

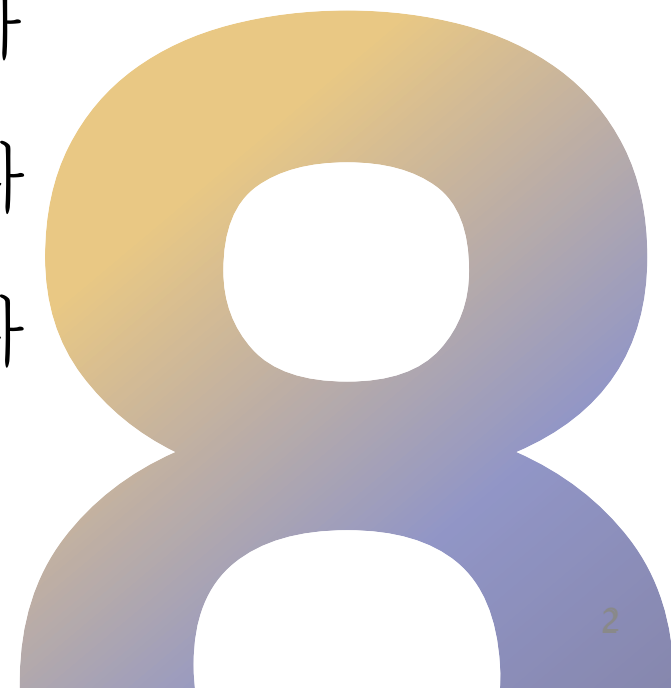
08

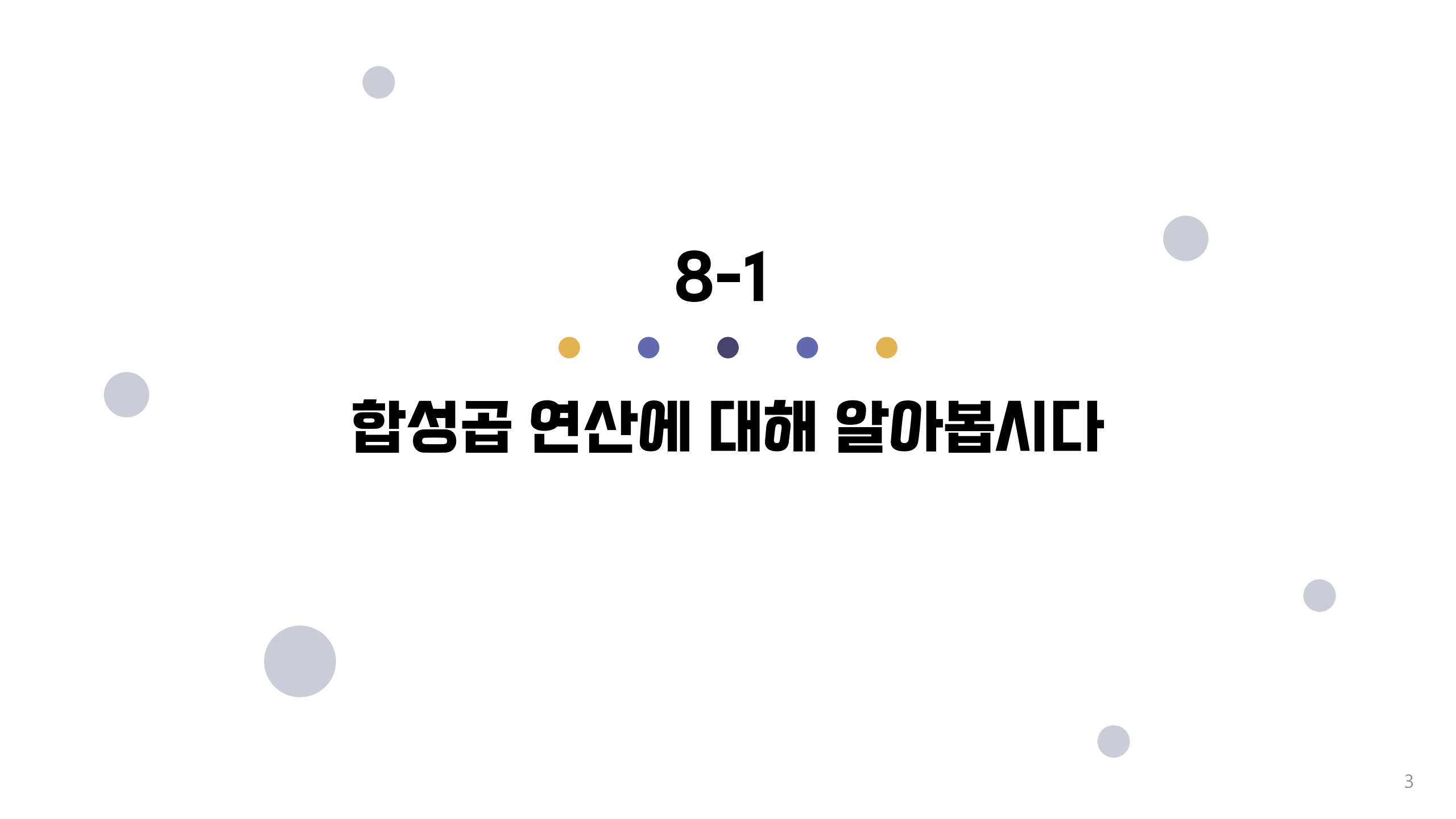
# 이미지를 분류합니다

4팀  
신원 서용민

# I N D E X

- 8-1 ● 합성곱 연산에 대해 알아봅니다
- 8-2 ● 풀링 연산에 대해 알아봅니다
- 8-3 ● 합성곱 신경망의 구조를 알아봅니다
- 8-4 ● 합성곱 신경망을 만들고 훈련합니다
- 8-5 ● 케라스로 합성곱 신경망을 만듭니다





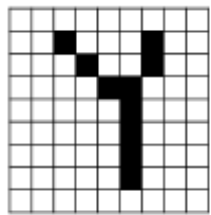
8-1

**합성곱 연산에 대해 알아보시다**

# 다층 신경망 & 합성곱 신경망

## 다층 신경망

앞 장의 다층 신경망(완전 연결 신경망)은 이미지를 1차원 데이터로 변환한 후 입력층으로 사용해야 했음.



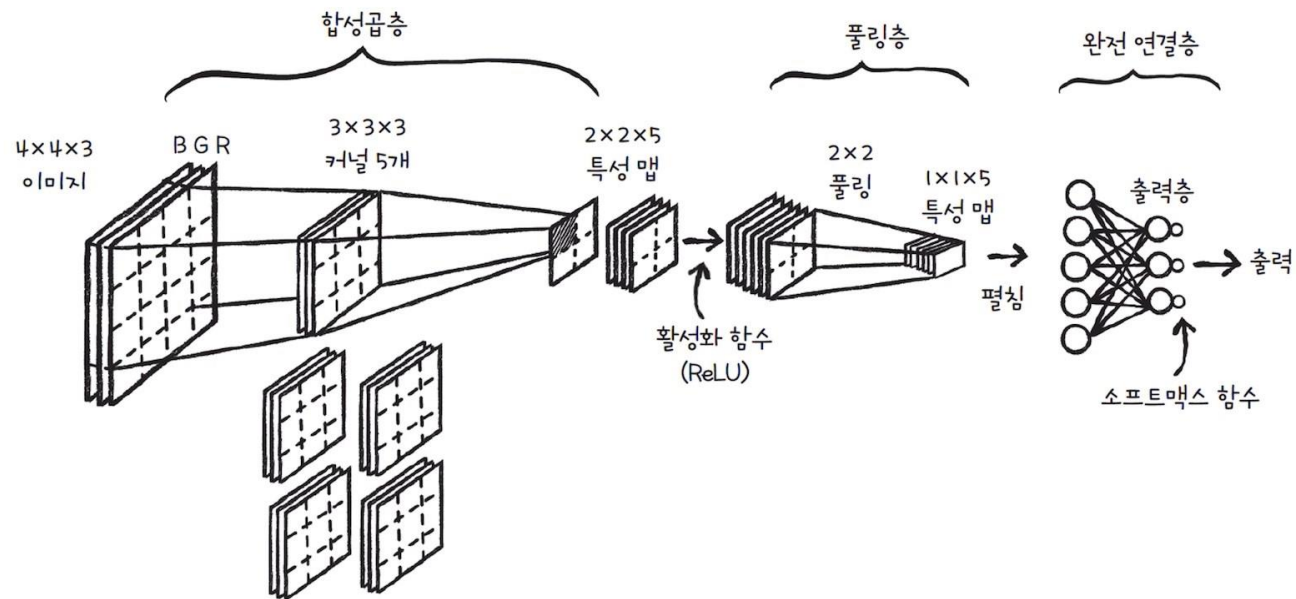
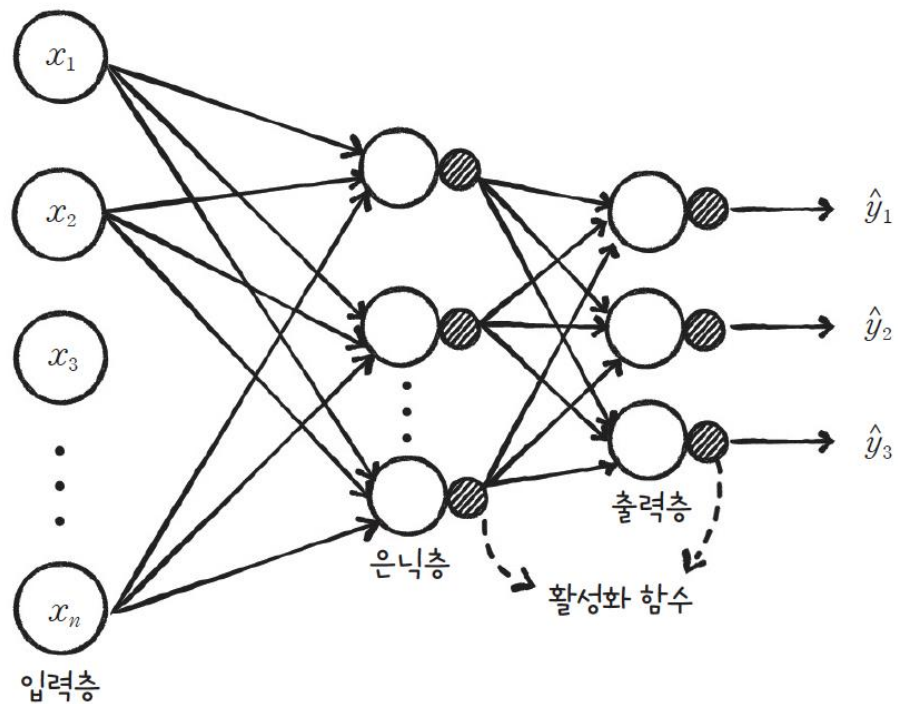
↓ 변환



## 합성곱 신경망

반면 합성곱 신경망은 이미지의 공간적인 구조 정보를 보존하면서 학습할 수 있음.

# 다층 신경망 구조 & 합성곱 신경망 구조



# 합성곱 연산

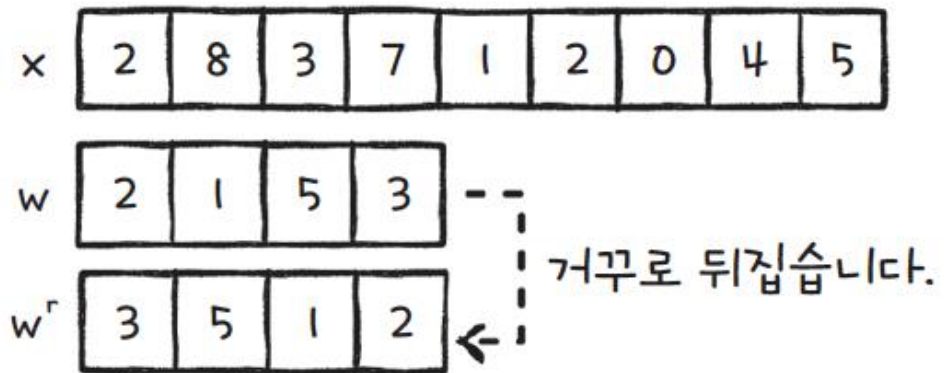
합성곱(convolution):

하나의 함수와 또 다른 함수를 반전 이동한 값을 곱한 후, 구간에 대해 적분하는 수학 연산자.  
 $f^*g$ 로 표시.

수식으로 나타내면:  $(f * g)(t) = \int_{-\infty}^{\infty} f(\tau)g(t - \tau) d\tau$

이산 함수의 경우:  $(f * g)(m) = \sum_n f(n)g(m - n)$

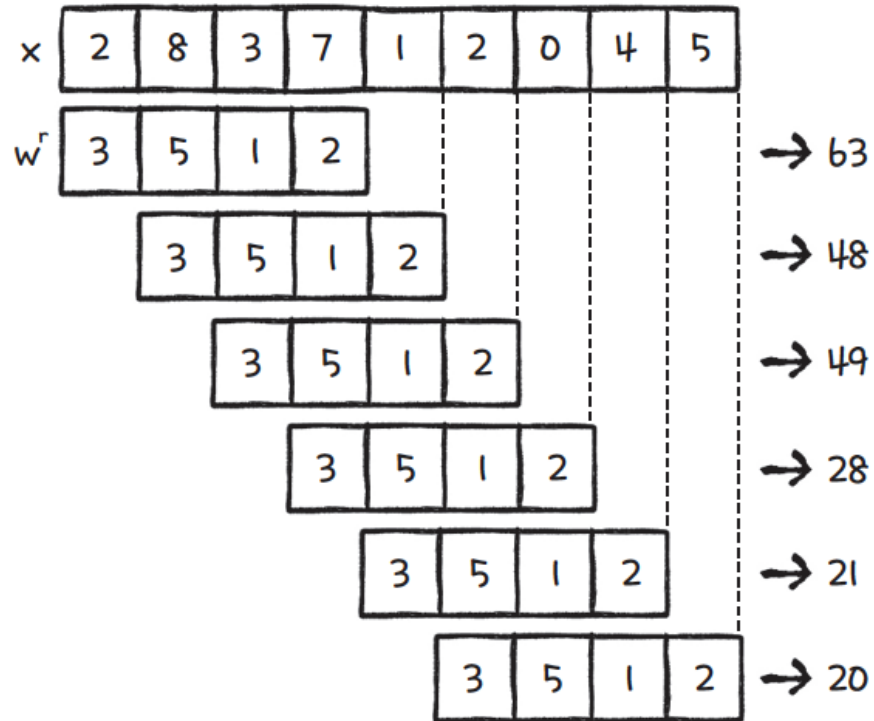
# 합성곱 연산을 그림으로 이해



1차배열 x와 w

W<sup>r</sup>: 배열 w의 원소 순서를 뒤집은 것.

# 합성곱 연산을 그림으로 이해

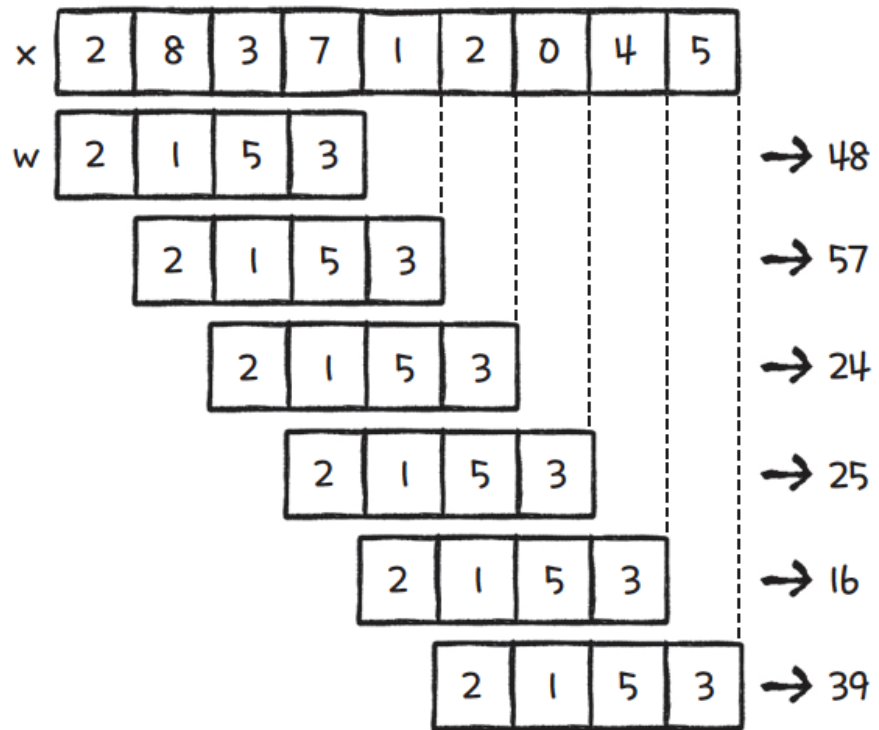


뒤집어진 배열  $w$ 과  $x$ 배열의 각 원소끼리 곱해진 후에 더해짐.

