

# Lecture 9 | CNN Architectures

오늘 알아볼 것 -

- AlexNet
- VGG
- GoogLeNet
- ResNet

## AlexNet

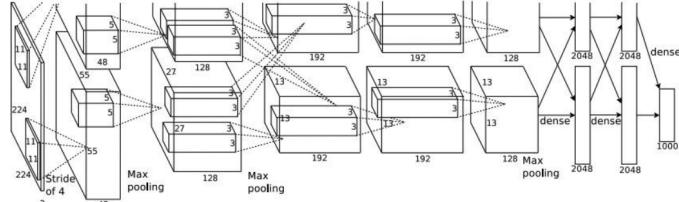
AlexNet은 아래 그림처럼 CONV layer , Max pooling layer, Full connected layer로 연결되어있는 형태를 말한다.

### Case Study: AlexNet

[Krizhevsky et al. 2012]

#### Architecture:

CONV1  
MAX POOL1  
NORM1  
CONV2  
MAX POOL2  
NORM2  
CONV3  
CONV4  
CONV5  
Max POOL3  
FC6  
FC7  
FC8



Q1.

Input:  $227 \times 227 \times 3$  images , First layer (CONV1): 96  $11 \times 11$  필터(stride = 4) 인 상황에서 output volume size는?

A1.

$(227 - 11) / 4 + 1 = 55$  이므로, output volume size =  $55 \times 55 \times 96$ 이다

Q2. 위의 layer의 parameter 총 개수는 몇개인가?

A2. 각 필터당  $11 \times 11 \times 3$  개 이므로,  $11 \times 11 \times 3 \times 96$ 개이다.

Q3. Second layer (POOL1):  $3 \times 3$  필터 (stride)인 상황에서 output volume size는?

A3. first layer의 output volume size가  $55 \times 55 \times 96$ 이다. 그러면 이를 input으로 생각하면  $(55-3)/2+1 = 27$ 이므로 output volume size =  $27 \times 27 \times 96$ 이다.

Q4. 위의 layer의 parameter 총 개수는 몇개인가?

A4. 0개. (pooling layer에는 parameter가 없다. 그저 max값을 뽑아내기만 할 뿐)

아래에는 input부터 시작하여 AlexNet의 전체 layer에 대한 정보가 주어져있다 시작은  $11 \times 11$  크기의 필터를 사용하였는데, 가면 갈수록 그 값이 작아지고 마지막에는  $3 \times 3$  크기의 필터를 사용하는 것을 볼 수 있다.

또, 우측 하단 부분을 보면 여러 설명들도 적혀있다. ReLU를 처음 사용했고, Norm layer도 있고, SGD Momentum, Learning rate 등에 대한 값들도 적혀있다.

이를 처음 사용할 때 GTX 580와 3GB의 메모리가 있는 컴퓨터를 이용하였다고 한다. 2개의 GPU를 이용하여, 각 GPU별로 절반 만큼의 depth씩 가지고 연산을 진행하곤 했다.

CONV1, CONV2, CONV4, CONV에서는 다른 특징에 대한 정보까지 함께 엮어서 하는 연산이 없기 때문에 각 GPU별 데이터 교환 없이 진행될 수 있다. 그러나 CONV3, FC6, FC7, FC8에서는 모든 특징들의 데이터를 필요로 했기 때문에 GPU간의 데이터 교환이 필요하다. FC layer의 각 뉴런들은 이전 layer의 모든 데이터들과 연결되어 있다는 사실을 알 수 있다.

이 AlexNet은 2012년 ImageNet Challenge에서 정확도를 급격하게 상승시켜 1등을 하였다.

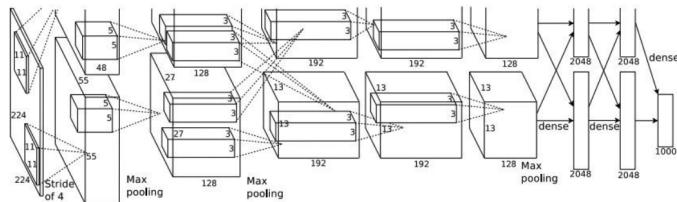
# Case Study: AlexNet

[Krizhevsky et al. 2012]

Full (simplified) AlexNet architecture:

[227x227x3] INPUT

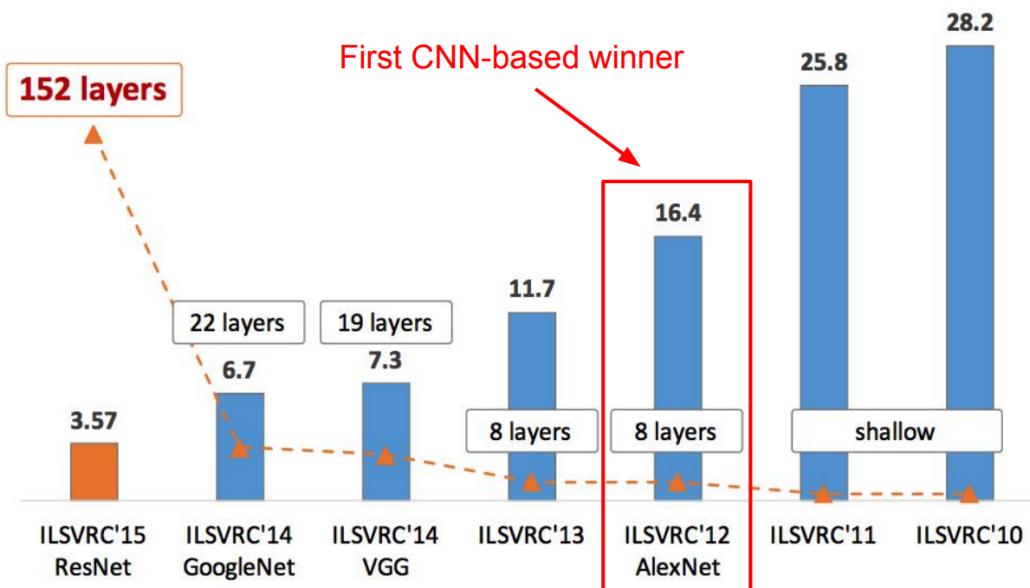
[55x55x96] CONV1: 96 11x11 filters at stride 4, pad 0  
 [27x27x96] MAX POOL1: 3x3 filters at stride 2  
 [27x27x96] NORM1: Normalization layer  
 [27x27x256] CONV2: 256 5x5 filters at stride 1, pad 2  
 [13x13x256] MAX POOL2: 3x3 filters at stride 2  
 [13x13x256] NORM2: Normalization layer  
 [13x13x384] CONV3: 384 3x3 filters at stride 1, pad 1  
 [13x13x384] CONV4: 384 3x3 filters at stride 1, pad 1  
 [13x13x256] CONV5: 256 3x3 filters at stride 1, pad 1  
 [6x6x256] MAX POOL3: 3x3 filters at stride 2  
 [4096] FC6: 4096 neurons  
 [4096] FC7: 4096 neurons  
 [1000] FC8: 1000 neurons (class scores)



## Details/Retrospectives:

- first use of ReLU
- used Norm layers (not common anymore)
- heavy data augmentation
- dropout 0.5
- batch size 128
- SGD Momentum 0.9
- Learning rate 1e-2, reduced by 10 manually when val accuracy plateaus
- L2 weight decay 5e-4
- 7 CNN ensemble: 18.2% -> 15.4%

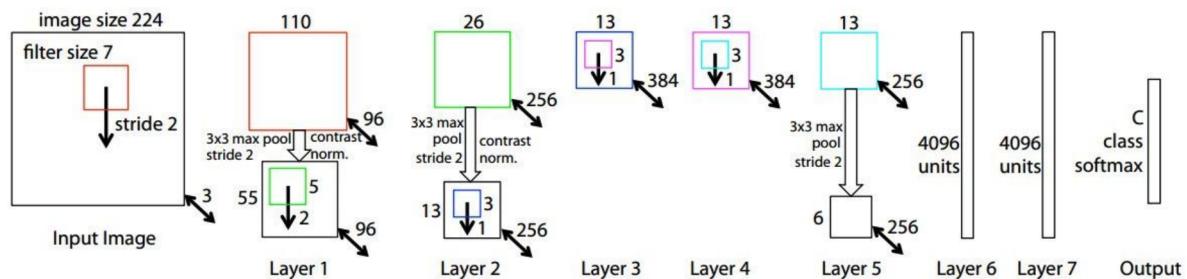
Figure copyright Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton, 2012. Reproduced with permission.



