24 : print 함수를 호출하여 convolution[:, :, 1] 행렬 값을 출력합니다.

26 : np.zeros 함수를 호출하여 0값으로 채워진 2x2x2 행렬을 생성하여 max_pooled 변수에 할당합니다. 이 경우에는 맨 뒤에 오는 2의 depth(깊이)를 갖는 2x2 행렬로 해석합니다.

28 : fn값을 0에서 2 미만까지 바꾸어가며 29~32줄을 2회 수행합니다.

29 : row값을 0에서 2 미만까지 30~32줄을 2회 수행합니다.


30 : col값을 0에서 2 미만까지 31~32줄을 2회 수행합니다.

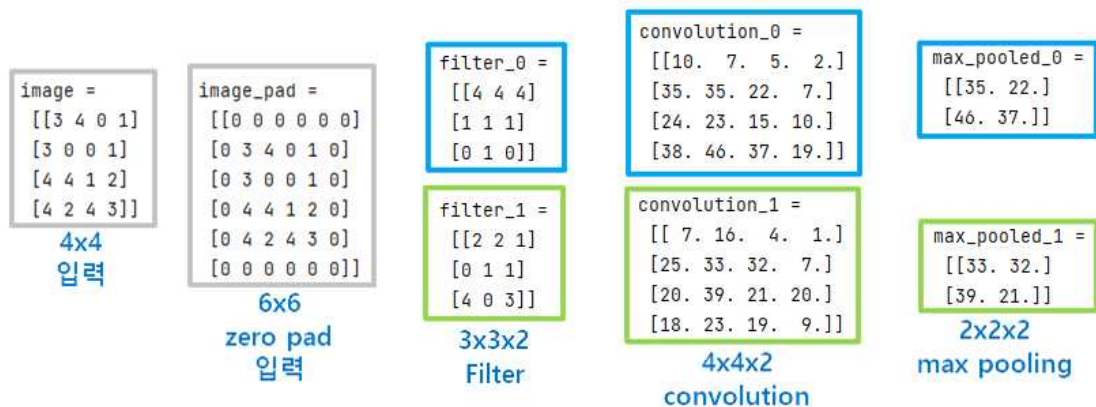
31 : convolution 행렬의 [2*row 이상 (2*row+2) 미만, 2*col 이상 (2*col+2) 미만, fn 깊이]에 있는 행렬을 window 변수가 가리키게 합니다. 예를 들어, (row=0, col=0, fn=0)일 경우 window 변수는 convolution[0:2, 0:2, 0] 행렬을 가리키게 됩니다.

32 : np.max 함수를 호출하여 window가 가리키는 행렬의 항목 중에 최대값을 갖는 항목을 max_pooled 행렬의 (row, col, fn) 항목에 할당합니다.

34 : print 함수를 호출하여 max_pooled[:, :, 0] 행렬 값을 출력합니다.

35 : print 함수를 호출하여 max_pooled[:, :, 1] 행렬 값을 출력합니다.

2.  버튼을 눌러 프로그램을 실행시킵니다. 다음은 실행 결과 화면입니다.



출력 결과를 앞의 그림과 비교하면서 살펴봅니다.

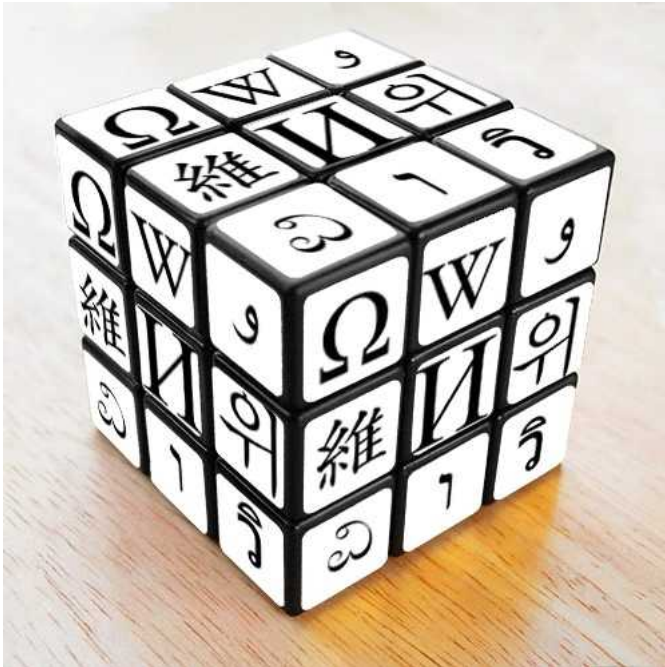
*** 독자의 이해를 돕기 위해 결과 화면을 편집하였습니다.

07 필터 역할 살펴보기

앞에서 우리는 입력 이미지에 3x3 필터를 적용하여 합성 곱을 수행하는 과정을 살펴보았습니다. 이 과정을 영상 인식에서는 필터링이라고 합니다. 필터링이란 이미지나 영상에서 원하는 정보만 걸러내는 작업을 말합니다. 이 과정에서 잡음을 걸러내어 영상을 깨끗하게 만들거나 잔선을 제거하여 흐리게 만들거나 부드러운 느낌을 걸러내어 선명한 느낌의 영상을 만들 수 있습니다. 필터링은 필터, 커널, 마스크, 윈도우 등으로 불리는 작은 크기의 행렬을 이용합니다. 필터링 연산의 결과는 행렬의 크기와 각 항목의 값에 의해 결정됩니다. 여기서는 필터링을 통해 추출되는 이미지의 특징을 구체적으로 살펴봅니다.

다음은 여러 나라의 문자가 표시된 큐브 이미지입니다. 이 큐브 이미지에 몇 가지 필터를 적

용하여 이미지의 특징을 추출해 봅니다.



부드러운 이미지 추출하기

먼저 평균값 필터를 사용해 봅니다. 평균값 필터는 이웃 화소들을 평균화하여 이미지를 부드럽게 만들어주는 역할을 합니다. 평균값 필터는 다음과 같습니다.

$$K = \frac{1}{9} \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

이 필터는 주변의 모든 픽셀을 더해준 후, 필터의 원소 개수만큼 나누어줍니다.

1. 다음과 같이 예제를 작성합니다.

517_1.py

```
01 import numpy as np
02 import cv2
03 import matplotlib.pyplot as plt
04
05 image_color = cv2.imread('cube.png')
```

```

06 print('image_color.shape =', image_color.shape)
07 image = cv2.cvtColor(image_color, cv2.COLOR_BGR2GRAY)
08 print('image.shape =', image.shape)
09
10 filter = np.array([
11     [1,1,1],
12     [1,1,1],
13     [1,1,1],
14 ])/9
15
16 image_pad = np.pad(image,((1,1), (1,1)))
17 print('image_pad.shape =', image_pad.shape)
18
19 convolution = np.zeros_like(image)
20
21 for row in range(image.shape[0]):
22     for col in range(image.shape[1]):
23         window = image_pad[row:row+3, col:col+3]
24         convolution[row, col] = np.clip(np.sum(window*filter), 0, 255)
25
26 images = [image, convolution]
27 labels = ['gray', 'convolution']
28
29 plt.figure(figsize=(10, 5))
30 for i in range(len(images)):
31     plt.subplot(1,2,i+1)
32     plt.xticks([])
33     plt.yticks([])
34     plt.imshow(images[i], cmap=plt.cm.gray)
35     plt.xlabel(labels[i])
36 plt.show()

```

02 : import문을 이용하여 cv2 모듈을 불러옵니다. cv2는 OpenCV에 대한 파이썬 라이브러리입니다. OpenCV 라이브러리는 영상 처리 라이브러리입니다.

03 : import문을 이용하여 matplotlib.pyplot 모듈을 plt라는 이름으로 불러옵니다. 여기서는 matplotlib.pyplot 모듈을 이용하여 29~36 줄에서 이미지를 출력합니다.

05 : cv2 모듈의 imread 함수를 호출하여 cube.png 파일을 읽어와 image_color 변수로 가리키게 합니다. imread 함수는 numpy.ndarray 객체를 내어줍니다. numpy.ndarray는 numpy 모듈에서 제공하는 배열입니다.

06 : print 함수를 호출하여 image_color의 모양을 출력합니다.

07 : cv2 모듈의.cvtColor 함수를 호출하여 image_color 변수가 가리키는 그림의 색깔을 바꾼 후, 바뀐 그림을 image 변수가 가리키도록 합니다. BGR 형식의 파일을 GRAY 형식의 파

일로 바꿉니다. OpenCV의 색깔 형식을 RGB라고 하나 실제로는 바이트 데이터의 순서가 반대인 BGR 형식입니다.

08 : print 함수를 호출하여 image의 모양을 출력합니다.

10~14 : np.array 함수를 호출하여, 1로 채워진 3x3 행렬을 생성하여 각 항목을 9로 나뉘준 후, filter 변수에 할당합니다. 이 필터는 영상 처리에서 사용하는 필터로 평균값 필터라고 하며, 그림을 흐릿하게 하여 잔선을 제거해 주는 역할을 합니다.

16 : np.pad 함수를 호출하여 image 행렬과 같은 모양의 행렬을 내부적으로 생성한 후, 행을 상하로 한 칸씩, 열을 좌우로 한 칸씩 늘려 0으로 채운 후, image_pad 변수에 할당합니다.

17 : print 함수를 호출하여 image_pad의 모양을 출력합니다.

19 : np.zeros_like 함수를 호출하여 0으로 채워진 image 모양과 같은 행렬을 생성합니다.

21 : row값을 0에서 image의 세로 크기 미만까지 바꾸어가며 22~24줄을 수행합니다.

22 : col값을 0에서 image의 가로 크기 미만까지 바꾸어가며 23~24줄을 수행합니다.

23 : image_pad 행렬의 row 이상 (row+3) 미만까지, col 이상 (col+3) 미만까지의 행렬을 window 변수가 가리키게 합니다. 예를 들어, (row=0, col=0)일 경우 window 변수는 image_pad[0:3, 0:3] 행렬을 가리키게 됩니다.

24 : np.sum 함수를 호출하여 window*filter 행렬의 모든 항목의 값을 더하고, np.clip 함수를 이용하여 0에서 255 사이로 제한해 준 후에, convolution 행렬의 (row, col) 항목에 할당합니다.

21 : print 함수를 호출하여 convolution 행렬 값을 출력합니다.

26 : image와 convolution을 리스트에 넣은 후, images 변수에 할당합니다. images는 29~36 줄에서 그릴 이미지 리스트입니다.

27 : 'gray'와 'convolution' 문자열을 리스트에 넣은 후, labels 변수에 할당합니다.

29 : plt.figure 함수를 호출하여 새로운 그림을 만들 준비를 합니다. figure 함수는 내부적으로 그림을 만들고 편집할 수 있게 해 주는 함수입니다. figsize는 그림의 인치 단위의 크기를 나타냅니다. 여기서는 가로 10인치, 세로 5인치의 그림을 그린다는 의미입니다.

