ACADENTIAL

Section 14.
Convolution Neural Networks (CNN)

ACADENTIAL

목차

ACADENTIAL

Objective
학습 목표

1부

- 이미지의 특성에 대해 이해하기
- Convolutional layer의 작동원리 이해하기
- Convolutional layer의 variant들 이해하기

2부

- 대표적인 CNN 모델들 이해하기
  - LeNet
  - AlexNet
  - VGGNet
  - InceptionNet, GoogLeNet
  - ResNet
  - DenseNet

ACADENTIAL

Objective
학습 목표

3부

- PyTorch로 Convolutional Layer 및 CNN 모델 구현하기
- VGGNet 모델 뜯어보기
- ResNet 모델 뜯어보기
- 직접 만든 CNN 모델, ResNet, VGGNet을 활용한 CV Project

ACADENTIAL

CNN 1부:
CNN에 대한 기본 이론 및 개념 정리

ACADENTIAL

14-1. 이미지 데이터의 특성

Convolution Neural Network

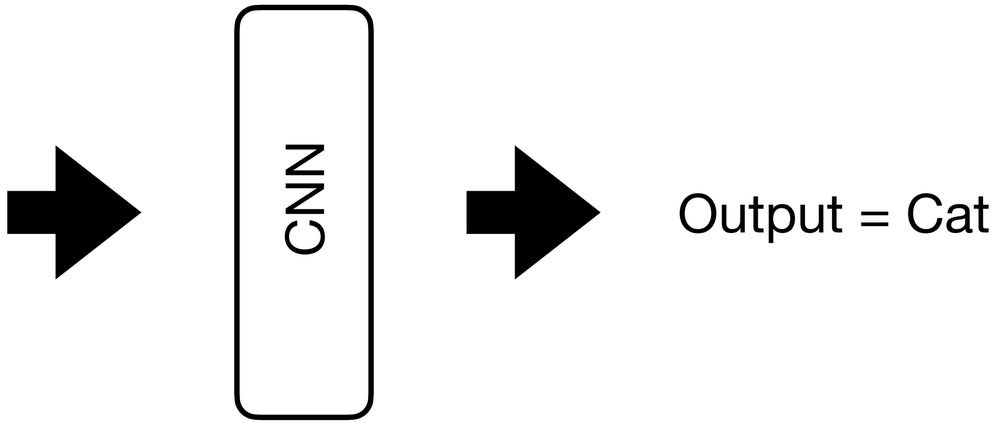
What is Convolution Neural Network (CNN)?

Convolution Neural Network:

- 이미지 데이터 (2D pixel data)의 처리에 특화된 모델
- Convolution Layer들로 구성됨!

먼저 이미지의 특성에 대해서 살펴보자!

Input Data 출처: dreamstime.com



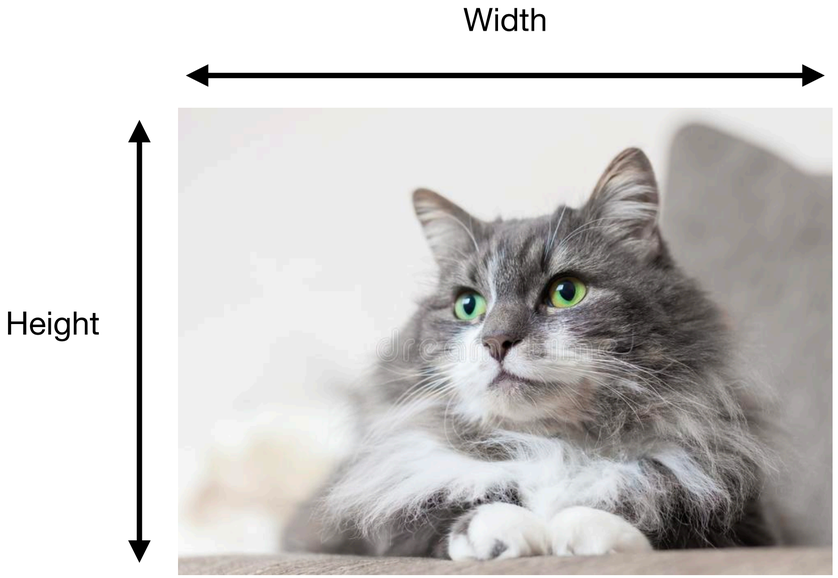
Convolution Neural Network

Image Data의 특성

참고: (normalize하기 전에) RGB의 이미지에서
각 pixel의 channel은 0~255사이의 값을 가진다.

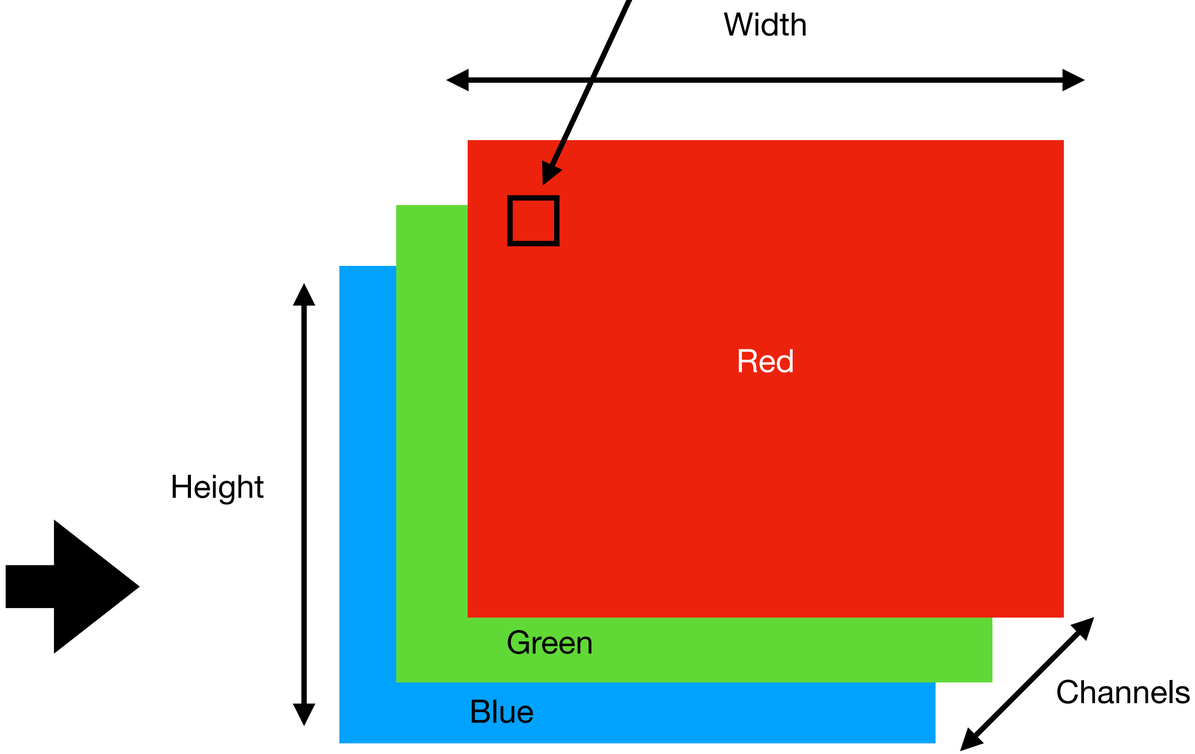
특성 1.

(Colored) Image data == 3D tensor



Input Data

출처: dreamstime.com



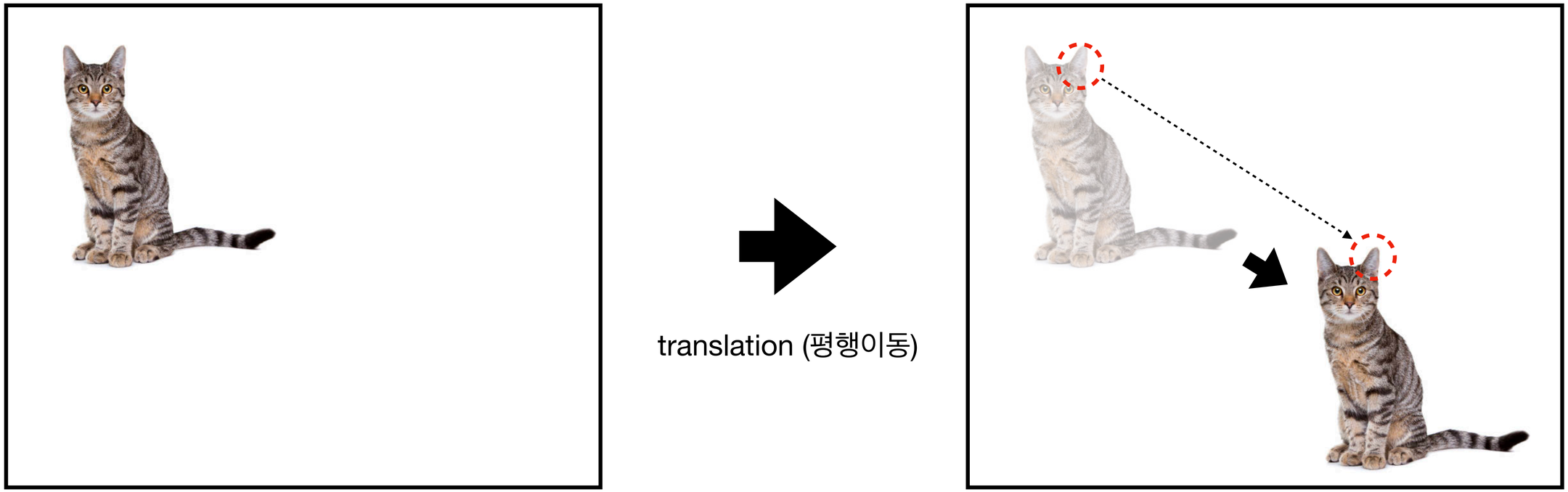
3차원의 tensor of shape == (Height, Width, 3)

Convolution Neural Network

Image Data의 특성

특성 2

“Translation Equivariance”: 이미지가 평행이동 했을때, 이미지의 특징에 해당되는 부분도 동일하게 평행이동한다!



ACADENTIAL

Convolution Neural Network

Image Data의 특성

참고

"Translation Equivariance" ≠ "Translation Invariance"

"어떤 특징이 **Translation Invariant함**"은 **이미지가 평행이동**하여도, **해당 특징은 바뀌지 않는** invariant 하다는 것.

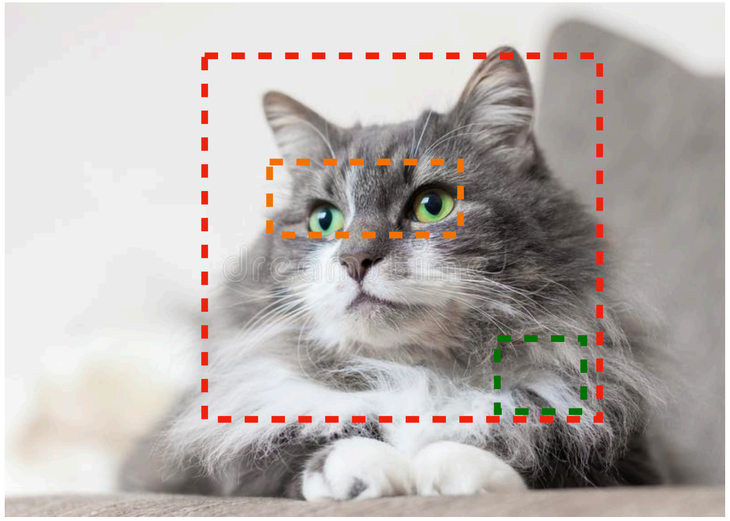
"어떤 특징이 **Translation Equivariant함**"은 **이미지가 평행이동**하였을 때, **해당 특징도 동일하게 평행이동**하는 것.

Convolution Neural Network

Image Data의 특성

특성 3

Hierarchical한 feature: local한 low-level feature에서 global한 higher-level feature로 구성되어 있다.

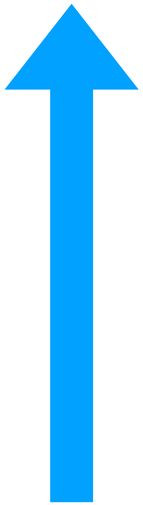


Object

Object parts

Edges and Textures

High-level (abstract), Global features



Low-level, Local features

출처: dreamstime.com

ACADENTIAL

Convolution Neural Network

Image Data의 특성 정리

- 특성 1:
  - Image == 3D tensor
- 특성 2:
  - Translation Equivariant한 feature
- 특성 3:
  - Hierarchical한 feature

CNN은 다음과 같은 이미지의 특성에 적합하다!

ACADENTIAL

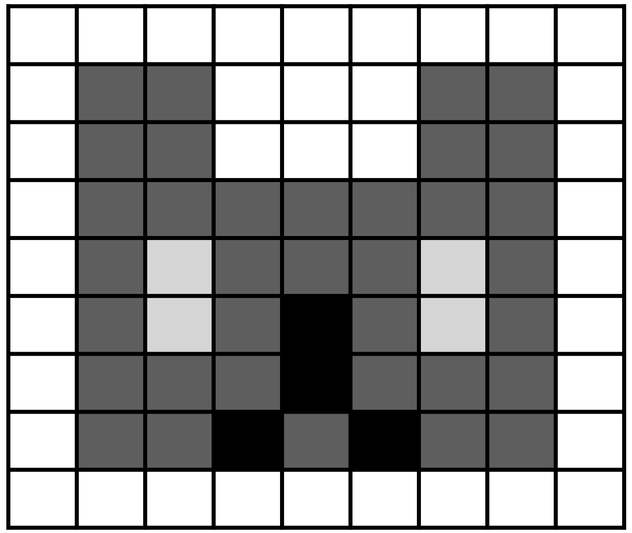
14-2. CNN Layer의 작동 원리

ACADENTIAL

Convolution Neural Network

Convolutional Layer의 작동원리

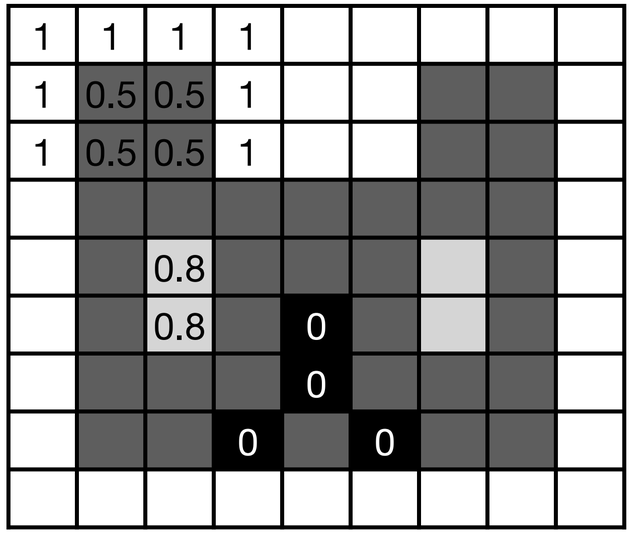
출처: dreamstime.com



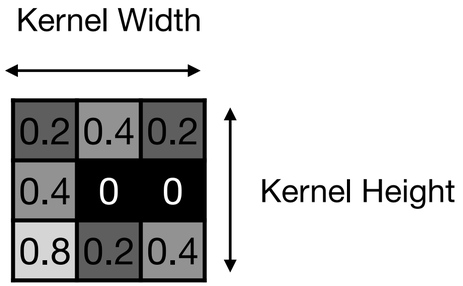
Pixeled Data로 바꿔서 살펴보자. (그림이 별로여도 양해 부탁드려요 ㅋㅋ)

Convolution Neural Network

Convolutional Layer의 작동원리



Input Image

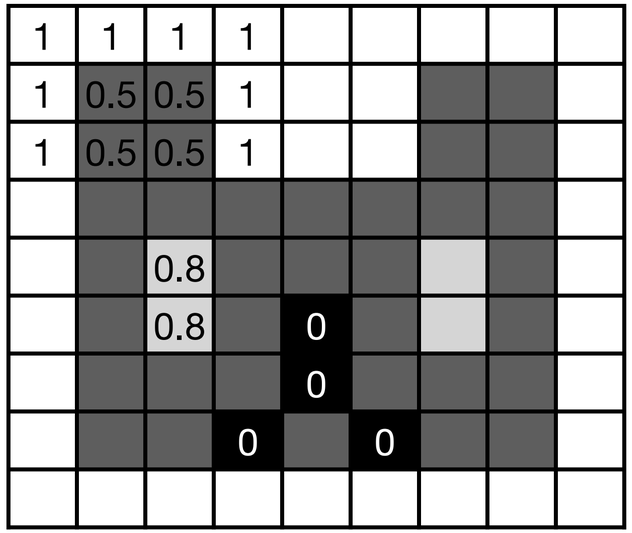


CNN weight $\mathbf{w}$

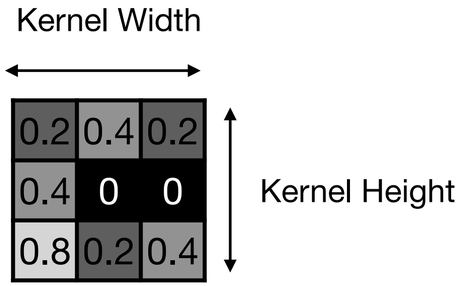
- Convolutional Layer의 weight은 2D tensor (Kernel Width, Kernel Height)로 구성되어 있다.

Convolution Neural Network

Convolutional Layer의 작동원리



Input Image

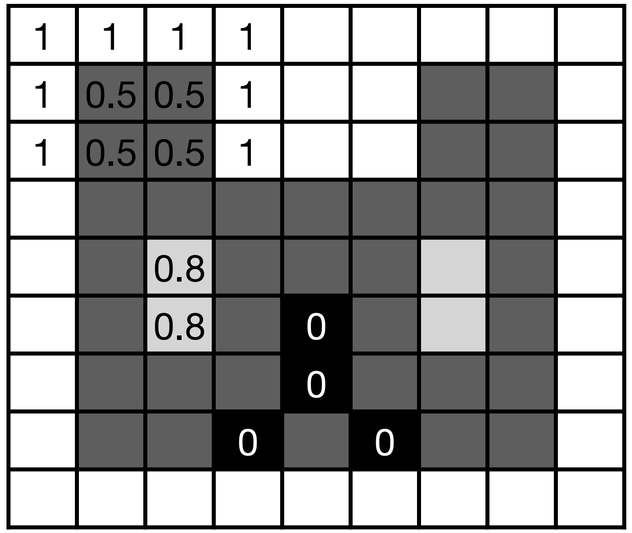


CNN weight $\mathbf{w}$

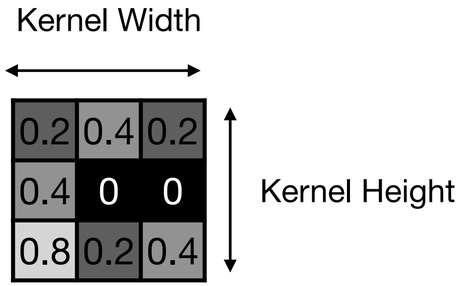
- Convolutional Layer의 weight은 2D tensor (Kernel Width, Kernel Height)로 구성되어 있다.
- 예를 들어 (3, 3)인 위와 같은 Conv. Layer의 weight가 있다고 가정해보자.

Convolution Neural Network

Convolutional Layer의 작동원리



Input Image

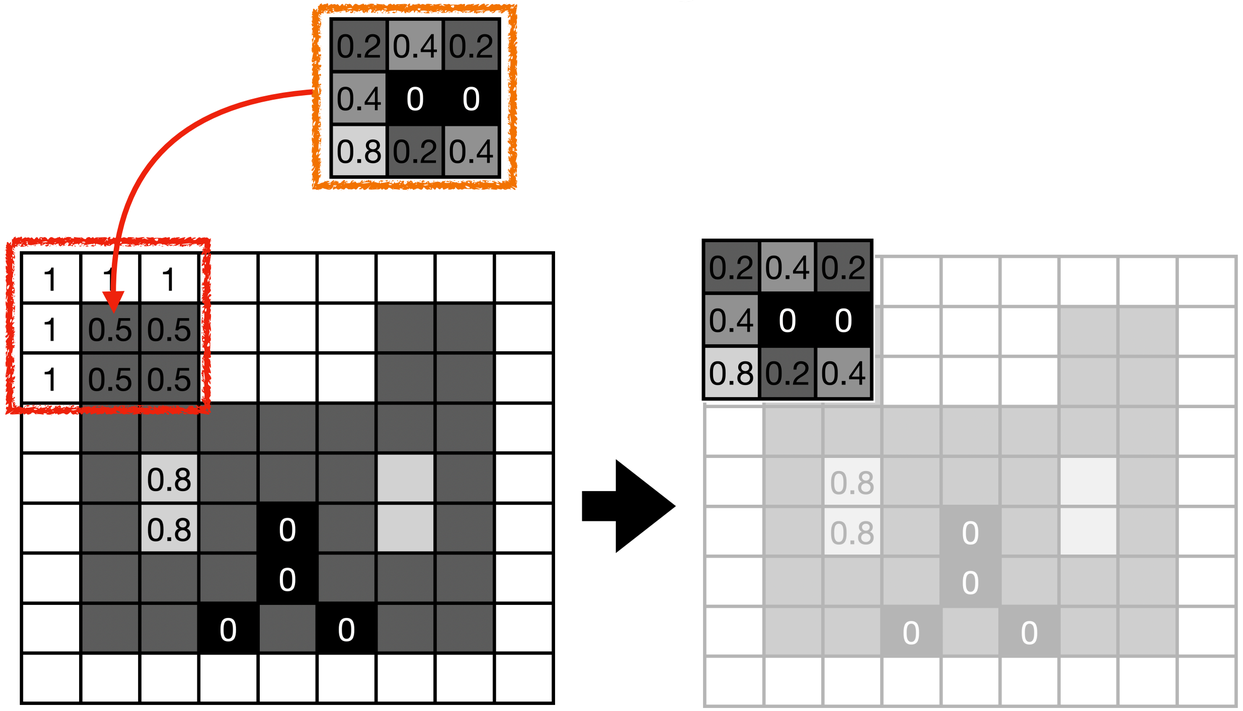


CNN weight $\mathbf{w}$

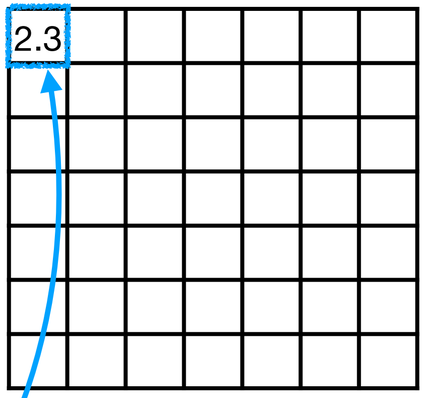
- Convolutional Layer의 weight은 2D tensor (Kernel Width, Kernel Height)로 구성되어 있다.
- 예를 들어 (3, 3)인 위와 같은 Conv. Layer의 weight가 있다고 가정해보자.
- 일반적으로 Convolutional Layer의 Weight은 Kernel 혹은 Kernel Weight이라고도 불린다.

Convolution Neural Network

Convolutional Layer의 작동원리



Convolutional Layer의 출력값



$$\sum \begin{bmatrix} 1 & 1 & 1 \\ 1 & 0.5 & 0.5 \\ 1 & 0.5 & 0.5 \end{bmatrix} \odot \begin{bmatrix} 0.2 & 0.4 & 0.2 \\ 0.4 & 0 & 0 \\ 0.8 & 0.2 & 0.4 \end{bmatrix} = \sum \begin{bmatrix} 0.2 & 0.4 & 0.2 \\ 0.4 & 0 & 0 \\ 0.8 & 0.1 & 0.2 \end{bmatrix} = 2.3$$

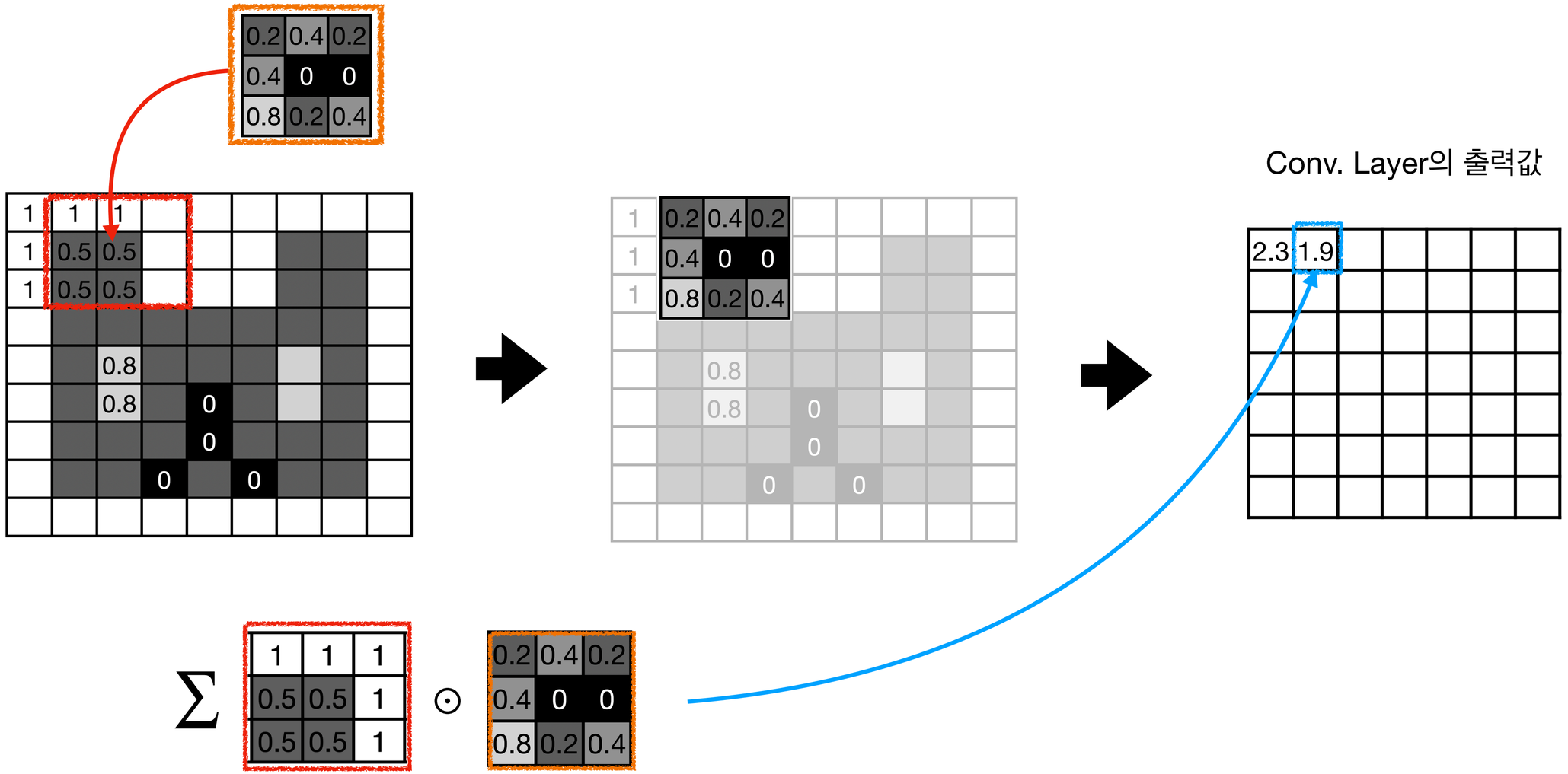
Element-wise하게 서로 곱해서 곱한 값들을 다 더해준다!
즉,

$$(1 \times 0.2) + (1 \times 0.4) + (1 \times 0.2)$$
$$+(1 \times 0.4) + (0.5 \times 0) + (0.5 \times 0)$$
$$+(1 \times 0.8) + (0.5 \times 0.2) + (0.5 \times 0.4)$$
$$= 2.3$$

ACADENTIAL

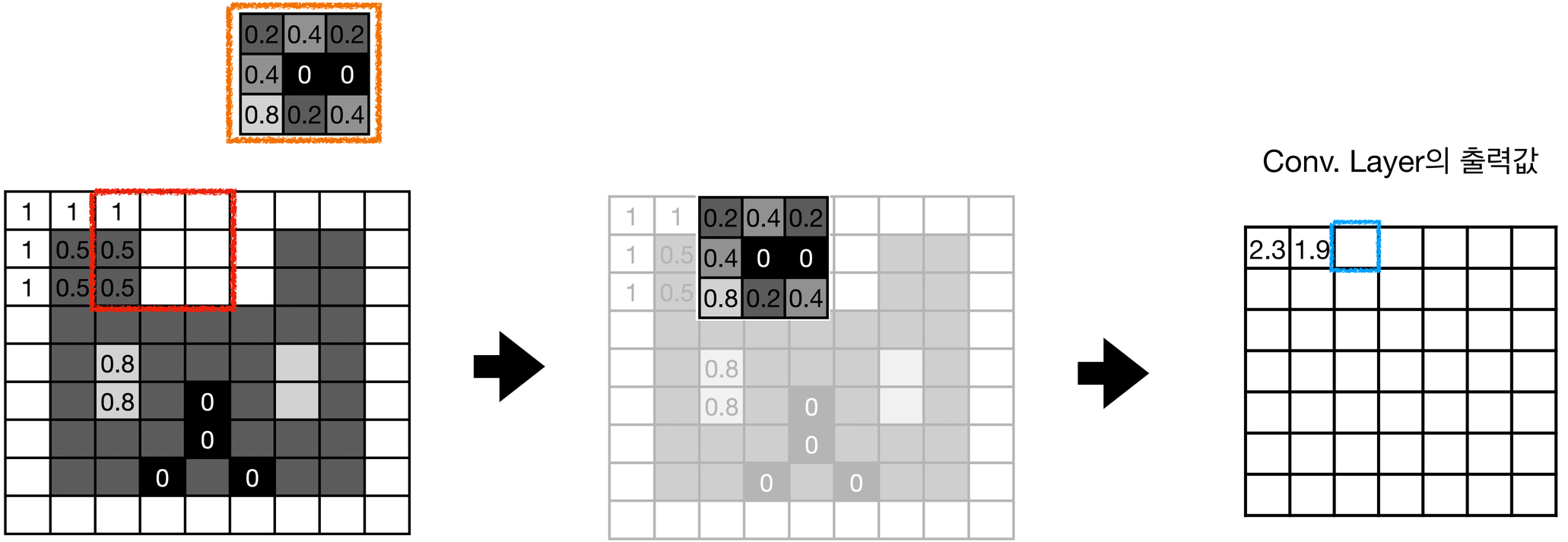
Convolution Neural Network

Convolutional Layer의 작동원리



Convolution Neural Network

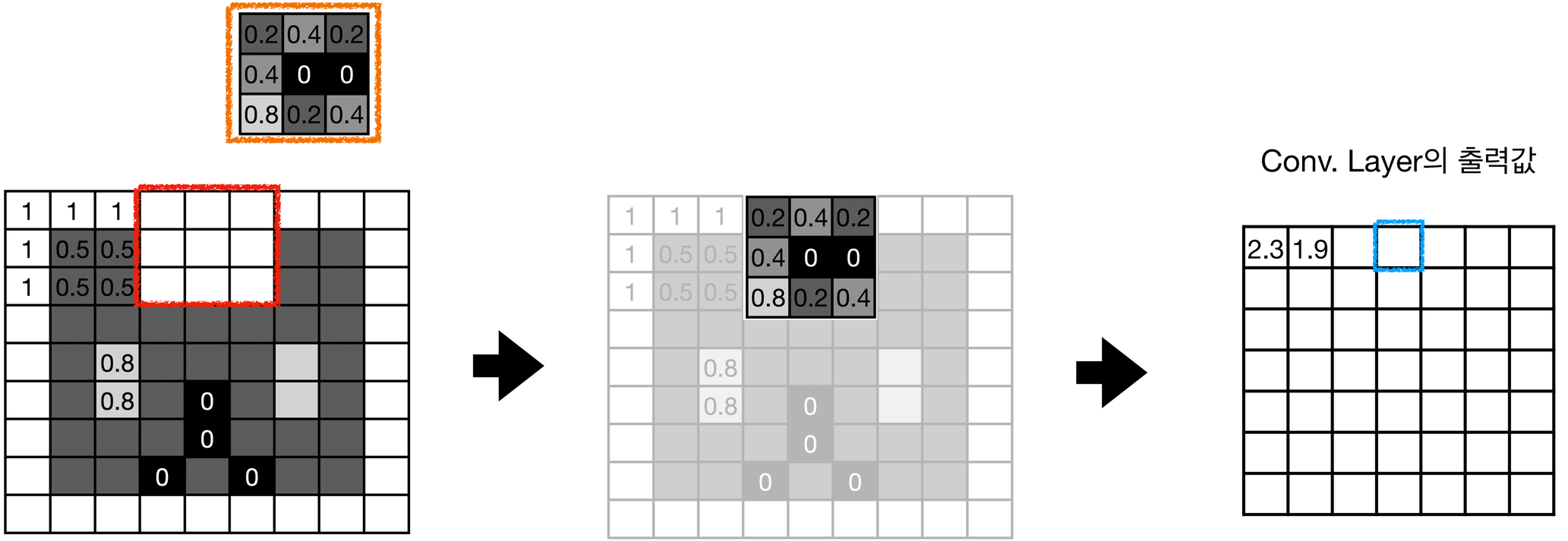
Convolutional Layer의 작동원리



다음 window에 대해서도 차례대로 구해준다.