2013년에는 ZFNet이 1등을 했는데, 이는 AlexNet에서 몇몇 hyperparameter들을 향상 시켜 만들었다.

## ZFNet

[Zeiler and Fergus, 2013]



AlexNet but:

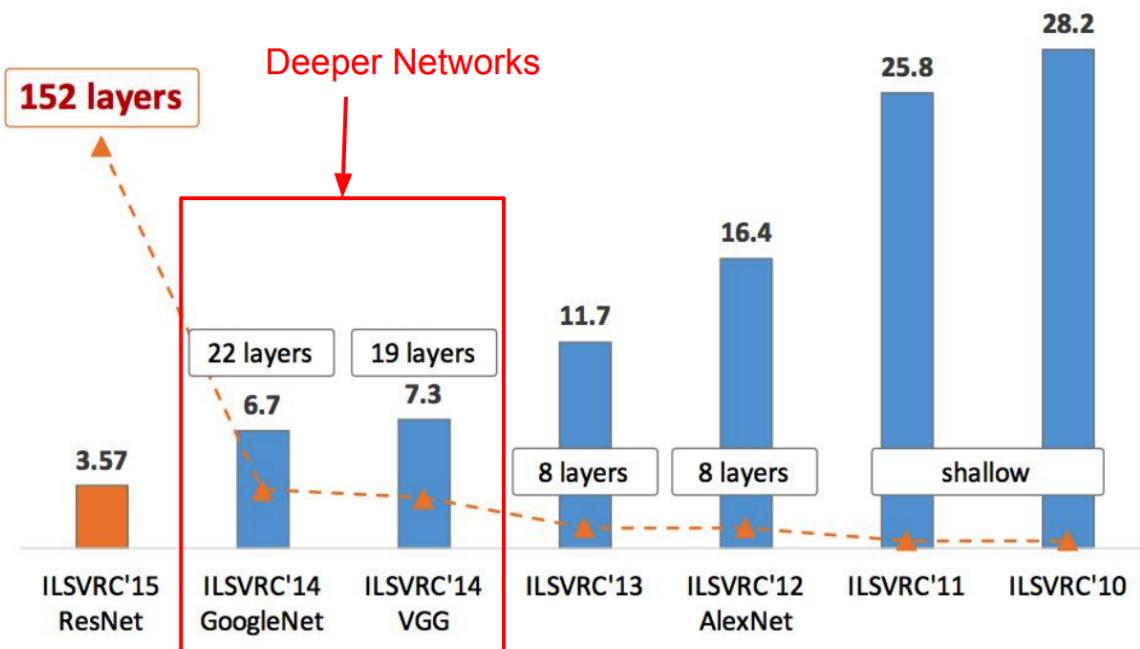
CONV1: change from (11x11 stride 4) to (7x7 stride 2)

CONV3,4,5: instead of 384, 384, 256 filters use 512, 1024, 512

ImageNet top 5 error: 16.4% → 11.7%

TODO: remake figure

이전의 AlexNet과 ZFNet은 8개의 layer로 진행했는데, 2014년부터 layer의 개수가 증가하기 시작했다. 2014년에 우승한 Network는 GoogleNet은 22개의 layer를 사용했고, 이와 유사한 정확도를 가진 VGG도 19개의 layer로 구성되어있다.



## VGGNet

이는 더 많은 layer로, 그리고 작은 크기의 filter를 사용했다. CONV에서 오직  $3 \times 3$  크기 (stride=1)의 필터를 사용했고, Max Pooling layer에서는  $2 \times 2$  크기 (stride = 2)를 사용했다.

## Case Study: VGGNet

[Simonyan and Zisserman, 2014]

Small filters, Deeper networks

8 layers (AlexNet)

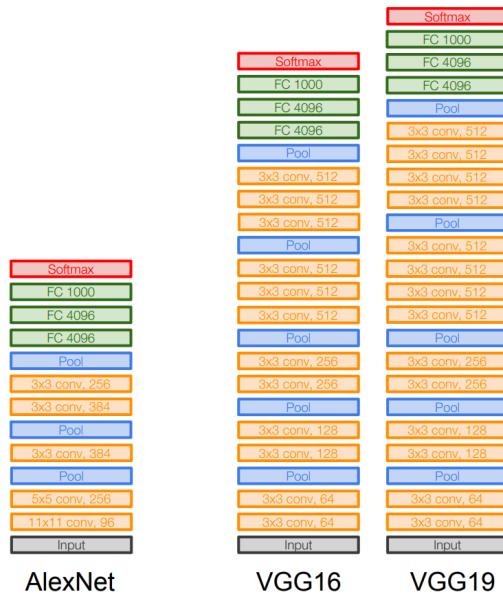
-> 16 - 19 layers (VGG16Net)

Only  $3 \times 3$  CONV stride 1, pad 1  
and  $2 \times 2$  MAX POOL stride 2

11.7% top 5 error in ILSVRC'13

(ZFNet)

-> 7.3% top 5 error in ILSVRC'14



Q1. 작은 필터를 이용한 이유는?

A1. 적은 필터를 이용하면 parameter의 개수가 적게 필요하다. 그럼에도 이를 여러 층을 쌓아 이용하면 큰 사이즈의 필터와 동일한 효과를 낼 수 있다.

$3 \times 3$  필터(stride 1)를 3개 이용하면  $7 \times 7$  필터를 이용하는 것과 동일한 효과를 볼 수 있다.  $3 \times 3$  필터를 사용한다고 했을 때, 첫 번째 layer에서는  $3 \times 3$  필터의 효과를 내지만, 2번째 layer에서는  $3 \times 3$  필터에 의해 모인 데이터를 다시 3개씩 묶어서 보기 때문에  $5 \times 5$ 의 크기의 데이터를 한번에 묶어서 보게 된다. 또, 3번째 layer에서는  $5 \times 5$  크기의 필터에 의해 모인 데이터를 3개씩 묶어서 보기 때문에  $7 \times 7$  크기의 데이터를 한번에 묶어서 볼 수 있게 된다. 여기서  $3 \times 3$  필터를 3개 사용하면 총 parameter은 27개지만,  $7 \times 7$  필터를 이용하면 49개의 parameter가 필요하다. 즉 작은 크기의 필터를 여러번 사용하는 것이 적은 수의 parameter를 사용한다는 것이다.

또, 여러 층이 될수록 점점 비선형이 되어 가서 더 좋은 결과를 낼 수도 있다.

