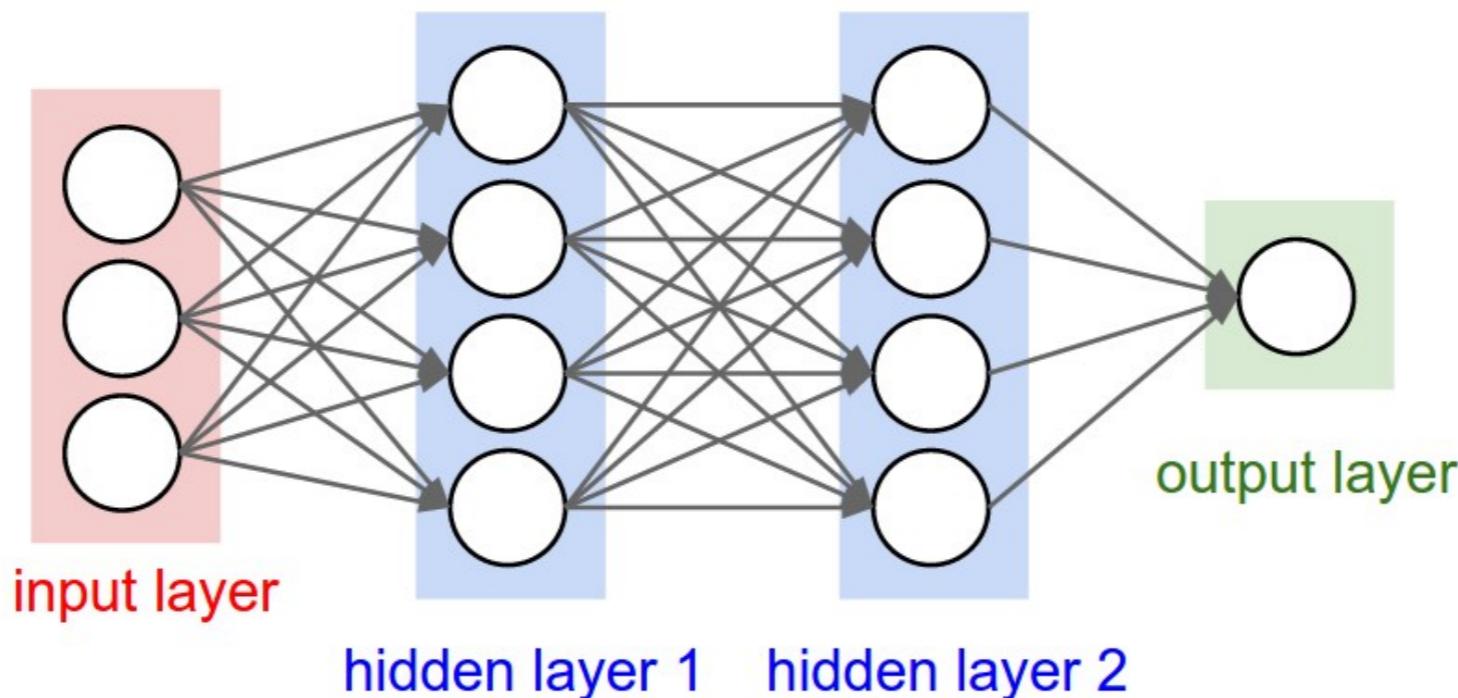


# Convolutional Neural Network

안남혁

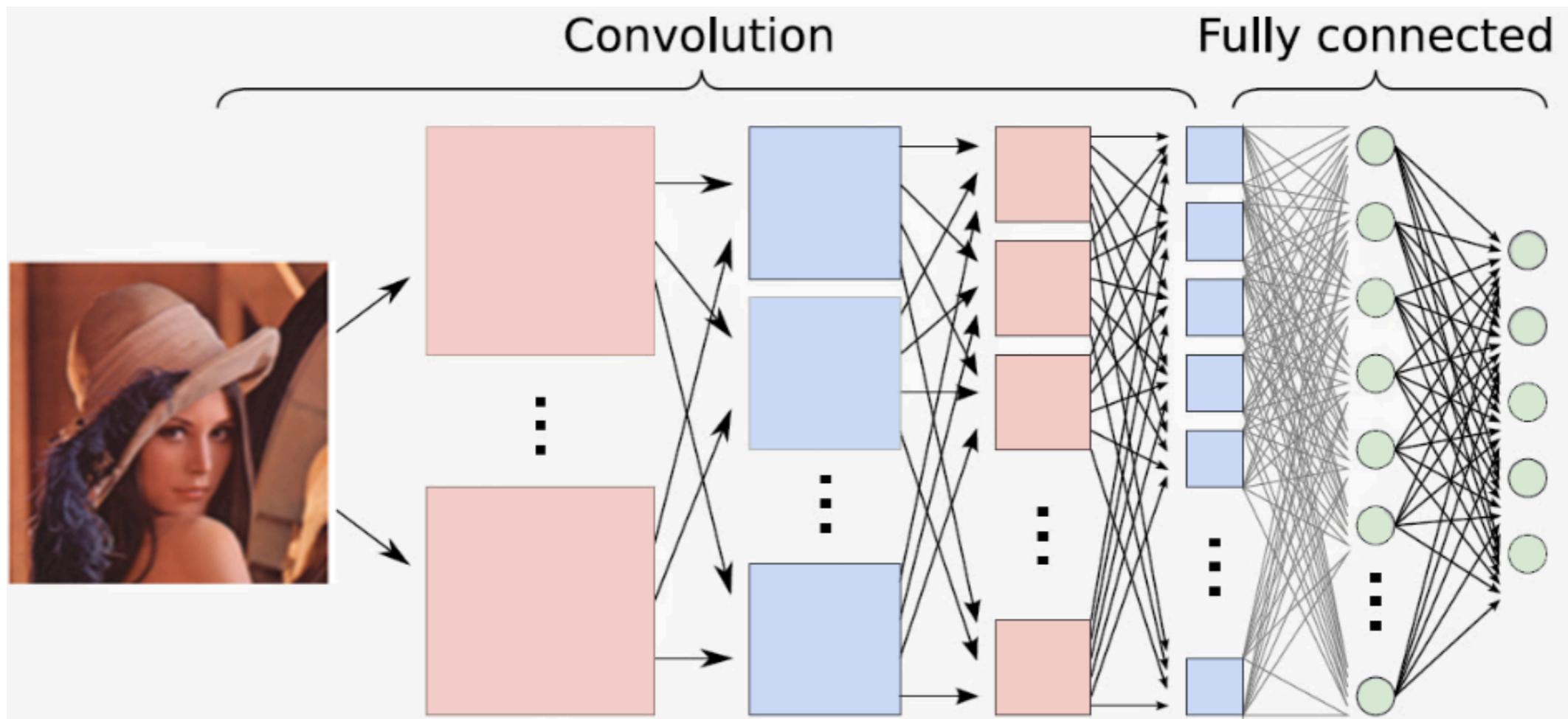
# 뉴럴 네트워크

- **입력층, 은닉층, 출력층**으로 이루어진 멀티 레이어 퍼셉트론
- 각 층은 **활성함수**(activation function)가 존재
- 뉴럴 네트워크의 학습은 back propagation + SGD에 의해



# 컨볼루션 뉴럴 네트워크

- 이미지를 처리할 때 많이 사용되는 네트워크
  - 컨볼루션(convolution), 풀링(pooling), FC 레이어로 구성
  - 인간의 시각 인지 작용을 모방한 디자인