Convolution Neural Network

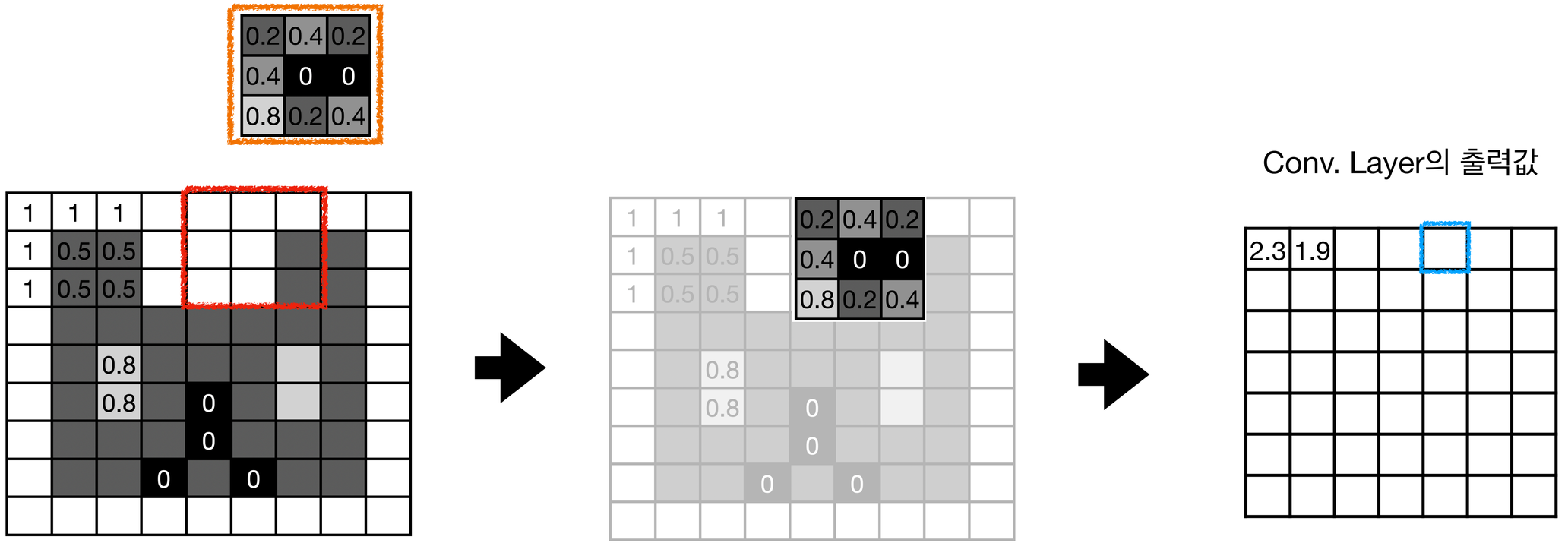
Convolutional Layer의 작동원리



다음 window에 대해서도 차례대로 구해준다. (반복)

Convolution Neural Network

Convolutional Layer의 작동원리

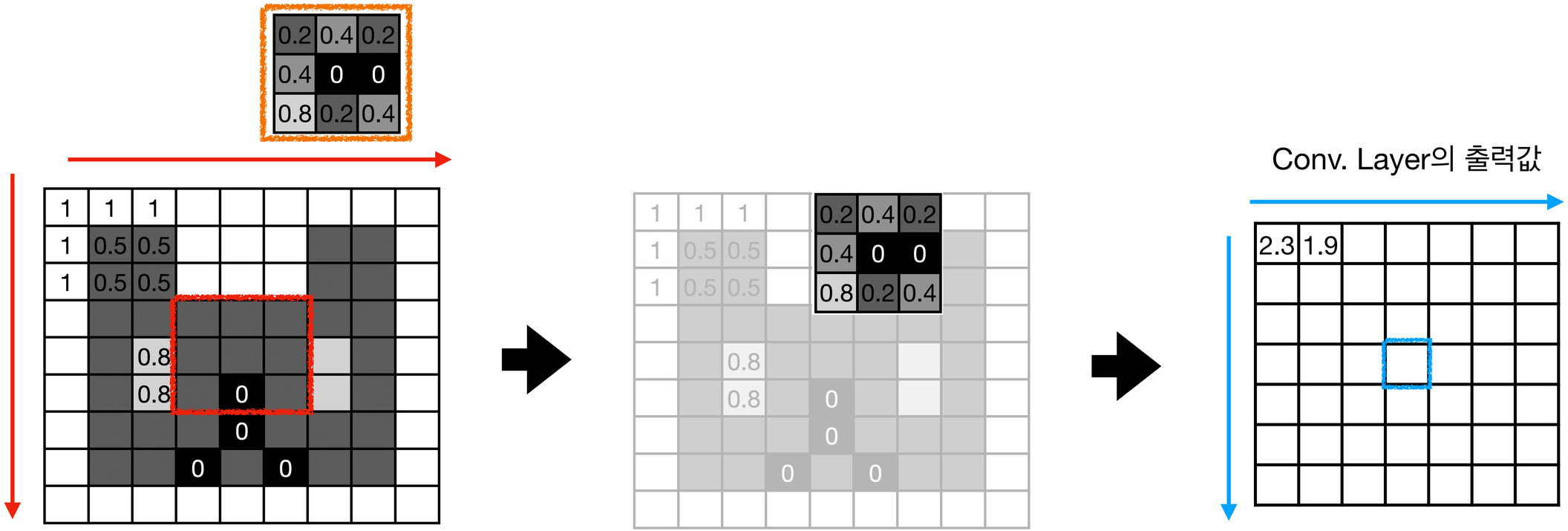


다음 window에 대해서도 차례대로 구해준다. (반복)

ACADENTIAL

Convolution Neural Network

Convolutional Layer의 작동원리



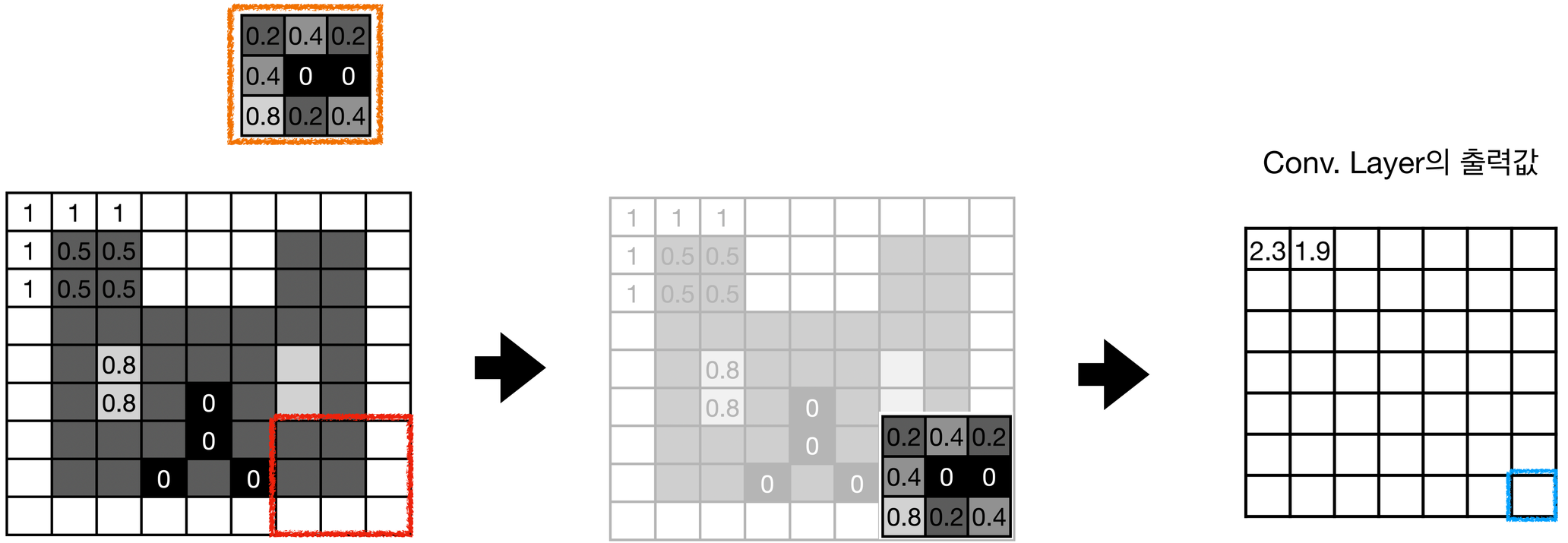
다음 window에 대해서도 차례대로 구해준다. (반복)

위 과정을 height랑 width에 걸쳐서 수행하여 Conv. layer의 출력 값을 구한다!

ACADENTIAL

Convolution Neural Network

Convolutional Layer의 작동원리



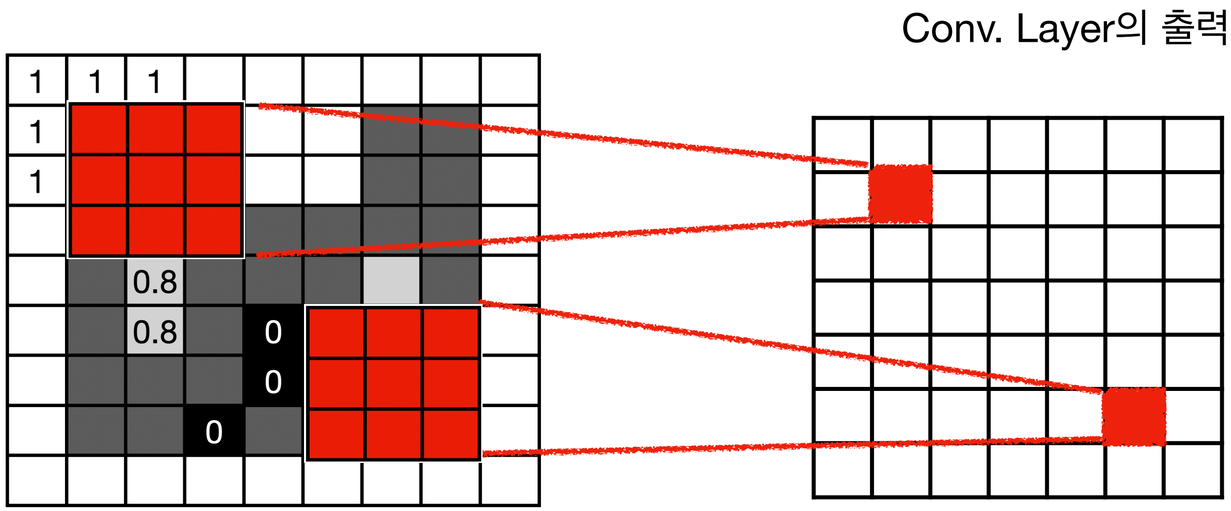
다음 window에 대해서도 차례대로 구해준다. (반복)

위 과정을 height랑 width에 걸쳐서 수행하여 Conv. Layer의 출력 값을 구한다!

ACADENTIAL

Convolution Neural Network

Convolutional Layer의 작동원리



$$y = \mathbf{w} * \mathbf{x} + b$$

* == convolution operation

Conv. Layer은 Translation Equivariance을 만족시킨다!

ACADENTIAL

Convolution Neural Network

Multiple input, output channels

Multiple Input channels

- 앞에서 살펴본 예제: Grey-colored image (즉, input channel == 1)
- 만약에 RGB-colored image가 input이라면? (즉, input channel == 3)

ACADENTIAL

Convolution Neural Network

Multiple input, output channels

Multiple Input channels

- 앞에서 살펴본 예제: Grey-colored image (즉, input channel == 1)
- 만약에 RGB-colored image가 input이라면? (즉, input channel == 3)

Multiple Output channels

- 앞에서 살펴본 예제: CNN layer의 출력값은 하나의 channel로만 구성됨. (즉 output channel == 1)
- 만약에 output channel > 1로 구성하려면?

ACADENTIAL

Convolution Neural Network

Multiple input, output channels

Multiple Input channels

- 앞에서 살펴본 예제: Grey-colored image (즉, input channel == 1)
- 만약에 RGB-colored image가 input이라면? (즉, input channel == 3)

Multiple Output channels

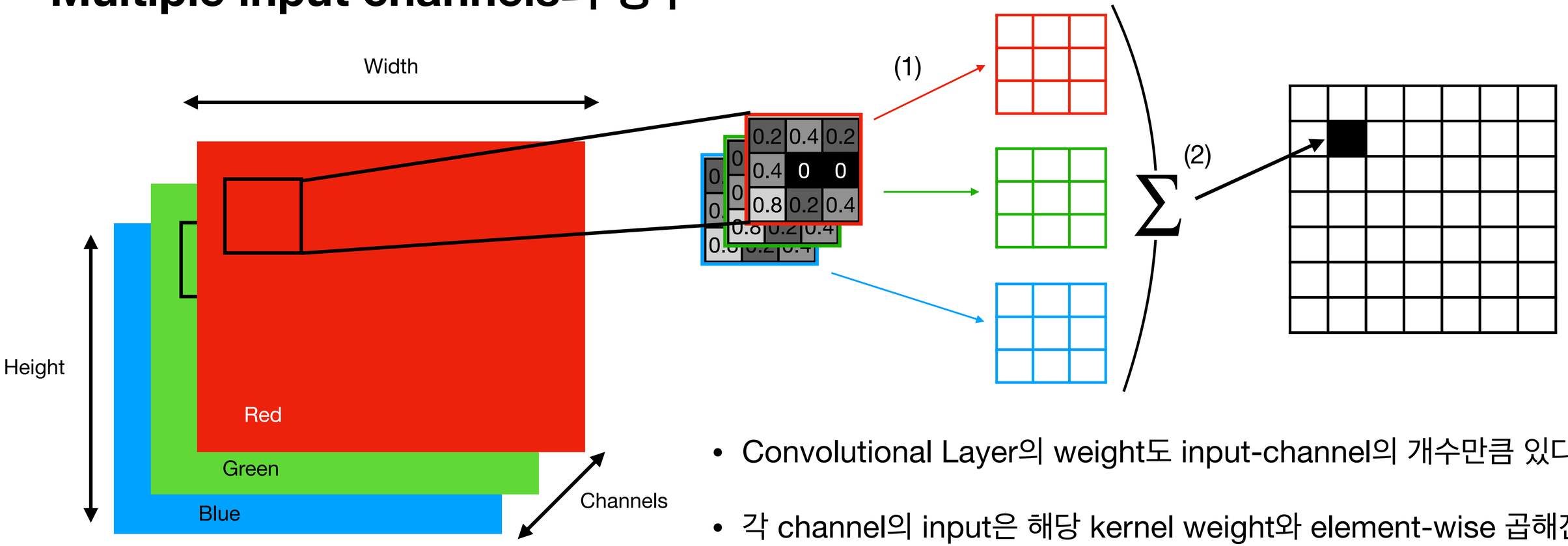
- 앞에서 살펴본 예제: CNN layer의 출력값은 하나의 channel로만 구성됨. (즉 output channel == 1)
- 만약에 output channel > 1로 구성하려면?

즉, extension to multiple input channels and output channels!

ACADENTIAL

Convolution Neural Network

Multiple input channels의 경우

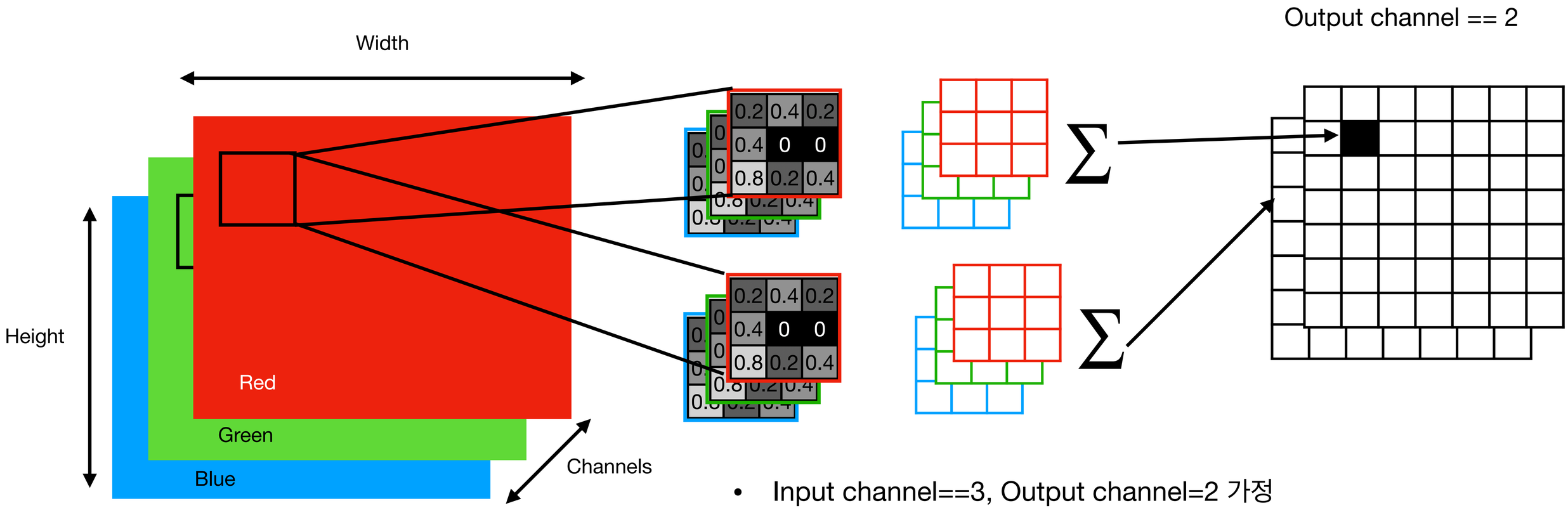


3차원의 tensor of shape == (Height, Width, Channel)

- Convolutional Layer의 weight도 input-channel의 개수만큼 있다.
- 각 channel의 input은 해당 kernel weight와 element-wise 곱해진다. (1)
- 각 channel마다 element-wise 곱해진 값들을 다 더해준다. (2)

Convolution Neural Network

Multiple output channels의 경우



Input channel == 3

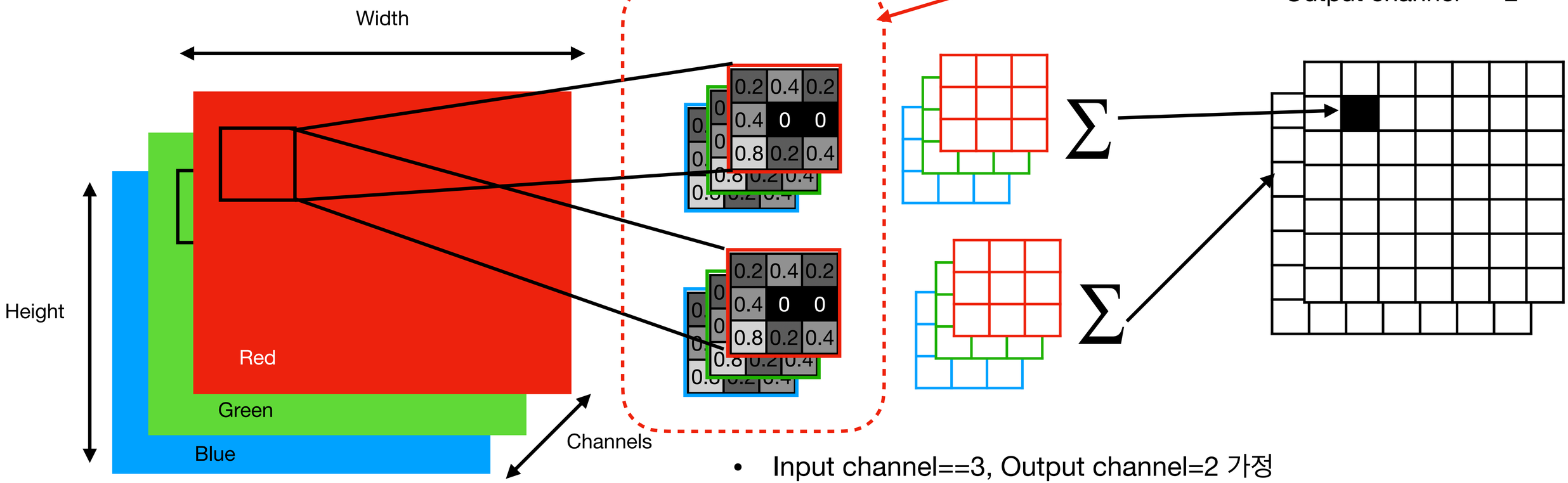
- Input channel==3, Output channel=2 가정
- 동일한 output channel에 해당되는 feature들끼리 $\sum$ 한다 (즉, 더해준다)

Convolution Neural Network

Multiple output channels의 경우

Conv. Kernel의 Weight
= (H, W, In-Channel, Out-Channel)

Output channel == 2

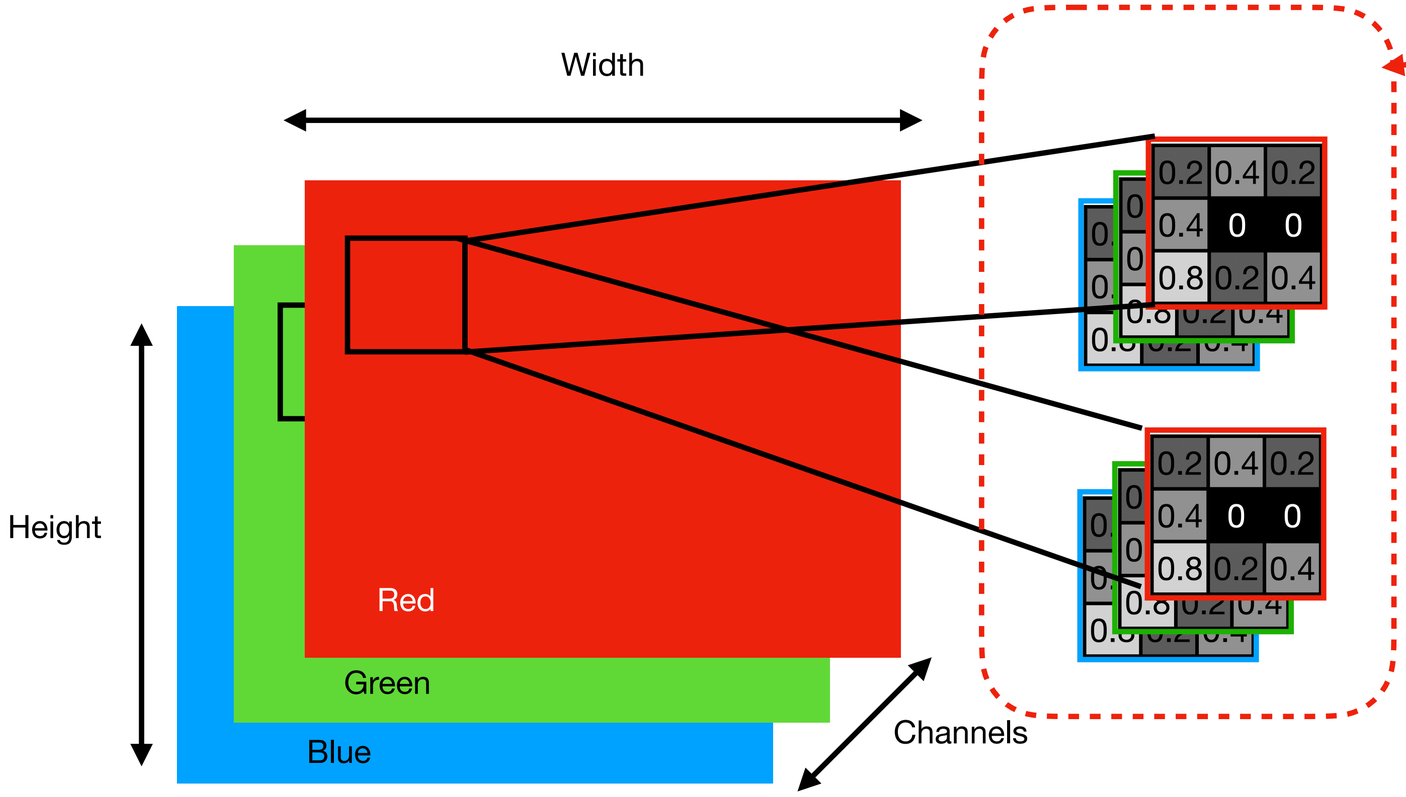


Input channel == 3

- Input channel==3, Output channel=2 가정
- 동일한 output channel에 해당되는 feature들끼리 $\sum$ 한다 (즉, 더해준다)

Convolution Neural Network

Multiple output channels의 경우



Input channel == 3

Conv. Kernel의 Weight
= $(H_K, W_K, C_{in}, C_{out})$

참고 사항:

- **Conv. Layer의 weight의 shape**은
- $(H_K, W_K, C_{in}, C_{out})$ 가 된다.
- H_K = Kernel Height, W_K= Kernel Width
- C_{in} = in channel, C_{out} = out channel.

즉, weight parameter의 개수 = $H_K \cdot W_K \cdot C_{in} \cdot C_{out}$

ACADENTIAL

14-3. Dilated Operation

ACADENTIAL

Convolution Neural Networks

Variants of Convolutional layer

Variants:

- Dilated operation
- Strided operation
- Padding
- Pooling

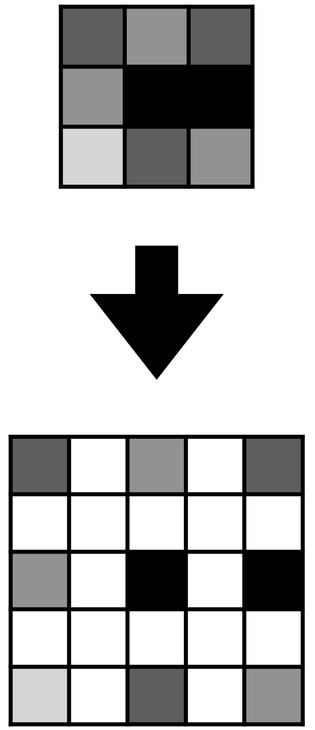
ACADENTIAL

Convolution Neural Networks

Dilated Operation

- Dilation: 기존의 kernel을 **“팽창” / “확장”** 시키는 것.
- “Dilation = n”은 원래 Conv. Kernel을 n배 확장시킨다는 의미이다!

Dilation=1 (default, 기존의 Conv. Kernel)



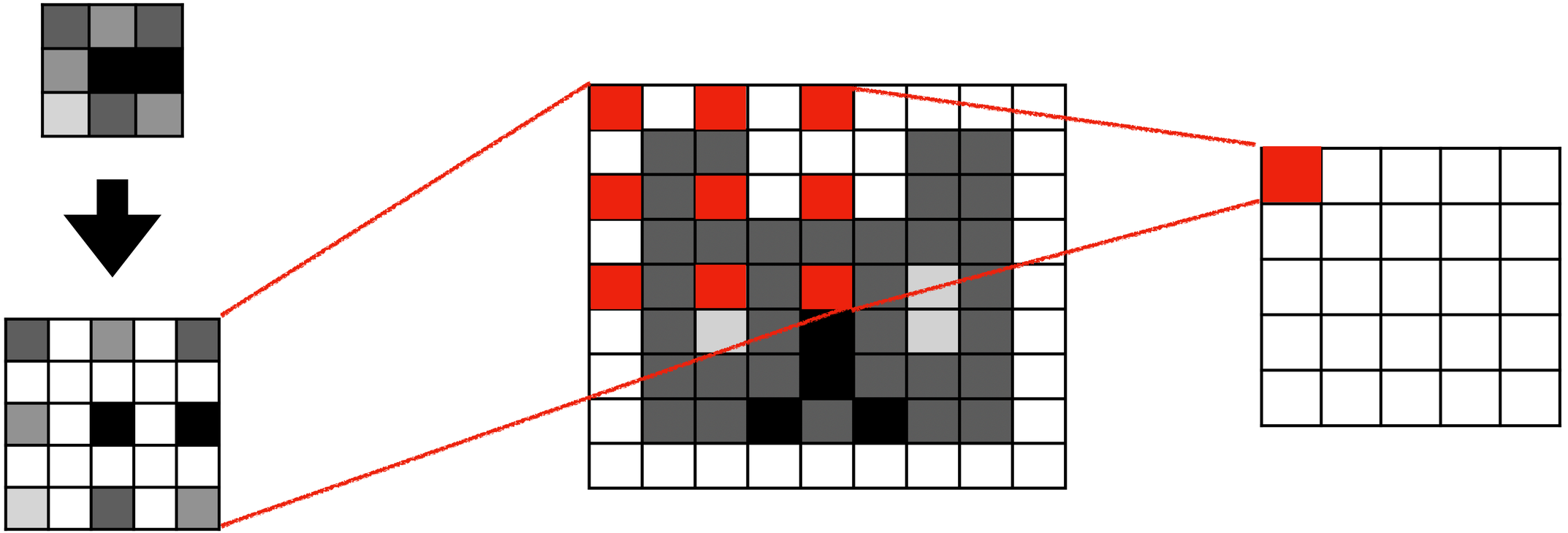
Dilation=2 (2배로 확장시킨 것)

ACADENTIAL

Convolution Neural Networks

Dilated Operation

Dilation=1 (default, 기존의 CNN kernel)



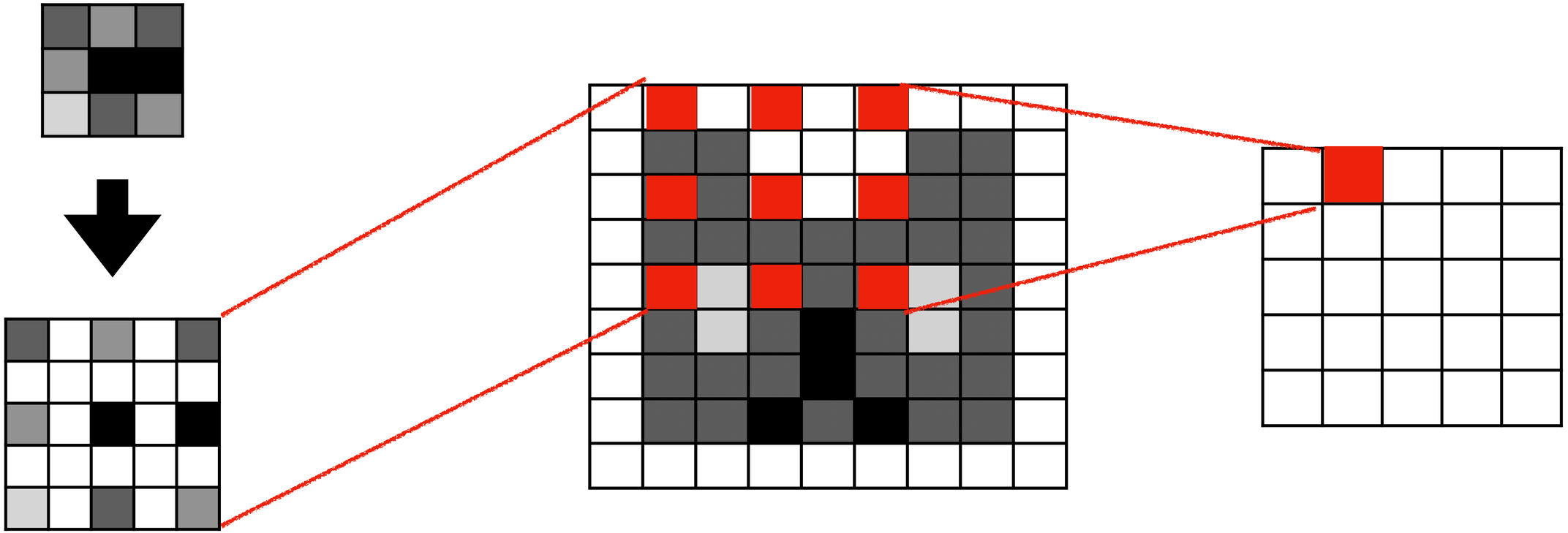
Dilation=2 (2배로 확장시킨 것)

ACADENTIAL

Convolution Neural Networks

Dilated Operation

Dilation=1 (default, 기존의 CNN kernel)



Dilation=2 (2배로 확장시킨 것)

ACADENTIAL

14-4. Strided Operation

ACADENTIAL

Convolution Neural Networks

Strided Operation

Stride: Conv. Kernel이 이미지 상에서 **“몇 칸 씩”** 이동하는지 조절하는 값.