아래는 전체 layer에서 사용하는 메모리와, parameter를 계산해놓은 것이다. 최종 부분을 보면 이미지당 96MB정도의 메모리와, 138M의 parameters가 필요하다. 이는 오직 forward만 계산한 것으로, backward까지 하려면 2배정도 곱해주면 된다.

대부분의 메모리는 CONV layer에서 사용되고, 대부분의 parameter들은 FC layer에서 사용되는 것도 알 수 있다.

INPUT: [224x224x3]	memory: 224*224*3=150K	params: 0	(not counting biases)
CONV3-64: [224x224x64]	memory: 224*224*64=3.2M	params: (3*3*3)*64 = 1,728	
CONV3-64: [224x224x64]	memory: 224*224*64=3.2M	params: (3*3*64)*64 = 36,864	
POOL2: [112x112x64]	memory: 112*112*64=800K	params: 0	
CONV3-128: [112x112x128]	memory: 112*112*128=1.6M	params: (3*3*64)*128 = 73,728	
CONV3-128: [112x112x128]	memory: 112*112*128=1.6M	params: (3*3*128)*128 = 147,456	
POOL2: [56x56x128]	memory: 56*56*128=400K	params: 0	
CONV3-256: [56x56x256]	memory: 56*56*256=800K	params: (3*3*128)*256 = 294,912	
CONV3-256: [56x56x256]	memory: 56*56*256=800K	params: (3*3*256)*256 = 589,824	
CONV3-256: [56x56x256]	memory: 56*56*256=800K	params: (3*3*256)*256 = 589,824	
POOL2: [28x28x256]	memory: 28*28*256=200K	params: 0	
CONV3-512: [28x28x512]	memory: 28*28*512=400K	params: (3*3*256)*512 = 1,179,648	
CONV3-512: [28x28x512]	memory: 28*28*512=400K	params: (3*3*512)*512 = 2,359,296	
CONV3-512: [28x28x512]	memory: 28*28*512=400K	params: (3*3*512)*512 = 2,359,296	
POOL2: [14x14x512]	memory: 14*14*512=100K	params: 0	
CONV3-512: [14x14x512]	memory: 14*14*512=100K	params: (3*3*512)*512 = 2,359,296	
CONV3-512: [14x14x512]	memory: 14*14*512=100K	params: (3*3*512)*512 = 2,359,296	
CONV3-512: [14x14x512]	memory: 14*14*512=100K	params: (3*3*512)*512 = 2,359,296	
POOL2: [7x7x512]	memory: 7*7*512=25K	params: 0	
FC: [1x1x4096]	memory: 4096	params: 7*7*512*4096 = 102,760,448	
FC: [1x1x4096]	memory: 4096	params: 4096*4096 = 16,777,216	
FC: [1x1x1000]	memory: 1000	params: 4096*1000 = 4,096,000	

VGG16

TOTAL memory: 24M \* 4 bytes ~= 96MB / image (only forward! ~\*2 for bwd)

TOTAL params: 138M parameters

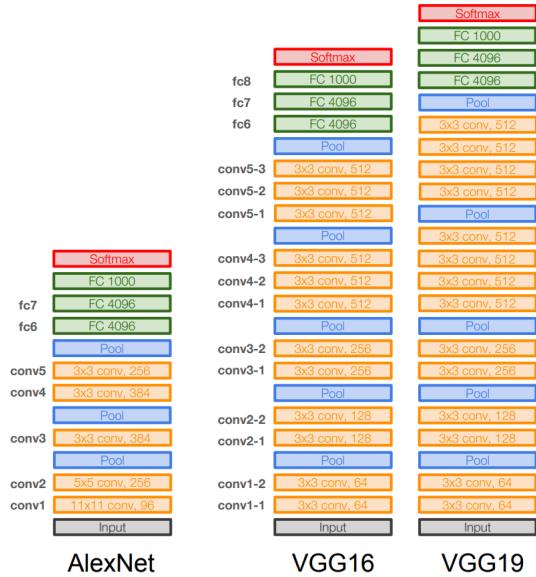
2014년에 ImageNet challenge에서 2등을 하였다. 아래는 VGGNet에 대한 몇몇 내용들을 적어놓았다. layer의 개수로 나누어 VGG16, VGG19로 나뉜다.

## Case Study: VGGNet

[Simonyan and Zisserman, 2014]

### Details:

- ILSVRC'14 2nd in classification, 1st in localization
- Similar training procedure as Krizhevsky 2012
- No Local Response Normalisation (LRN)
- Use VGG16 or VGG19 (VGG19 only slightly better, more memory)
- Use ensembles for best results
- FC7 features generalize well to other tasks



# GoogleNet

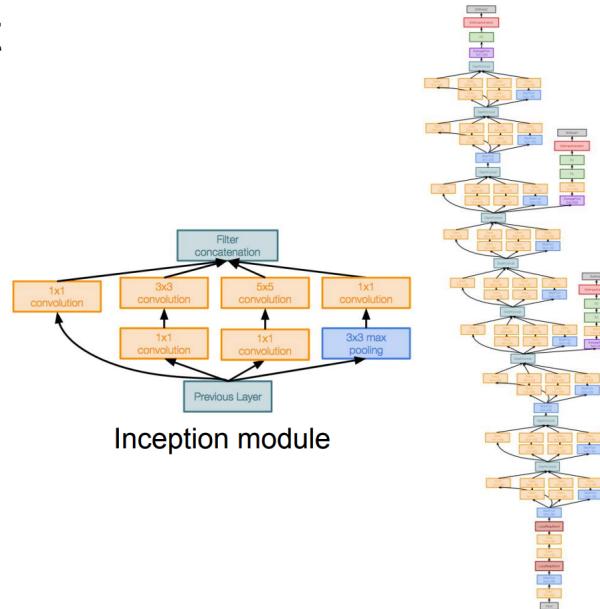
이는 VGG보다 더 깊은, 22개의 layer을 사용하는 network이다. 뒤에 더 설명할 Inception module을 사용했고, FC layer를 사용하지 않았다.

## Case Study: GoogLeNet

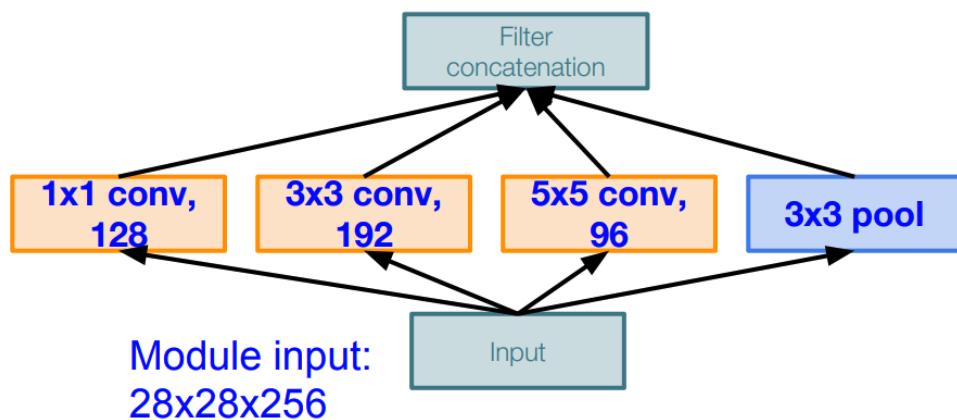
[Szegedy et al., 2014]

Deeper networks, with computational efficiency

- 22 layers
- Efficient “Inception” module
- No FC layers
- Only 5 million parameters!  
12x less than AlexNet
- ILSVRC’14 classification winner  
(6.7% top 5 error)



Inception Module : 여러 종류의 layer( $1 \times 1$  CONV,  $3 \times 3$  CONV,  $5 \times 5$  CONV,  $3 \times 3$  Max pool)들을 동일한 input에 대해 병렬적으로 통과시키고, 각 output들을 연결하여 하나의 output을 만들어 내는 형태이다.