# 컨볼루션

- 입력값과 필터를 통해 결과값을 도출하는 과정
  - 벡터의 내적과 비슷하게 계산 가능

1	1	1	0	0
0	0	1	1	1
0	1	1	0	0
0	0	0	0	1
1	0	0	0	1

입력

0	1	0
1	1	1
0	1	0

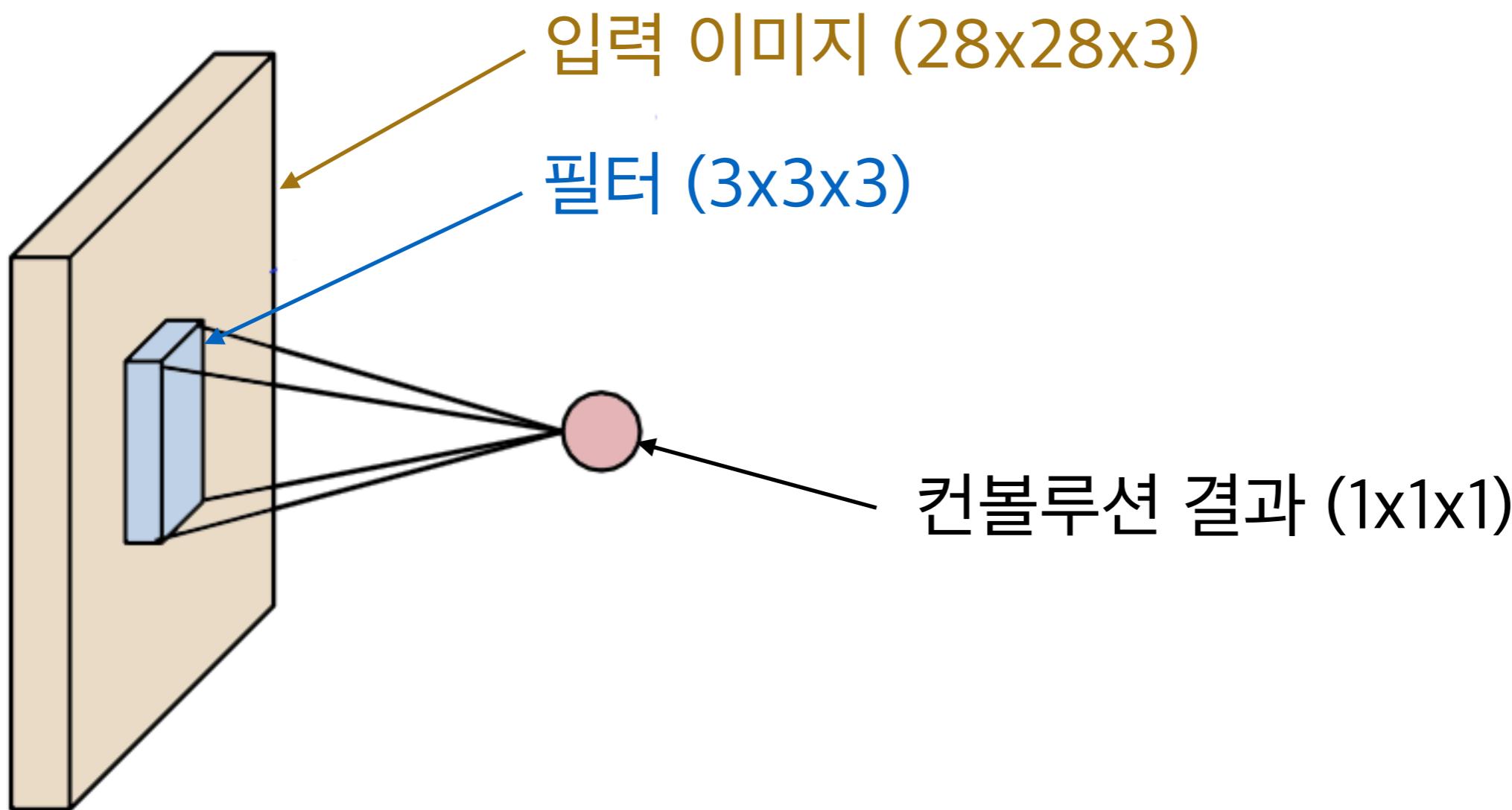
X

3	4	3
2	3	2
1	1	1

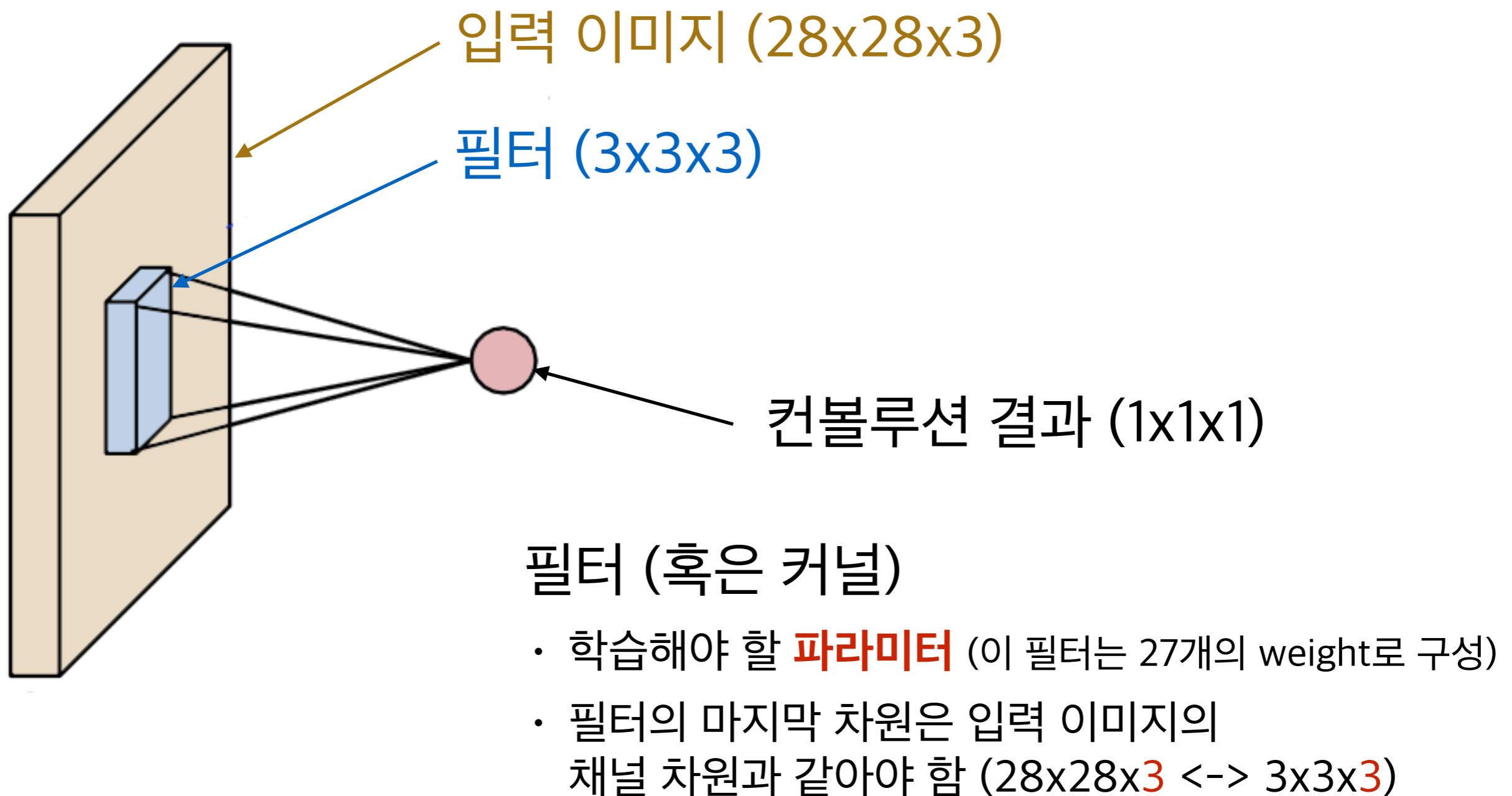
필터

결과

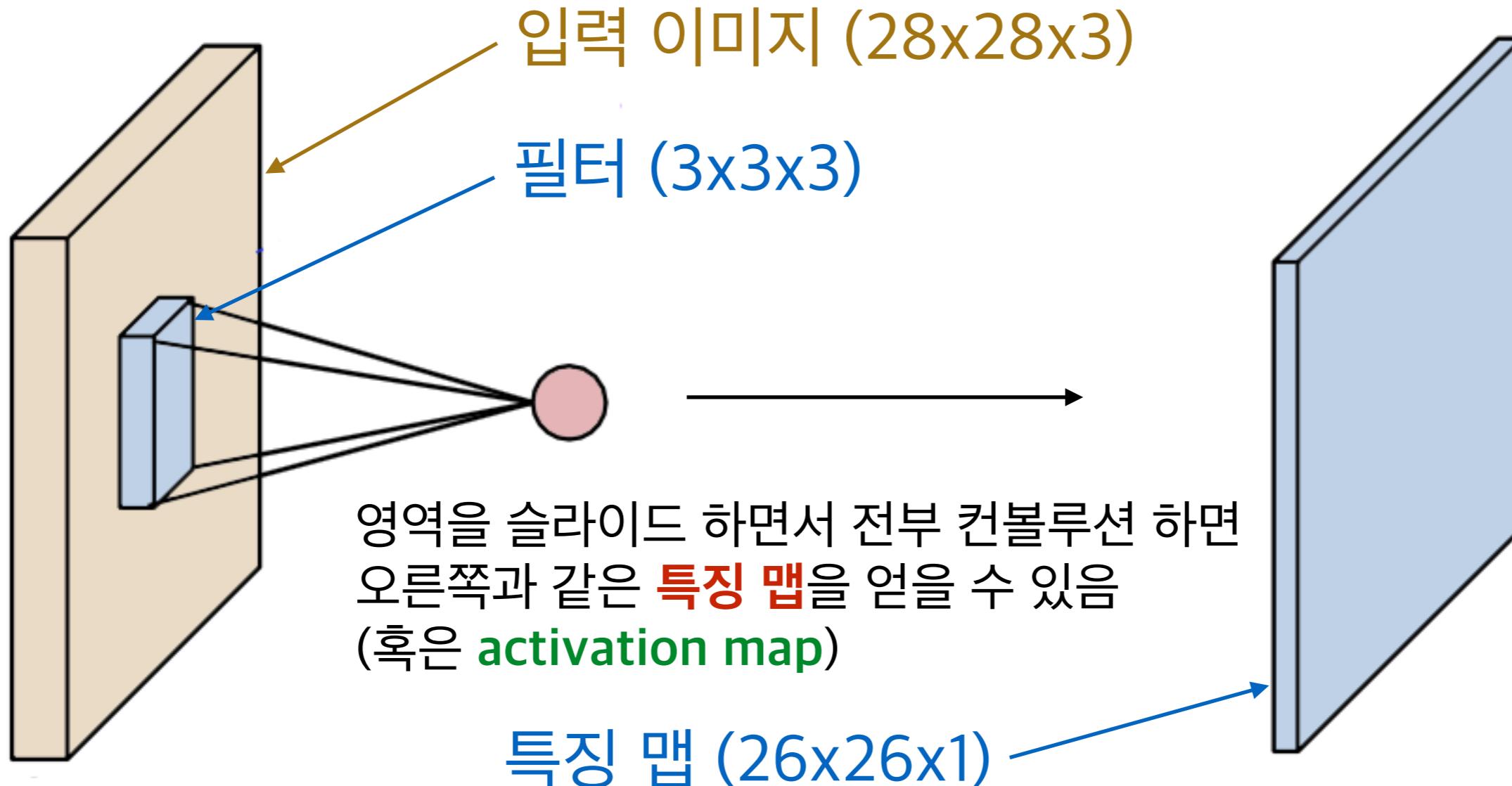
# 컨볼루션 레이어



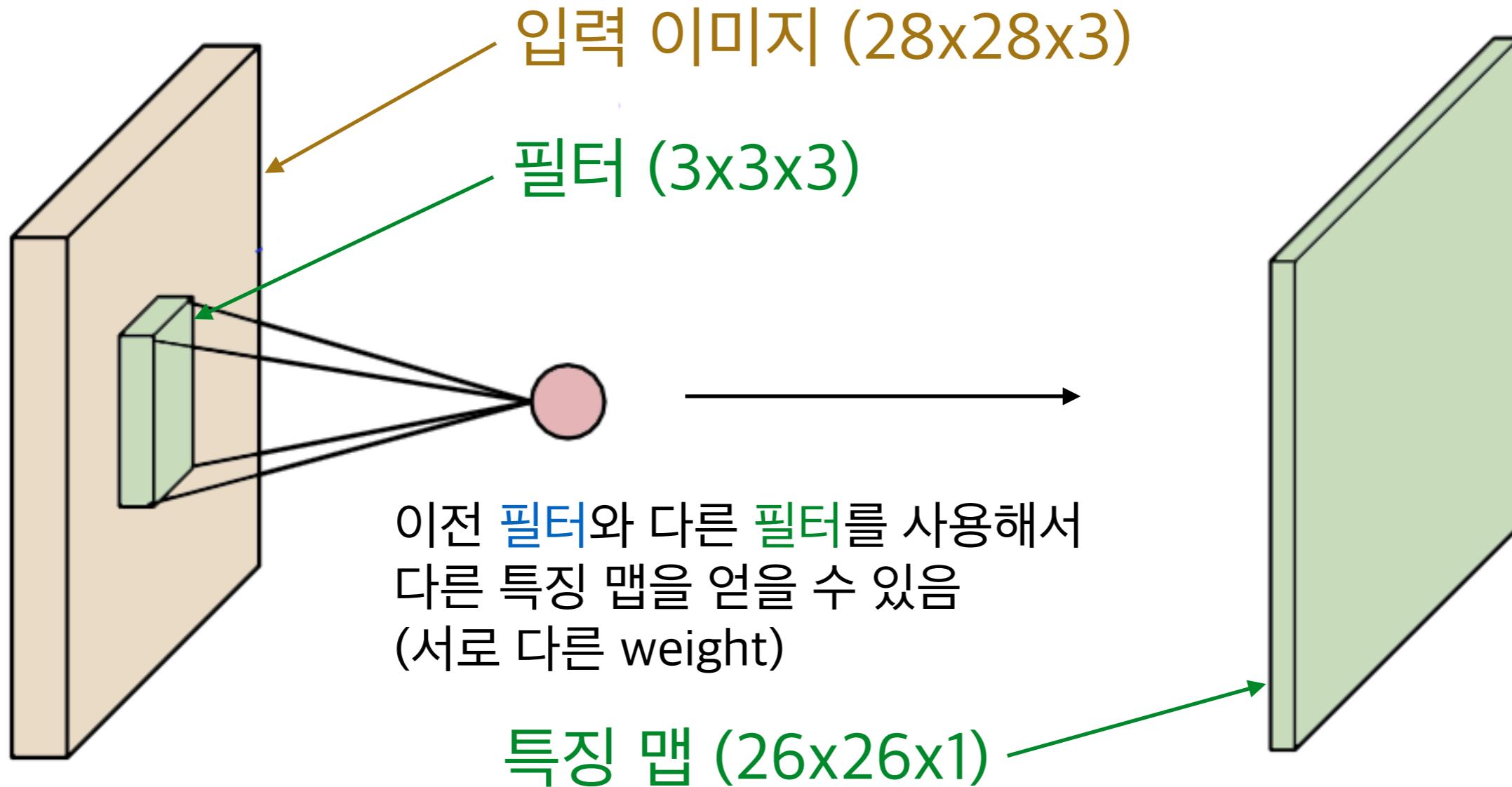
# 컨볼루션 레이어



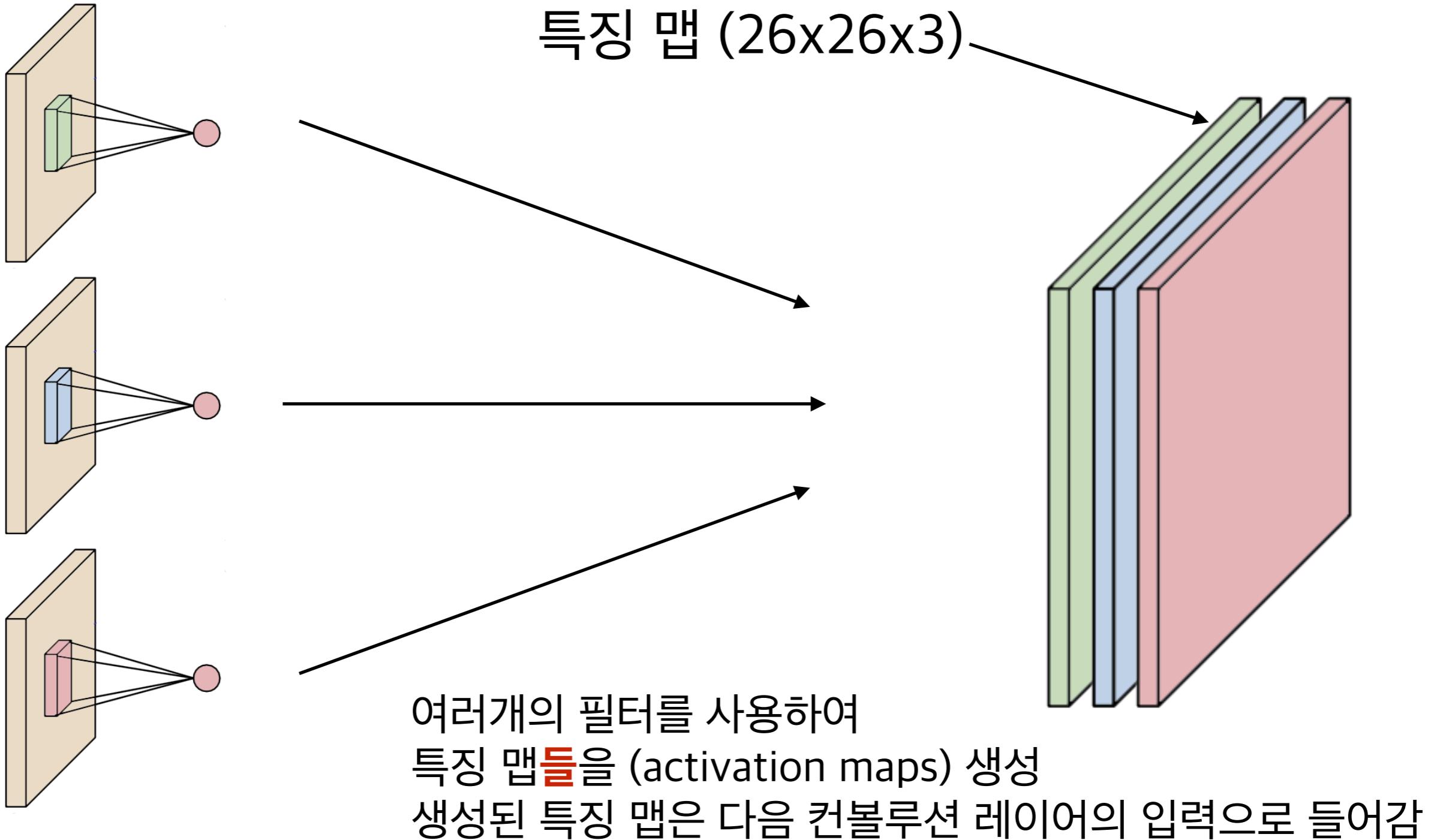
# 컨볼루션 레이어



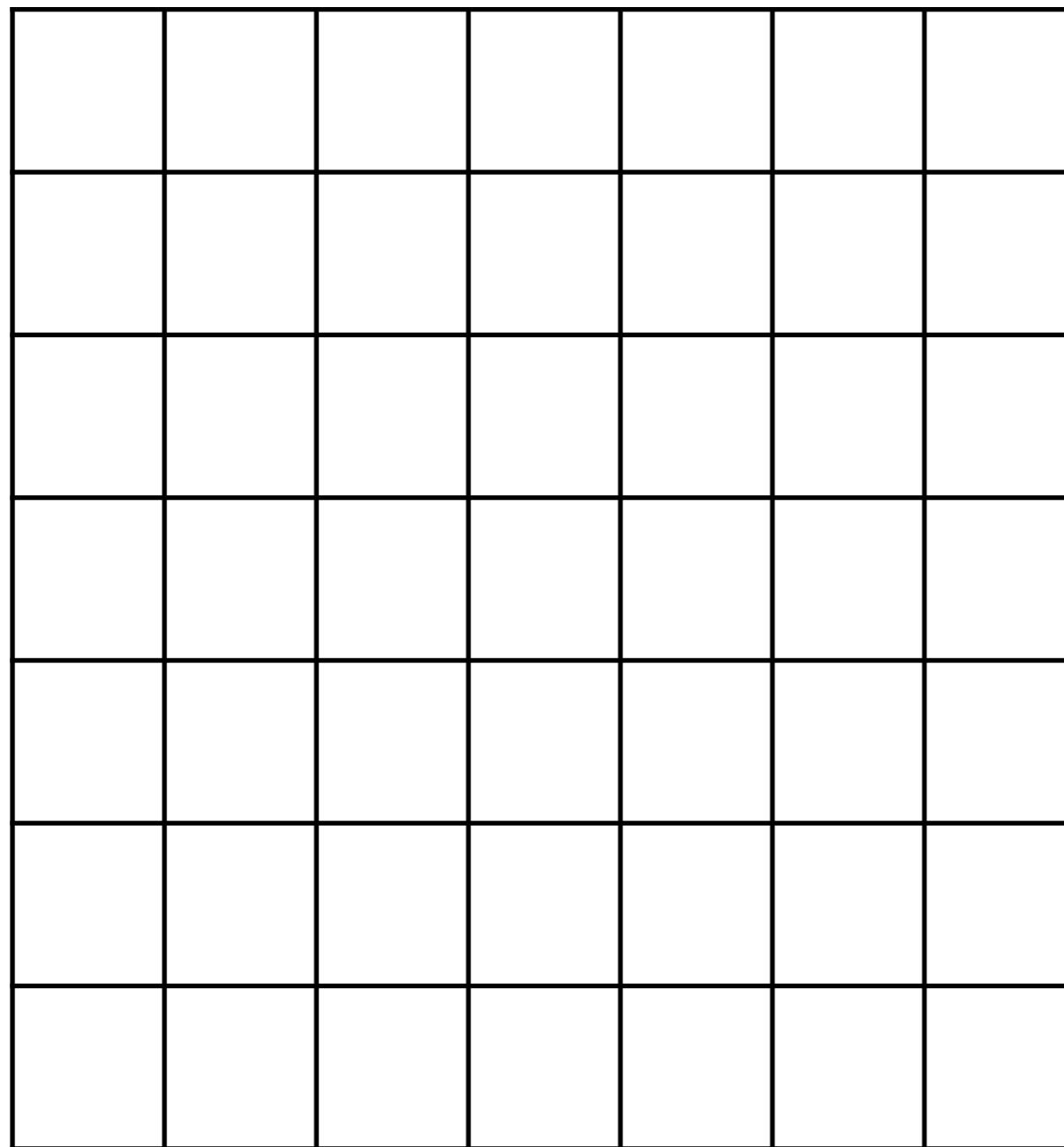
# 컨볼루션 레이어



# 컨볼루션 레이어



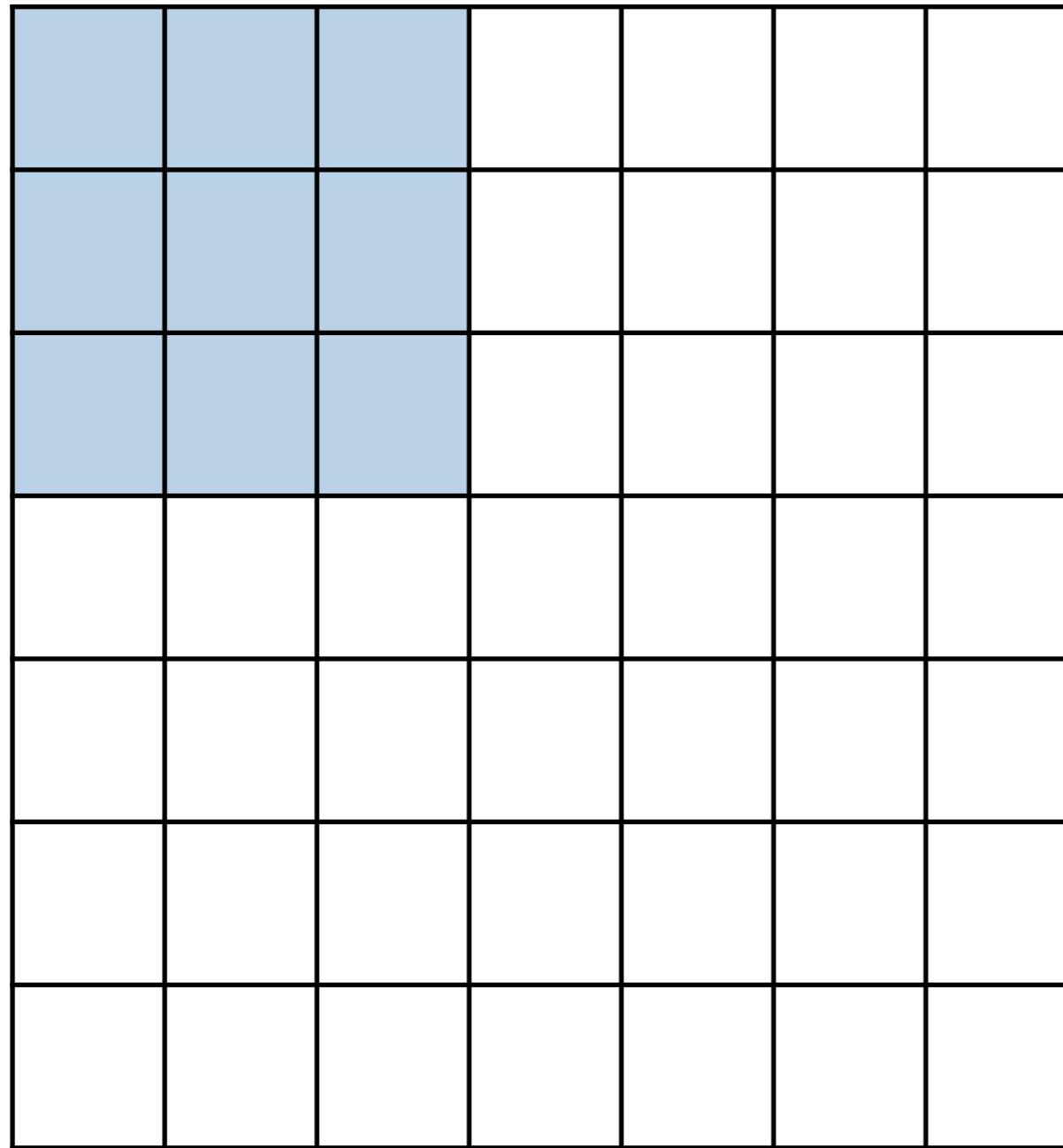
# 컨볼루션 예제



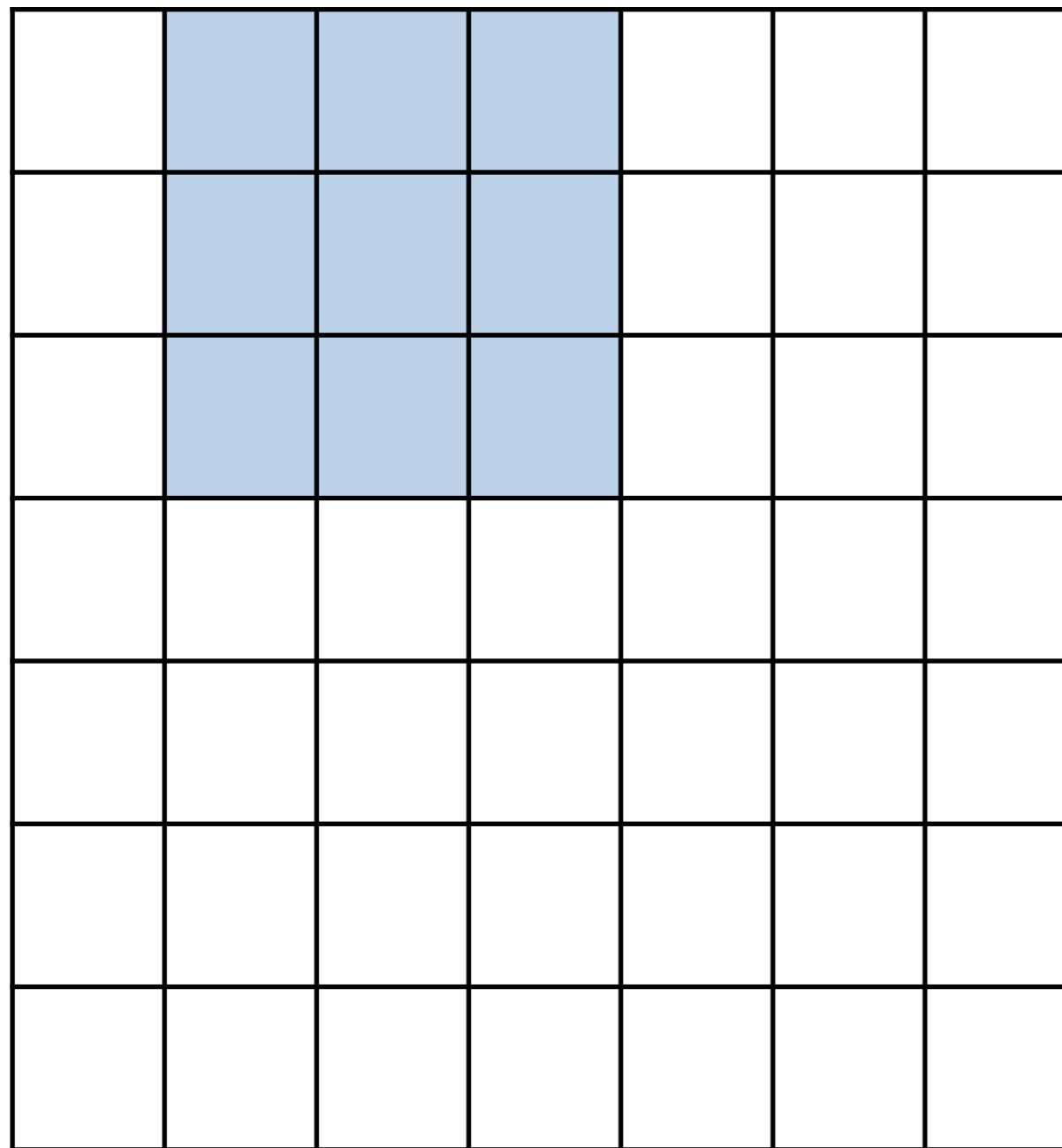
7

7

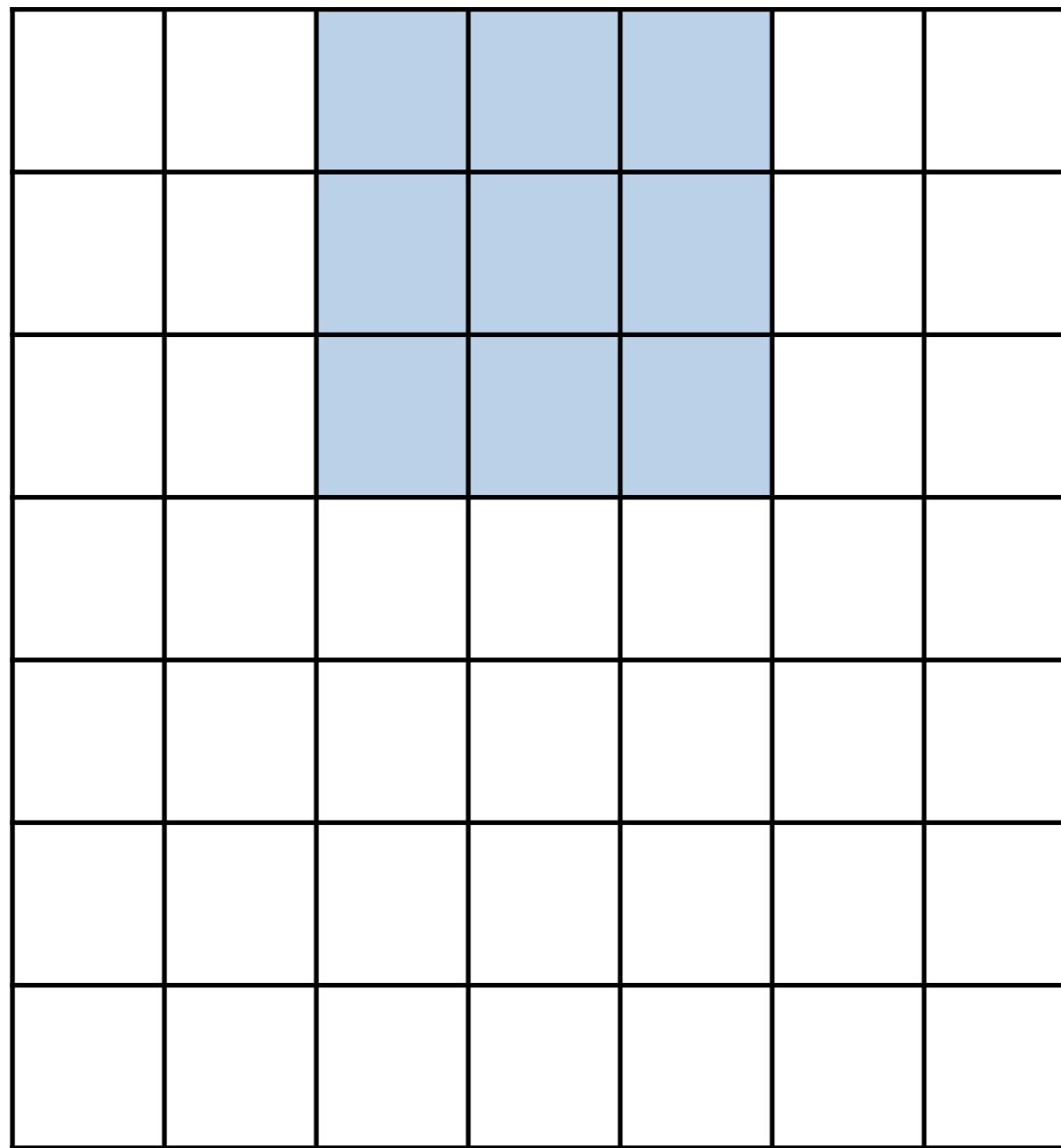
# 컨볼루션 예제 (3x3 필터)



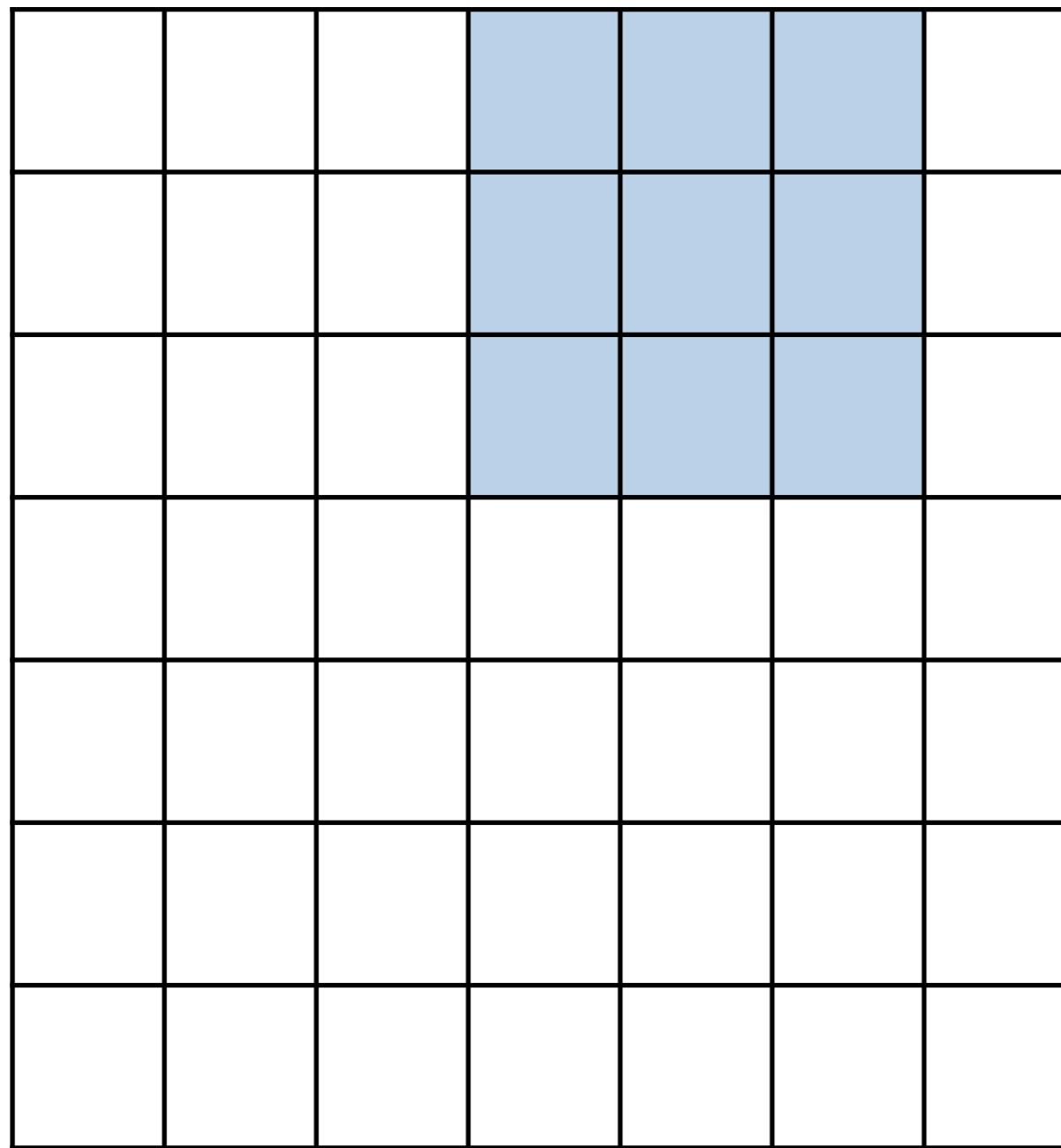
# 컨볼루션 예제 (3x3 필터)



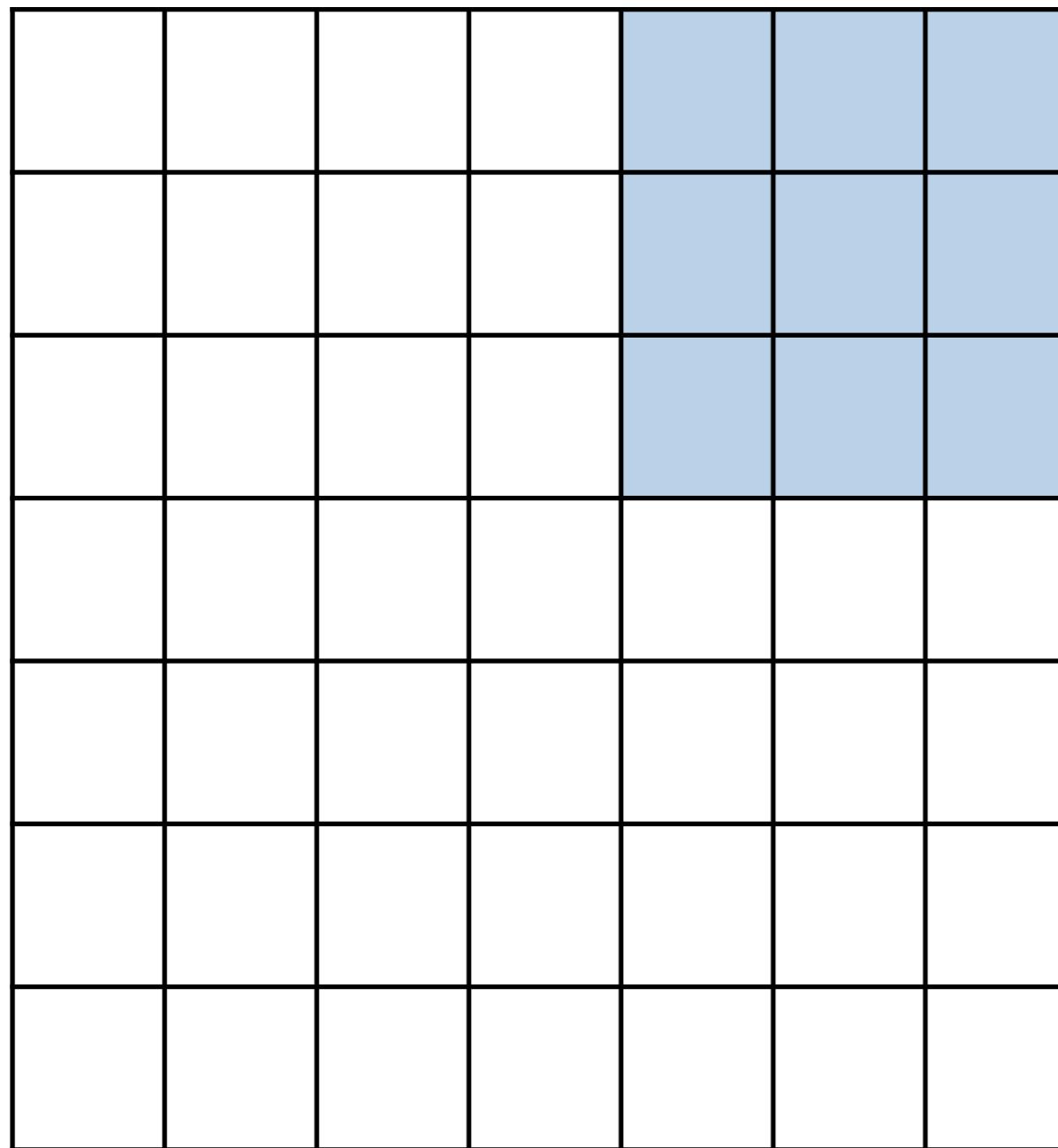
# 컨볼루션 예제 (3x3 필터)