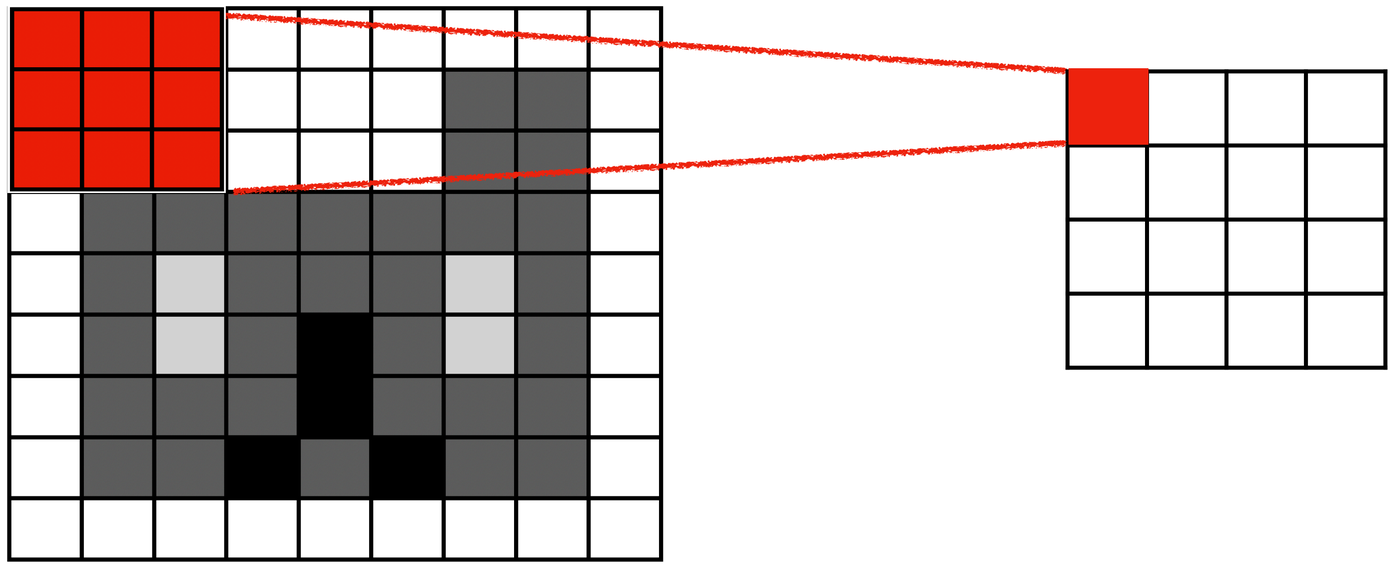
- Stride == 1: 한 칸 씩 이동
- Stride == n: n 칸 씩 이동

ACADENTIAL

Convolution Neural Networks

Strided Operation

예를 들어 **Stride == 2**을 살펴보자

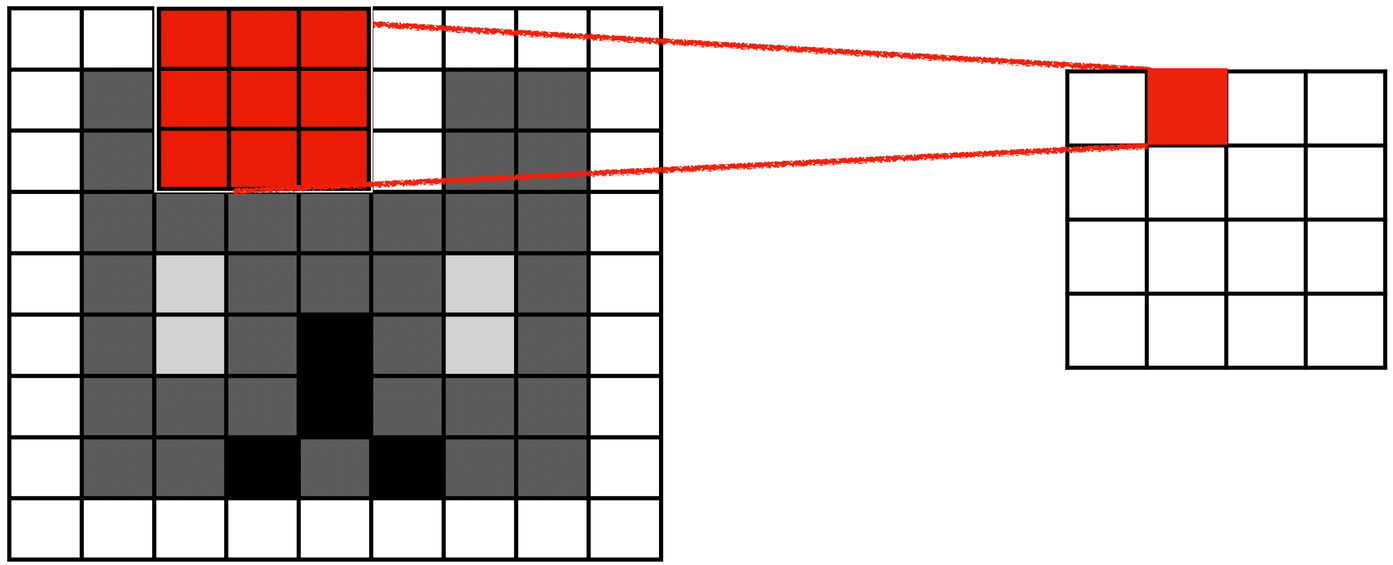


ACADENTIAL

Convolution Neural Networks

Strided Operation

예를 들어 **Stride == 2**을 살펴보자

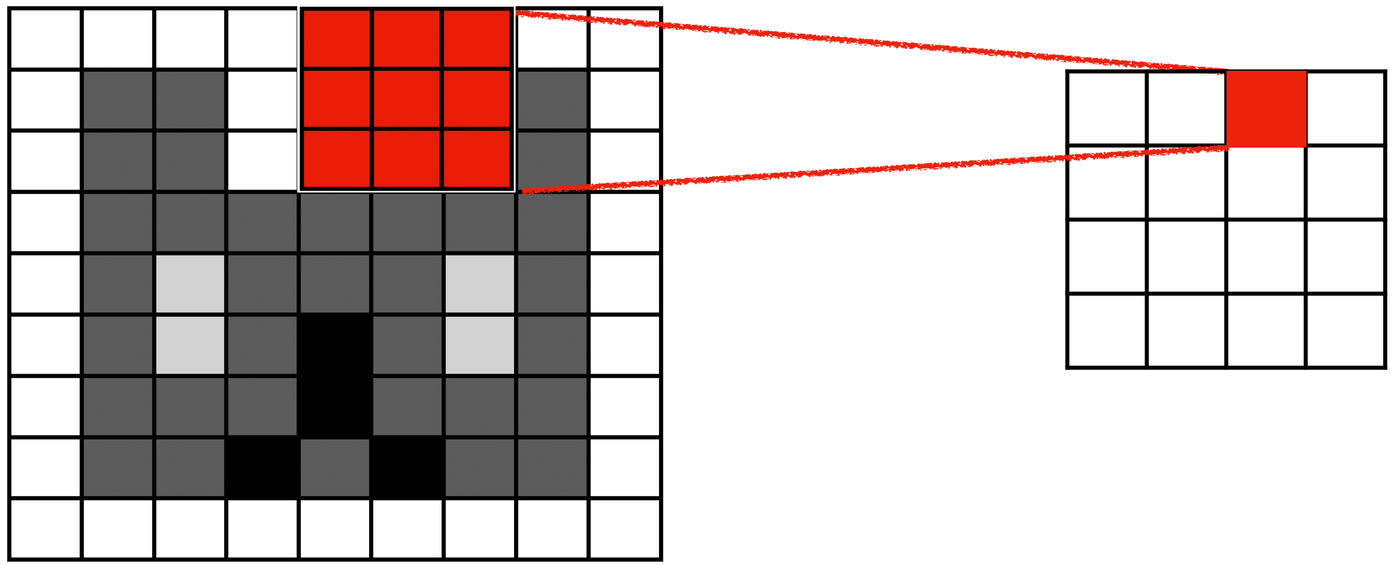


ACADENTIAL

Convolution Neural Networks

Strided Operation

예를 들어 **Stride == 2**을 살펴보자

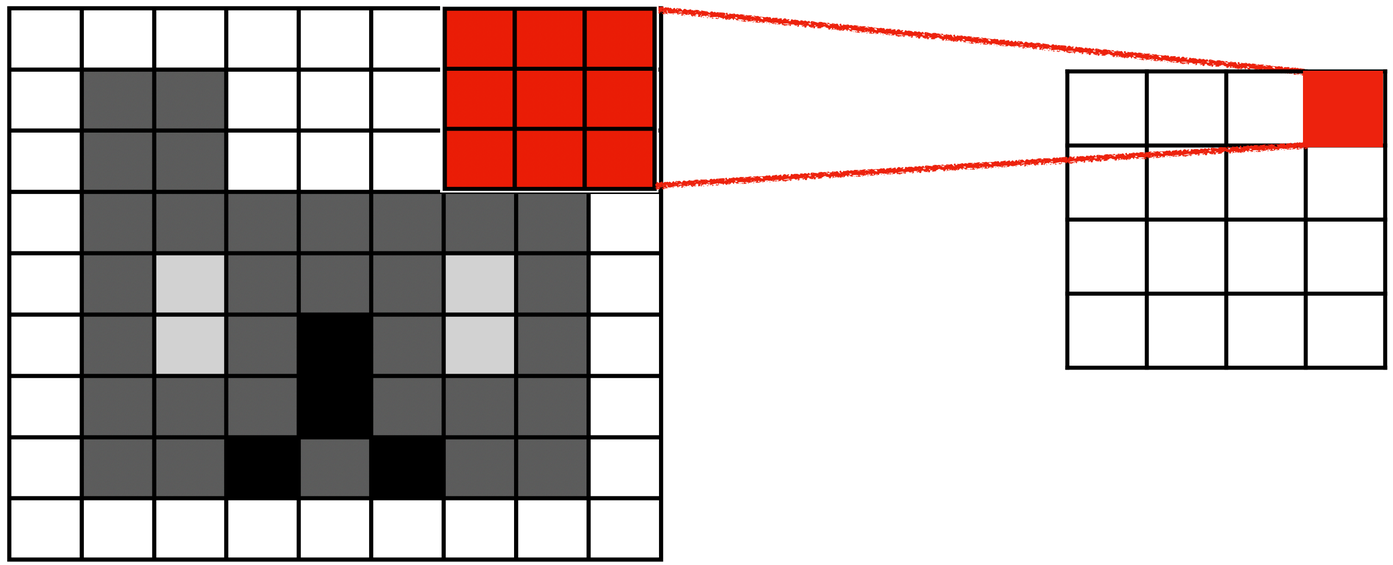


ACADENTIAL

Convolution Neural Networks

Strided Operation

예를 들어 **Stride == 2**을 살펴보자



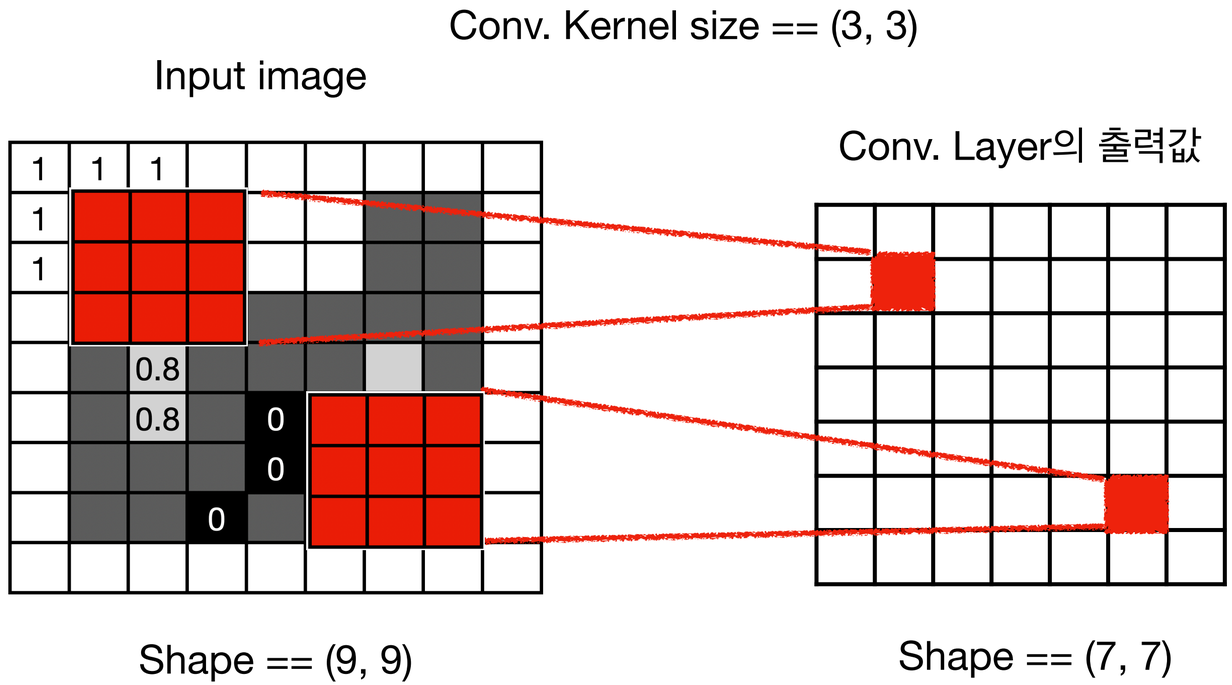
ACADENTIAL

14-5. Padding

Convolution Neural Networks

Padding

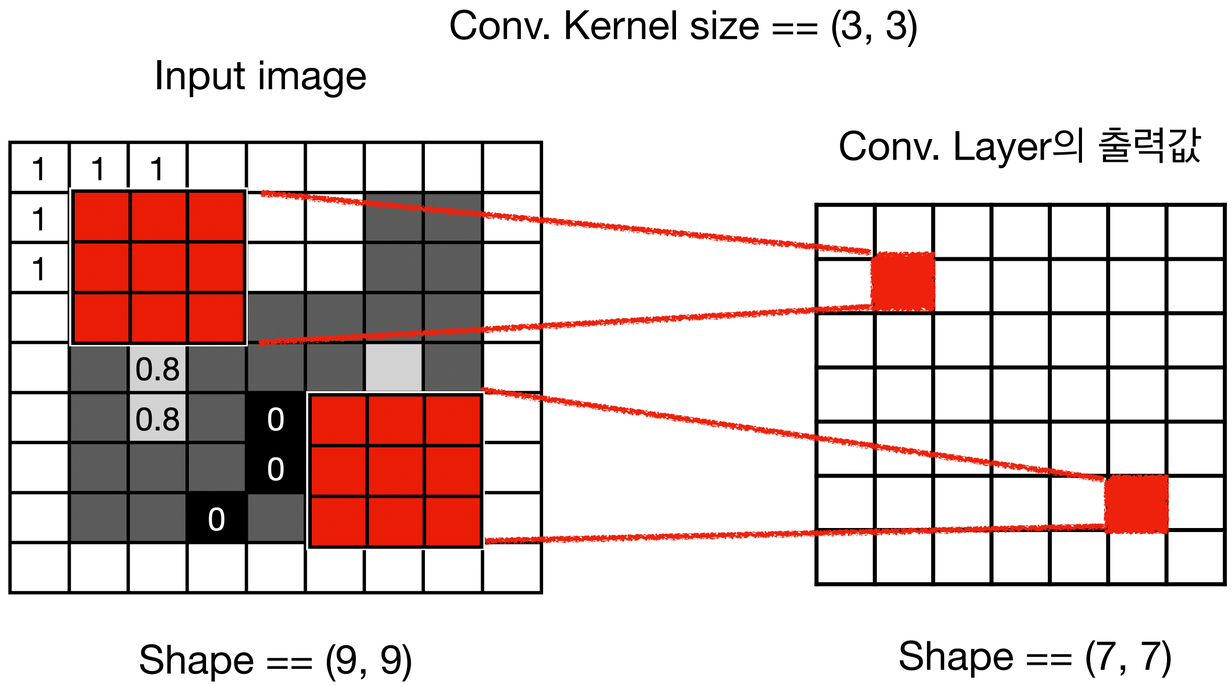
- 앞에서 보았다시피, Conv. Layer의 출력 값은 input image의 크기를 줄인다.



Convolution Neural Networks

Padding

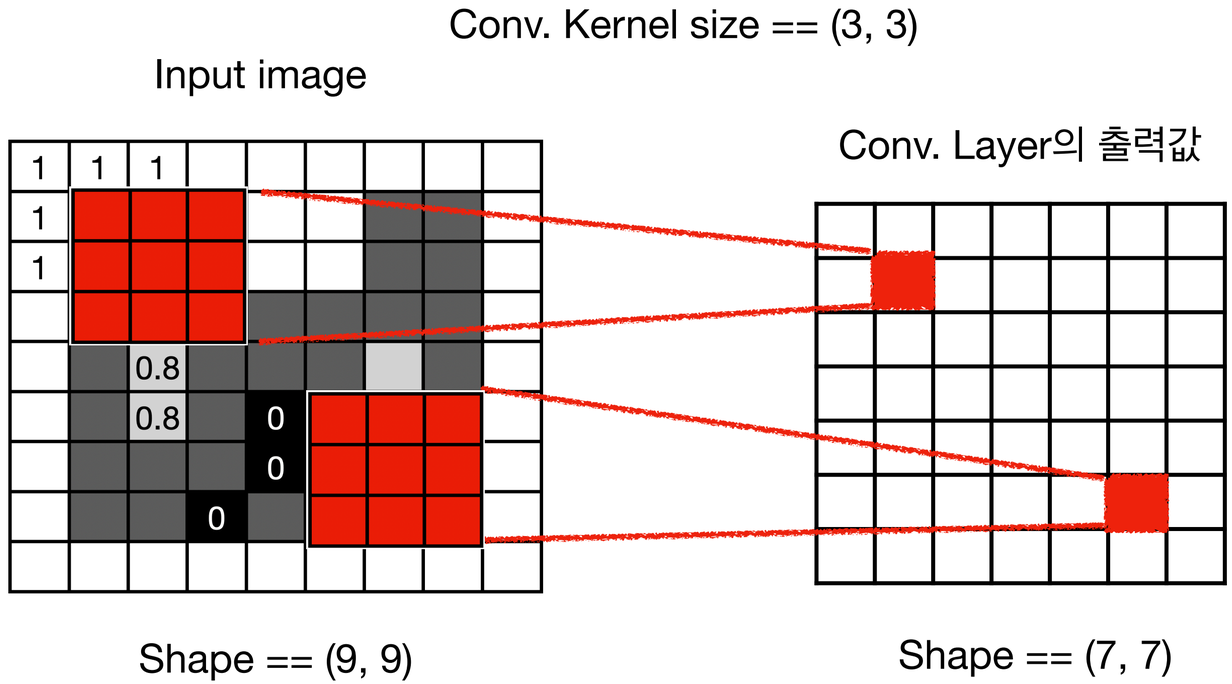
- 앞에서 보았다시피, Conv. Layer의 출력 값은 input image의 크기를 줄인다.
- (3, 3) 크기의 kernel 사용시 input image의 shape이 (9, 9) → (7, 7)로 줄어들었다.



Convolution Neural Networks

Padding

- 앞에서 보았다시피, Conv. Layer의 출력 값은 input image의 크기를 줄인다.
- (3, 3) 크기의 kernel 사용시 input image의 shape이 (9, 9) → (7, 7)로 줄어들었다.
- 만약에 줄어드는 것을 방지하려면?

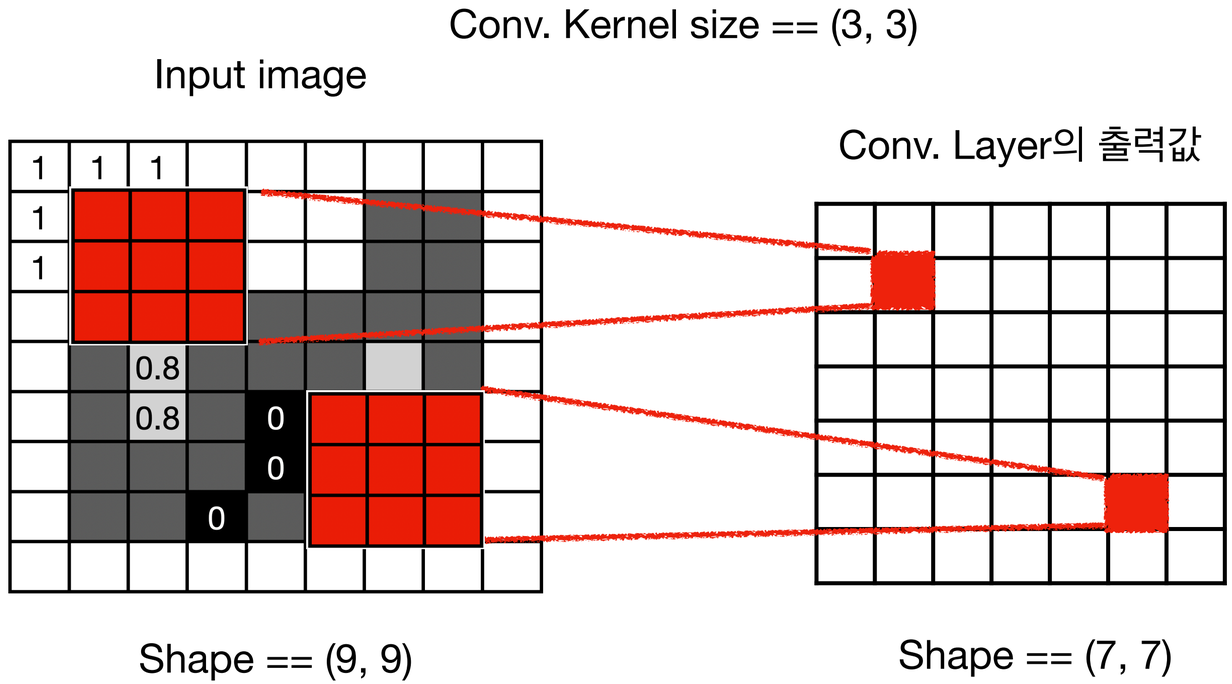


Convolution Neural Networks

Padding

- 앞에서 보았다시피, Conv. Layer의 출력 값은 input image의 크기를 줄인다.
- (3, 3) 크기의 kernel 사용시 input image의 shape이 (9, 9) → (7, 7)로 줄어들었다.
- 만약에 줄어드는 것을 방지하려면?

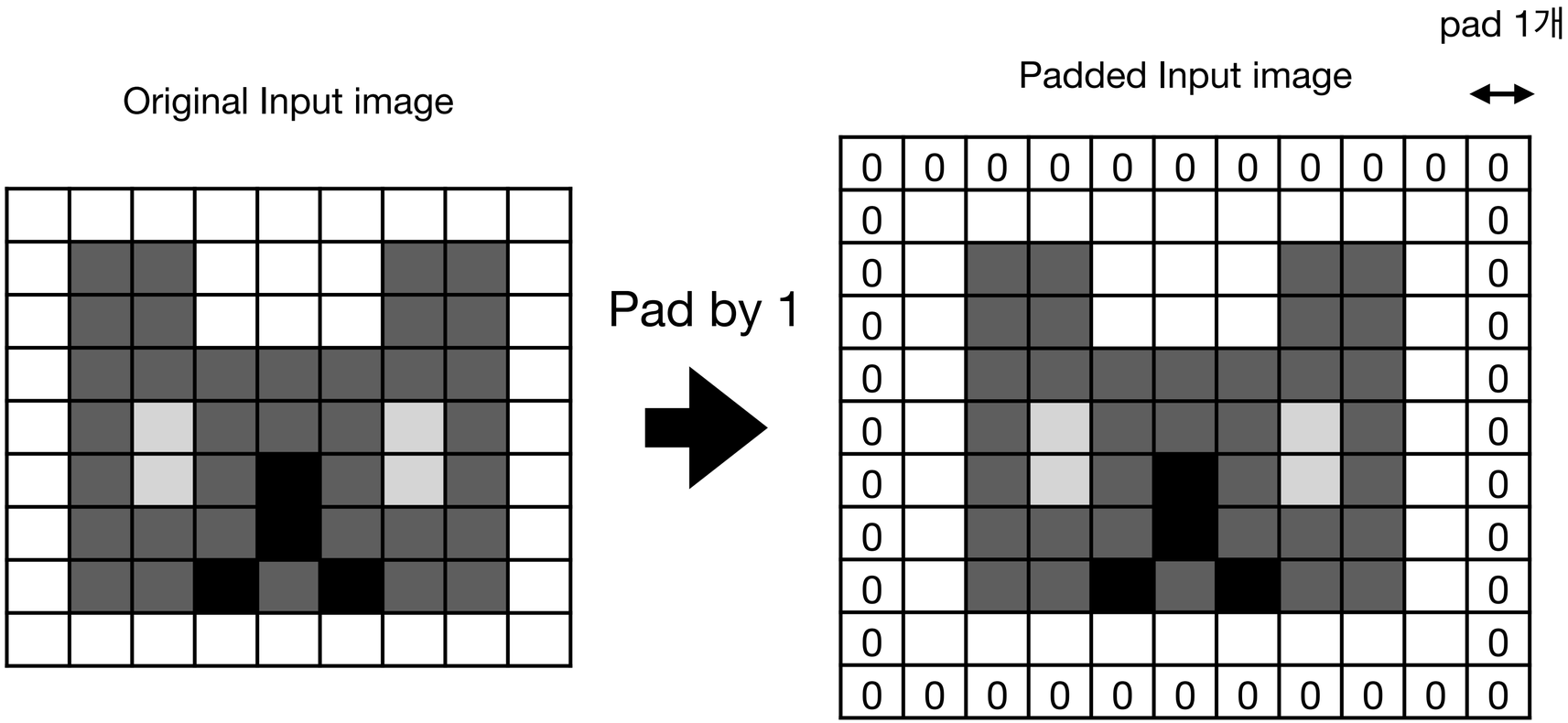
Padding



Convolutional Neural Networks

Padding

- Padding: Image에 테두리에 0을 추가함. (일반적으로 zero-padding을 사용하나 다른 방법의 padding도 존재함)

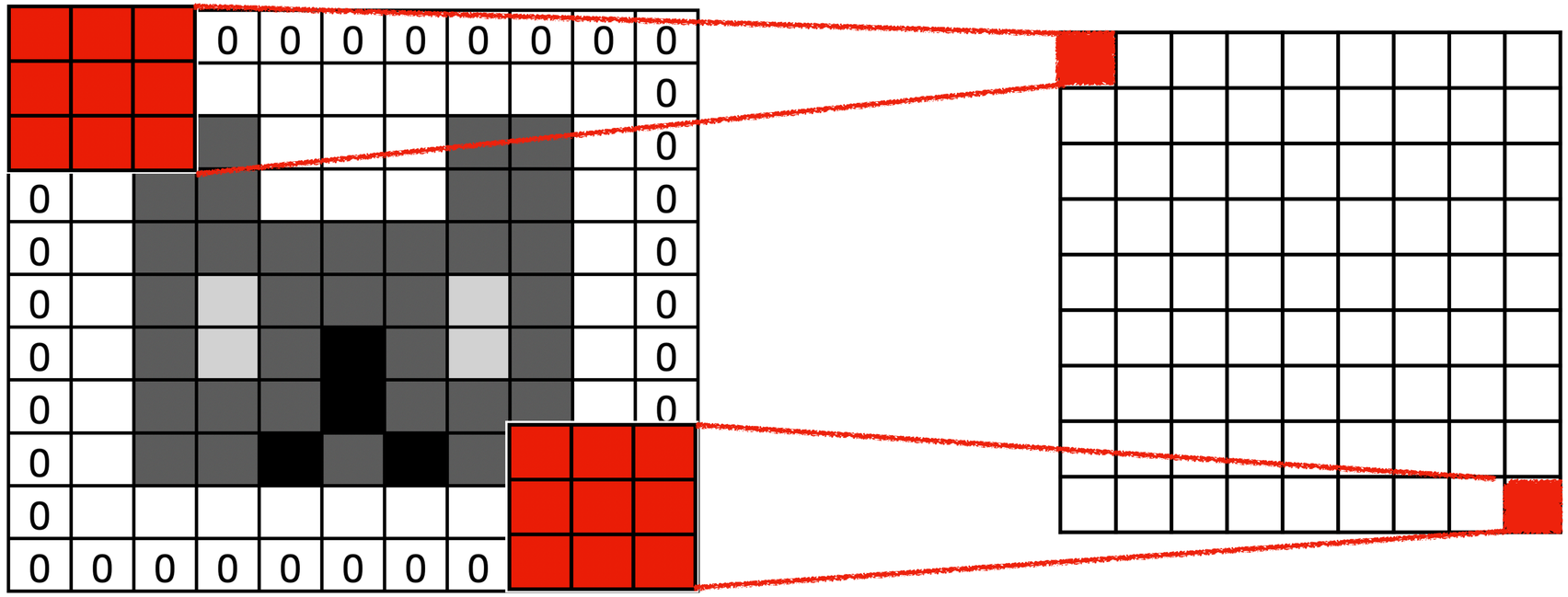


Convolutional Neural Networks

Padding

pad 1개

Padded Input image ↔



Padded image shape == (11, 11)

출력된 image shape == (9, 9)

original image shape 와 동일하다!

ACADENTIAL

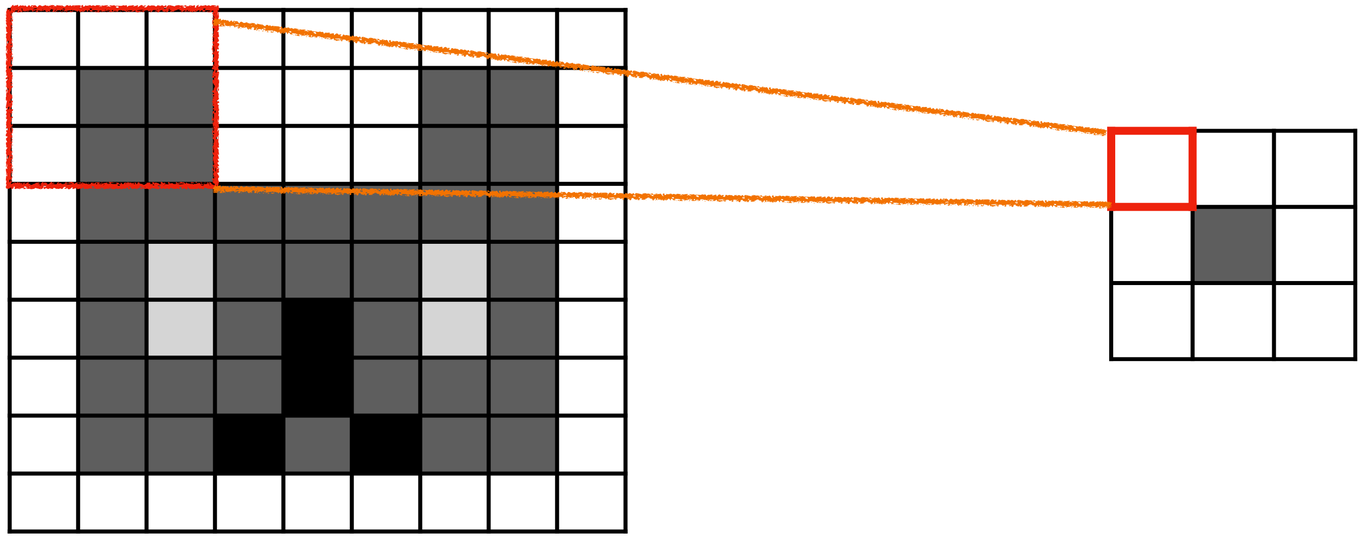
14-6. Pooling

Convolutional Neural Networks

Pooling

- Pooling: 이미지의 크기를 축소하기 위해서 (즉, 해상도를 낮추기 위해서) window의 mean 혹은 max 값으로 뽑는 것.