Naive Inception module

Q. Inception Module의 문제는 무엇일까? (Hint : 복잡도)

Q1. 각각 layer들의 output size는 얼마인가?

A1.

$1 \times 1$  CONV :  $28 \times 28 \times 128$  size

$3 \times 3$  CONV :  $28 \times 28 \times 192$  size

$5 \times 5$  CONV :  $28 \times 28 \times 96$  size

$3 \times 3$  POOL :  $28 \times 28 \times 256$  size

( $1 \times 1$  CONV 이외에서는 크기를 보존하기 위해 여기서는 zero패딩을 이용한다)

전부를 연결하면,  $28 \times 28 \times 672$  size의 output이 만들어진다.

각 부분별로 연산량을 확인해보자.

$1 \times 1$  CONV :  $28 \times 28 \times 128 \times 1 \times 1 \times 256$

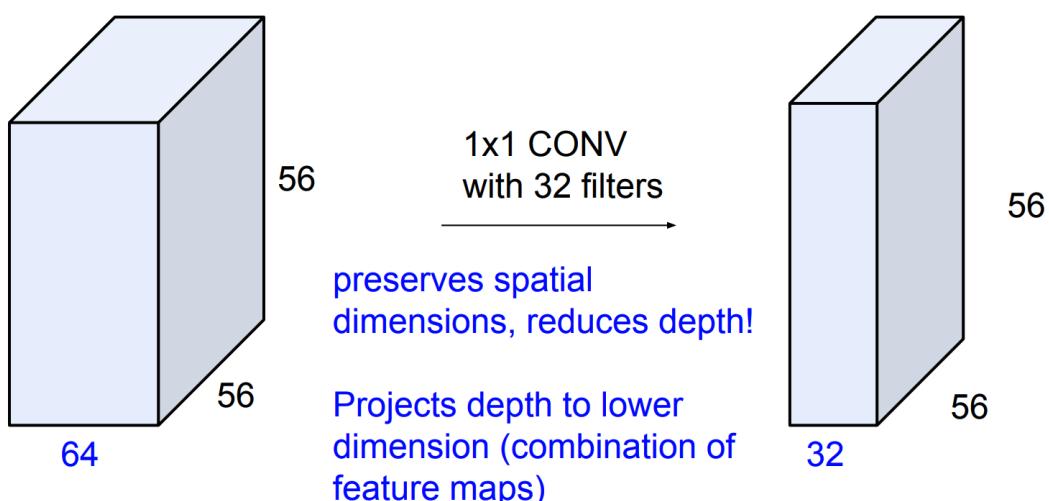
$3 \times 3$  CONV :  $28 \times 28 \times 192 \times 3 \times 3 \times 256$

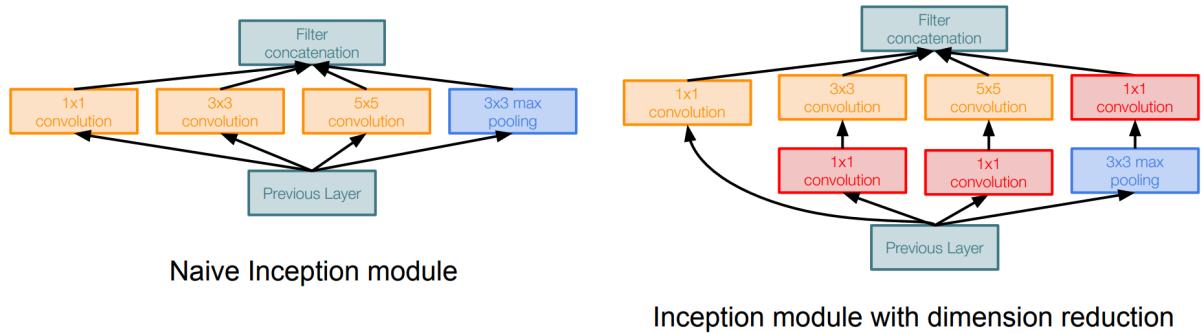
$5 \times 5$  CONV :  $28 \times 28 \times 96 \times 5 \times 5 \times 256$

이렇게 되면 최종적으로 854M개의 연산이 필요하다. 즉, 상당히 expensive한 연산임을 알 수 있다.

이를 해결하기 위한 구글의 해결책은 bottleneck layer를 이용하는 것이다. 이를 이용하여 차원 감소를 하여 연산 시간을 줄이고자 한다.

$1 \times 1$  layer를 이용하여 길이는 보존하되, depth를 줄이는 것이다. 이를 각 부분에 넣어서, depth를 감소시킴으로서 연산시간을 줄일 수 있다.

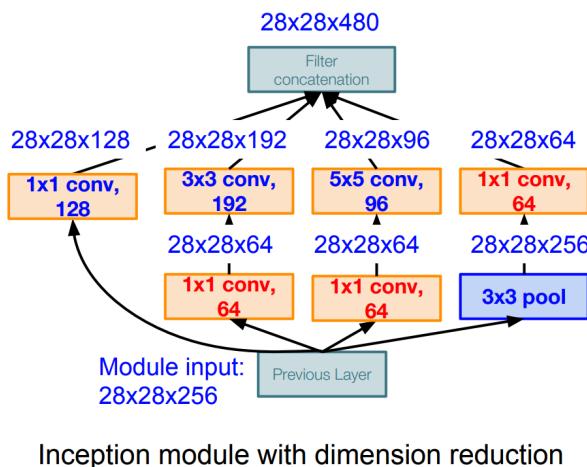




이전에는 854M의 연산이 필요했다면 이렇게 하면 358M의 연산으로 GoogleNet을 진행할 수 있다.

## Case Study: GoogLeNet

[Szegedy et al., 2014]



Using same parallel layers as naive example, and adding "1x1 conv, 64 filter" bottlenecks:

### Conv Ops:

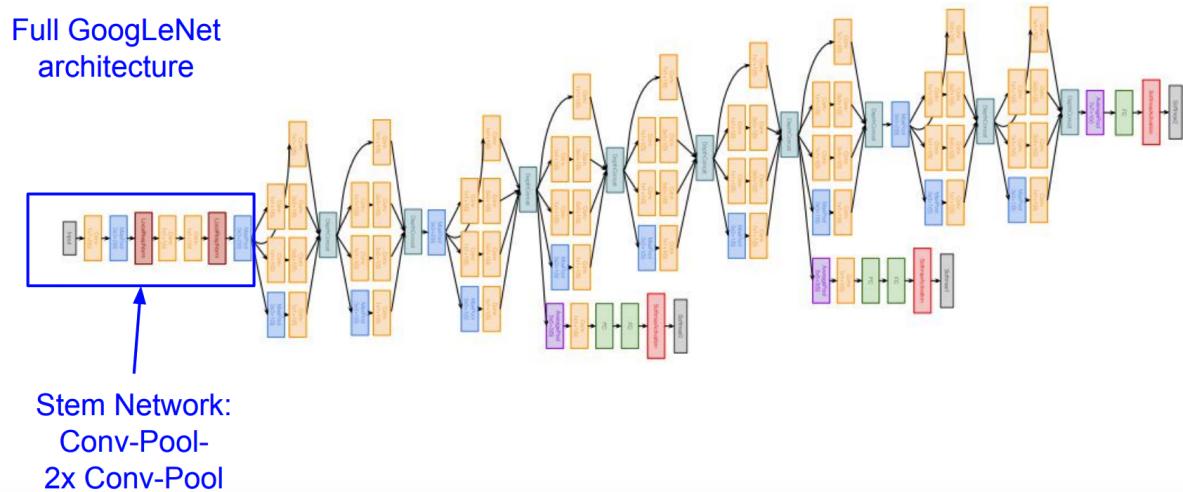
- [1x1 conv, 64] 28x28x64x1x1x256
- [1x1 conv, 64] 28x28x64x1x1x256
- [1x1 conv, 128] 28x28x128x1x1x256
- [3x3 conv, 192] 28x28x192x3x3x64
- [5x5 conv, 96] 28x28x96x5x5x64
- [1x1 conv, 64] 28x28x64x1x1x256

