
안녕하세요!

Lecture 9. CNN Architectures

김영은

Review

Last time: Deep learning frameworks

Caffe
(UC Berkeley)



Caffe2
(Facebook)

Torch
(NYU / Facebook)



PyTorch
(Facebook)

Theano
(U Montreal)



TensorFlow
(Google)

Paddle
(Baidu)

CNTK
(Microsoft)

MXNet
(Amazon)
Developed by U Washington, CMU, MIT,
Hong Kong U, etc but main framework of
choice at AWS

And others...

- (1) Easily build big computational graphs
- (2) Easily compute gradients in computational graphs
- (3) Run it all efficiently on GPU (wrap cuDNN, cuBLAS, etc)

Review

```
class ReLU(torch.autograd.Function):  
    def forward(self, x):  
        self.save_for_backward(x)  
        return x.clamp(min=0)  
  
    def backward(self, grad_y):  
        x, = self.saved_tensors  
        grad_input = grad_y.clone()  
        grad_input[x < 0] = 0  
        return grad_input
```

```
import torch  
from torch.autograd import Variable  
  
N, D_in, H, D_out = 64, 1000, 100, 10  
x = Variable(torch.randn(N, D_in))  
y = Variable(torch.randn(N, D_out), requires_grad=False)  
  
model = torch.nn.Sequential(  
    torch.nn.Linear(D_in, H),  
    torch.nn.ReLU(),  
    torch.nn.Linear(H, D_out))  
loss_fn = torch.nn.MSELoss(size_average=False)  
  
learning_rate = 1e-4  
for t in range(500):  
    y_pred = model(x)  
    loss = loss_fn(y_pred, y)  
  
    model.zero_grad()  
    loss.backward()  
  
    for param in model.parameters():  
        param.data -= learning_rate * param.grad.data
```

INDEX



AlexNet



VGGnet



GoogleNet



ResNet

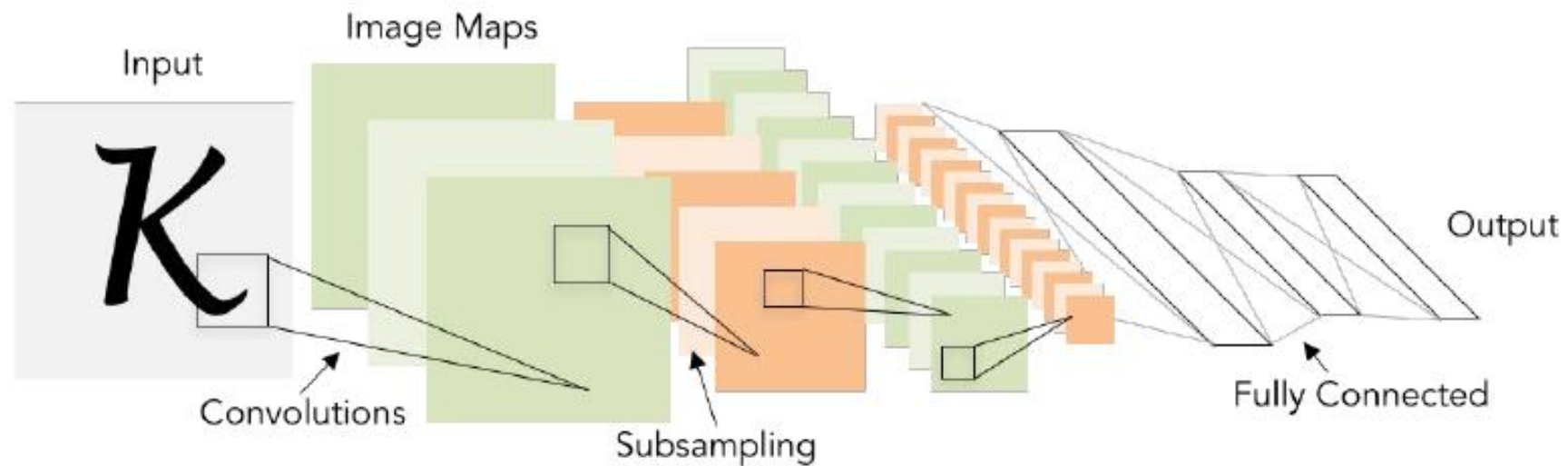


Others



Review: LeNet-5

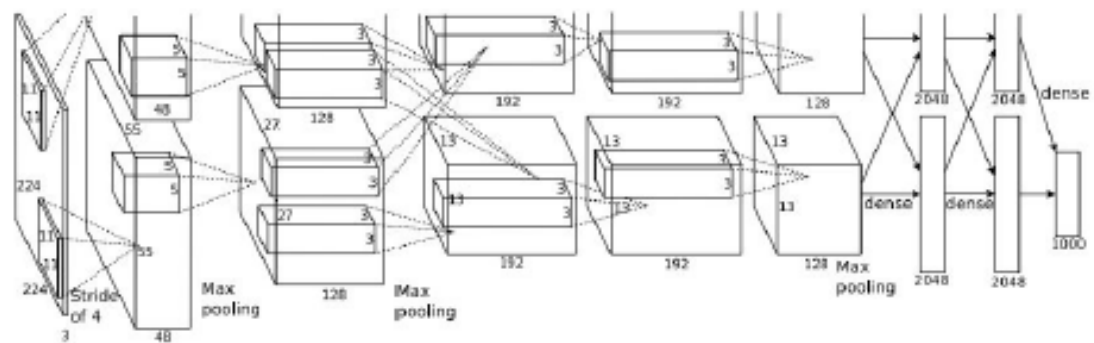
[LeCun et al., 1998]



Conv filters were 5x5, applied at stride 1
Subsampling (Pooling) layers were 2x2 applied at stride 2
i.e. architecture is [CONV-POOL-CONV-POOL-FC-FC]

Case Study: AlexNet

[Krizhevsky et al. 2012]