Max Pooling with (3, 3) window size



ACADENTIAL

Convolutional Neural Networks

Pooling을 하는 이유

1. Feature size을 줄이기 위해서
2. 더 큰 Receptive Field 확보.
3. Global한 특징을 더 잘 추출하기 위해서

ACADENTIAL

Convolutional Neural Networks

Pooling을 하는 이유

(1) Feature size을 줄이기 위해서

- 예시: 이미지 분류 문제의 경우, 최종 출력값은 1차원의 Logit 벡터이다.
- 이미지는 Conv. Layer와 Pooling Layer을 교차 통과
- 이미지 라벨과 관련된 추상적인 feature들을 추려냄
- 압축된 형태의 feature를 마지막 Conv. Layer에서 출력
- Feature size을 줄이면서 더 추상적이고 압축된 형태의 feature를 구하고자 Pooling을 사용하는 것!

Convolutional Neural Networks

Pooling을 하는 이유

(2) 더 큰 Receptive Field 확보

- Receptive Field = 출력된 feature map에서 pixel 하나에 영향을 미치는 입력 feature pixel들의 개수 (window 크기)
- Pooling 사용시, 입력받는 Convolutional Layer은 더 큰 영역을 한번에 보는 셈
- Pooling 사용하지 않을시, 입력받는 Convolutional Layer은 더 작은, local한 영역을 보는 셈

Convolutional Neural Networks

Pooling을 하는 이유

(3) Global한 특징을 더 잘 추출하기 위해서

- CNN에서 출력 Layer에 더 가까운 Convolutional Layer일수록 더 abstract한, 더 global한 feature을 학습하게 됨.
- 뒤에 위치한 Convolutional Layer일수록 더 큰 Receptive Field을 가져야 더 Global한 특징을 파악하게 됨

Convolution Neural Network

Input and Output shape of CNN

- Input image shape (H_{in}, W_{in}), Stride (S_H, S_W), Dilation (D_H, D_W), Padding (P_H, P_W) Convolution Kernel (K_H, K_W)라고 가정했을시,

- Conv. Layer의 Output shape (H_{out}, W_{out})은 다음과 같다:

$$H_{out} = \left\lfloor \frac{H_{in} + 2P_H - D_H(K_H - 1) - 1}{S_H} + 1 \right\rfloor$$

내림 함수 $\lfloor x \rfloor$: x 보다 작으면서 가장 큰 정수

$$W_{out} = \left\lfloor \frac{W_{in} + 2P_W - D_W(K_W - 1) - 1}{S_W} + 1 \right\rfloor$$

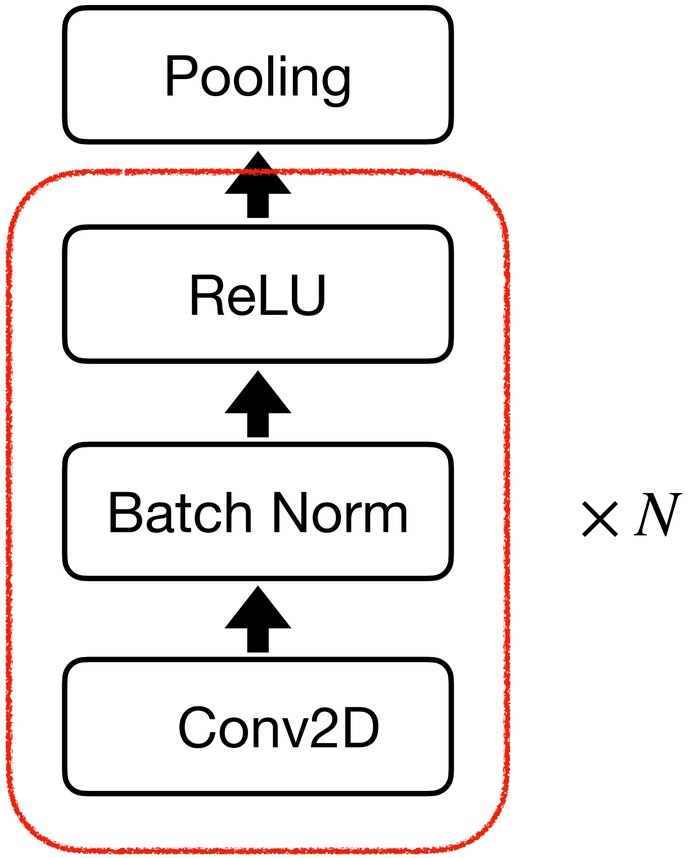
전혀 외울 필요없다!
하지만 이해하고 넘어가기!

Convolutional Neural Networks

Stack of CNN (구성)

일반적으로 Conv2d → Batch Norm → ReLU → Pooling으로 구성되어 있다.

“Conv2d → Batch Norm → ReLU”가 하나의 Block으로 구성되어 몇 번 반복된후 Pooling 되기도 한다.



Conv2D == 2D Conv. Layer

ACADENTIAL

14-7. PyTorch로 구현해보는 Convolutional Layer 및 CNN 모델

ACADENTIAL

14-8. 대표적인 CNN 모델들 소개

ACADENTIAL

Convolutional Neural Network

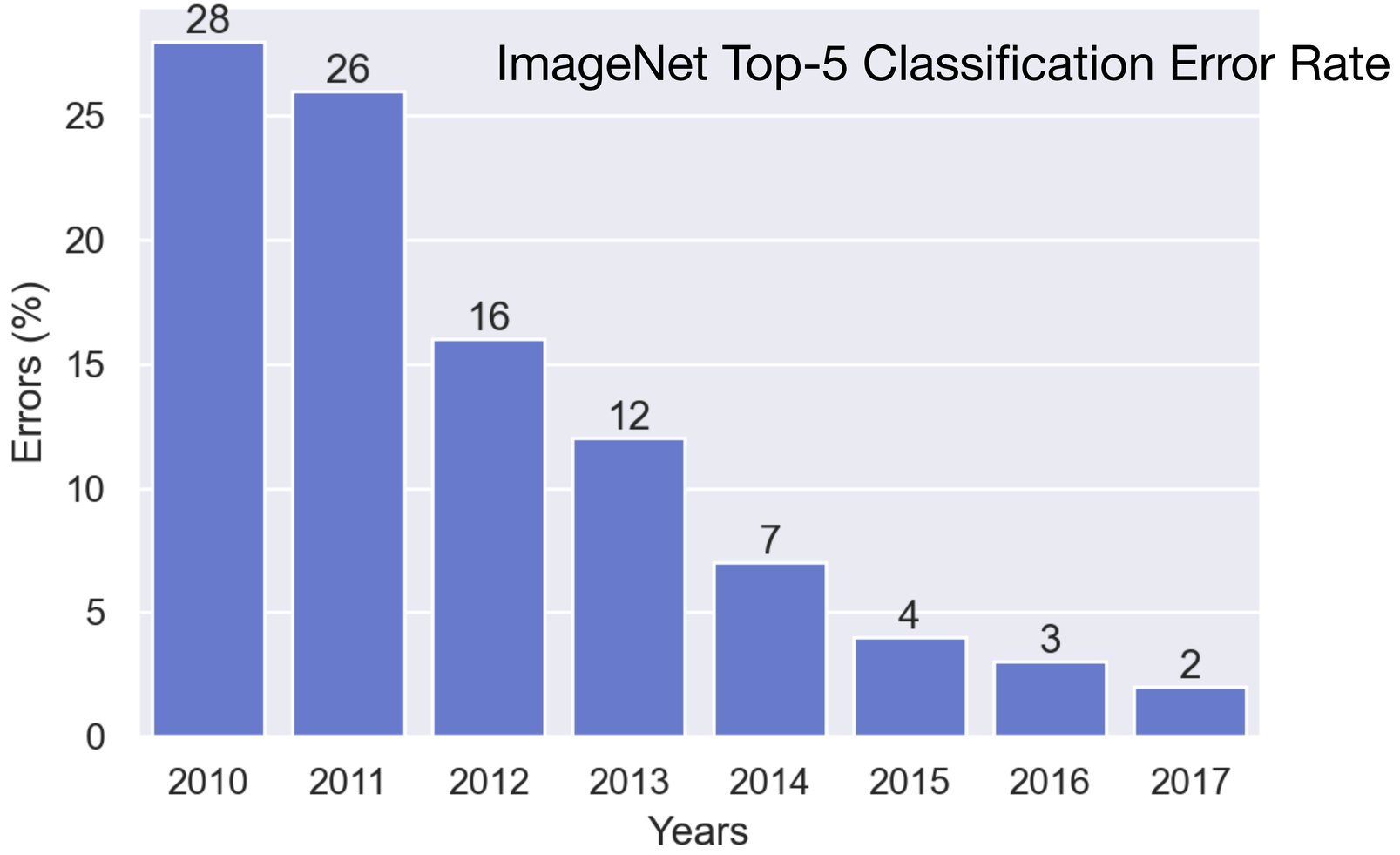
CNN 모델의 계보

2010~2011: Traditional Computer Vision techniques

2012: AlexNet

2014: VGGNet and GoogLeNet

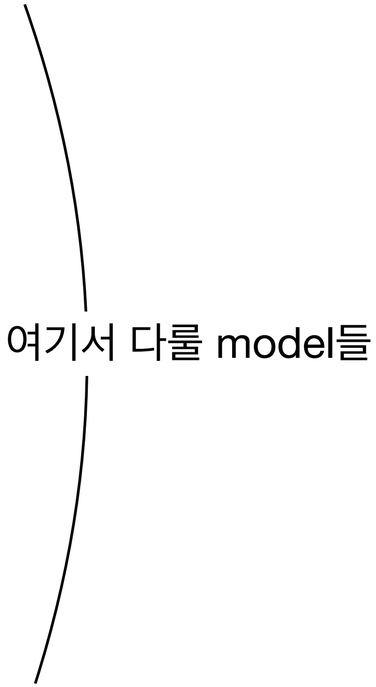
2015: ResNet



Convolutional Neural Network

대표적인 CNN 모델

- LeNet
- AlexNet
- VGGNet
- InceptionNet, GoogLeNet
- ResNet
- DenseNet
- Squeeze and Excitation Networks
- MobileNet, MNasNet



ACADENTIAL

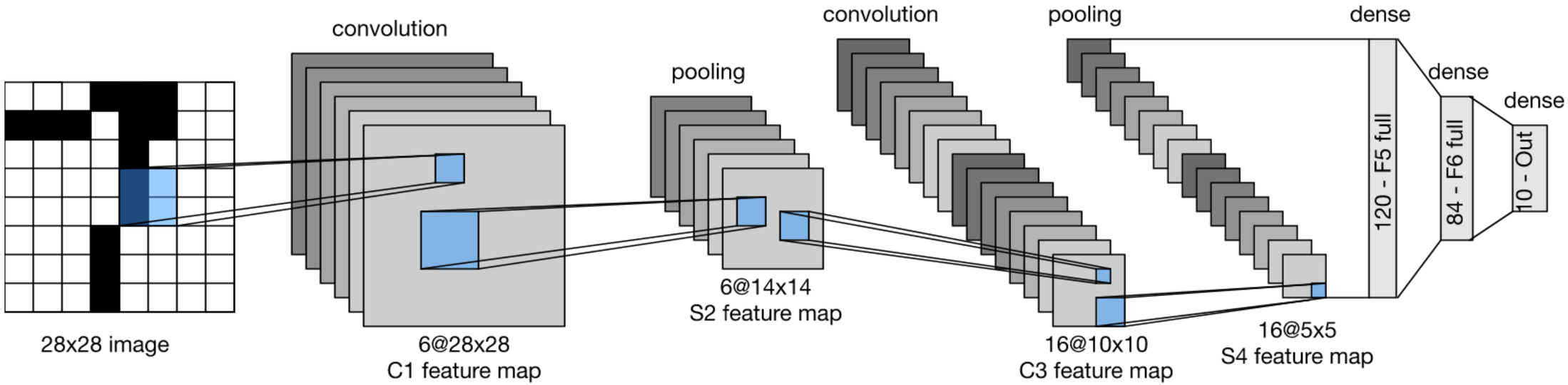
LeNet

ACADENTIAL

Convolutional Neural Network

LeNet 1989

출처: https://d2l.ai/chapter_convolutional-neural-networks/lenet.html



2 개의 Convolutional Layer, 3개의 Fully Connected Layer로 구성되어 있다!
가장 초창기의 CNN 모델

LeCun, Y., Boser, B., Denker, J.S., Henderson, D., Howard, R.E., Hubbard, W. and Jackel, L.D., 1989. Backpropagation applied to handwritten zip code recognition. *Neural computation*, *1*(4), pp.541-551.

ACADENTIAL

AlexNet

ACADENTIAL

Convolutional Neural Network

AlexNet

특징:

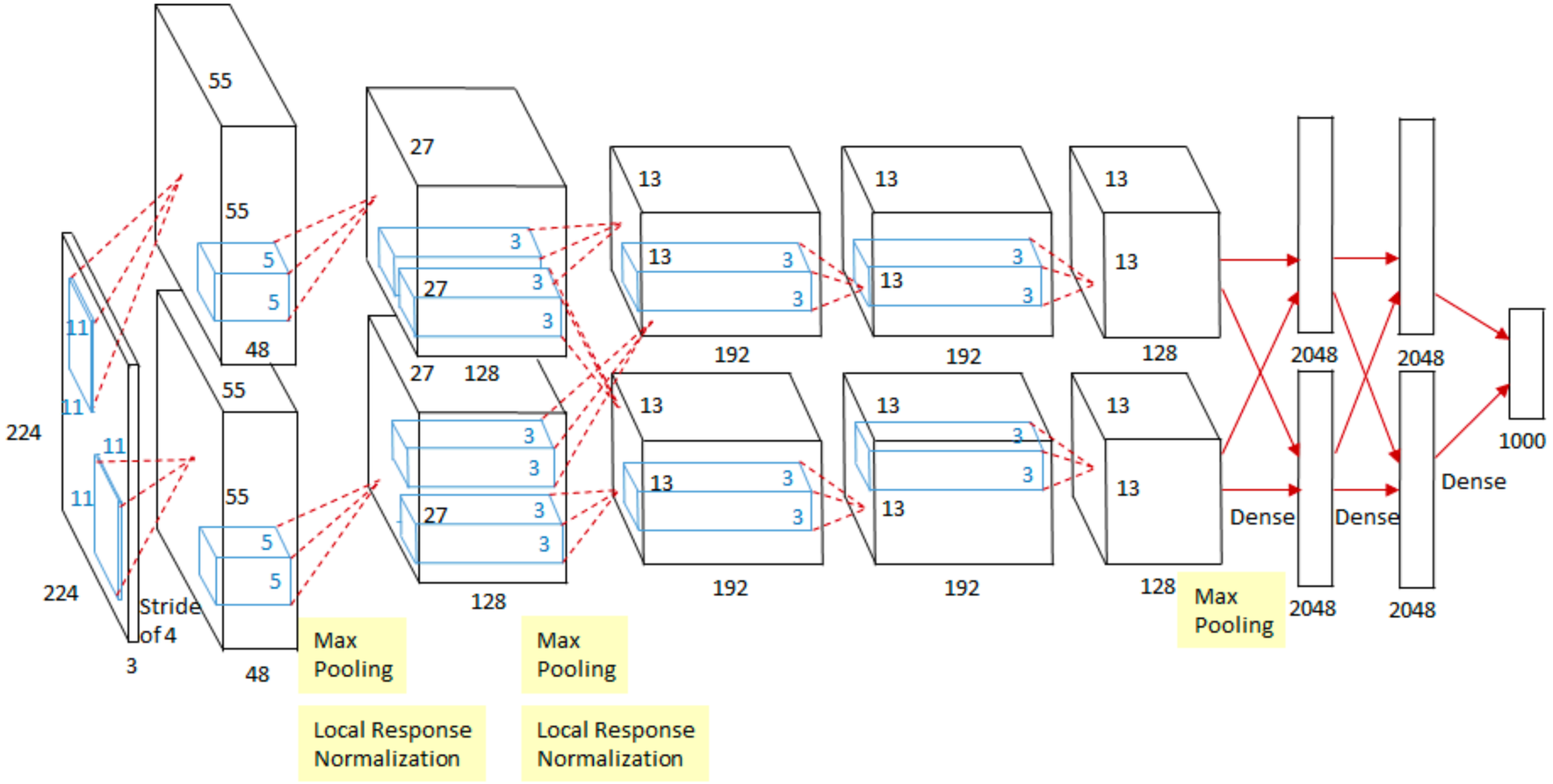
- 5개의 Conv layer들과, 3개의 FC layer들로 구성됨.
- (Tanh 대신에) ReLU을 activation function을 사용함.
- 2개의 GPU에 모델을 나눠서 학습함.
- Local Response Normalization을 사용함.
- Overlapping Max Pooling을 사용함.
- 첫번째, 2번째 FC layer에서 Dropout을 사용함.

“ImageNet Classification with Deep Convolutional Neural Networks”
Krizhevsky, A., Sutskever, I. and Hinton, G.E., 2012. Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems*, *25*.

ACADENTIAL

Convolutional Neural Network

AlexNet

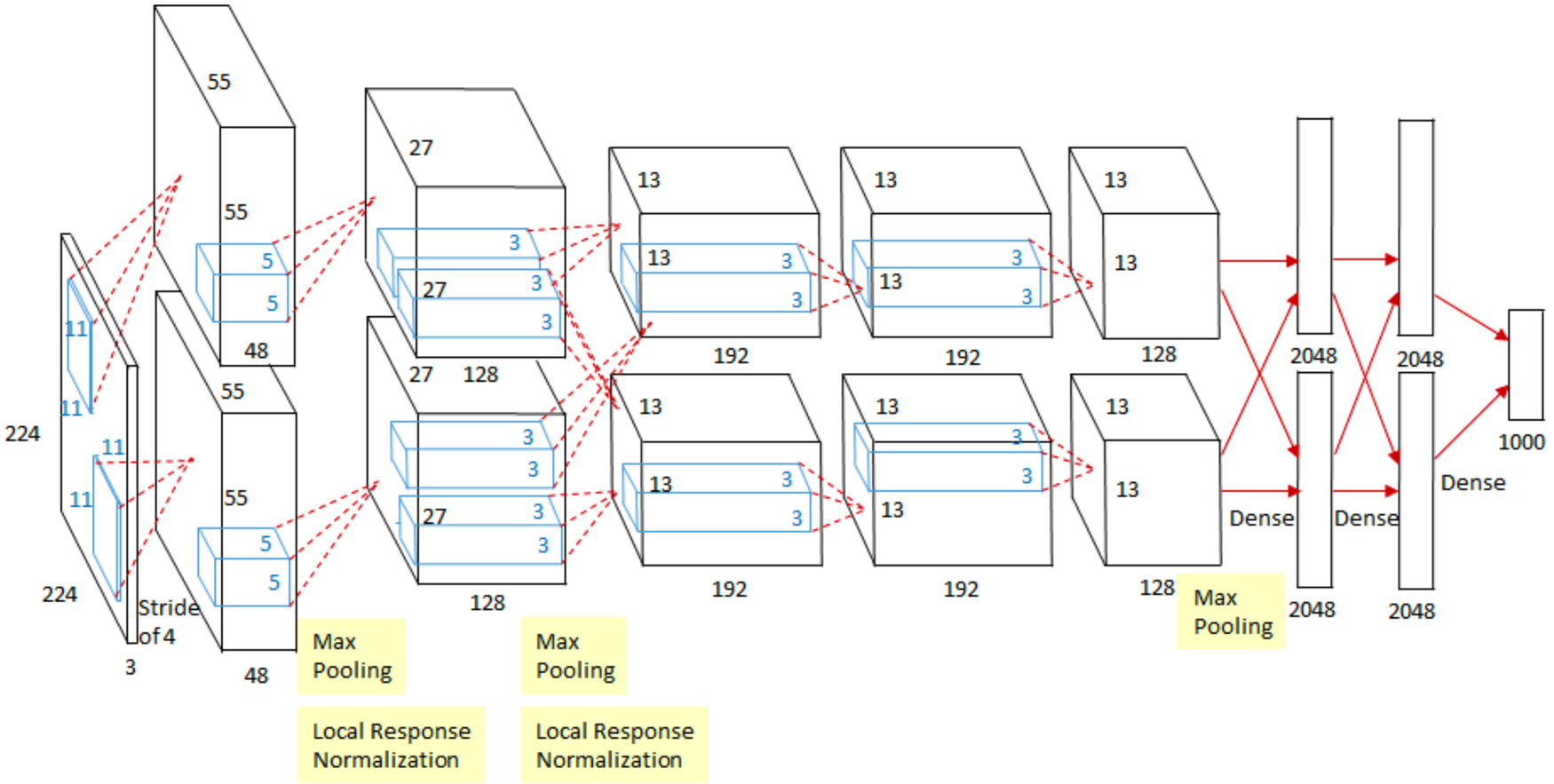


"ImageNet Classification with Deep Convolutional Neural Networks"
Krizhevsky, A., Sutskever, I. and Hinton, G.E., 2012. Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems*, *25*.

ACADENTIAL

Convolutional Neural Network

AlexNet



Model architecture:

1. 96개의 Conv kernel (11x11x3), Max Pooling
2. 256개의 Conv kernel (5x5x48), Max Pooling
3. 384개의 Conv kernel (3x3x256)
4. 384개의 Conv kernel (3x3x192)
5. 256개의 Conv kernel (3x3x192)
6. 4096개의 neuron으로 구성된 FC layer
7. 6번과 동일
8. 1000개의 output class으로 mapping해주는 FC layer

"ImageNet Classification with Deep Convolutional Neural Networks"
Krizhevsky, A., Sutskever, I. and Hinton, G.E., 2012. Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems*, *25*.

Convolutional Neural Network

AlexNet

ILSVRC-2012 (ImageNet)에서 1위

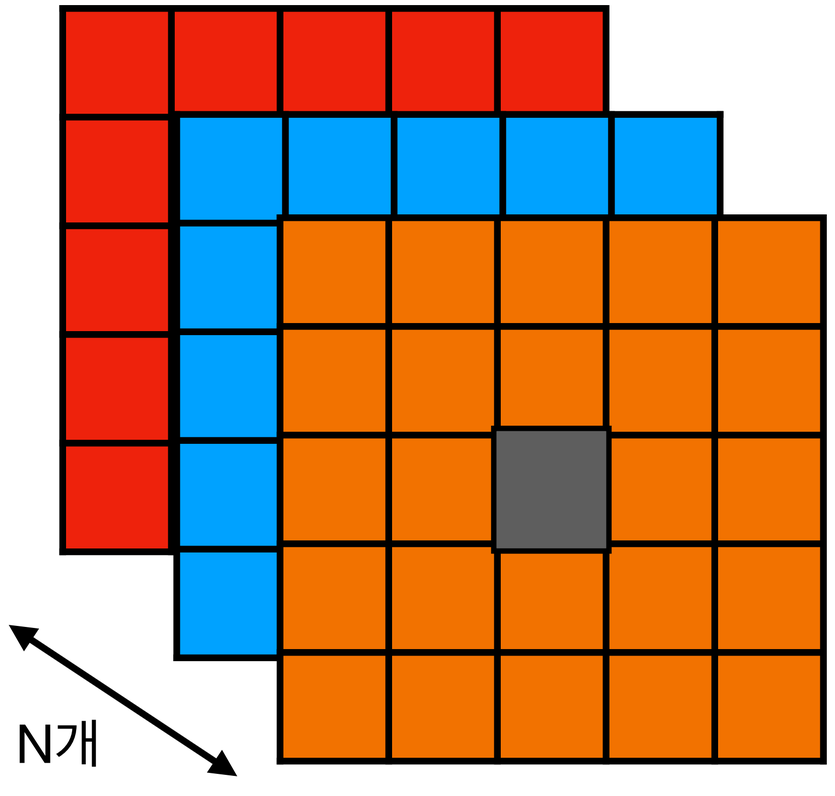
Model	Top-1 (val)	Top-5 (val)	Top-5 (test)
SIFT + FVs [7]	—	—	*26.2%*
1 CNN	40.7%	18.2%	—
5 CNNs	38.1%	16.4%	**16.4%**
1 CNN*	39.0%	16.6%	—
7 CNNs*	36.7%	15.4%	**15.3%**

참고 사항

Local Response Normalization이란?

$$b^i_{x,y} = a^i_{x,y} / \left(k + \alpha \sum_{j=\max(0,i-n/2)}^{\min(N-1,i+n/2)} (a^j_{x,y})^2 \right)^\beta$$

- 동일한 위치에 놓였고, N개의 adjacent한 (즉 주변부의) channel 들을 함께 normalize하는 것.
- (최근의 CNN에서는 더 이상 사용되지는 않는다)



ACADENTIAL

14-9. VGGNet

ACADENTIAL

Convolutional Neural Network

VGGNet

VGGNet의 특징:

- Conv Kernel의 크기를 (3x3)으로 줄여서 Conv Layer을 더 많이 쌓음 (~19개).
- 5개의 Max Pooling으로 feature map의 resolution을 점차 줄임.
- ILSVRC-2014 (ImageNet)에서 상위권 성적.

Simonyan, K. and Zisserman, A., 2014. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*.

Convolutional Neural Network

VGGNet

VGGNet의 특징:

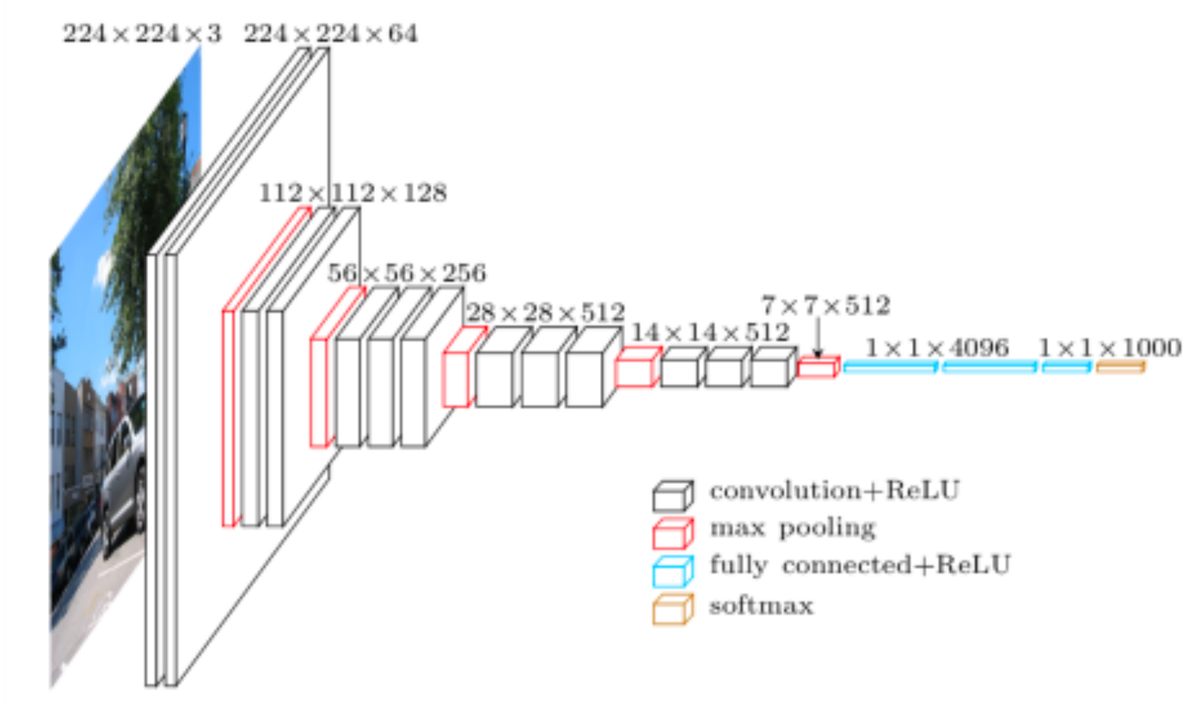
- Conv Kernel의 크기를 (3x3)으로 줄여서 Conv Layer을 더 많이 쌓음 (16개~19개).
- 5개의 Max Pooling으로 feature map의 resolution을 점차 줄임
- ILSVRC-2014 (ImageNet)에서 상위권 성적.

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	conv3-64 **LRN**	conv3-64 **conv3-64**	conv3-64 conv3-64	conv3-64 conv3-64	conv3-64 conv3-64
maxpool					
conv3-128	conv3-128	conv3-128 **conv3-128**	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 **conv1-256**	conv3-256 conv3-256 **conv3-256**	conv3-256 conv3-256 conv3-256 **conv3-256**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
FC-4096					
FC-4096					
FC-1000					
soft-max					

Simonyan, K. and Zisserman, A., 2014. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*.

Convolutional Neural Network

VGGNet



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	conv3-64 **LRN**	conv3-64 **conv3-64**	conv3-64 conv3-64	conv3-64 conv3-64	conv3-64 conv3-64
maxpool					
conv3-128	conv3-128	conv3-128 **conv3-128**	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 **conv1-256**	conv3-256 conv3-256 **conv3-256**	conv3-256 conv3-256 conv3-256 **conv3-256**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
FC-4096					
FC-4096					
FC-1000					
soft-max					

Simonyan, K. and Zisserman, A., 2014. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*.

ACADENTIAL

Convolutional Neural Network

VGGNet

“Conv Kernel의 크기를 (3x3)으로 줄여서 Conv Layer을 더 많이 쌓음 (16개~19개).”

Simonyan, K. and Zisserman, A., 2014. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*.

Convolutional Neural Network

VGGNet

“Conv Kernel의 크기를 (3x3)으로 줄여서 Conv Layer을 더 많이 쌓음 (16개~19개).”

효과:

1. **kernel의 크기가 작은** Conv. kernel **여러 개**를 쌓은 경우의 receptive field의 크기와
2. **kernel의 크기가 큰** Conv. kernel **하나**의 경우의 receptive field의 크기는 비슷하지만

Simonyan, K. and Zisserman, A., 2014. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*.

Convolutional Neural Network

VGGNet

“Conv Kernel의 크기를 (3x3)으로 줄여서 Conv Layer을 더 많이 쌓음 (16개~19개).”

효과:

1. **kernel의 크기가 작은** Conv. kernel **여러 개**를 쌓은 경우 (1) 의 receptive field의 크기와
2. **kernel의 크기가 큰** Conv. kernel **하나**의 경우 (2) 의 receptive field의 크기는 비슷하지만

- (1) 의 경우가 parameter의 개수가 **더 적다.** (즉, **Parameter efficiency 개선**)
- 또한 (1)의 경우가 layer을 더 많이 쌓을 수 있으므로 더 complex하거나 정교한 decision boundary을 modeling할 수 있게된다. (즉, **증가된 non-linearity**)

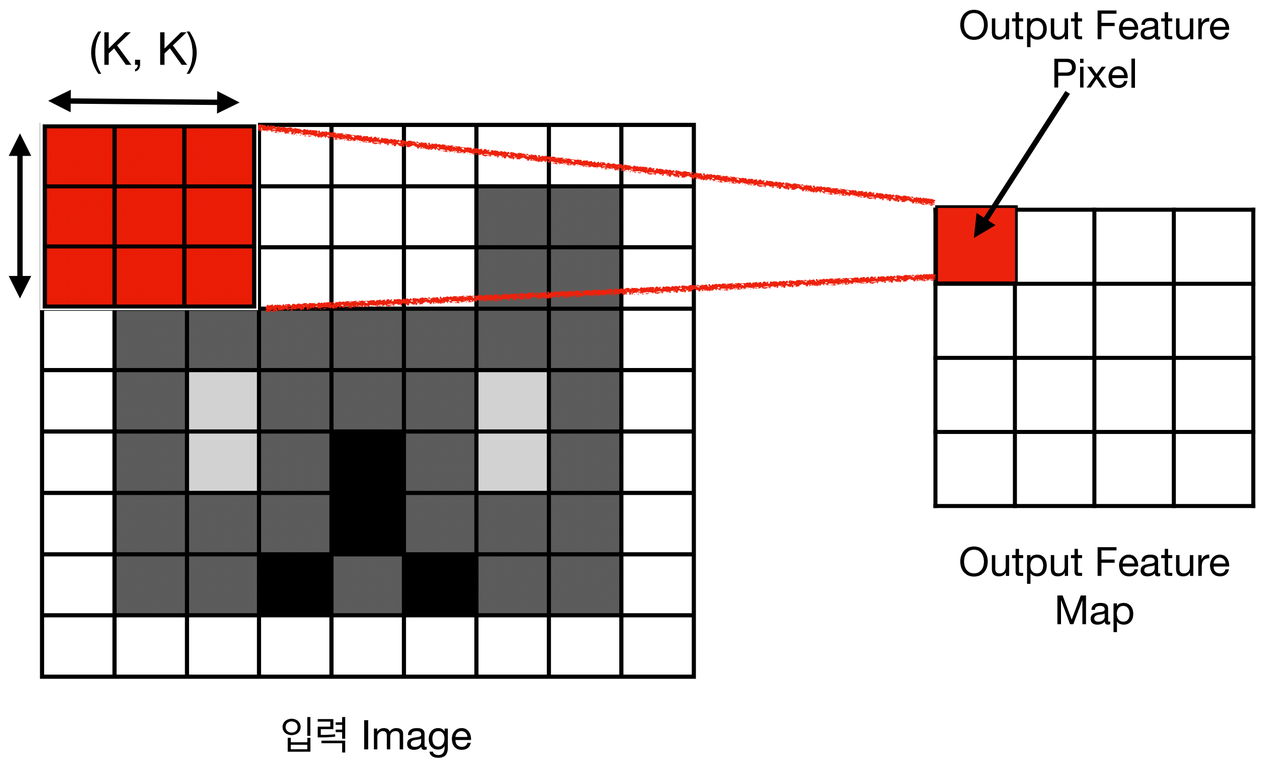
Simonyan, K. and Zisserman, A., 2014. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*.