**Total: 358M ops**

Compared to 854M ops for naive version  
Bottleneck can also reduce depth after pooling layer

아래의 그림이 GoogleNet의 전체적인 형태이다. 처음에는 Stem Network로, conv와 pool을 이용하고, 중간에는 앞에서 보았던 Inception Module들을 여럿 이용하고, 마지막 output을 위한 Classifier 부분이 존재한다. 마지막 부분을 보면 FC layer가 없어진 것을 알 수 있다.

또, Inception Module의 중간중간에 보면 몇몇 layer들이 더 있는 것을 알 수 있다. 적은 수의 layer를 이용하여 결과를 내는 부분이다. 즉 총 3부분에서 output을 계산하여 더 좋은 결과를 내기 위한 방법이다. layer가 깊어질수록, gradient vanishing 문제가 발생할 수 있는데, 이런 것을 방지하는 부분이다.

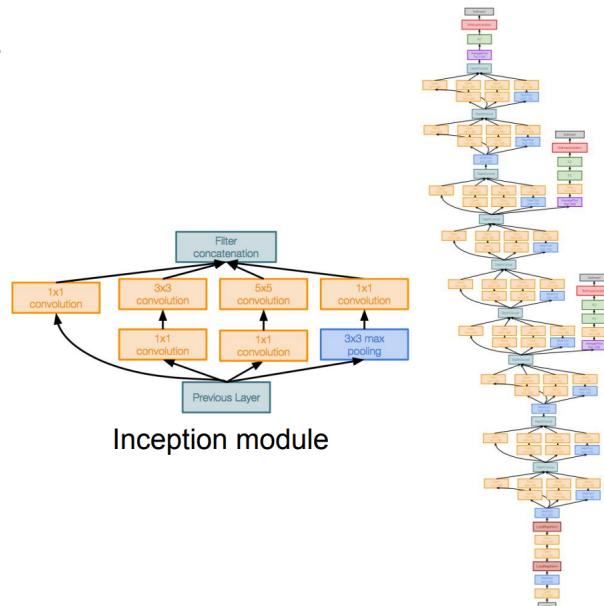


## Case Study: GoogLeNet

[Szegedy et al., 2014]

Deeper networks, with computational efficiency

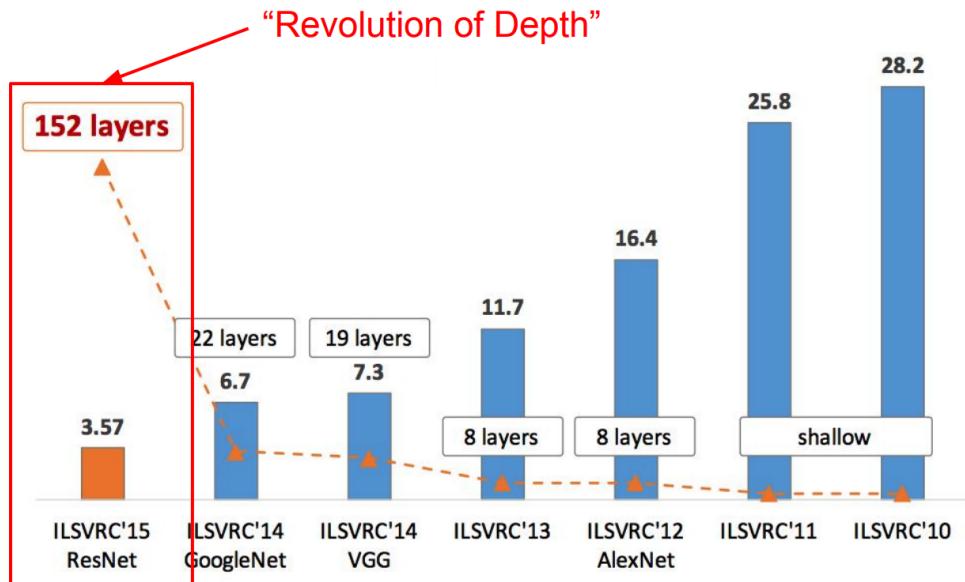
- 22 layers
- Efficient “Inception” module
- No FC layers
- 12x less params than AlexNet
- ILSVRC’14 classification winner (6.7% top 5 error)



## ResNet

2015년에는, 급격하게 많은 layer을 사용하여 ResNet이 Image Net Challenge에서 우승을 하였다.

# ImageNet Large Scale Visual Recognition Challenge (ILSVRC)



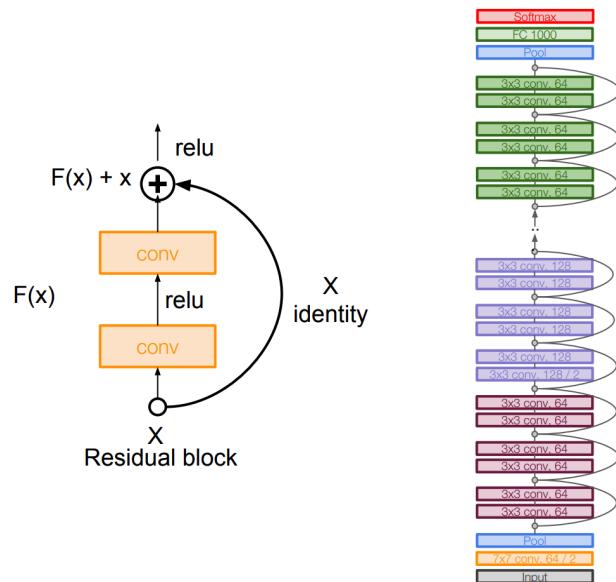
이들은 152개의 layer을 사용하여 3.57%의 오차율을 기록하였다.

# Case Study: ResNet

[He et al., 2015]

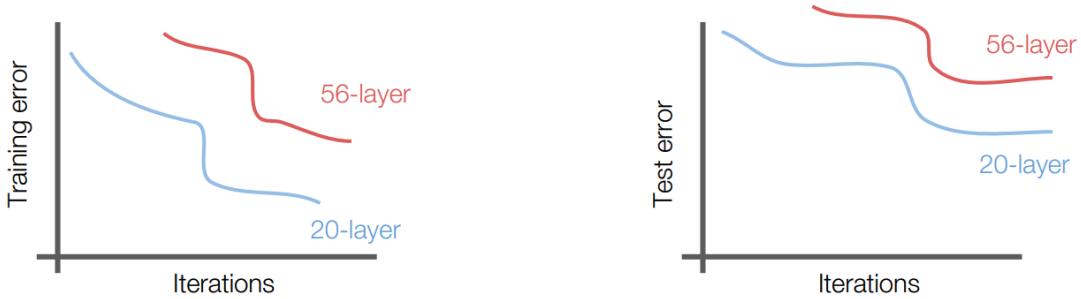
## Very deep networks using residual connections

- 152-layer model for ImageNet
  - ILSVRC'15 classification winner (3.57% top 5 error)
  - Swept all classification and detection competitions in ILSVRC'15 and COCO'15!