30 : 0에서 images 리스트 크기 미만에 대해


31 : plt.subplot 함수를 호출하여 그림 창을 분할하여 하위 그림을 그립니다. 1,2는 각각 행의 개수와 열의 개수를 의미합니다. i+1은 하위 그림의 위치를 나타냅니다.

32, 33 : plt.xticks, plt.yticks 함수를 호출하여 x, y 축 눈금을 설정합니다. 여기서는 빈 리스트를 주어 눈금 표시를 하지 않습니다.

34 : plt.imshow 함수를 호출하여 images[i] 항목의 그림을 내부적으로 그립니다. cmap은 color map의 약자로 gray는 그림을 흑백으로 표현해 줍니다.

35 : plt.xlabel 함수를 호출하여 x 축에 라벨을 붙여줍니다. 라벨의 값은 labels[i]입니다.

36 : plt.show 함수를 호출하여 내부적으로 그린 그림을 화면에 그립니다.

2.  버튼을 눌러 프로그램을 실행시킵니다. 다음은 실행 결과 화면입니다.

```
image_color.shape = (500, 500, 3)
image.shape = (500, 500)
image_pad.shape = (502, 502)
```

원 이미지의 크기는 500x500x3이며, 흑백으로 처리한 후에는 500x500의 크기가 됩니다. 그리고 패딩을 수행한 후에는 502x502의 크기가 됩니다.



gray



convolution

오른쪽 이미지가 왼쪽 이미지에 비해 조금 흐릿한 것을 확인합니다.

선명한 이미지 추출하기


다음은 Sharpening 필터를 사용해 봅시다. Sharpening 필터는 경계를 선명하게 표현합니다. Sharpening 필터는 다음과 같습니다.

$$K = \begin{bmatrix} -1 & -1 & -1 \\ -1 & 9 & -1 \\ -1 & -1 & -1 \end{bmatrix}$$

1. 다음과 같이 예제를 수정합니다.

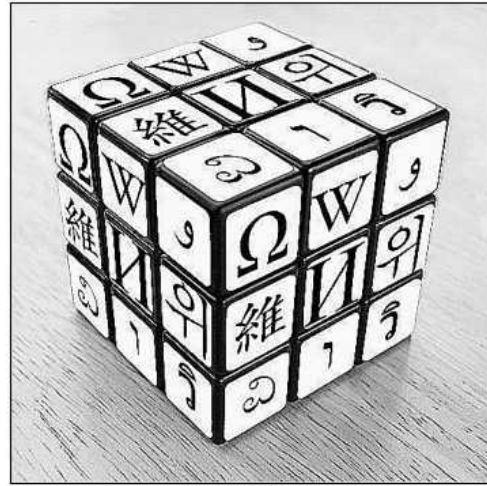
```
10 filter = np.array([
11     [-1,-1,-1],
12     [-1, 9,-1],
13     [-1,-1,-1]
14 ])
```

10~14 : np.array 함수를 호출하여, 예제와 같은 숫자로 채워진 3x3 행렬을 생성한 후, filter 변수에 할당합니다. 이 필터는 영상 처리에서 사용하는 필터로 sharpening 필터라고 하며, 그림을 선명하게 해 주는 역할을 합니다.

2.  버튼을 눌러 프로그램을 실행시킵니다. 다음은 실행 결과 화면입니다.



gray



convolution

오른쪽 이미지가 왼쪽 이미지에 비해 경계선이 선명한 것을 확인합니다.

경계선 추출하기


다음은 Edge detection 필터를 사용해 봅시다. Edge detection 필터는 경계선을 추출합니다. Edge detection 필터는 다음과 같습니다.

$$K = \begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$$

1. 다음과 같이 예제를 수정합니다.

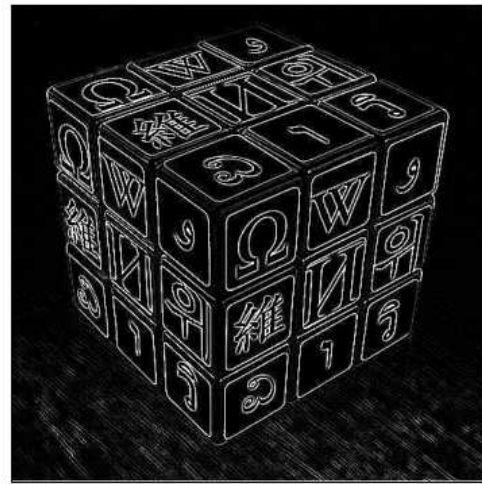
```
10 filter = np.array([
11     [-1,-1,-1],
12     [-1, 8,-1],
13     [-1,-1,-1]
14 ])
```

10~14 : np.array 함수를 호출하여, 예제와 같은 숫자로 채워진 3x3 행렬을 생성한 후, filter 변수에 할당합니다. 이 필터는 영상 처리에서 사용하는 필터로 edge detection 필터라고 하며, 사물의 경계선을 추출해 주는 역할을 합니다.

2.  버튼을 눌러 프로그램을 실행시킵니다. 다음은 실행 결과 화면입니다.



gray



convolution

왼쪽 이미지에서 경계선이 추출된 것을 확인합니다.

*** 기존의 영상 인식에서는 프로그래머가 필터를 정해 주어야 했으나 CNN에서는 이러한 필터들이 학습을 통해 만들어지게 됩니다. 즉, CNN에서 필터는 학습의 대상이 되는 가중치가 됩니다.

이미지 단순화

모으기(Pooling)는 합성 곱을 이용하여 얻어낸 이미지를 단순화하는 연산을 말합니다. 여기서는 max pooling을 이용하여 이미지를 단순화해 봅니다.

1. 다음과 같이 예제를 수정합니다.

517_2.py

```
01 import numpy as np
02 import cv2
03 import matplotlib.pyplot as plt
04
05 image_color = cv2.imread('cube.png')
06 print('image_color.shape =', image_color.shape)
07 image = cv2.cvtColor(image_color, cv2.COLOR_BGR2GRAY)
08 print('image.shape =', image.shape)
09
10 filter = np.array([
11     [-1,-1,-1],
12     [-1, 8,-1],
```

```

13     [-1,-1,-1]
14 ])
15
16 image_pad = np.pad(image,((1,1), (1,1)))
17 print('image_pad.shape =', image_pad.shape)
18
19 convolution = np.zeros_like(image)
20
21 for row in range(image.shape[0]):
22     for col in range(image.shape[1]):
23         window = image_pad[row:row+3, col:col+3]
24         convolution[row, col] = np.clip(np.sum(window*filter), 0, 255)
25
26 max_pooled = np.zeros((int(image.shape[0]/2),int(image.shape[1]/2)))
27
28 for row in range(0, int(image.shape[0]/2)):
29     for col in range(0, int(image.shape[1]/2)):
30         window = convolution[2*row:2*row+2, 2*col:2*col+2]
31         max_pooled[row, col] = np.max(window)
32
33 images = [image, convolution, max_pooled]
34 labels = ['gray', 'convolution', 'max_pooled']
35
36 plt.figure(figsize=(15, 5))
37 for i in range(len(images)):
38     plt.subplot(1,3,i+1)
39     plt.xticks([])
40     plt.yticks([])
41     plt.imshow(images[i], cmap=plt.cm.gray)
42     plt.xlabel(labels[i])
43 plt.show()

```

26 : np.zeros 함수를 호출하여 0값으로 채워진 (image 가로 크기/2)*(image 세로 크기/2) 행렬을 생성하여 max_pooled 변수에 할당합니다.

28 : row값을 0에서 (image 세로 크기/2) 미만까지 29~31줄을 수행합니다.

29 : col값을 0에서 (image 가로 크기/2) 미만까지 30~31줄을 수행합니다.

30 : convolution 행렬의 2*row 이상 (2*row+2) 미만까지, 2*col 이상 (2*col+2) 미만까지의 행렬을 window 변수가 가리키게 합니다. 예를 들어, (row=0, col=0)일 경우 window 변수는 convolution[0:2, 0:2] 행렬을 가리키게 됩니다.


31 : np.max 함수를 호출하여 window가 가리키는 행렬의 항목 중에 최대값을 갖는 항목을 max_pooled 행렬의 (row, col) 항목에 할당합니다.

33 : images 리스트에 max_pooled를 추가합니다.

34 : labels 리스트에 'max_pooled' 문자열을 추가합니다.

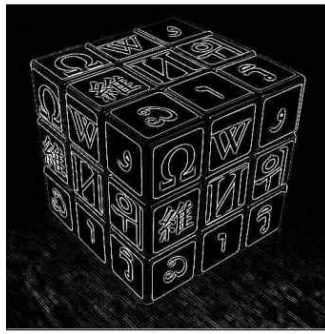
36 : 그림의 가로 크기를 15인치로 변경합니다.

38 : 그림의 열의 개수를 3으로 변경합니다.

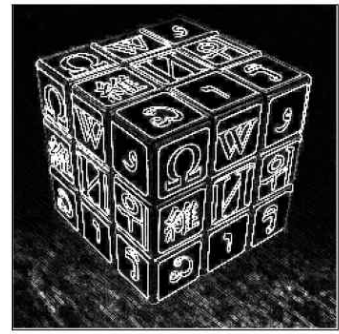
2.  버튼을 눌러 프로그램을 실행시킵니다. 다음은 실행 결과 화면입니다.



gray



convolution

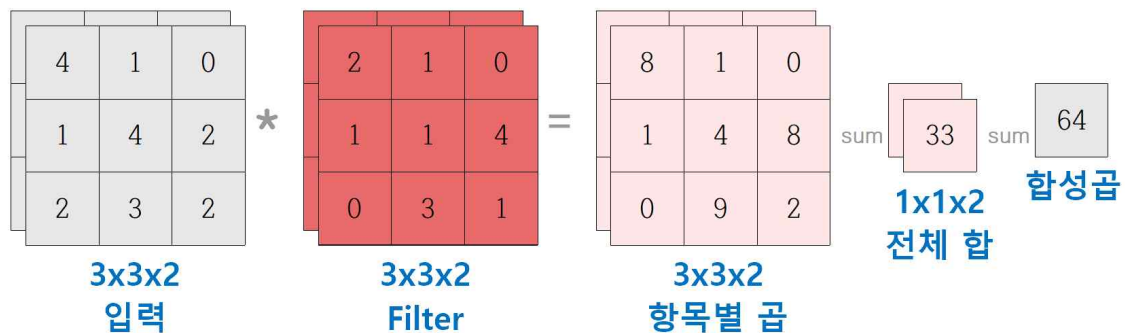


max_pooled

맨 오른쪽의 이미지는 중간 이미지에 대해 max pooling을 거쳐 단순화한 이미지입니다.