$w$ 이  $x$ 배열 끝에 도달할때 까지 수행 됨.



# 교차 상관 연산



합성곱 신경망에서는 대부분 신경망을 만들때 교차 상관을 사용함.

합성곱을 적용하든 뒤집지않고 교차 상관을 적용하든 상관없음.

# 합성곱 구현

```
[1] import numpy as np
    w = np.array([2,1,5,3])
    x = np.array([2,8,3,7,1,2,0,4,5])
```

```
[2] w_r=np.flip(w)
    print(w_r)
```

[3 5 1 2]

```
▶ w_r2=w[::-1] #전체 데이터를 역순으로 출력해줌.
  print(w_r)
```

[3 5 1 2]

[1]:  
넘파이 배열로 w, x 정의

[2]:  
넘파이 flip 함수로 배열 뒤집기

[3]:  
파이썬 슬라이스 연산자로도 뒤집기 가능

# 싸이파이로 합성곱, 교차 상관 수행

```
[10] from scipy.signal import convolve  
convolve(x, w, mode='valid')
```

```
array([63, 48, 49, 28, 21, 20])
```

싸이파이의 합성곱 함수 convolve 수행

```
from scipy.signal import correlate  
correlate(x, w, mode='valid')
```

```
array([48, 57, 24, 25, 16, 39])
```

싸이파이의 교차 상관 함수 correlate 수행

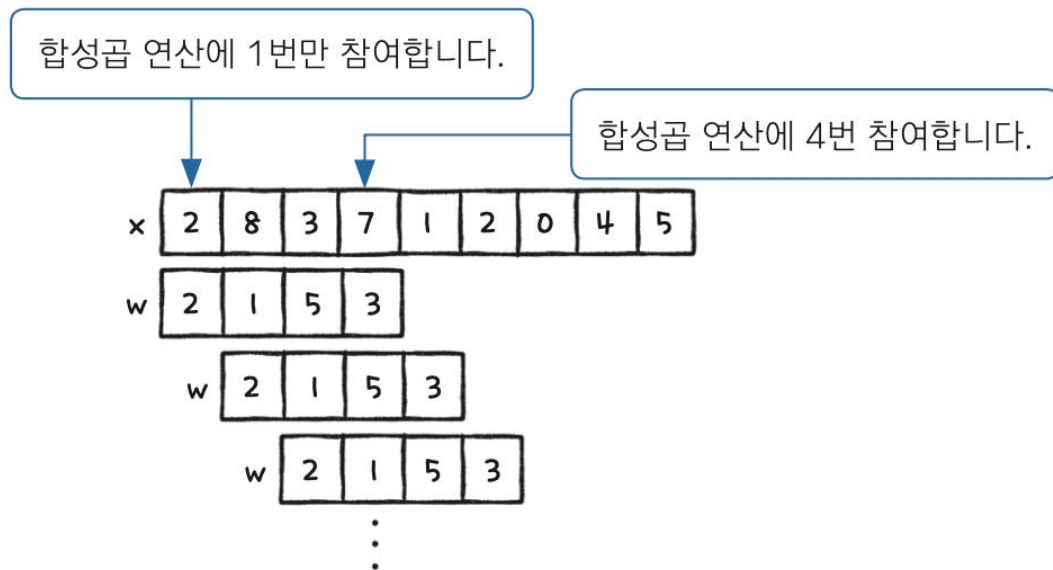
# 패딩과 스트라이드

패딩(padding)은 원본 배열 양 끝에 빈 원소를 추가하는 것.

스트라이드(stride)는 미끄러지는 배열의 간격을 조절하는 것.

# 밸리드 패딩

밸리드 패딩(valid padding):

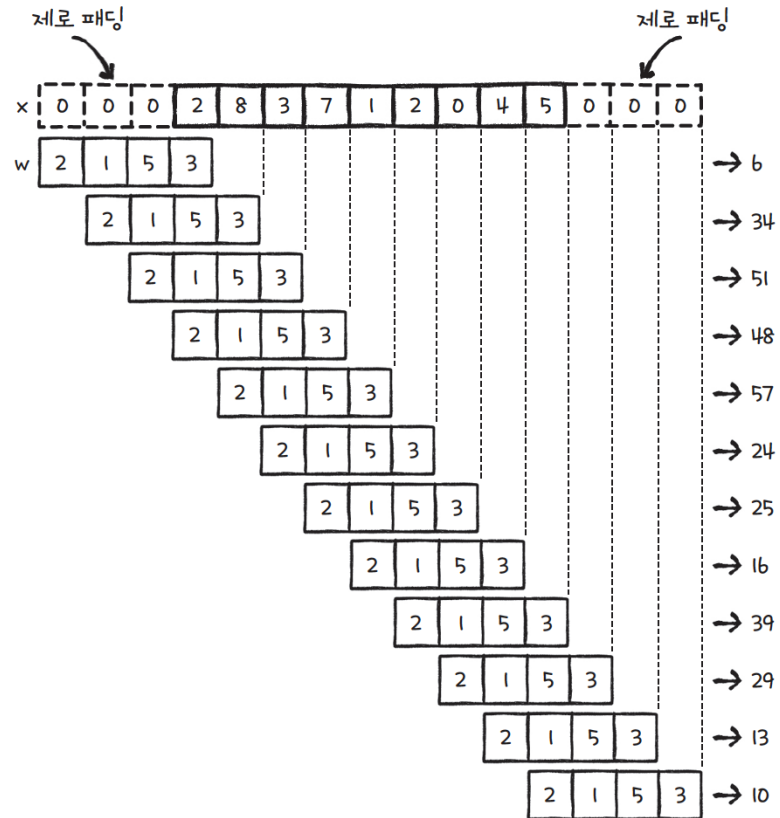


원본 배열에 패딩을 추가하지 않고  
미끄러지는 배열이 원본 배열의 끝으로 갈 때까지  
교차 상관 수행.

원본 배열의 각 원소가 연산에 참여하는 정도가 다름  
(양 끝 원소 연산 참여도가 낮음).

결과로 얻는 배열의 크기는 원본 배열보다 작게 됨.

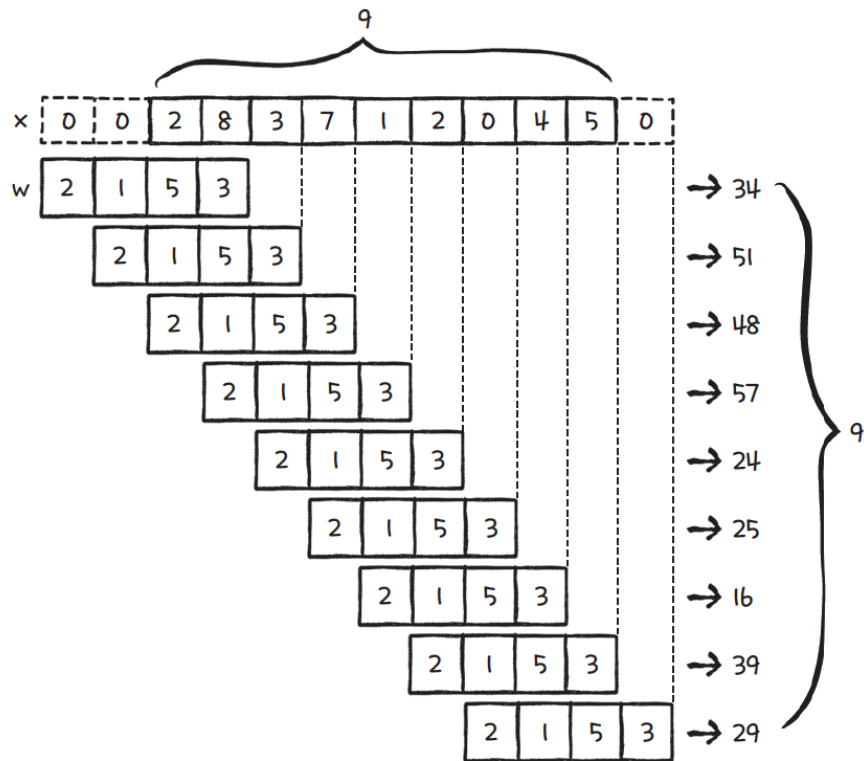
# 풀패딩



풀 패딩은 원본 배열 원소의 연산 참여도를 동일하게 만듦.

이를 위해 원본 배열 양 끝에 가상의 원소 0(제로 패딩)을 추가함.

# 세임 패딩



세임 패딩은 출력 배열의 길이를 원본 배열의 길이와 동일하게 만듭니다.

이를 위해 출력 배열의 길이가 원본 배열의 길이와 같아지도록 원본 배열에 제로 패딩을 추가.

# 밸리드 패딩, 풀패딩, 세임 패딩 코드 구현

```
[12] correlate(x, w, mode='valid')  
array([48, 57, 24, 25, 16, 39])
```

밸리드 패딩 : mode='valid'

```
correlate(x, w, mode='full') #원본 배열 원소의 참여도 동일해짐.  
array([ 6, 34, 51, 48, 57, 24, 25, 16, 39, 29, 13, 10])
```

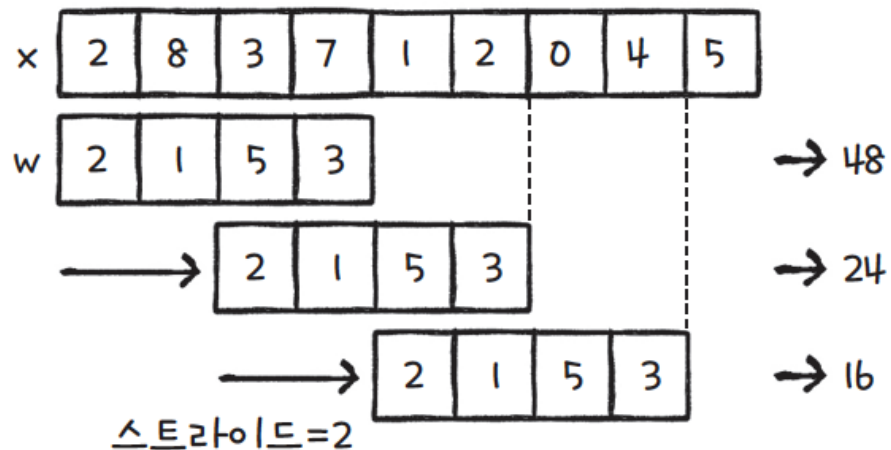
풀 패딩 : mode='full'

```
[14] correlate(x, w, mode='same') #출력 배열길이가 입력 배열 길이와 9로 같음.  
array([34, 51, 48, 57, 24, 25, 16, 39, 29])
```

세임 패딩 : mode='same'



# 스트라이드

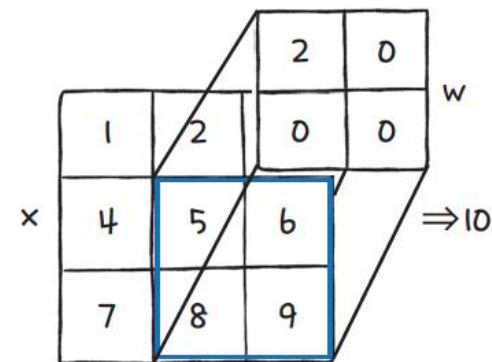
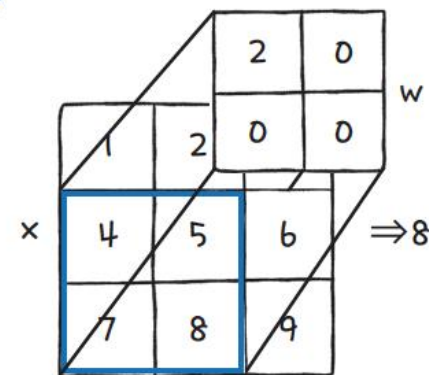
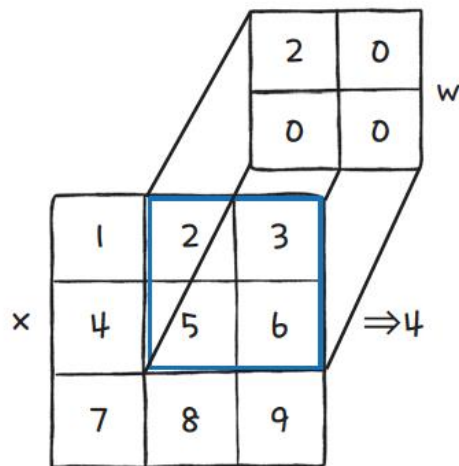
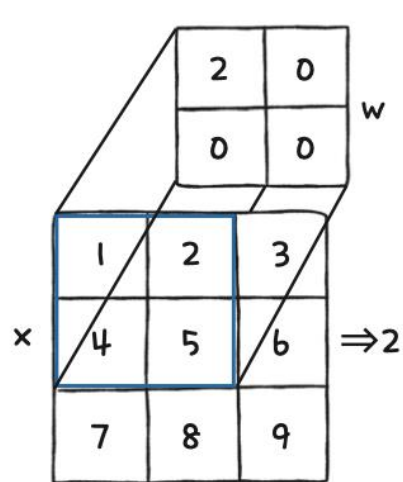


미끄러지는 간격을 조정.

스트라이드를 2로 지정하면 2칸씩 미끄러지며 연산을 수행.

합성곱 신경망에서는 보통 스트라이드를 1로 지정.

# 2차원 배열의 합성곱 수행



2차원 원본 배열  $x$ 와 미끄러지는 배열  $w$

원본 배열 왼쪽 모서리 끝에  $w$  배열을 맞추고 합성곱 수행.

오른쪽으로  $w$ 배열을 1칸 옮기고 끝에 도달하면 아래로 1칸 내려서 다시

왼쪽 끝부터 합성곱 수행.

## 2차원 배열의 세임 패딩

1	2	3	0
4	5	6	0
7	8	9	0
0	0	0	0

1	2	3	0
4	5	6	0
7	8	9	0
0	0	0	0

1	2	3	0
4	5	6	0
7	8	9	0
0	0	0	0

1	2	3	0
4	5	6	0
7	8	9	0
0	0	0	0

1	2	3	0
4	5	6	0
7	8	9	0
0	0	0	0

1	2	3	0
4	5	6	0
7	8	9	0
0	0	0	0

1	2	3	0
4	5	6	0
7	8	9	0
0	0	0	0

1	2	3	0
4	5	6	0
7	8	9	0
0	0	0	0

1	2	3	0
4	5	6	0
7	8	9	0
0	0	0	0

오른쪽과 아래쪽 모서리에 제로 패딩 추가됨

## 2차원 배열에서 슬라이드

1	2	3	0
4	5	6	0
7	8	9	0
0	0	0	0

1	2	3	0
4	5	6	0
7	8	9	0
0	0	0	0

1	2	3	0
4	5	6	0
7	8	9	0
0	0	0	0

1	2	3	0
4	5	6	0
7	8	9	0
0	0	0	0

미끄러지는 방향은 유지, 미끄러지는 간격의 크기만 커짐  
위에서는 슬라이드를 2로 지정

## 2차원 배열의 합성곱 구현

```
[15] x = np.array([[1,2,3],
                  [4,5,6],
                  [7,8,9]])
w = np.array([[2,0],[0,0]])
from scipy.signal import correlate2d
correlate2d(x,w,mode='valid')

array([[ 2,  4],
       [ 8, 10]])
```

correlate2d() 함수를 이용, 2차원 배열 합성곱 계산

 correlate2d(x,w,mode='same')

```
array([[ 2,  4,  6],
       [ 8, 10, 12],
       [14, 16, 18]])
```

세임 패딩 계산.