Q. layer를 많이 사용하는 것이 오차를 줄여줄까?

A. 이는 아니다. 실제로 많은 layer을 사용하는 것이 큰 오차를 낸다고 한다. 그런데, 그냥 layer가 많이 쌓여서 그렇다고 할 수는 없다. 아래의 그림을 보면, test layer에 오차가 있더라도, training layer에는 별 문제가 없어야 한다. 그저 overfit이 더 커져야 하는데, training layer에서도 오차가 커지는 현상을 발견할 수 있다.



사람들은, layer가 깊어질수록 최적화 문제를 해결하는 것이 더 힘들어진다고 하기 때문이라고 한다.

그러면 앞의 layer에 대한 정보를 깊은 layer에 조금 넣어주면, 이를 해결할 수 있지 않을까 해서 ResNet에서는 이를 다음과 같이 해결하였다.

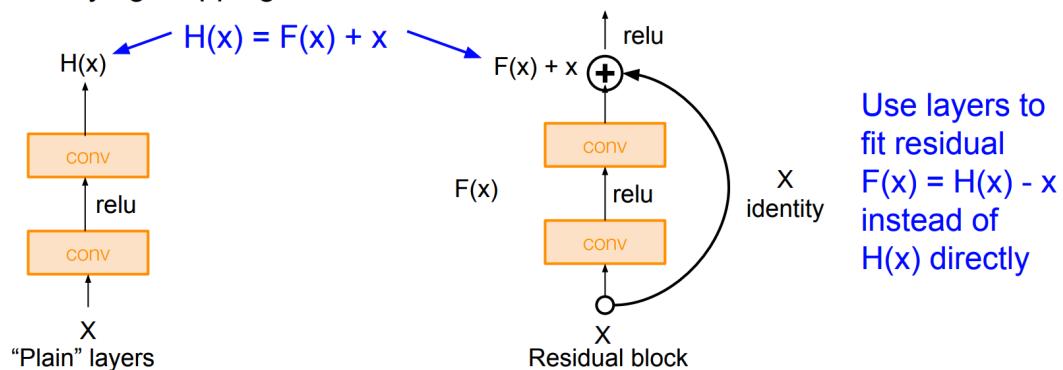
연산을 한 후에, 이전의 block을 들고와서 더해주는 것이다.

이는 해결 방법이 아닌, 그저 가설을 해결할 수 있는 방법이라고 생각하면 된다.

## Case Study: ResNet

[He et al., 2015]

Solution: Use network layers to fit a residual mapping instead of directly trying to fit a desired underlying mapping



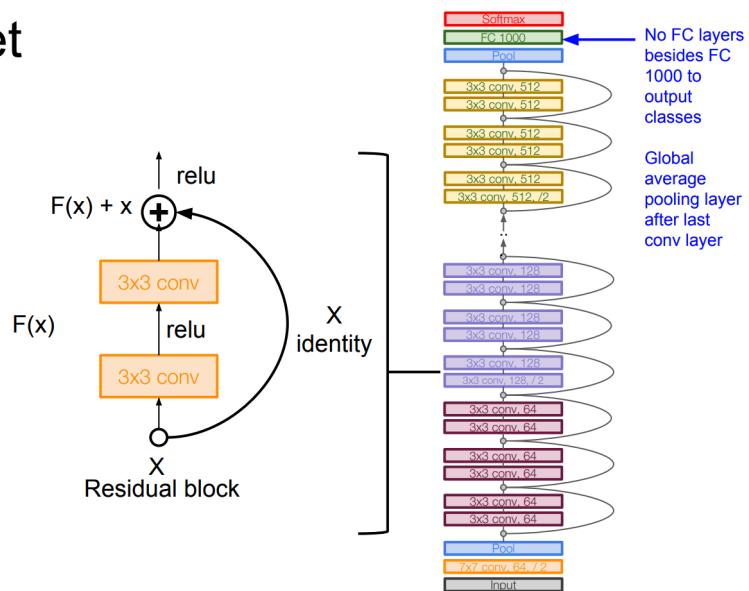
아래는 ResNet의 전체적인 형태이다. 앞에서 설명한 것처럼 Residual block을 이용하여 여러 layer를 사용했을 때의 문제를 해결하고자 했다. 또, 중간에 filter의 개수를 2배로 늘리고, stride=2를 사용하기도 한다. 시작 부분에는 CONV layer를 사용하고, 마지막에 FC1000 이외의 FC layer은 존재하지 않는다. 최종 깊이는 34, 50, 101, 152인 경우를 사용한다고 한다.

# Case Study: ResNet

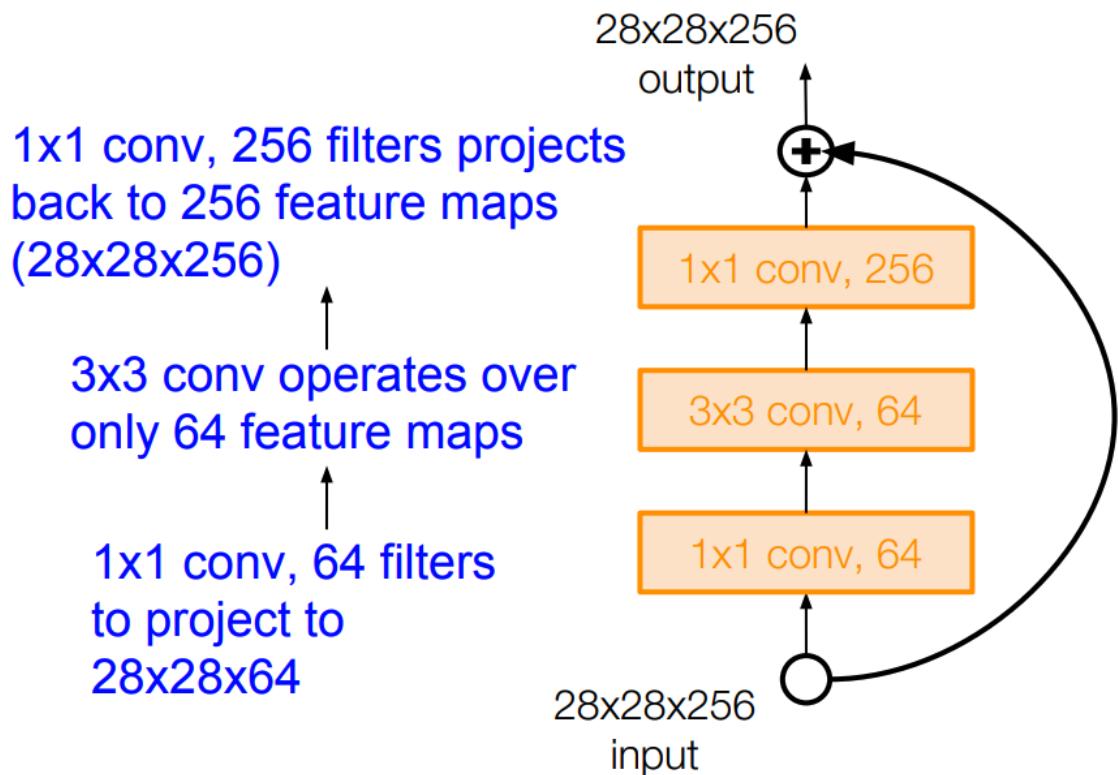
[He et al., 2015]

Full ResNet architecture:

- Stack residual blocks
- Every residual block has two 3x3 conv layers
- Periodically, double # of filters and downsample spatially using stride 2 (/2 in each dimension)
- Additional conv layer at the beginning
- No FC layers at the end (only FC 1000 to output classes)



여기서, ResNet-50+(101, 52)의 경우에는 GoogleNet처럼 bottleneck을 사용하여 효율을 증가시킨다.