ACADENTIAL

참고 사항

Receptive Field란 무엇인가?

- (K, K)크기의 Conv. kernel의 경우:
- Output feature map의 각 “output feature pixel”은 (이전 Conv Layer로부터 출력된) “input feature map” 상에서 **(K, K) 크기의 영역**에 **대응**됨.



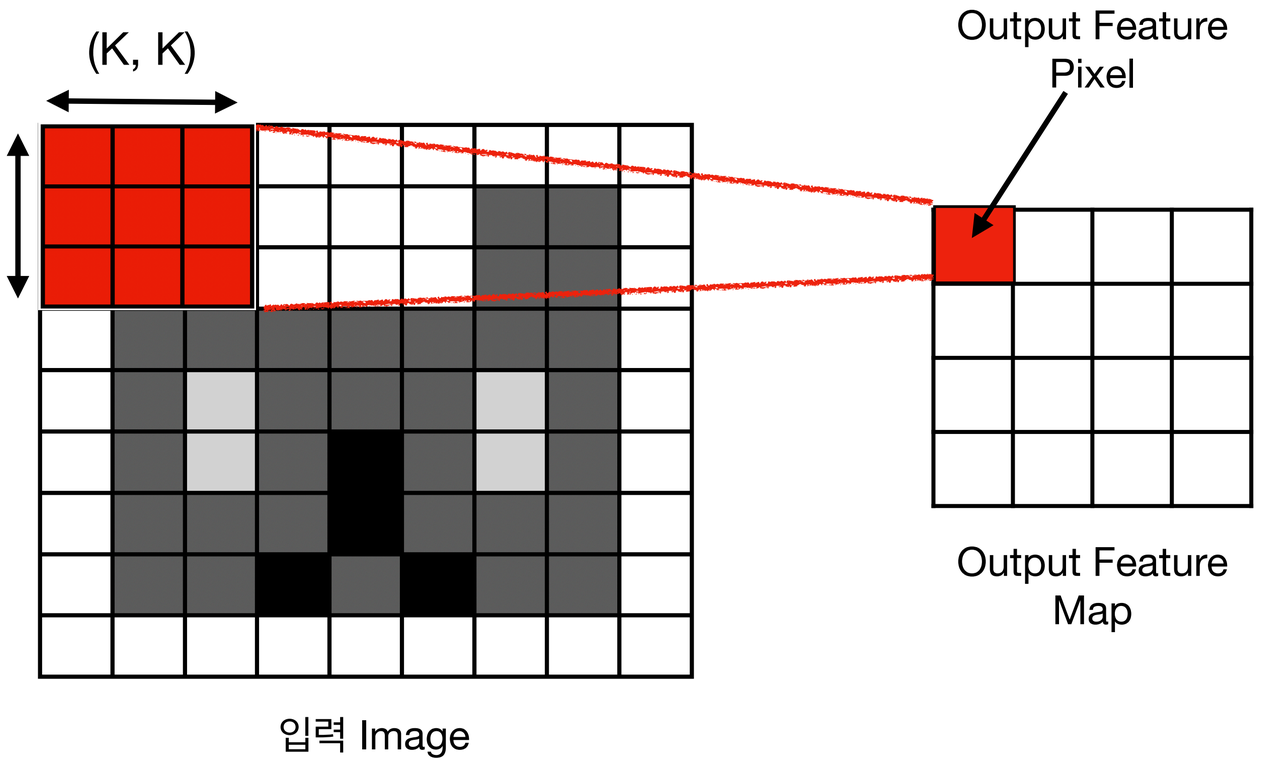
Receptive Field = 각 Output feature pixel이 입력 Image 상에서 영향을 받는 pixel 영역의 크기

ACADENTIAL

참고 사항

Receptive Field란 무엇인가?

- (K, K)크기의 Conv. kernel의 경우:
- Output feature map의 각 “output feature pixel”은 (이전 Conv Layer로부터 출력된) “input feature map” 상에서 **(K, K) 크기의 영역**에 **대응**됨.
- Conv. Layer가 하나만 있는 경우, receptive field은 곧 Convolutional Kernel의 크기.
- 만약에 Conv. Layer가 여러 개라면?

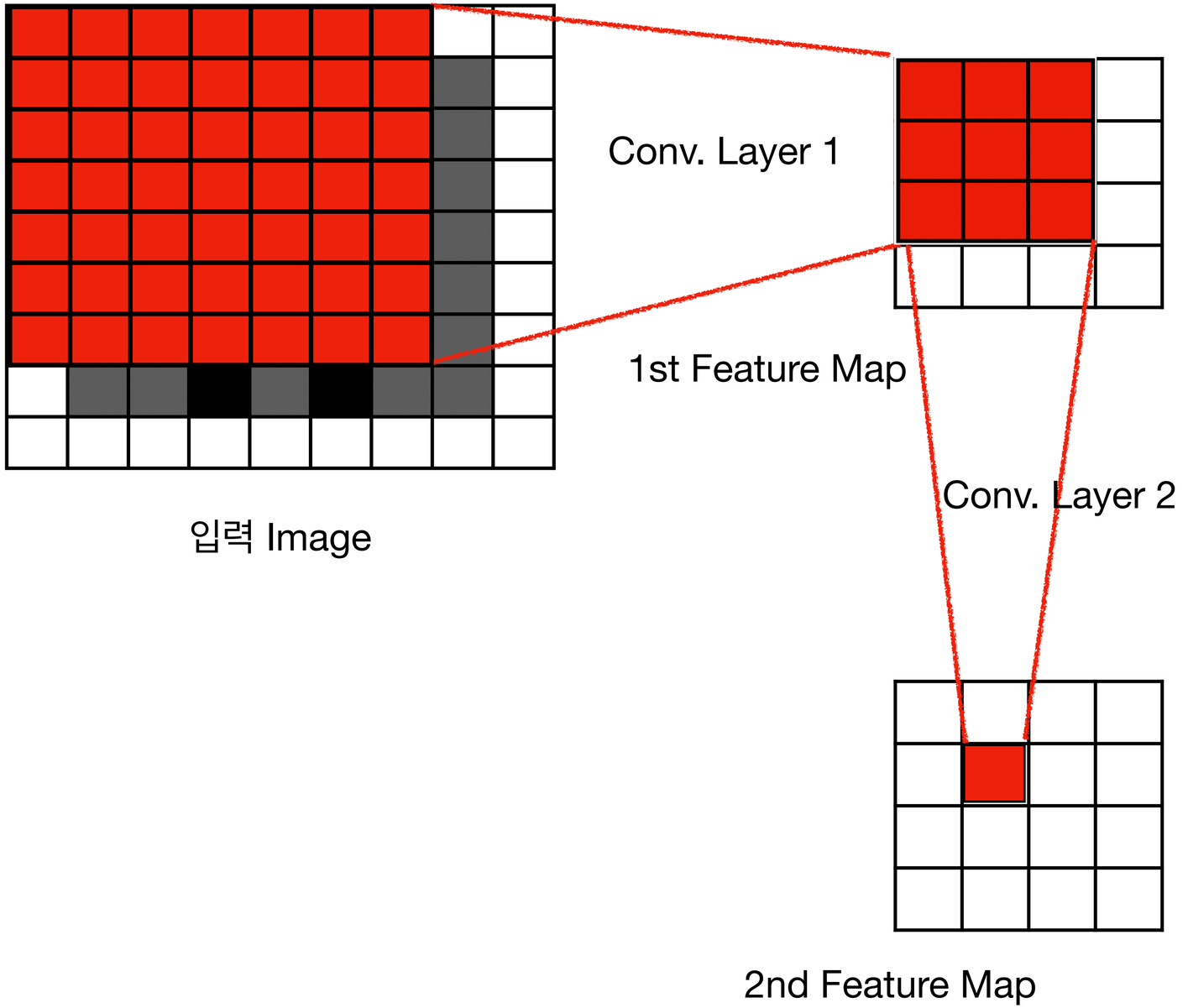


Receptive Field = 각 Output feature pixel이 입력 Image 상에서 영향을 받는 pixel 영역의 크기

참고 사항

Receptive Field란 무엇인가?

- 만약에 Conv. Layer가 여러 개라면?
- 후속의 **Output Feature pixel이 입력 Image 상**에서 **더 큰 영역의 pixel**들에 대응된다.

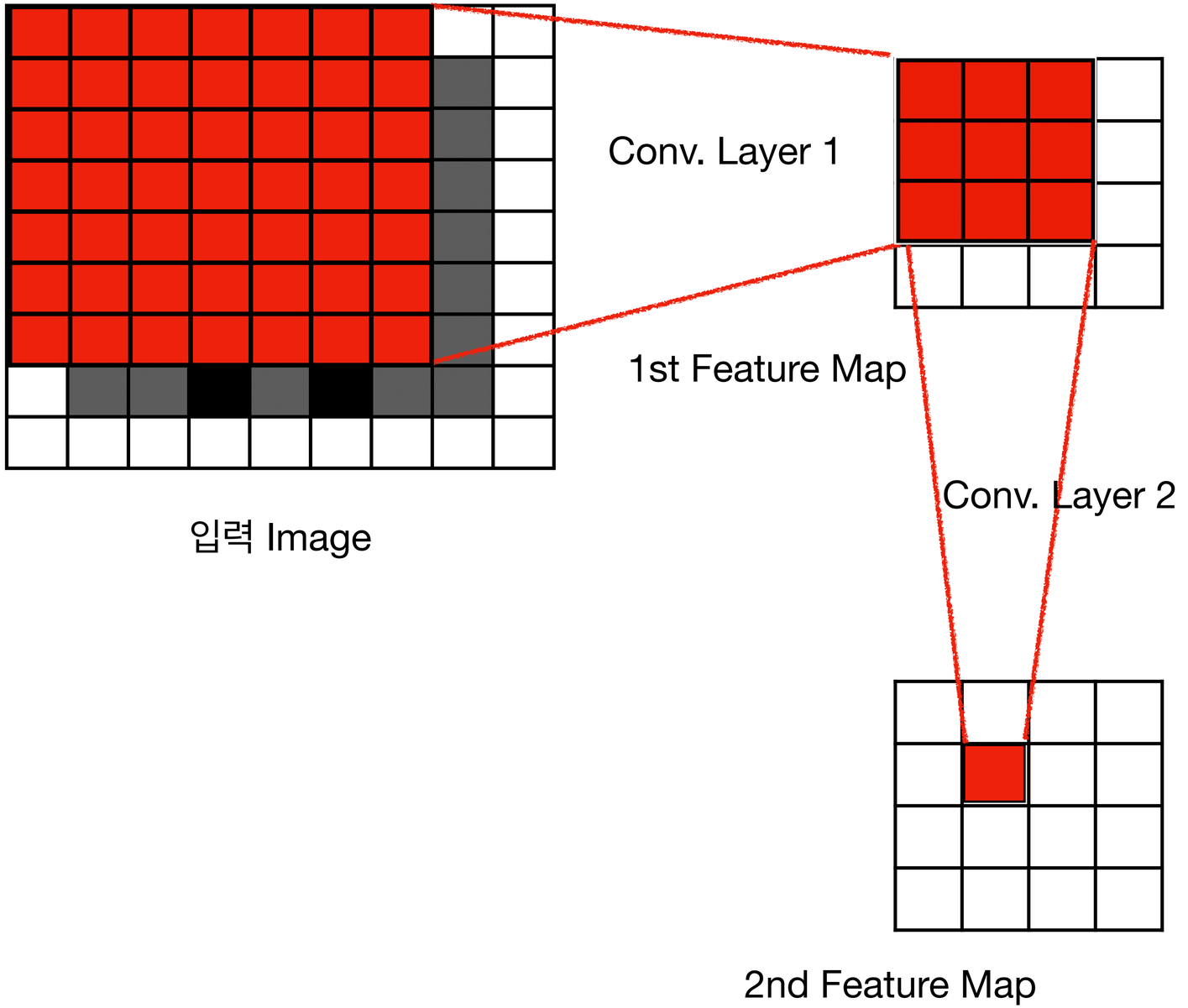


ACADENTIAL

참고 사항

Receptive Field란 무엇인가?

- 만약에 Conv. Layer가 여러 개라면?
- 후속의 **Output Feature pixel이 입력 Image 상**에서 **더 큰 영역의 pixel**들에 대응된다.
- 즉, Receptive Field의 크기가 점차 커진다.

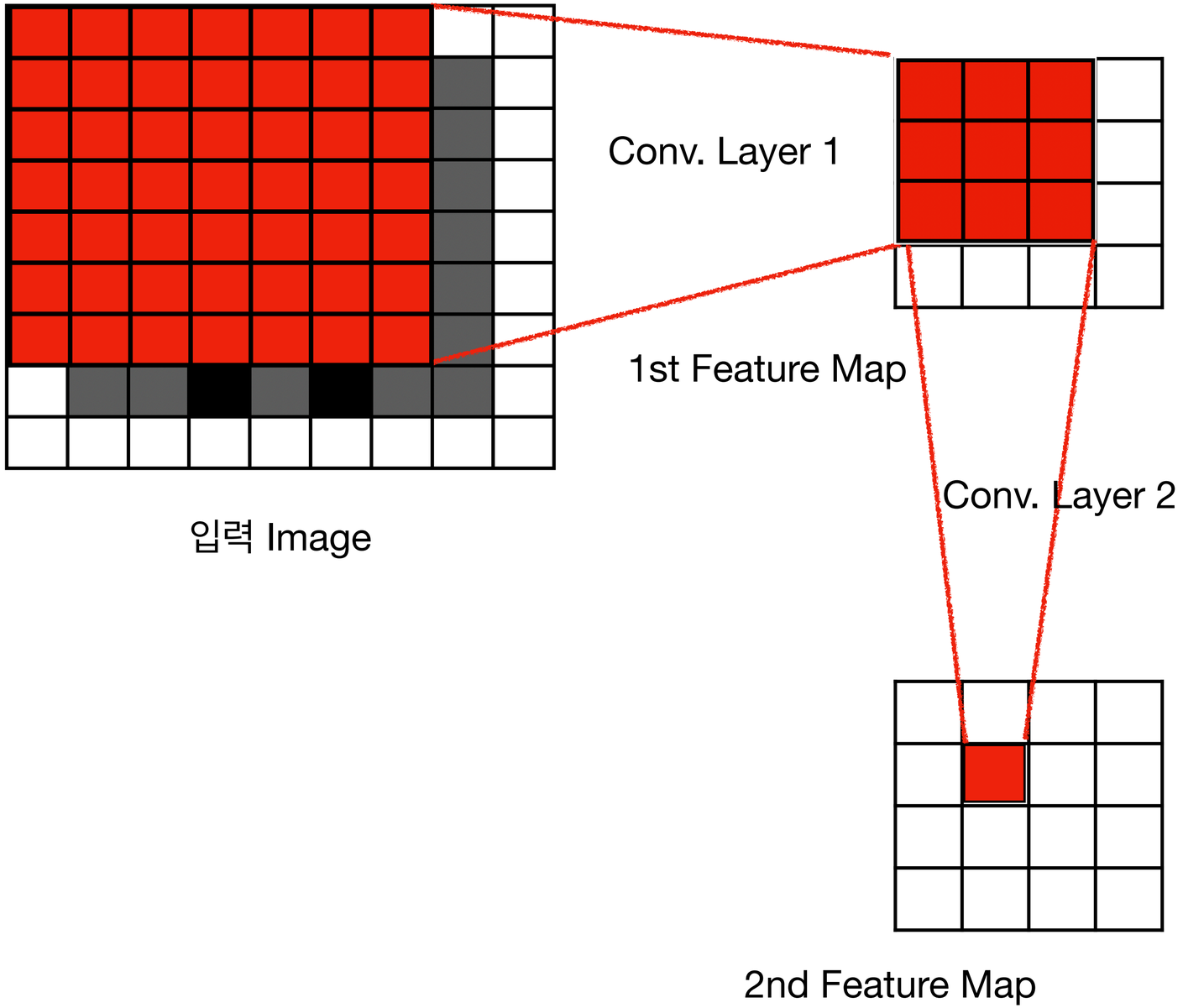


ACADENTIAL

참고 사항

Receptive Field란 무엇인가?

- 만약에 Conv. Layer가 여러 개라면?
- 후속의 **Output Feature pixel이 입력 Image 상**에서 **더 큰 영역의 pixel**들에 대응된다.
- 즉, Receptive Field의 크기가 점차 커진다.
- 예를 들어 (3x3) Conv. Kernel 2개를 쌓은 경우 Receptive Field은 (7x7)이 된다!

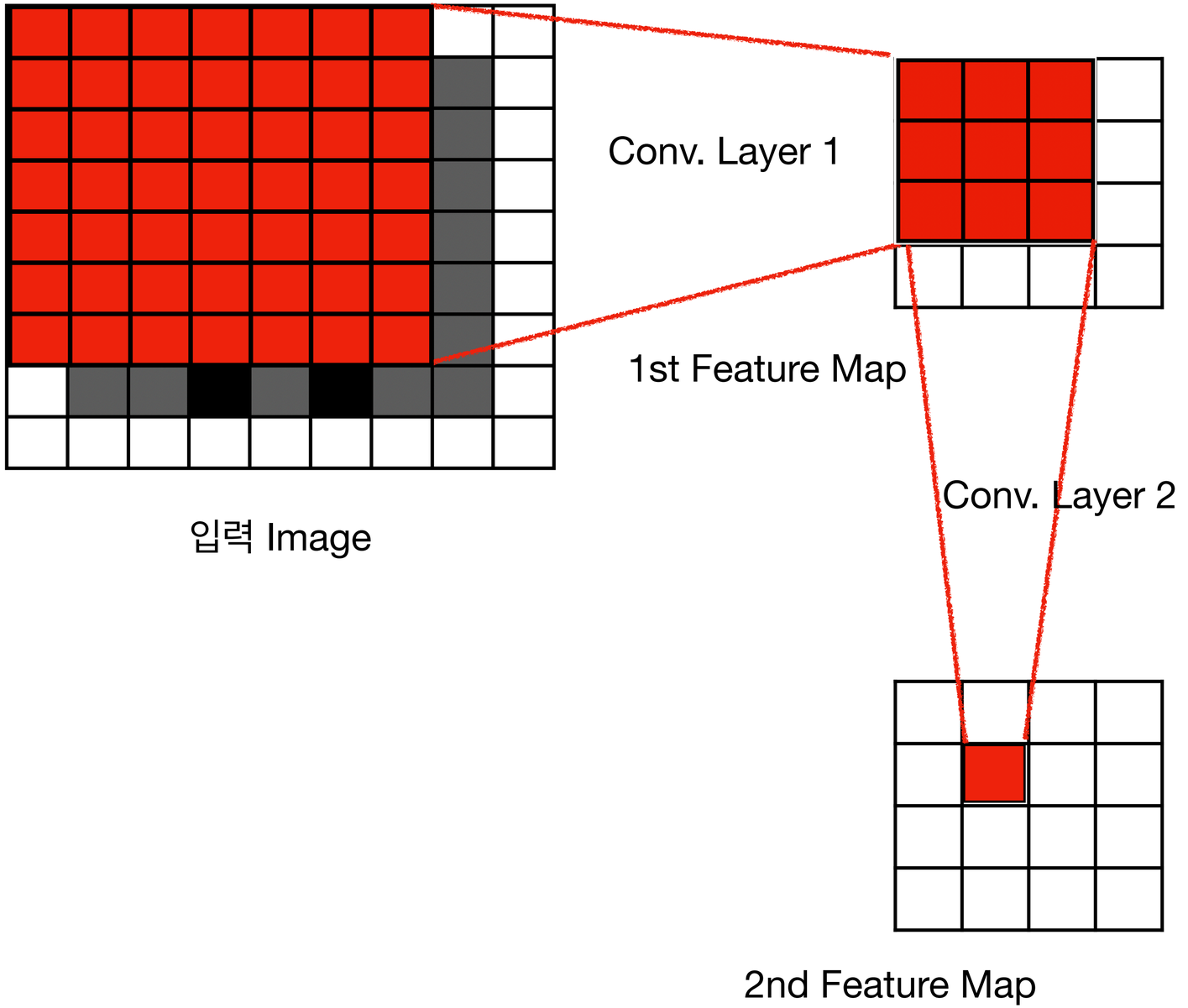


ACADENTIAL

참고 사항

Receptive Field란 무엇인가?

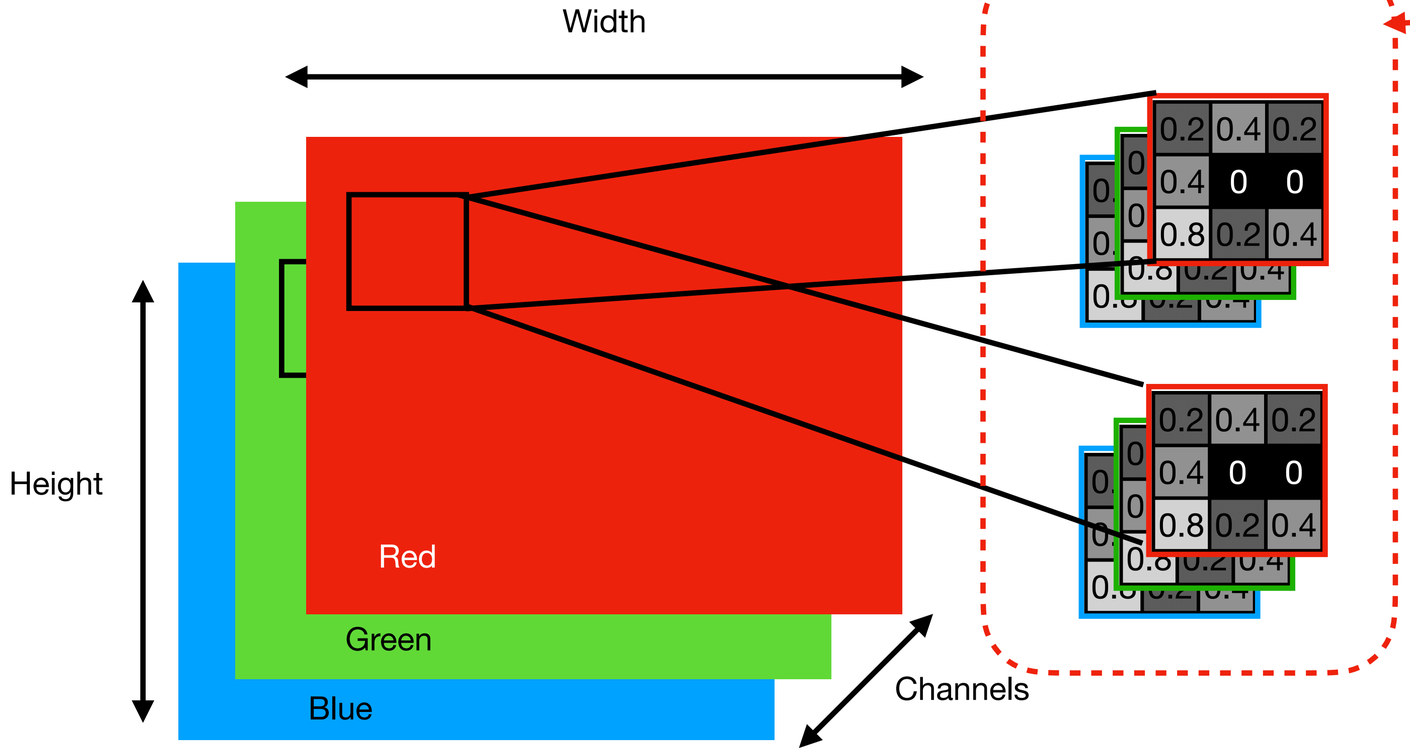
- 만약에 Conv. Layer가 여러 개라면?
- 후속의 **Output Feature pixel이 입력 Image 상**에서 **더 큰 영역의 pixel**들에 대응된다.
- 즉, Receptive Field의 크기가 점차 커진다.
- 예를 들어 (3x3) Conv. Kernel 2개를 쌓은 경우 Receptive Field은 (7x7)이 된다!
- 다음으로 Conv. Kernel의 Weight parameter 의 개수를 살펴보자.



ACADENTIAL

참고 사항

Conv. Kernel의 크기에 따른 Parameter의 개수



Input channel == 3

Conv. Kernel의 Weight
$= (H_K, W_K, C_{in}, C_{out})$

참고 사항:

- **CNN weight의 shape**은
- $(H_K, W_K, C_{in}, C_{out})$ 가 된다.
- H_K = Kernel Height, W_K= Kernel Width
- C_{in} = in channel, C_{out} = out channel.

즉, weight parameter의 개수 = $H_K \cdot W_K \cdot C_{in} \cdot C_{out}$

참고 사항

Conv. Kernel의 크기에 따른 Parameter의 개수

	(3x3) Conv. Kernel 2개		(7x7) Conv. Kernel 1개
Receptive Field Size	7x7	=	7x7
Weight parameter 개수	$18 \times C_{in}C_{out}$	<	$49 \times C_{in}C_{out}$

- 따라서 VGG에서는 (7x7) Conv. Kernel을 쓰는 대신에 (3x3) Conv. Kernel 2개를 사용하였다!
- 더 Parameter Efficient하고 Receptive Field은 동일하다!
- 그리고 Depth을 늘림 (Layer을 더 쌓음)으로서 더 복잡한 decision boundary을 모델링할 수 있다!

(3x3) Conv Kernel 2개의 Weight Parameter 개수:
$2 \times (3 \times 3 \times C_{in} \times C_{out}) = 18 \times C_{in}C_{out}$

(7x7) Conv Kernel 2개의 Weight Parameter 개수:
$7 \times 7 \times C_{in} \times C_{out} = 49 \times C_{in}C_{out}$

Convolutional Neural Network

VGGNet: 특징

1. **(7x7)대신에 2개의 (3x3) Conv. Kernel의 사용:**

    1. Receptive Field은 유지하돼 Parameter Efficiency을 높일 수 있다.

    2. Parameter을 줄이는 것은 model capacity을 줄이는 것이므로 일종의 regularization의 효과도 볼 수 있다.

Simonyan, K. and Zisserman, A., 2014. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*.

Convolutional Neural Network

VGGNet: 특징

1. **(7x7)대신에 2개의 (3x3) Conv. Kernel의 사용:**
    1. Receptive Field은 유지하돼 Parameter Efficiency을 높일 수 있다.
    2. Parameter을 줄이는 것은 model capacity을 줄이는 것이므로 일종의 regularization의 효과도 볼 수 있다.
2. **(1x1) Conv. Kernel을 사용:**
    1. Non-linearity을 추가하여 더 복잡한 decision boundary을 모델링하기 위해서.
    2. 참고로 (1x1) Conv. Kernel은 feature map상의 각 pixel에 대해서 fully connected layer을 적용하는 것과 동일하다.

Simonyan, K. and Zisserman, A., 2014. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*.

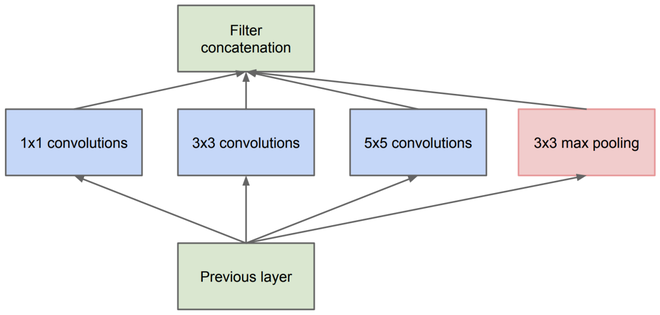
ACADENTIAL

14-10. Inception Net & GoogLeNet

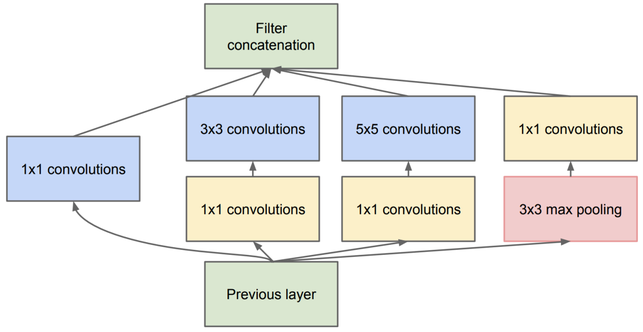
Convolutional Neural Network

GoogLeNet, InceptionNet

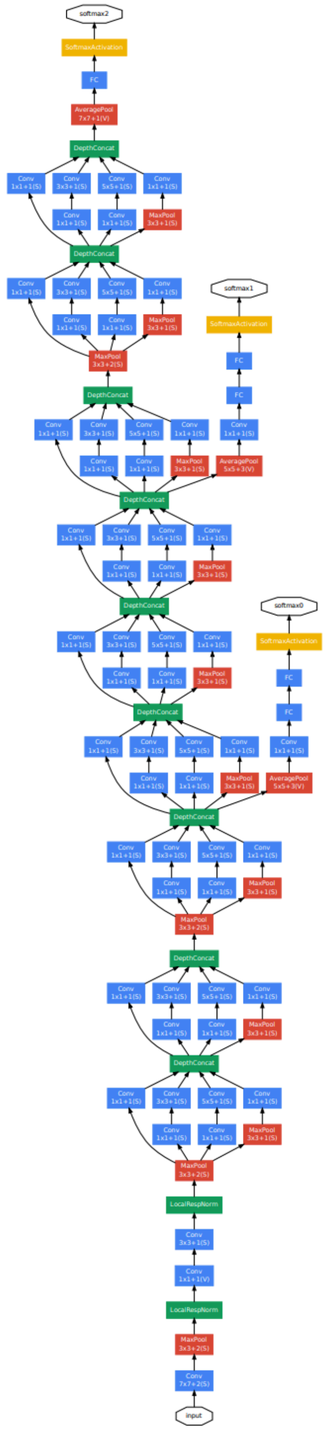
1. Going Deeper with Convolutions 논문
2. ILSVRC 2014 winning solution



(a) Inception module, naïve version



(b) Inception module with dimension reductions



Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., Erhan, D., Vanhoucke, V. and Rabinovich, A., 2015. Goin deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition* (pp. 1-9).

ACADENTIAL

Convolutional Neural Network

GoogLeNet, InceptionNet

특징:

1. Inception 모듈 제안
    1. Network in Network의 구조
    2. 다른 크기의 Conv. Kernel들을 가지고 있는 Conv. Layer들이 병렬되어 있다.
2. 1x1 Conv. Layer을 사용하여 dimensionality을 축소.
3. Auxiliary Loss을 통해 Vanishing Gradient 문제 해결

Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., Erhan, D., Vanhoucke, V. and Rabinovich, A., 2015. Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition* (pp. 1-9).

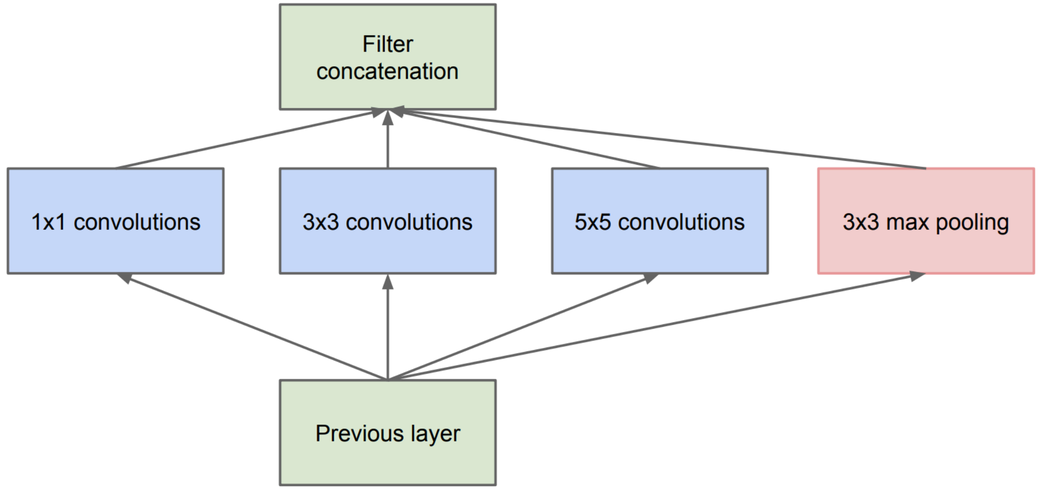
Convolutional Neural Network

Inception Module

- Inception 모듈에서는 **다양한 Conv. Kernel을 사용**한다.