Input: 227x227x3 images

First layer (CONV1): 96 11x11 filters applied at stride 4

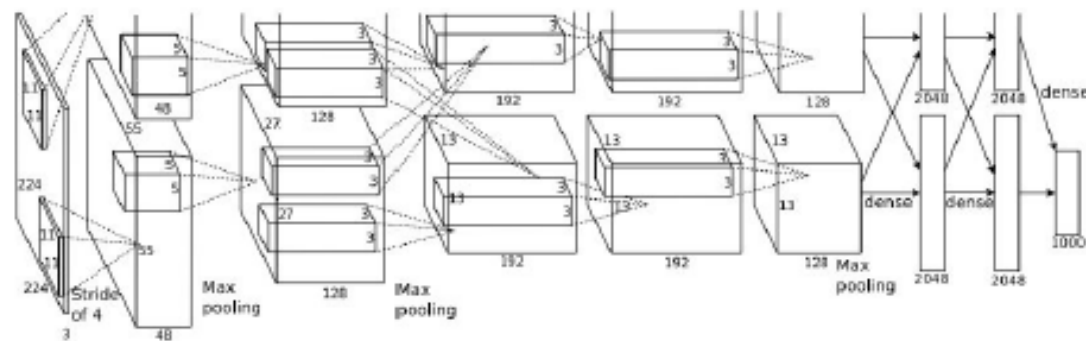
=>

Output volume **[55x55x96]** Hint: $(227-11)/4+1 = 55$

Parameters: $(11*11*3)*96 = 35K$

Case Study: AlexNet

[Krizhevsky et al. 2012]



Input: 227x227x3 images

After CONV1: 55x55x96

Second layer (POOL1): 3x3 filters applied at stride 2

Output volume: 27x27x96 Hint: $(55-3)/2+1 = 27$

Parameters: 0!

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

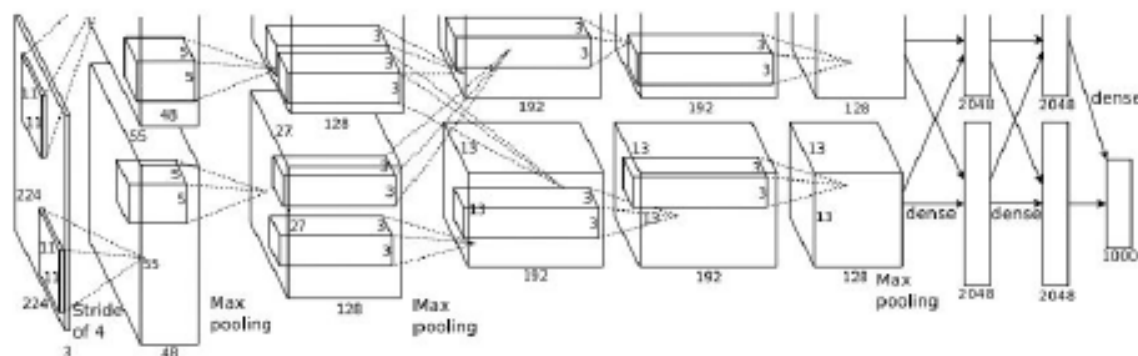
[13x13x256] **CONV5**: 256 3x3 filters at stride 1, pad 1

[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.

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

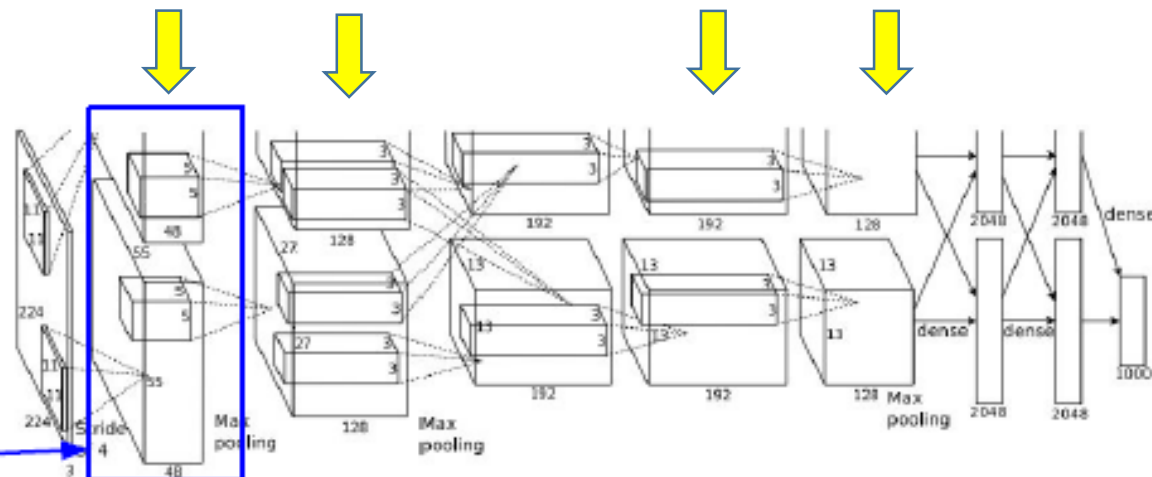
[13x13x256] **CONV5**: 256 3x3 filters at stride 1, pad 1

[6x6x256] **MAX POOL3**: 3x3 filters at stride 2

[4096] **FC6**: 4096 neurons

[4096] **FC7**: 4096 neurons

[1000] **FC8**: 1000 neurons (class scores)