# 텐서플로로 합성곱 수행

지금까지 싸이파이로 합성곱을 구현

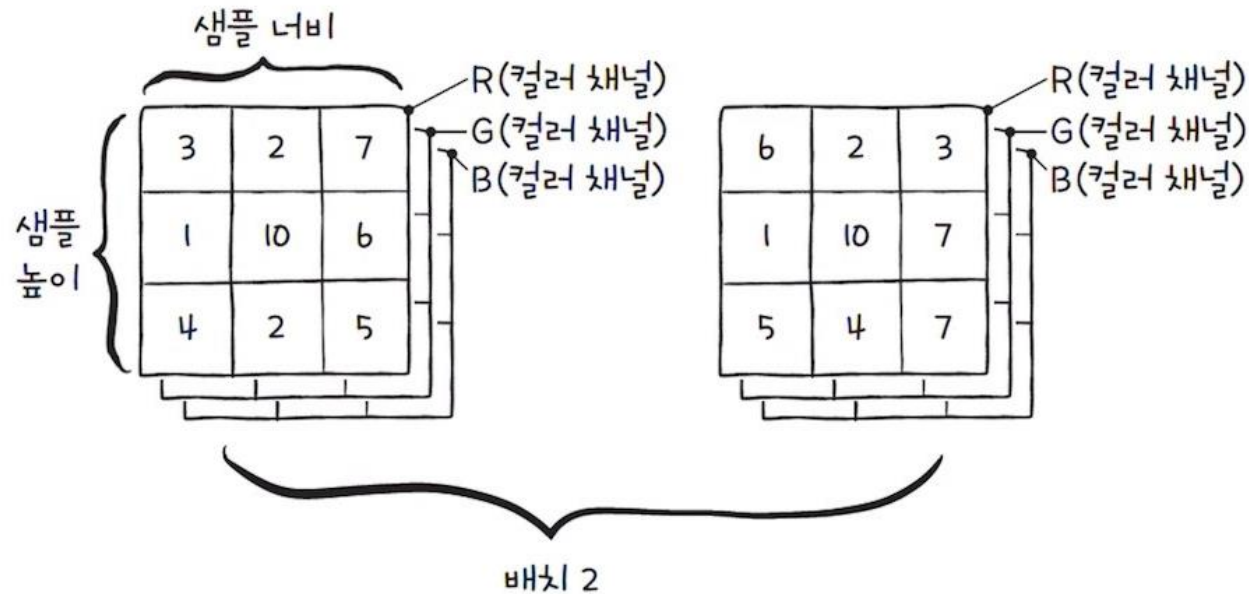
텐서플로에서도 합성곱 함수가 제공됨

이제 앞에서의 원본배열을 입력, 미끄러지는 배열을 가중치 라고 부를 것

# 합성곱 신경망의 입력

텐서플로의 2차원 합성곱 수행 함수는 `conv2d()`.  
이 함수는 입력으로 4차원 배열을 받음.

# 합성곱 신경망의 입력



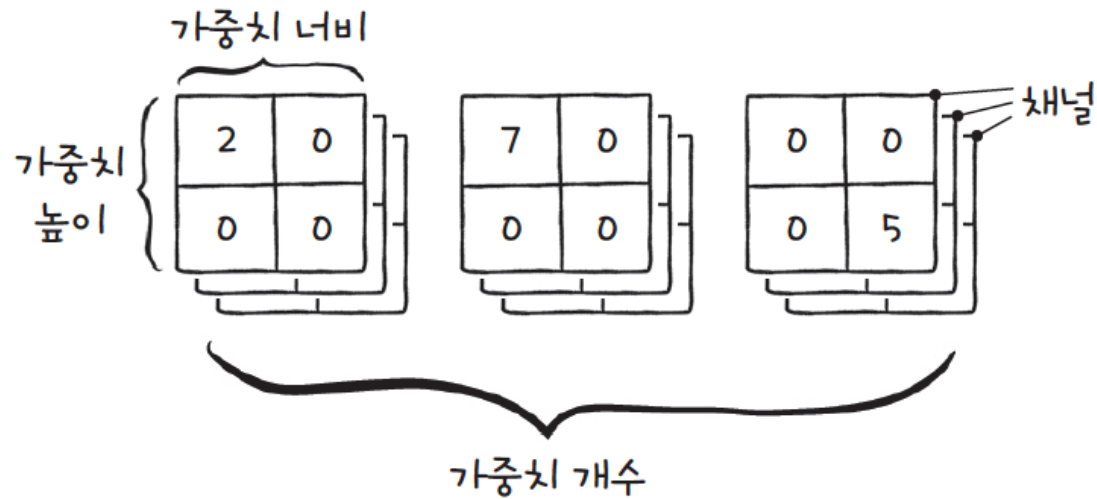
위는 입력 4차원 배열의 모습.

입력을 4차원 배열로 표현하면 (2,3,3,3)

(배치, 샘플 높이, 샘플 너비, 채널)



# 합성곱 신경망의 가중치



가중치도 4차원.

( 2 , 2 , 3 , 3 )

( 가중치 높이, 너비, 채널, 가중치 개수 )

# 합성곱 수행을 위해 배열 차원 변경

```
[17] import tensorflow as tf
      x_4d = x.astype(np.float).reshape(1,3,3,1)
      w_4d = w.reshape(2,2,1,1)
```

x, w를 넘파이 reshape() 메서드로 2차원 배열에서 4차원 배열로 바꿈.

텐서플로는 실수형 입력을 기대하므로 넘파이 astype() 메서드로 입력의 자료형을 실수로 바꿔줌.

# 합성곱 수행

```
[19] c_out = tf.nn.conv2d(x_4d, w_4d, strides=1, padding='SAME')
```

```
[20] c_out.numpy().reshape(3,3)
```

```
array([[ 2.,  4.,  6.],  
       [ 8., 10., 12.],  
       [14., 16., 18.]])
```

[19]

스트라이드 1, 패딩은 세임 패딩 적용

conv2d() 함수는 결과값으로 Tensor 객체를 반환. 텐서플로에서는 다차원 배열을 텐서라고 부름.

Tensor 객체의 numpy() 메서드를 사용시, 텐서를 넘파이 배열로 변환 가능

[20]

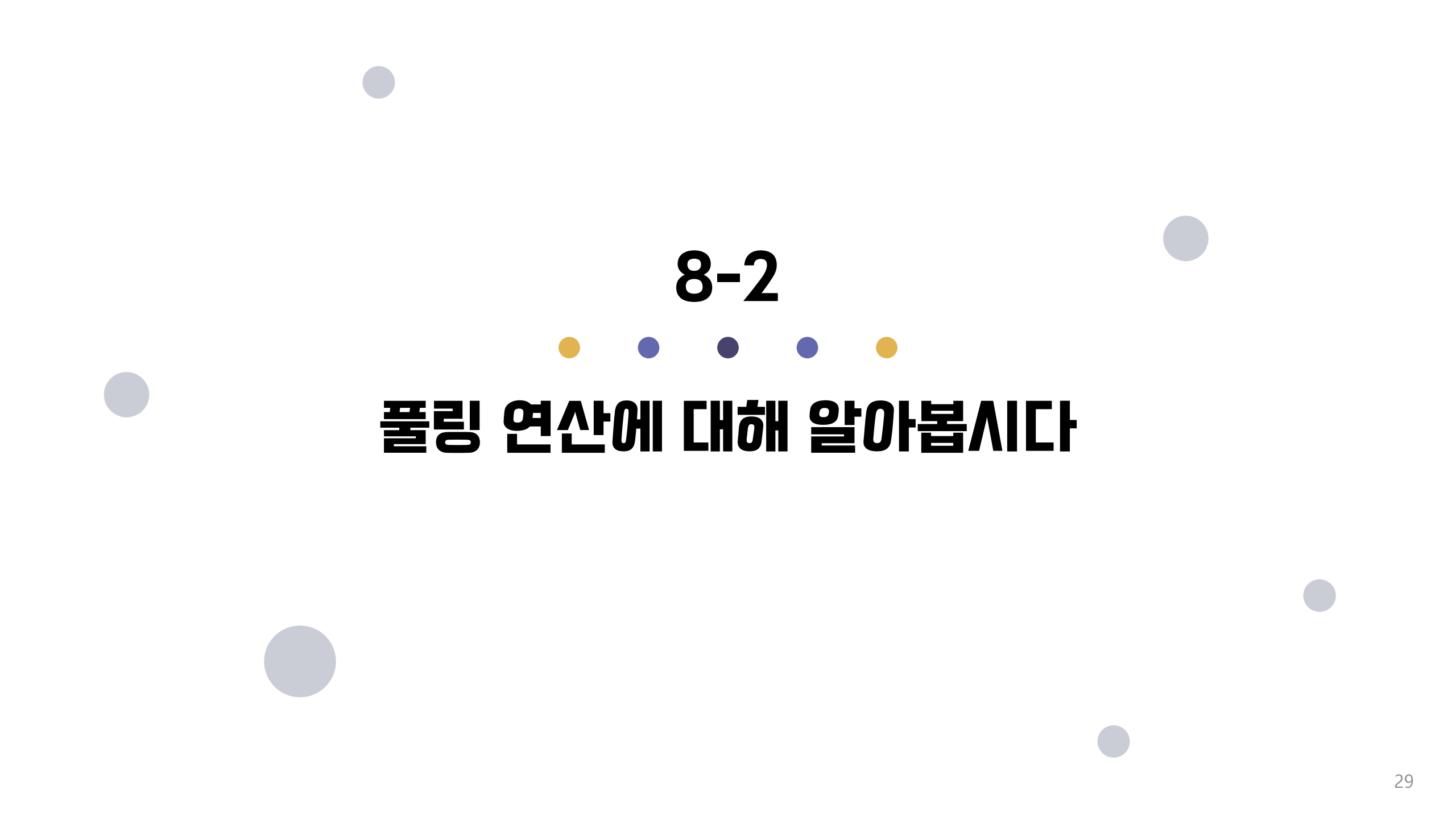
배치와 컬러 차원 제거하고 (3,3)크기로 변환하여 출력

# 합성곱의 가중치

합성곱의 가중치를 필터 또는 커널이라고 부름.

이 책에서는 합성곱 필터 1개 지칭은 ‘커널’

필터 전체를 지칭할때는 일반 신경망과 동일하게 ‘가중치’ 라고 지칭.



8-2

**폴링 연산에 대해 알아보시다**

# 풀링 연산

## 합성곱층:

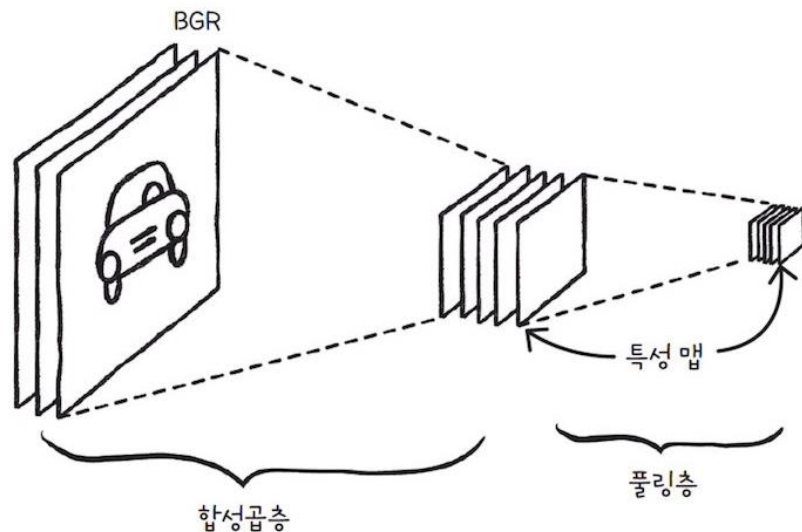
합성곱 신경망에서 합성곱이 일어나는 층

## 풀링층:

풀링이 일어나는 층

## 특성 맵(feature map):

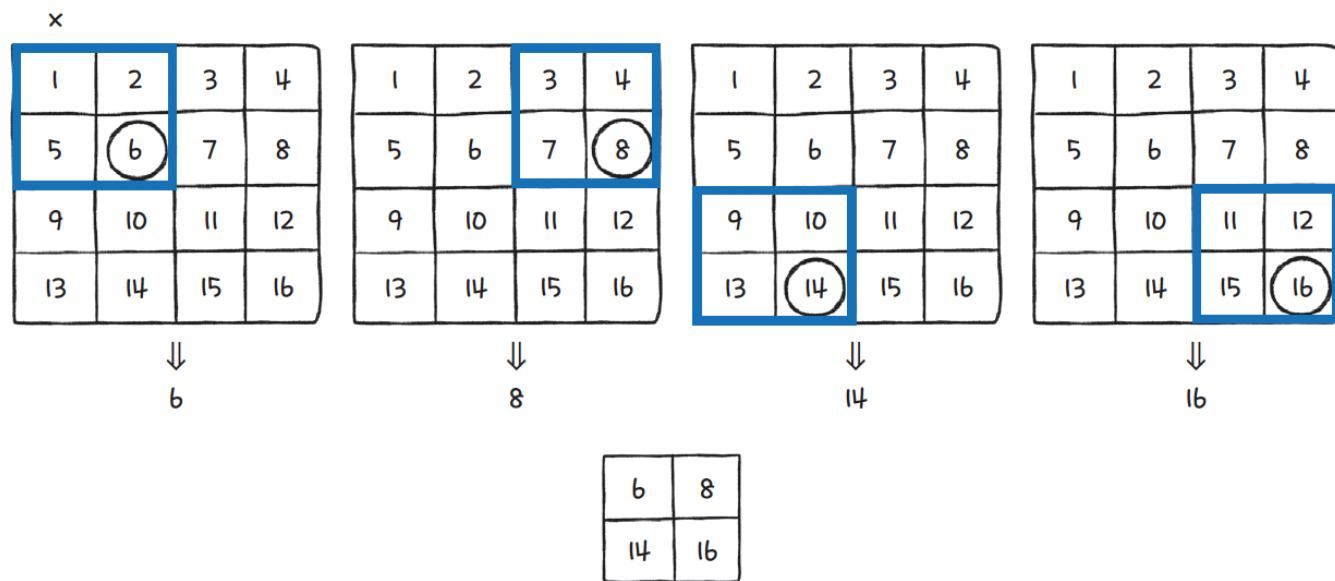
합성곱층과 풀링층에서 만들어진 결과  
입력이 합성곱층을 통과할 때 합성곱과 활성화 함수가 적용되어 특성 맵이 만들어짐.  
그 다음 특성 맵이 풀링층을 통과하여 또 다른 특성 맵이 만들어짐.



# 풀링 연산

풀링:

특성 맵을 스캔하여 최댓값을 고르거나 평균값을 계산하는 것.



위는 최대 풀링 그림(특성 맵을 스캔하며 최댓값을 고름)

# 최대 풀링과 평균 풀링

풀링 영역의 크기는 보통 2x2를 지정.

일반적으로 스트라이드는 풀링의 한 모서리 크기로 지정 -(풀링 영역이 겹쳐지지 않도록)

2x2풀링은 특성 맵의 크기를 절반으로 줄임 (면적은 1/4)

=>특성 맵의 한 요소가 입력의 더 넓은 영역을 대표하는 효과를 나타냄.

최대 풀링 : 특성 맵 위를 스캔하며 최대값을 고름.

평균 풀링 : 풀링 영역의 평균값을 계산

=>최대 풀링은 가장 큰 특징을 유지시키는 성질이 있으므로 이미지 분류작업에 잘 맞음.

(평균 풀링은 특징들을 희석시킬 가능성이 높음)



# 최대 풀링과 평균 풀링 수행

```
[23] x = np.array([[1,2,3,4],  
                  [5,6,7,8],  
                  [9,10,11,12],  
                  [13,14,15,16]])  
x = x.reshape(1,4,4,1)
```

```
[24] p_out = tf.nn.max_pool2d(x, ksize=2, strides=2, padding="VALID")  
p_out.numpy().reshape(2,2)  
  
array([[ 6.,  8.],  
       [14., 16.]], dtype=float32)
```

[23]

1~16 값이 들어간 4x4크기 배열을 만들어 1x4x4x1 크기 배열로 변형한 것.

[24]

max\_pool2d()함수로 최대 풀링을 수행.