# 컨볼루션 예제 (3x3 필터)

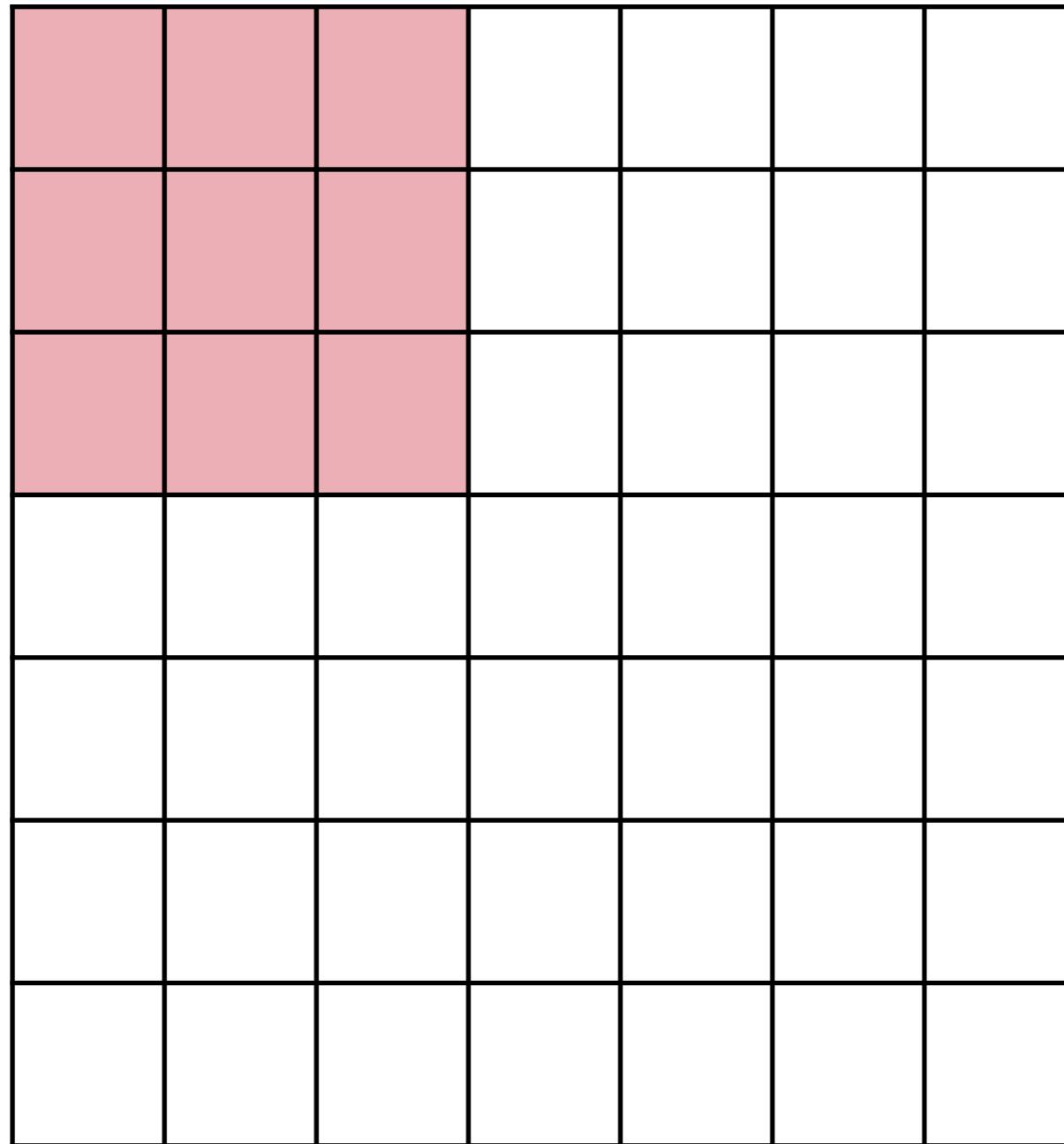


# 컨볼루션 예제 (3x3 필터)



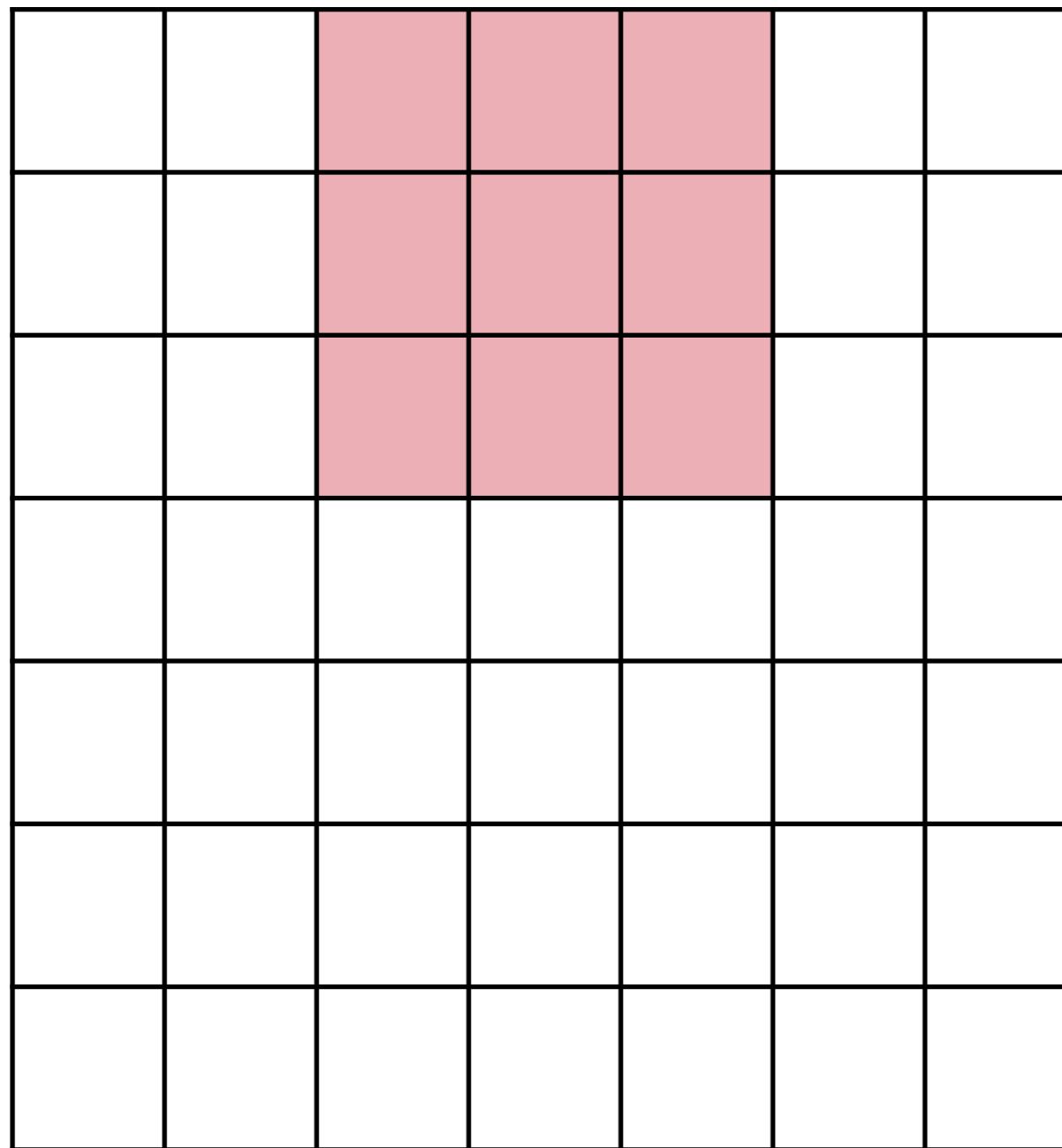
-> 5x5 특징 맵

# 컨볼루션 예제 (3x3 필터)



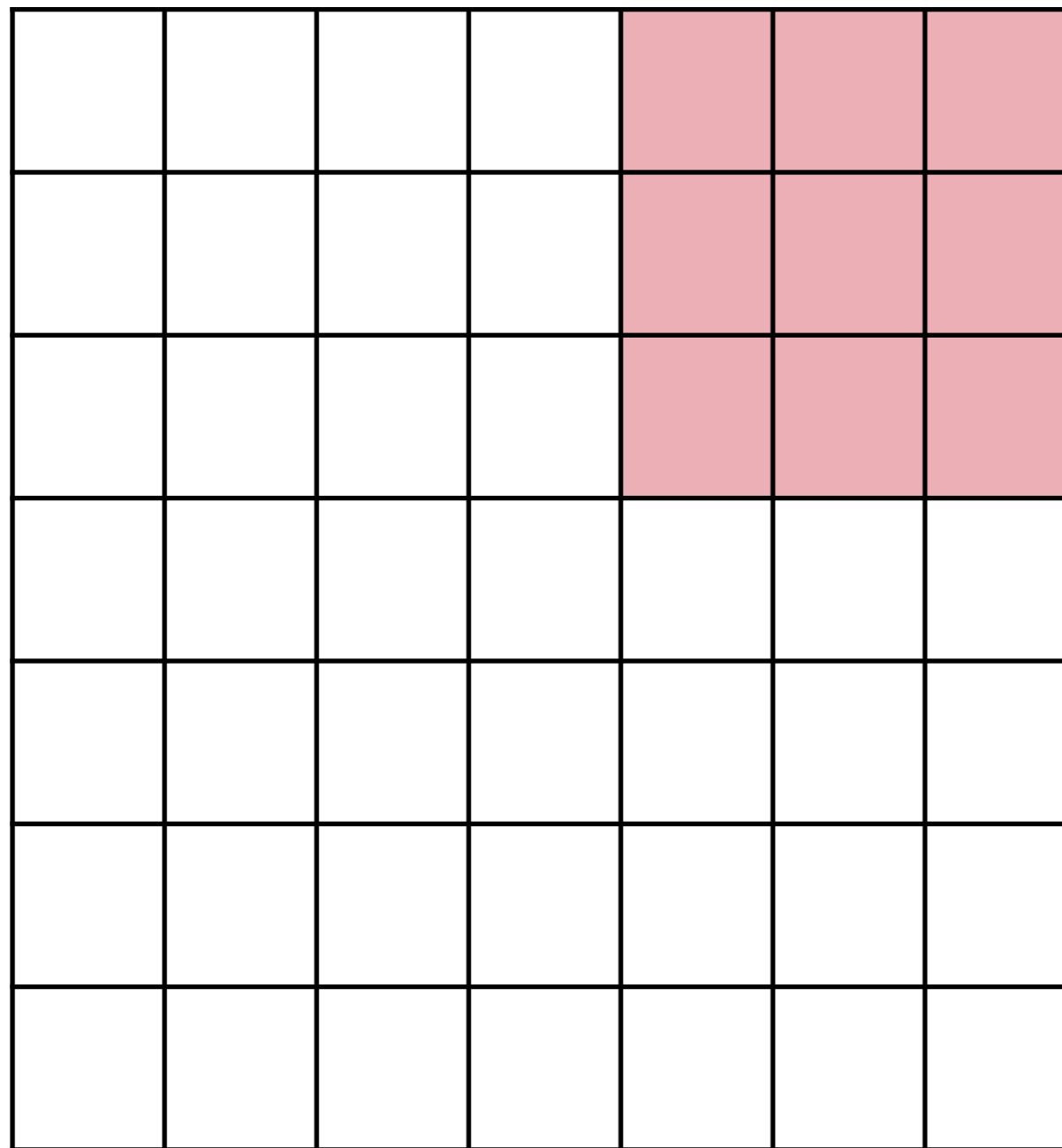
간격(**stride**)이 2인 컨볼루션

# 컨볼루션 예제 (3x3 필터)



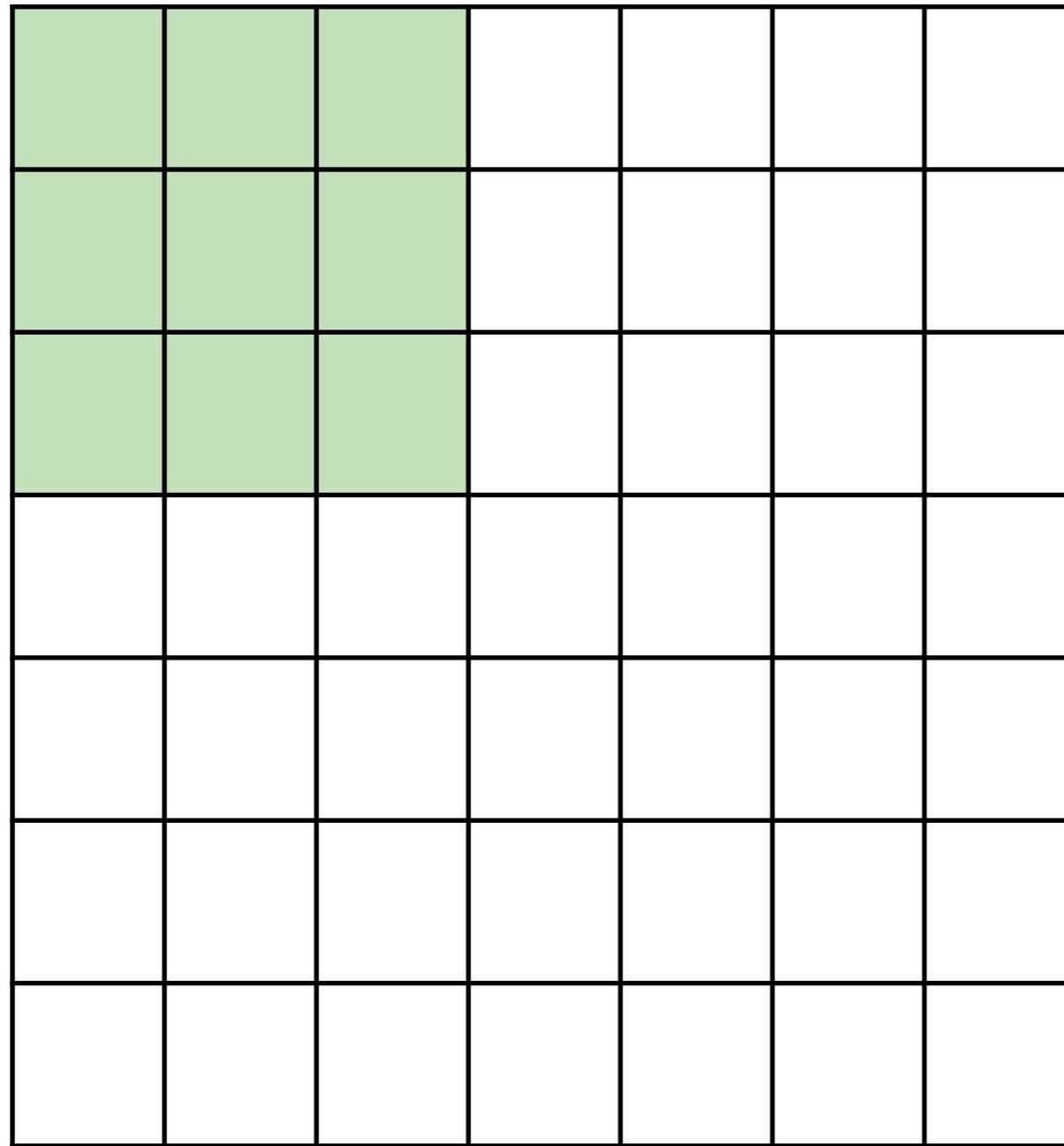
간격(stride)이 2인 컨볼루션

# 컨볼루션 예제 (3x3 필터)



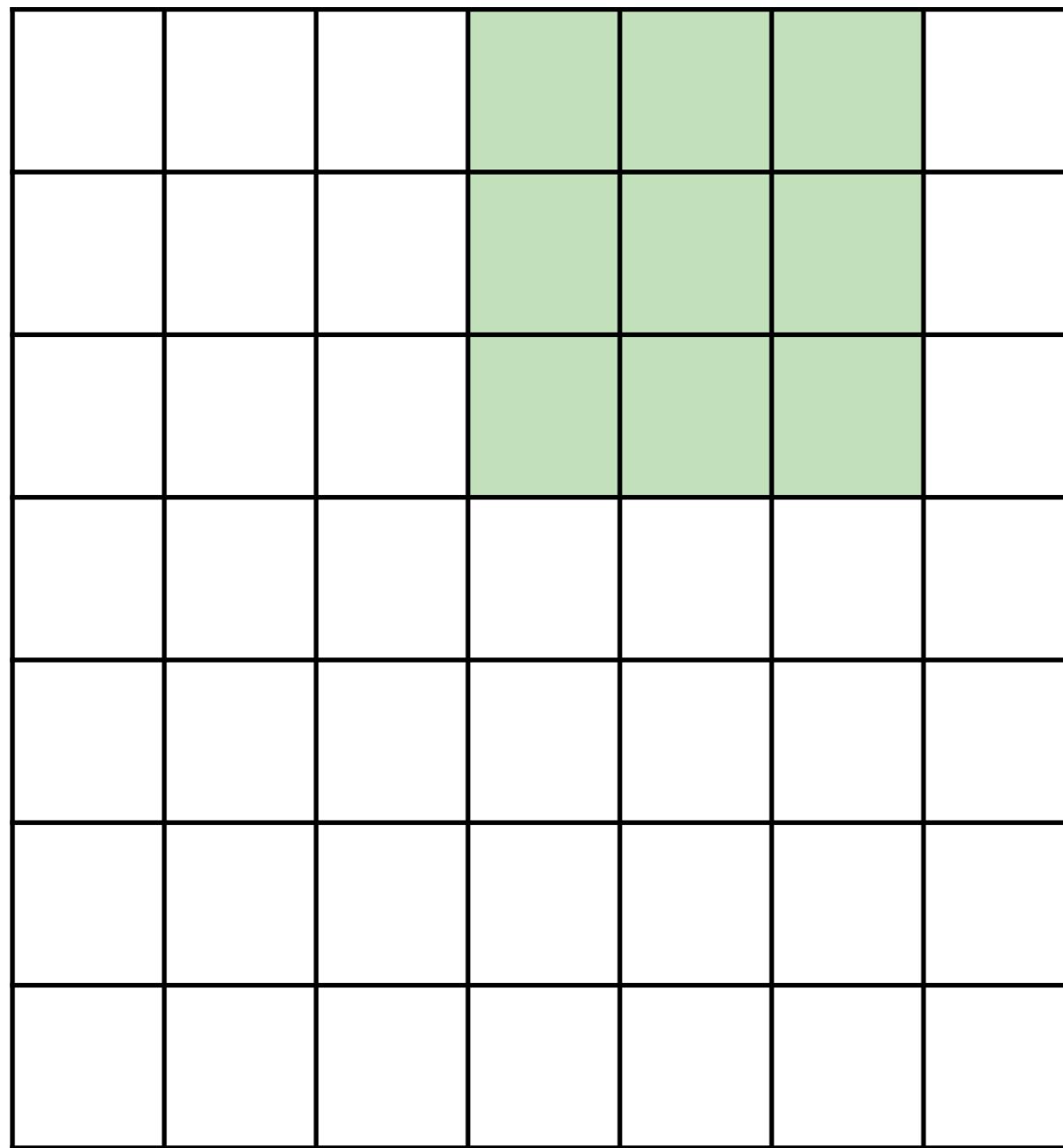
간격(stride)이 2인 컨볼루션  
=> 3x3 특징 맵

# 컨볼루션 예제 (3x3 필터)



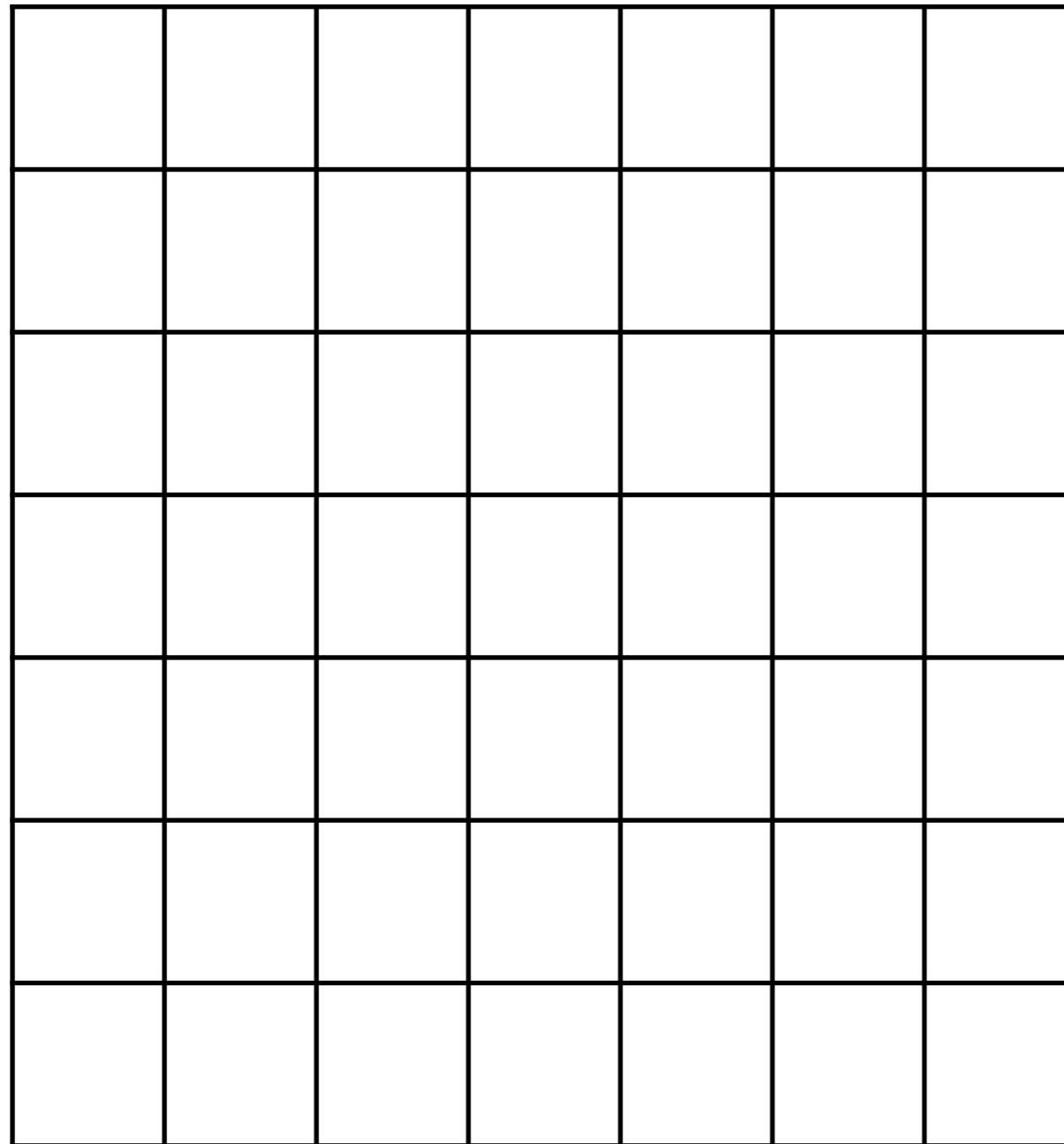
간격(stride)이 3인 컨볼루션

# 컨볼루션 예제 (3x3 필터)



간격(stride)이 3인 컨볼루션

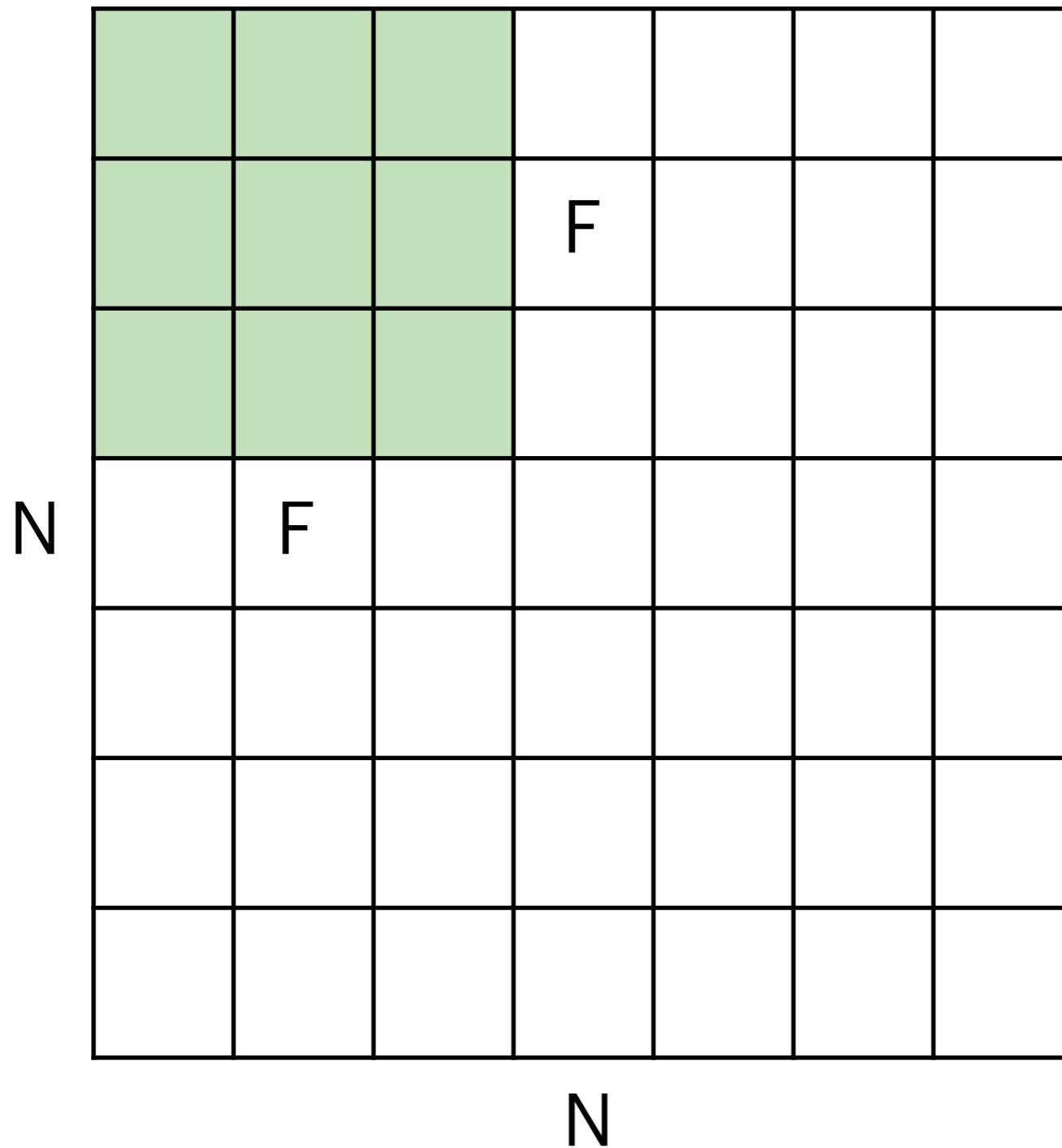
# 컨볼루션 예제 (3x3 필터)



7

간격(stride)이 3인 컨볼루션  
=> 에러!

# 컨볼루션 예제 (3x3 필터)



특징 맵 크기 공식:  
**(N - F) / stride + 1**

$$\text{stride } 1 \rightarrow (7-3)/1+1 = 5$$

$$\text{stride } 2 \rightarrow (7-3)/2+1 = 3$$

$$\text{stride } 3 \rightarrow (7-3)/3+1 = 2.3$$

페딩

- 컨볼루션의 문제점:
    - 이미지 (혹은 특징맵) 크기가 너무 빠르게 줄어듬
    - 그래서 레이어를 깊게 쌓을 수가 없음
    - (가끔) 입력과 출력의 크기를 똑같이 만들고 싶다
  - **패딩!**
    - 입력 맵 끝부분에 0으로 이루어진 패딩을 추가
    - 어차피 0이기 때문에 결과를 손상 시키지는 않음
    - 일반적으로 많이 쓰이는 방식

파딩

# 패딩

0	0	0	0	0	0	0	0	0
0			F					0
0								0
0	F							0
0								0
0								0
0								0
0								0
0	0	0	0	0	0	0	0	0

$N + (\text{pad} * 2)$

특징 맵 크기 공식:  
 $(N - F + \text{pad} * 2) / \text{stride} + 1$

# 패딩

0	0	0	0	0	0	0	0	0
0			F					0
0								0
0	F							0
0								0
0								0
0								0
0								0
0	0	0	0	0	0	0	0	0

N+(pad\*2)

특징 맵 크기 공식:  
 $(N - F + \text{pad}*2) / \text{stride} + 1$

3x3 stride 1, pad 1

$$\rightarrow (7-3+2)/1+1 = 7$$

5x5 stride 1, pad 2

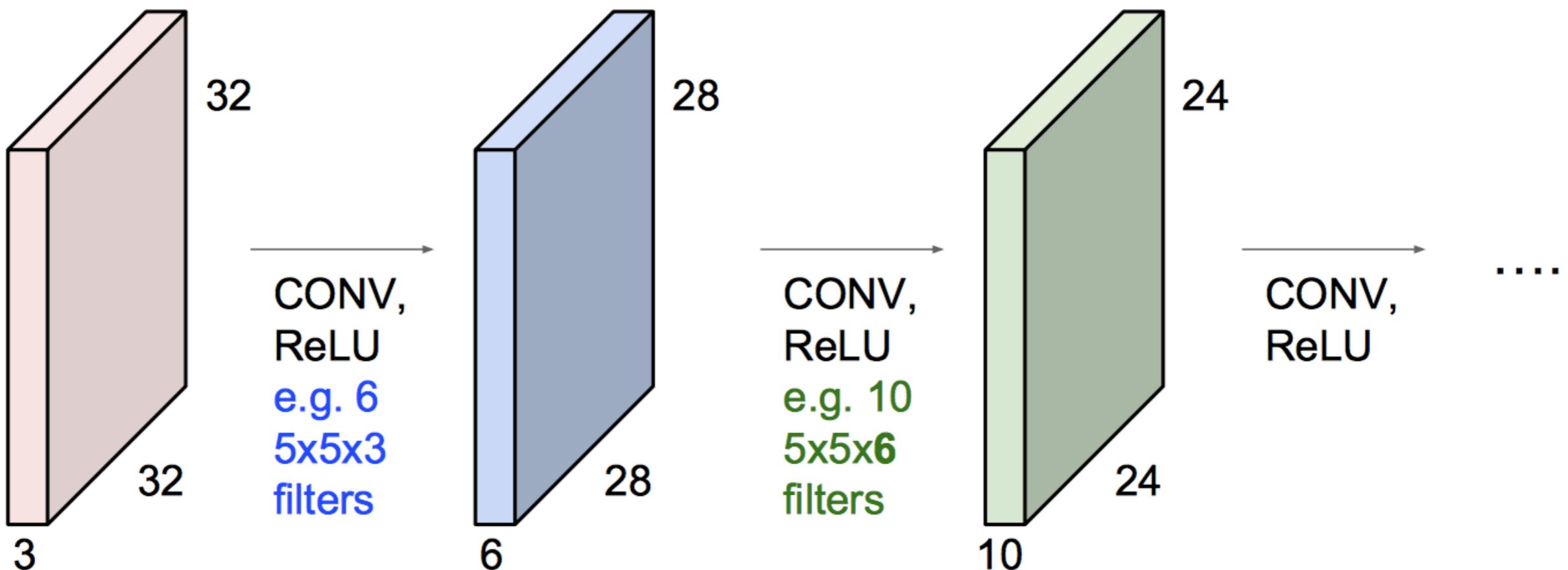
$$\rightarrow (7-5+4)/1+1 = 7$$

7x7 stride 1, pad 3

$$\rightarrow (7-7+6)/1+1 = 7$$

# 컨볼루션 뉴럴 네트워크

- 뉴럴 네트워크처럼 컨볼루션 레이어를 여러 층 쌓은 형태
  - Conv - ReLU - Conv - ReLU - ...



# 퀴즈 1

- 레이어의 입력이  $64 \times 64 \times 3$
- $5 \times 5$  필터 @10개 (stride 1, pad 2)
- 레이어 출력의 크기는?

# 퀴즈 1

- 레이어의 입력이  $64 \times 64 \times 3$
- $5 \times 5$  필터 @10개 (stride 1, pad 2)
- 레이어 출력의 크기는?  
 $=> (64-5+2*2)/1 + 1 = 64$   
 $=> \text{64x64x10}$

# 퀴즈 2

- 레이어의 입력이  $64 \times 64 \times 3$
- $5 \times 5$  필터 @10개 (stride 1, pad 2)
- 이 레이어의 파라미터 개수는?

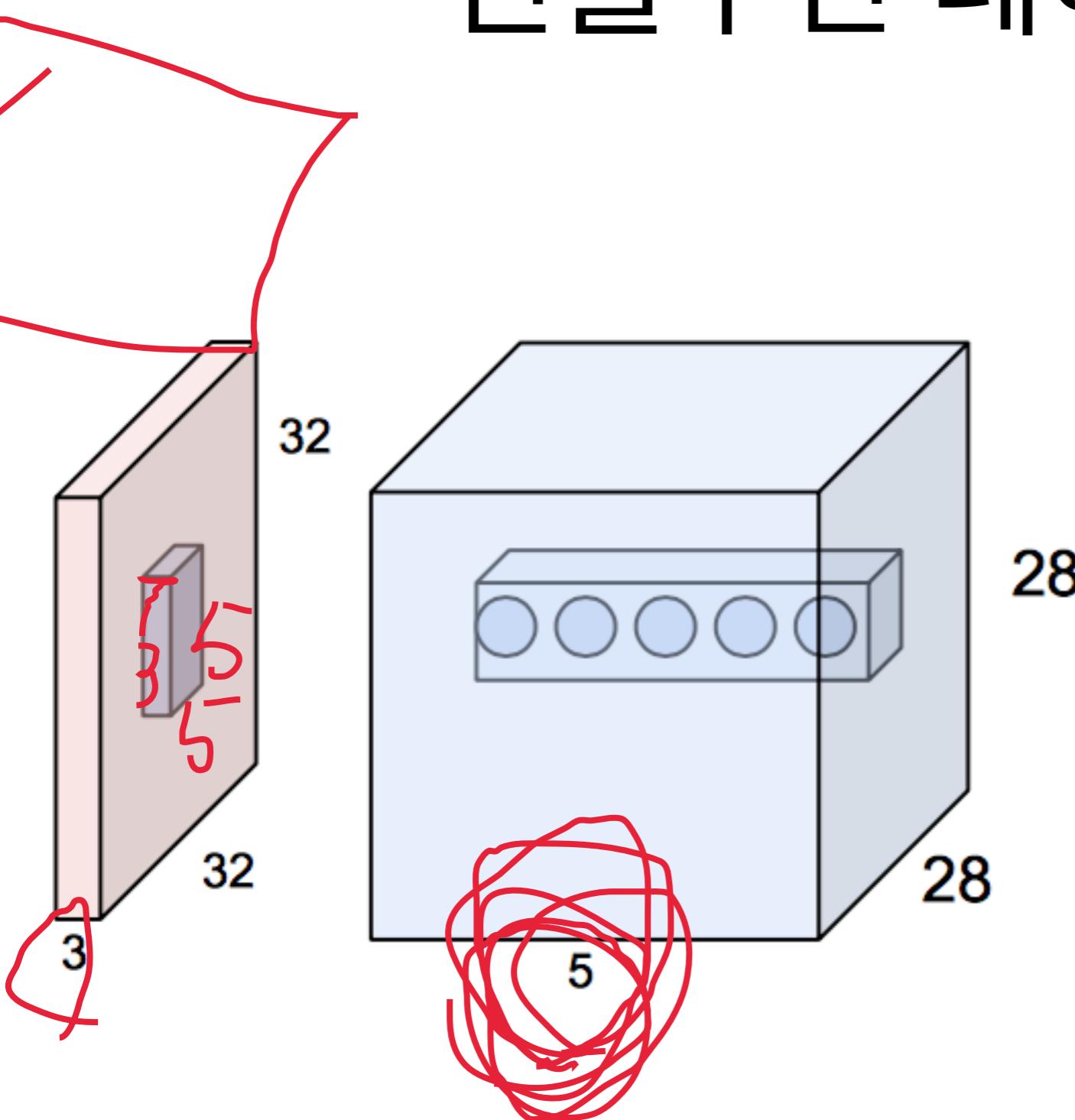
# 퀴즈 2

- 레이어의 입력이  $64 \times 64 \times 3$
- $5 \times 5$  필터 @10개 (stride 1, pad 2)
- 이 레이어의 파라미터 개수는?  
=> 각 필터가  $5 \times 5 \times 3 + 1$  개의 파라미터 (bias 1개)  
=>  $76 \times 10 = 760$

# 컨볼루션 레이어의 특징

- Local connectivity
  - 뉴런은 이전 레이어의 뉴런과 모두 연결되지 않고, 작은 영역만 연결됨
  - 작은 영역을 **receptive field** (수용장)라고 부름
  - 뇌에서 시각 신호를 처리할 때 하나의 뉴런이 시각 정보의 일정 부분만 처리하는 과정과 유사한 매커니즘
- Parameter sharing (tying)
  - 하나의 필터를 슬라이딩하면서 출력 특징맵을 생성
  - 이와 같은 파라미터 공유 방식으로 필요한 파라미터를 대폭 감소시키며, 효율적인 네트워크를 학습이 가능

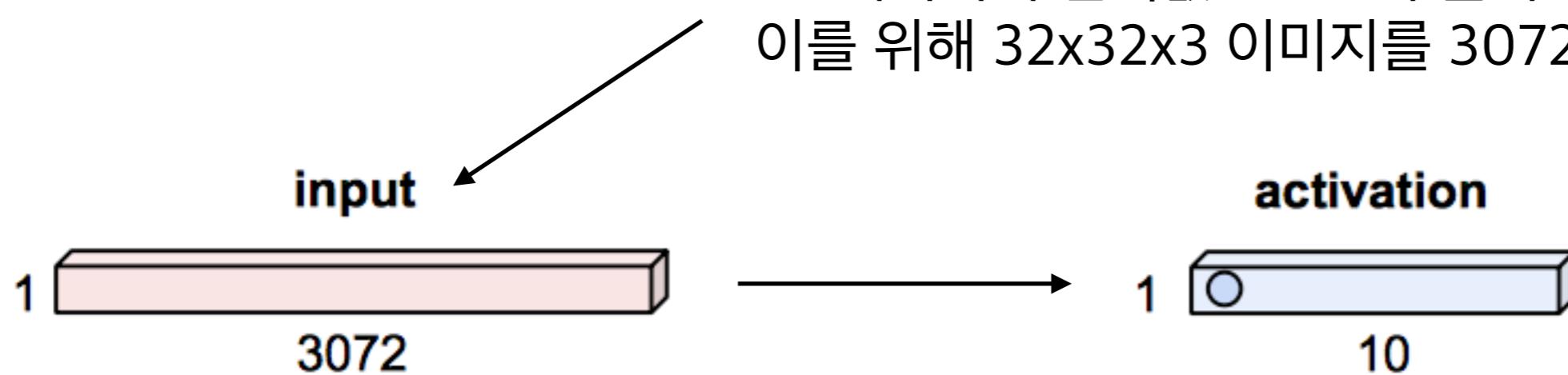
# 컨볼루션 레이어의 특징



- 필터 크기가  $5 \times 5 \Rightarrow$  수용장이  $5 \times 5$  크기
- 5개의 서로 다른 뉴런이 입력값의 같은 영역(수용장)과 연관되어 있음
- 파라미터의 수는  $5 \times 5 \times 3 \times 5 = 75 \times 5$