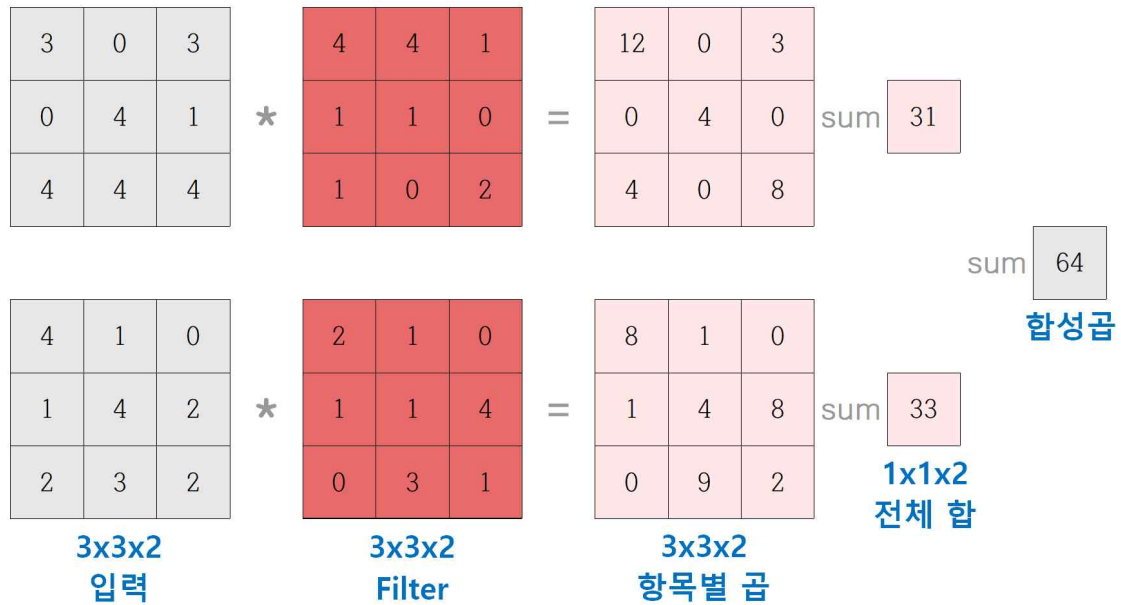
08 3x3x2 입력

여기서는 3x3x2 입력 이미지에 대해 합성 곱을 수행하는 과정을 살펴봅니다. 다음은 3x3x2 이미지에 대해 합성 곱을 수행하는 과정을 나타냅니다.



3x3x2 입력 이미지는 3x3x1 입력 이미지 2 장이 겹쳐진 상태로 입력되는 하나의 입체 이미지입니다. 겹쳐진 정도를 depth(깊이) 또는 channel이라고 합니다. 각각의 이미지는 RGB와 같이 색깔의 특징을 가진 이미지일 수도 있고, 다른 여러 가지 필터를 통해 추출된 특징을 가진 이미지일 수도 있습니다. 입력 이미지에 합성 곱을 위해 적용되는 필터의 두께(깊이)는 이미지의 두께와 같아야 합니다. 여기서 필터는 3x3x2로 이미지의 두께와 같습니다. 예를 들어, 이미지의 크기가 3x3x3이라면 필터의 크기는 3x3x3이 되며, 이미지의 크기가 3x3x4라면 필터의 크기도 3x3x4가 되어야 합니다. 각 층의 이미지와 필터는 각각의 대응되는 항목이 개별적으로 곱해져 항목별 곱을 구성한 후에 9개의 항목이 모두 더해져 중간 결과 값을 얻은 후에 층별로 더해진 값을 모두 더해 최종 결과 값 하나를 얻게 됩니다. 이 과정은 3, 4, 5 층

등 임의의 개수의 층에도 적용됩니다. 다음은 3x3x2 입력 이미지에 대해 구체적으로 수행되는 합성곱 과정을 나타냅니다.



지금까지의 과정을 예제를 통해 정리해 봅니다.

1. 다음과 같이 예제를 작성합니다.

518_1.py

```
01 import numpy as np
02
03 np.random.seed(1)
04
05 image = np.random.randint(5, size=(3,3,2))
06 print('image_0 =\n', image[:, :, 0])
07 print('image_1 =\n', image[:, :, 1])
08
09 filter = np.random.randint(5, size=(3,3,2))
10 print('filter_0 =\n', filter[:, :, 0])
11 print('filter_1 =\n', filter[:, :, 1])
12
13 image_x_filter = image * filter
14 print('image_x_filter_0 =\n', image_x_filter[:, :, 0])
15 print('image_x_filter_1 =\n', image_x_filter[:, :, 1])
16
17 convolution = np.sum(image_x_filter)
18 print('convolution =\n', convolution)
```

05 : 0이상 5미만의 정수를 가진 3x3x2 행렬을 생성하여 image 변수에 할당합니다. 이 경우에는 2의 depth(깊이)를 갖는 3x3 행렬로 해석합니다.

06 : print 함수를 호출하여 image[:, :, 0] 행렬 값을 출력합니다.

07 : print 함수를 호출하여 image[:, :, 1] 행렬 값을 출력합니다.

09 : 0이상 5미만의 정수를 가진 3x3x2 행렬을 생성하여 filter 변수에 할당합니다. 이 경우에는 2의 depth(깊이)를 갖는 3x3 행렬로 해석합니다. 필터의 깊이 2는 입력 이미지의 깊이 2와 같아야 합니다.

10 : print 함수를 호출하여 filter[:, :, 0] 행렬 값을 출력합니다.

11 : print 함수를 호출하여 filter[:, :, 1] 행렬 값을 출력합니다.


13 : image와 filter를 곱한 후, 결과 행렬을 image_x_filter에 할당합니다. 파이썬에서 같은 크기를 갖는 두 개의 NumPy 행렬을 곱하면 같은 위치에 있는 항목끼리 곱셈이 수행됩니다. 3차원 행렬에 대해서도 마찬가지입니다.

14 : print 함수를 호출하여 image_x_filter[:, :, 0] 행렬 값을 출력합니다.

15 : print 함수를 호출하여 image_x_filter[:, :, 1] 행렬 값을 출력합니다.

17 : np.sum 함수를 호출하여 image_x_filter 행렬의 모든 항목 값을 더하여 convolution 변수에 할당합니다.

18 : print 함수를 호출하여 convolution 변수 값을 출력해 봅니다.

2.  버튼을 눌러 프로그램을 실행시킵니다. 다음은 실행 결과 화면입니다.

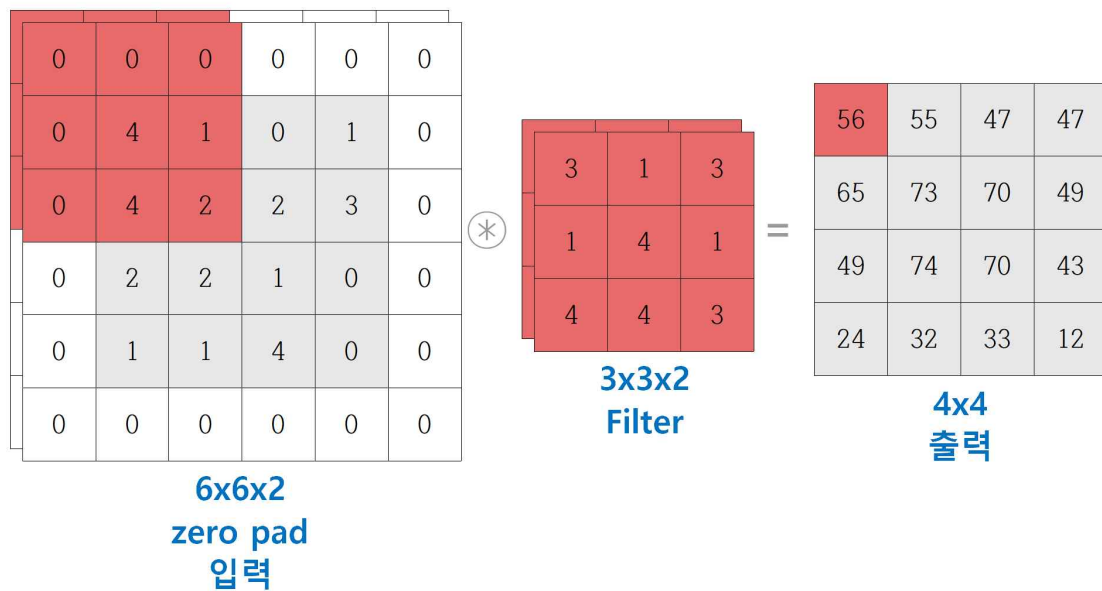
<pre>image_0 = [[3 0 3] [0 4 1] [4 4 4]] image_1 = [[4 1 0] [1 4 2] [2 3 2]]</pre>	<pre>filter_0 = [[4 4 1] [1 1 0] [1 0 2]] filter_1 = [[2 1 0] [1 1 4] [0 3 1]]</pre>	<pre>image_x_filter_0 = [[12 0 3] [0 4 0] [4 0 8]] image_x_filter_1 = [[8 1 0] [1 4 8] [0 9 2]]</pre>	<pre>convolution = 64</pre>
3x3x2 입력	3x3x2 Filter	3x3x2 항목별 곱	합성곱

출력 결과를 앞의 그림과 비교하면서 살펴봅니다.