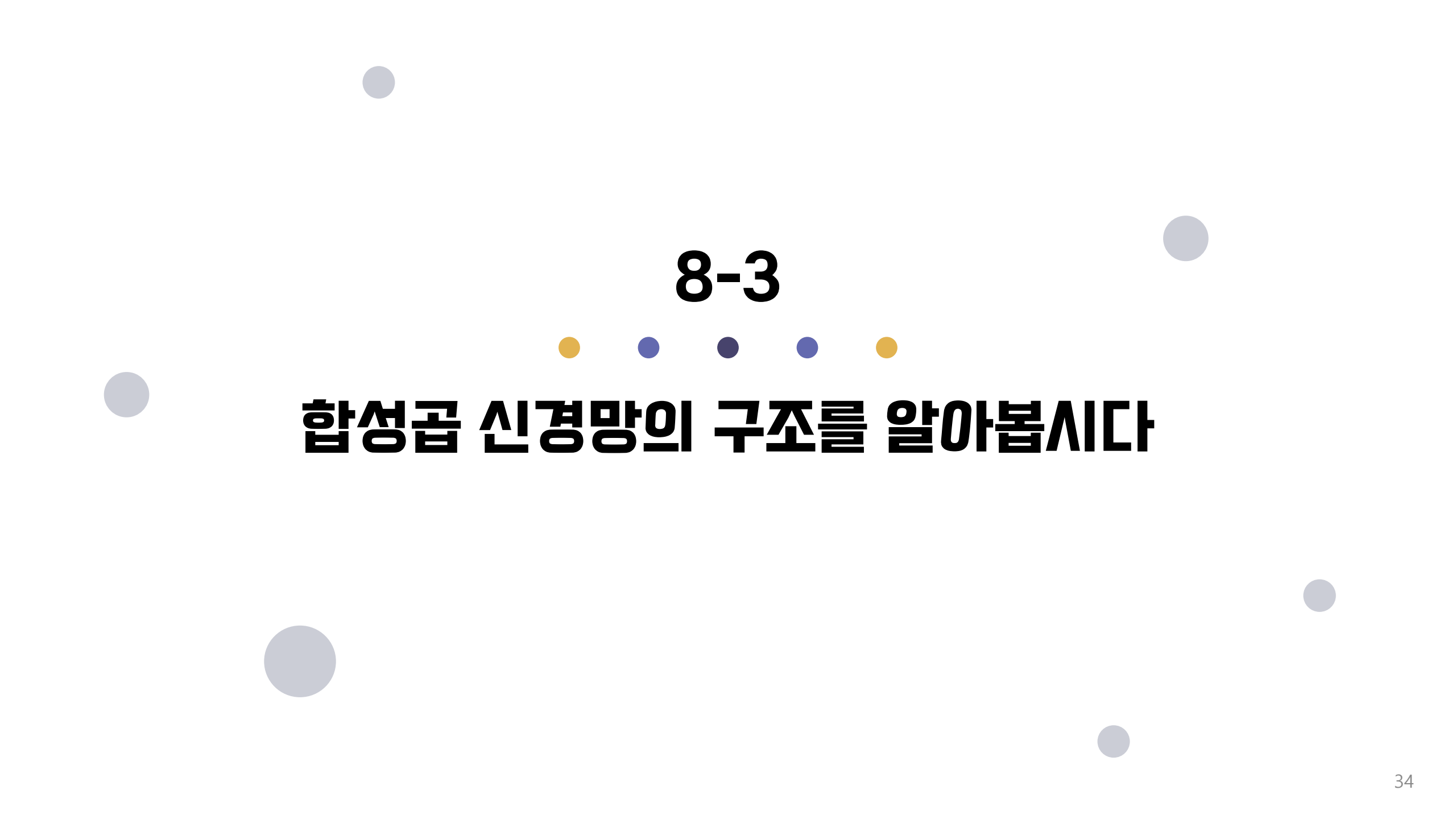
ksize 매개변수에 풀링의 크기

strides 매개변수에 스트라이드 크기

max\_pool2d()함수가 반환한 텐서 객체를 numpy()메서드로 변환=> 2x2크기 2차원 배열로 변형

=> 최대 풀링 수행

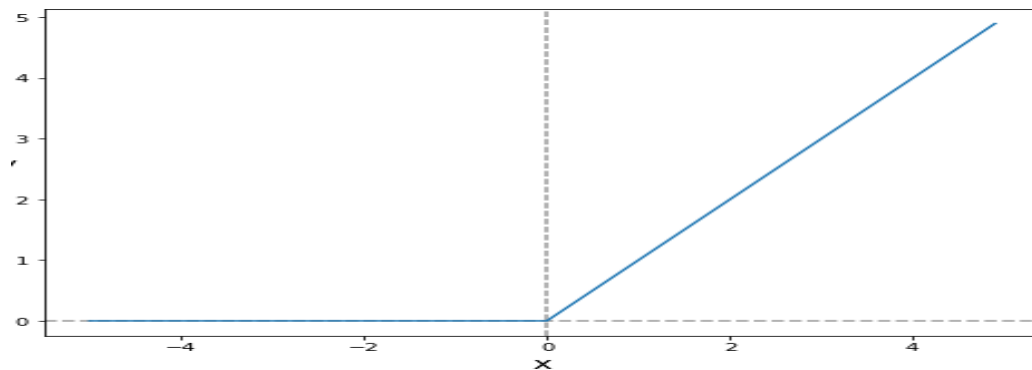
추가적으로 풀링층에는 학습되는 가중치가 없고,  
풀링층 통과 전후로 배치, 채널 크기 동일함.



8-3

# 합성곱 신경망의 구조를 알아봅시다

# 합성곱 신경망의 구조



이전까지 은닉층에 시그모이드 함수를 활성화 함수로 사용.

출력층에는 이진 분류시: 시그모이드 함수 사용  
다중 분류시: 소프트맥스 함수 사용

ReLU 함수: 주로 합성곱 층에 적용되는 활성화 함수.(합성곱 신경망의 성능을 높여줌).  
0보다 작은 값은 0으로 만들고, 0보다 큰 값은 그대로 통과 시킴.

# 렐루 함수 구현

```
[25] def relu(x):  
      return np.maximum(x,0)
```

```
[26] x=np.array([-1,2,-3,4,-5])  
      relu(x)  
  
      array([0, 2, 0, 4, 0])
```

```
▶ r_out=tf.nn.relu(x)  
   r_out.numpy()  
  
   array([0, 2, 0, 4, 0])
```

[25]

넘파이의 맥시멈 함수를 통해 렐루 함수를 간단히 구현 가능  
(두 매개변수 값 중 큰 값을 반환 하는 함수)

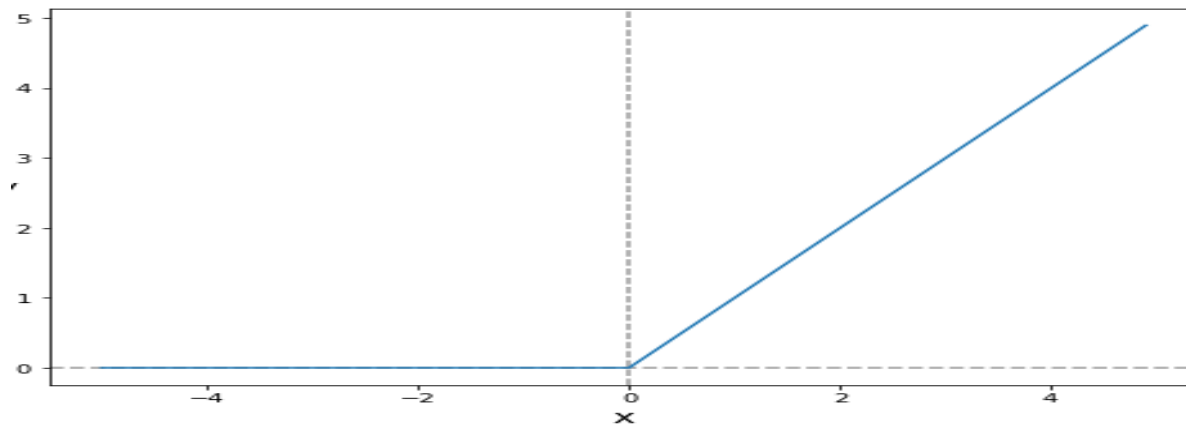
[26]

relu 함수의 반환값을 보면 음수는 0이 되고  
양수만 그대로 반환되는 것을 확인 가능

[27]

텐서플로에서 제공하는 렐루 함수는 relu()임  
렐루 함수는 Tensor 객체로 반환하므로,  
출력할 때는 넘파이로 반환해줘야 함.

# 렐루 함수의 도함수



렐루 함수의 도함수

입력  $x$ 값이 0보다 크면 1이고 0보다 작으면 0임

$x=0$ 일때의 도함수는 없는것이 맞지만 대부분 딥러닝 패키지들은

$x=0$ 일때의 도함수를 0으로 생각해줌

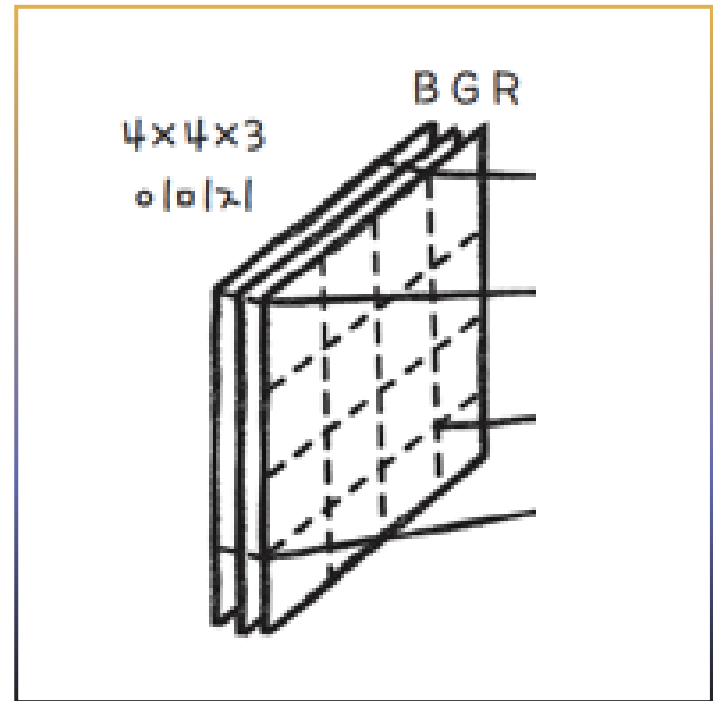
# 합성곱 신경망의 입력 데이터

앞장의 다층 신경망과 달리

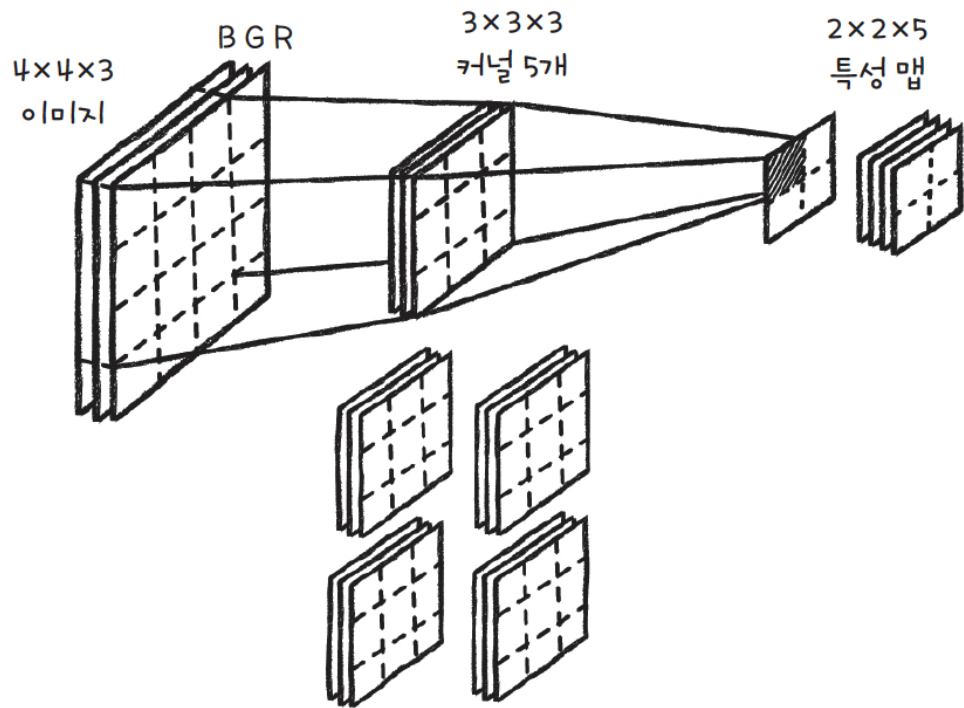
합성곱 신경망은 이미지의 2차원 형태를 그대로 입력으로 사용  
(1차원 배열로 펼치지 않아 이미지 정보가 손상되지 않음)

이미지는 채널이라는 차원을 하나 더 가지고 있음

색상을 표현하기 위한 정보 : **RGB** (빨강, 초록, 파랑의 조합)