# FC 레이어

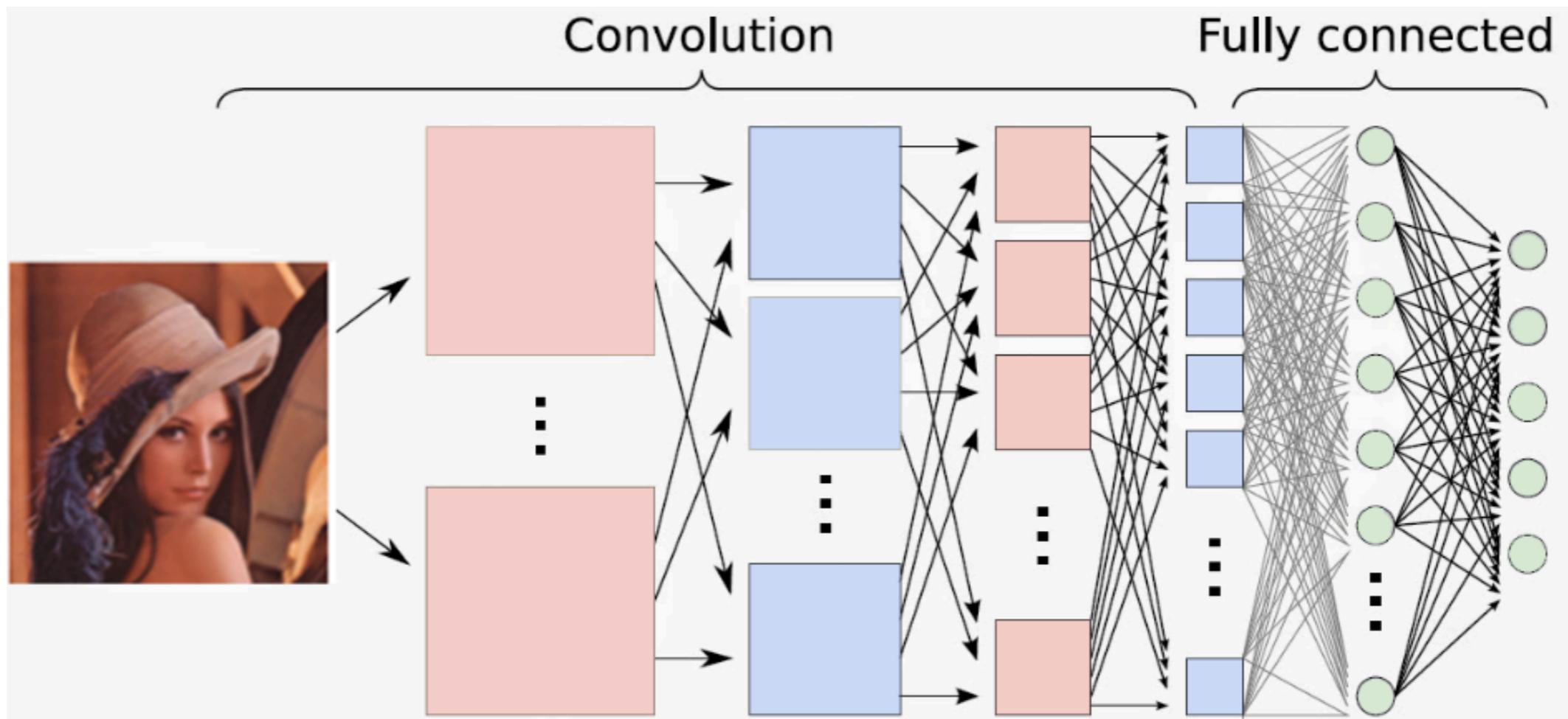
- FC 레이어의 입력값은 Nx1의 꼴이어야 함  
이를 위해 32x32x3 이미지를 3072x1로 변환



- 컨볼루션 레이어와 달리 뉴런이 입력값의 모든 뉴런과 연결됨
- 별도의 파라미터 공유가 이루어지지 않음
- 파라미터 수는  $3072 \times 10$

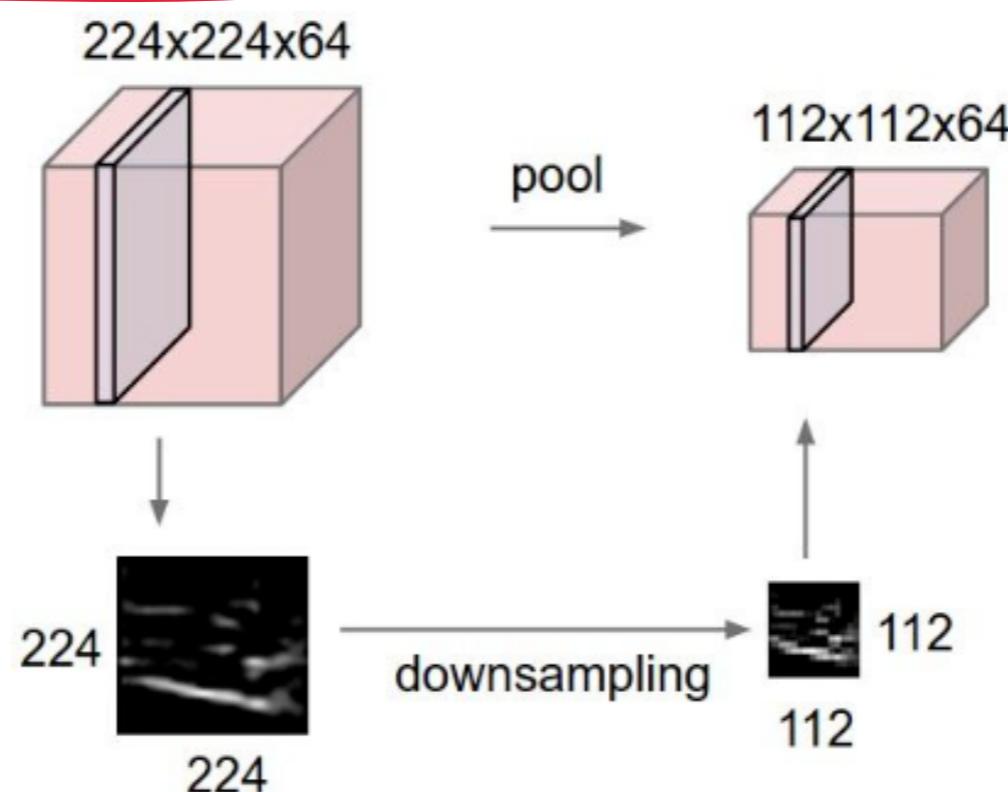
# 컨볼루션 뉴럴 네트워크

- 이미지를 처리할 때 많이 사용되는 네트워크
  - **컨볼루션**(convolution), **풀링**(pooling), **FC** 레이어로 구성
  - 인간의 시각 인지 작용을 모방한 디자인



# 풀링 레이어

- 입력 이미지(특징 맵)의 크기를 낮추는 레이어
  - 연산 속도와 메모리 요구량을 낮출 수 있음
  - 일반적으로 최대값을 이용하는 맥스 풀링 (Max pooling) 레이어를 사용함
  - 풀링 레이어는 **파라미터가 없음** (단순하게 최대, 평균 등을 계산하기 때문)



# 맥스 풀링 (Max pooling)

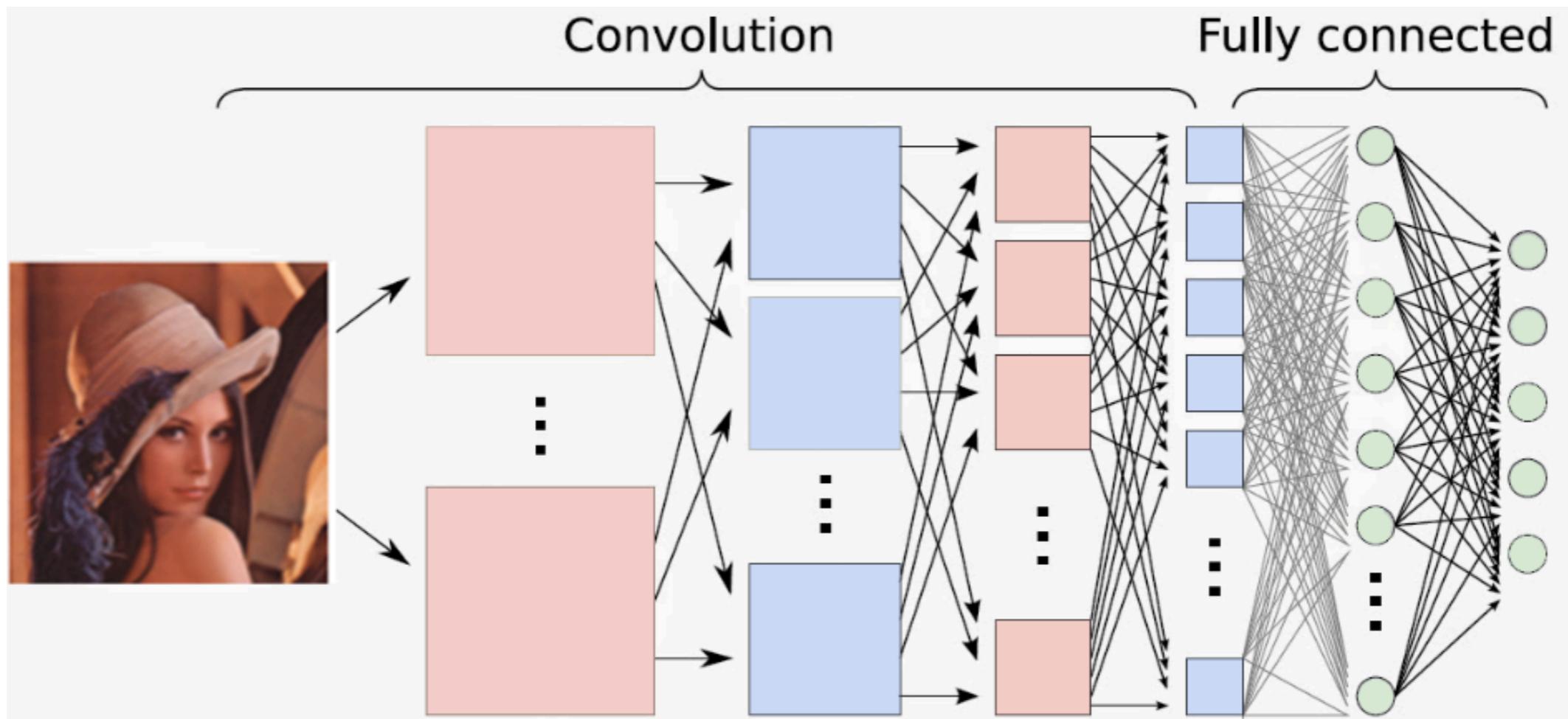
1	2	3	4
5	6	7	8
9	8	7	6
5	4	3	2

2x2, stride가 2인  
맥스 풀링

6	8
9	7

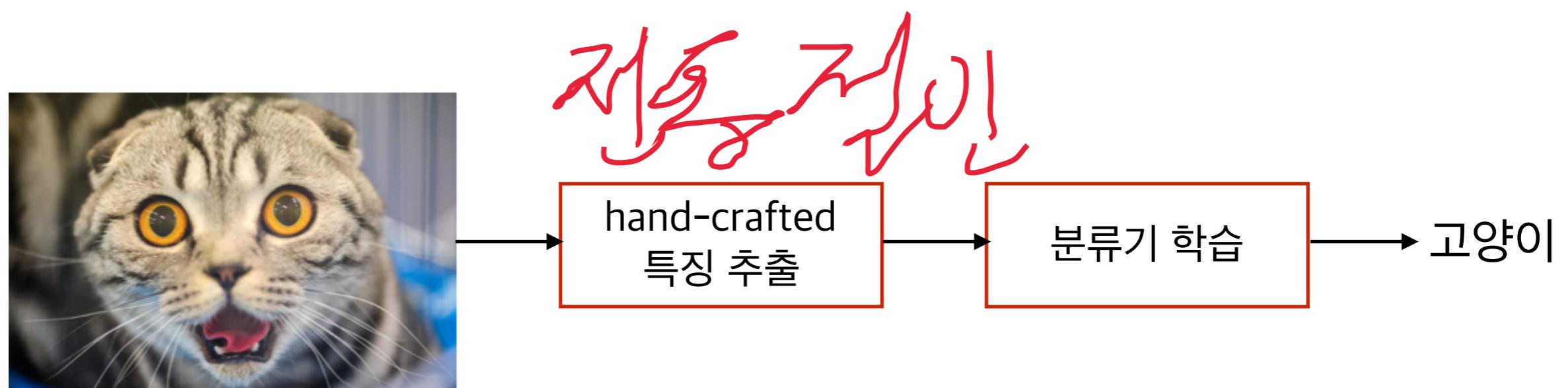
# 컨볼루션 뉴럴 네트워크

- 이미지를 처리할 때 많이 사용되는 네트워크
  - **컨볼루션**(convolution), **풀링**(pooling), **FC** 레이어로 구성
  - 인간의 시각 인지 작용을 모방한 디자인



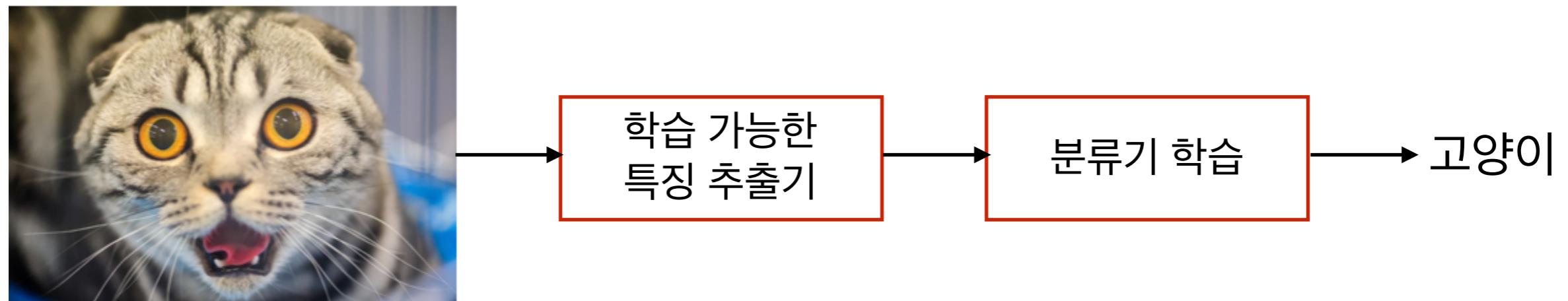
# CNN의 장점

- 전통적인 이미지 분류 프로세스
  - 특징 추출 모듈과 분류기 모듈이 별도로 존재
  - 특징 추출은 전통적인 컴퓨터비전 알고리즘으로 수행 (hand-crafted)
  - 추출된 특징을 입력으로 분류기를 학습하여 이미지 분류
  - 분류기를 학습 할 때 특징 추출부는 고정시키고 분류기만 학습

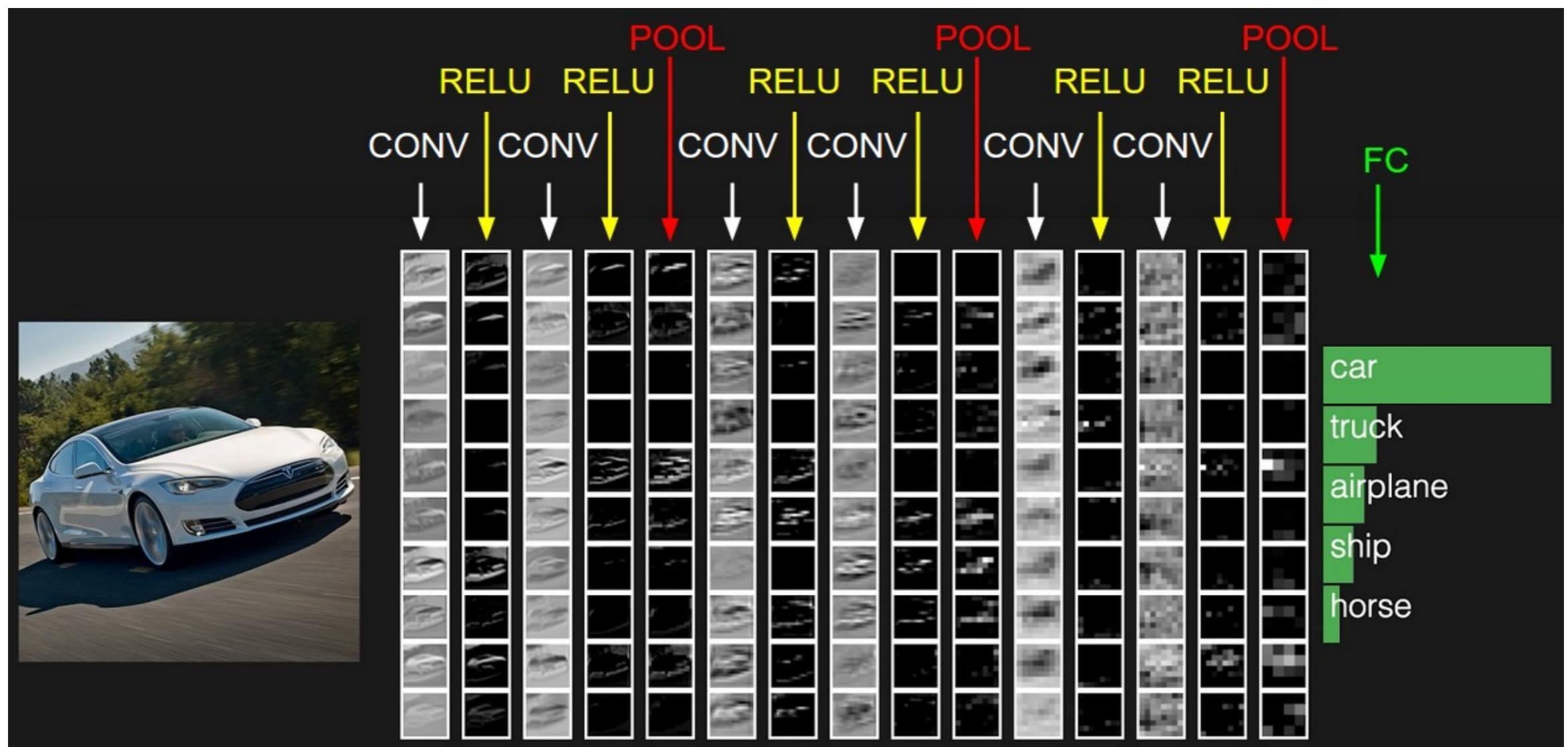


# CNN의 장점

- 딥러닝 기반 이미지 분류 프로세스
  - 특징 추출 모듈과 분류기를 하나의 거대한 뉴럴 네트워크로 구성
  - 특징 추출은 컨볼루션 레이어가, 분류기는 FC 레이어가 주로 담당
  - 특징 추출과 분류를 한번에 학습할 수 있음 (end-to-end model)
  - 높은 성능, 그러나 데이터가 많이 필요



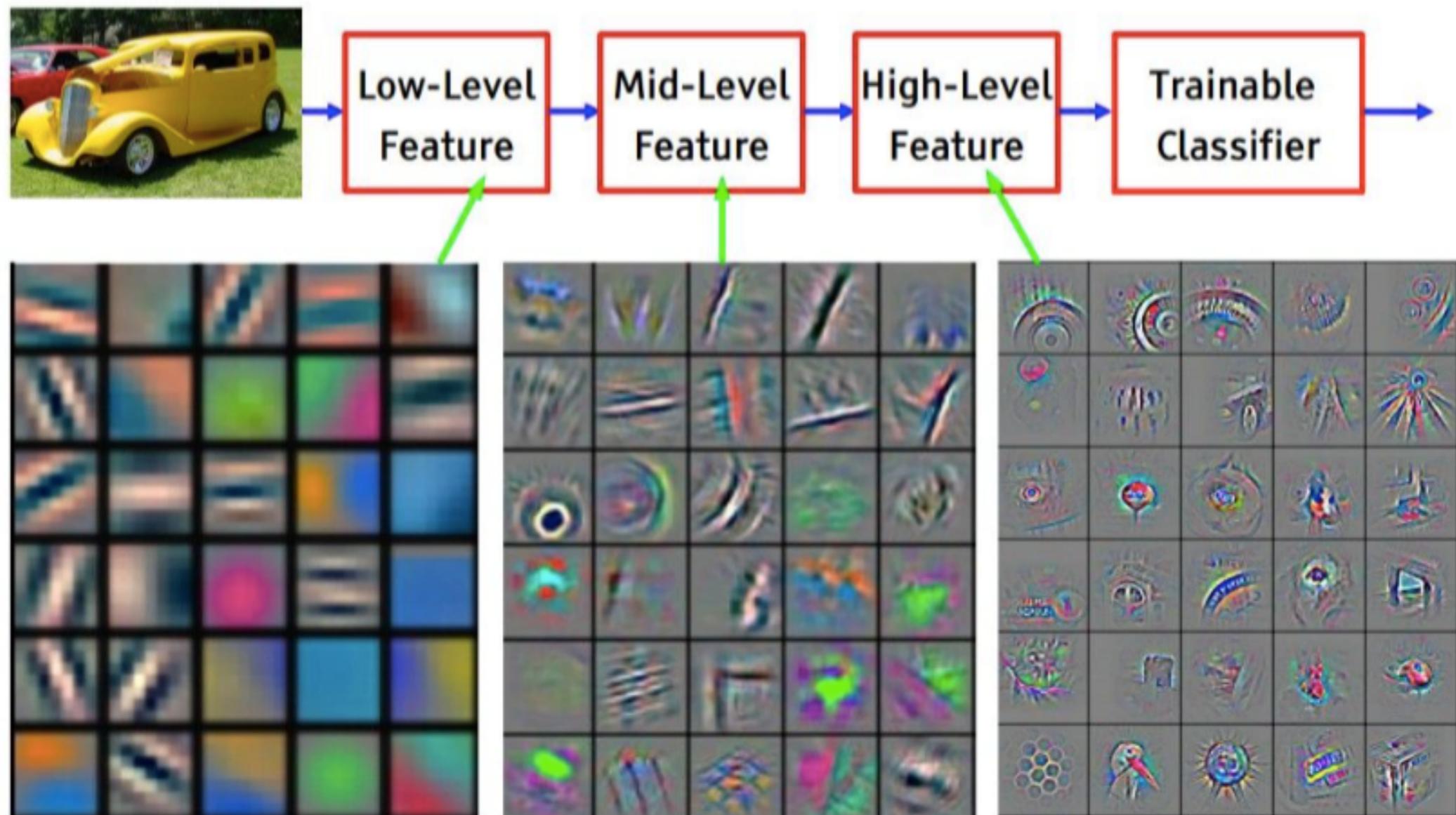
# CNN=특징추출+분류기



# Data-driven

- 전통적인 이미지 처리는 사람이 직접 특징 추출부를 디자인
  - 별도의 학습이 필요하지 않지만, 높은 성능을 위해 많은 노력이 필요
  - 인간의 시각 인지 메커니즘을 모방한 추출부도 존재 (가버 필터)
- 딥러닝(CNN)은 많은 데이터를 통해 특징 추출까지 학습
  - 하지만 학습된 추출부는 가버 필터와 비슷한 모양의 필터도 생성
  - -> **좋은 데이터가 많으면** 별도의 노력 없이 높은 성능을 낼 수 있다

# Data-driven



Feature visualization of convolutional net trained on ImageNet from [Zeiler & Fergus 2013]

# 딥러닝의 시작: 이미지넷 대회



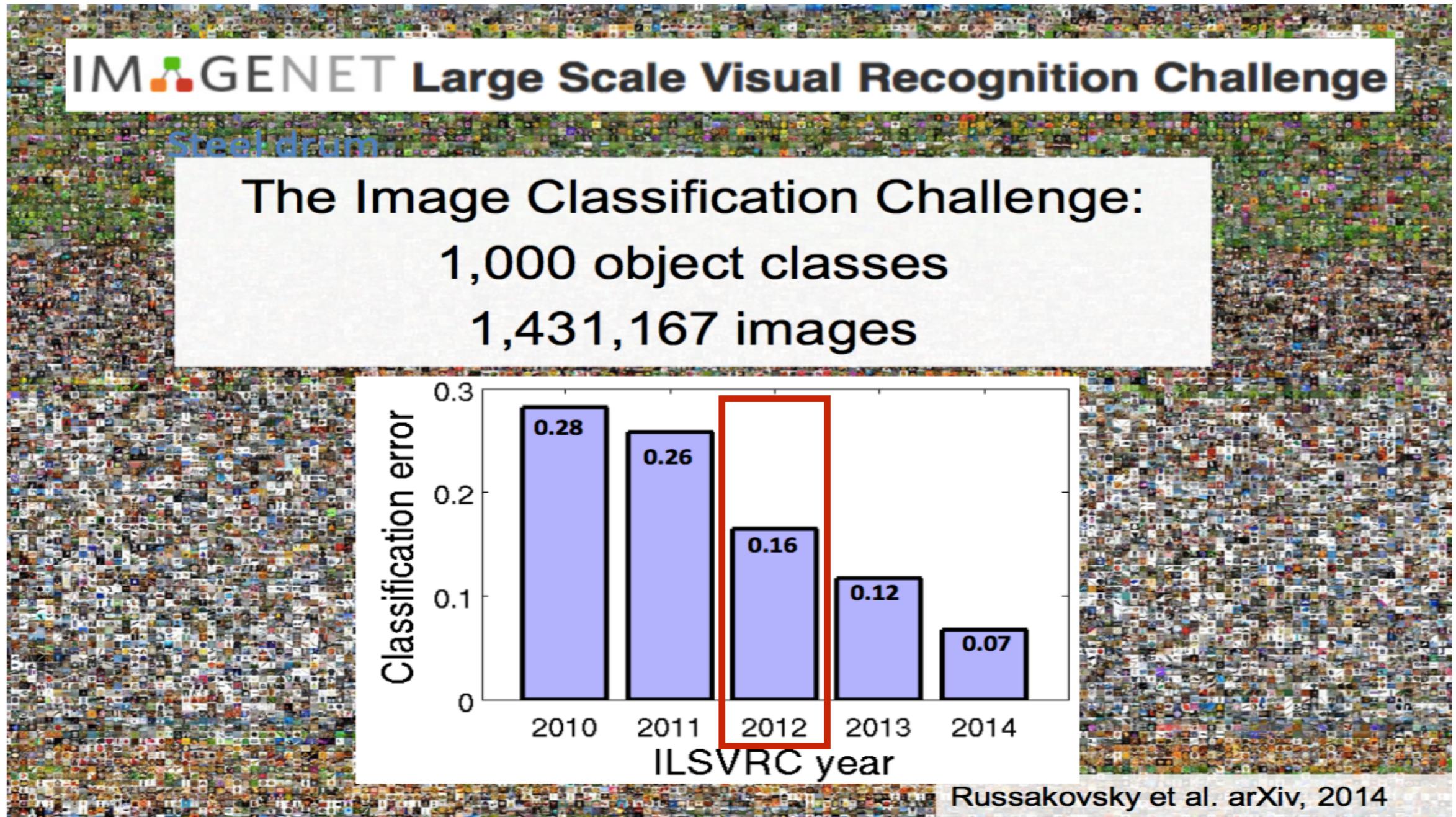
IM<sub>▲</sub>GENET

[www.image-net.org](http://www.image-net.org)

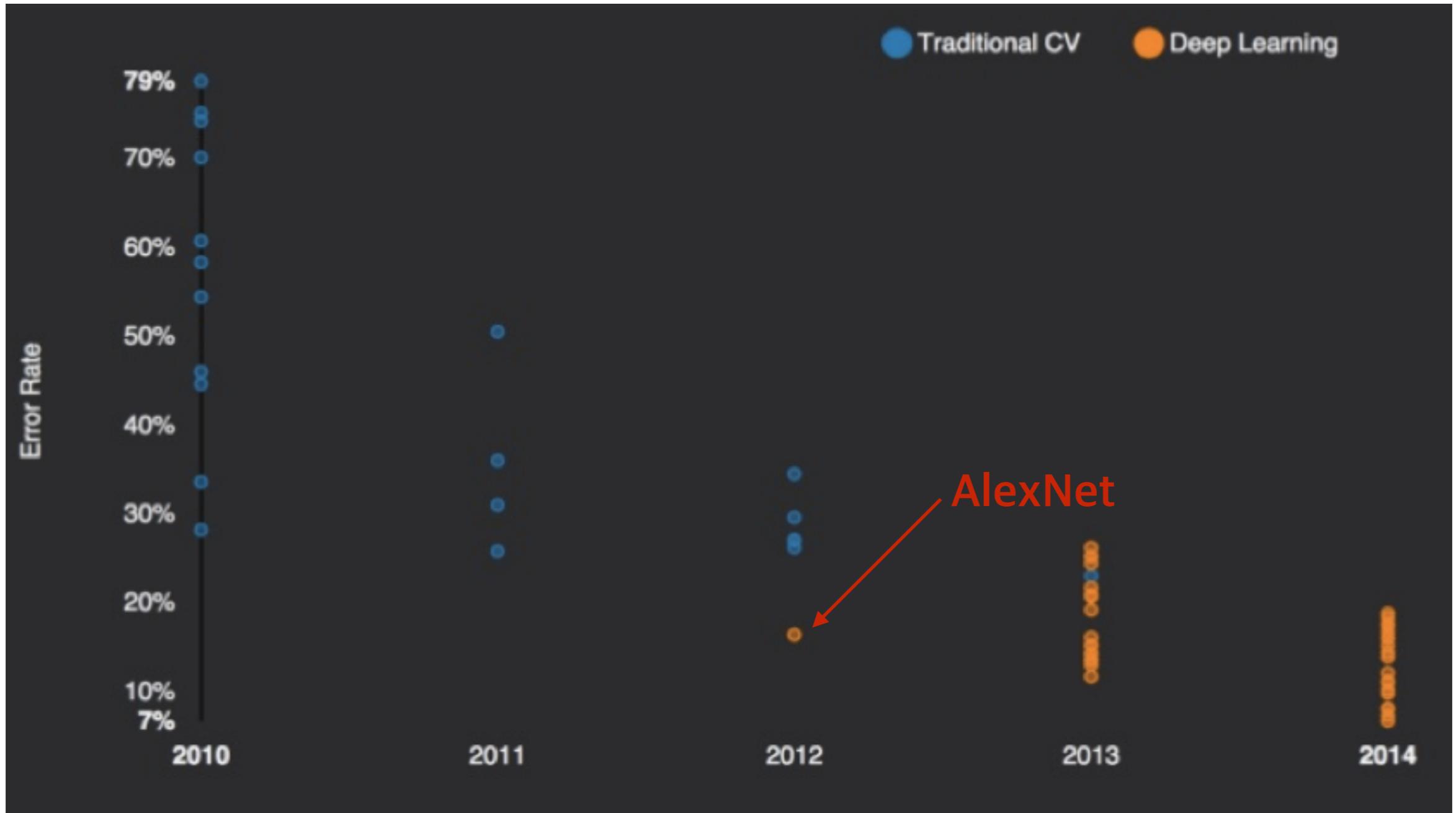
**22K** categories and **14M** images

- Animals
  - Bird
  - Fish
  - Mammal
  - Invertebrate
- Plants
  - Tree
  - Flower
  - Food
  - Materials
- Structures
  - Artifact
  - Tools
  - Appliances
  - Structures
- Person
- Scenes
  - Indoor
  - Geological Formations
  - Sport Activities

# 딥러닝의 시작: 이미지넷 대회



# 딥러닝의 시작: 이미지넷 대회

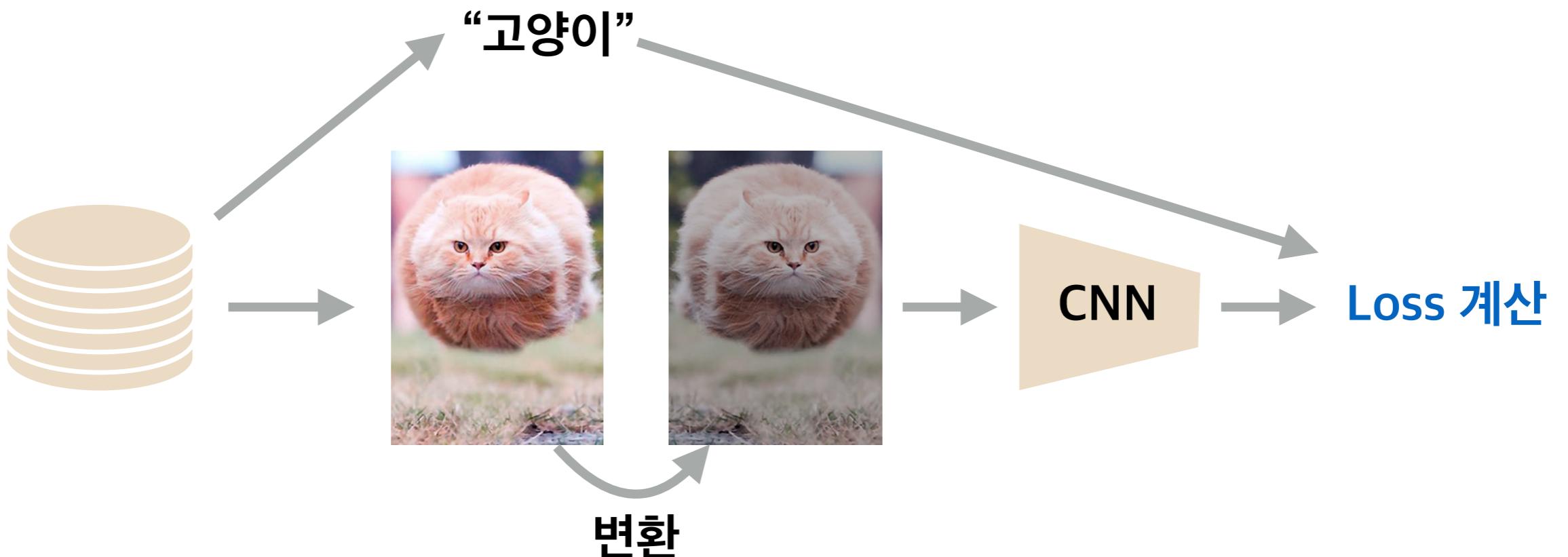


# CNNs in Practice

# Data augmentation

- 딥러닝은 네트워크가 커질 수록 더 많은 데이터를 필요로 함
  - 정해진 정답은 없지만 경험 상 10k~?
  - 데이터가 작으면 과적합의 가능성도 매우 커짐
- 데이터가 적으면 뺑튀기를 하자: **Data augmentation**
  - 입력 데이터를 변형하여 데이터의 개수를 늘리는 전처리 작업
  - 모델의 성능 향상
  - 이미지 인식 분야에서 보편적으로 사용

# Data augmentation



# 예시

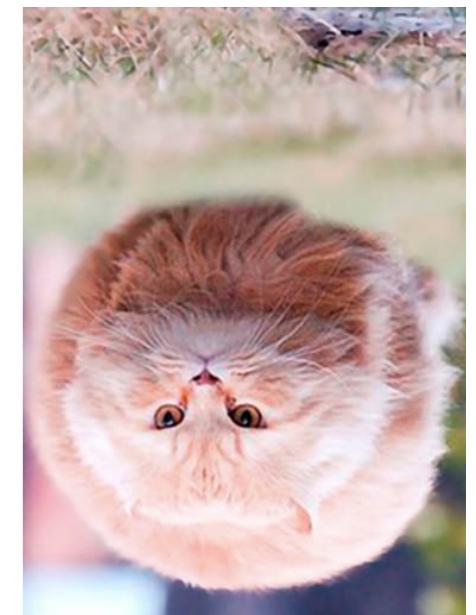


# 예시



- 좌우/상하 반전
- 노이즈
- 랜덤 크롭(crop)
- ...