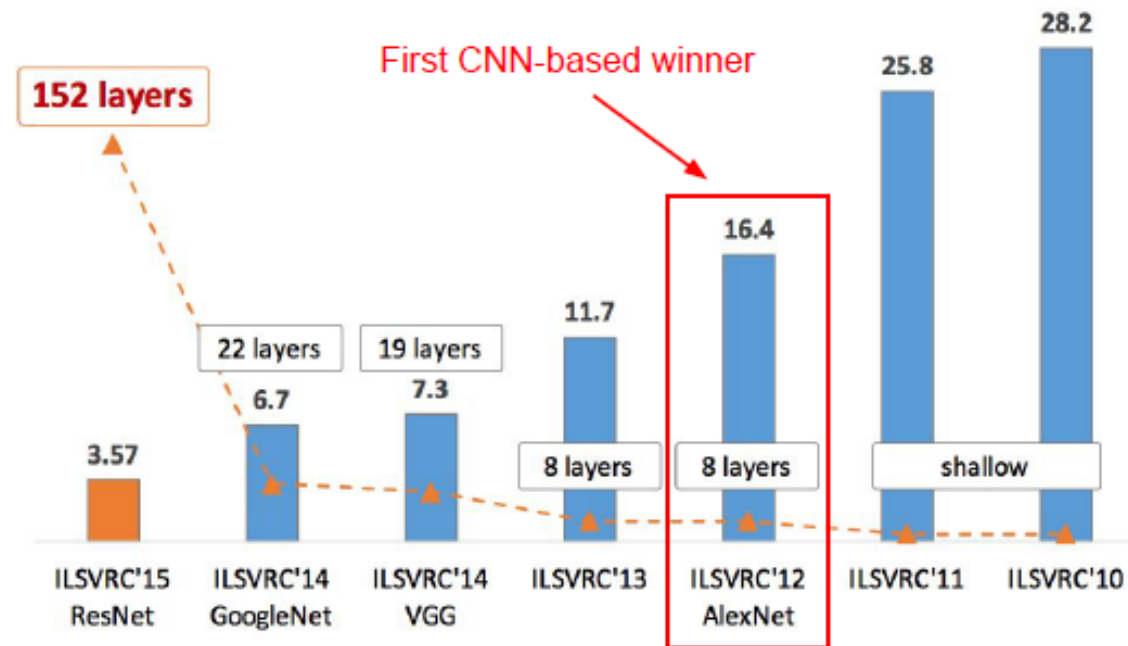
[55x55x48] x 2

Historical note: Trained on GTX 580 GPU with only 3 GB of memory. Network spread across 2 GPUs, half the neurons (feature maps) on each GPU.

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



ImageNet Large Scale Visual Recognition Challenge (ILSVRC) winners

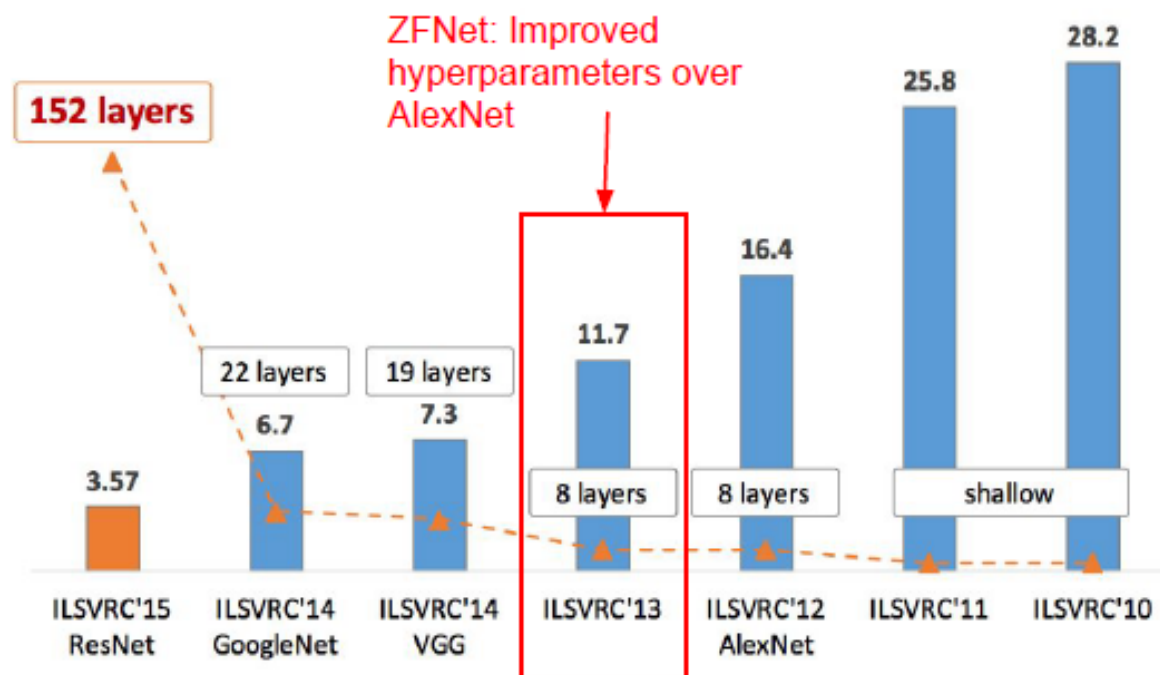


- ✓ cut the error rate!
- ✓ first CNN based winner!

Figure copyright Kaiming He, 2016. Reproduced with permission.



ImageNet Large Scale Visual Recognition Challenge (ILSVRC) winners

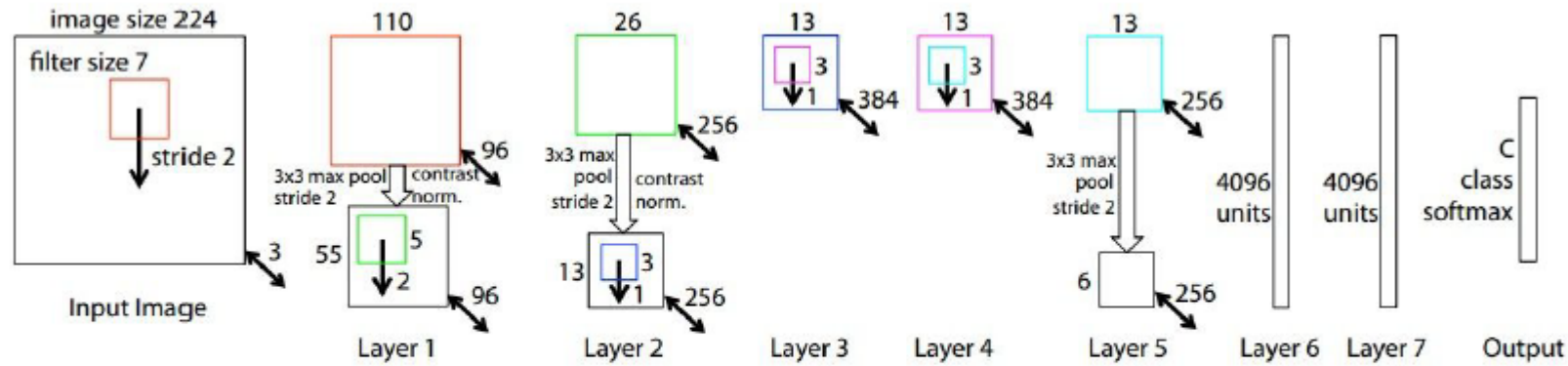


✓ Change stride size and filter's number!