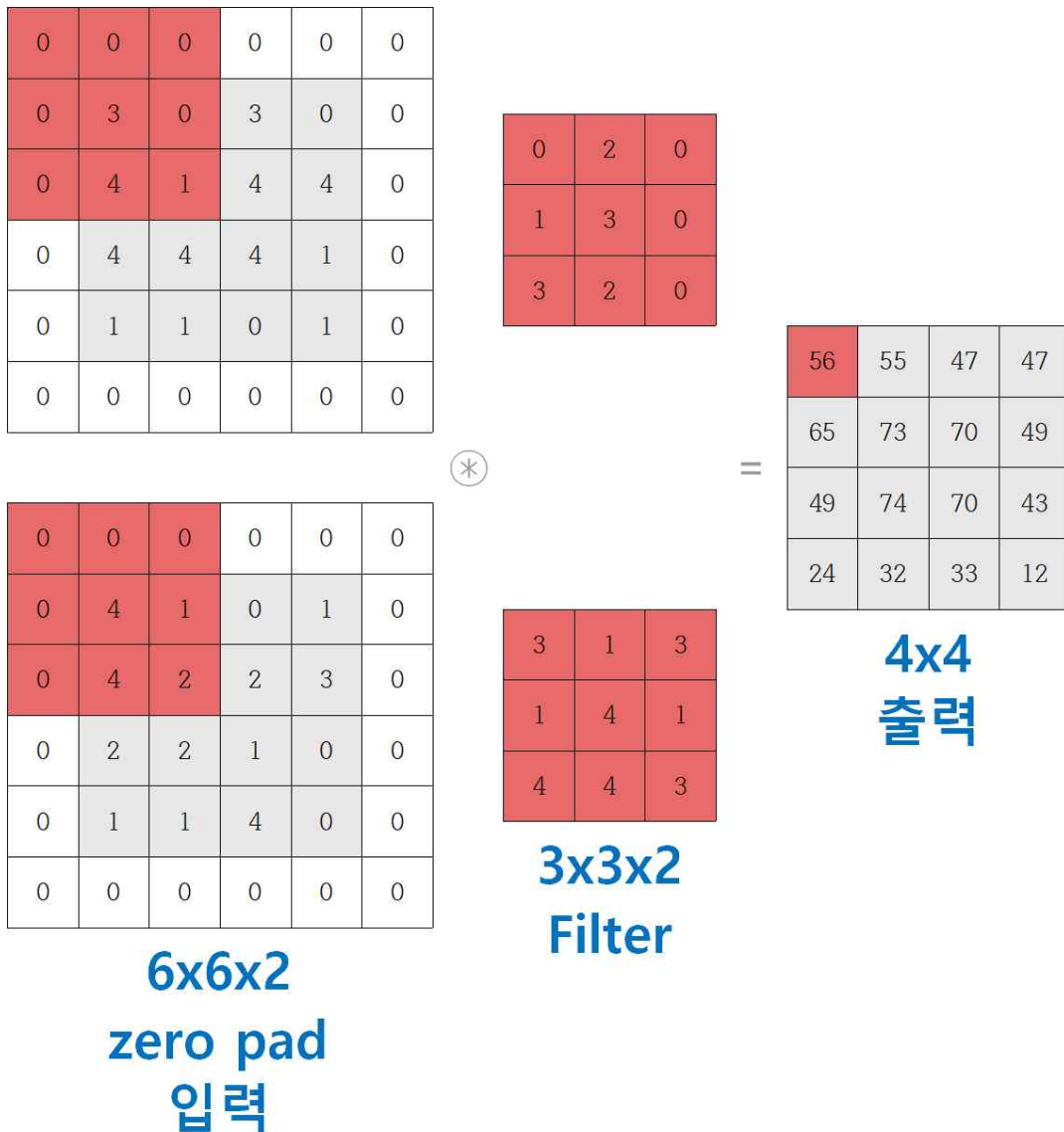
*** 독자의 이해를 돕기 위해 결과 화면을 편집하였습니다.

09 6x6x2 입력

다음은 4x4x2 입력 이미지를 상하 좌우로 1칸씩 zero padding을 하여 6x6x2 이미지로 변경한 후, 합성 곱을 수행한 결과를 보여줍니다.



6x6x2 입력 이미지는 3x3x2 필터와 합성 곱이 수행된 후, 4x4x1 출력 이미지를 생성합니다. 입력 이미지의 두께와 상관없이 같은 두께의 필터와 합성 곱이 수행된 후에는 하나의 결과 이미지를 생성하게 됩니다. 여기서 출력 이미지의 깊이는 생략되었습니다. 다음은 6x6x2 입력 이미지에 대해 구체적으로 수행되는 합성 곱 과정을 나타냅니다.



2층의 깊이로 구성된 6x6 입력 이미지 1개는 2층의 깊이로 구성된 3x3 필터 1개와 층별로 합성 곱이 수행된 후, 1층 깊이의 4x4 출력 이미지 1개를 생성해 내게 됩니다.

지금까지의 과정을 예제를 통해 정리해 봅니다.

1. 다음과 같이 예제를 작성합니다.

519_1.py

```
01 import numpy as np
02
03 np.random.seed(1)
04
05 image = np.random.randint(5, size=(4,4,2))
```

```

06 print('image_0 =\n', image[:, :, 0])
07 print('image_1 =\n', image[:, :, 1])
08
09 filter = np.random.randint(5, size=(3,3,2))
10 print('filter_0 =\n', filter[:, :, 0])
11 print('filter_1 =\n', filter[:, :, 1])
12
13 image_pad = np.pad(image, ((1,1), (1,1), (0,0)))
14 print('image_pad_0 =\n', image_pad[:, :, 0])
15 print('image_pad_1 =\n', image_pad[:, :, 1])
16
17 convolution = np.zeros((4,4))
18
19 for row in range(4):
20     for col in range(4):
21         window = image_pad[row:row+3, col:col+3]
22         convolution[row, col] = np.sum(window*filter)
23
24 print('convolution =\n', convolution)

```

05 : 0이상 5미만의 정수를 가진 4x4x2 행렬을 생성하여 image 변수에 할당합니다. 이 경우에는 2의 depth(깊이)를 갖는 4x4 행렬로 해석합니다.

06 : print 함수를 호출하여 image[:, :, 0] 행렬 값을 출력합니다.

07 : print 함수를 호출하여 image[:, :, 1] 행렬 값을 출력합니다.

13 : np.pad 함수를 호출하여 image 행렬과 같은 모양의 행렬을 내부적으로 생성한 후, 행을 상하로 한 칸씩, 열을 좌우로 한 칸씩 늘려 0으로 채운 후, image_pad 변수에 할당합니다. 깊이 부분은 늘리지 않습니다.

14 : print 함수를 호출하여 image_pad[:, :, 0] 행렬 값을 출력합니다.

15 : print 함수를 호출하여 image_pad[:, :, 1] 행렬 값을 출력합니다.

17 : np.zeros 함수를 호출하여 0값으로 채워진 4x4 행렬을 생성하여 convolution 변수에 할당합니다.


19 : row값을 0에서 4 미만까지 바꾸어가며 20~22줄을 4회 수행합니다.

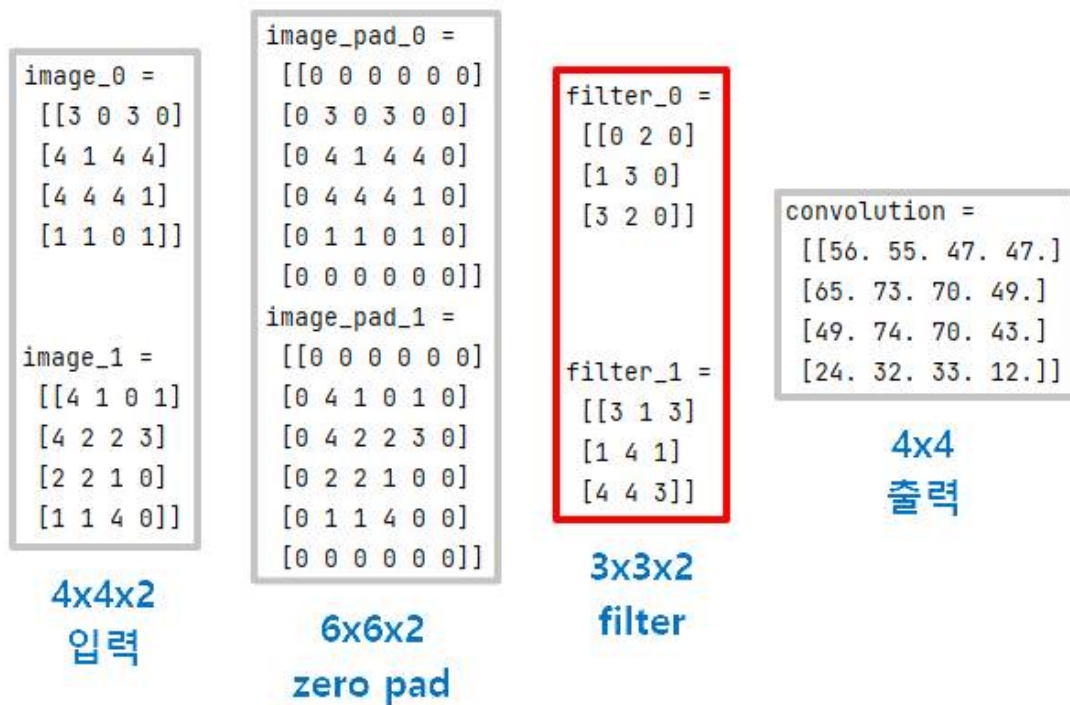
20 : col값을 0에서 4 미만까지 바꾸어가며 21~22줄을 4회 수행합니다.

21 : image_pad 행렬의 row 이상 (row+3) 미만까지, col 이상 (col+3) 미만까지의 행렬을 window 변수가 가리키게 합니다. 예를 들어, (row=0, col=0)일 경우 window 변수는 image_pad[0:3, 0:3] 행렬을 가리키게 됩니다.

22 : np.sum 함수를 호출하여 window*filter 행렬의 모든 항목의 값을 더해서 convolution 행렬의 (row, col) 항목에 할당합니다.

24 : print 함수를 호출하여 convolution 행렬 값을 출력합니다.

2.  버튼을 눌러 프로그램을 실행시킵니다. 다음은 실행 결과 화면입니다.

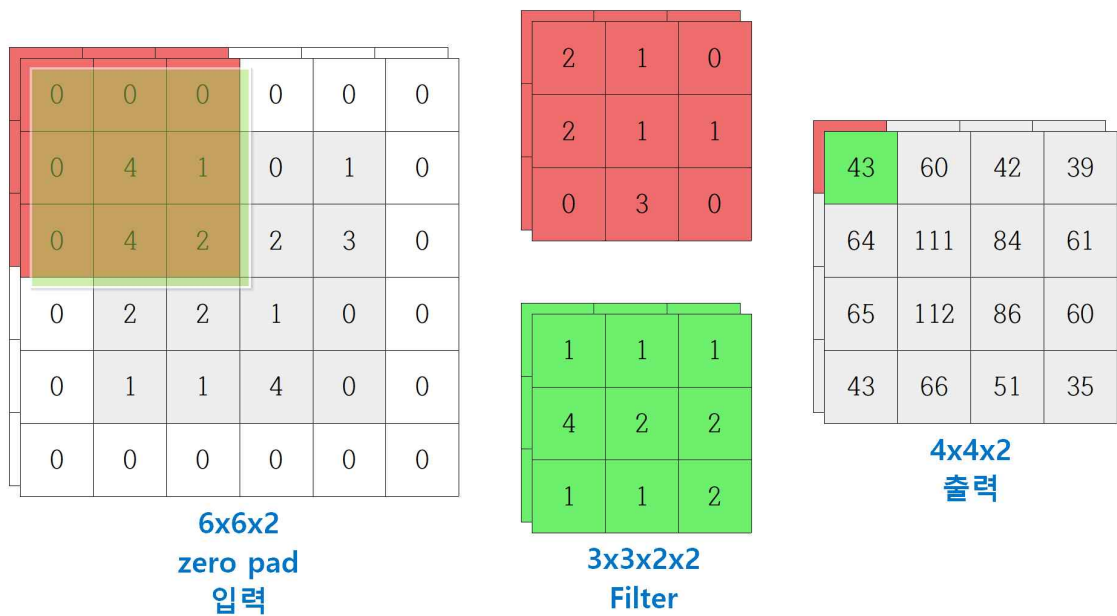


출력 결과를 앞의 그림과 비교하면서 살펴봅니다.