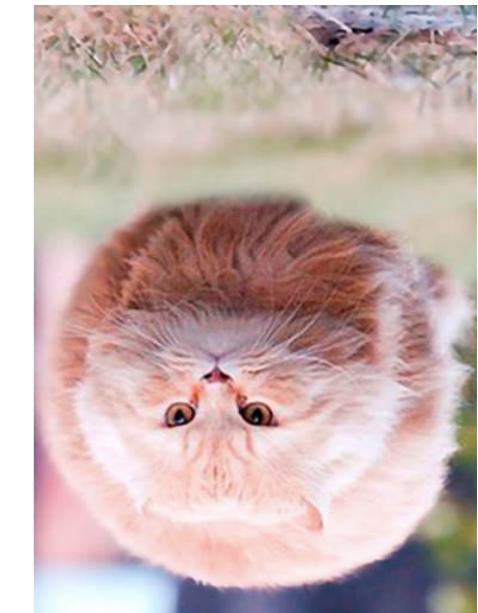
# 예시



- 좌우/상하 반전
- 노이즈
- 랜덤 크롭(crop)
- ...

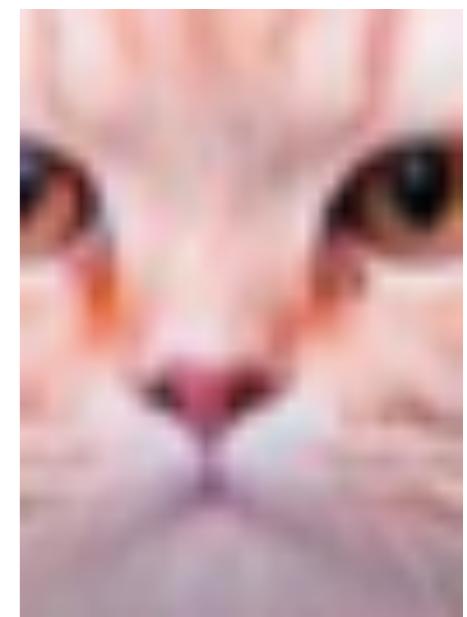
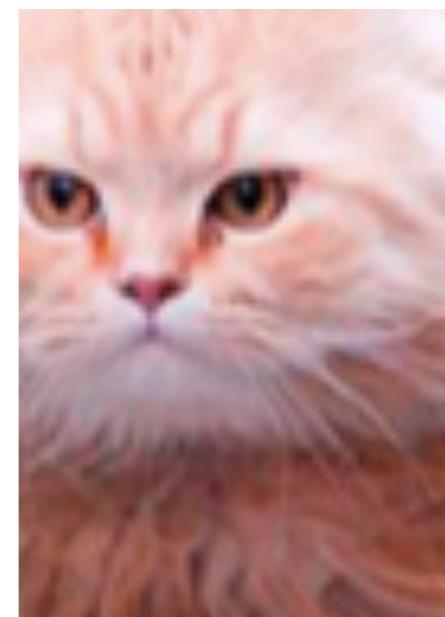
# 예시

- 좌우/상하 반전
- 노이즈
- 랜덤 크롭(crop)
- ...



# 예시

- 좌우/상하 반전
- 노이즈
- 랜덤 크롭(crop)
- ...



# 이외에도

- 적용 분야의 선행 지식을 바탕으로 **창의적인 변형**이 필요
  - 음성 인식: 사람의 목소리에 바람 소리 합성
  - 하지만 적용 분야에 적합한 data augmentation을 사용해야 함!
  - 차량 번호판 인식에 상하 반전??
  - MRI 이미지에 노이즈??

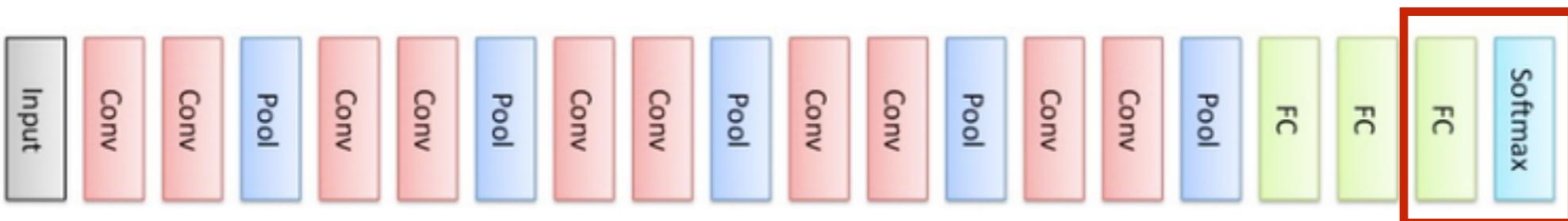
# 전이 학습 (Transfer learning)

- 딥러닝이 “항상” 데이터가 많이 필요한 것은 아님!
  - 처음부터 모델을 학습시키는 것은 많은 데이터가 필요
  - 내가 적용하려는 도메인과 **비슷한 도메인**의 데이터가 있다면 활용 가능
  - 예시: 강아지 분류 데이터 -> 고양이 분류 데이터
- **전이 학습**
  - 비슷한 도메인으로 학습한 모델을 적용 도메인에 맞게 수정한 뒤,  
**수정한 레이어만 별도로 학습** (추가로 다른 레이어도 학습 가능)
- 다른 도메인에 **사전 학습**된 모델을 이용한 **fine-tuning**
  - 랜덤한 초기값부터 학습하지 않고, 미리 학습된 모델을 불러와서 학습
  - 이미 어느정도 학습이 된 상태이기 때문에 수렴이 빠름

# 전이 학습 예시

## 1. 타겟 도메인이 기존 데이터와 비슷하고 타겟 데이터가 적다

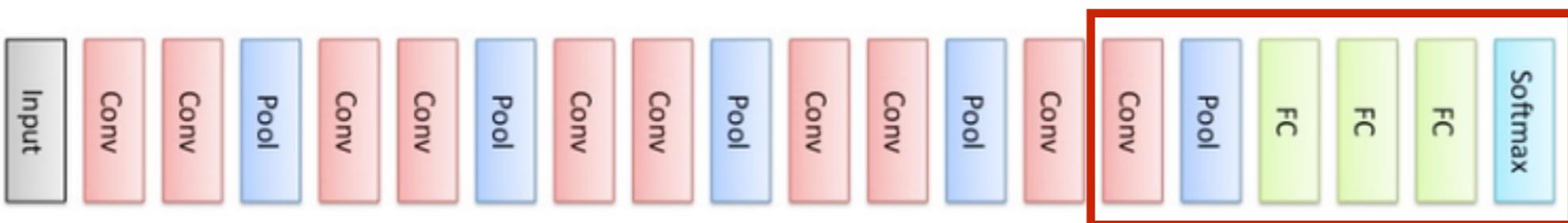
- 마지막 FC 레이어를 타겟 도메인에 맞게 수정하고 이 레이어만 따로 학습
- 예를 들어 기존 데이터가 클래스 1000개, 타겟 도메인이 10개라면;
- 마지막 FC 레이어의 출력값 개수를 10개로 변경



# 전이 학습 예시

## 2. 타겟 도메인이 기존 데이터와 비슷하고 타겟 데이터가 많다

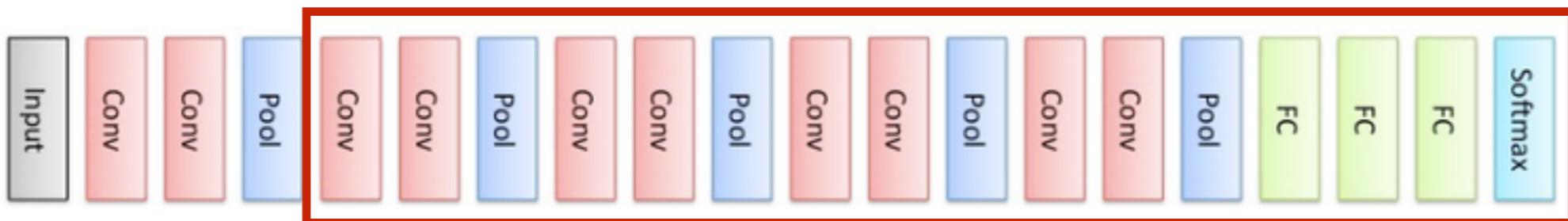
- 수정한 마지막 레이어와 추가로 몇 개 레이어를 fine-tune
- 데이터의 개수에 따라 추가로 더 학습할 레이어를 정하기
- 데이터가 매우 많다면 전체 레이어를 학습 할 수 있음 (학습 시간은..)



# 전이 학습 예시

## 3. 타겟 도메인이 기존 데이터와 다르고 타겟 데이터가 많다

- 이 경우 하위 레이어도 학습해야 할 수 있음
- 많은 레이어를 fine-tuning 하기

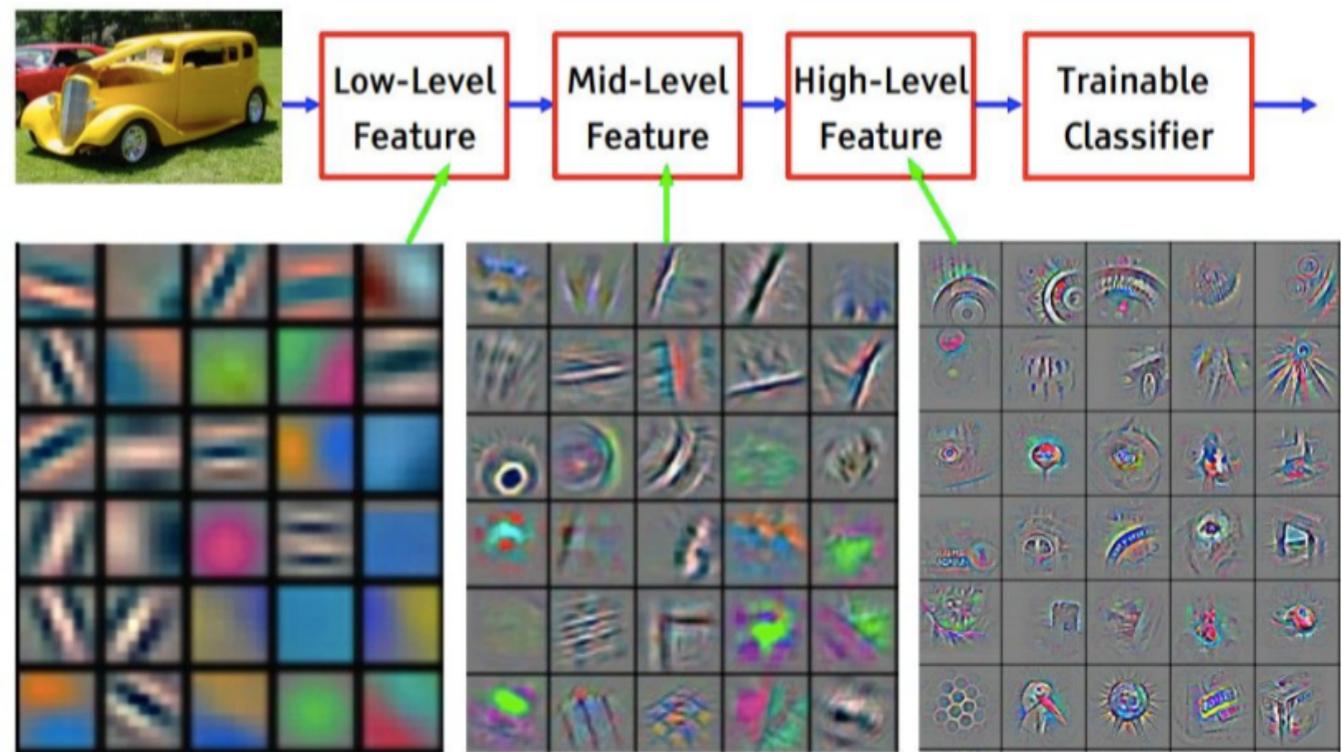


# 전이 학습 예시

4. 타겟 도메인이 기존 데이터와 다르고 타겟 데이터가 적다
  - 꼭 딥러닝을 써야하나요..?

# 전이 학습 정리

- 도메인이 비슷하면 (특히 이미지) 모든 레이어를 처음부터 학습 할 필요 없음!
- 타겟 도메인과 기존 도메인 데이터의 특성에 따라 얼마나 fine-tuning 할지 고려하기



Feature visualization of convolutional net trained on ImageNet from [Zeiler & Fergus 2013]

# 사전 학습 모델 (pre-trained)

- 내가 적용하려는 분야에 해당하는 모델이 **기존에 학습**되어 있다면? -> 전이 학습할 필요 없이 바로 사용 가능
  - 많은 연구자들이 인터넷에 코드와 사전 학습 모델을 올려 놓음 (깃허브)
  - 이미지를 특징 추출할 때 VGGNet, ResNet과 같은 모델을 많이 사용
  - TensorFlow도 최근 이미지 물체 인식 코드 공개!
  - <https://github.com/tensorflow/models>

Model	TF-Slim File	Checkpoint	Top-1 Accuracy	Top-5 Accuracy
Inception V1	Code	inception_v1_2016_08_28.tar.gz	69.8	89.6
Inception V2	Code	inception_v2_2016_08_28.tar.gz	73.9	91.8
Inception V3	Code	inception_v3_2016_08_28.tar.gz	78.0	93.9
Inception V4	Code	inception_v4_2016_09_09.tar.gz	80.2	95.2
Inception-ResNet-v2	Code	inception_resnet_v2_2016_08_30.tar.gz	80.4	95.3
ResNet V1 50	Code	resnet_v1_50_2016_08_28.tar.gz	75.2	92.2
ResNet V1 101	Code	resnet_v1_101_2016_08_28.tar.gz	76.4	92.9
ResNet V1 152	Code	resnet_v1_152_2016_08_28.tar.gz	76.8	93.2
ResNet V2 50^	Code	resnet_v2_50_2017_04_14.tar.gz	75.6	92.8
ResNet V2 101^	Code	resnet_v2_101_2017_04_14.tar.gz	77.0	93.7
ResNet V2 152^	Code	resnet_v2_152_2017_04_14.tar.gz	77.8	94.1
ResNet V2 200	Code	TBA	79.9*	95.2*
VGG 16	Code	vgg_16_2016_08_28.tar.gz	71.5	89.8
VGG 19	Code	vgg_19_2016_08_28.tar.gz	71.1	89.8
MobileNet_v1_1.0_224	Code	mobilenet_v1_1.0_224_2017_06_14.tar.gz	70.7	89.5
MobileNet_v1_0.50_160	Code	mobilenet_v1_0.50_160_2017_06_14.tar.gz	59.9	82.5
MobileNet_v1_0.25_128	Code	mobilenet_v1_0.25_128_2017_06_14.tar.gz	41.3	66.2

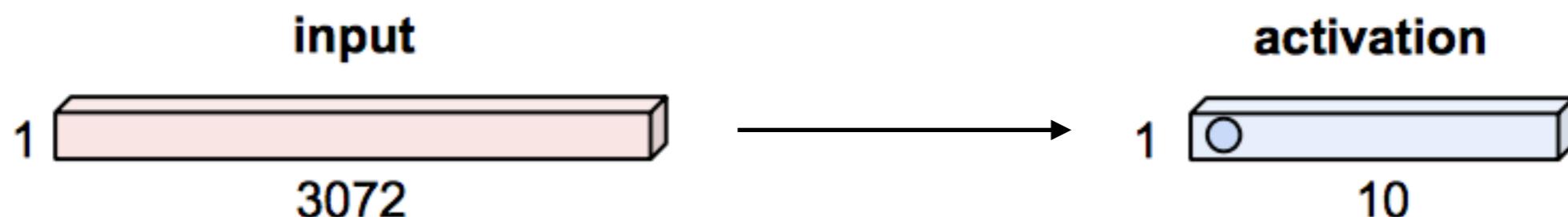
# Tensorflow Object Detection API

Creating accurate machine learning models capable of localizing and identifying multiple objects in a single image remains a core challenge in computer vision. The TensorFlow Object Detection API is an open source framework built on top of TensorFlow that makes it easy to construct, train and deploy object detection models. At Google we've certainly found this codebase to be useful for our computer vision needs, and we hope that you will as well.



# 입력 이미지 크기 문제

- 일반적인 CNN은 입력 이미지 **크기가 고정**되어야 함
  - FC 레이어는 고정된 크기의 입력만 받을 수 있기 때문
  - 그래서 대부분의 딥러닝 모델은 입력 이미지를 맞추는 전처리 작업 수행
  - 예) 이미지 랜덤 크롭, 리사이징 등등



# 이미지 전처리

## 1. 평균 이미지를 구해서 정규화

- 학습 데이터에서 평균 이미지를 구하고, 입력 이미지에서 평균 이미지를 빼서 정규화 (분산은 따로 고려하지 않음)
- AlexNet에서 사용한 방법

## 2. 채널별로 평균값을 구한 뒤 정규화

- R,G,B 각 채널별로 평균값을 구하고, 입력 이미지에서 각 평균별로 정규화하는 방법
- VGGNet 이후에 대세적으로 사용되는 정규화
- 이미지넷에서 얻은 평균 값  $R = 123.68$   $G = 116.78$   $B = 103.94$  를 사용하는 것이 일반적

# Summary

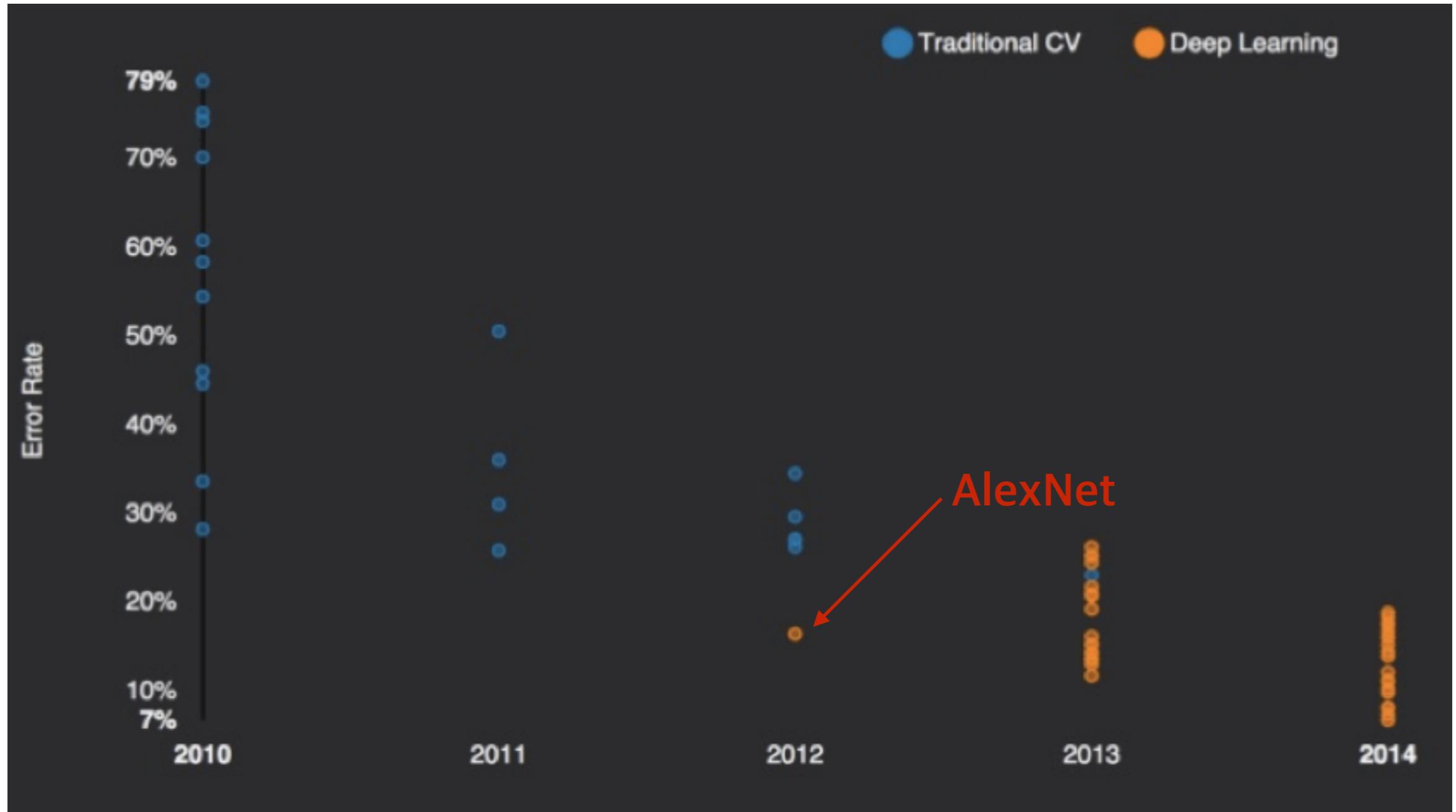
- CNN은 컨볼루션 / 풀링 / FC 레이어로 구성됨
- 컨볼루션 레이어는 이미지의 특징을 뽑는 레이어
- 풀링 레이어는 이미지 크기를 줄여 계산량을 줄이는 레이어
- FC 레이어는 뽑혀진 특징을 분류하는 레이어 (MLP와 동일)
- 전이 학습!!
- 데이터가 적을 경우 augmentation (많아도 하면 좋음)

# CNN 주요 모델

# CNN 주요 모델

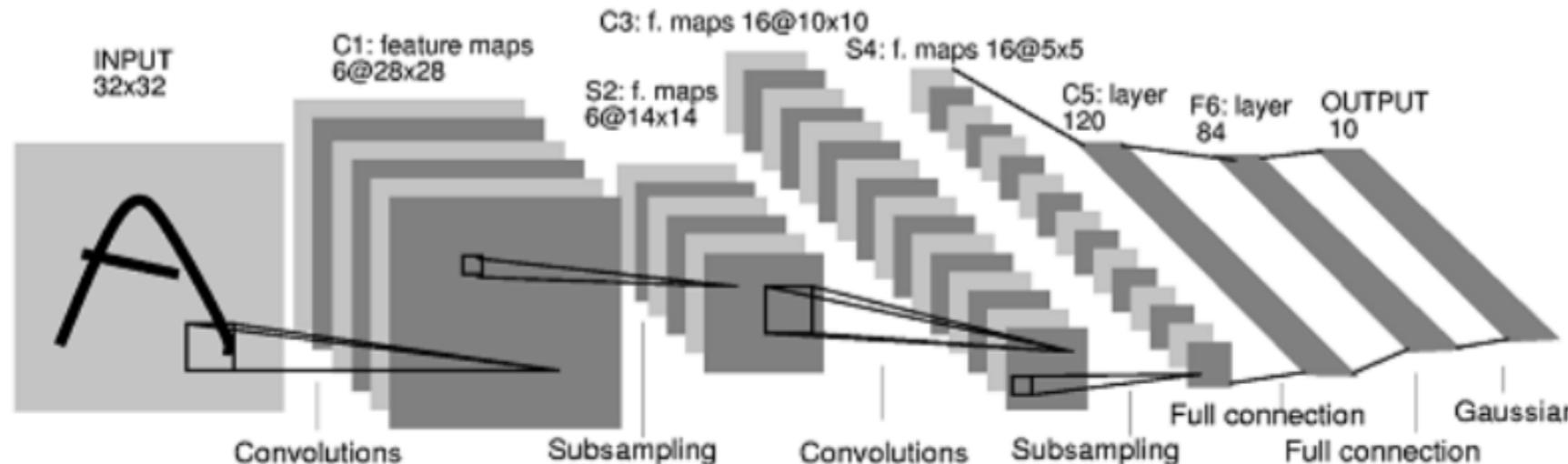
- AlexNet (2012)
- VGGNet (2014)
- GoogLeNet (2014)
- ResNet (2015)

# 이미지넷 대회



1998

LeCun et al.



# of transistors



$10^6$

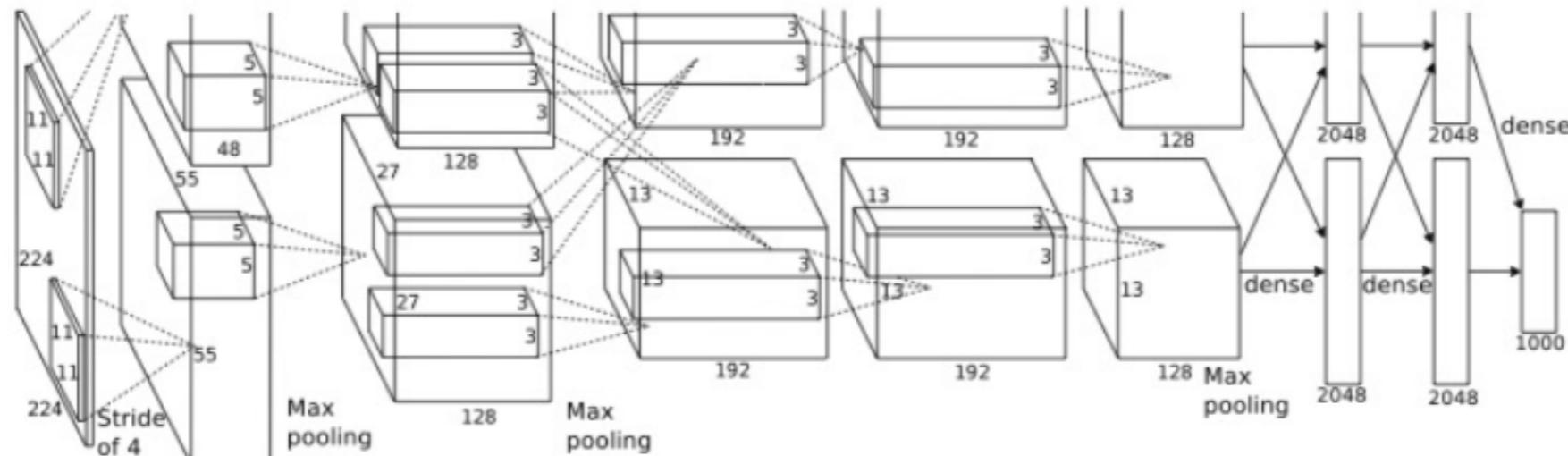
pentium® II

# of pixels used in training

$10^7$  NIST

2012

Krizhevsky  
et al.