Figure copyright Kai Ming He, 2016. Reproduced with permission.

ZFNet

[Zeiler and Fergus, 2013]



TODO: remake figure

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%

ImageNet Large Scale Visual Recognition Challenge (ILSVRC) winners

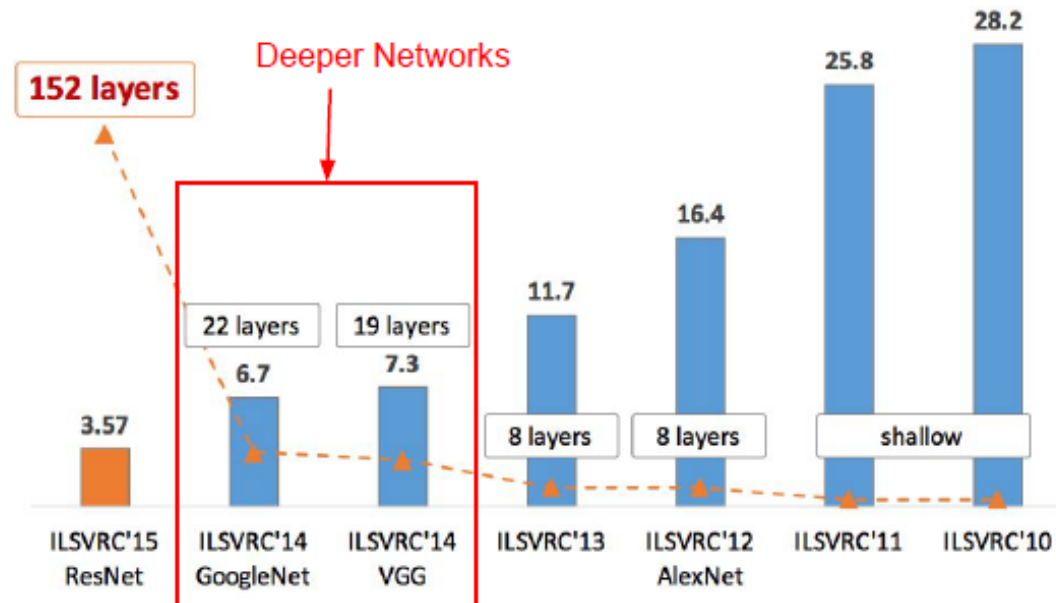


Figure copyright Kai Ming He, 2016. Reproduced with permission.

“Much deeper network, Much smaller filters”

Case Study: VGGNet

[Simonyan and Zisserman, 2014]

Small filters, Deeper networks

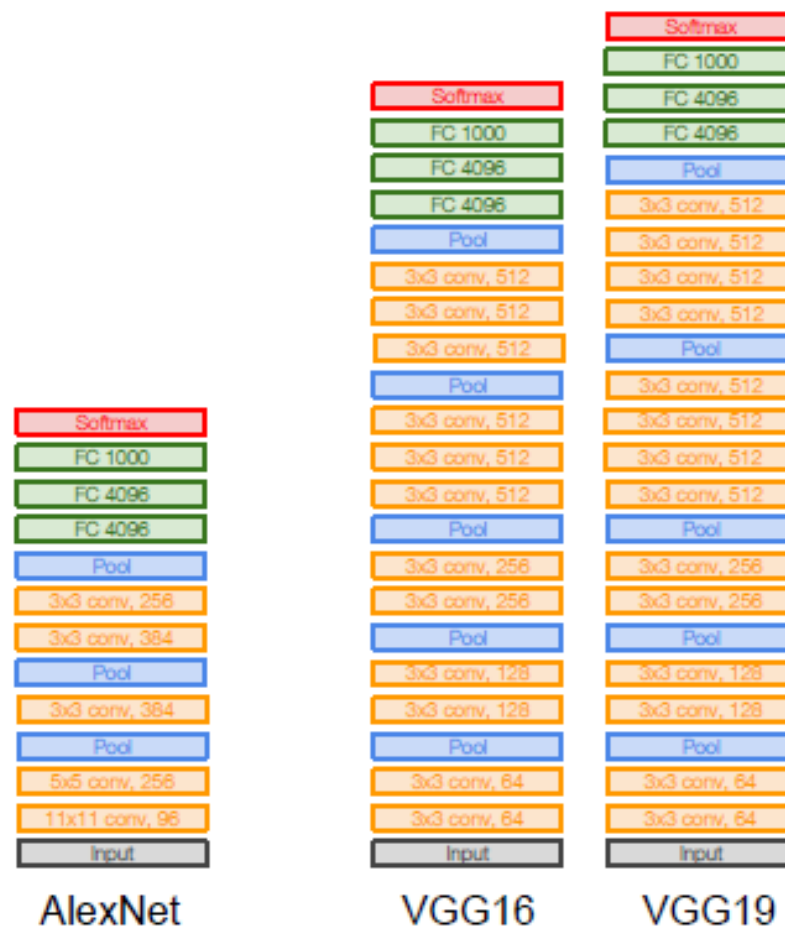
8 layers (AlexNet)

-> 16 - 19 layers (VGG16Net)

Only 3x3 CONV stride 1, pad 1
and 2x2 MAX POOL stride 2

11.7% top 5 error in ILSVRC'13
(ZFNet)