*** 독자의 이해를 돕기 위해 결과 화면을 편집하였습니다.

10 6x6x2 입력 필터 늘리기

다음은 6x6x2 입력 이미지에 대해 2개의 필터를 이용하여 2개의 출력 이미지를 얻는 과정을 보여줍니다. 필터는 이미지의 특징을 추출하는 역할을 하므로 여기서는 입력 이미지에서 2가지 특징을 추출하게 됩니다. 일반적으로 필터의 개수만큼 이미지의 특징이 추출됩니다. 필터의 두께는 입력 이미지의 두께와 같아야 합니다.



6x6x2 입력 이미지는 3x3x2 필터 2개와 합성 곱이 수행된 후, 4x4x2 출력 이미지를 생성합니다. 입력 이미지의 두께와 상관없이 같은 두께의 필터 하나와 합성 곱이 수행된 후에는 하나의 결과 이미지를 생성하게 됩니다. 여기서는 2개의 필터가 사용되었으므로 2개의 결과 이미지를 생성하며, 결과 이미지들은 겹쳐져 이미지의 개수만큼의 깊이를 갖는 하나의 이미지를 구성합니다. 다음은 6x6x2 입력 이미지에 대해 3x3x2 필터 2개를 적용하여 합성 곱을 수행하는 과정을 구체적으로 보여줍니다.

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

0	0	3
3	0	0
2	3	3

3	3	4
4	3	4
1	4	0

32	47	35	39
40	80	40	31
20	49	52	24
16	28	17	10

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

2	1	0
2	1	1
0	3	0

1	1	1
4	2	2
1	1	2

43	60	42	39
64	111	84	61
65	112	86	60
43	66	51	35

6x6x2
zero pad
입력

3x3x2x2
Filter

4x4x2
출력

2층의 깊이로 구성된 6x6 입력 이미지 1개는 2층의 깊이로 구성된 3x3 필터 2개와 층별로 합성 곱이 수행된 후, 2층 깊이의 4x4 출력 이미지를 생성해 내게 됩니다.

지금까지의 과정을 예제를 통해 정리해 봅니다.

1. 다음과 같이 예제를 작성합니다.

5110_1.py

```
01 import numpy as np
```

```

02
03 np.random.seed(1)
04
05 image = np.random.randint(5, size=(4,4,2))
06 print('image_0 =\n', image[:, :, 0])
07 print('image_1 =\n', image[:, :, 1])
08
09 filter = np.random.randint(5, size=(3,3,2,2))
10 print('filter_00 =\n', filter[:, :, 0, 0])
11 print('filter_10 =\n', filter[:, :, 1, 0])
12 print('filter_01 =\n', filter[:, :, 0, 1])
13 print('filter_11 =\n', filter[:, :, 1, 1])
14
15 image_pad = np.pad(image, ((1,1), (1,1), (0,0)))
16 print('image_pad_0 =\n', image_pad[:, :, 0])
17 print('image_pad_1 =\n', image_pad[:, :, 1])
18
19 convolution = np.zeros((4,4,2))
20
21 for fn in range(2):
22     for row in range(4):
23         for col in range(4):
24             window = image_pad[row:row+3, col:col+3]
25             convolution[row, col, fn] = np.sum(window*filter[:, :, :, fn])
26
27 print('convolution_0 =\n', convolution[:, :, 0])
28 print('convolution_1 =\n', convolution[:, :, 1])

```

09 : 0이상 5미만의 정수를 가진 3x3x2x2 행렬을 생성하여 filter 변수에 할당합니다. 이렇게 하면 2의 깊이를 갖는 3x3 크기의 필터 2개가 생성됩니다. 세 번째에 오는 2는 필터의 깊이로, 마지막에 오는 2는 필터의 개수로 해석합니다.

10 : print 함수를 호출하여 0번 filter의 0번 깊이를 출력해 봅니다.

11 : print 함수를 호출하여 0번 filter의 1번 깊이를 출력해 봅니다.

12 : print 함수를 호출하여 1번 filter의 0번 깊이를 출력해 봅니다.

13 : print 함수를 호출하여 1번 filter의 1번 깊이를 출력해 봅니다.

19 : np.zeros 함수를 호출하여 0값으로 채워진 4x4x2 행렬을 생성하여 convolution 변수에 할당합니다. 이 경우에는 2의 depth(깊이)를 갖는 4x4 행렬로 해석합니다. convolution 행렬의 깊이 2는 09줄에서 정의된 filter 행렬의 개수 2와 같아야 합니다.

21 : fn값을 0에서 2 미만까지 바꾸어가며 22~25줄을 2회 수행합니다. fn은 filter number를 의미합니다.

22 : row값을 0에서 4 미만까지 바꾸어가며 23~25줄을 4회 수행합니다.


23 : col값을 0에서 4 미만까지 바꾸어가며 24~25줄을 4회 수행합니다.

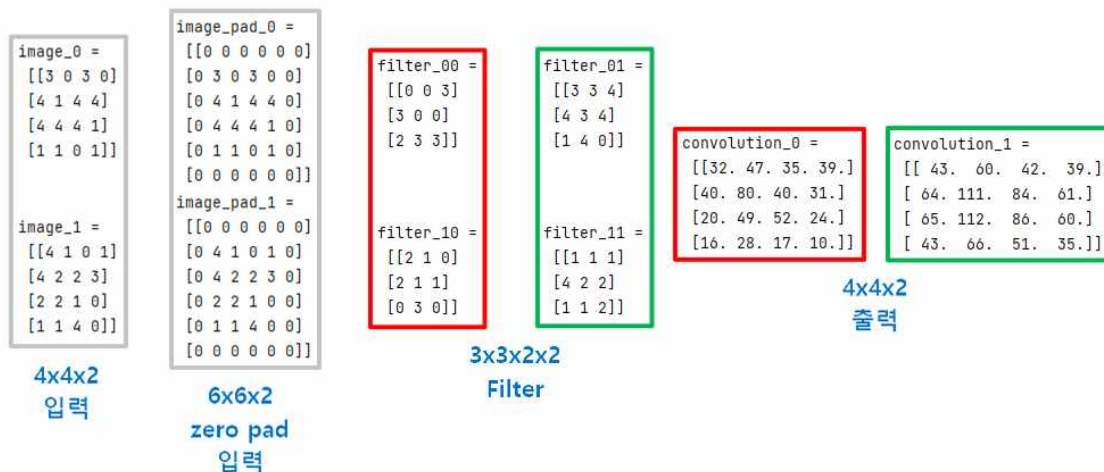
24 : image_pad 행렬의 row 이상 (row+3) 미만까지, col 이상 (col+3) 미만까지의 행렬을 window 변수가 가리키게 합니다. 예를 들어, (row=0, col=0)일 경우 window 변수는 image_pad[0:3, 0:3] 행렬을 가리키게 됩니다.

25 : np.sum 함수를 호출하여 window*filter[:, :, fn] 행렬의 모든 항목의 값을 더해 convolution 행렬의 (row, col, fn) 항목에 할당합니다.

27 : print 함수를 호출하여 convolution[:, :, 0] 행렬 값을 출력합니다.

28 : print 함수를 호출하여 convolution[:, :, 1] 행렬 값을 출력합니다.

2.  버튼을 눌러 프로그램을 실행시킵니다. 다음은 실행 결과 화면입니다.



출력 결과를 앞의 그림과 비교하면서 살펴봅니다.

*** 독자의 이해를 돕기 위해 결과 화면을 편집하였습니다.

11 6x6x3 입력

여기서는 6x6x3 입력 이미지에 대해 2개의 필터를 이용하여 2개의 출력 이미지를 얻는 예제를 수행하여 봅니다.

1. 다음과 같이 예제를 작성합니다.

5111_1.py

```
01 import numpy as np
02
03 np.random.seed(1)
04
05 image = np.random.randint(5, size=(4,4,3))
06 print('image_0 =\n', image[:, :, 0])
07 print('image_1 =\n', image[:, :, 1])
```

```

08 print('image_2 =\n', image[:, :, 2])
09
10 filter = np.random.randint(5, size=(3,3,3,2))
11 print('filter_00 =\n', filter[:, :, 0, 0])
12 print('filter_01 =\n', filter[:, :, 1, 0])
13 print('filter_02 =\n', filter[:, :, 2, 0])
14 print('filter_10 =\n', filter[:, :, 0, 1])
15 print('filter_11 =\n', filter[:, :, 1, 1])
16 print('filter_12 =\n', filter[:, :, 2, 1])
17
18 image_pad = np.pad(image, ((1,1), (1,1), (0,0)))
19 print('image_pad_0 =\n', image_pad[:, :, 0])
20 print('image_pad_1 =\n', image_pad[:, :, 1])
21 print('image_pad_2 =\n', image_pad[:, :, 2])
22
23 convolution = np.zeros((4,4,2))
24
25 for fn in range(2):
26     for row in range(4):
27         for col in range(4):
28             window = image_pad[row:row+3, col:col+3]
29             convolution[row, col, fn] = np.sum(window*filter[:, :, :, fn])
30
31 print('convolution_0 =\n', convolution[:, :, 0])
32 print('convolution_1 =\n', convolution[:, :, 1])

```

05 : 입력 이미지의 깊이를 3으로 바꿔 줍니다.

08 : 깊이 2의 이미지 출력 부분을 추가합니다.


10 : 필터의 깊이를 3으로 바꿔줍니다. 필터의 깊이는 입력 이미지의 깊이와 같아야 합니다.

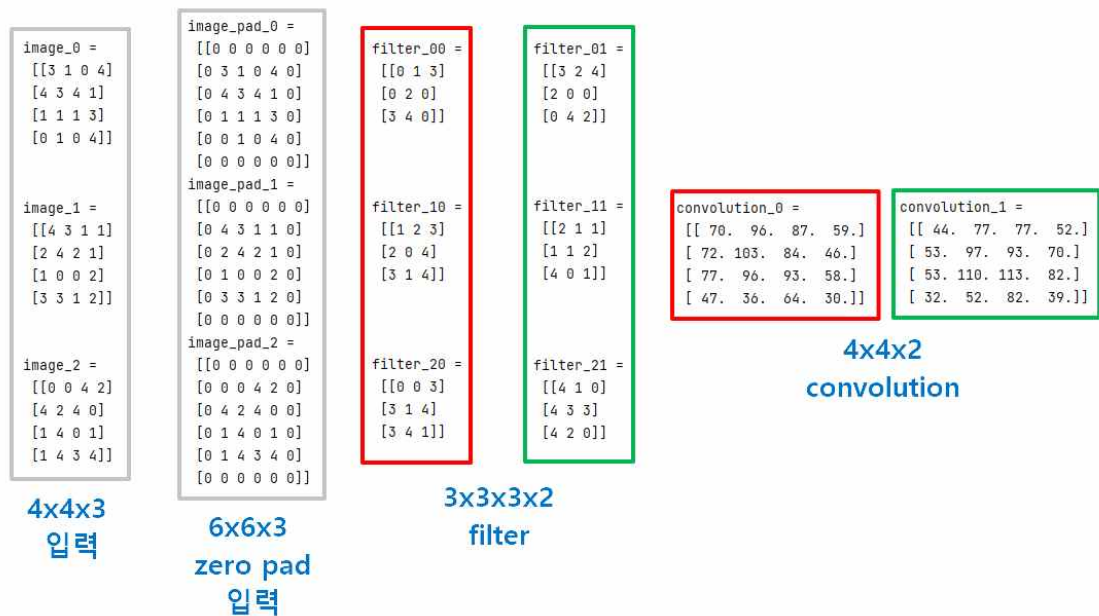
13 : 0번 filter의 2번 깊이 출력 부분을 추가합니다.