# of transistors GPUs



$10^9$



# of pixels used in training

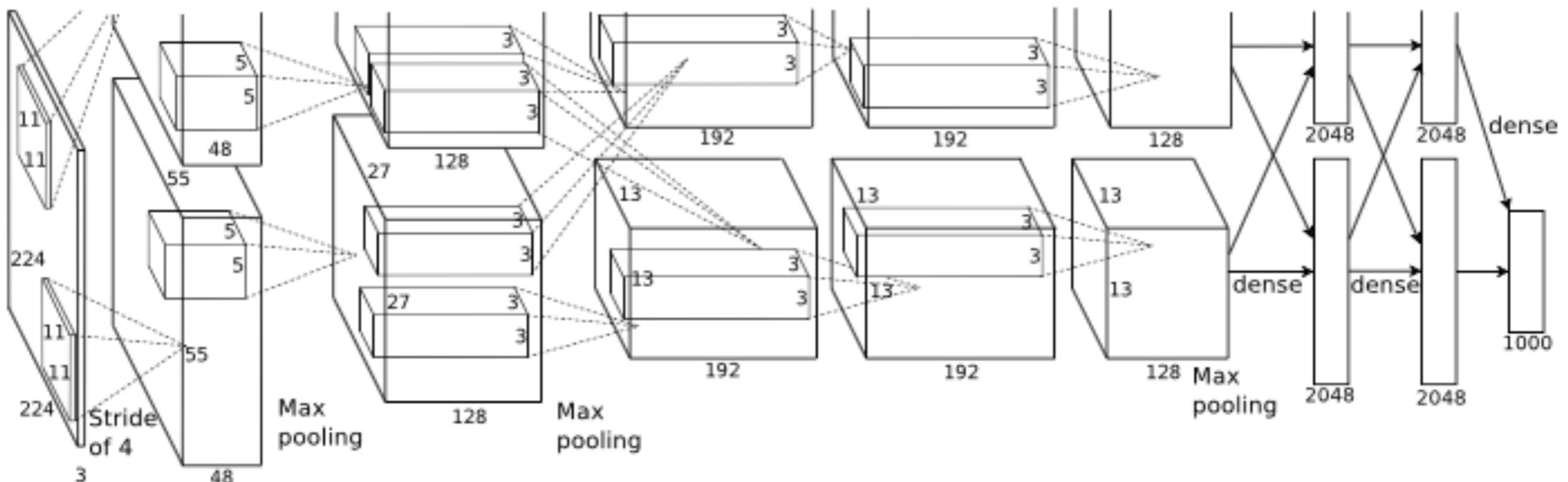
$10^{14}$  IMAGENET

# AlexNet (2012)

- 이미지넷 대회에서 선보인 첫 딥러닝 모델
  - 전통적인 컴퓨터 비전 방법을 매우 큰 격차로 따돌림
- AlexNet의 주요 포인트
  - Dropout과 ReLU (dropout은 FC 레이어에만 사용)
  - Data augmentation 사용
  - SGD+모멘텀 사용, weight decay(L2 reg), 양상을
  - Local Response Normalization: 정규화 기법, 지금은 안쓰이는 방법
  - 파라미터 수: **60M**

# AlexNet 아키텍쳐

- 5개 컨볼루션 레이어 + FC 레이어 2개 + 분류 레이어
  - 3x3 MAX stride 2인 풀링 <- 풀링 영역이 겹치면 성능이 더 좋다고 함
- 입력 이미지는 224x224x3으로 고정
  - 일반적인 컨볼루션 네트워크는 입력 이미지 사이즈가 고정되어야 함



# Data augmentation

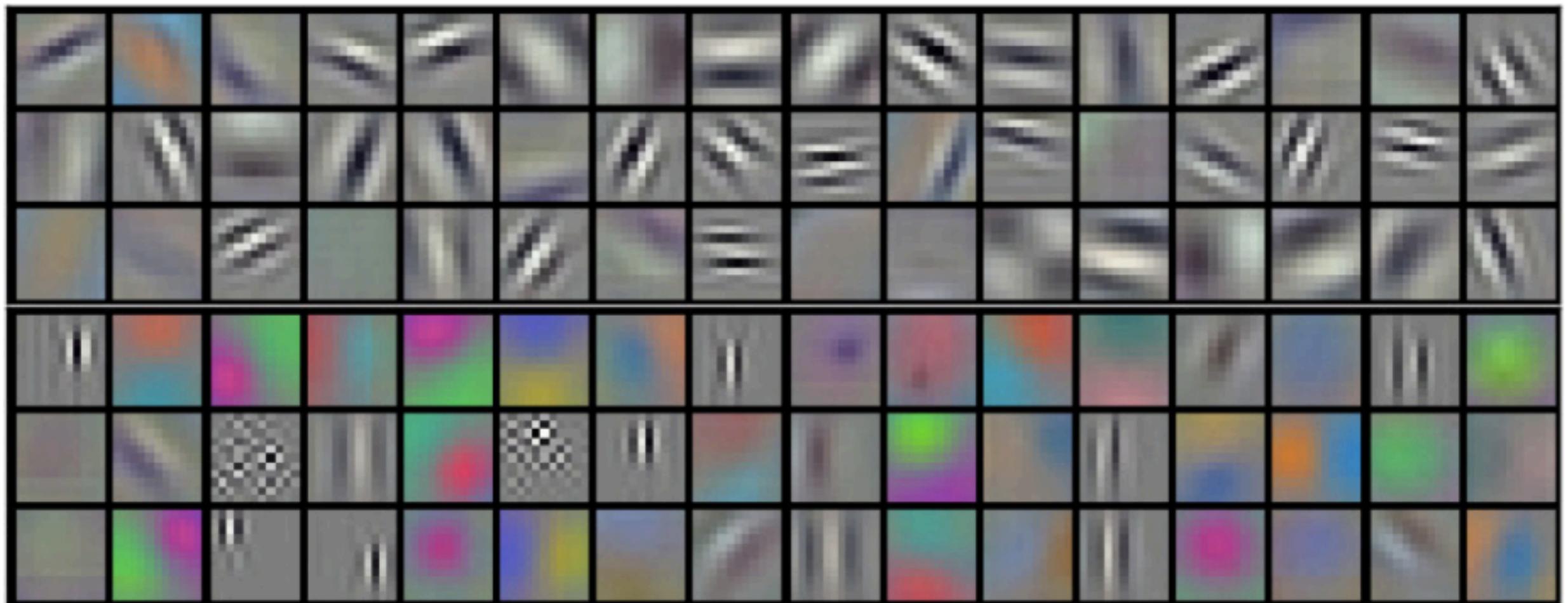
- 과적합을 줄이기 위해 data augmentation을 수행
  - CPU에서 data augmentation, GPU에서 메인 연산
- 학습과 테스트 단계의 augmentation이 다름
  - **학습**: 이미지를 256x256 크기로 리사이즈 -> 224x224 크기로 랜덤 크롭
  - 이미지당 2048개의 경우의 수, 상하 대칭된 이미지도 똑같이 수행
  - **테스트**: 랜덤 크롭 대신, 상하좌우/중앙 5개의 이미지 x 2 (상하 대칭), 총 10개의 이미지를 추출하여 **softmax의 평균**을 통해 최종 예측
- 추가로 PCA를 통해 테스트 이미지의 RGB 채널 강도를 변화

# 기타 세팅

- SGD+모멘텀, 배치 사이즈는 128
  - 모멘텀은 0.9, weight decay는 0.0005
- weight 초기화는 가우시안  $\text{std}=0.01$
- Dropout 확률은 0.5
- 초기 Learning rate는 0.01
  - validation error의 성능 향상이 멈추면 10배 감소시킴
- 7개의 CNN을 **양상블**: 18.2% -> 15.4%로 최종 성능 향상

# AlexNet 필터 시각화

- 첫 컨볼루션 레이어의 필터를 (11x11x3) 시각화
  - 에지, 텍스처, 컬러 등의 정보를 잘 추출하는 것을 확인 가능



# VGGNet (2014)

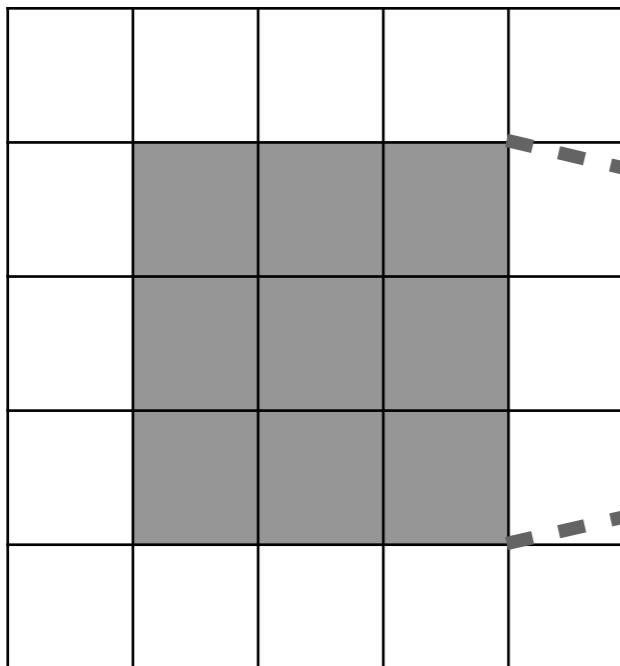
- 2014년 이미지넷 대회 2위
  - 그러나 매우 간단한 아키텍쳐로 지금까지 널리 사용되는 모델
- VGGNet의 주요 포인트
  - 3x3 컨볼루션 사용
  - 매우 깊은 모델 (19 레이어)
  - 강한 data augmentation
  - 정말 많은 파라미터 (140M)

# 3x3 컨볼루션

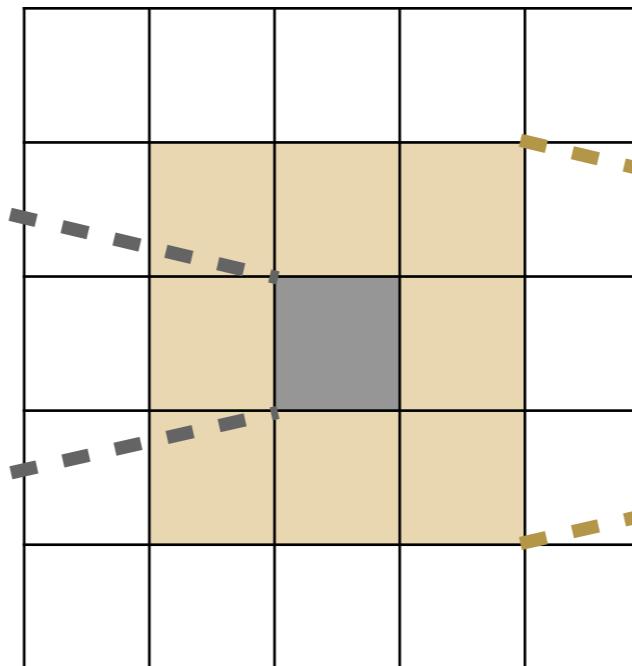
- 기존 네트워크는 대체로 큰 필터를 사용 ( $11 \times 11$ ,  $7 \times 7$ ..)
  - VGGNet은 모든 컨볼루션 레이어에서 **3x3**, pad 1 stride 1을 적용
  - 이미지 다운샘플은 풀링 레이어에서만 수행
- 왜 작은 컨볼루션을 사용할까?
  - 작은 컨볼루션을 쌓으면 큰 크기의 컨볼루션과 **동일한 성능**
  - 같은 성능, 그러나 **파라미터 수가 적고, 더 많은 활성 함수를 사용 가능**

# 3x3 컨볼루션

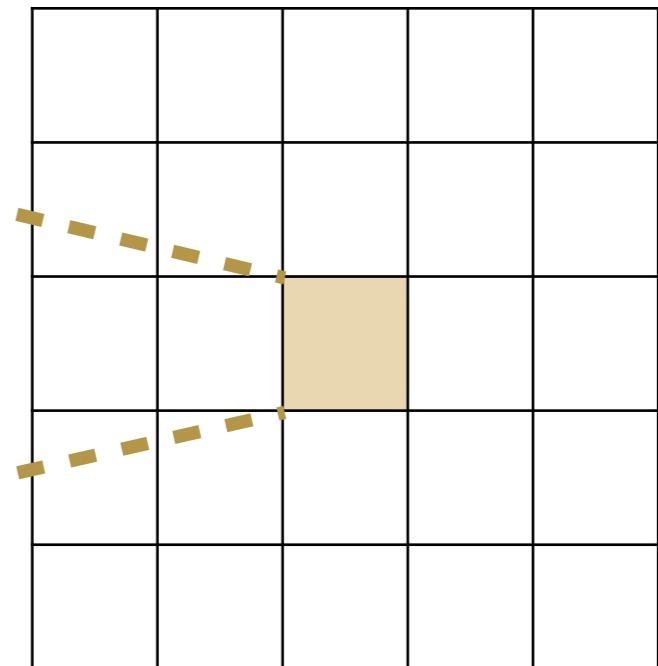
- 레이어2의 뉴런은 입력 이미지의 어느 영역을 커버하나?
  - Receptive field의 관점에서 3x3을 여러 층 쌓으면 큰 크기의 필터와 동일
  - 예) 3x3 2층은 5x5와, 3층은 7x7와 같은 receptive field
  - 그래서 작은 크기의 필터를 여러층 쌓으면 큰 필터와 비슷한 **표현력**을 가짐



입력



레이어 1



레이어 2

# 3x3 컨볼루션

- 작은 크기의 컨볼루션을 여러 층 쌓으면 생기는 장점
  - 파라미터 수 감소**: 5x5 2층 vs 3x3 4층  $\rightarrow$  3x3이 더 적은 파라미터
  - 비슷하게 필요한 연산량도 줄어드는 장점
  - 활성 함수를 더 많이** 사용하여 네트워크가 더 discriminative 해짐
- 그래서, 대부분의 네트워크가 3x3 컨볼루션을 기본으로 사용하는 추세
  - 3x3 컨볼루션이 최소 단위 컨볼루션이기 때문
  - 1x1 컨볼루션은 receptive field가 없기 때문에 사용 불가 (다른 용도로 사용)

# 아키텍처

- 2x2 Max 풀링 (stride=2) 사용
- 나머지는 AlexNet과 동일
- LPN은 성능 향상이 크지 않고 연산을 많이 잡아먹기 때문에 제외

ConvNet Configuration					
A	A-LRN	B	C	D	E
11 weight layers	11 weight layers	13 weight layers	16 weight layers	16 weight layers	19 weight layers
input (224 × 224 RGB image)					
conv3-64 LRN	conv3-64 <b>conv3-64</b>	conv3-64 <b>conv3-64</b>	conv3-64 conv3-64	conv3-64 conv3-64	conv3-64 conv3-64
maxpool					
conv3-128	conv3-128 <b>conv3-128</b>	conv3-128 <b>conv3-128</b>	conv3-128 conv3-128	conv3-128 conv3-128	conv3-128 conv3-128
maxpool					
conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256 <b>conv1-256</b>	conv3-256 conv3-256 <b>conv3-256</b>	conv3-256 conv3-256 <b>conv3-256</b>
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 <b>conv1-512</b>	conv3-512 conv3-512 <b>conv3-512</b>	conv3-512 conv3-512 <b>conv3-512</b>
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 <b>conv1-512</b>	conv3-512 conv3-512 <b>conv3-512</b>	conv3-512 conv3-512 <b>conv3-512</b>
maxpool					
FC-4096					
FC-4096					
FC-1000					
soft-max					

# 학습 디테일

- SGD+모멘텀, 배치 사이즈는 256
  - 모멘텀은 0.9, weight decay는 0.0005
- Gradient vanishing을 해결하기 위한 weight 초기화 전략
  - 레이어가 깊으면 gradient vanishing 현상이 일어남
  - 이를 해결하기 위해 먼저 A 타입 (11 레이어)을 학습하고 이를 이용해서 B 타입 네트워크를 fine-tuning (추가한 레이어는 랜덤 초기화)
  - 이 때는 배치 정규화가 없었기 때문에 번거롭게 학습, 요즘은 BN 넣어서 바로 학습 가능 (BN 짱짱)
- 다른 세팅은 AlexNet과 비슷

# Data augmentation

- 다양한 이미지 스케일링 전략
    - AlexNet은 256x256 리사이즈 -> 224x224 크롭 하지만 VGGNet은 리사이즈 크기를 다양하게 설정
    - **싱글스케일**: 리사이즈 이미지 크기  $S$ 를 256, 384로 설정 먼저  $S=256$ 인 경우로 학습하고,  $S=384$ 로 바꾸고 fine-tune
    - **멀티스케일**:  $S$ 를 [256, 384] 사이의 랜덤 값으로 설정 마찬가지로  $S=256$ 인 모델을 통해 fine-tune
- > 이미지에 존재하는 물체의 크기를 다양하게 입력으로 받을 수 있음

레이어	출력 크기	메모리 요구량	파라미터 수 (Bias X)
입력층	[224x224x3]	$224*224*3=150K$	0
CONV3-64	[224x224x64]	$224*224*64=3.2M$	$3*3*3*64=1.7K$
CONV3-64	[224x224x64]	$224*224*64=3.2M$	$3*3*64*64=37K$
POOL2	[112x112x64]	$112*112*64=800K$	0
CONV3-128	[112x112x128]	$112*112*128=1.6M$	$3*3*64*128=74K$
CONV3-128	[112x112x128]	$112*112*128=1.6M$	$3*3*128*128=147K$
POOL2	[56x56x128]	$56*56*128=400K$	0
CONV3-256	[56x56x256]	$56*56*256=800K$	$3*3*128*256=295K$
CONV3-256	[56x56x256]	$56*56*256=800K$	$3*3*256*256=590K$
CONV3-256	[56x56x256]	$56*56*256=800K$	$3*3*256*256=590K$
POOL2	[28x28x256]	$28*28*256=200K$	0
CONV3-512	[28x28x512]	$28*28*512=400K$	$3*3*256*512=1180K$
CONV3-512	[28x28x512]	$28*28*512=400K$	$3*3*512*512=2359K$
CONV3-512	[28x28x512]	$28*28*512=400K$	$3*3*512*512=2359K$
POOL2	[14x14x512]	$14*14*512=100K$	0
CONV3-512	[14x14x512]	$14*14*512=100K$	$3*3*512*512=2359K$
CONV3-512	[14x14x512]	$14*14*512=100K$	$3*3*512*512=2359K$
CONV3-512	[14x14x512]	$14*14*512=100K$	$3*3*512*512=2359K$
POOL2	[7x7x512]	$7*7*512=25K$	0
FC	[1x1x4096]	4096	$7*7*512*4096=102M$
FC	[1x1x4096]	4096	$4096*4096=1.6M$
FC	[1x1x1000]	1000	$4096*1000=4M$
Total		24M	138M

대부분의 메모리 요구량은  
앞쪽 컨볼루션 레이어에서  
발생

레이어	출력 크기	메모리 요구량	파라미터 수 (Bias X)
입력층	[224x224x3]	$224*224*3=150K$	0
CONV3-64	[224x224x64]	$224*224*64=3.2M$	$3*3*3*64=1.7K$
CONV3-64	[224x224x64]	$224*224*64=3.2M$	$3*3*64*64=37K$
POOL2	[112x112x64]	$112*112*64=800K$	0
CONV3-128	[112x112x128]	$112*112*128=1.6M$	$3*3*64*128=74K$
CONV3-128	[112x112x128]	$112*112*128=1.6M$	$3*3*128*128=147K$
POOL2	[56x56x128]	$56*56*128=400K$	0
CONV3-256	[56x56x256]	$56*56*256=800K$	$3*3*128*256=295K$
CONV3-256	[56x56x256]	$56*56*256=800K$	$3*3*256*256=590K$
CONV3-256	[56x56x256]	$56*56*256=800K$	$3*3*256*256=590K$
POOL2	[28x28x256]	$28*28*256=200K$	0
CONV3-512	[28x28x512]	$28*28*512=400K$	$3*3*256*512=1180K$
CONV3-512	[28x28x512]	$28*28*512=400K$	$3*3*512*512=2359K$
CONV3-512	[28x28x512]	$28*28*512=400K$	$3*3*512*512=2359K$
POOL2	[14x14x512]	$14*14*512=100K$	0
CONV3-512	[14x14x512]	$14*14*512=100K$	$3*3*512*512=2359K$
CONV3-512	[14x14x512]	$14*14*512=100K$	$3*3*512*512=2359K$
CONV3-512	[14x14x512]	$14*14*512=100K$	$3*3*512*512=2359K$
POOL2	[7x7x512]	$7*7*512=25K$	0
FC	[1x1x4096]	4096	$7*7*512*4096=102M$
FC	[1x1x4096]	4096	$4096*4096=1.6M$
FC	[1x1x1000]	1000	$4096*1000=4M$
Total		24M	138M

레이어	출력 크기	메모리 요구량	파라미터 수 (Bias X)
입력층	[224x224x3]	$224*224*3=150K$	0
CONV3-64	[224x224x64]	$224*224*64=3.2M$	$3*3*3*64=1.7K$
CONV3-64	[224x224x64]	$224*224*64=3.2M$	$3*3*64*64=37K$
POOL2	[112x112x64]	$112*112*64=800K$	0
CONV3-128	[112x112x128]	$112*112*128=1.6M$	$3*3*64*128=74K$
CONV3-128	[112x112x128]	$112*112*128=1.6M$	$3*3*128*128=147K$
POOL2	[56x56x128]	$56*56*128=400K$	0
CONV3-256	[56x56x256]	$56*56*256=800K$	$3*3*128*256=295K$
CONV3-256	[56x56x256]	$56*56*256=800K$	$3*3*256*256=590K$
CONV3-256	[56x56x256]	$56*56*256=800K$	$3*3*256*256=590K$
POOL2	[28x28x256]	$28*28*256=200K$	0
CONV3-512	[28x28x512]	$28*28*512=400K$	$3*3*256*512=1180K$
CONV3-512	[28x28x512]	$28*28*512=400K$	$3*3*512*512=2359K$
CONV3-512	[28x28x512]	$28*28*512=400K$	$3*3*512*512=2359K$
POOL2	[14x14x512]	$14*14*512=100K$	0
CONV3-512	[14x14x512]	$14*14*512=100K$	$3*3*512*512=2359K$
CONV3-512	[14x14x512]	$14*14*512=100K$	$3*3*512*512=2359K$
CONV3-512	[14x14x512]	$14*14*512=100K$	$3*3*512*512=2359K$
POOL2	[7x7x512]	$7*7*512=25K$	0
FC	[1x1x4096]	4096	$7*7*512*4096=102M$
FC	[1x1x4096]	4096	$4096*4096=1.6M$
FC	[1x1x1000]	1000	$4096*1000=4M$
Total		24M	138M

대부분의 메모리 요구량은  
앞쪽 컨볼루션 레이어에서  
발생

대부분의 파라미터가  
FC 레이어에서 생성

레이어	출력 크기	메모리 요구량	파라미터 수 (Bias X)
입력층	[224x224x3]	$224*224*3=150K$	0
CONV3-64	[224x224x64]	$224*224*64=3.2M$	$3*3*3*64=1.7K$
CONV3-64	[224x224x64]	$224*224*64=3.2M$	$3*3*64*64=37K$
POOL2	[112x112x64]	$112*112*64=800K$	0
CONV3-128	[112x112x128]	$112*112*128=1.6M$	$3*3*64*128=74K$
CONV3-128	[112x112x128]	$112*112*128=1.6M$	$3*3*128*128=147K$
POOL2	[56x56x128]	$56*56*128=400K$	0
CONV3-256	[56x56x256]	$56*56*256=800K$	$3*3*128*256=295K$
CONV3-256	[56x56x256]	$56*56*256=800K$	$3*3*256*256=590K$
CONV3-256	[56x56x256]	$56*56*256=800K$	$3*3*256*256=590K$
POOL2	[28x28x256]	$28*28*256=200K$	0
CONV3-512	[28x28x512]	$28*28*512=400K$	$3*3*256*512=1180K$
CONV3-512	[28x28x512]	$28*28*512=400K$	$3*3*512*512=2359K$
CONV3-512	[28x28x512]	$28*28*512=400K$	$3*3*512*512=2359K$
POOL2	[14x14x512]	$14*14*512=100K$	0
CONV3-512	[14x14x512]	$14*14*512=100K$	$3*3*512*512=2359K$
CONV3-512	[14x14x512]	$14*14*512=100K$	$3*3*512*512=2359K$
CONV3-512	[14x14x512]	$14*14*512=100K$	$3*3*512*512=2359K$
POOL2	[7x7x512]	$7*7*512=25K$	0
FC	[1x1x4096]	4096	$7*7*512*4096=102M$
FC	[1x1x4096]	4096	$4096*4096=1.6M$
FC	[1x1x1000]	1000	$4096*1000=4M$
Total		24M	138M

대부분의 메모리 요구량은  
앞쪽 컨볼루션 레이어에서  
발생

Forward시 메모리 요구량은  
 $24M*4\text{byte}=93\text{MB}$ ,  
backprop 포함하면 186MB  
배치사이즈 32면 6GB

대부분의 파라미터가  
FC 레이어에서 생성

# 지금까지 Summary

- AlexNet
  - Dropout + ReLU
  - 데이터 augmentation
- VGGNet
  - 작은 크기 컨볼루션 (3x3)
  - 더 강력한 augmentation
  - 모델은 깊으면 깊을수록 좋다
  - 하지만 엄청나게 많은 파라미터.. (1.3억개)

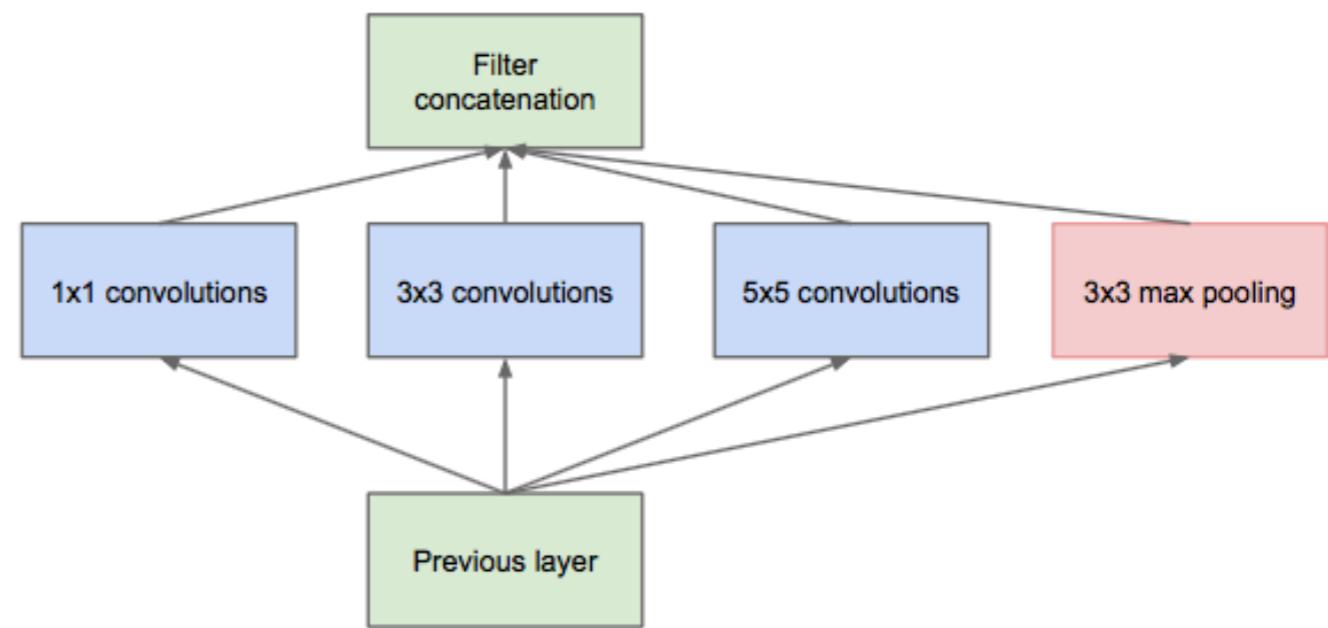
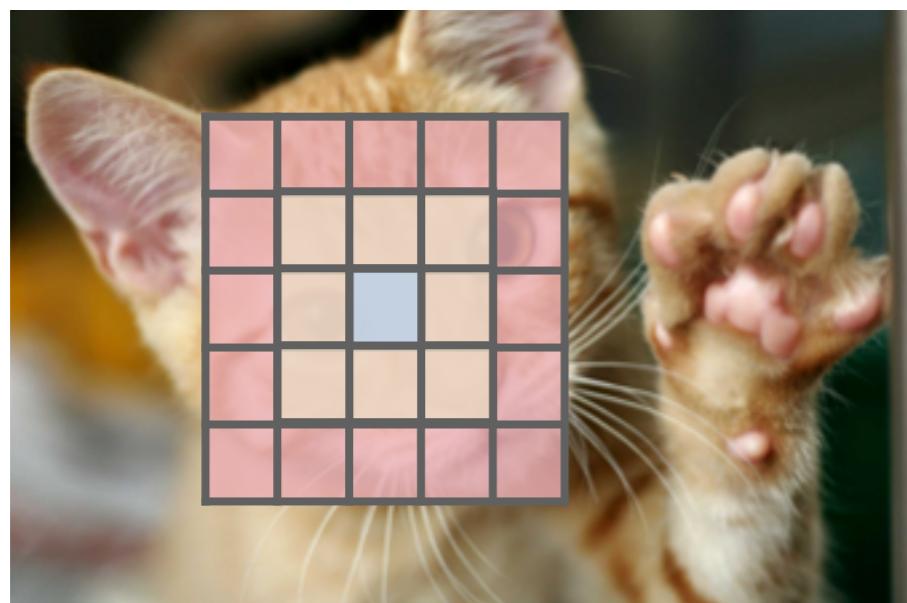
# GoogLeNet (2014)

- 2014년 이미지넷 대회 1위
  - 의외로 VGGNet에 밀려 자주 사용되지는 않음 (대신 구글이 엄청 밀어줌)
- GoogLeNet의 주요 포인트
  - **Inception** 모듈
  - Gradient vanishing을 막기 위해 보조 분류기 사용
  - **Average 풀링**
  - 파라미터 수를 대폭 줄임 (6.8M) 이는 AlexNet 대비 12배 감소한 수치