-> 7.3% top 5 error in ILSVRC'14



Case Study: VGGNet

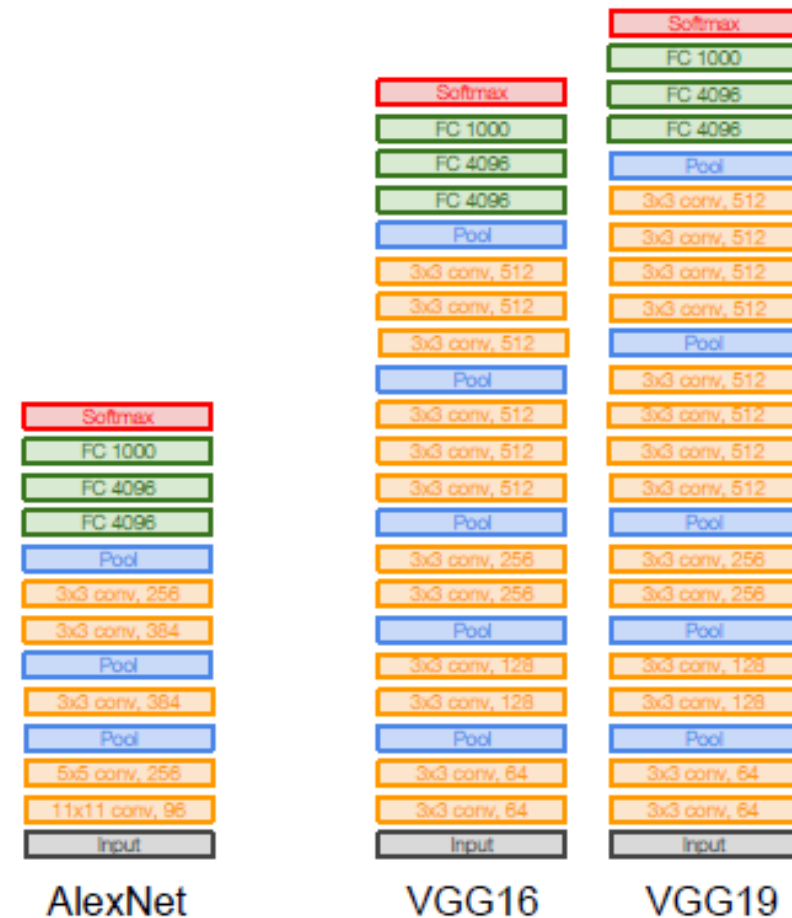
[Simonyan and Zisserman, 2014]

Q: Why use smaller filters? (3x3 conv)

Stack of three 3x3 conv (stride 1) layers has same **effective receptive field** as one 7x7 conv layer

But deeper, more non-linearities

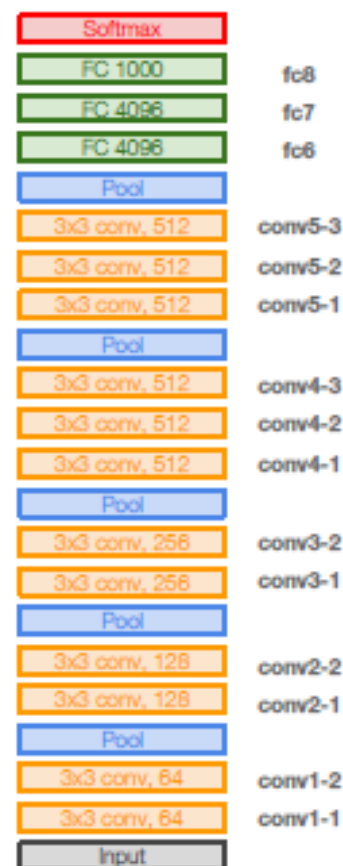
And fewer parameters: $3 * (3^2 C^2)$ vs. $7^2 C^2$ for C channels per layer



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

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

TOTAL params: 138M parameters



VGG16

Common names

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

Note:

Most memory is in early CONV

Most params are in late FC

TOTAL memory: $24\text{M} * 4 \text{ bytes} \approx 96\text{MB}$ / image (only forward! $\sim *2$ for bwd)