# 합성곱층에서 일어나는 일

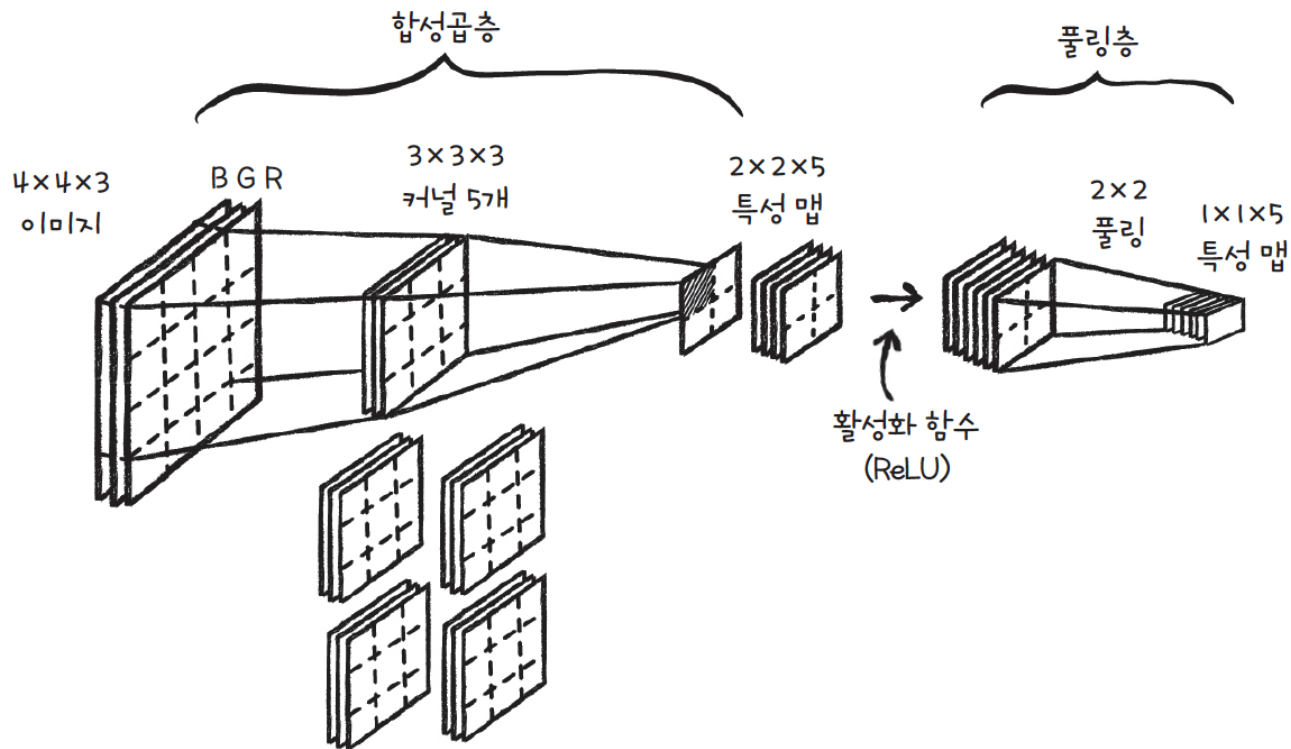


이미지의 모든 채널에서 합성곱이 한 번에 적용되어야 하므로  
커널의 마지막 차원과 입력 채널의 개수가 같아야 함

4x4x3 이미지 위를 3x3x3 커널이 이동하며 합성곱을 4번 수행  
=> 2x2 크기 커널 생성

커널이 5개이므로 특성맵 5개가 생성됨

# 풀링층에서 일어나는 일



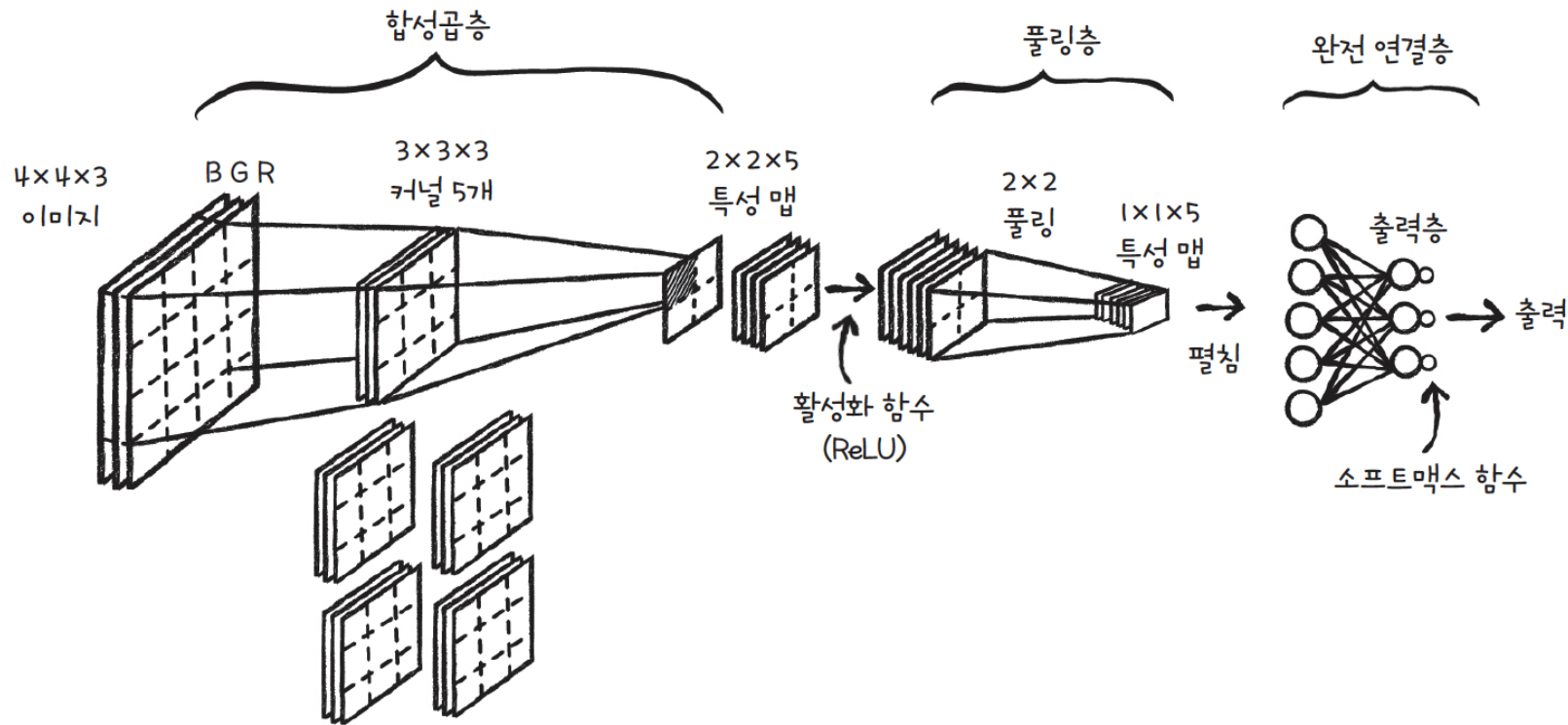
합성곱층을 통해 특성 맵이 만들어졌음

이 특성 맵에 활성화 함수로  
렐루 함수를 적용, 풀링 적용.

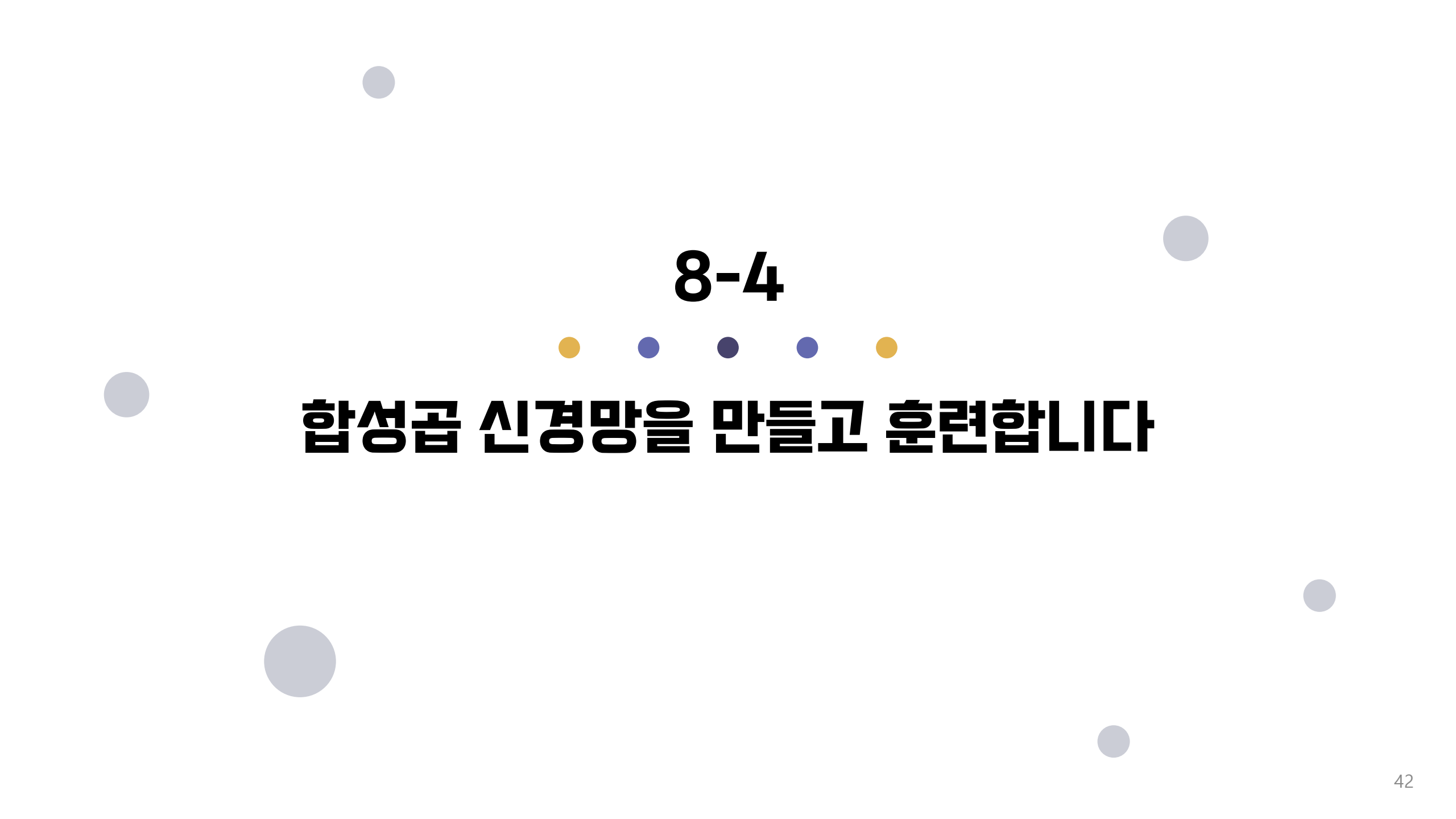
풀링은 특성 맵의 크기를  
 $2 \times 2 \times 5$ 에서  $1 \times 1 \times 5$  크기로 줄여 줌



# 특성 맵을 펼쳐 완전 연결 신경망에 주입



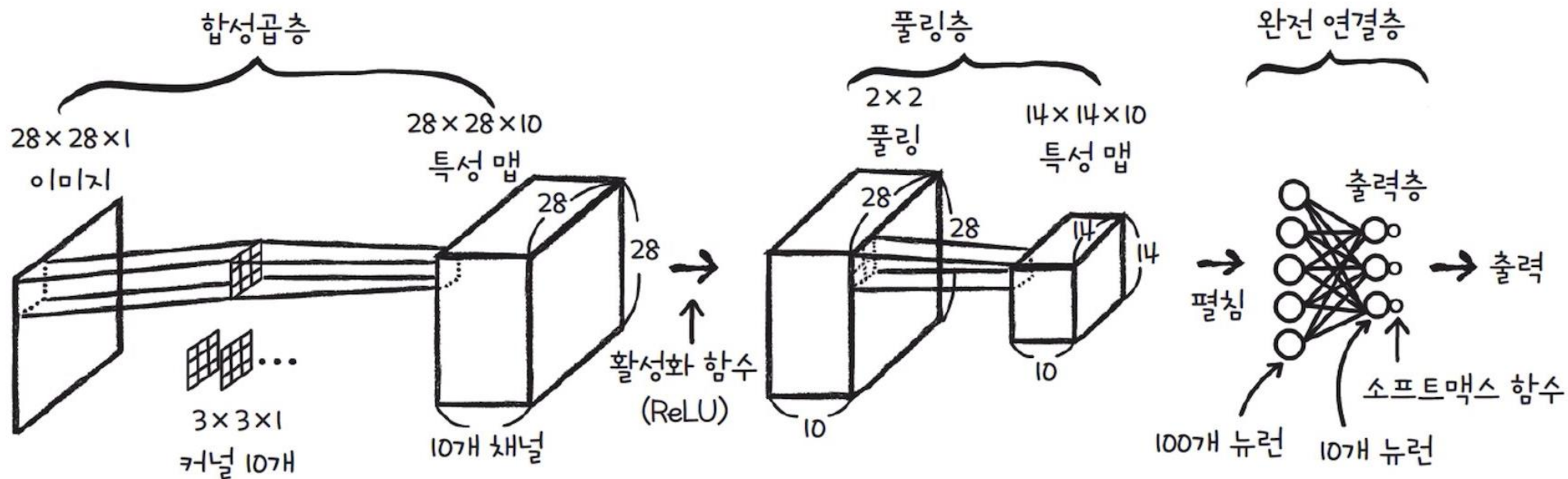
합성곱층과 풀링층을 통과시켜 얻은 특성 맵은  
일렬로 펼쳐 완전 연결 연결층에 입력으로 주입



8-4

**합성곱 신경망을 만들고 훈련합니다**

# 합성곱 신경망 구조



# ConvolutionNetwork

```
def __init__(self, n_kernels=10, units=10, batch_size=32, learning_rate=0.1):  
    self.n_kernels = n_kernels # 합성곱의 커널 개수  
    self.kernel_size = 3      # 커널 크기  
    self.optimizer = None     # 옵티마이저  
    self.conv_w = None        # 합성곱 층의 가중치  
    self.conv_b = None        # 합성곱 층의 절편  
    self.units = units        # 은닉층의 뉴런 개수  
    self.batch_size = batch_size # 배치 크기  
    self.w1 = None            # 은닉층의 가중치  
    self.b1 = None            # 은닉층의 절편  
    self.w2 = None            # 출력층의 가중치  
    self.b2 = None            # 출력층의 절편  
    self.a1 = None            # 은닉층의 활성화 출력  
    self.losses = []          # 훈련 손실  
    self.val_losses = []      # 검증 손실  
    self.lr = learning_rate   # 학습률
```

새로 추가된 변수

# 정방향 계산

```
def forpass(self, x):  
    # 3x3 합성곱 연산을 수행합니다.  
    c_out = tf.nn.conv2d(x, self.conv_w, strides=1, padding='SAME') + self.conv_b  
    # 렐루 활성화 함수를 적용합니다.  
    r_out = tf.nn.relu(c_out)  
    # 2x2 최대 풀링을 적용합니다.  
    p_out = tf.nn.max_pool2d(r_out, ksize=2, strides=2, padding='VALID')  
    # 첫 번째 배치 차원을 제외하고 출력을 일렬로 펼칩니다.  
    f_out = tf.reshape(p_out, [x.shape[0], -1])  
    z1 = tf.matmul(f_out, self.w1) + self.b1  
    a1 = tf.nn.relu(z1)  
    z2 = tf.matmul(a1, self.w2) + self.b2  
    return z2
```

# 첫 번째 층의 선형 식을 계산합니다  
# 활성화 함수를 적용합니다  
# 두 번째 층의 선형 식을 계산합니다.

# 정방향 계산

7장

```
def forpass(self, x):  
    z1 = np.dot(x, self.w1) + self.b1      # 첫 번째 층의 선형 식을 계산합니다  
    self.a1 = self.sigmoid(z1)            # 활성화 함수를 적용합니다  
    z2 = np.dot(self.a1, self.w2) + self.b2 # 두 번째 층의 선형 식을 계산합니다.  
    return z2
```

```
# 첫 번째 배치 차원을 제외하고 출력을 일렬로 펼칩니다.  
f_out = tf.reshape(p_out, [x.shape[0], -1])  
z1 = tf.matmul(f_out, self.w1) + self.b1      # 첫 번째 층의 선형 식을 계산합니다  
a1 = tf.nn.relu(z1)                          # 활성화 함수를 적용합니다  
z2 = tf.matmul(a1, self.w2) + self.b2        # 두 번째 층의 선형 식을 계산합니다.  
return z2
```

8장

# 역방향 계산

## 7장

```
def backprop(self, x, err):
    m = len(x)          # 샘플 개수
    # 출력층의 가중치와 절편에 대한 그래디언트를 계산합니다.
    w2_grad = np.dot(self.a1.T, err) / m
    b2_grad = np.sum(err) / m
    # 시그모이드 함수까지 그래디언트를 계산합니다.
    err_to_hidden = np.dot(err, self.w2.T) * self.a1 * (1 - self.a1)
    # 은닉층의 가중치와 절편에 대한 그래디언트를 계산합니다.
    w1_grad = np.dot(x.T, err_to_hidden) / m
    b1_grad = np.sum(err_to_hidden, axis=0) / m
    return w1_grad, b1_grad, w2_grad, b2_grad
```

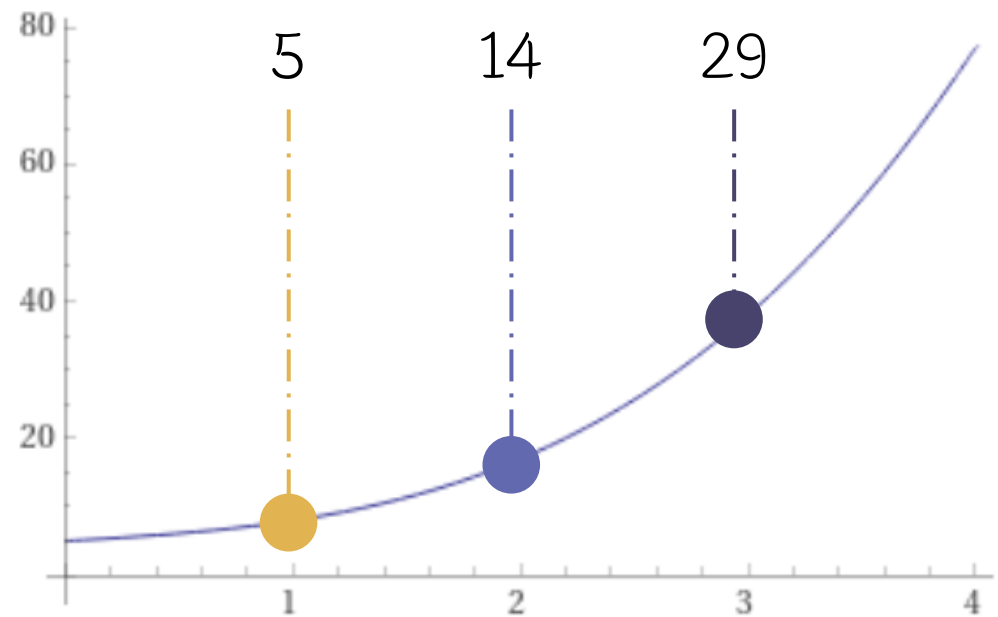
# Automatic Differentiation

```
x = tf.Variable(np.array([1.0, 2.0, 3.0]))  
with tf.GradientTape() as tape:  
    y = x ** 3 + 2 * x + 5  
  
# 그래디언트를 계산합니다.  
print(tape.gradient(y, x))  
  
tf.Tensor([ 5. 14. 29.], shape=(3,), dtype=float64)
```

배열 크기  
3x1

데이터 타입  
64비트 실수

$$y' = 3x^2 + 2$$



$$y = x^3 + 2x + 5$$



# Training() 수정

```
def training(self, x, y):  
    m = len(x)                # 샘플 개수를 저장합니다.  
    with tf.GradientTape() as tape:  
        z = self.forpass(x)   # 정방향 계산을 수행합니다.  
        # 손실을 계산합니다.  
        loss = tf.nn.softmax_cross_entropy_with_logits(y, z)  
        loss = tf.reduce_mean(loss)  
  
    weights_list = [self.conv_w, self.conv_b,  
                    self.w1, self.b1, self.w2, self.b2]  
    # 가중치에 대한 그래디언트를 계산합니다.  
    grads = tape.gradient(loss, weights_list)  
    # 가중치를 업데이트합니다.  
    self.optimizer.apply_gradients(zip(grads, weights_list))
```