16 : 1번 filter의 2번 깊이 출력 부분을 추가합니다.

21 : image_pad의 2번 깊이 출력 부분을 추가합니다.

23 : np.zeros 함수를 호출하여 0값으로 채워진 4x4x2 행렬을 생성하여 convolution 변수에 할당합니다. convolution 행렬의 깊이 2는 10줄에서 정의된 filter 행렬의 개수 2와 같아야 합니다.

2.  버튼을 눌러 프로그램을 실행시킵니다. 다음은 실행 결과 화면입니다.

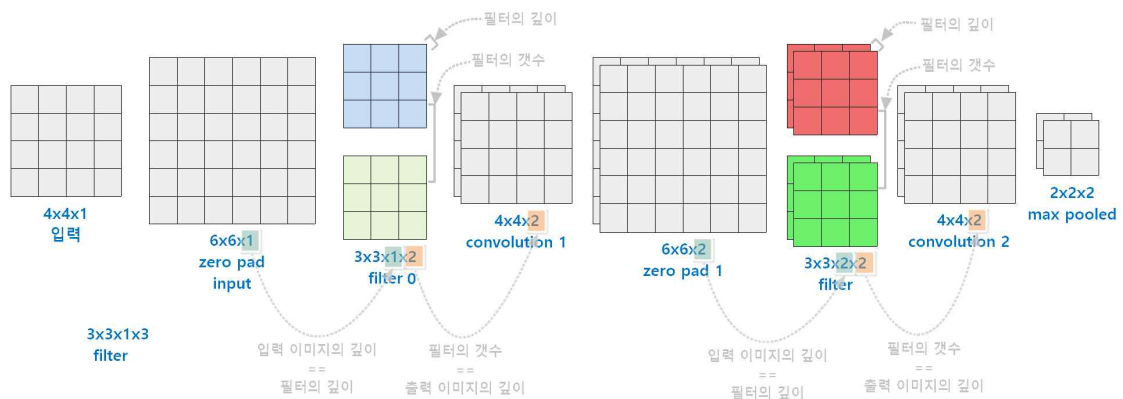


입력의 깊이는 필터의 깊이를 결정하여 필터의 개수는 출력의 깊이를 결정합니다.

*** 독자의 이해를 돕기 위해 결과 화면을 편집하였습니다.

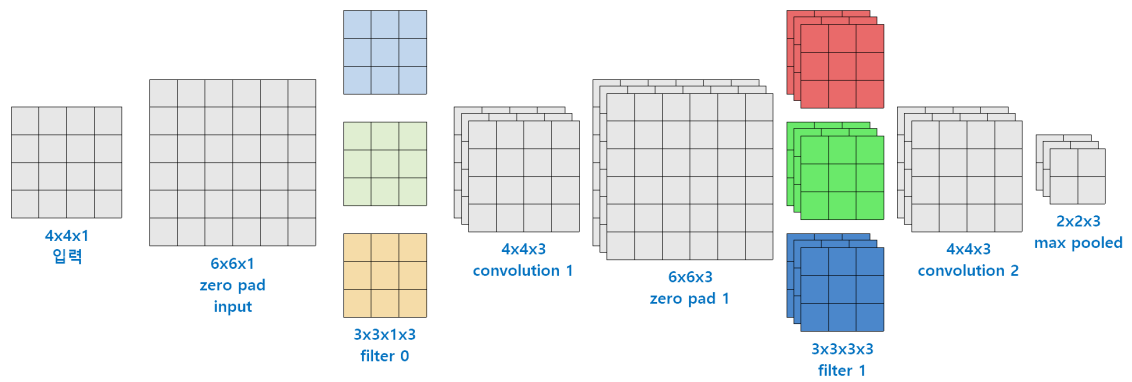
12 필터의 깊이와 개수

다음 그림은 입력 이미지의 깊이와 필터의 깊이, 필터의 개수와 출력 이미지의 깊이의 관계를 보여줍니다.



입력의 깊이는 필터의 깊이를 결정하여 필터의 개수는 출력의 깊이를 결정합니다.

다음 그림은 필터의 개수가 늘어났을 때, 출력의 깊이가 늘어나는 상황을 보여줍니다.



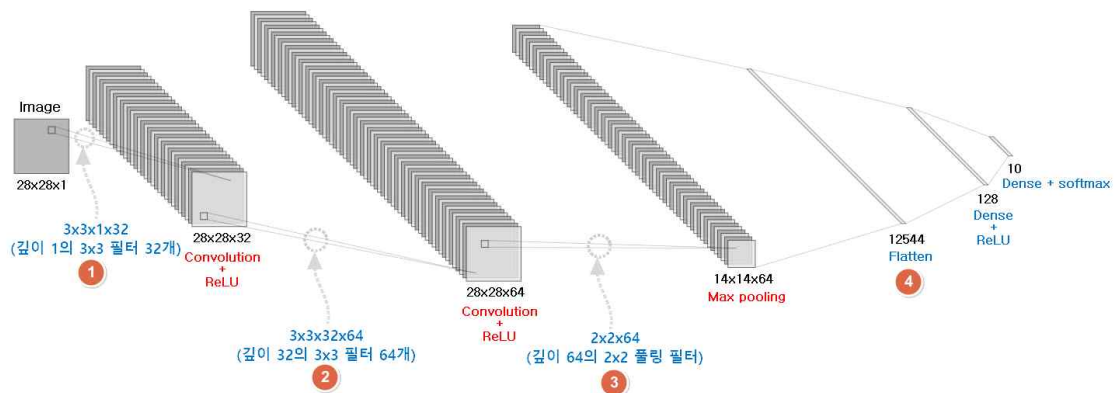
이상 CNN의 순전파 과정을 살펴보았습니다. 뒤에서 우리는 CNN 역전파 과정도 살펴보입니다.

02 CNN 활용 맛보기

앞에서 우리는 CNN의 주요 동작을 살펴보고 구현해 보았습니다. 여기서는 CNN 인공 신경망을 학습시켜 숫자와 그림을 인식해 보도록 합니다. CNN 인공 신경망은 앞에서 살펴본 완전 연결 인공 신경망에 비해 이미지 인식률이 훨씬 높습니다. 그러나 학습 시간은 훨씬 더 길다는 단점이 있습니다.

01 Conv2D-Conv2D-MaxPooling2D

다음은 합성곱 2단계, 모으기 1단계, 완전 연결 층으로 구성된 인공 신경망입니다.



이 신경망의 입력 층은 28x28x1의 이미지입니다. 입력 층의 깊이가 1이므로 사용할 필터의 깊이도 1이 되어야 합니다. 1차 합성곱 출력 층은 28x28x32로 깊이가 32입니다. 출력 층의 깊이가 32일 경우 사용할 필터의 개수는 32개가 되어야 합니다. ❶ 그래서 필터는 3x3x1x32의 4차원 행렬이 됩니다. 1차 합성곱은 ReLU 함수를 거쳐 2차 합성곱의 입력이 됩니다. 2차 합성곱 입력 층의 깊이는 32이므로 사용할 필터의 깊이도 32가 되어야 합니다. 2차 합성곱 출력 층은 28x28x64로 깊이가 64입니다. 출력 층의 깊이가 64이므로 사용할 필터의 개수는 64개가 되어야 합니다. ❷ 그래서 필터는 3x3x32x64의 4차원 행렬이 됩니다. 2차 합성곱은 ReLU 함수를 거쳐 모으기의 입력이 됩니다. ❸ 모으기에 사용되는 필터는 2x2x64를 사용하여 모으기 출력 이미지는 14x14x64가 됩니다. ❹ 모으기 출력 이미지는 Flatten 과정을 거쳐 14x14x64 = 12544개의 1차원 이미지로 재구성됩니다.

이 신경망을 예제를 통해 살펴봅니다.

1. 다음과 같이 예제를 작성합니다.

521_1.py

```
01 import tensorflow as tf
02
03 mnist = tf.keras.datasets.fashion_mnist
```

```

04
05 (x_train, y_train), (x_test, y_test) = mnist.load_data()
06 x_train, x_test = x_train / 255.0, x_test / 255.0
07 x_train = x_train.reshape((60000, 28, 28, 1))
08 x_test = x_test.reshape((10000, 28, 28, 1))
09
10 model = tf.keras.Sequential([
11     tf.keras.layers.InputLayer(input_shape=(28, 28, 1)),
12     tf.keras.layers.Conv2D(32, (3, 3), activation='relu', padding='same'),
13     tf.keras.layers.Conv2D(64, (3, 3), activation='relu', padding='same'),
14     tf.keras.layers.MaxPooling2D((2, 2)),
15     tf.keras.layers.Flatten(),
16     tf.keras.layers.Dense(128, activation='relu'),
17     tf.keras.layers.Dense(10, activation='softmax')
18 ])
19
20 model.summary()
21
22 model.compile(optimizer='adam',
23               loss='sparse_categorical_crossentropy',
24               metrics=['accuracy'])
25
26 model.fit(x_train, y_train, epochs=5)
27
28 model.evaluate(x_test, y_test)

```

01 : import문을 이용하여 tensorflow 모듈을 tf라는 이름으로 불러옵니다. tensorflow 모듈은 구글에서 제공하는 인공 신경망 라이브러리입니다.

03 : mnist 변수를 생성한 후, tf.keras.datasets.fashion_mnist 모듈을 가리키게 합니다. fashion_mnist 모듈은 신발, 옷, 가방 등의 패션 데이터를 가진 모듈입니다. fashion_mnist 모듈에는 6만개의 학습용 패션 데이터와 1만개의 시험용 패션 데이터가 있습니다.

05 : mnist.load_data 함수를 호출하여 패션 데이터를 읽어와 x_train, y_train, x_test, y_test 변수가 가리키게 합니다. x_train, x_test 변수는 각각 6만개의 학습용 패션 데이터와 1만개의 시험용 패션 데이터를 가리킵니다. y_train, y_test 변수는 각각 6만개의 학습용 패션 데이터 라벨과 1만개의 시험용 패션 데이터 라벨을 가리킵니다.