ResNet의 특징들을 정리하자면, 아래와 같다.

# Case Study: ResNet

[He et al., 2015]

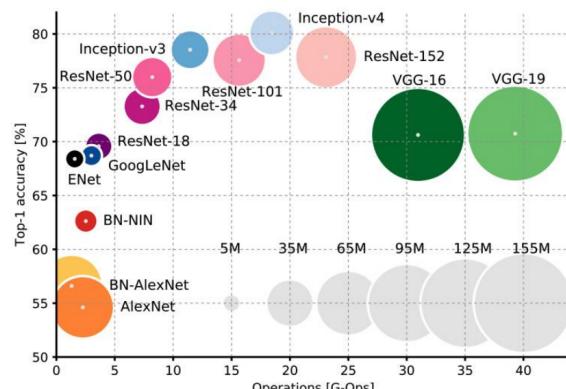
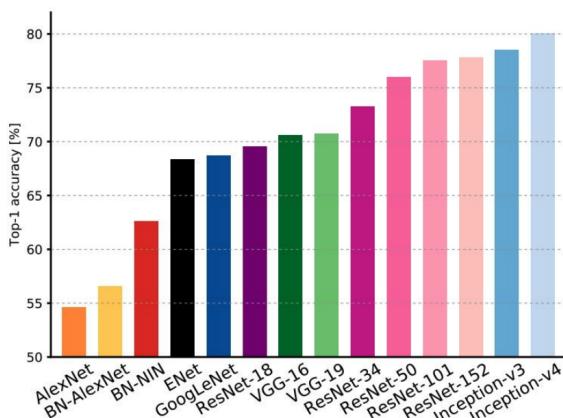
Training ResNet in practice:

- Batch Normalization after every CONV layer
- Xavier/2 initialization from He et al.
- SGD + Momentum (0.9)
- Learning rate: 0.1, divided by 10 when validation error plateaus
- Mini-batch size 256
- Weight decay of 1e-5
- No dropout used

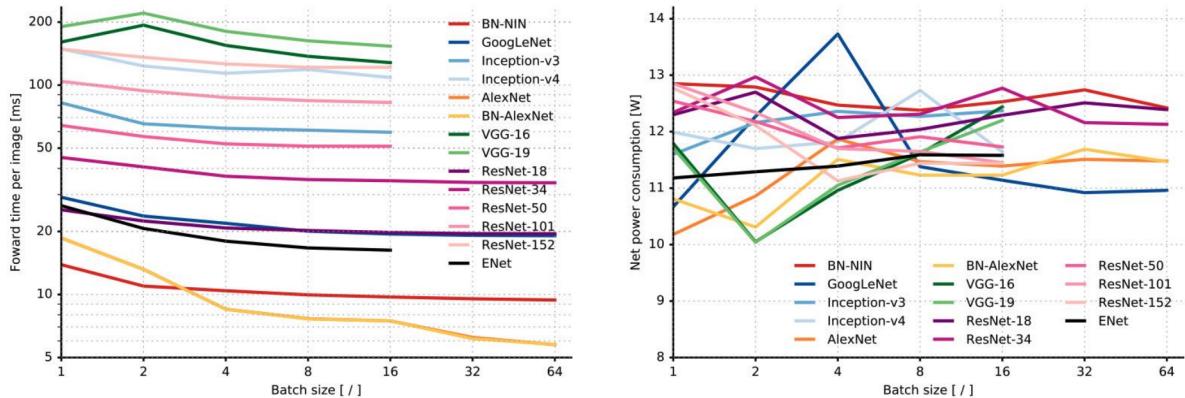
위에서 유명한 Network들에 대해 알아보았다. (AlexNet, VGG, GoogleNet, ResNet)

아래의 표는 각각의 Network들을 비교한 도표이다.

살펴보면, VGG는 연산량이 상당히 크지만 정확도는 꽤 괜찮다. GoogleNet은 이보다 훨씬 더 연산량이 적지만 정확도는 이와 비슷하다. 또 AlexNet은 너무나 간단한 연산이고 layer 개수도 적기에 연산량도 작고, 정확도도 많이 떨어진다. 마지막에 본 ResNet은 연산량도 엄청 많지 않고 적당하며, 정확도 또한 상당히 우수하다는 것을 알 수 있다.



아래는 Forward 연산 시간을 보여주는 도표이다. 역시나, VGG가 다른 것들에 비해 상당히 큰 시간을 소요한다는 것을 알 수 있다.



An Analysis of Deep Neural Network Models for Practical Applications, 2017.

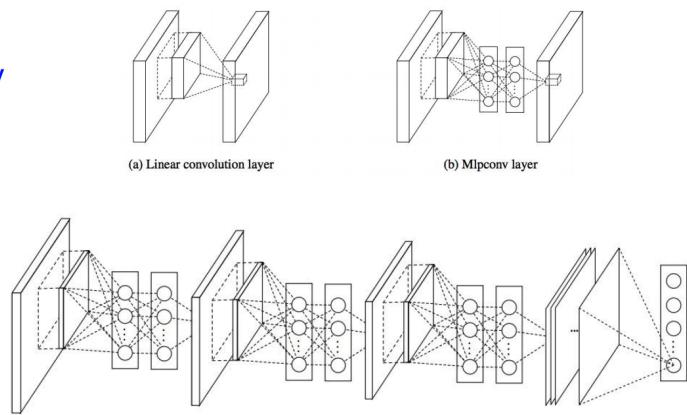
위에서 설명한 Network들 이외에도 여러 네트워크들이 존재한다. 이번에는 다른 것들에 대해 알아보자.

## Network in Network(NiN)

### Network in Network (NiN)

[Lin et al. 2014]

- Mlpconv layer with “micronetwork” within each conv layer to compute more abstract features for local patches
- Micronetwork uses multilayer perceptron (FC, i.e. 1x1 conv layers)
- Precursor to GoogLeNet and ResNet “bottleneck” layers
- Philosophical inspiration for GoogLeNet

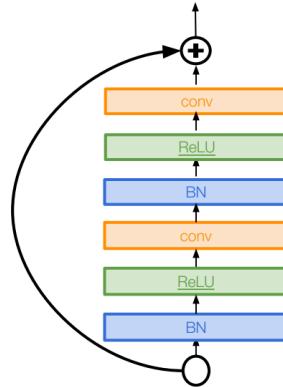


## Improve ResNet

# Identity Mappings in Deep Residual Networks

[He et al. 2016]

- Improved ResNet block design from creators of ResNet
- Creates a more direct path for propagating information throughout network (moves activation to residual mapping pathway)
- Gives better performance

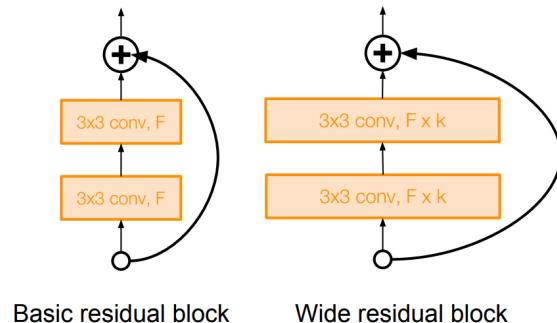


Wide Residual Network는 필터의 크기를 늘리는 것이다. 이렇게 하면 152 layer의 효과를 50-layer로 낼 수 있다고 한다. 또, 병렬 연산이 가능해서 속도면에서도 효율적이라고 한다.