# Fit() 수정

```
def fit(self, x, y, epochs=100, x_val=None, y_val=None):
    self.init_weights(x.shape, y.shape[1])    # 은닉층과 출력층의 가중치를 초기화합니다.
    self.optimizer = tf.optimizers.SGD(learning_rate=self.lr)
    # epochs만큼 반복합니다.
    for i in range(epochs):
        print('에포크', i, end=' ')
        # 제너레이터 함수에서 반환한 미니배치를 순환합니다.
        batch_losses = []
        for x_batch, y_batch in self.gen_batch(x, y):
            print('.', end='')
            self.training(x_batch, y_batch)
            # 배치 손실을 기록합니다.
            batch_losses.append(self.get_loss(x_batch, y_batch))
        print()
        # 배치 손실 평균내어 훈련 손실 값으로 저장합니다.
        self.losses.append(np.mean(batch_losses))
        # 검증 세트에 대한 손실을 계산합니다.
        self.val_losses.append(self.get_loss(x_val, y_val))
```

# init\_weights() 수정

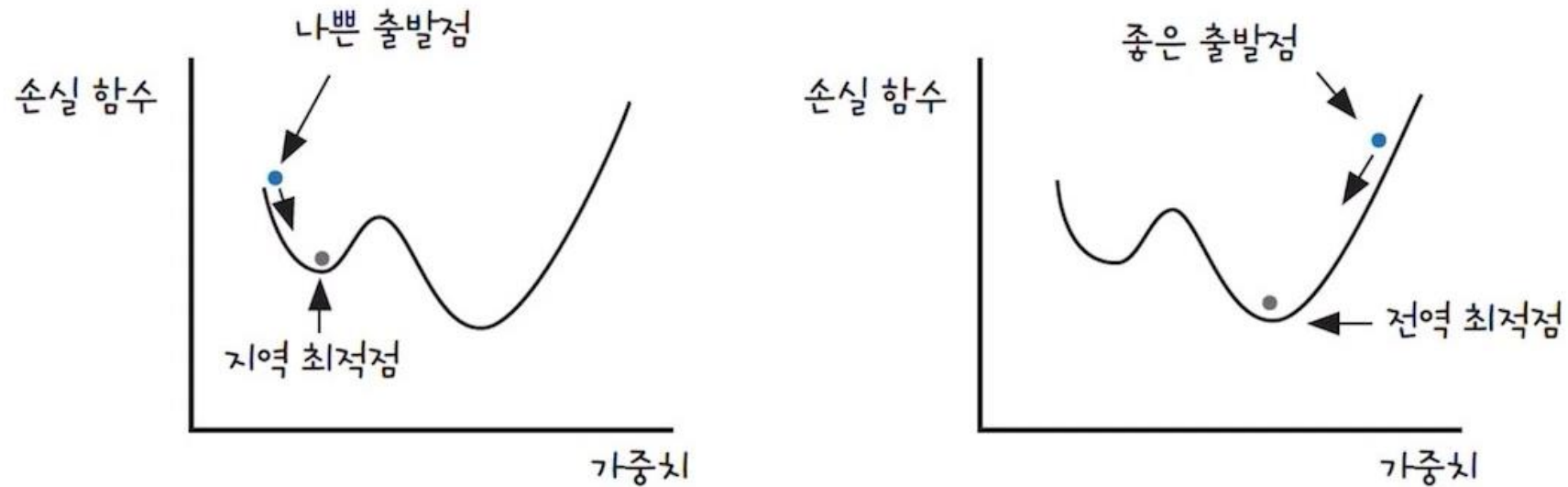
```
def init_weights(self, input_shape, n_classes):  
    g = tf.initializers.glorot_uniform()  
    self.conv_w = tf.Variable(g((3, 3, 1, self.n_kernels)))  
    self.conv_b = tf.Variable(np.zeros(self.n_kernels), dtype=float)  
    n_features = 14 * 14 * self.n_kernels  
    self.w1 = tf.Variable(g((n_features, self.units)))  
    self.b1 = tf.Variable(np.zeros(self.units), dtype=float)  
    self.w2 = tf.Variable(g((self.units, n_classes)))  
    self.b2 = tf.Variable(np.zeros(n_classes), dtype=float)
```

#Units=10

# (특성 개수, 은닉층의 크기)  
# 은닉층의 크기  
# (은닉층의 크기, 클래스 개수)  
# 클래스 개수

# 변수 초기화

## 경사하강법



출발점에 따라 결과가 달라질 수 있음

# 분산 조정 기반 초기화

Xavier Glorot Initialization

He Initialization

확률 분포를 기반으로 추출한 값으로 가중치를 초기화 하되,  
가중치 별로 이 확률 분포의 분산을 동적으로 조절해주는 방법

분산을 조정할 때에는 fan in, fan out 사용

# 분산 조정 기반 초기화

fan in : 해당 레이어에 들어오는 input tensor의 차원 크기

fan out : 해당 레이어가 출력하는 output tensor의 차원 크기

Fully Connected Layer

1000 x 100

fan in    fan out

# Xavier 초기화

fan in과 fan out을 모두 고려

Vanishing Gradient를 해결하기 위해 만들어짐

Sigmoid, Tanh 활성화 함수로 사용하는 신경망에서 많이 사용

$$\text{glorot unifom: } \text{unif}(-\text{limit}, +\text{limit}), \text{limit} = \sqrt{\frac{6}{\text{fan in} + \text{fan out}}}$$

$$\text{glorot normal: } \text{normal}(\text{mean}=0, \text{stddev}), \text{stddev} = \sqrt{\frac{2}{\text{fan in} + \text{fan out}}}$$

# He 초기화

Xaiver와 유사하지만 Neuron의 fan out을 고려 X

ReLU 활성화 함수로 사용하는 신경망에서 많이 사용

ReLU가 0 이하의 activation을 제거하기 때문에 fan in에 집중

$$he\ unifom: unif(-limit, +limit), limit = \sqrt{\frac{6}{fan\ in}}$$

$$he\ normal: normal(mean=0, stddev), stddev = \sqrt{\frac{2}{fan\ in}}$$



# Perdict() 수정

```
def predict(self, x):
    z = self.forpass(x)          # 정방향 계산을 수행합니다.
    return np.argmax(z, axis=1)  # 가장 큰 값의 인덱스를 반환합니다.

def score(self, x, y):
    # 예측과 타깃 열 벡터를 비교하여 True의 비율을 반환합니다.
    return np.mean(self.predict(x) == np.argmax(y, axis=1))

def update_val_loss(self, x_val, y_val):
    z = self.forpass(x_val)      # 정방향 계산을 수행합니다.
    a = self.softmax(z)          # 활성화 함수를 적용합니다.
    a = np.clip(a, 1e-10, 1-1e-10) # 출력 값을 클리핑합니다.
    # 크로스 엔트로피 손실과 규제 손실을 더하여 리스트에 추가합니다.
    val_loss = np.sum(-y_val*np.log(a))
    self.val_losses.append((val_loss + self.reg_loss()) / len(y_val))
```

7장

predict(self, x):

return np.argmax( **z** , axis=1)

```
def predict(self, x):
    z = self.forpass(x)          # 정방향 계산을 수행합니다.
    return np.argmax(z.numpy(), axis=1) # 가장 큰 값의 인덱스를 반환합니다.

def score(self, x, y):
    # 예측과 타깃 열 벡터를 비교하여 True의 비율을 반환합니다.
    return np.mean(self.predict(x) == np.argmax(y, axis=1))

def get_loss(self, x, y):
    z = self.forpass(x)          # 정방향 계산을 수행합니다.
    # 손실을 계산하여 저장합니다.
    loss = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(y, z))
    return loss.numpy()
```

8장

predict(self, x):

return np.argmax( **z.numpy()** , axis=1)

# 검증 손실 계산

```
def predict(self, x):
    z = self.forpass(x)          # 정방향 계산을 수행합니다.
    return np.argmax(z, axis=1)  # 가장 큰 값의 인덱스를 반환합니다.

def score(self, x, y):
    # 예측과 타깃 열 벡터를 비교하여 True의 비율을 반환합니다.
    return np.mean(self.predict(x) == np.argmax(y, axis=1))

def update_val_loss(self, x_val, y_val):
    z = self.forpass(x_val)      # 정방향 계산을 수행합니다.
    a = self.softmax(z)          # 활성화 함수를 적용합니다.
    a = np.clip(a, 1e-10, 1-1e-10) # 출력 값을 클리핑합니다.
    # 크로스 엔트로피 손실과 규제 손실을 더하여 리스트에 추가합니다.
    val_loss = np.sum(-y_val*np.log(a))
    self.val_losses.append((val_loss + self.reg_loss()) / len(y_val))
```

7장

update\_val\_loss(self, x\_val, y\_val)

```
a = self.softmax(z)
a = np.clip(a, 1e-10, 1-1e-10)
val_loss = np.sum(-y_val*np.log(a))
```

```
def predict(self, x):
    z = self.forpass(x)          # 정방향 계산을 수행합니다.
    return np.argmax(z.numpy(), axis=1) # 가장 큰 값의 인덱스를 반환합니다.

def score(self, x, y):
    # 예측과 타깃 열 벡터를 비교하여 True의 비율을 반환합니다.
    return np.mean(self.predict(x) == np.argmax(y, axis=1))

def get_loss(self, x, y):
    z = self.forpass(x)          # 정방향 계산을 수행합니다.
    # 손실을 계산하여 저장합니다.
    loss = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(y, z))
    return loss.numpy()
```

8장

get\_loss(self, x, y)

```
loss =
    tf.reduce_mean(
        tf.nn.softmax_cross_entropy_with_logits(y, z)
    )
```

# 합성곱 신경망 훈련

## 1. 데이터 세트 불러오기

```
(x_train_all, y_train_all), (x_test, y_test) = tf.keras.datasets.fashion_mnist.load_data()
```

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/train-labels-idx1-ubyte.gz  
32768/29515 [=====] - 0s 0us/step
```

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/train-images-idx3-ubyte.gz  
26427392/26421880 [=====] - 0s 0us/step
```

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/t10k-labels-idx1-ubyte.gz  
8192/5148 [=====] - 0s 0us/step
```

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/t10k-images-idx3-ubyte.gz  
4423680/4422102 [=====] - 0s 0us/step
```

# 합성곱 신경망 훈련

## 2. 훈련 데이터 세트를 훈련 세트와 검증 세트로 나누기

```
from sklearn.model_selection import train_test_split
x_train, x_val, y_train, y_val = train_test_split(x_train_all, y_train_all, stratify=y_train_all,
                                                  test_size=0.2, random_state=42)
```

## 3. 타깃을 원-핫 인코딩으로 변환

```
y_train_encoded = tf.keras.utils.to_categorical(y_train)
y_val_encoded = tf.keras.utils.to_categorical(y_val)
```

## 4. 입력 데이터 준비

```
x_train = x_train.reshape(-1, 28, 28, 1)
x_val = x_val.reshape(-1, 28, 28, 1)
```

# 합성곱 신경망 훈련

## 5. 입력 데이터 표준화 전처리

```
x_train = x_train / 255  
x_val = x_val / 255
```

## 6. 모델 훈련

커널 개수

뉴런 개수

배치 크기

학습률

```
cn = ConvolutionNetwork(n_kernels=10, units=100, batch_size=128, learning_rate=0.01)  
cn.fit(x_train, y_train_encoded,  
       x_val=x_val, y_val=y_val_encoded, epochs=20)
```

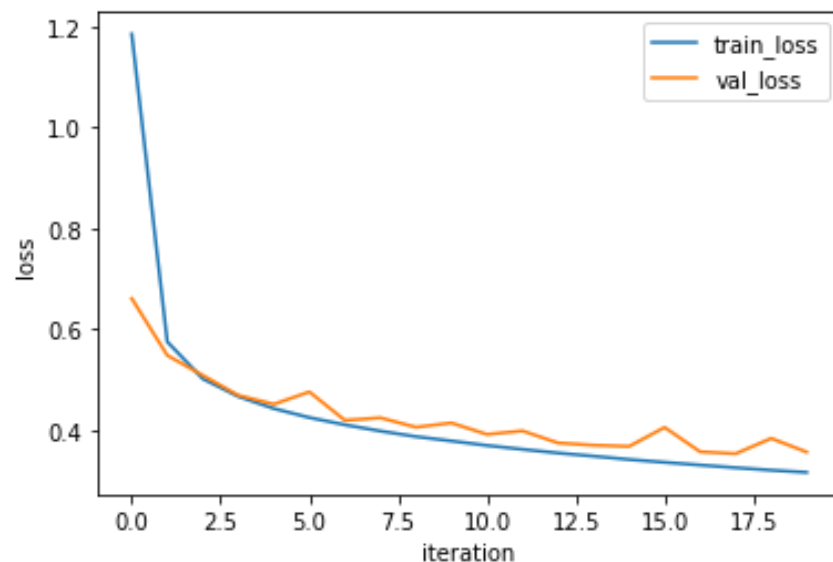
```
에포크 0 .....  
에포크 1 .....  
에포크 2 .....
```

·  
·  
·

# 합성곱 신경망 훈련

## 7. 훈련, 검증 손실 그래프

```
plt.plot(cn.losses)
plt.plot(cn.val_losses)
plt.ylabel('loss')
plt.xlabel('iteration')
plt.legend(['train_loss', 'val_loss'])
plt.show()
```



## 8. 검증 세트의 정확도 확인

```
cn.score(x_val, y_val_encoded)
```

0.8771666666666667

● ● ● 7장  
87.7% > 81.5%

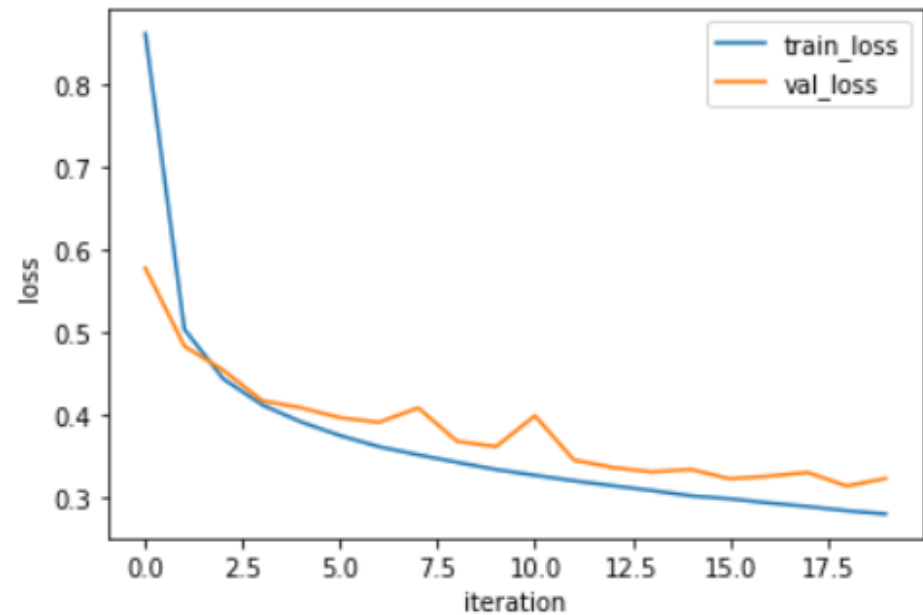
# He 초기화 훈련

```
def init_weights(self, input_shape, n_classes):  
    g = tf.initializers.he_uniform() #he_uniform.  
    self.conv_w = tf.Variable(g((3, 3, 1, self.n_kernels)))  
    self.conv_b = tf.Variable(np.zeros(self.n_kernels), dtype=float)  
    n_features = 14 * 14 * self.n_kernels  
    self.w1 = tf.Variable(g((n_features, self.units))) # (특성 개수, 은닉층의 크기)  
    self.b1 = tf.Variable(np.zeros(self.units), dtype=float) # 은닉층의 크기  
    self.w2 = tf.Variable(g((self.units, n_classes))) # (은닉층의 크기, 클래스 개수)  
    self.b2 = tf.Variable(np.zeros(n_classes), dtype=float) # 클래스 개수
```