  (1x1, 3x3, 5x5)와 3x3 Max-pooling

- 각 kernel, pooling으로부터 출력된 feature map을 concatenation해서 최종 출력한다.

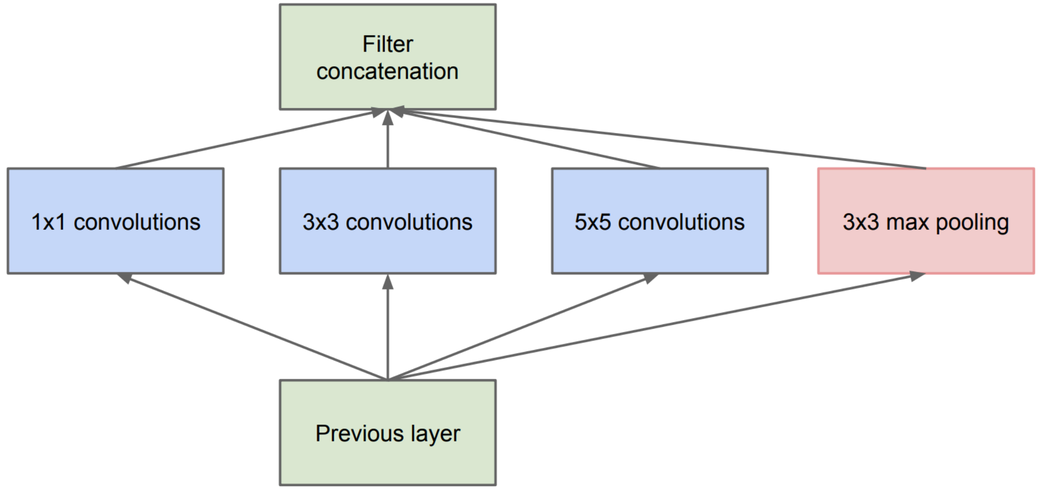


Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., Erhan, D., Vanhoucke, V. and Rabinovich, A., 2015. Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition* (pp. 1-9).

Convolutional Neural Network

Inception Module

- 다양한 kernel와 Pooling을 이용하는 이유:
  - 다양한 representation을 뽑아낼 수 있고, 이를 통해 의미있는 feature을 뽑아낼 수 있기 때문에!

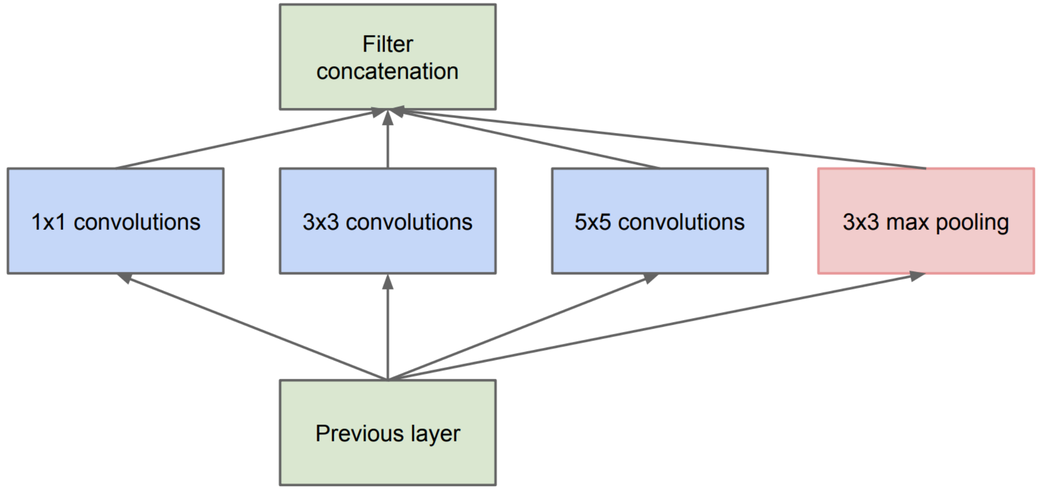


Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., Erhan, D., Vanhoucke, V. and Rabinovich, A., 2015. Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition* (pp. 1-9).

Convolutional Neural Network

Inception Module

- 다양한 kernel와 Pooling을 이용하는 이유:
  - 다양한 representation을 뽑아낼 수 있고, 이를 통해 의미있는 feature을 뽑아낼 수 있기 때문에!
  - (1x1) → (3x3) → (5x5) 순으로 local에서 더 넓은 context의 정보를 담아낼 수 있다.
  - Max-pooling: 기존 방법들에서 max-pooling이 성능 개선에 도움이 되어서 추가함.

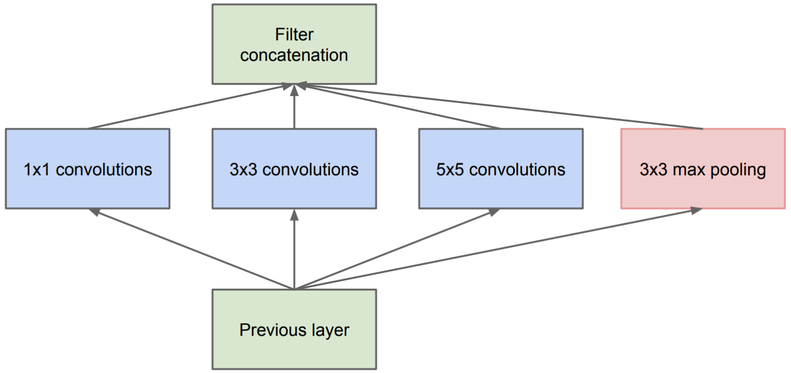


Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., Erhan, D., Vanhoucke, V. and Rabinovich, A., 2015. Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition* (pp. 1-9).

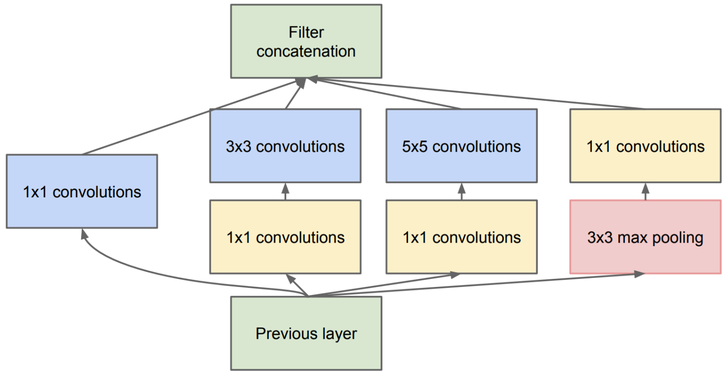
Convolutional Neural Network

(1x1) Conv. Layer to reduce dimensionality

- Inception module은 feature들을 concatenation해서 출력한다.
- 따라서 다음 layer에서 입력받는 C_{in}은 매우 크다.



(a) Inception module, naïve version



(b) Inception module with dimension reductions

Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., Erhan, D., Vanhoucke, V. and Rabinovich, A., 2015. Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition* (pp. 1-9).

Convolutional Neural Network

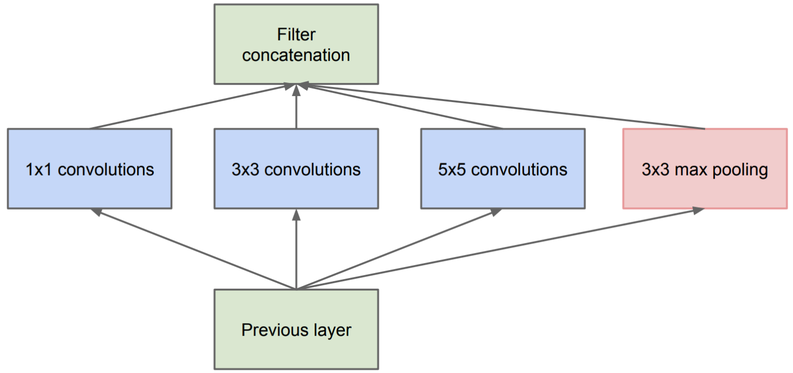
(1x1) Conv. Layer to reduce dimensionality

- 앞서 살펴봤듯이, Conv. Kernel의 weight 개수는

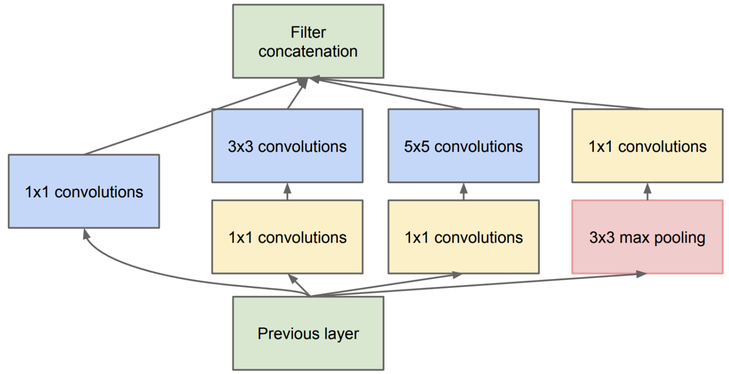
$$H_K \cdot W_K \cdot C_{in} \cdot C_{out}$$

- 따라서 **(3x3), (5x5)의 경우 C_{in}을 그대로 사용하는 것은 부담스럽다.**

- 높은 computational cost & 큰 weight parameter 개수



(a) Inception module, naïve version



(b) Inception module with dimension reductions

Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., Erhan, D., Vanhoucke, V. and Rabinovich, A., 2015. Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition* (pp. 1-9).

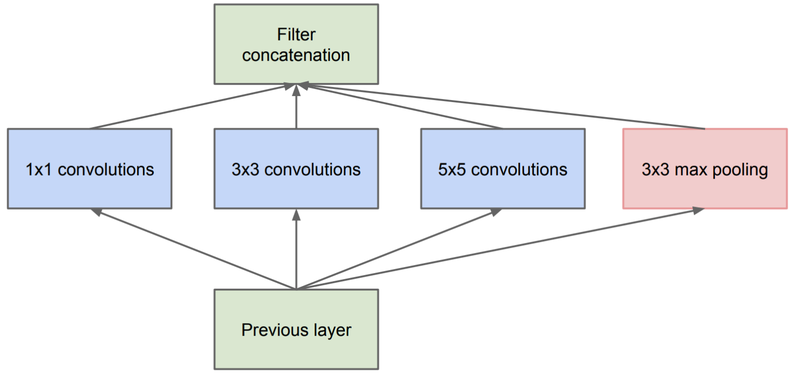
Convolutional Neural Network

(1x1) Conv. Layer to reduce dimensionality

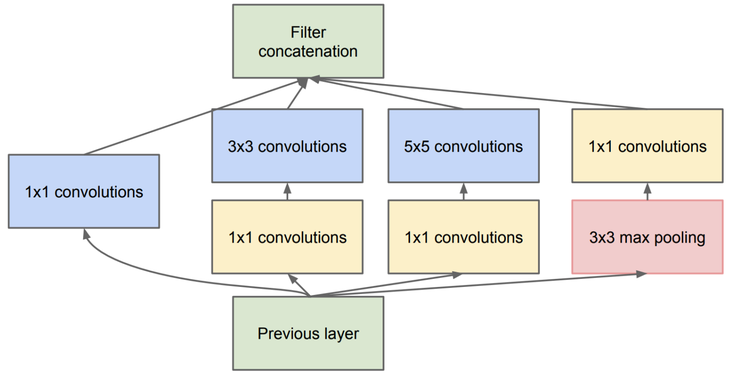
- 앞서 살펴봤듯이, Conv. Kernel의 weight 개수는

$$H_K \cdot W_K \cdot C_{in} \cdot C_{out}$$

- 따라서 **(3x3), (5x5)의 경우 C_{in}을 그대로 사용하는 것은 부담스럽다.**
- 높은 computational cost & 큰 weight parameter 개수
- 또한 Max-pooling은 $C_{in} = C_{out}$이므로 **다음 layer로 갈수록 channel의 수는 계속해서 커져버리고 "blow up"한다!**



(a) Inception module, naïve version



(b) Inception module with dimension reductions

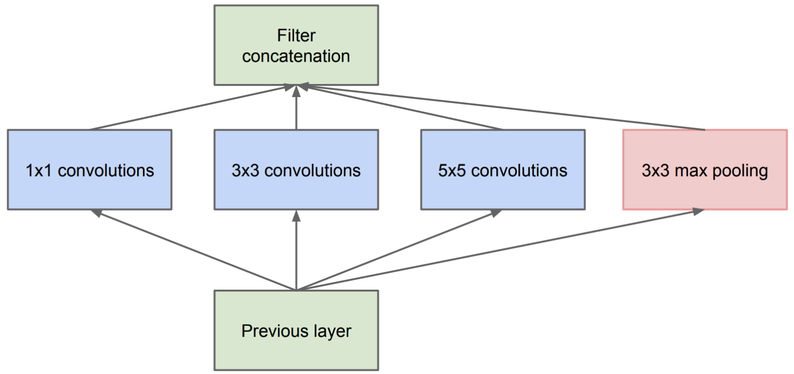
Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., Erhan, D., Vanhoucke, V. and Rabinovich, A., 2015. Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition* (pp. 1-9).

Convolutional Neural Network

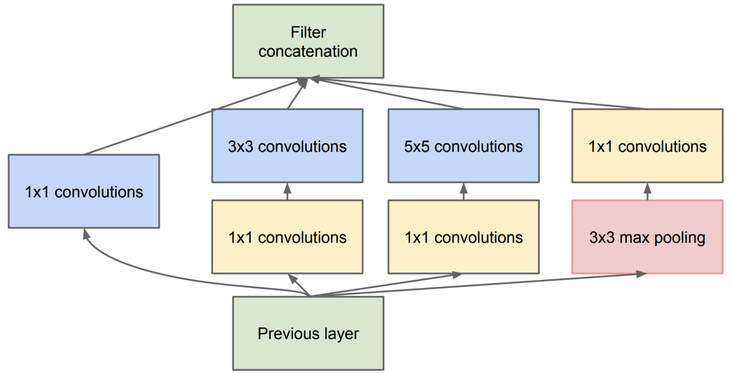
(1x1) Conv. Layer to reduce dimensionality

- 높은 computational cost & 큰 weight parameter 개수

따라서 dimensionality reduction이 필요하다!



(a) Inception module, naïve version



(b) Inception module with dimension reductions

Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., Erhan, D., Vanhoucke, V. and Rabinovich, A., 2015. Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition* (pp. 1-9).

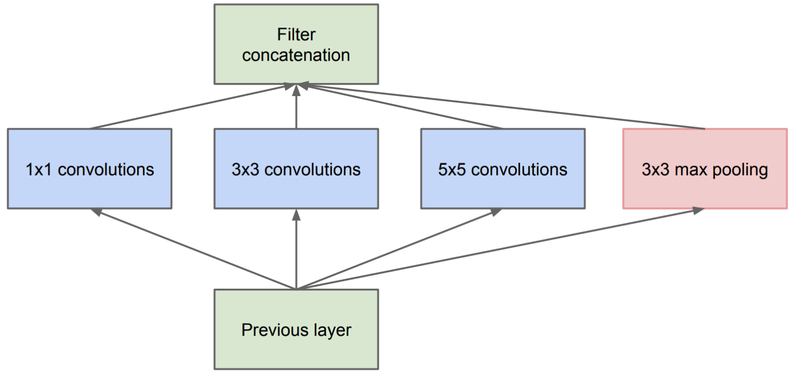
Convolutional Neural Network

(1x1) Conv. Layer to reduce dimensionality

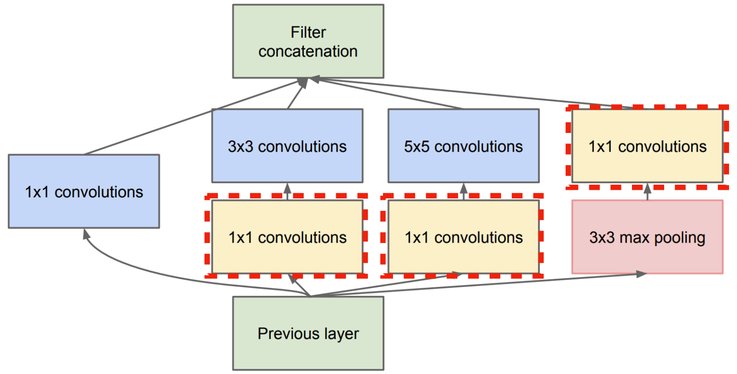
따라서 dimensionality reduction이 필요하다!

- (1x1) Convolution을 사용하여 $C_{in} \rightarrow C'_{in}$ 으로 줄여준다.

$$(C_{in} > C'_{in})$$



(a) Inception module, naïve version



(b) Inception module with dimension reductions

Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., Erhan, D., Vanhoucke, V. and Rabinovich, A., 2015. Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition* (pp. 1-9).

Convolutional Neural Network

Auxiliary Loss을 통해 Vanishing Gradient 문제 해결

- Layer을 너무 많이 쌓게 되면 Vanishing Gradient의 문제가 발생한다.
- 이를 해결하고자 **Auxiliary Loss** (부수적 손실함수)를 Network 중간 중간에 추가해준다.

Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., Erhan, D., Vanhoucke, V. and Rabinovich, A., 2015. Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition* (pp. 1-9).

Convolutional Neural Network

Auxiliary Loss을 통해 Vanishing Gradient 문제 해결

- Auxiliary Loss의 과정:
    1. Intermediate Inception Module의 출력값을 사용하여 label에 대해 예측한다.
        1. 즉, “intermediate prediction”을 구한다
    2. “intermediate prediction”와 label을 비교하여 Auxiliary Loss을 구한다.

Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., Erhan, D., Vanhoucke, V. and Rabinovich, A., 2015. Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition* (pp. 1-9).

Convolutional Neural Network

Auxiliary Loss을 통해 Vanishing Gradient 문제 해결

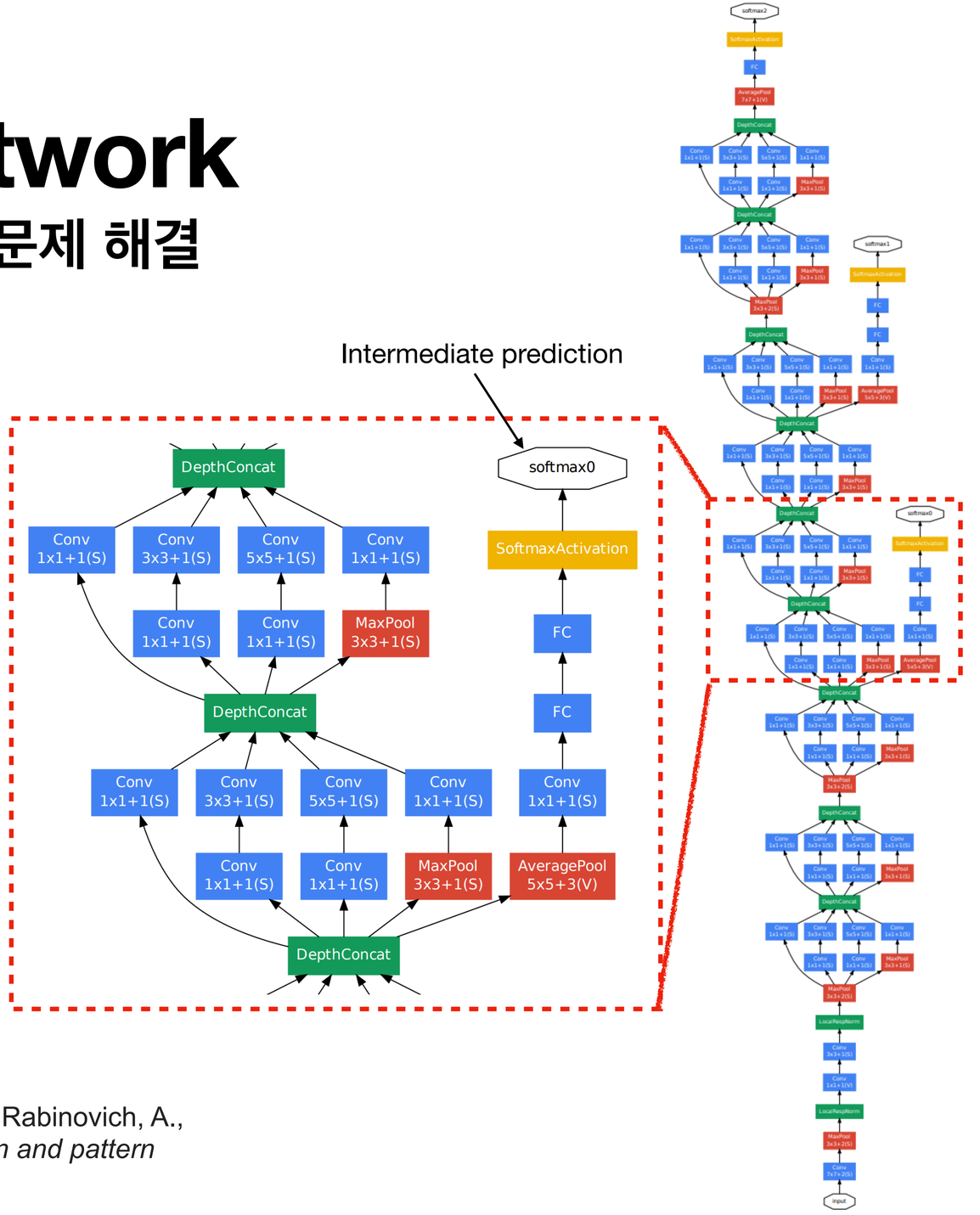
- Auxiliary Loss의 과정:
    1. Intermediate Inception Module의 출력값을 사용하여 label에 대해 예측한다.
        1. 즉, “intermediate prediction”을 구한다
    2. “intermediate prediction”와 label을 비교하여 Auxiliary Loss을 구한다.
- Auxiliary Loss을 통해 Network 중간중간에 “Gradient을 주입”하는 셈!

Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., Erhan, D., Vanhoucke, V. and Rabinovich, A., 2015. Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition* (pp. 1-9).

Convolutional Neural Network

Auxiliary Loss을 통해 Vanishing Gradient 문제 해결

- Auxiliary Loss의 과정:
  1. Intermediate Inception Module의 출력값을 사용하여 label에 대해 예측한다. (즉, “intermediate prediction”을 구한다)
  2. “intermediate prediction”와 label을 비교하여 Auxiliary Loss을 구한다.
- Auxiliary Loss을 통해 Network 중간중간에 “Gradient을 주입”하는 셈!



Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., Erhan, D., Vanhoucke, V. and Rabinovich, A., 2015. Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition* (pp. 1-9).

ACADENTIAL

14-11. ResNet

Convolutional Neural Network

ResNet

- **Layer을 너무 많이 쌓게 되면**
  1. **Vanishing Gradient/Exploding Gradients의 문제**가 발생하고
  2. 오히려 **training error가 높아지는 “degradation problem”**이 발생한다.

He, K., Zhang, X., Ren, S. and Sun, J., 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition* (pp. 770-778).

ACADENTIAL

Convolutional Neural Network

ResNet

- **Degradation problem:**
  - 그렇다면 왜 Layer가 **더 깊은 (deeper) NN**이 **얕은 (shallow) NN**보다 **training 성능**이 떨어지는가?

He, K., Zhang, X., Ren, S. and Sun, J., 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition* (pp. 770-778).

Convolutional Neural Network

ResNet

- **Degradation problem:**
  - 그렇다면 왜 Layer가 더 깊은 (deeper) NN이 얕은 (shallow) NN보다 training 성능이 떨어지는가?
  - 만약에 **"deeper NN"의 마지막 layer**들이 단순히 **identity mapping ($x \rightarrow x$)을 학습**하였다면 **shallow NN와 동일해진다**
  - **즉, shallow NN**보다 **deeper NN이** 못할 이유가 없다.

He, K., Zhang, X., Ren, S. and Sun, J., 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition* (pp. 770-778).

Convolutional Neural Network

ResNet

- Degradation problem:
  - 왜 Layer가 더 깊은 (deeper) NN이 얕은 (shallow) NN보다 training 성능이 떨어지는가?
  - 만약에 “deeper NN”의 마지막 layer들이 단순히 identity mapping ($x \rightarrow x$)을 학습하였다면 shallow NN보다 못할 이유가 없다.
  - 혹시 Neural Network의 Layer들이 **identity mapping을 학습하는데 어려움을 겪는게 아닐까?**

He, K., Zhang, X., Ren, S. and Sun, J., 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition* (pp. 770-778).

Convolutional Neural Network

ResNet

- Degradation problem:
  - 왜 Layer가 더 깊은 (deeper) NN이 얕은 (shallow) NN보다 training 성능이 떨어지는가?
  - 만약에 “deeper NN”의 마지막 layer들이 단순히 identity mapping ($x \rightarrow x$)을 학습하였다면 shallow NN보다 못할 이유가 없다.
  - 혹시 Neural Network의 Layer들이 identity mapping을 학습하는데 어려움을 겪는게 아닐까?
  - **Residual Connection 제안됨!**

He, K., Zhang, X., Ren, S. and Sun, J., 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition* (pp. 770-778).

ACADENTIAL

Convolutional Neural Network

ResNet

Residual Connection:

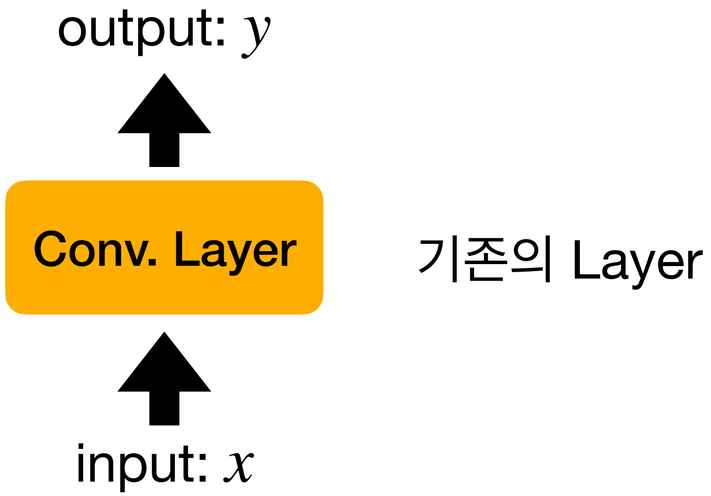
- **기존의 NN layer의 출력값에 인풋값을 더해주는 것.**

Convolutional Neural Network

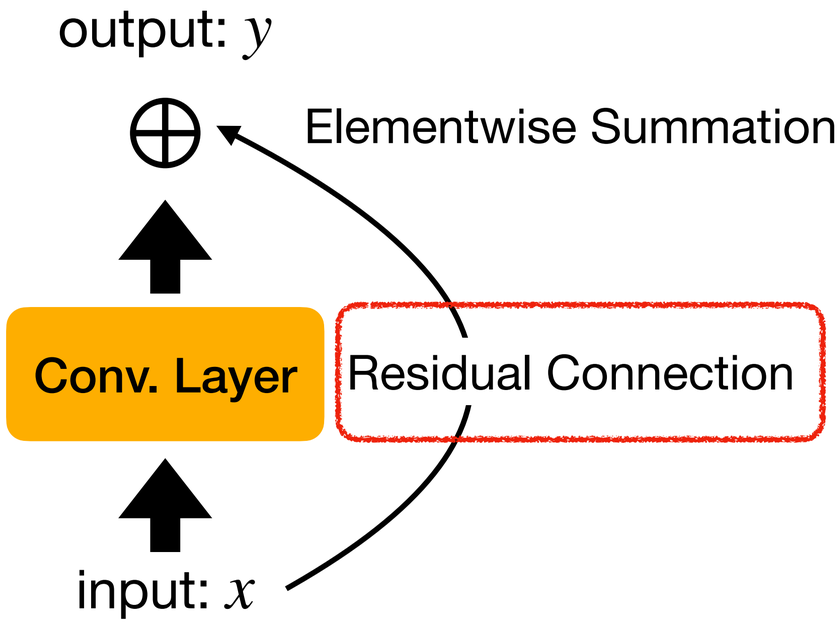
ResNet

Residual Connection:

- **기존의 NN layer의 출력값에 인풋값을 더해주는 것.**



기존의 Layer

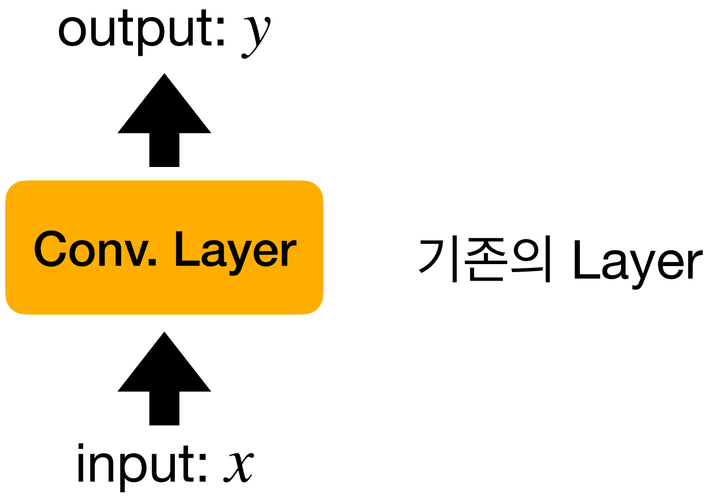


Convolutional Neural Network

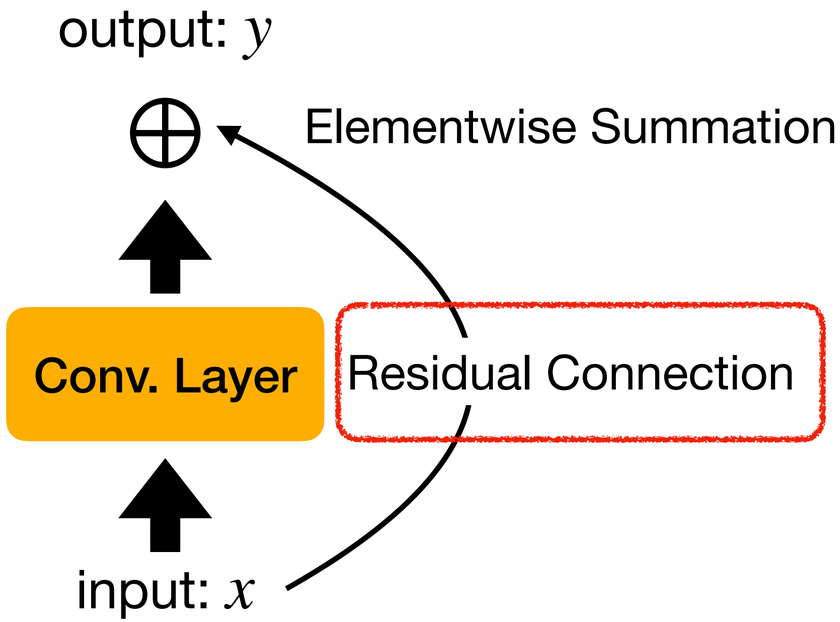
ResNet

Residual Connection:

- **기존의 NN layer의 출력값에 인풋값을 더해주는 것.**
- 그렇다면 **어떤 효과**가 있는 것인가?



기존의 Layer



Convolutional Neural Network

ResNet

Residual Connection은 어떤 효과가 있는 것인가?

- 만약에 최종적으로 학습하려는 함수가 "$y = H(x)$"라고 가정해보자.
- 기존의 CNN에서는 Convolutional Layer로 $y = H(x)$을 바로 학습하려고 함.