06 : x_train, x_test 변수가 가리키는 6만개, 1만개의 그림은 각각 28x28 픽셀로 구성된 그림이며, 1픽셀의 크기는 8비트로 0에서 255사이의 숫자를 가집니다. 모든 픽셀의 숫자를 255.0으로 나누어 각 픽셀을 0.0에서 1.0사이의 실수로 바꾸어 인공 신경망에 입력하게 됩니다.

07, 08 : x_train, x_test 변수가 가리키는 6만개, 1만개의 그림은 각각 28x28 픽셀, 28x28 픽셀로 구성되어 있습니다. 이 예제에서 소개하는 인공 신경망의 경우 그림 데이터를 입력할 때 28x28 픽셀을 28x28x1로 변경하여 입력하게 됩니다. 그래서 11줄에 있는 InputLayer 클

래스는 28x28x1을 입력으로 받도록 구성됩니다. 28x28x1은 1의 깊이를 갖는 28x28 크기의 이미지를 의미합니다.

10~18 : tf.keras.Sequential 클래스를 이용하여 CNN 인공 신경망을 생성합니다.

11 : tf.keras.layers.InputLayer 함수를 이용하여 내부적으로 keras 라이브러리에서 제공하는 tensor를 생성하고, 입력의 모양을 정해줍니다.

12 : tf.keras.layers.Conv2D 클래스를 이용하여 합성 곱 신경망 층을 생성합니다. 합성 곱 출력 층의 개수는 32개(즉, 출력 층의 깊이는 32), 사용할 필터의 크기는 3x3, 활성화 함수는 relu, padding을 'same'으로 설정하여 출력 이미지의 크기를 입력 이미지의 크기와 같게 합니다. 앞에서 살펴보았던 padding zero의 동작을 수행합니다. 여기서 입력 이미지의 깊이는 1이기 때문에 3x3 필터의 깊이도 1이 되어야 하며, 합성 곱 출력 층의 개수가 32개(즉, 출력 층의 깊이는 32)이기 때문에 필터의 개수는 32개가 되어야 합니다. 즉, 깊이 1의 3x3 필터가 32개가 생성됩니다. 필터의 개수는 출력 층의 깊이와 같아야 합니다. 이 층에서 필요한 가중치 변수의 개수는 $3 \times 3 \times 1 \times 32 = 288$ 개, 편향 변수의 개수는 32개가 되어 총 320개의 매개변수가 필요합니다. 편향의 개수는 필터의 개수와 같아야 합니다.

13 : tf.keras.layers.Conv2D 클래스를 이용하여 합성 곱 신경망 층을 생성합니다. 합성 곱 출력 층의 개수는 64개(즉, 출력 층의 깊이는 64), 사용할 필터의 크기는 3x3, 활성화 함수는 relu, padding을 'same'으로 설정하여 출력 이미지의 크기를 입력 이미지의 크기와 같게 합니다. 여기서 입력 이미지의 깊이는 32이기 때문에 3x3 필터의 깊이도 32가 되어야 하며, 합성 곱 출력 층의 개수가 64개(즉, 출력 층의 깊이는 64)이기 때문에 필터의 개수는 64개가 되어야 합니다. 즉, 깊이 32의 3x3 필터가 64개가 생성됩니다. 필터의 개수는 출력 층의 깊이와 같아야 합니다. 이 층에서 필요한 가중치 변수의 개수는 $3 \times 3 \times 32 \times 64 = 18432$ 개, 편향 변수의 개수는 64개가 되어 총 18496개의 매개변수가 필요합니다. 편향의 개수는 필터의 개수와 같아야 합니다.

14 : tf.keras.layers.MaxPooling2D 클래스를 이용하여 풀링 층을 생성합니다. 풀링 층에 사용할 풀링 필터의 크기는 2x2입니다. 이전 합성 곱 층의 깊이가 64이므로 풀링 층의 깊이도 64가 되어야 하며, 크기는 반으로 줄어 14x14가 됩니다.

15 : tf.keras.layers.Flatten 클래스를 이용하여 이전 풀링 층을 완전 연결 층의 입력 층으로 모양을 변경할 수 있도록 합니다. 이 과정에서 $14 \times 14 \times 64 = 12544$ 개의 입력 노드로 변경됩니다.

16 : tf.keras.layers.Dense 클래스를 이용하여 신경망 층을 생성합니다. 여기서는 단위 인공 신경 128개를 생성합니다. 활성화 함수는 'relu' 함수를 사용합니다.

17 : tf.keras.layers.Dense 클래스를 이용하여 신경망 층을 생성합니다. 여기서는 단위 인공 신경 10개를 생성합니다. 활성화 함수는 'softmax' 함수를 사용합니다.


20 : model.summary 함수를 호출하여 생성된 모델의 개요를 출력합니다.

22~24 : model.compile 함수를 호출하여 내부적으로 인공 신경망을 구성합니다. 인공 신경망을 구성할 때에는 적어도 2개의 함수를 정해야 합니다. loss 함수와 optimizer 함수, 즉, 손실 함수와 최적화 함수를 정해야 합니다. 손실 함수로는 sparse_categorical_crossentropy 함수를 사용하고 최적화 함수는 adam 함수를 사용합니다. fit 함수 로그에는 기본적으로 손실 값만 표시됩니다. metrics 매개 변수는 학습 측정 항목 함수를 전달할 때 사용합니다. 'accuracy'는 학습의 정확도를 출력해 줍니다.

26 : model.fit 함수를 호출하여 인공 신경망에 대한 학습을 시작합니다. fit 함수에는 학습을

몇 회 수행할지도 입력해 줍니다. epochs는 학습 횟수를 의미하며, 여기서는 5회 학습을 수행하도록 합니다.

28 : model.evaluate 함수를 호출하여 인공 신경망의 학습 결과를 평가합니다. 여기서는 학습이 끝난 인공 신경망 함수에 x_test 값을 주어 학습 결과를 평가해 봅니다.

2.  버튼을 눌러 프로그램을 실행시킵니다. 다음은 실행 결과 화면입니다.

Model: "sequential"

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 28, 28, 32)	320 1
conv2d_1 (Conv2D)	(None, 28, 28, 64)	18496 2
max_pooling2d (MaxPooling2D)	(None, 14, 14, 64)	0
flatten (Flatten)	(None, 12544) 3	0
dense (Dense)	(None, 128)	1605760 4
dense_1 (Dense)	(None, 10)	1290 5
Total params: 1,625,866 6		
Trainable params: 1,625,866		
Non-trainable params: 0		

❶ 1차 합성곱 단계에서 사용되는 가중치는 $3 \times 3 \times 1 \times 32$ 의 크기를 갖습니다. 편향은 가중치의 개수만큼 필요하여 32개가 됩니다. 그래서 1차 합성곱에 필요한 매개변수의 총 개수는 $(3 \times 3 \times 1 \times 32 + 32) = 320$ 이 됩니다. ❷ 2차 합성곱 단계에서 사용되는 가중치는 $3 \times 3 \times 32 \times 64$ 의 크기를 갖습니다. 편향은 가중치의 개수만큼 필요하여 64개가 됩니다. 그래서 2차 합성곱에 필요한 매개변수의 총 개수는 $(3 \times 3 \times 32 \times 64 + 64) = 18496$ 이 됩니다. ❸ Flatten 층은 완전 연결 층의 입력 층으로 $14 \times 14 \times 64 = 12544$ 개의 노드로 구성됩니다. ❹ 은닉 층의 노드는 128개로 Flatten 층과 은닉 층 연결에 필요한 가중치의 개수는 12544×128 이 됩니다. 편향은 가중치의 개수만큼 필요하여 128개가 됩니다. 그래서 Flatten 층과 은닉 층 연결에 필요한 매개변수의 총 개수는 $(12544 \times 128 + 128) = 1605760$ 이 됩니다. ❺ 출력 층의 노드는 10개로 은닉 층과 출력 층 연결에 필요한 가중치의 개수는 128×10 이 됩니다. 편향은 가중치의 개수만큼 필요하여 10개가 됩니다. 그래서 은닉 층과 출력 층 연결에 필요한 매개변수의 총 개수는 $(128 \times 10 + 10) = 1290$ 이 됩니다. ❻ 그래서 학습의 대상이 되는 매개변수의 총 개수는 1625866이 됩니다.

```

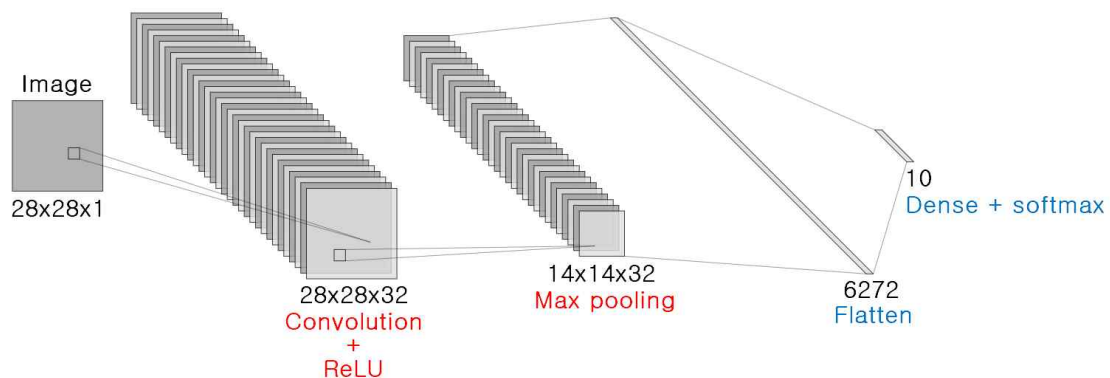
Epoch 1/5
1875/1875 [=====] - 73s 39ms/step - loss: 0.3504 - accuracy: 0.8744 ①
Epoch 2/5
1875/1875 [=====] - 71s 38ms/step - loss: 0.2155 - accuracy: 0.9206
Epoch 3/5
1875/1875 [=====] - 72s 38ms/step - loss: 0.1614 - accuracy: 0.9402
Epoch 4/5
1875/1875 [=====] - 72s 39ms/step - loss: 0.1191 - accuracy: 0.9556
Epoch 5/5
1875/1875 [=====] - 71s 38ms/step - loss: 0.0837 - accuracy: 0.9690 ②
313/313 [=====] - 2s 7ms/step - loss: 0.2573 - accuracy: 0.9240 ③

```

패션 MNIST 데이터의 학습 과정의 정확도는 ① 87.44%에서 시작해서 ② 96.90%로 끝납니다. 완전 연결 인공 신경망에 비해 7% 정도 정확도가 높습니다. ③ 학습이 끝난 후에, evaluate 함수로 시험 데이터를 예측한 결과는 92.40%로 학습 데이터의 예측 결과에 비해 정확도가 떨어진 상태입니다. 시험 데이터에 대한 예측 결과도 완전 연결 인공 신경망에 비해 6% 정도 정확도가 높습니다.