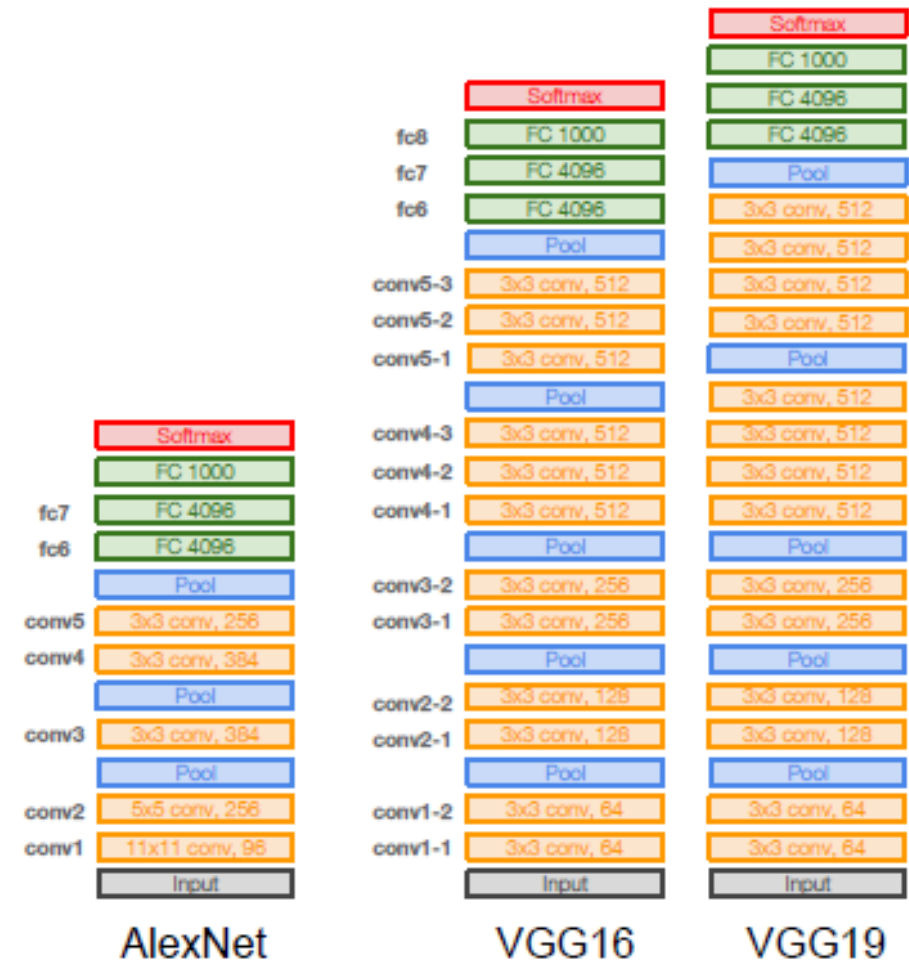
TOTAL params: 138M parameters

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

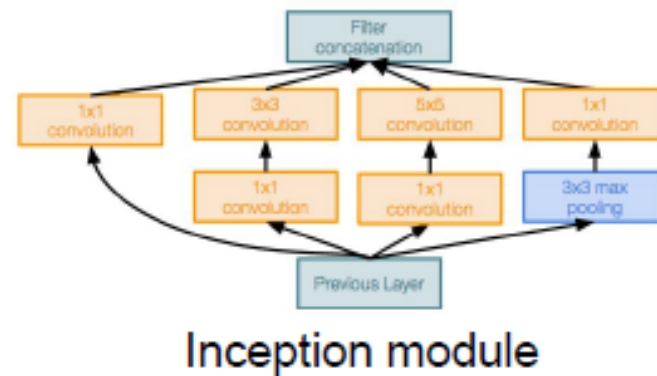


Case Study: GoogLeNet

[Szegedy et al., 2014]

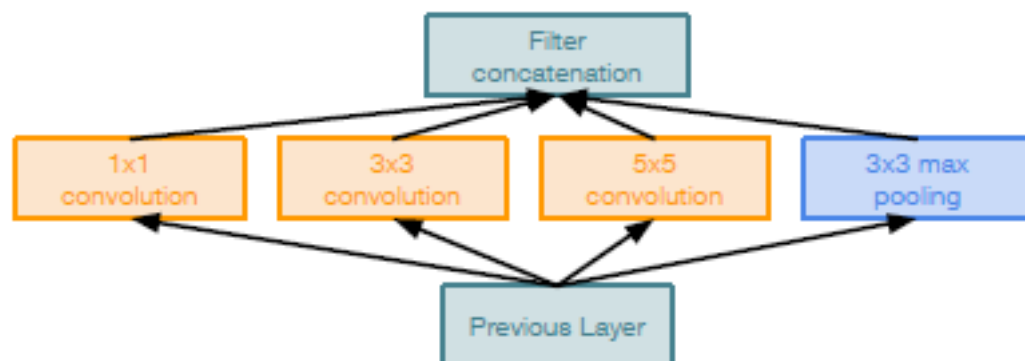
Deeper networks, with computational efficiency

- 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)



Case Study: GoogLeNet

[Szegedy et al., 2014]



Naive Inception module

Apply parallel filter operations on the input from previous layer:

- Multiple receptive field sizes for convolution (1x1, 3x3, 5x5)
- Pooling operation (3x3)

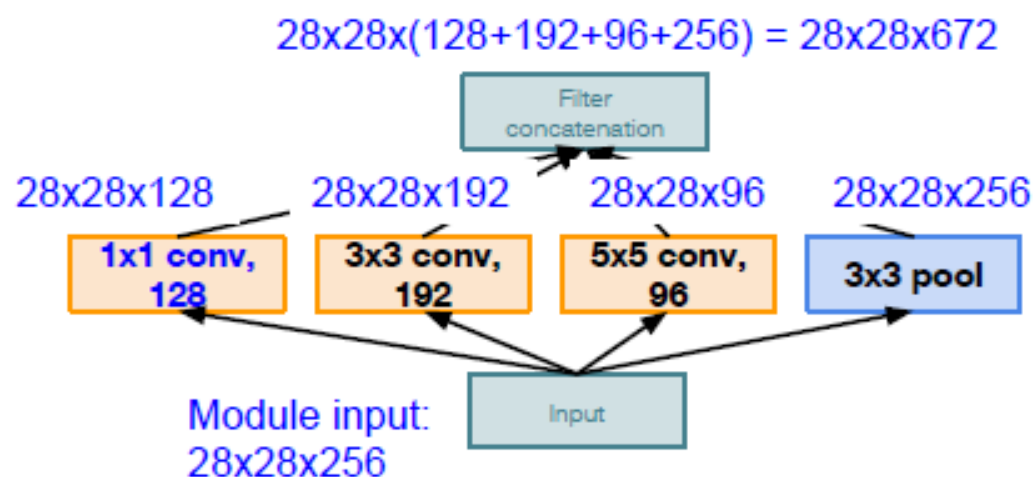
Concatenate all filter outputs together depth-wise

Case Study: GoogLeNet

[Szegedy et al., 2014]

Example:

Q3: What is output size after filter concatenation?



Naive Inception module

Q: What is the problem with this?
[Hint: Computational complexity]

Conv Ops:

[1x1 conv, 128] $28 \times 28 \times 128 \times 1 \times 1 \times 256$

[3x3 conv, 192] $28 \times 28 \times 192 \times 3 \times 3 \times 256$

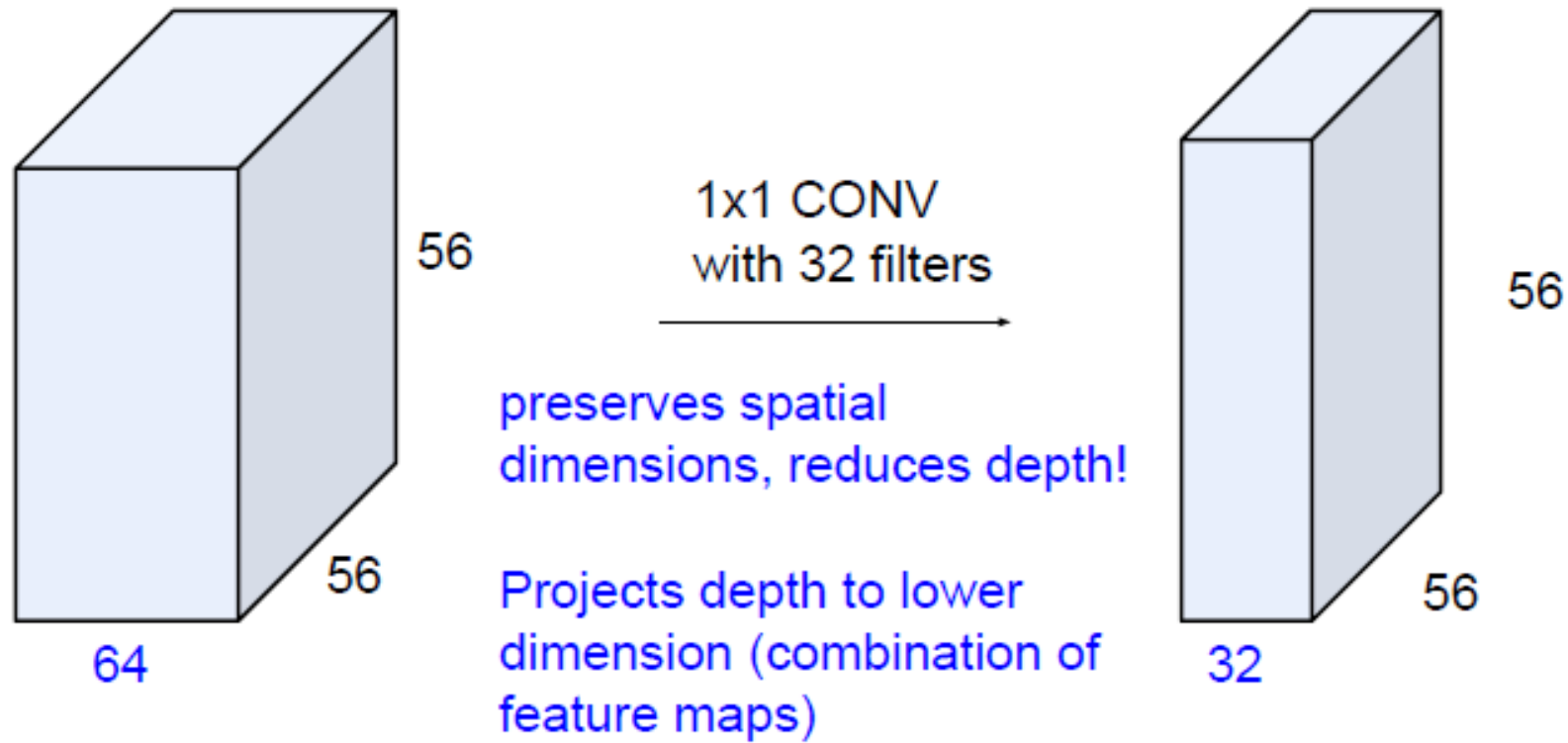
[5x5 conv, 96] $28 \times 28 \times 96 \times 5 \times 5 \times 256$

Total: 854M ops

Very expensive compute

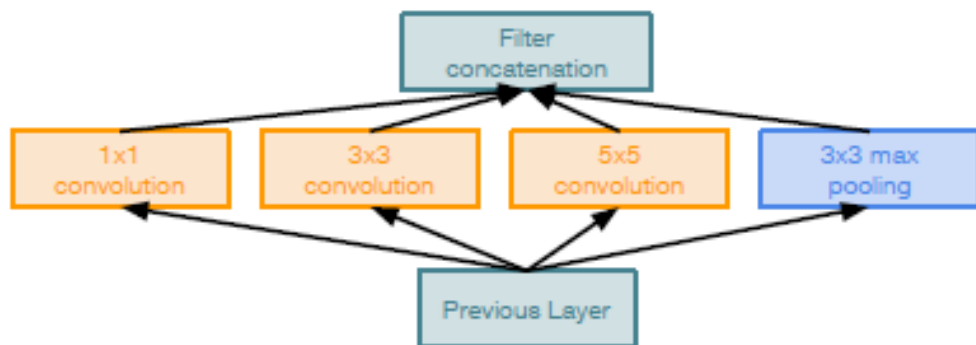
Pooling layer also preserves feature depth, which means total depth after concatenation can only grow at every layer!

Reminder: 1x1 convolutions



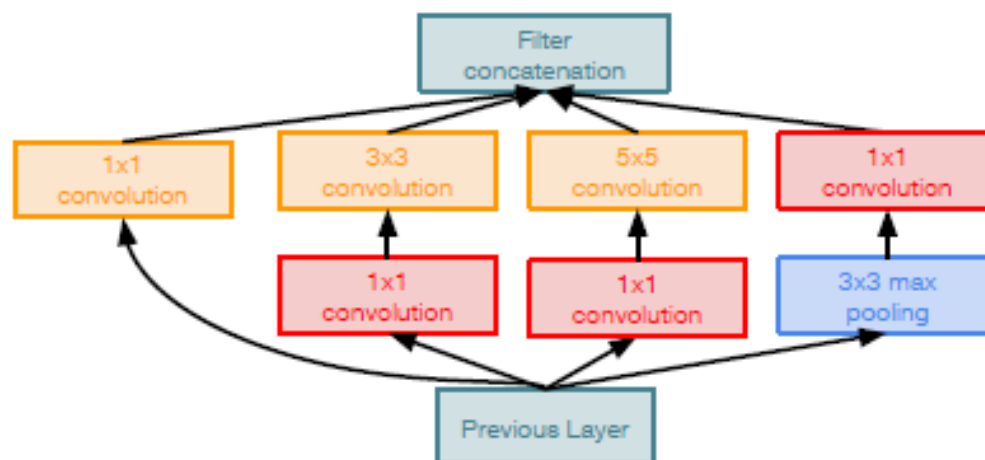
Case Study: GoogLeNet

[Szegedy et al., 2014]