Convolutional Neural Network

ResNet

Convolutional Neural Network

ResNet

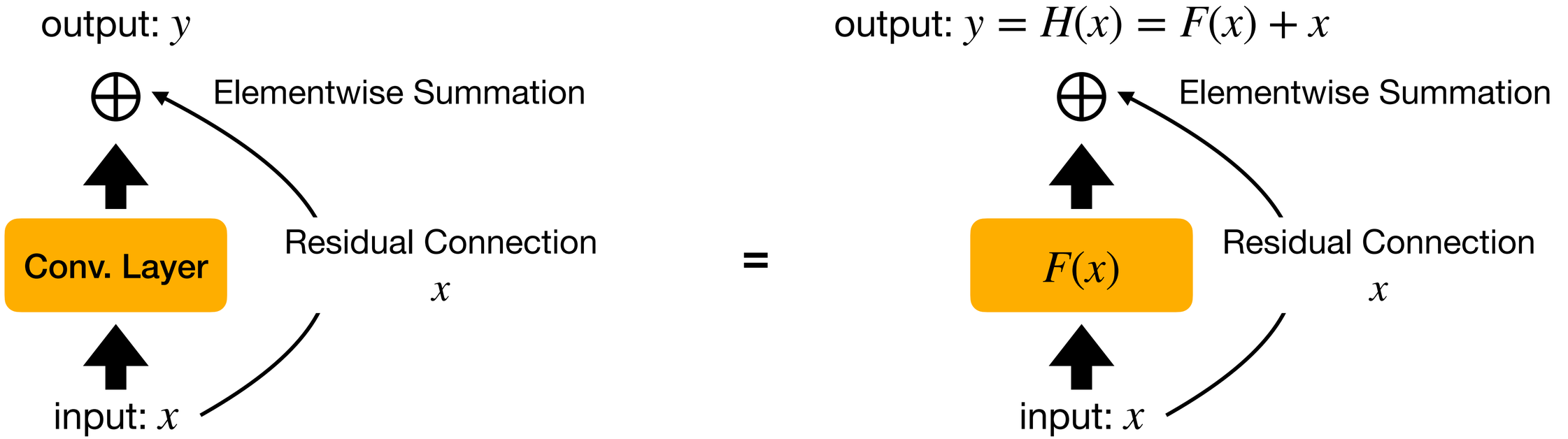
Residual Connection은 어떤 효과가 있는 것인가?

- ResNet에서 Convolutional Layer은 "$F(x) = H(x) - x$"을 학습하게 된다.
- 즉, $H(x)$**와** x**간의 차이**를 학습하는 셈이다.
- 만약에 Layer가 너무 깊어져 버리는 경우, F(x)은 단순히 0을 학습하면 됨.

ACADENTIAL

Convolutional Neural Network

ResNet

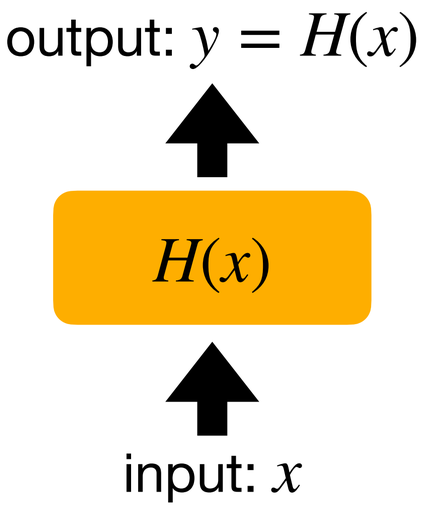


$$F(x) = H(x) - x$$

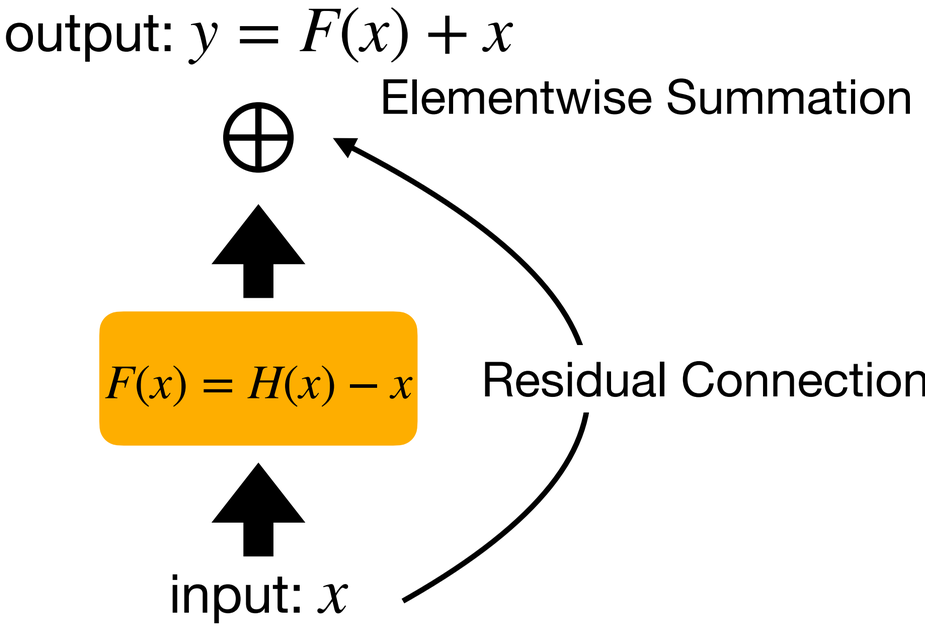
Convolutional Neural Network

ResNet

논문의 저자: "$H(x) - x$을 학습하는 것이 $H(x)$을 학습하는 것보다 쉬울 것이다"



기존의 Layer



Convolutional Neural Network

ResNet

method	top-1 err.	top-5 err.
VGG [41] (ILSVRC'14)	-	8.43†
GoogLeNet [44] (ILSVRC'14)	-	7.89
VGG [41] (v5)	24.4	7.1
PReLU-net [13]	21.59	5.71
BN-inception [16]	21.99	5.81
ResNet-34 B	21.84	5.71
ResNet-34 C	21.53	5.60
ResNet-50	20.74	5.25
ResNet-101	19.87	4.60
ResNet-152	**19.38**	**4.49**

Table 4. Error rates (%) of **single-model** results on the ImageNet validation set (except † reported on the test set).

VGG와 GoogLeNet (InceptionNet)과
비교해서 더 개선된 성능 확보!

ACADENTIAL

14-12. DenseNet

Convolutional Neural Network

DenseNet

어떤 문제를 풀려고 하는가?

- (ResNet과 동일한 motivation) **Neural Network가 깊어질수록 앞단의 Layer들은 gradient가 vanish할 수 있다.**
- **Layer들간의 Information Flow (≈Gradient)을 극대화하자!**

ACADENTIAL

Convolutional Neural Network

DenseNet

어떤 문제를 풀려고 하는가?

- (ResNet과 동일한 motivation) **Neural Network가 깊어질수록 앞단의 Layer들은 gradient가 vanish할 수 있다.**
- **Layer들간의 Information Flow (≈Gradient)을 극대화하자!**
- 어떻게 하면 **layer들간의 information flow**을 **극대화**할 수 있을까?

layer들간의 connectivity (연결량) 을 늘리자!

Convolutional Neural Network

DenseNet

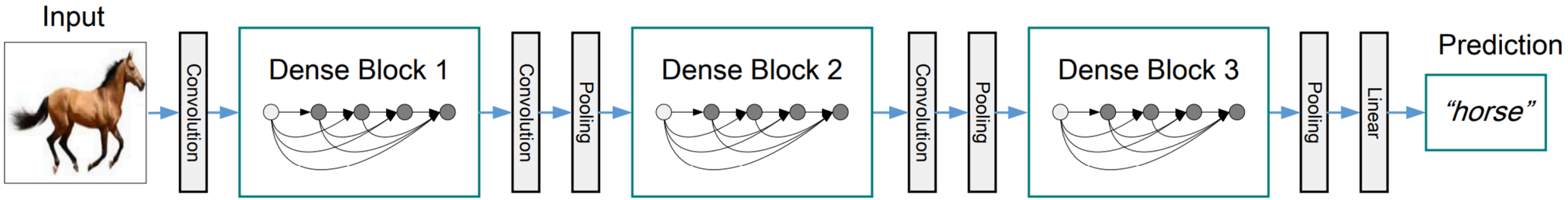


Figure 2: A deep DenseNet with three dense blocks. The layers between two adjacent blocks are referred to as transition layers and change feature-map sizes via convolution and pooling.

Convolutional Neural Network

DenseNet

- DenseNet의 architecture:
  - Block을 구성하는 각 CNN Layer은 **동일한 block**에 속해있고, **선행하는 layer들의 출력값**의 **concatenation**을 입력받음.
  - 즉, 각 CNN layer은 해당 block의 다른 layer들과 “densely connected”되어 있다.

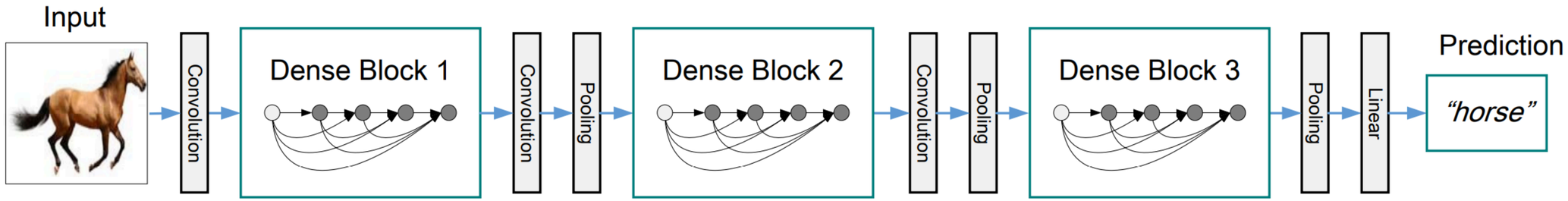


Figure 2: A deep DenseNet with three dense blocks. The layers between two adjacent blocks are referred to as transition layers and change feature-map sizes via convolution and pooling.

Convolutional Neural Network

DenseNet

- DenseNet의 architecture:
  - Block을 구성하는 각 CNN Layer은 **동일한 block**에 속해있고, **선행하는 layer들의 출력값**의 **concatenation**을 입력받음.
  - 즉, 각 CNN layer은 해당 block의 다른 layer들과 “densely connected”되어 있다.
  - 예를 들어, N번째 layer은 (N-1)개의 Layer와 연결됨!
  - 그리고 각 layer의 input feature = $k_0 + (l-1)k$

k_0=dense block의 input channel 수,

k=각 conv. layer의 output channel 수

Convolutional Neural Network

DenseNet

- DenseNet의 architecture (참고 사항):
  - Bottleneck으로 (1x1) conv. layer을 사용함.
  - BN → ReLU → Conv(1x1) → BN → ReLU → Conv(3x3) 으로 각 layer가 구성되어 있음.
  - 원래 Conv(1x1)에서 channel 수를 "$k_0 + (l-1)k$" 에서 "$4k$"로 축소하여 parameter efficiency을 늘렸다!

ACADENTIAL

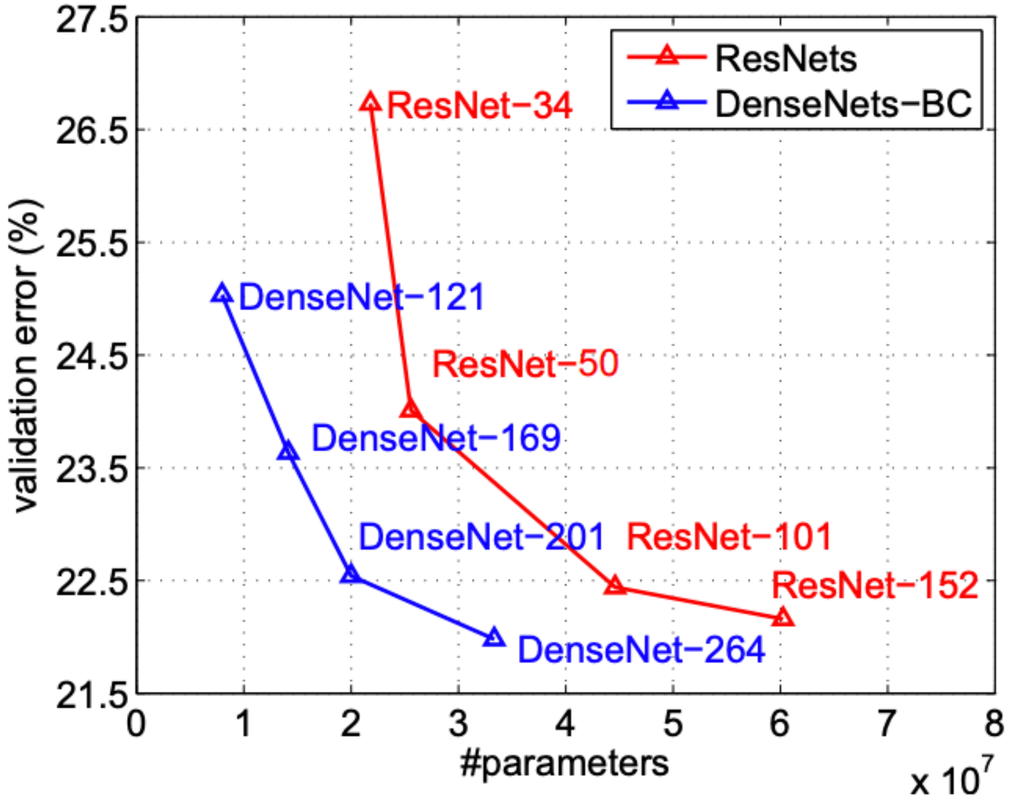
Convolutional Neural Network

DenseNet

- DenseNet의 장점:
  - Dense connectivity을 통해서 gradient flow가 원활하다! → easy to train
  - Bottleneck layer로 parameter efficiency가 높다.
  - feature들을 “재활용”하기 때문에 모델을 더 적은 parameter로도 representational power가 더 크다.

Convolutional Neural Network

DenseNet



ResNet과 비교했을시, 동일 parameter 대비 성능이 DenseNet이 더 높다!

Figure 3: Comparison of the DenseNets and ResNets top-1 error rates (single-crop testing) on the ImageNet validation dataset as a function of learned parameters (*left*)

ACADENTIAL

14-13. PyTorch로 구현된 VGGNet 뜯어보기

ACADENTIAL

14-14. PyTorch로 구현된 VGGNet 뜯어보기

ACADENTIAL

14-15. 직접 만든 CNN 모델, VGGNet, ResNet을 활용한 CV 프로젝트

ACADENTIAL

14-16. Section Summary

ACADENTIAL

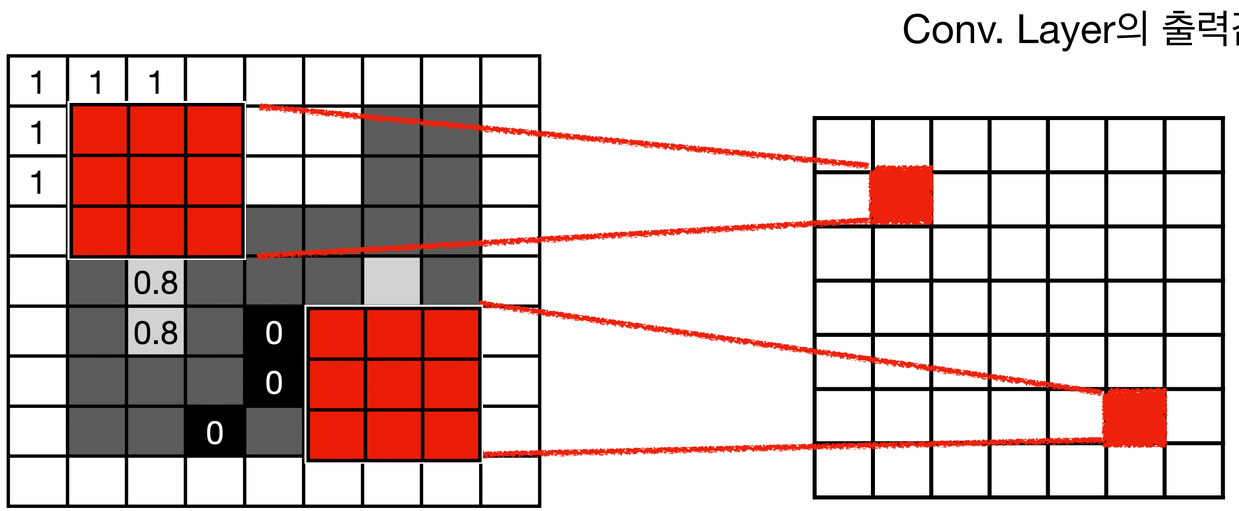
Section Summary

CNN - 이미지 데이터의 특성

1. (Colored) Image == 3D tensor
2. Translation Equivariance
3. Hierarchical한 Feature

Section Summary

CNN - Convolutional Layer의 작동원리



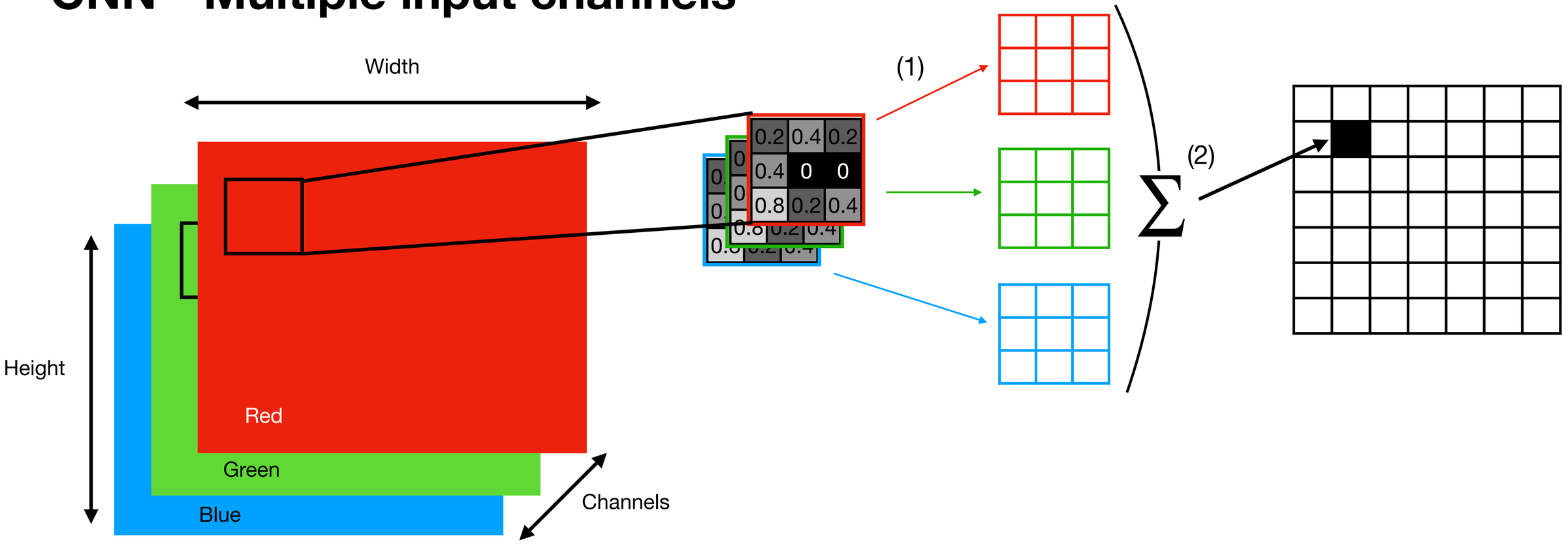
$$y = \mathbf{w} * \mathbf{x} + b$$

* == convolution operation

Conv. Layer은 Translation Equivariance을 만족시킨다!

Section Summary

CNN - Multiple input channels

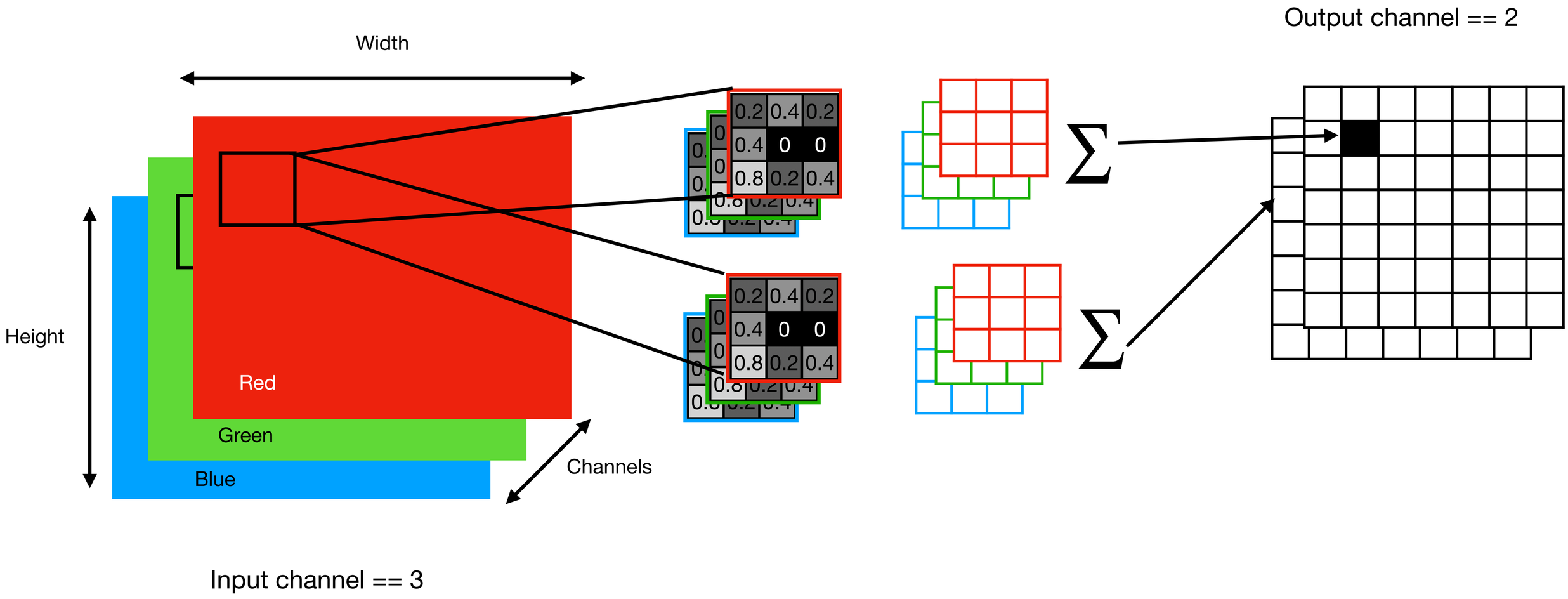


3차원의 tensor of shape == (Height, Width, Channel)

ACADENTIAL

Section Summary

CNN - Multiple output channels



ACADENTIAL

Section Summary

Variants of Convolutional layer

Variants:

- **Dilated operation**: “확장된” Convolutional kernel
- **Strided operation**: “여러 칸을 건너띄는” Conv. kernel
- **Padding**: input feature의 주변부 (boundary)에 “특정값들을 추가해 주는 것”.
- **Pooling**: input feature을 더 작은 크기로 “sub sampling”하여 줄여주는 것

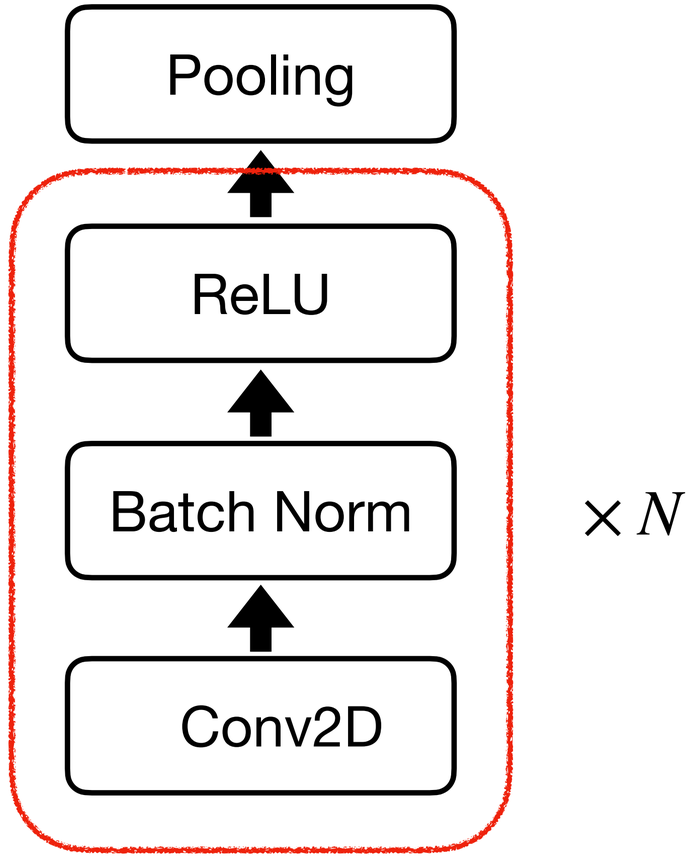
ACADENTIAL

Section Summary

Stack of CNN (구성)

일반적인 구성:

Conv2d → Batch Norm → ReLU → Pooling



ACADENTIAL

Section Summary

	특징
LeNet	CNN의 가장 초창기 모델 2개의 Convolutional Layer, 3개의 Fully Connected Layer로 구성되어 있음.
AlexNet	
VGGNet	

	특징
Inception Net	
ResNet	
DenseNet	

ACADENTIAL

Section Summary

	특징
LeNet	CNN의 가장 초창기 모델 2개의 Convolutional Layer, 3개의 Fully Connected Layer로 구성되어 있음.
AlexNet	2개의 CNN Branch로 나눠져 있음. (Tanh 대신에) ReLU activation을 사용함. 5개의 Conv Layer들과 3개의 FC layer들로 구성됨. 3개의 Max Pooling으로 구성됨.
VGGNet	

	특징
Inception Net	
ResNet	
DenseNet	

ACADENTIAL

Section Summary

	특징
LeNet	CNN의 가장 초창기 모델 2개의 Convolutional Layer, 3개의 Fully Connected Layer로 구성되어 있음.
AlexNet	2개의 CNN Branch로 나눠져 있음. (Tanh 대신에) ReLU activation을 사용함. 5개의 Conv Layer들과 3개의 FC layer들로 구성됨. 3개의 Max Pooling으로 구성됨.
VGGNet	**Conv Kernel의 크기를 (3x3)으로 줄이고,** **더 많은 Conv Layer (16~19개) 들을 쌓음.** → **Parameter Efficiency 개선** 5개의 Max Pooling으로 구성됨

	특징
Inception Net	
ResNet	
DenseNet	

ACADENTIAL

Section Summary

	특징
LeNet	CNN의 가장 초창기 모델 2개의 Convolutional Layer, 3개의 Fully Connected Layer로 구성되어 있음.
AlexNet	2개의 CNN Branch로 나눠져 있음. (Tanh 대신에) ReLU activation을 사용함. 5개의 Conv Layer들과 3개의 FC layer들로 구성됨. 3개의 Max Pooling으로 구성됨.
VGGNet	**Conv Kernel의 크기를 (3x3)으로 줄이고,** **더 많은 Conv Layer (16~19개) 들을 쌓음.** → **Parameter Efficiency 개선** 5개의 Max Pooling으로 구성됨

	특징
Inception Net	**(1x1, 3x3, 5x5) Conv Kernel**와 **3x3 Max Pooling**을 **병렬적**으로 쌓고 출력된 feature을 **concatenate**한다. **(1x1) convolution**을 사용해서 **Feature의 Dimension**을 **축소**. Intermediate Layer에서 **Auxiliary Loss**을 사용.
ResNet	
DenseNet	

ACADENTIAL

Section Summary

	특징
LeNet	CNN의 가장 초창기 모델 2개의 Convolutional Layer, 3개의 Fully Connected Layer로 구성되어 있음.
AlexNet	2개의 CNN Branch로 나눠져 있음. (Tanh 대신에) ReLU activation을 사용함. 5개의 Conv Layer들과 3개의 FC layer들로 구성됨. 3개의 Max Pooling으로 구성됨.
VGGNet	**Conv Kernel의 크기를 (3x3)으로 줄이고,** **더 많은 Conv Layer (16~19개) 들을 쌓음.** → **Parameter Efficiency 개선** 5개의 Max Pooling으로 구성됨

	특징
Inception Net	**(1x1, 3x3, 5x5) Conv Kernel**와 **3x3 Max Pooling**을 **병렬적**으로 쌓고 출력된 feature을 **concatenate**한다. **(1x1) convolution**을 사용해서 **Feature의 Dimension**을 **축소**. Intermediate Layer에서 **Auxiliary Loss**을 사용.
ResNet	Vanishing Gradient / Exploding Gradient의 문제를 해결하고자 **Residual Connection**을 제안
DenseNet	

ACADENTIAL

Section Summary

	특징
LeNet	CNN의 가장 초창기 모델 2개의 Convolutional Layer, 3개의 Fully Connected Layer로 구성되어 있음.
AlexNet	2개의 CNN Branch로 나눠져 있음. (Tanh 대신에) ReLU activation을 사용함. 5개의 Conv Layer들과 3개의 FC layer들로 구성됨. 3개의 Max Pooling으로 구성됨.
VGGNet	**Conv Kernel의 크기를 (3x3)으로 줄이고,** **더 많은 Conv Layer (16~19개) 들을 쌓음.** → **Parameter Efficiency 개선** 5개의 Max Pooling으로 구성됨

	특징
Inception Net	**(1x1, 3x3, 5x5) Conv Kernel**와 **3x3 Max Pooling**을 **병렬적**으로 쌓고 출력된 feature을 **concatenate**한다. **(1x1) convolution**을 사용해서 **Feature의 Dimension**을 **축소**. Intermediate Layer에서 **Auxiliary Loss**을 사용.
ResNet	Vanishing Gradient / Exploding Gradient의 문제를 해결하고자 **Residual Connection**을 제안
DenseNet	Vanishing Gradient의 문제를 해결하고자 **Layer들간의 Connectivity을 늘림**. Block에서 **N번째 Conv Layer**은 (같은 Block에 속한) **N-1개의 이전 Layer들의 출력값**들의 **Concatenation**을 입력받음.

ACADENTIAL

Next Up!

ACADENTIAL

Next Up

Recurrent Neural Networks

- 이번 Section에서는 이미지 데이터에 대한 모델인 CNN을 살펴보았다.
- 그렇다면, “시계열 데이터” (Times series)은 어떻게 모델링 할 수 있을까?

ACADENTIAL

Next Up

Recurrent Neural Networks

- 이번 챕터에서는 이미지 데이터에 대한 모델인 CNN을 살펴보았다.
- 그렇다면, “시계열 데이터” (Times series)은 어떻게 모델링 할 수 있을까?

Recurrent Neural Networks (RNN)

다음 섹션에서는 RNN에 대해서 살펴보자!