# He 초기화 훈련

## 훈련, 검증 손실 그래프

```
plt.plot(cn.losses)
plt.plot(cn.val_losses)
plt.ylabel('loss')
plt.xlabel('iteration')
plt.legend(['train_loss', 'val_loss'])
plt.show()
```



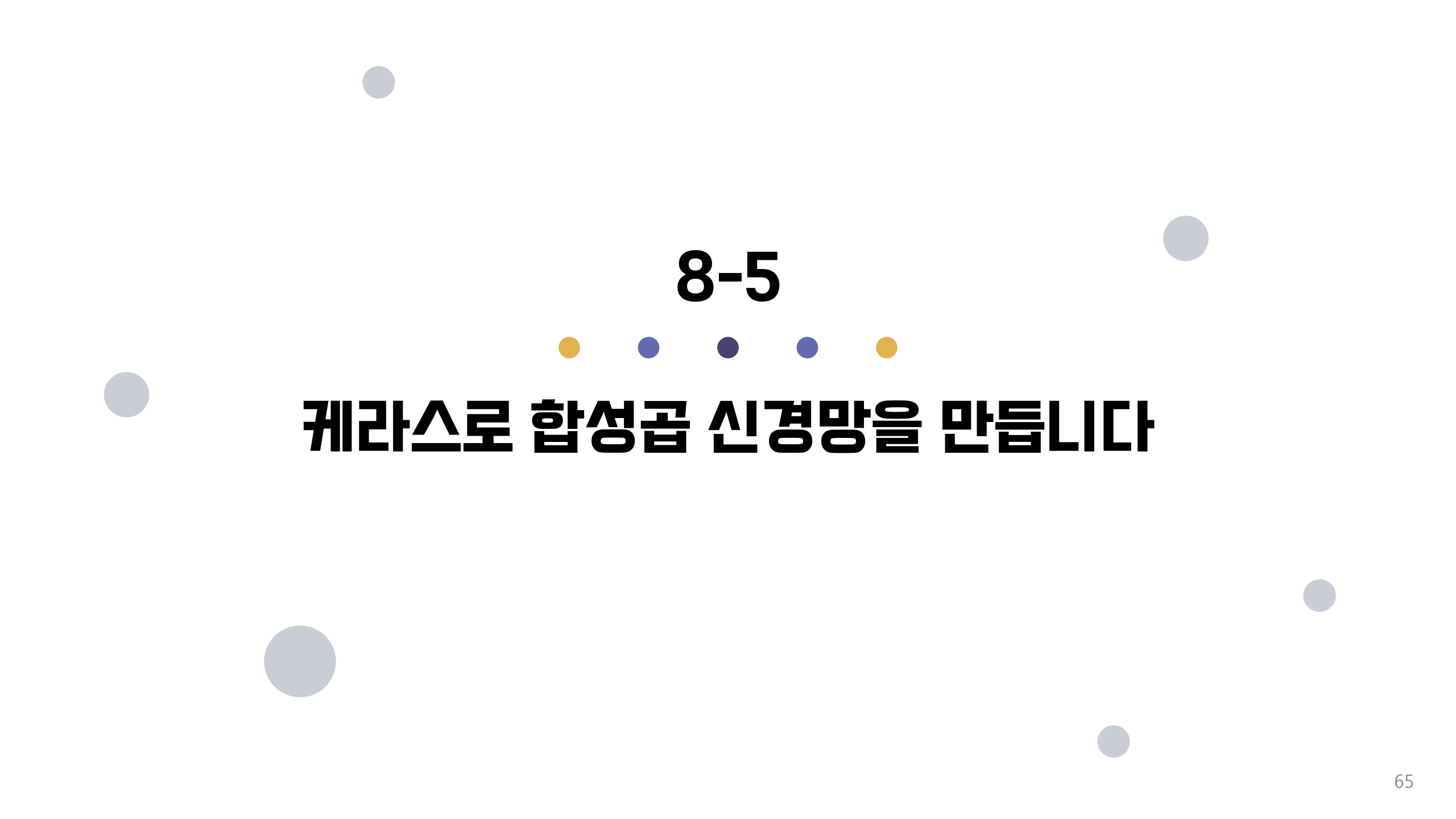
## 정확도 확인

```
cn.score(x_val, y_val_encoded)
```

0.88625

● ● ● Xavier  
88.6% > 87.7%





8-5



**케라스로 합성곱 신경망을 만듭니다**

# 케라스로 합성곱 신경망 만들기

## 1. 데이터 세트 불러오기

```
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense
```

## 2. 합성곱층 쌓기

```
conv1 = tf.keras.Sequential()  
conv1.add(Conv2D(10, (3, 3), activation='relu', padding='same', input_shape=(28, 28, 1)))
```

커널 개수    커널 크기

높이, 너비, 컬러채널  
배치 차원을 제외한 입력의 크기

# 케라스로 합성곱 신경망 만들기

## 3. 풀링층 쌓기

```
conv1.add(MaxPooling2D((2, 2)))
```

(높이, 너비)

Strides = 풀링 크기(기본값). Padding = 'valid'(기본값)

## 4. 완전 연결층에 주입할 수 있도록 특성 맵 펼치기

```
conv1.add(Flatten())
```

## 5. 완전 연결층 쌓기

```
conv1.add(Dense(100, activation='relu'))  
conv1.add(Dense(10, activation='softmax'))
```

# 케라스로 합성곱 신경망 만들기

## 6. 모델 구조 살펴보기

```
conv1.summary()
```

```
Model: "sequential"
```

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 28, 28, 10)	100
max_pooling2d (MaxPooling2D)	(None, 14, 14, 10)	0
flatten (Flatten)	(None, 1960)	0
dense (Dense)	(None, 100)	196100
dense_1 (Dense)	(None, 10)	1010

```
=====
Total params: 197,210
Trainable params: 197,210
Non-trainable params: 0
=====
```

크기

파라미터 개수

28 x 28 x 10

$3 \times 3 \times 1 \times 10 + 10$

14 x 14 x 10

$14 \times 14 \times 10 \times 100 + 100$

$10 \times 100 + 10$

# 합성곱 신경망 훈련

## 1. 모델 컴파일

```
conv1.compile(optimizer='adam', loss='categorical_crossentropy',  
              metrics=['accuracy'])
```

### Adaptive Moment Estimation

손실함수의 값이 최적 값에 가까워질수록 학습율을 낮춰 손실 함수의 값이 안정적으로 수렴

# 합성곱 신경망 훈련

## 2. 아담 옵티마이저 사용

```
history = conv1.fit(x_train, y_train_encoded, epochs=20,  
                    validation_data=(x_val, y_val_encoded))
```

```
Epoch 1/20  
1500/1500 [=====] - 12s 4ms/step - loss: 0.5890 - accuracy: 0.8004 - val_loss: 0.3309 - val_accuracy: 0.8803  
Epoch 2/20  
1500/1500 [=====] - 6s 4ms/step - loss: 0.2993 - accuracy: 0.8909 - val_loss: 0.2882 - val_accuracy: 0.8976  
Epoch 3/20  
1500/1500 [=====] - 6s 4ms/step - loss: 0.2476 - accuracy: 0.9078 - val_loss: 0.2748 - val_accuracy: 0.9008  
Epoch 4/20  
1500/1500 [=====] - 6s 4ms/step - loss: 0.2254 - accuracy: 0.9164 - val_loss: 0.2554 - val_accuracy: 0.9076  
Epoch 5/20  
1500/1500 [=====] - 6s 4ms/step - loss: 0.1994 - accuracy: 0.9257 - val_loss: 0.2504 - val_accuracy: 0.9125
```

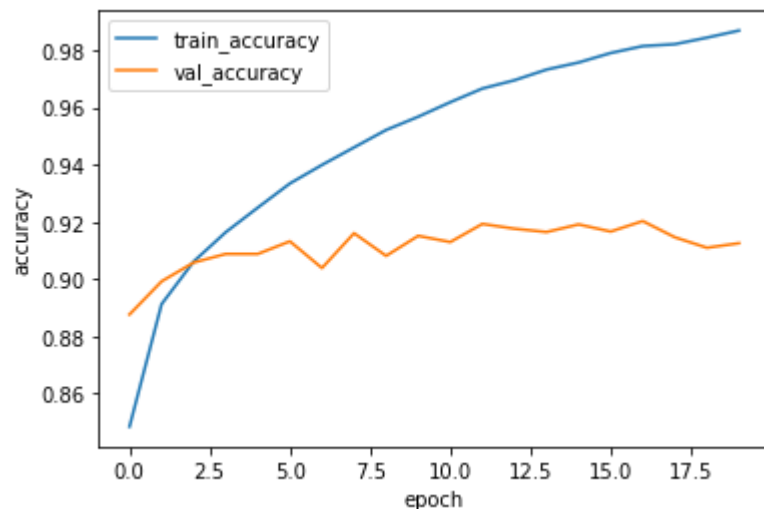
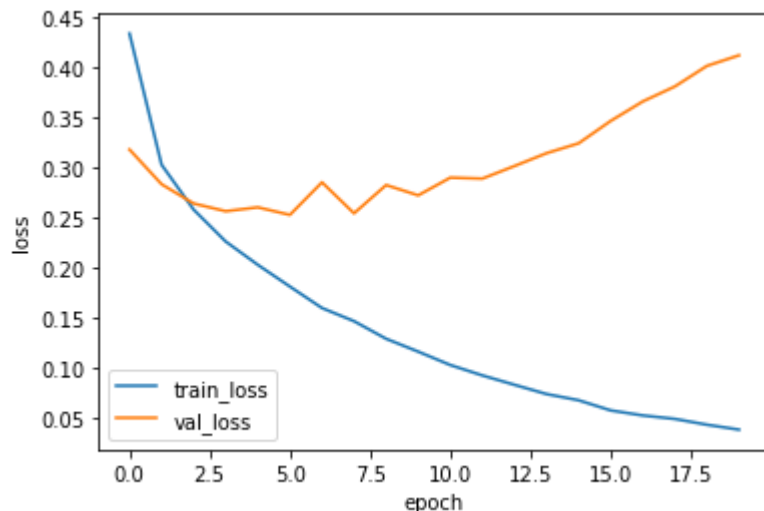
•  
•  
•

# 합성곱 신경망 훈련

## 3. 손실, 정확도 그래프

```
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train_loss', 'val_loss'])
plt.show()
```

```
plt.plot(history.history['accuracy'])
plt.plot(history.history['val_accuracy'])
plt.ylabel('accuracy')
plt.xlabel('epoch')
plt.legend(['train_accuracy', 'val_accuracy'])
plt.show()
```



# 합성곱 신경망 훈련

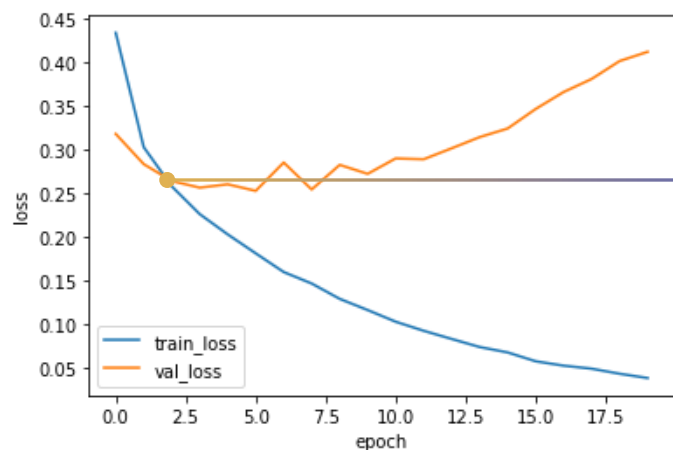
## 4. 정확도

```
loss, accuracy = conv1.evaluate(x_val, y_val_encoded, verbose=0)

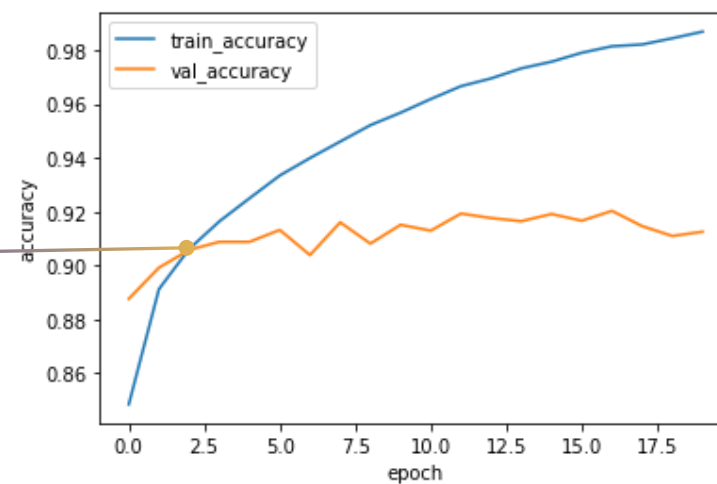
print(accuracy)

0.9124166369438171
```

before  
91.2% > 87.7%



Error  
과대적합

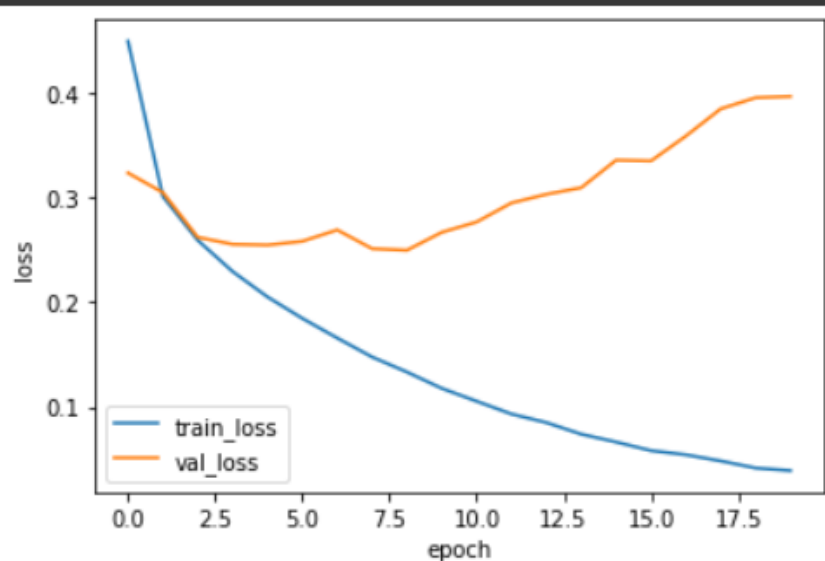




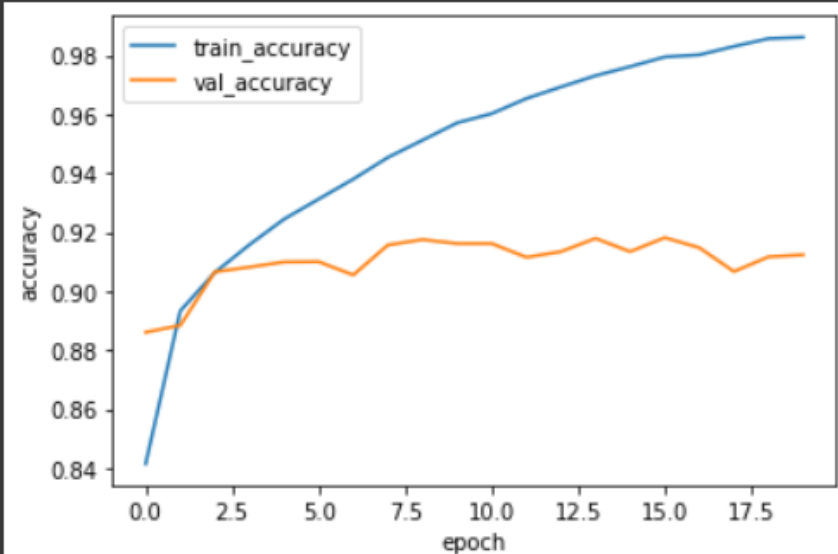
# 케라스로 He 초기화 훈련

## 손실, 정확도 그래프

```
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train_loss', 'val_loss'])
plt.show()
```



```
plt.plot(history.history['accuracy'])
plt.plot(history.history['val_accuracy'])
plt.ylabel('accuracy')
plt.xlabel('epoch')
plt.legend(['train_accuracy', 'val_accuracy'])
plt.show()
```



# 케라스로 He 초기화 훈련

정확도

```
loss, accuracy = conv1.evaluate(x_val, y_val_encoded, verbose=0)

print(accuracy)

0.9123333096504211
```