## Wide Residual Networks

[Zagoruyko et al. 2016]

- Argues that residuals are the important factor, not depth
- Use wider residual blocks ( $F \times k$  filters instead of  $F$  filters in each layer)
- 50-layer wide ResNet outperforms 152-layer original ResNet
- Increasing width instead of depth more computationally efficient (parallelizable)

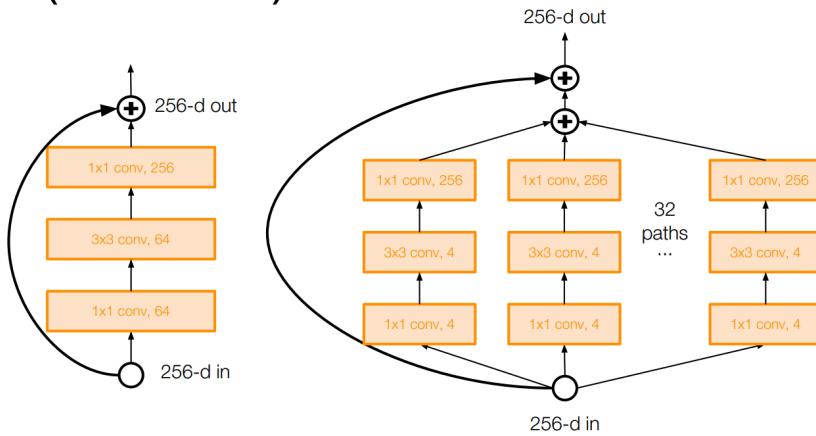


ResNeXt는 아래의 그림처럼 여러 path들을 병렬적으로 연산하는 것이다.

# Aggregated Residual Transformations for Deep Neural Networks (ResNeXt)

[Xie et al. 2016]

- Also from creators of ResNet
- Increases width of residual block through multiple parallel pathways (“cardinality”)
- Parallel pathways similar in spirit to Inception module

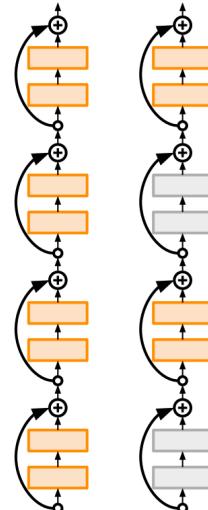


위에서, depth가 깊어질수록 최적화 문제가 발생한다고 가설을 세웠고, 이를 해결하기 위해서 만들어진 것이 residual block이다. 이를 보다 더 최적화하기 위해서 dropout을 이용하여 몇몇의 layer를 dropout시키는 방법을 제안한 것이 아래의 그림이다.

## Deep Networks with Stochastic Depth

[Huang et al. 2016]

- Motivation: reduce vanishing gradients and training time through short networks during training
- Randomly drop a subset of layers during each training pass
- Bypass with identity function
- Use full deep network at test time

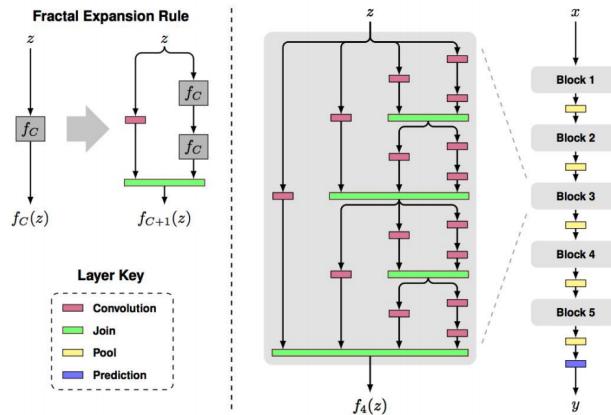


ResNet이 최적이 아닐거라는 생각에서, Residual block을 사용하지 않은, 새로운 방법이 바로 FractalNet이다. 이는 아래의 그림처럼 짧은 depth와 깊은 depth가 병렬적으로 이루어지고 이들을 한번에 연산하여 해결하는 방법이다.

# FractalNet: Ultra-Deep Neural Networks without Residuals

[Larsson et al. 2017]

- Argues that key is transitioning effectively from shallow to deep and residual representations are not necessary
- Fractal architecture with both shallow and deep paths to output
- Trained with dropping out sub-paths
- Full network at test time



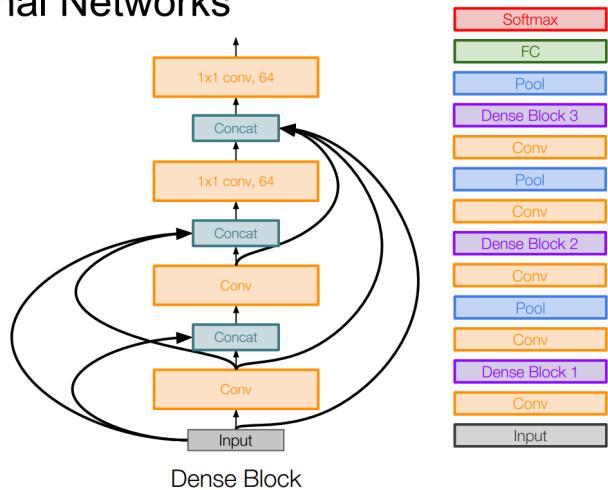
이 또한 ResNet이외의 방법을 제안한 것으로, 아래의 그림처럼 이전의 block이 앞의 여러 layer에 영향을 주는 형태로 만들어진다.

## Beyond ResNets...

### Densely Connected Convolutional Networks

[Huang et al. 2017]

- Dense blocks where each layer is connected to every other layer in feedforward fashion
- Alleviates vanishing gradient, strengthens feature propagation, encourages feature reuse



이는 상당히 효율적인 형태의 네트워크인 SqueezeNet이다. 이전의 네트워크들은 모두 정확성을 다루며 만들어졌다면, 이는 효율성 측면에서 접근하여 어떻게 하면 효율적인 네트워크를 만들 수 있는지에 대한 것이다.

## Efficient networks...

### SqueezeNet: AlexNet-level Accuracy With 50x Fewer Parameters and <0.5Mb Model Size

[Iandola et al. 2017]

- Fire modules consisting of a ‘squeeze’ layer with 1x1 filters feeding an ‘expand’ layer with 1x1 and 3x3 filters
- AlexNet level accuracy on ImageNet with 50x fewer parameters
- Can compress to 510x smaller than AlexNet (0.5Mb)

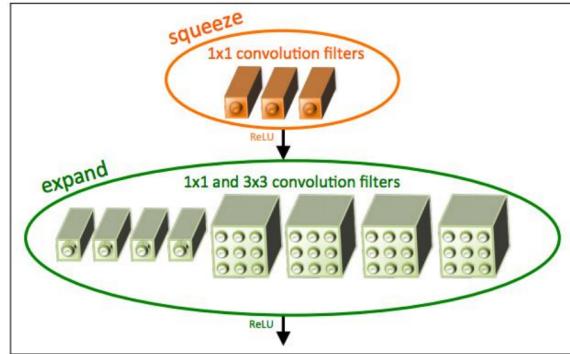


Figure copyright Iandola, Han, Moskewicz, Ashraf, Dally, Keutzer, 2017. Reproduced with permission.