# Inception 모듈

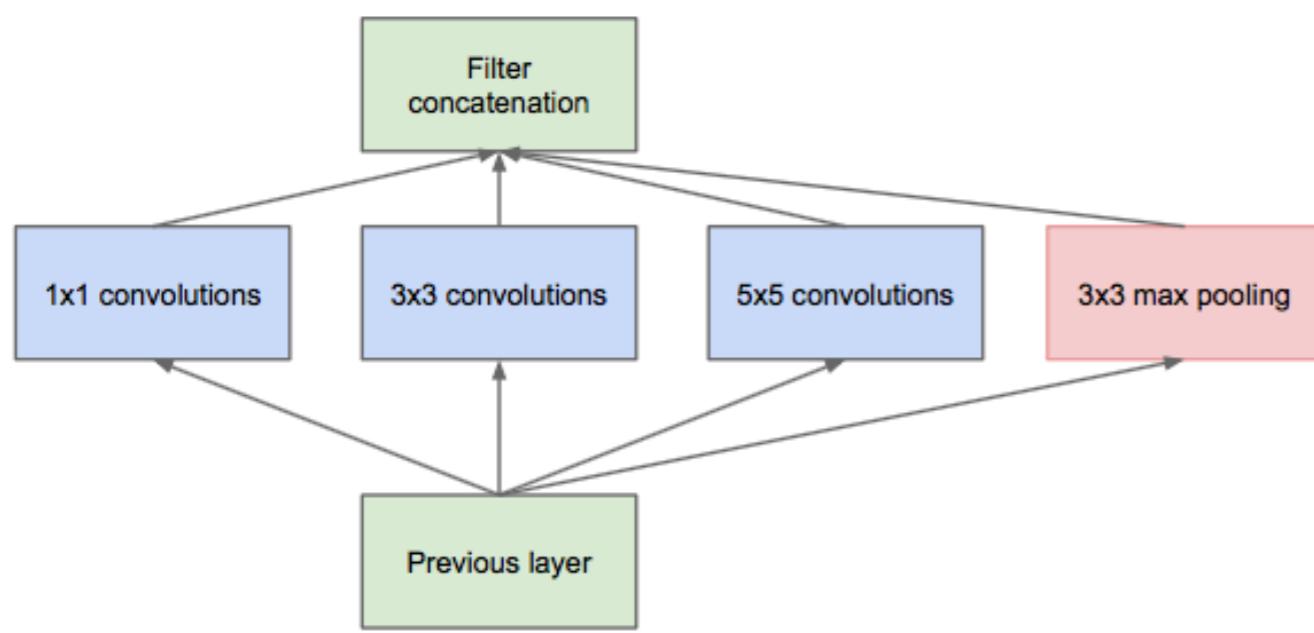
- Inception 모듈

- 1x1, 3x3, 5x5 컨볼루션과 3x3 MAX 풀링을 동시에 적용한 모듈
- 각 레이어의 결과값을 channel-wise하게 합침
- 동시에 **여러 형태의 구조**를 뽑아낼 수 있음
- 예) 5x5는 큰 특징을, 1x1은 매우 로컬한 특징들을 검출

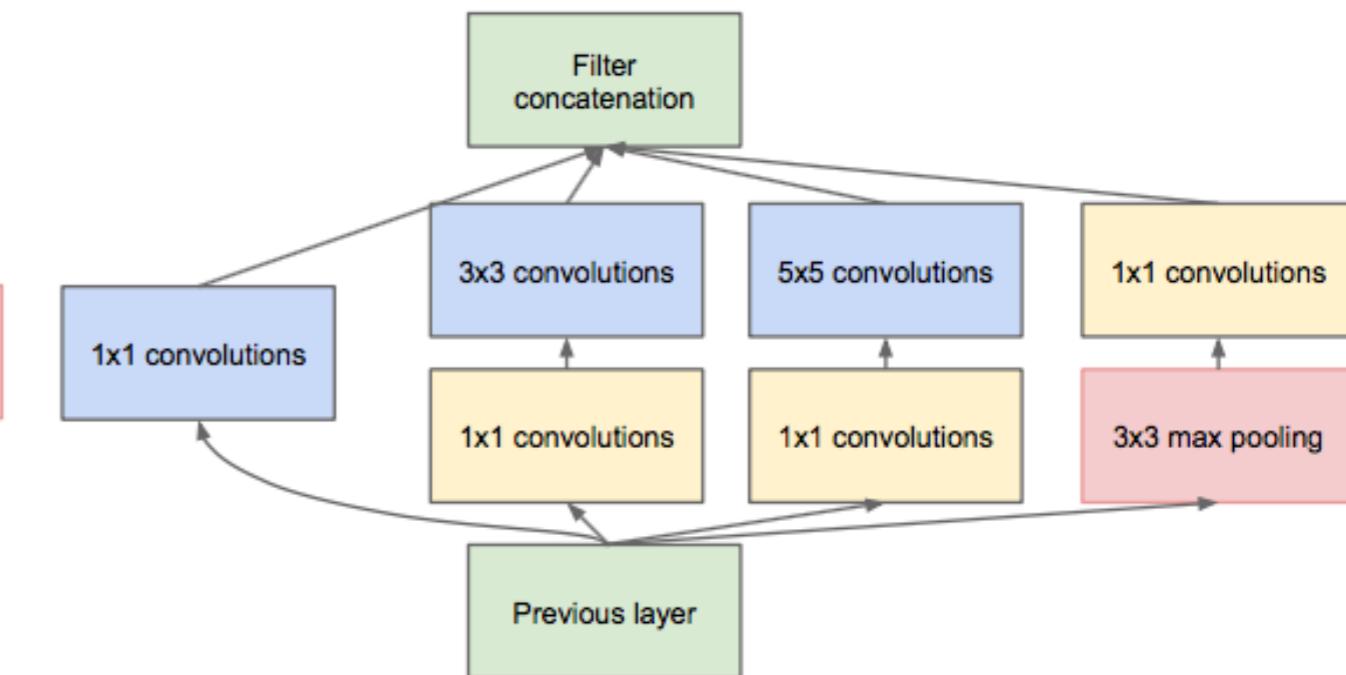


# Bottleneck 디자인

- 여러개의 필터를 조합해서 사용하기 때문에 계산이 많음
  - 특히  $5 \times 5$  컨볼루션
  - 이를 해결하기 위해  $1 \times 1$  컨볼루션을 이용한 **bottleneck 디자인** 사용
  - $1 \times 1$  컨볼루션을 사용해서 **차원을 낮춤** -> 메인 필터 ( $3 \times 3$ ,  $5 \times 5$ ) 적용

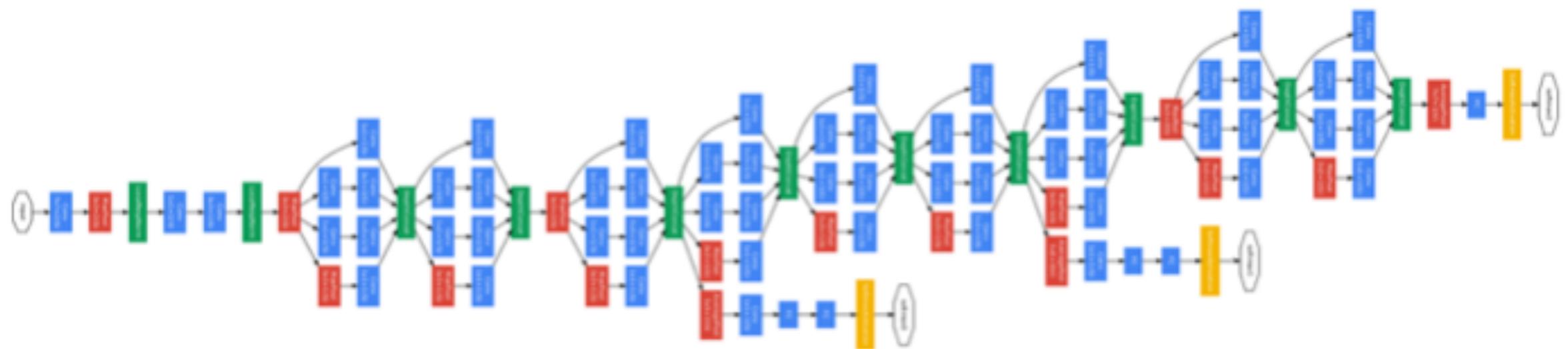


기본 inception 모듈

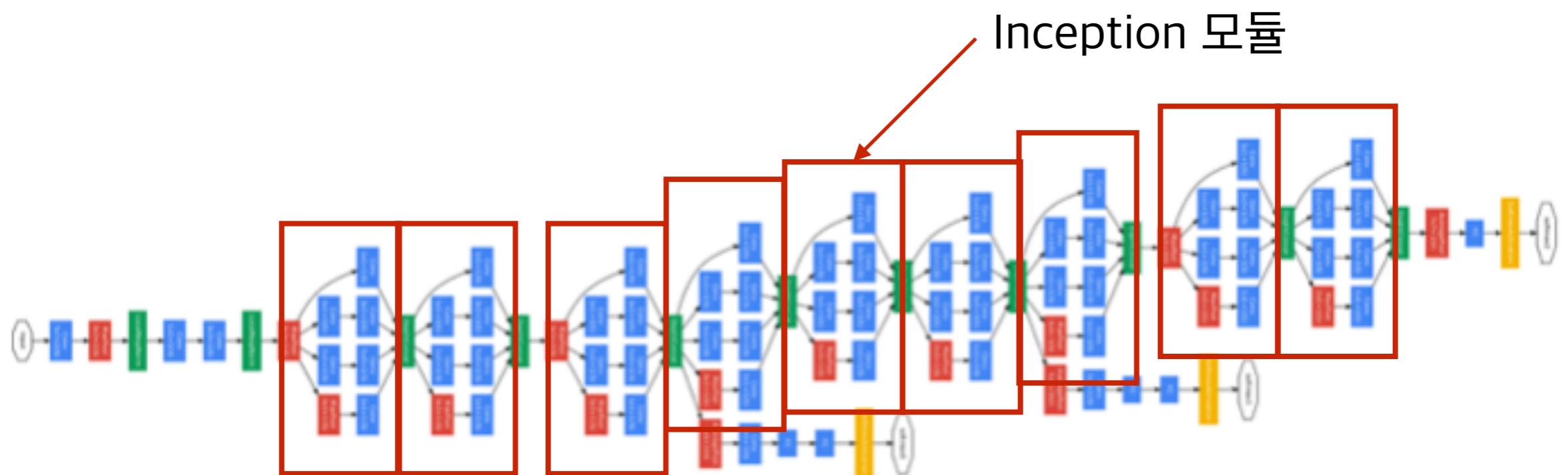


bottleneck inception 모듈

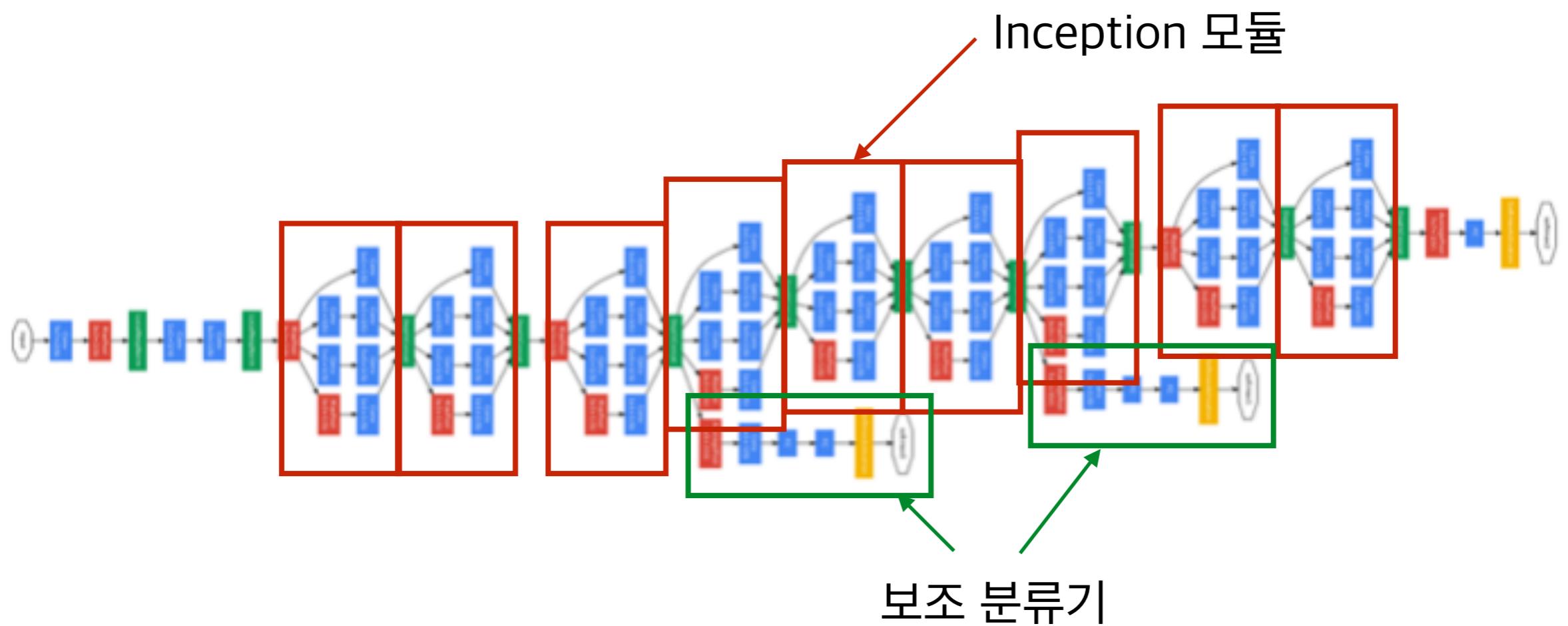
# GoogLeNet 아키텍쳐



# GoogLeNet 아키텍쳐



# GoogLeNet 아키텍쳐



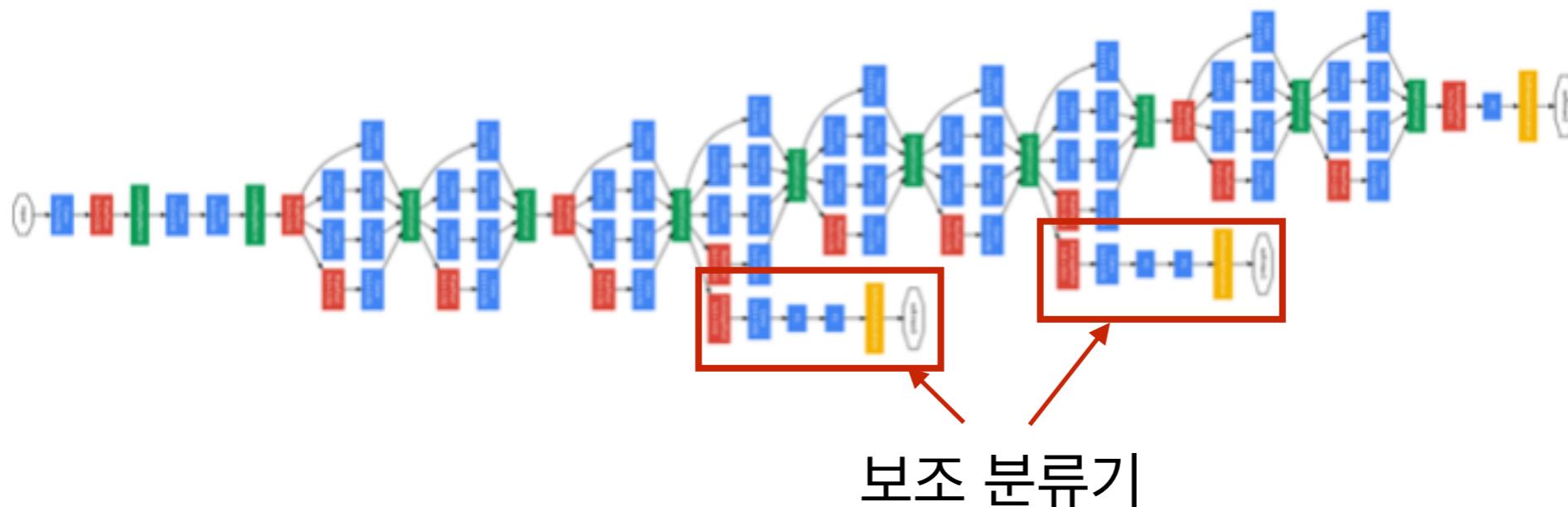
# 보조 분류기

- 레이어가 깊기 때문에 gradient vanishing의 우려
  - 배치 정규화가 없었던 시절
  - Vanishing을 막기 위해 보조 분류기를 추가해서 그라디언트를 전파
  - 최종 loss = 메인 분류기 loss + 보조 분류기 loss \* 0.3
  - 테스트시에는 제거



# 보조 분류기

- 레이어가 깊기 때문에 gradient vanishing의 우려
  - 배치 정규화가 없었던 시절
  - Vanishing을 막기 위해 보조 분류기를 추가해서 그라디언트를 전파
  - 최종 loss = 메인 분류기 loss + 보조 분류기 loss \* 0.3
  - 테스트시에는 제거



# 파라미터 수를 줄이기

- VGGNet은 FC 레이어에서 대부분의 파라미터가 생성 (AlexNet 또한 마찬가지)

FC	[1x1x4096]	4096	$7*7*512*4096=102M$
FC	[1x1x4096]	4096	$4096*4096=1.6M$
FC	[1x1x1000]	1000	$4096*1000=4M$
Total		24M	138M

) 대부분의 파라미터가  
FC 레이어에서 생성

# 파라미터 수를 줄이기

- VGGNet은 FC 레이어에서 대부분의 파라미터가 생성 (AlexNet 또한 마찬가지)

FC	[1x1x4096]	4096	$7*7*512*4096=102M$
FC	[1x1x4096]	4096	$4096*4096=1.6M$
FC	[1x1x1000]	1000	$4096*1000=4M$
Total		24M	138M

) 대부분의 파라미터가  
FC 레이어에서 생성

- FC 레이어를 최대한 줄여서 파라미터 생성을 억제하자
  - **Average 풀링**
  - 마지막 최종 레이어만 남겨놓고 FC 레이어 모두 삭제

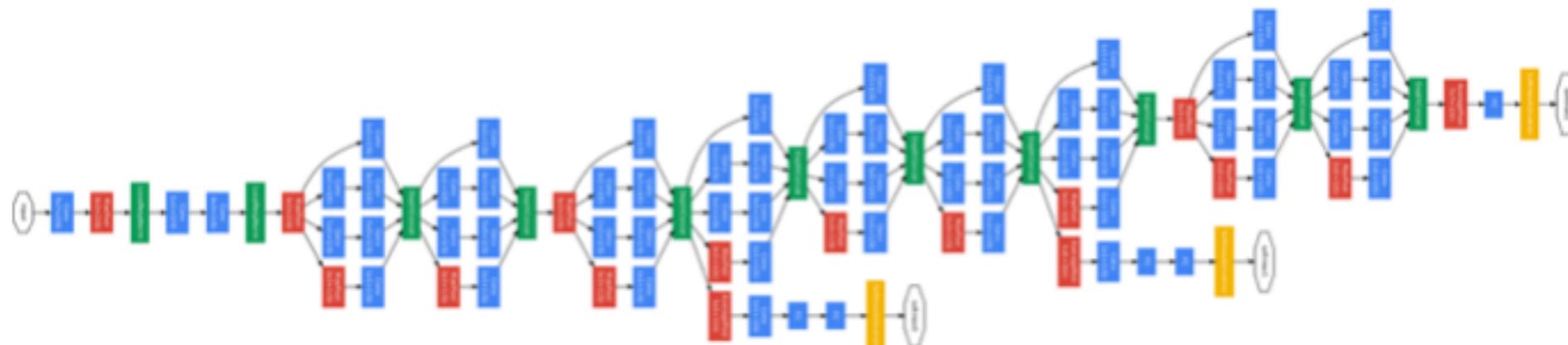
# 파라미터 수를 줄이기

- VGGNet은 FC 레이어에서 대부분의 파라미터가 생성 (AlexNet 또한 마찬가지)

FC	[1x1x4096]	4096	$7*7*512*4096=102M$
FC	[1x1x4096]	4096	$4096*4096=1.6M$
FC	[1x1x1000]	1000	$4096*1000=4M$
Total		24M	138M

) 대부분의 파라미터가  
FC 레이어에서 생성

- FC 레이어를 최대한 줄여서 파라미터 생성을 억제하자
  - Average 풀링
  - 마지막 최종 레이어만 남겨놓고 FC 레이어 모두 삭제



# 파라미터 수를 줄이기

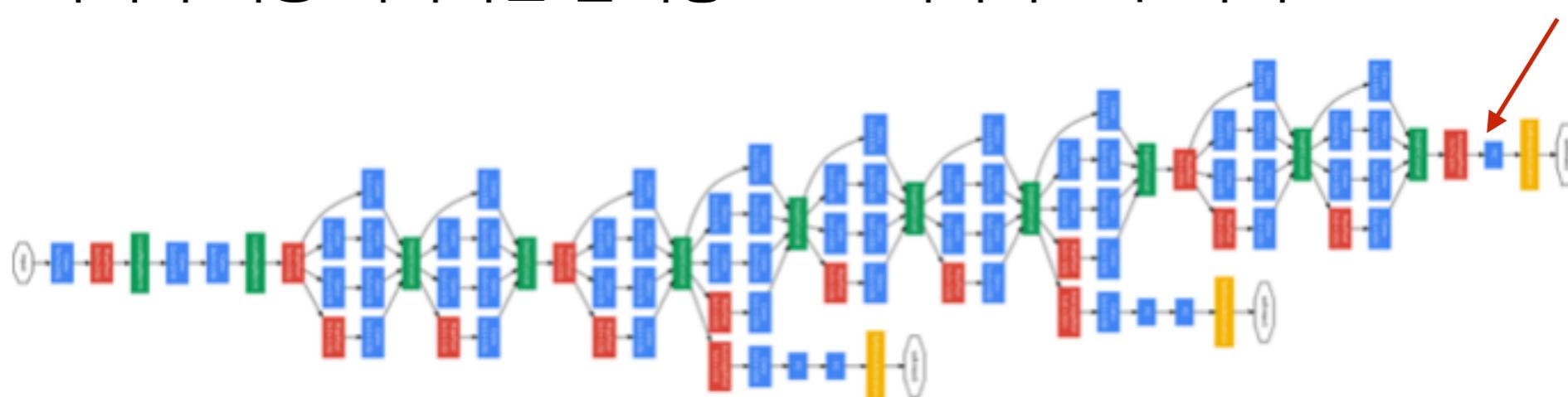
- VGGNet은 FC 레이어에서 대부분의 파라미터가 생성 (AlexNet 또한 마찬가지)

FC	[1x1x4096]	4096	$7*7*512*4096=102M$
FC	[1x1x4096]	4096	$4096*4096=1.6M$
FC	[1x1x1000]	1000	$4096*1000=4M$
Total		24M	138M

) 대부분의 파라미터가  
FC 레이어에서 생성

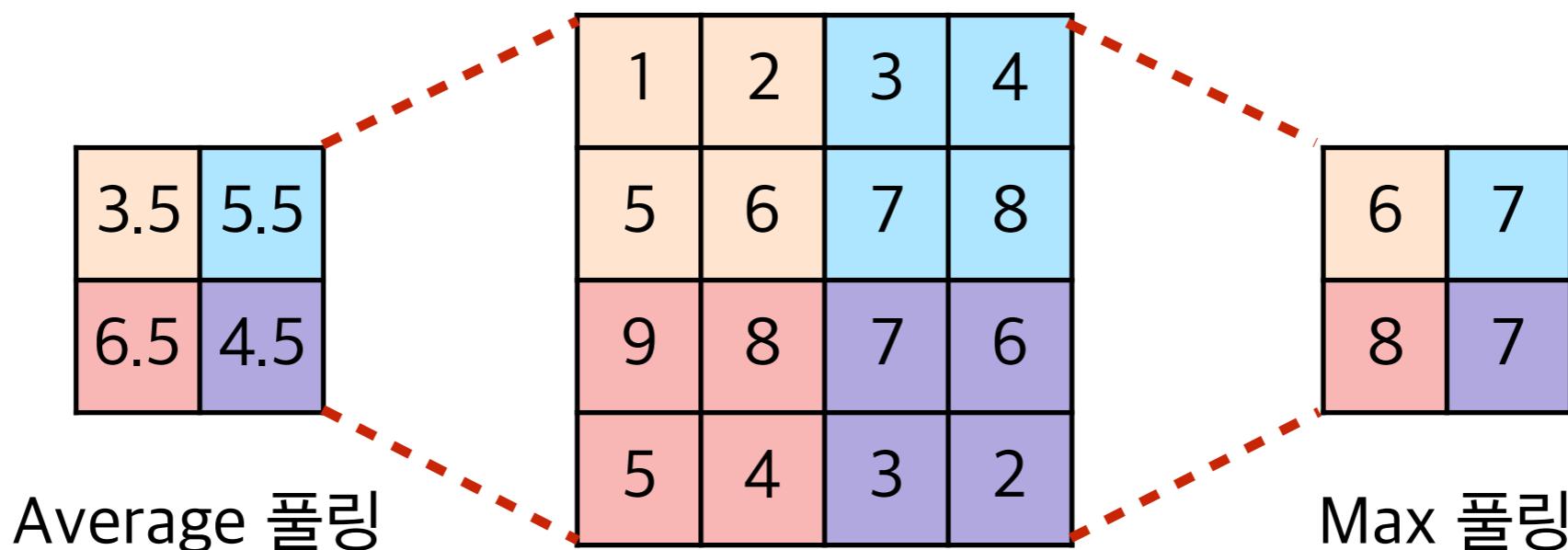
- FC 레이어를 최대한 줄여서 파라미터 생성을 억제하자
  - Average 풀링
  - 마지막 최종 레이어만 남겨놓고 FC 레이어 모두 삭제

FC 레이어가 없음



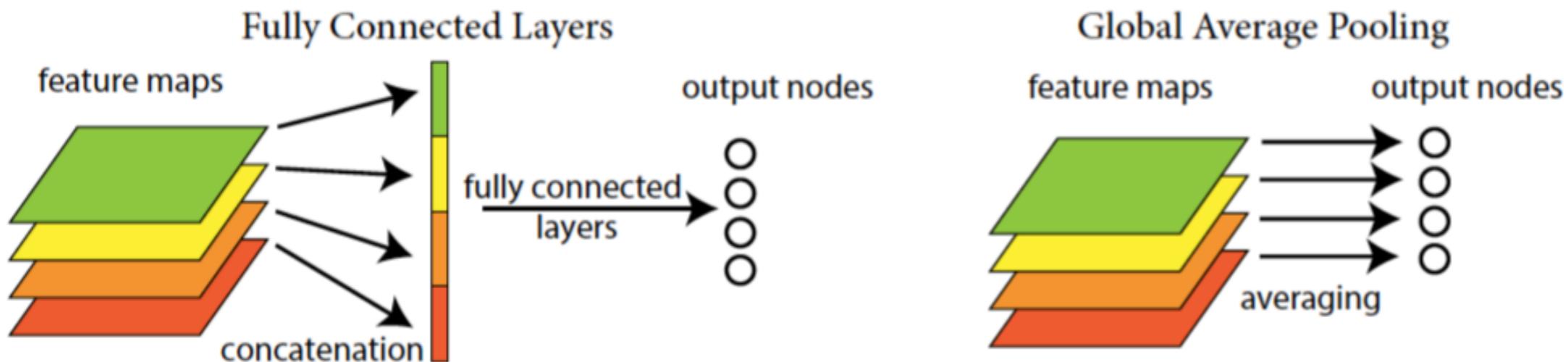
# Average 풀링

- 다시 정리:
  - FC 레이어는 파라미터가 많아서 과적합의 우려가 있음
  - 그럼 FC 레이어를 지우자!
  - 이를 Average 풀링을 통해서 해결 가능



# Average 풀링

1. 마지막 컨볼루션 결과를 각 채널별로 average 풀링 적용
  - 예) 마지막 컨볼루션 특징맵이  $7 \times 7 \times 2048 \rightarrow 1 \times 1 \times 2048$
2. 클래스 개수에 맞추기 위한 FC  $\rightarrow$  softmax 계산
  1. 꼭 필요한 FC를 제외하고 별도의 FC가 없음
  2. 네트워크의 입력이 꼭 고정될 필요 없음  
(하지만 다른 이미지 크기는 성능이 하락할 가능성이 있음)



# 파라미터 개수

# 성능

<b>Team</b>	<b>Year</b>	<b>Place</b>	<b>Error (top-5)</b>	
SuperVision	2012	1st	16.4%	
SuperVision	2012	1st	15.3%	(AlexNet)
Clarifai	2013	1st	11.7%	
Clarifai	2013	1st	11.2%	
MSRA	2014	3rd	7.35%	
VGG	2014	2nd	7.32%	
GoogLeNet	2014	1st	6.67%	

# ResNet (2015)

- 2015년 이미지넷 대회 1위
  - Microsoft Research Asia (MSRA) 팀
- ResNet의 주요 포인트
  - **Residual 연결**
  - 매우매우 깊은 네트워크
  - PReLU, He 초기화 방법 (사실 이건 그 전 논문 PReLUNet에서 제안)

[1] He, Kaiming, et al. "Deep residual learning for image recognition."

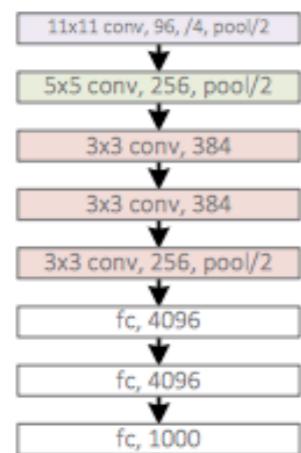
Proceedings of the IEEE conference on computer vision and pattern recognition. 2016.

[2] He, Kaiming, et al. "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification."