Xavier

→ 91.23% < 91.24%

# 드롭아웃

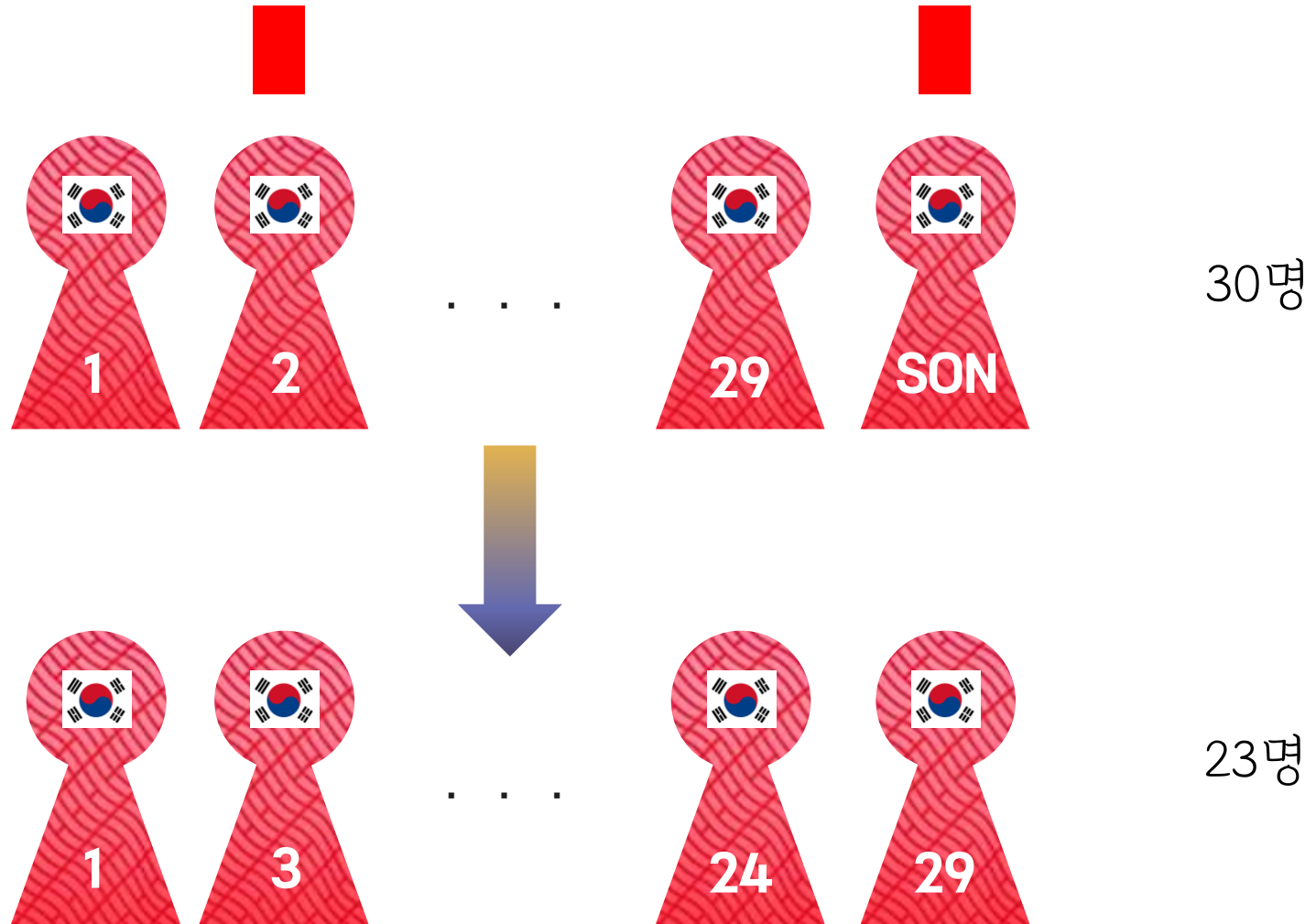
개념:

무작위로 일부 뉴런 비활성화 시켜  
특정 뉴런에 과도하게 의존하여 훈련하는 것을 방지

# 드롭아웃



# 드롭아웃



# 드롭아웃

개념:

무작위로 일부 뉴런 비활성화 시켜  
특정 뉴런에 과도하게 의존하여 훈련하는 것을 방지

모델을 훈련시킬 때만 적용하는 기법  
테스트, 실전 적용 X

상대적으로, 테스트와 실전의 출력값 > 훈련 출력값

테스트나 실전에서는 출력값을 드롭아웃 비율만큼 낮춰야 함

# 드롭아웃

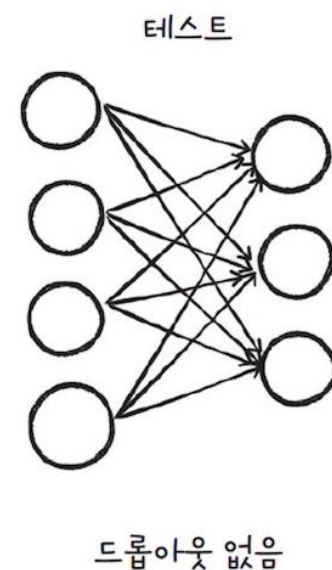
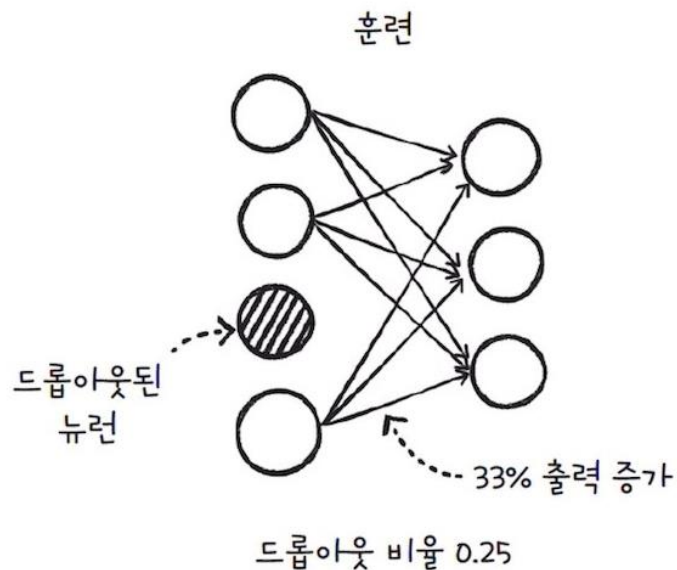
## 텐서플로 & 딥러닝 프레임워크

훈련 할 때 드롭아웃의 비율만큼 뉴런의 출력을 높여서 훈련

드롭아웃의 비율 = 0.25

$$\frac{1}{1 - 0.25} = 1.33 \dots$$

33% 출력 증가



# 드롭아웃 적용해 합성곱 신경망 구현

## 1. 케라스로 만든 합성곱 신경망에 드롭아웃 적용

```
from tensorflow.keras.layers import Dropout

conv2 = tf.keras.Sequential()
conv2.add(Conv2D(10, (3, 3), activation='relu', padding='same', input_shape=(28, 28, 1)))
conv2.add(MaxPooling2D((2, 2)))
conv2.add(Flatten())
conv2.add(Dropout(0.5))
conv2.add(Dense(100, activation='relu'))
conv2.add(Dense(10, activation='softmax'))
```



# 드롭아웃 적용해 합성곱 신경망 구현

## 2. 드롭아웃층 확인

```
conv2.summary()
```

```
Model: "sequential_1"
```

Layer (type)	Output Shape	Param #
=====		
conv2d_1 (Conv2D)	(None, 28, 28, 10)	100
max_pooling2d_1 (MaxPooling2D)	(None, 14, 14, 10)	0
flatten_1 (Flatten)	(None, 1960)	0
dropout (Dropout)	(None, 1960)	0
dense_2 (Dense)	(None, 100)	196100
dense_3 (Dense)	(None, 10)	1010
=====		
Total params: 197,210		
Trainable params: 197,210		
Non-trainable params: 0		

훈련되는 가중치 X

텐서의 차원 유지

# 드롭아웃 적용해 합성곱 신경망 구현

## 3. 훈련

```
conv2.compile(optimizer='adam', loss='categorical_crossentropy',  
              metrics=['accuracy'])
```

```
history = conv2.fit(x_train, y_train_encoded, epochs=20,  
                   validation_data=(x_val, y_val_encoded))
```

```
Epoch 1/20  
1500/1500 [=====] - 7s 4ms/step - loss: 0.6617 - accuracy: 0.7664 - val_loss: 0.3429 - val_accuracy: 0.8806  
Epoch 2/20  
1500/1500 [=====] - 6s 4ms/step - loss: 0.3853 - accuracy: 0.8604 - val_loss: 0.3116 - val_accuracy: 0.8899  
Epoch 3/20  
1500/1500 [=====] - 6s 4ms/step - loss: 0.3441 - accuracy: 0.8731 - val_loss: 0.2965 - val_accuracy: 0.8938  
Epoch 4/20  
1500/1500 [=====] - 6s 4ms/step - loss: 0.3050 - accuracy: 0.8851 - val_loss: 0.2852 - val_accuracy: 0.8963
```

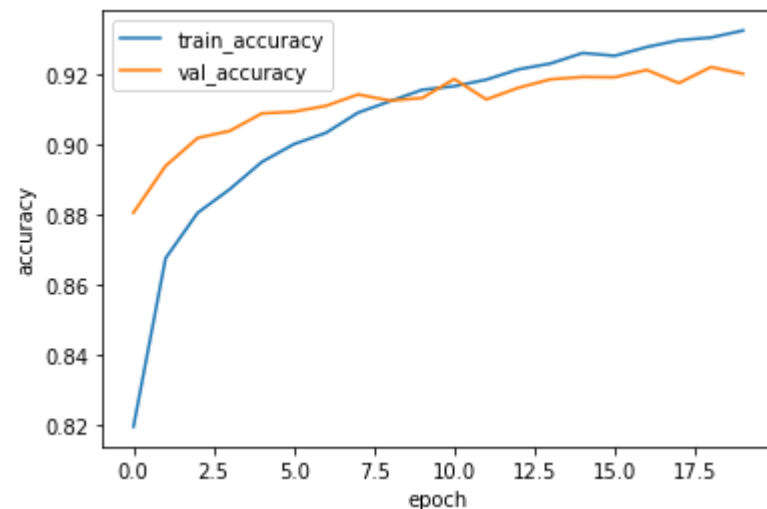
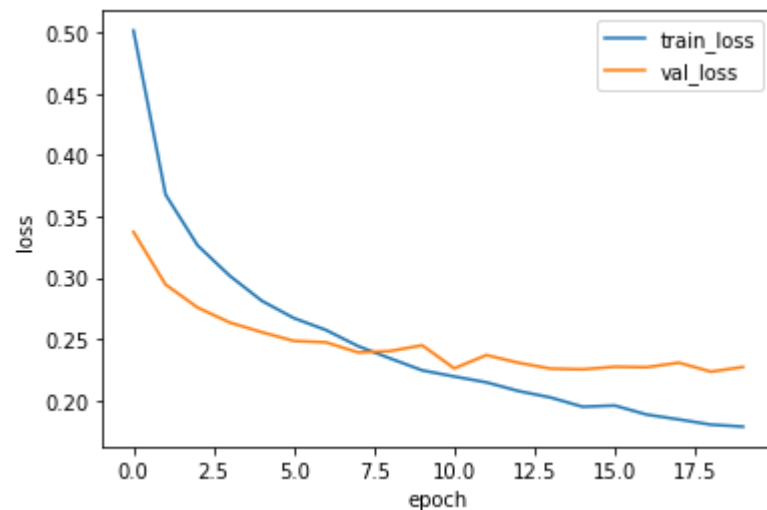
•  
•  
•

# 드롭아웃 적용해 합성곱 신경망 구현

## 4. 손실, 정확도 그래프

```
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train_loss', 'val_loss'])
plt.show()
```

```
plt.plot(history.history['accuracy'])
plt.plot(history.history['val_accuracy'])
plt.ylabel('accuracy')
plt.xlabel('epoch')
plt.legend(['train_accuracy', 'val_accuracy'])
plt.show()
```



# 드롭아웃 적용해 합성곱 신경망 구현

## 5. 정확도

```
loss, accuracy = conv2.evaluate(x_val, y_val_encoded, verbose=0)

print (accuracy)

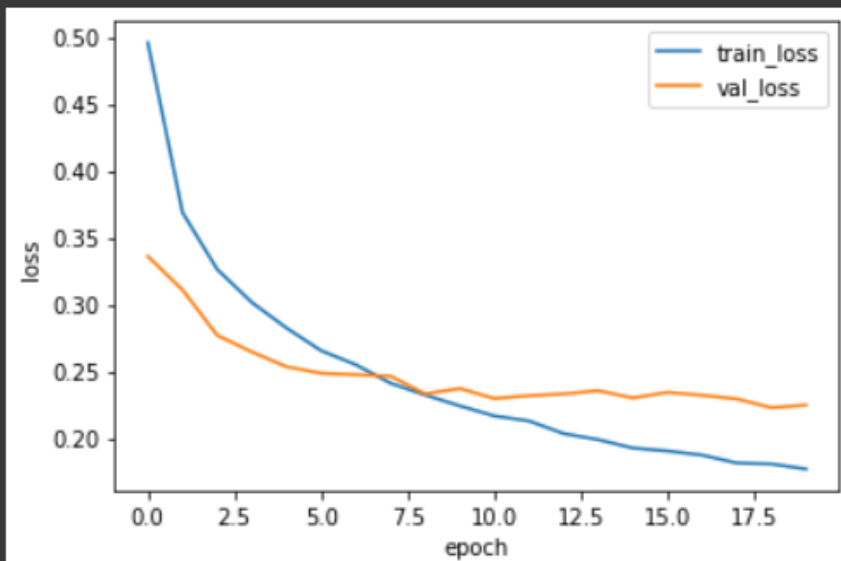
0.9210000038146973
```

● ● ● before  
92.1% > 91.2%

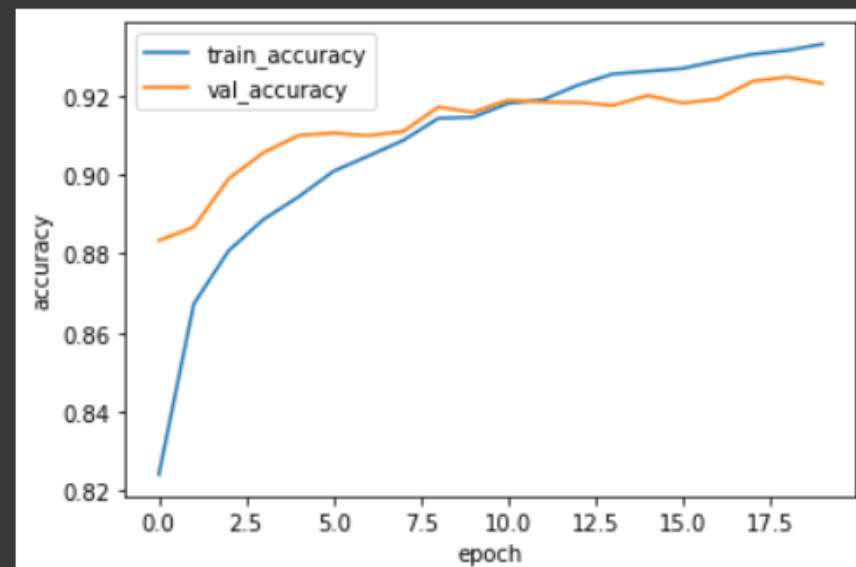
# 드롭아웃 적용해 He 초기화 훈련

## 손실, 정확도 그래프

```
plt.plot(history.history['loss'])  
plt.plot(history.history['val_loss'])  
plt.ylabel('loss')  
plt.xlabel('epoch')  
plt.legend(['train_loss', 'val_loss'])  
plt.show()
```



```
plt.plot(history.history['accuracy'])  
plt.plot(history.history['val_accuracy'])  
plt.ylabel('accuracy')  
plt.xlabel('epoch')  
plt.legend(['train_accuracy', 'val_accuracy'])  
plt.show()
```



# 드롭아웃 적용해 He 초기화 훈련

정확도

```
loss, accuracy = conv2.evaluate(x_val, y_val_encoded, verbose=0)

print(accuracy)

0.9229999780654907
```

● ● ● Xavier  
92.3% > 92.1%



THANK YOU

## - Reference -

[Do it! 딥러닝 입문 8장 - Google Slides](#)

[합성곱 신경망\(Convolution Neural Network\) - 딥 러닝을 이용한 자연어 처리 입문 \(wikidocs.net\)](#)