ACADENTIAL

Appendix

ACADENTIAL

VGGNet 뜯어보기

VGG11

```
VGG(
  (features): Sequential(
    (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): ReLU(inplace=True)
    (2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (3): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (4): ReLU(inplace=True)
    (5): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (6): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (7): ReLU(inplace=True)
    (8): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (9): ReLU(inplace=True)
    (10): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (11): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (12): ReLU(inplace=True)
    (13): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (14): ReLU(inplace=True)
    (15): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (16): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (17): ReLU(inplace=True)
    (18): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (19): ReLU(inplace=True)
    (20): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (avgpool): AdaptiveAvgPool2d(output_size=(7, 7))
  (classifier): Sequential(
    (0): Linear(in_features=25088, out_features=4096, bias=True)
    (1): ReLU(inplace=True)
    (2): Dropout(p=0.5, inplace=False)
    (3): Linear(in_features=4096, out_features=4096, bias=True)
    (4): ReLU(inplace=True)
    (5): Dropout(p=0.5, inplace=False)
    (6): Linear(in_features=4096, out_features=1000, bias=True)
  )
)
```

VGG11_BN

```
VGG(
  (features): Sequential(
    (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): ReLU(inplace=True)
    (3): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (4): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (5): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (6): ReLU(inplace=True)
    (7): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (8): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (9): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (10): ReLU(inplace=True)
    (11): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (12): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (13): ReLU(inplace=True)
    (14): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (15): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (16): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (17): ReLU(inplace=True)
    (18): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (19): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (20): ReLU(inplace=True)
    (21): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (22): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (23): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (24): ReLU(inplace=True)
    (25): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (26): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (27): ReLU(inplace=True)
    (28): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (avgpool): AdaptiveAvgPool2d(output_size=(7, 7))
  (classifier): Sequential(
    (0): Linear(in_features=25088, out_features=4096, bias=True)
    (1): ReLU(inplace=True)
    (2): Dropout(p=0.5, inplace=False)
    (3): Linear(in_features=4096, out_features=4096, bias=True)
    (4): ReLU(inplace=True)
    (5): Dropout(p=0.5, inplace=False)
    (6): Linear(in_features=4096, out_features=1000, bias=True)
  )
)
```

VGG11

각 CNN Block

```
VGG(
  (features): Sequential(
    (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): ReLU(inplace=True)
    (2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (3): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (4): ReLU(inplace=True)
    (5): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (6): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (7): ReLU(inplace=True)
    (8): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (9): ReLU(inplace=True)
    (10): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (11): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (12): ReLU(inplace=True)
    (13): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (14): ReLU(inplace=True)
    (15): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (16): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (17): ReLU(inplace=True)
    (18): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (19): ReLU(inplace=True)
    (20): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (avgpool): AdaptiveAvgPool2d(output_size=(7, 7))
  (classifier): Sequential(
    (0): Linear(in_features=25088, out_features=4096, bias=True)
    (1): ReLU(inplace=True)
    (2): Dropout(p=0.5, inplace=False)
    (3): Linear(in_features=4096, out_features=4096, bias=True)
    (4): ReLU(inplace=True)
    (5): Dropout(p=0.5, inplace=False)
    (6): Linear(in_features=4096, out_features=1000, bias=True)
  )
)
```

VGG11_BN

```
VGG(
  (features): Sequential(
    (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): ReLU(inplace=True)
    (3): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (4): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (5): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (6): ReLU(inplace=True)
    (7): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (8): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (9): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (10): ReLU(inplace=True)
    (11): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (12): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (13): ReLU(inplace=True)
    (14): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (15): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (16): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (17): ReLU(inplace=True)
    (18): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (19): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (20): ReLU(inplace=True)
    (21): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (22): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (23): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (24): ReLU(inplace=True)
    (25): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (26): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (27): ReLU(inplace=True)
    (28): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (avgpool): AdaptiveAvgPool2d(output_size=(7, 7))
  (classifier): Sequential(
    (0): Linear(in_features=25088, out_features=4096, bias=True)
    (1): ReLU(inplace=True)
    (2): Dropout(p=0.5, inplace=False)
    (3): Linear(in_features=4096, out_features=4096, bias=True)
    (4): ReLU(inplace=True)
    (5): Dropout(p=0.5, inplace=False)
    (6): Linear(in_features=4096, out_features=1000, bias=True)
  )
)
```

VGG11

```
VGG(
  (features): Sequential(
    (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): ReLU(inplace=True)
    (2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (3): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (4): ReLU(inplace=True)
    (5): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (6): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (7): ReLU(inplace=True)
    (8): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (9): ReLU(inplace=True)
    (10): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (11): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (12): ReLU(inplace=True)
    (13): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (14): ReLU(inplace=True)
    (15): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (16): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (17): ReLU(inplace=True)
    (18): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (19): ReLU(inplace=True)
    (20): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (avgpool): AdaptiveAvgPool2d(output_size=(7, 7))
  (classifier): Sequential(
    (0): Linear(in_features=25088, out_features=4096, bias=True)
    (1): ReLU(inplace=True)
    (2): Dropout(p=0.5, inplace=False)
    (3): Linear(in_features=4096, out_features=4096, bias=True)
    (4): ReLU(inplace=True)
    (5): Dropout(p=0.5, inplace=False)
    (6): Linear(in_features=4096, out_features=1000, bias=True)
  )
)
```

Conv. Layer: 8개

Linear Layer (FC): 3개

→ 8+3 = 11

VGG11_BN

```
VGG(
  (features): Sequential(
    (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): ReLU(inplace=True)
    (3): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (4): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (5): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (6): ReLU(inplace=True)
    (7): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (8): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (9): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (10): ReLU(inplace=True)
    (11): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (12): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (13): ReLU(inplace=True)
    (14): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (15): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (16): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (17): ReLU(inplace=True)
    (18): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (19): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (20): ReLU(inplace=True)
    (21): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (22): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (23): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (24): ReLU(inplace=True)
    (25): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (26): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (27): ReLU(inplace=True)
    (28): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (avgpool): AdaptiveAvgPool2d(output_size=(7, 7))
  (classifier): Sequential(
    (0): Linear(in_features=25088, out_features=4096, bias=True)
    (1): ReLU(inplace=True)
    (2): Dropout(p=0.5, inplace=False)
    (3): Linear(in_features=4096, out_features=4096, bias=True)
    (4): ReLU(inplace=True)
    (5): Dropout(p=0.5, inplace=False)
    (6): Linear(in_features=4096, out_features=1000, bias=True)
  )
)
```

VGG11

```
VGG(
  (features): Sequential(
    (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): ReLU(inplace=True)
    (2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (3): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (4): ReLU(inplace=True)
    (5): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (6): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (7): ReLU(inplace=True)
    (8): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (9): ReLU(inplace=True)
    (10): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (11): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (12): ReLU(inplace=True)
    (13): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (14): ReLU(inplace=True)
    (15): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (16): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (17): ReLU(inplace=True)
    (18): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (19): ReLU(inplace=True)
    (20): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (avgpool): AdaptiveAvgPool2d(output_size=(7, 7))
  (classifier): Sequential(
    (0): Linear(in_features=25088, out_features=4096, bias=True)
    (1): ReLU(inplace=True)
    (2): Dropout(p=0.5, inplace=False)
    (3): Linear(in_features=4096, out_features=4096, bias=True)
    (4): ReLU(inplace=True)
    (5): Dropout(p=0.5, inplace=False)
    (6): Linear(in_features=4096, out_features=1000, bias=True)
  )
)
```

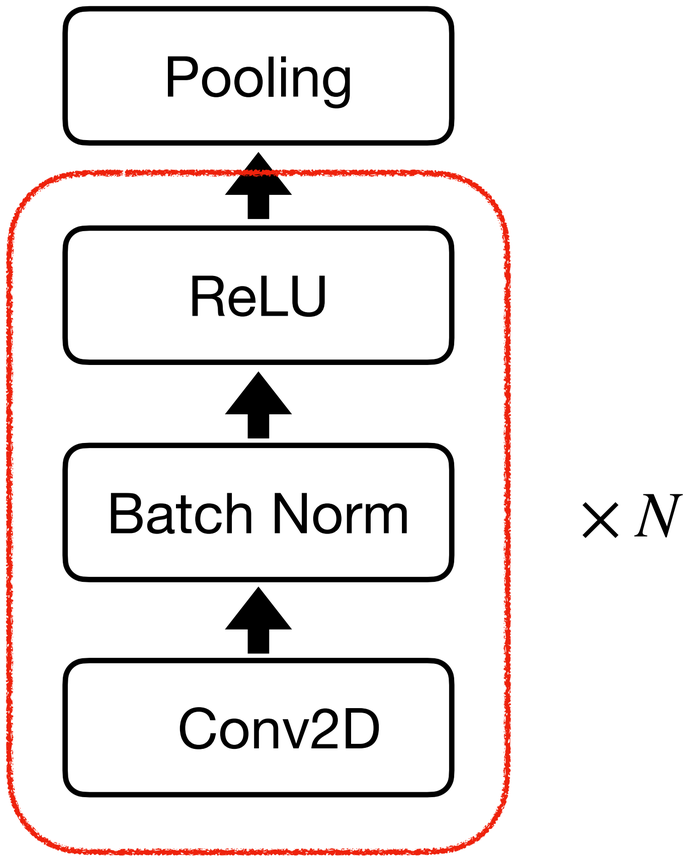
VGG11_BN에서는 Batch Normalization이 추가되었다!

VGG11_BN

```
VGG(
  (features): Sequential(
    (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): ReLU(inplace=True)
    (3): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (4): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (5): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (6): ReLU(inplace=True)
    (7): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (8): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (9): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (10): ReLU(inplace=True)
    (11): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (12): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (13): ReLU(inplace=True)
    (14): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (15): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (16): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (17): ReLU(inplace=True)
    (18): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (19): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (20): ReLU(inplace=True)
    (21): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (22): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (23): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (24): ReLU(inplace=True)
    (25): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (26): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (27): ReLU(inplace=True)
    (28): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (avgpool): AdaptiveAvgPool2d(output_size=(7, 7))
  (classifier): Sequential(
    (0): Linear(in_features=25088, out_features=4096, bias=True)
    (1): ReLU(inplace=True)
    (2): Dropout(p=0.5, inplace=False)
    (3): Linear(in_features=4096, out_features=4096, bias=True)
    (4): ReLU(inplace=True)
    (5): Dropout(p=0.5, inplace=False)
    (6): Linear(in_features=4096, out_features=1000, bias=True)
  )
)
```

일반적인 구성:

Conv2d → Batch Norm → ReLU → Pooling



$\times N$

VGG11_BN

```
VGG(
  (features): Sequential(
    (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): ReLU(inplace=True)
    (3): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (4): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (5): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (6): ReLU(inplace=True)
    (7): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (8): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (9): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (10): ReLU(inplace=True)
    (11): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (12): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (13): ReLU(inplace=True)
    (14): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (15): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (16): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (17): ReLU(inplace=True)
    (18): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (19): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (20): ReLU(inplace=True)
    (21): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (22): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (23): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (24): ReLU(inplace=True)
    (25): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (26): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (27): ReLU(inplace=True)
    (28): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (avgpool): AdaptiveAvgPool2d(output_size=(7, 7))
  (classifier): Sequential(
    (0): Linear(in_features=25088, out_features=4096, bias=True)
    (1): ReLU(inplace=True)
    (2): Dropout(p=0.5, inplace=False)
    (3): Linear(in_features=4096, out_features=4096, bias=True)
    (4): ReLU(inplace=True)
    (5): Dropout(p=0.5, inplace=False)
    (6): Linear(in_features=4096, out_features=1000, bias=True)
  )
)
```

VGG11

```
VGG(
  (features): Sequential(
    (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): ReLU(inplace=True)
    (2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (3): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (4): ReLU(inplace=True)
    (5): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (6): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (7): ReLU(inplace=True)
    (8): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (9): ReLU(inplace=True)
    (10): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (11): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (12): ReLU(inplace=True)
    (13): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (14): ReLU(inplace=True)
    (15): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (16): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (17): ReLU(inplace=True)
    (18): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (19): ReLU(inplace=True)
    (20): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (avgpool): AdaptiveAvgPool2d(output_size=(7, 7))
  (classifier): Sequential(
    (0): Linear(in_features=25088, out_features=4096, bias=True)
    (1): ReLU(inplace=True)
    (2): Dropout(p=0.5, inplace=False)
    (3): Linear(in_features=4096, out_features=4096, bias=True)
    (4): ReLU(inplace=True)
    (5): Dropout(p=0.5, inplace=False)
    (6): Linear(in_features=4096, out_features=1000, bias=True)
  )
)
```

VGG11과 VGG13의 차이점:

- **VGG11**: channel이 **64→128**인 Conv. Layer 1개로 구성.
- **VGG13**: **64→64**, **64→128**, **128→128**인 Conv. Layer 3개로 구성.

VGG13

```
VGG(
  (features): Sequential(
    (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): ReLU(inplace=True)
    (2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (3): ReLU(inplace=True)
    (4): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (5): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (6): ReLU(inplace=True)
    (7): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (8): ReLU(inplace=True)
    (9): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (10): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (11): ReLU(inplace=True)
    (12): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (13): ReLU(inplace=True)
    (14): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (15): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (16): ReLU(inplace=True)
    (17): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (18): ReLU(inplace=True)
    (19): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (20): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (21): ReLU(inplace=True)
    (22): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (23): ReLU(inplace=True)
    (24): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (avgpool): AdaptiveAvgPool2d(output_size=(7, 7))
  (classifier): Sequential(
    (0): Linear(in_features=25088, out_features=4096, bias=True)
    (1): ReLU(inplace=True)
    (2): Dropout(p=0.5, inplace=False)
    (3): Linear(in_features=4096, out_features=4096, bias=True)
    (4): ReLU(inplace=True)
    (5): Dropout(p=0.5, inplace=False)
    (6): Linear(in_features=4096, out_features=1000, bias=True)
  )
)
```

```python
from torchvision.models import vgg11,

# 예를 들어서 vgg11 모델을 불러오는 코드는 다음과 같습니다.
model = vgg11()
```

```python
@register_model()
@handle_legacy_interface(weights=("pretrained", VGG11_Weights.IMAGENET1K_V1))
def vgg11(
    *, weights: Optional[VGG11_Weights] = None, progress: bool = True, **kwargs: Any
) -> VGG:
    """VGG-11 from `Very Deep Convolutional Networks for Large-Scale Image Recognition <h

    Args:
        weights (:class:`~torchvision.models.VGG11_Weights`, optional): The
            pretrained weights to use. See
            :class:`~torchvision.models.VGG11_Weights` below for
            more details, and possible values. By default, no pre-trained
            weights are used.
        progress (bool, optional): If True, displays a progress bar of the
            download to stderr. Default is True.
        **kwargs: parameters passed to the ``torchvision.models.vgg.VGG``
            base class. Please refer to the `source code
            <https://github.com/pytorch/vision/blob/main/torchvision/models/vgg.py>`_
            for more details about this class.

    .. autoclass:: torchvision.models.VGG11_Weights
        :members:
    """
    weights = VGG11_Weights.verify(weights)

    return _vgg("A", False, weights, progress, **kwargs)
```

```python
from torchvision.models import vgg11,

# 예를 들어서 vgg11 모델을 불러오는 코드는 다음과 같습니다.
model = vgg11()
```

```python
@register_model()
@handle_legacy_interface(weights=("pretrained", VGG11_Weights.IMAGENET1K_V1))
def vgg11(
    *, weights: Optional[VGG11_Weights] = None, progress: bool = True, **kwargs: Any
) -> VGG:
    """VGG-11 from `Very Deep Convolutional Networks for Large-Scale Image Recognition <h

    Args:
        weights (:class:`~torchvision.models.VGG11_Weights`, optional): The
            pretrained weights to use. See
            :class:`~torchvision.models.VGG11_Weights` below for
            more details, and possible values. By default, no pre-trained
            weights are used.
        progress (bool, optional): If True, displays a progress bar of the
            download to stderr. Default is True.
        **kwargs: parameters passed to the ``torchvision.models.vgg.VGG``
            base class. Please refer to the `source code
            <https://github.com/pytorch/vision/blob/main/torchvision/models/vgg.py>`_
            for more details about this class.

    .. autoclass:: torchvision.models.VGG11_Weights
        :members:
    """
    weights = VGG11_Weights.verify(weights)

    return _vgg("A", False, weights, progress, **kwargs)
```

```python
@register_model()
@handle_legacy_interface(weights=("pretrained", VGG11_Weights.IMAGENET1K_V1))
def vgg11(
    *, weights: Optional[VGG11_Weights] = None, progress: bool = True, **kwargs: Any
) -> VGG:
    """VGG-11 from `Very Deep Convolutional Networks for Large-Scale Image Recognition <h

    Args:
        weights (:class:`~torchvision.models.VGG11_Weights`, optional): The
            pretrained weights to use. See
            :class:`~torchvision.models.VGG11_Weights` below for
            more details, and possible values. By default, no pre-trained
            weights are used.
        progress (bool, optional): If True, displays a progress bar of the
            download to stderr. Default is True.
        **kwargs: parameters passed to the ``torchvision.models.vgg.VGG``
            base class. Please refer to the `source code
            <https://github.com/pytorch/vision/blob/main/torchvision/models/vgg.py>`_
            for more details about this class.

    .. autoclass:: torchvision.models.VGG11_Weights
        :members:
    """
    weights = VGG11_Weights.verify(weights)

    return _vgg("A", False, weights, progress, **kwargs)
```

```python
def _vgg(
    cfg: str,
    batch_norm: bool,
    weights: Optional[WeightsEnum],
    progress: bool,
    **kwargs: Any
) -> VGG:
    if weights is not None:
        kwargs["init_weights"] = False
        if weights.meta["categories"] is not None:
            _ovewrite_named_param(
                kwargs, "num_classes", len(weights.meta["categories"])
            )
    model = VGG(make_layers(cfgs[cfg], batch_norm=batch_norm), **kwargs)
    if weights is not None:
        model.load_state_dict(weights.get_state_dict(progress=progress))
    return model
```

```python
@register_model()
@handle_legacy_interface(weights=("pretrained", VGG11_Weights.IMAGENET1K_V1))
def vgg11(
    *, weights: Optional[VGG11_Weights] = None, progress: bool = True, **kwargs: Any
) -> VGG:
    """VGG-11 from `Very Deep Convolutional Networks for Large-Scale Image Recognition <h

    Args:
        weights (:class:`~torchvision.models.VGG11_Weights`, optional): The
            pretrained weights to use. See
            :class:`~torchvision.models.VGG11_Weights` below for
            more details, and possible values. By default, no pre-trained
            weights are used.
        progress (bool, optional): If True, displays a progress bar of the
            download to stderr. Default is True.
        **kwargs: parameters passed to the ``torchvision.models.vgg.VGG``
            base class. Please refer to the `source code
            <https://github.com/pytorch/vision/blob/main/torchvision/models/vgg.py>`_
            for more details about this class.

    .. autoclass:: torchvision.models.VGG11_Weights
        :members:
    """
    weights = VGG11_Weights.verify(weights)

    return _vgg("A", False, weights, progress, **kwargs)
```

```python
def _vgg(
    cfg: str,
    batch_norm: bool,
    weights: Optional[WeightsEnum],
    progress: bool,
    **kwargs: Any
) -> VGG:
    if weights is not None:
        kwargs["init_weights"] = False
        if weights.meta["categories"] is not None:
            _ovewrite_named_param(
                kwargs, "num_classes", len(weights.meta["categories"])
            )
    model = VGG(make_layers(cfgs[cfg], batch_norm=batch_norm), **kwargs)
    if weights is not None:
        model.load_state_dict(weights.get_state_dict(progress=progress))
    return model
```

```python
cfgs: Dict[str, List[Union[str, int]]] = {
    "A": [64, "M", 128, "M", 256, 256, "M", 512, 512, "M", 512, 512, "M"],
    "B": [64, 64, "M", 128, 128, "M", 256, 256, "M", 512, 512, "M", 512, 512, "M"],
    "D": [
        64,
        64,
        "M",
        128,
        128,
        "M",
        256,
        256,
        256,
        "M",
        512,
        512,
        512,
        "M",
        512,
        512,
        512,
        "M",
    ],
    "E": [
        64,
        64,
        "M",
        128,
        128,
        "M",
```

```python
def make_layers(cfg: List[Union[str, int]], batch_norm: bool = False) -> nn.Sequential:
    layers: List[nn.Module] = []
    in_channels = 3
    for v in cfg:
        if v == "M":
            layers += [nn.MaxPool2d(kernel_size=2, stride=2)]
        else:
            v = cast(int, v)
            conv2d = nn.Conv2d(in_channels, v, kernel_size=3, padding=1)
            if batch_norm:
                layers += [conv2d, nn.BatchNorm2d(v), nn.ReLU(inplace=True)]
            else:
                layers += [conv2d, nn.ReLU(inplace=True)]
            in_channels = v
    return nn.Sequential(*layers)
```

```python
def _vgg(
    cfg: str,
    batch_norm: bool,
    weights: Optional[WeightsEnum],
    progress: bool,
    **kwargs: Any
) -> VGG:
    if weights is not None:
        kwargs["init_weights"] = False
        if weights.meta["categories"] is not None:
            _ovewrite_named_param(
                kwargs, "num_classes", len(weights.meta["categories"])
            )
    model = VGG(make_layers(cfgs[cfg], batch_norm=batch_norm), **kwargs)
    if weights is not None:
        model.load_state_dict(weights.get_state_dict(progress=progress))
    return model
```

```python
cfgs: Dict[str, List[Union[str, int]]] = {
    "A": [64, "M", 128, "M", 256, 256, "M", 512, 512, "M", 512, 512, "M"],
    "B": [64, 64, "M", 128, 128, "M", 256, 256, "M", 512, 512, "M", 512, 512, "M"],
    "D": [
        64,
        64,
        "M",
        128,
        128,
        "M",
        256,
        256,
        256,
        "M",
        512,
        512,
        512,
        "M",
        512,
        512,
        512,
        "M",
    ],
    "E": [
        64,
        64,
        "M",
        128,
        128,
        "M",
```

```python
class VGG(nn.Module):
    def __init__(
        self,
        features: nn.Module,
        num_classes: int = 1000,
        init_weights: bool = True,
        dropout: float = 0.5,
    ) -> None:
        super().__init__()
        _log_api_usage_once(self)
        self.features = features
        self.avgpool = nn.AdaptiveAvgPool2d((7, 7))
        self.classifier = nn.Sequential(
            nn.Linear(512 * 7 * 7, 4096),
            nn.ReLU(True),
            nn.Dropout(p=dropout),
            nn.Linear(4096, 4096),
            nn.ReLU(True),
            nn.Dropout(p=dropout),
            nn.Linear(4096, num_classes),
        )
        if init_weights:
            for m in self.modules():
                if isinstance(m, nn.Conv2d):
                    nn.init.kaiming_normal_(
                        m.weight, mode="fan_out", nonlinearity="relu"
                    )
                    if m.bias is not None:
                        nn.init.constant_(m.bias, 0)
                elif isinstance(m, nn.BatchNorm2d):
                    nn.init.constant_(m.weight, 1)
                    nn.init.constant_(m.bias, 0)
                elif isinstance(m, nn.Linear):
                    nn.init.normal_(m.weight, 0, 0.01)
                    nn.init.constant_(m.bias, 0)

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        x = self.features(x)
        x = self.avgpool(x)
        x = torch.flatten(x, 1)
        x = self.classifier(x)
        return x
```

```python
def _vgg(
    cfg: str,
    batch_norm: bool,
    weights: Optional[WeightsEnum],
    progress: bool,
    **kwargs: Any
) -> VGG:
    if weights is not None:
        kwargs["init_weights"] = False
        if weights.meta["categories"] is not None:
            _ovewrite_named_param(
                kwargs, "num_classes", len(weights.meta["categories"])
            )
    model = VGG(make_layers(cfgs[cfg], batch_norm=batch_norm), **kwargs)
    if weights is not None:
        model.load_state_dict(weights.get_state_dict(progress=progress))
    return model
```

ACADENTIAL

ResNet 뜯어보기

ResNet 18

```
ResNet(
  (conv1): Conv2d(3, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3), bias=False)
  (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (relu): ReLU(inplace=True)
  (maxpool): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1, ceil_mode=False)
  (layer1): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
    (1): BasicBlock(
      (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
  )
  (layer2): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(64, 128, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (downsample): Sequential(
        (0): Conv2d(64, 128, kernel_size=(1, 1), stride=(2, 2), bias=False)
        (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      )
    )
    (1): BasicBlock(
      (conv1): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
  )
  (layer3): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(128, 256, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (downsample): Sequential(
        (0): Conv2d(128, 256, kernel_size=(1, 1), stride=(2, 2), bias=False)
        (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      )
    )
    (1): BasicBlock(
      (conv1): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
  )
  (layer4): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(256, 512, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (downsample): Sequential(
        (0): Conv2d(256, 512, kernel_size=(1, 1), stride=(2, 2), bias=False)
        (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      )
    )
    (1): BasicBlock(
      (conv1): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
  )
  (avgpool): AdaptiveAvgPool2d(output_size=(1, 1))
  (fc): Linear(in_features=512, out_features=1000, bias=True)
)
```

ResNet 18

```
ResNet(
  (conv1): Conv2d(3, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3), bias=False)
  (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (relu): ReLU(inplace=True)
  (maxpool): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1, ceil_mode=False)
  (layer1): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
    (1): BasicBlock(
      (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
  )
  (layer2): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(64, 128, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (downsample): Sequential(
        (0): Conv2d(64, 128, kernel_size=(1, 1), stride=(2, 2), bias=False)
        (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      )
    )
    (1): BasicBlock(
      (conv1): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
  )
  (layer3): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(128, 256, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (downsample): Sequential(
        (0): Conv2d(128, 256, kernel_size=(1, 1), stride=(2, 2), bias=False)
        (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      )
    )
    (1): BasicBlock(
      (conv1): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
  )
  (layer4): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(256, 512, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (downsample): Sequential(
        (0): Conv2d(256, 512, kernel_size=(1, 1), stride=(2, 2), bias=False)
        (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      )
    )
    (1): BasicBlock(
      (conv1): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
  )
  (avgpool): AdaptiveAvgPool2d(output_size=(1, 1))
  (fc): Linear(in_features=512, out_features=1000, bias=True)
)
```

여러 개의 ResNet Block들이 반복해서 쌓여 있는 형태!

ResNet 18

```
ResNet(
  (conv1): Conv2d(3, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3), bias=False)
  (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (relu): ReLU(inplace=True)
  (maxpool): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1, ceil_mode=False)
  (layer1): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
    (1): BasicBlock(
      (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
  )
  (layer2): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(64, 128, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (downsample): Sequential(
        (0): Conv2d(64, 128, kernel_size=(1, 1), stride=(2, 2), bias=False)
        (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      )
    )
    (1): BasicBlock(
      (conv1): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
  )
```

```python
class BasicBlock(nn.Module):
    expansion: int = 1

    def __init__(
        self,
        inplanes: int,
        planes: int,
        stride: int = 1,
        downsample: Optional[nn.Module] = None,
        norm_layer: Optional[Callable[..., nn.Module]] = None,
    ) -> None:
        super().__init__()
        self.conv1 = conv3x3(inplanes, planes, stride)
        self.bn1 = norm_layer(planes)
        self.relu = nn.ReLU(inplace=True)
        self.conv2 = conv3x3(planes, planes)
        self.bn2 = norm_layer(planes)
        self.downsample = downsample
        self.stride = stride

    def forward(self, x: Tensor) -> Tensor:
        identity = x

        out = self.conv1(x)
        out = self.bn1(out)
        out = self.relu(out)

        out = self.conv2(out)
        out = self.bn2(out)

        if self.downsample is not None:
            identity = self.downsample(x)

        out += identity
        out = self.relu(out)

        return out
```

ResNet 18

```
ResNet(
  (conv1): Conv2d(3, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3), bias=False)
  (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (relu): ReLU(inplace=True)
  (maxpool): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1, ceil_mode=False)
  (layer1): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
    (1): BasicBlock(
      (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
  )
  (layer2): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(64, 128, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (downsample): Sequential(
        (0): Conv2d(64, 128, kernel_size=(1, 1), stride=(2, 2), bias=False)
        (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      )
    )
    (1): BasicBlock(
      (conv1): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
  )
```

```python
class BasicBlock(nn.Module):
    expansion: int = 1

    def __init__(
        self,
        inplanes: int,
        planes: int,
        stride: int = 1,
        downsample: Optional[nn.Module] = None,
        norm_layer: Optional[Callable[..., nn.Module]] = None,
    ) -> None:
        super().__init__()
        self.conv1 = conv3x3(inplanes, planes, stride)
        self.bn1 = norm_layer(planes)
        self.relu = nn.ReLU(inplace=True)
        self.conv2 = conv3x3(planes, planes)
        self.bn2 = norm_layer(planes)
        self.downsample = downsample
        self.stride = stride

    def forward(self, x: Tensor) -> Tensor:
        identity = x

        out = self.conv1(x)
        out = self.bn1(out)
        out = self.relu(out)

        out = self.conv2(out)
        out = self.bn2(out)

        if self.downsample is not None:
            identity = self.downsample(x)

        out += identity
        out = self.relu(out)

        return out
```

VGG와 마찬가지로,

Conv 2D → BatchNorm → ReLU 의 구조

ResNet 18

```
ResNet(
  (conv1): Conv2d(3, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3), bias=False)
  (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (relu): ReLU(inplace=True)
  (maxpool): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1, ceil_mode=False)
  (layer1): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
    (1): BasicBlock(
      (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
  )
  (layer2): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(64, 128, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (downsample): Sequential(
        (0): Conv2d(64, 128, kernel_size=(1, 1), stride=(2, 2), bias=False)
        (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      )
    )
    (1): BasicBlock(
      (conv1): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
  )
```

```python
class BasicBlock(nn.Module):
    expansion: int = 1

    def __init__(
        self,
        inplanes: int,
        planes: int,
        stride: int = 1,
        downsample: Optional[nn.Module] = None,
        norm_layer: Optional[Callable[..., nn.Module]] = None,
    ) -> None:
        super().__init__()
        self.conv1 = conv3x3(inplanes, planes, stride)
        self.bn1 = norm_layer(planes)
        self.relu = nn.ReLU(inplace=True)
        self.conv2 = conv3x3(planes, planes)
        self.bn2 = norm_layer(planes)
        self.downsample = downsample
        self.stride = stride

    def forward(self, x: Tensor) -> Tensor:
        identity = x

        out = self.conv1(x)
        out = self.bn1(out)
        out = self.relu(out)

        out = self.conv2(out)
        out = self.bn2(out)

        if self.downsample is not None:
            identity = self.downsample(x)

        out += identity
        out = self.relu(out)

        return out
```

ResNet 18

```
ResNet(
  (conv1): Conv2d(3, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3), bias=False)
  (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (relu): ReLU(inplace=True)
  (maxpool): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1, ceil_mode=False)
  (layer1): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
    (1): BasicBlock(
      (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
  )
  (layer2): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(64, 128, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (downsample): Sequential(
        (0): Conv2d(64, 128, kernel_size=(1, 1), stride=(2, 2), bias=False)
        (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      )
    )
    (1): BasicBlock(
      (conv1): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
  )
```

```python
class BasicBlock(nn.Module):
    expansion: int = 1

    def __init__(
        self,
        inplanes: int,
        planes: int,
        stride: int = 1,
        downsample: Optional[nn.Module] = None,
        norm_layer: Optional[Callable[..., nn.Module]] = None,
    ) -> None:
        super().__init__()
        self.conv1 = conv3x3(inplanes, planes, stride)
        self.bn1 = norm_layer(planes)
        self.relu = nn.ReLU(inplace=True)
        self.conv2 = conv3x3(planes, planes)
        self.bn2 = norm_layer(planes)
        self.downsample = downsample
        self.stride = stride

    def forward(self, x: Tensor) -> Tensor:
        identity = x

        out = self.conv1(x)
        out = self.bn1(out)
        out = self.relu(out)

        out = self.conv2(out)
        out = self.bn2(out)

        if self.downsample is not None:
            identity = self.downsample(x)

        out += identity
        out = self.relu(out)

        return out
```

Residual Connection

ResNet 18

```
ResNet(
  (conv1): Conv2d(3, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3), bias=False)
  (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (relu): ReLU(inplace=True)
  (maxpool): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1, ceil_mode=False)
  (layer1): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
    (1): BasicBlock(
      (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
  )
  (layer2): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(64, 128, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (downsample): Sequential(
        (0): Conv2d(64, 128, kernel_size=(1, 1), stride=(2, 2), bias=False)
        (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      )
    )
    (1): BasicBlock(
      (conv1): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
  )
```

VGGNet에서는 Max Pool을 통해 Feature Map 크기를 축소 (Downsample) 하였다.

ResNet에서는 1 x 1 Kernel, 2 x 2 Stride의 Conv. Layer을 사용한다!

Max Pool Layer은 Learnable한 Parameter을 가지지 않기 때문에, stride 2 x 2인 Conv. Layer로 축소하는 것이 더 큰 모델의 capacity을 확보할 수 있다.

ResNet 18

```
ResNet(
  (conv1): Conv2d(3, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3), bias=False)
  (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (relu): ReLU(inplace=True)
  (maxpool): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1, ceil_mode=False)
  (layer1): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
    (1): BasicBlock(
      (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
  )
  (layer2): Sequential(
```

ResNet 34

```
ResNet(
  (conv1): Conv2d(3, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3), bias=False)
  (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (relu): ReLU(inplace=True)
  (maxpool): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1, ceil_mode=False)
  (layer1): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
    (1): BasicBlock(
      (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
    (2): BasicBlock(
      (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
  )
  (layer2): Sequential(
```

ResNet 18과 34은 서로 어떤 차이가 있을까?

ResNet 18

```
ResNet(
  (conv1): Conv2d(3, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3), bias=False)
  (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (relu): ReLU(inplace=True)
  (maxpool): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1, ceil_mode=False)
  (layer1): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
    (1): BasicBlock(
      (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
  )
  (layer2): Sequential(
```

ResNet 34

```
ResNet(
  (conv1): Conv2d(3, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3), bias=False)
  (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (relu): ReLU(inplace=True)
  (maxpool): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1, ceil_mode=False)
  (layer1): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
    (1): BasicBlock(
      (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
    (2): BasicBlock(
      (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
  )
  (layer2): Sequential(
```

ResNet은 총 4개의 “Layer”로 구성되어 있는데,
각 Layer에서 ResNet Block의 개수가 각 모델마다 약간씩 다르게 구성되어 있다.

예를 들어,
ResNet 18에서는 첫번째 Layer에서 ResNet Block이 2개,
ResNet 34에서는 첫번째 Layer에서 ResNet Block이 3개이다.

Building Block

		ResNet 18	ResNet 34	ResNet 50	ResNet 101	ResNet 152
layer name	output size	18-layer	34-layer	50-layer	101-layer	152-layer
conv1	112×112	7×7, 64, stride 2				
		3×3 max pool, stride 2				
Layer 1 conv2_x	56×56	$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$
Layer 2 conv3_x	28×28	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times8$
Layer 3 conv4_x	14×14	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times23$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times36$
Layer 4 conv5_x	7×7	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$
	1×1	average pool, 1000-d fc, softmax				
FLOPs		1.8×10^9	3.6×10^9	3.8×10^9	7.6×10^9	11.3×10^9