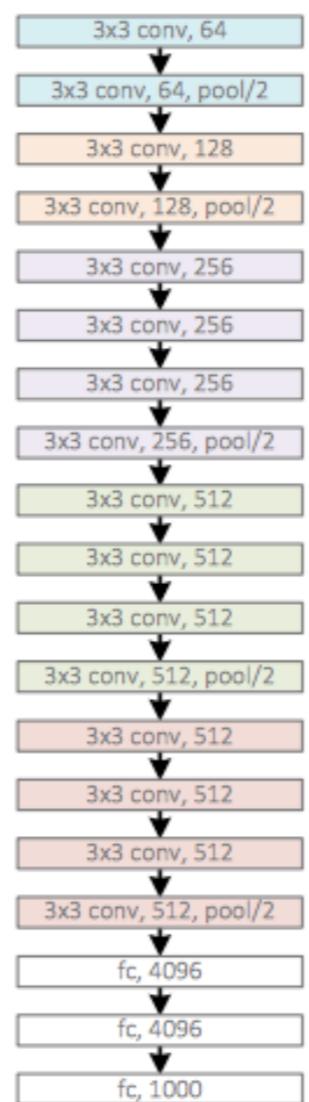
Proceedings of the IEEE international conference on computer vision. 2015.

# Revolution of Depth

AlexNet, 8 layers  
(ILSVRC 2012)



VGG, 19 layers  
(ILSVRC 2014)



GoogleNet, 22 layers  
(ILSVRC 2014)



Kaiming He, Xiangyu Zhang, Shaoqing Ren, & Jian Sun. "Deep Residual Learning for Image Recognition". CVPR 2016.

# Revolution of Depth

AlexNet, 8 layers  
(ILSVRC 2012)



VGG, 19 layers  
(ILSVRC 2014)



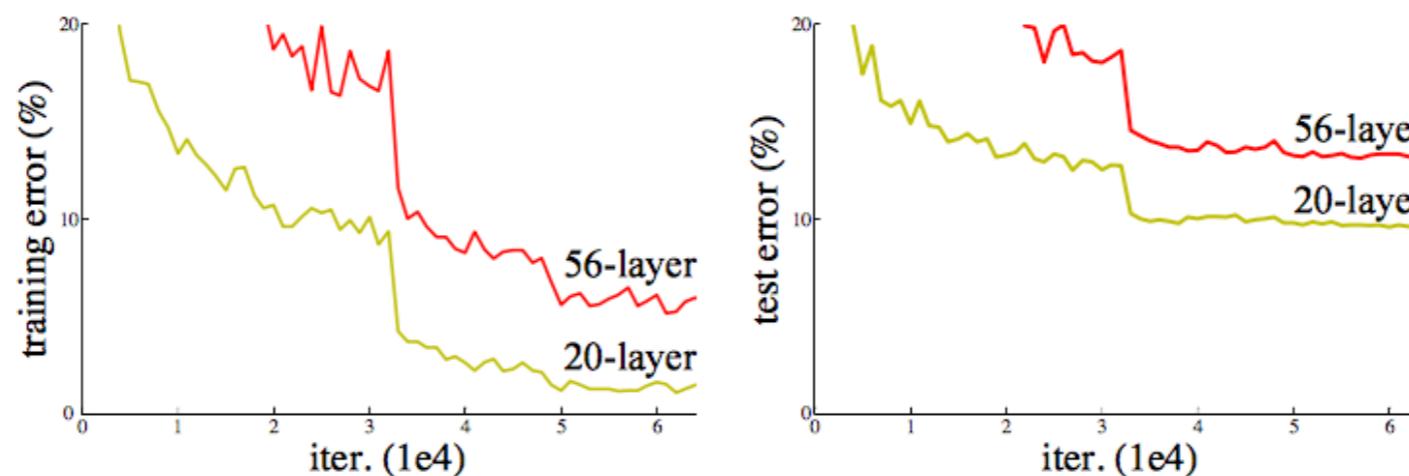
ResNet, 152 layers  
(ILSVRC 2015)



Kaiming He, Xiangyu Zhang, Shaoqing Ren, & Jian Sun. "Deep Residual Learning for Image Recognition". CVPR 2016.

# 그러나..

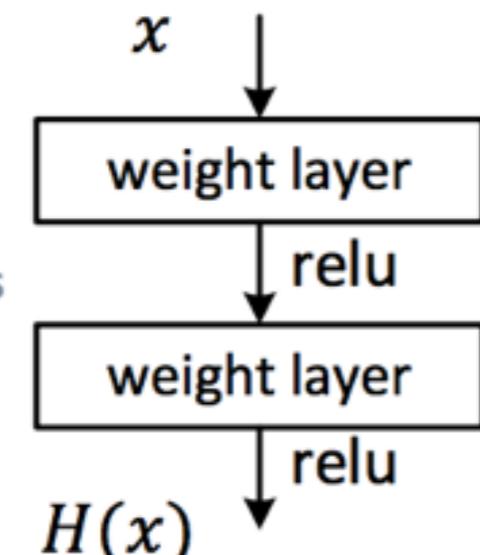
- 단순히 레이어를 쌓는 것만으로 성능이 올라가지 않음



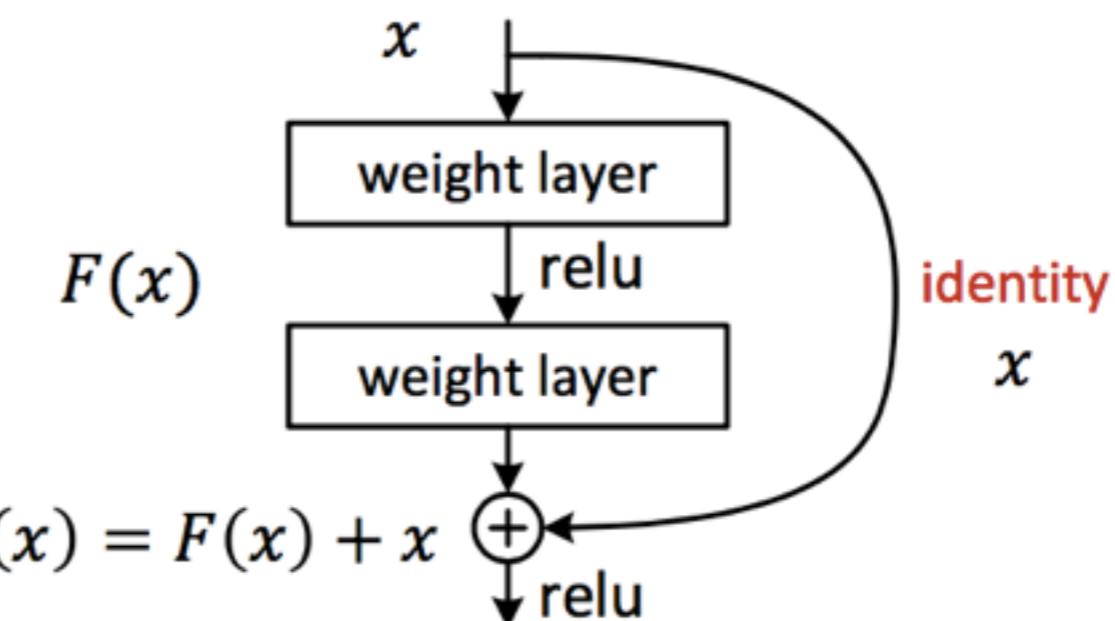
- VGGNet 스타일로 레이어를 깊게 쌓은 결과 **56 레이어** 짜리 네트워크가 **20 레이어** 네트워크보다 **테스트 / 학습 정확도 모두 낮음**
- 과적합에 의해 테스트 성능이 낮은 것은 일반적, 하지만 학습 데이터에 대해서도 성능이 낮은 것은 매우 의외의 결과
- 논문에서는 이를 **degradation** 문제라고 정의

# Deep Residual Learning

- Degradation 문제를 해결하기 위한 네트워크 디자인



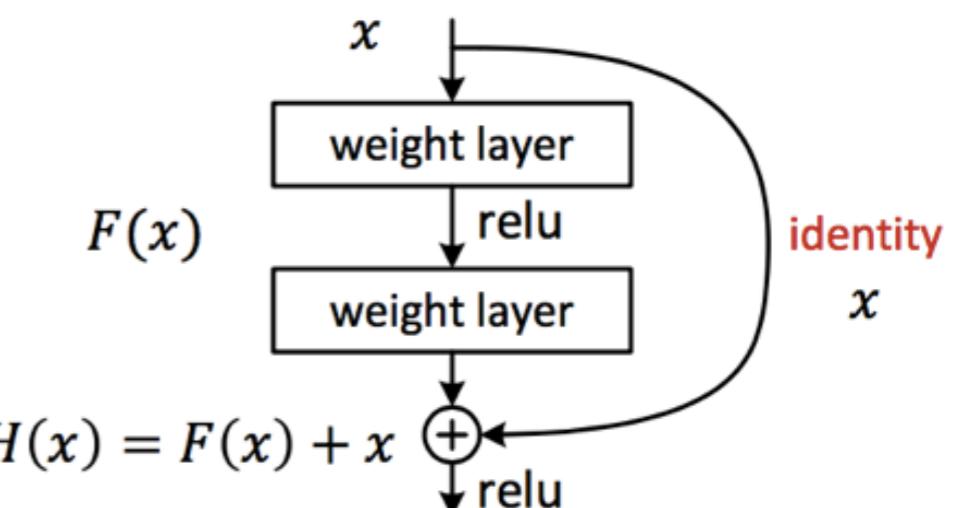
일반 네트워크



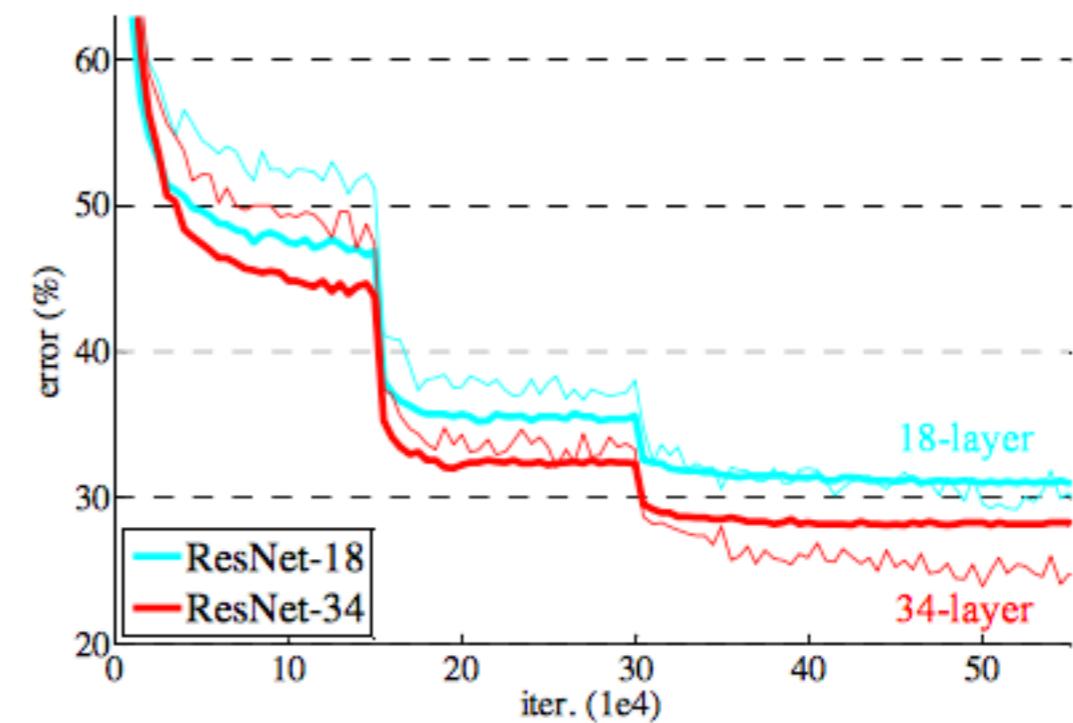
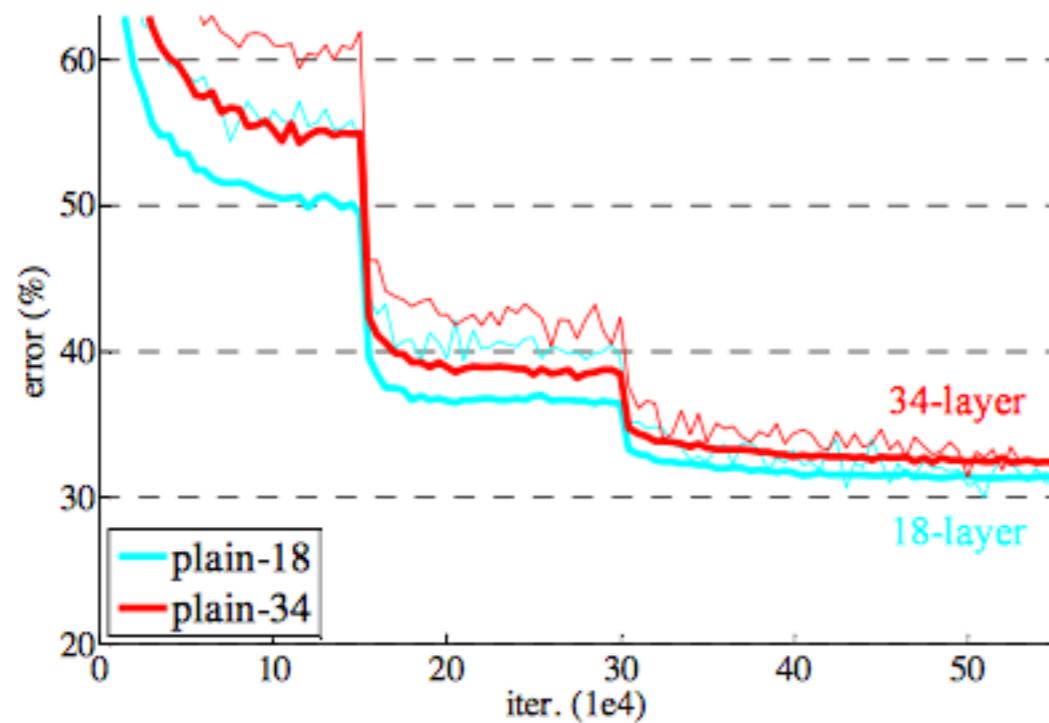
Residual 네트워크

# Residual Block

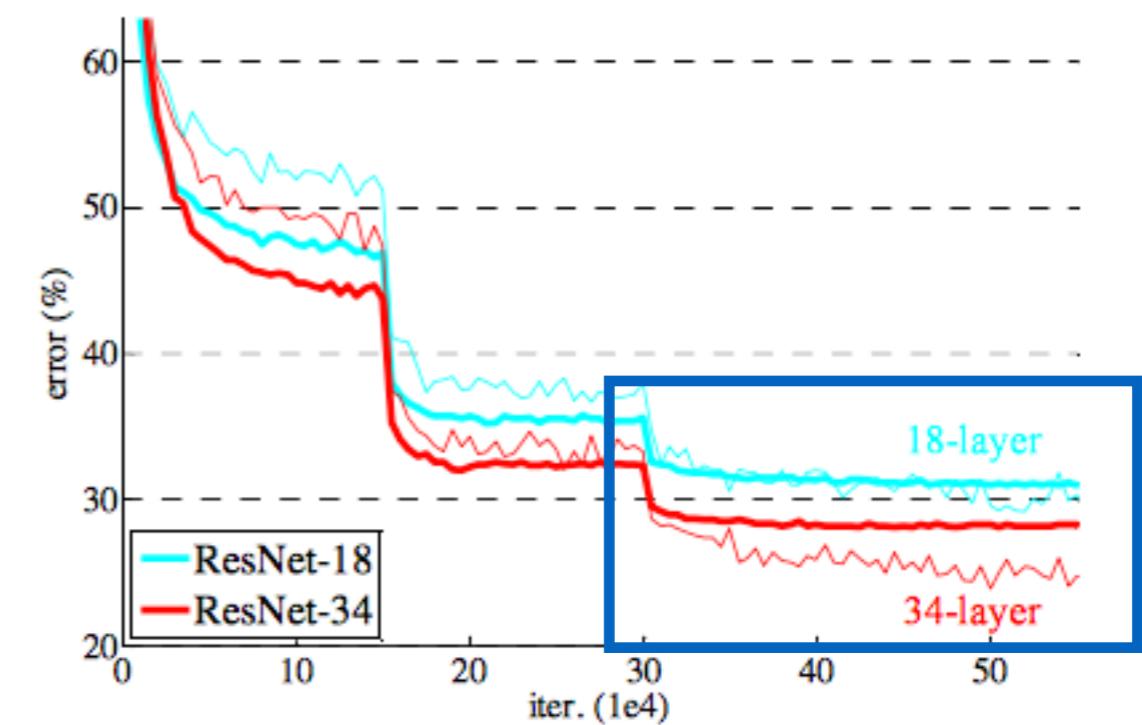
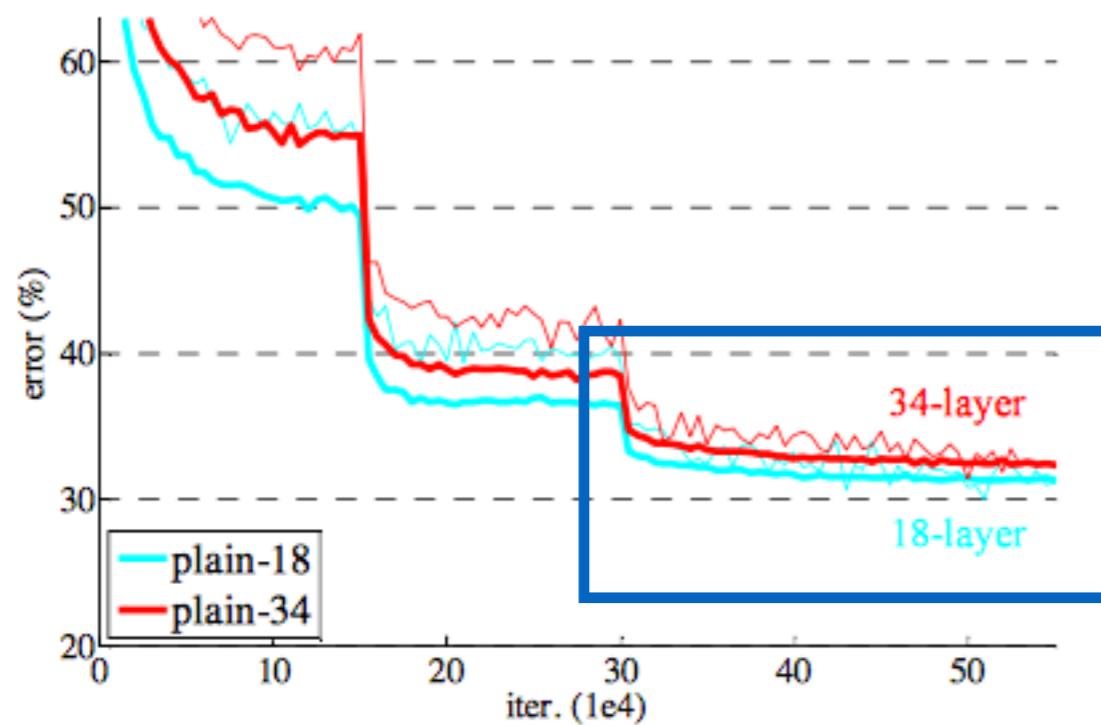
- 왜 잘될까?
  - Degradation 문제는 **gradient vanishing**이 원인:  
아무리 BN을 써도 레이어가 매우 깊으면 어쨌거나 그라디언트가 소멸됨
- ResNet은 어떻게 이를 해결했나
  - Residual 연결을 통해 그라디언트가 지름길로 흐름  
(원활한 그라디언트의 전파)
  - weight는 입력과 출력값의 차이를 학습 <- 그래서 residual  
(이전 모델들은 결과값을  
직접적으로 변환하도록 학습)



# Residual 연결 성능

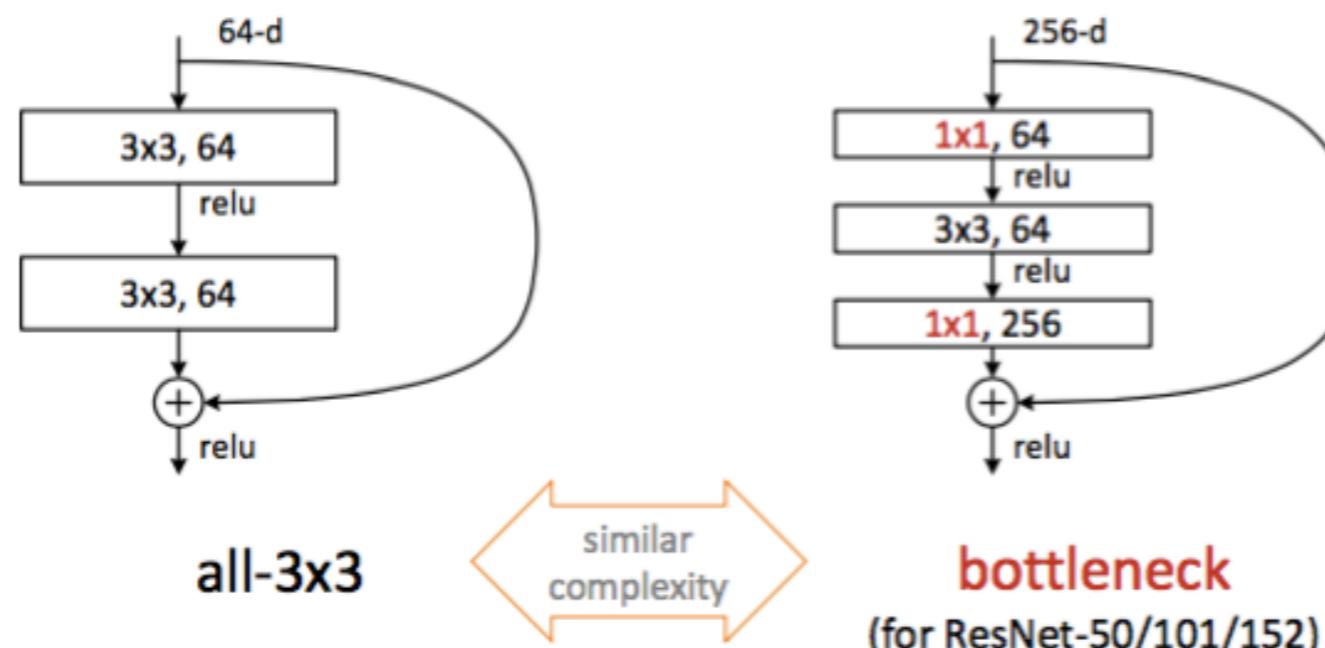


# Residual 연결 성능



# Deep Residual Bottleneck

- GoogLeNet과 마찬가지로 파라미터를 줄이기 위한 노력
  - 기존 residual block을 bottleneck 스타일로 재구성
  - 얼마나 줄어드나?
  - 기존:  $(3 \times 3) \times 64 \times 64 + (3 \times 3) \times 64 \times 64 = 73728$
  - 병목:  $(1 \times 1) \times 256 \times 64 + (3 \times 3) \times 64 \times 64 + (1 \times 1) \times 64 \times 256 = 69632$
  - bottleneck이 더 넓은 모델임! (차원이 64 vs 256)



# Leaky ReLU / PReLU

- ReLU는  $x < 0$  인 영역에서 결과값이 모두 0
  - 이 경우 그라디언트가 전파가 되지 못하는 문제 (**dead neuron** 현상)
  - $x < 0$ 인 영역에도 약간의 기울기를 가지게 하자
- Leaky ReLU / Parametric ReLU

$$f(x) = \begin{cases} x & \text{if } x > 0 \\ 0.01x & \text{otherwise} \end{cases}$$

$$f(x) = \begin{cases} x & \text{if } x > 0 \\ ax & \text{otherwise} \end{cases}$$

Leaky ReLU는  $x < 0$ 인 경우의 기울기를 하이퍼 파라미터로,  
PReLU는 기울기도 학습하는 활성 함수

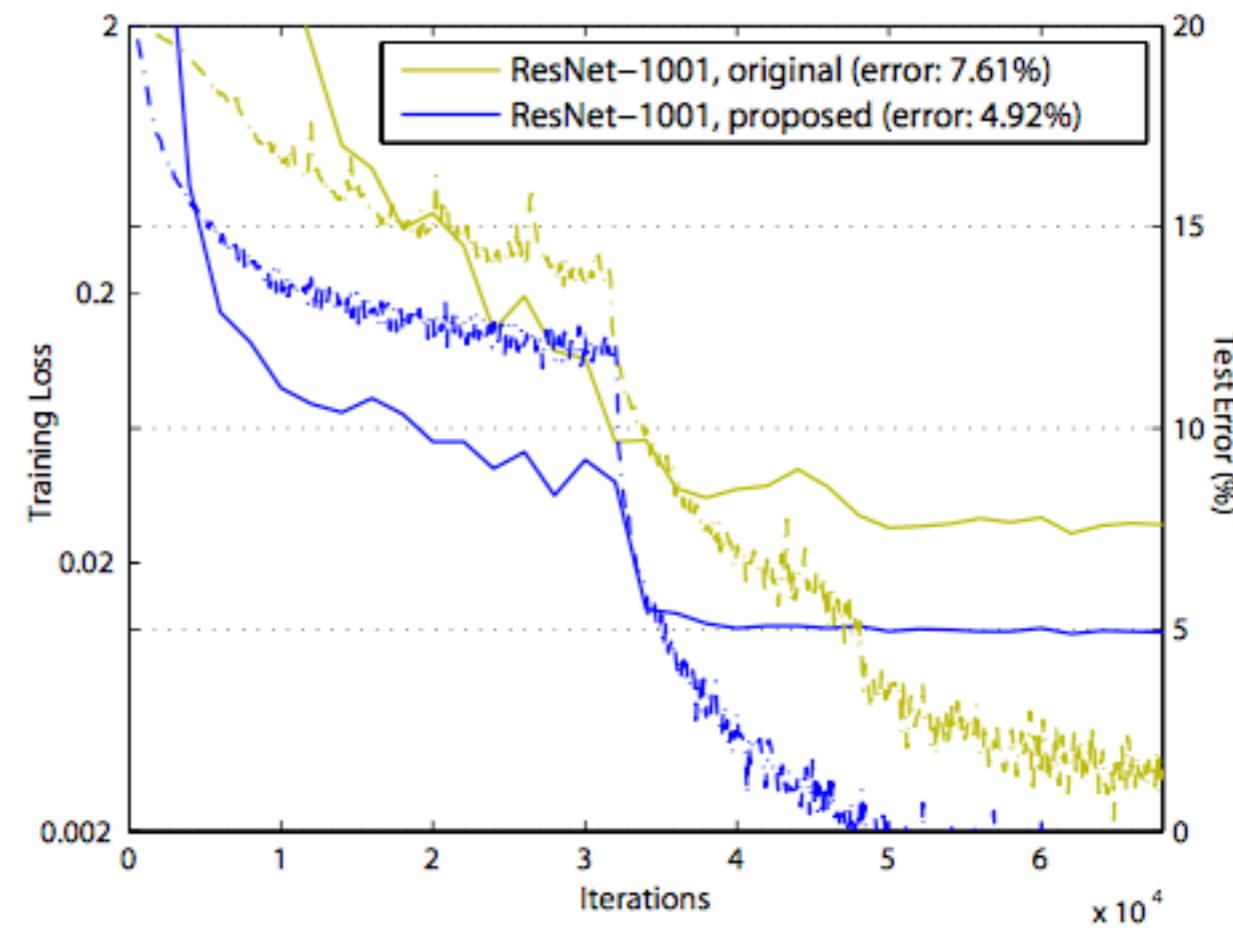
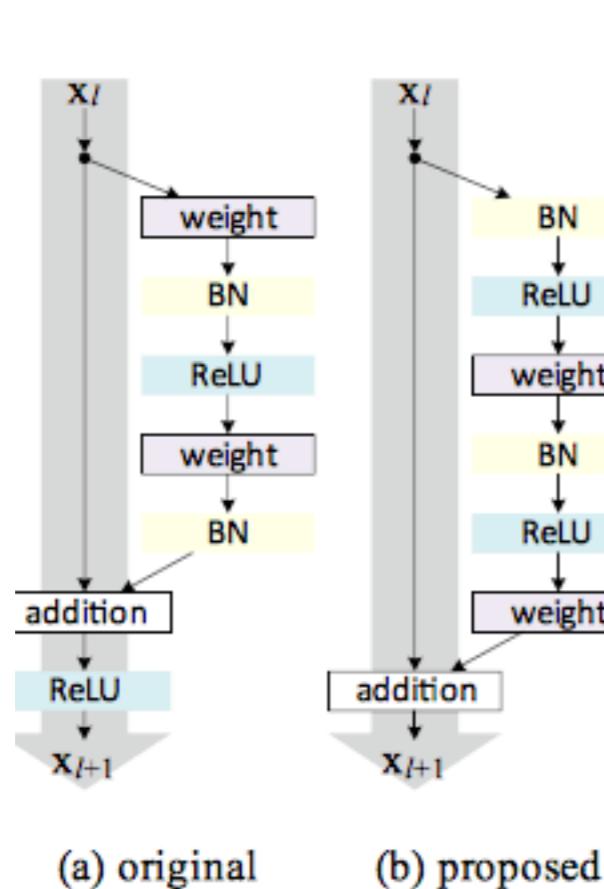
# ResNet 성능

method	top-5 err. ( <b>test</b> )
VGG [41] (ILSVRC'14)	7.32
GoogLeNet [44] (ILSVRC'14)	6.66
VGG [41] (v5)	6.8
PReLU-net [13]	4.94
BN-inception [16]	4.82
<b>ResNet (ILSVRC'15)</b>	<b>3.57</b>

참고로 사람이 대충 5% 정도 됩니다

# 그 후로도..

- 2015년 ResNet 발표 이후에도 **그라디언트 흐름 개선**에 관한 연구가 계속 진행됨



# 그 후로도..

- ResNexT (2017)
- Xception (2017)
- DensNet (2017)
- ...

[1] Xie, Saining, et al. "Aggregated residual transformations for deep neural networks." arXiv preprint arXiv:1611.05431 (2016).  
[2] Chollet, François. "Xception: Deep Learning with Depthwise Separable Convolutions." arXiv preprint arXiv:1610.02357 (2016).  
[3] Huang, Gao, et al. "Densely connected convolutional networks." arXiv preprint arXiv:1608.06993 (2016).