02 Conv2D-MaxPooling2D

다음은 합성곱 1단계, 모으기 1단계, 완전 연결 층으로 구성된 인공 신경망입니다.



이전 인공 신경망에 비해 합성곱 층은 1단계, 완전 연결 층의 은닉 층이 빠진 상태입니다.

이 신경망을 예제를 통해 살펴봅시다.

1. 다음과 같이 예제를 수정합니다.

522_1.py

```


10 model = tf.keras.Sequential([
11     tf.keras.layers.InputLayer(input_shape=(28, 28, 1)),
12     tf.keras.layers.Conv2D(32, (3, 3), activation='relu', padding='same'),
13     tf.keras.layers.MaxPooling2D((2, 2)),
14     tf.keras.layers.Flatten(),

```

```
15     tf.keras.layers.Dense(10, activation='softmax')
16 ])
```

13 : tf.keras.layers.MaxPooling2D 클래스를 이용하여 풀링 층을 생성합니다. 풀링 층에 사용할 풀링 필터의 크기는 2x2입니다. 이전 합성곱 층의 깊이가 32이므로 풀링 층의 깊이도 32가 되어야 하며, 이미지의 크기는 반으로 줄어 14x14가 됩니다.

14 : tf.keras.layers.Flatten 클래스를 이용하여 이전 풀링 층을 완전 연결 층의 입력 층으로 모양을 변경합니다. 이 과정에서 $14 \times 14 \times 32 = 6272$ 개의 입력 노드로 변경됩니다.

2.  버튼을 눌러 프로그램을 실행시킵니다. 다음은 실행 결과 화면입니다.

Model: "sequential"

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 28, 28, 32)	320 1
max_pooling2d (MaxPooling2D)	(None, 14, 14, 32)	0
flatten (Flatten)	(None, 6272) 2	0
dense (Dense)	(None, 10)	62730 3
Total params: 63,050 4		
Trainable params: 63,050		
Non-trainable params: 0		

1 합성곱 단계에서 사용되는 가중치는 $3 \times 3 \times 1 \times 32$ 의 크기를 갖습니다. 편향은 가중치의 개수만큼 필요하여 32개가 됩니다. 그래서 합성곱에 필요한 매개변수의 총 개수는 $(3 \times 3 \times 1 \times 32 + 32) = 320$ 이 됩니다. **2** Flatten 층은 완전 연결 층의 입력 층으로 $14 \times 14 \times 32 = 6272$ 개의 노드로 구성됩니다. **3** 출력 층의 노드는 10개로 Flatten 층과 출력 층 연결에 필요한 가중치의 개수는 6272×10 이 됩니다. 편향은 가중치의 개수만큼 필요하여 10개가 됩니다. 그래서 Flatten 층과 출력 층 연결에 필요한 매개변수의 총 개수는 $(6272 \times 10 + 10) = 62730$ 이 됩니다. **4** 그래서 학습의 대상이 되는 매개변수의 총 개수는 63050이 됩니다.

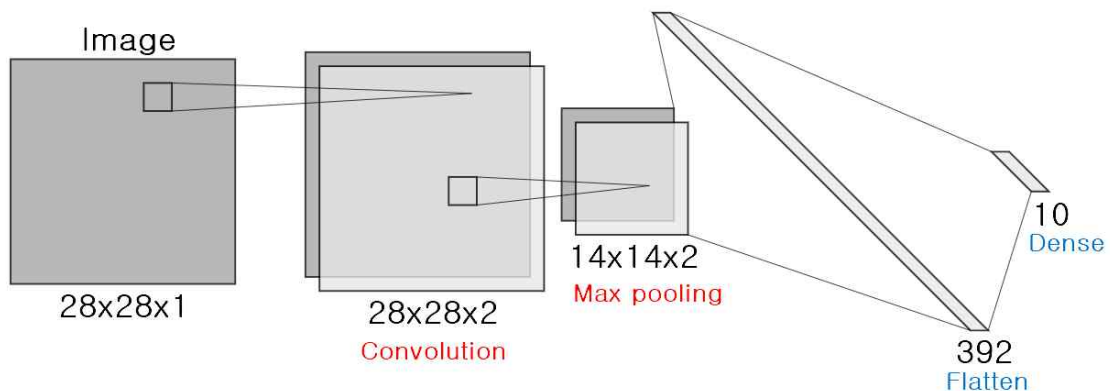
```
Epoch 1/5
1875/1875 [=====] - 12s 6ms/step - loss: 0.4410 - accuracy: 0.8452 1
Epoch 2/5
1875/1875 [=====] - 11s 6ms/step - loss: 0.3105 - accuracy: 0.8911
Epoch 3/5
1875/1875 [=====] - 12s 6ms/step - loss: 0.2745 - accuracy: 0.9039
Epoch 4/5
1875/1875 [=====] - 11s 6ms/step - loss: 0.2501 - accuracy: 0.9120
Epoch 5/5
1875/1875 [=====] - 10s 5ms/step - loss: 0.2342 - accuracy: 0.9174 2
313/313 [=====] - 1s 2ms/step - loss: 0.2973 - accuracy: 0.8962 3
```

패션 MNIST 데이터의 학습 과정의 정확도는 **1** 84.52%에서 시작해서 **2** 91.74%로 끝납니

다. 완전 연결 인공 신경망에 비해 2% 정도 정확도가 높습니다. ❸ 학습이 끝난 후에, evaluate 함수로 시험 데이터를 예측한 결과는 89.62%로 학습 데이터의 예측 결과에 비해 정확도가 떨어진 상태입니다. 시험 데이터에 대한 예측 결과도 완전 연결 인공 신경망에 비해 2% 정도 정확도가 높습니다.

03 필터 개수 줄여보기

다음은 합성 곱 1단계, 모으기 1단계, 완전 연결 층으로 구성된 인공 신경망입니다.



이전 인공 신경망에 비해 합성 곱 층에 사용하는 필터의 개수를 많이 줄인 상태입니다.

이 신경망을 예제를 통해 살펴봅니다.

1. 다음과 같이 예제를 수정합니다.

523_1.py


```
10 model = tf.keras.Sequential([
11     tf.keras.layers.InputLayer(input_shape=(28, 28, 1)),
12     tf.keras.layers.Conv2D(2, (3, 3), activation='relu', padding='same'),
13     tf.keras.layers.MaxPooling2D((2, 2)),
14     tf.keras.layers.Flatten(),
15     tf.keras.layers.Dense(10, activation='softmax')
16 ])
```

12 : tf.keras.layers.Conv2D 클래스를 이용하여 합성 곱 신경망 층을 생성합니다. 합성 곱 출력 층의 개수는 2개(즉, 출력 층의 깊이는 2), 사용할 필터의 크기는 3x3, 활성화 함수는 relu, padding을 'same'으로 설정하여 출력 이미지의 크기를 입력 이미지의 크기와 같게 합니다. 여기서 입력 이미지의 깊이는 1이기 때문에 3x3 필터의 깊이도 1이 되어야 하며, 합성 곱 출력 층의 개수가 2개(즉, 출력 층의 깊이는 2)이기 때문에 필터의 개수는 2개가 되어야 합니다. 즉, 깊이 1의 3x3 필터가 2개가 생성됩니다. 필터의 개수는 출력 층의 깊이와 같아야 합니다. 이 층에서 필요한 가중치 변수의 개수는 2*3*3*1=18개, 편향 변수의 개수는 2개가

되어 총 20개의 매개변수가 필요합니다. 편향의 개수는 필터의 개수와 같아야 합니다.

13 : `tf.keras.layers.MaxPooling2D` 클래스를 이용하여 풀링 층을 생성합니다. 풀링 층에 사용할 풀링 필터의 크기는 2×2 입니다. 이전 합성곱 층의 깊이가 2이므로 풀링 층의 깊이도 2가 되어야 하며, 이미지의 크기는 반으로 줄어 14×14 가 됩니다.

14 : `tf.keras.layers.Flatten` 클래스를 이용하여 이전 풀링 층을 완전 연결 층의 입력 층으로 모양을 변경합니다. 이 과정에서 $14 \times 14 \times 2 = 392$ 개의 입력 노드로 변경됩니다.

2.  버튼을 눌러 프로그램을 실행시킵니다. 다음은 실행 결과 화면입니다.

Model: "sequential"

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 28, 28, 2)	20 ❶
max_pooling2d (MaxPooling2D)	(None, 14, 14, 2)	0
flatten (Flatten)	(None, 392) ❷	0
dense (Dense)	(None, 10)	3930 ❸
Total params: 3,950 ❹		
Trainable params: 3,950		
Non-trainable params: 0		

❶ 합성곱 단계에서 사용되는 가중치는 $3 \times 3 \times 1 \times 2$ 의 크기를 갖습니다. 편향은 가중치의 개수만큼 필요하여 2개가 됩니다. 그래서 합성곱에 필요한 매개변수의 총 개수는 $(3 \times 3 \times 1 \times 2 + 2) = 20$ 이 됩니다. ❷ Flatten 층은 완전 연결 층의 입력 층으로 $14 \times 14 \times 2 = 392$ 개의 노드로 구성됩니다. ❸ 출력 층의 노드는 10개로 Flatten 층과 출력 층 연결에 필요한 가중치의 개수는 392×10 이 됩니다. 편향은 가중치의 개수만큼 필요하여 10개가 됩니다. 그래서 Flatten 층과 출력 층 연결에 필요한 매개변수의 총 개수는 $(392 \times 10 + 10) = 3930$ 이 됩니다. ❹ 그래서 학습의 대상이 되는 매개변수의 총 개수는 3950이 됩니다.

```
Epoch 1/5
1875/1875 [=====] - 7s 3ms/step - loss: 0.5914 - accuracy: 0.7950 ❶
Epoch 2/5
1875/1875 [=====] - 6s 3ms/step - loss: 0.4239 - accuracy: 0.8516
Epoch 3/5
1875/1875 [=====] - 6s 3ms/step - loss: 0.3995 - accuracy: 0.8599
Epoch 4/5
1875/1875 [=====] - 6s 3ms/step - loss: 0.3881 - accuracy: 0.8629
Epoch 5/5
1875/1875 [=====] - 6s 3ms/step - loss: 0.3783 - accuracy: 0.8664 ❷
313/313 [=====] - 1s 2ms/step - loss: 0.4031 - accuracy: 0.8582 ❸
```

패션 MNIST 데이터의 학습 과정의 정확도는 ❶ 79.50%에서 시작해서 ❷ 86.64%로 끝납니

다. 완전 연결 인공 신경망에 비해 3% 정도 정확도가 낮습니다. ❸ 학습이 끝난 후에, `evaluate` 함수로 시험 데이터를 예측한 결과는 85.82%로 학습 데이터의 예측 결과에 비해 정확도가 떨어진 상태입니다. 시험 데이터에 대한 예측 결과도 완전 연결 인공 신경망에 비해 1% 정도 정확도가 낮습니다.