Naive Inception module

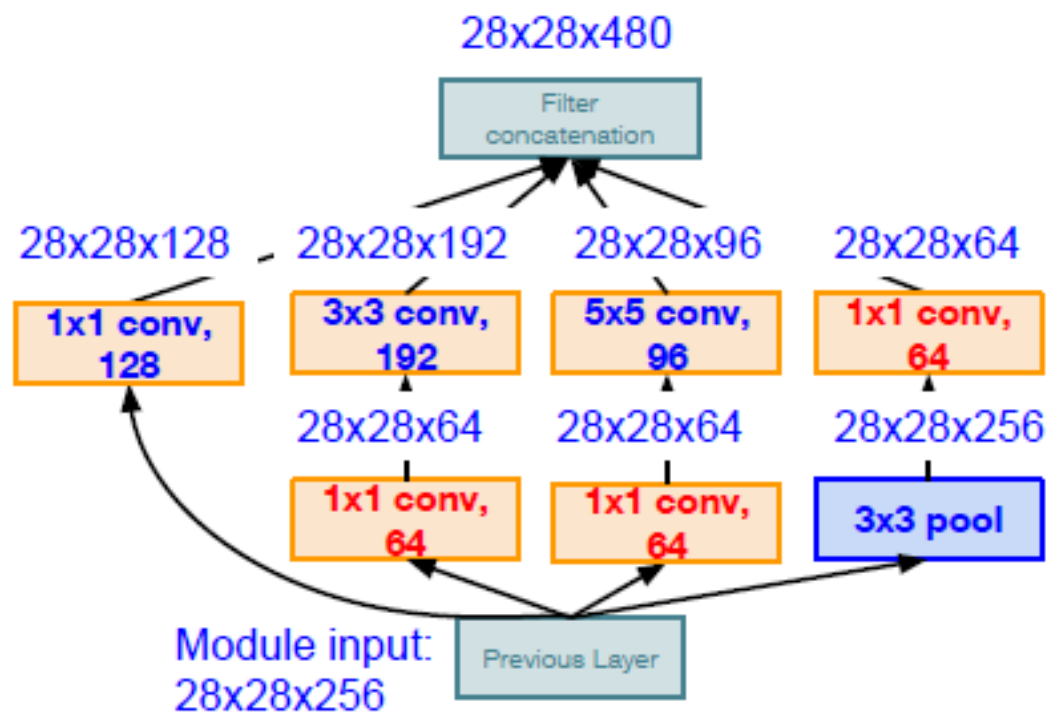
1x1 conv “bottleneck”
layers



Inception module with dimension reduction

Case Study: GoogLeNet

[Szegedy et al., 2014]



Inception module with dimension reduction

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

Conv Ops:

$[1 \times 1 \text{ conv}, 64] \quad 28 \times 28 \times 64 \times 1 \times 1 \times 256$
 $[1 \times 1 \text{ conv}, 64] \quad 28 \times 28 \times 64 \times 1 \times 1 \times 256$
 $[1 \times 1 \text{ conv}, 128] \quad 28 \times 28 \times 128 \times 1 \times 1 \times 256$
 $[3 \times 3 \text{ conv}, 192] \quad 28 \times 28 \times 192 \times 3 \times 3 \times 64$
 $[5 \times 5 \text{ conv}, 96] \quad 28 \times 28 \times 96 \times 5 \times 5 \times 64$
 $[1 \times 1 \text{ conv}, 64] \quad 28 \times 28 \times 64 \times 1 \times 1 \times 256$

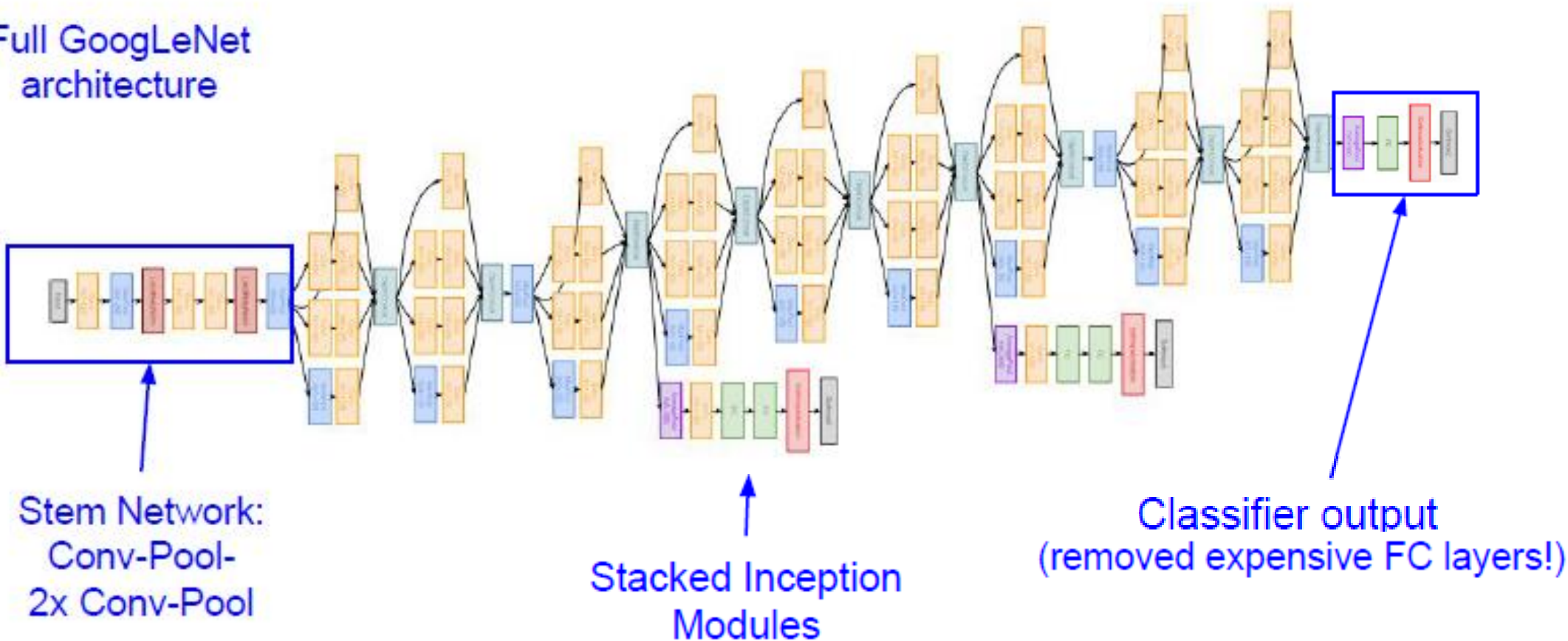
Total: 358M ops

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

Case Study: GoogLeNet = “Full Inception Architecture”

[Szegedy et al., 2014]

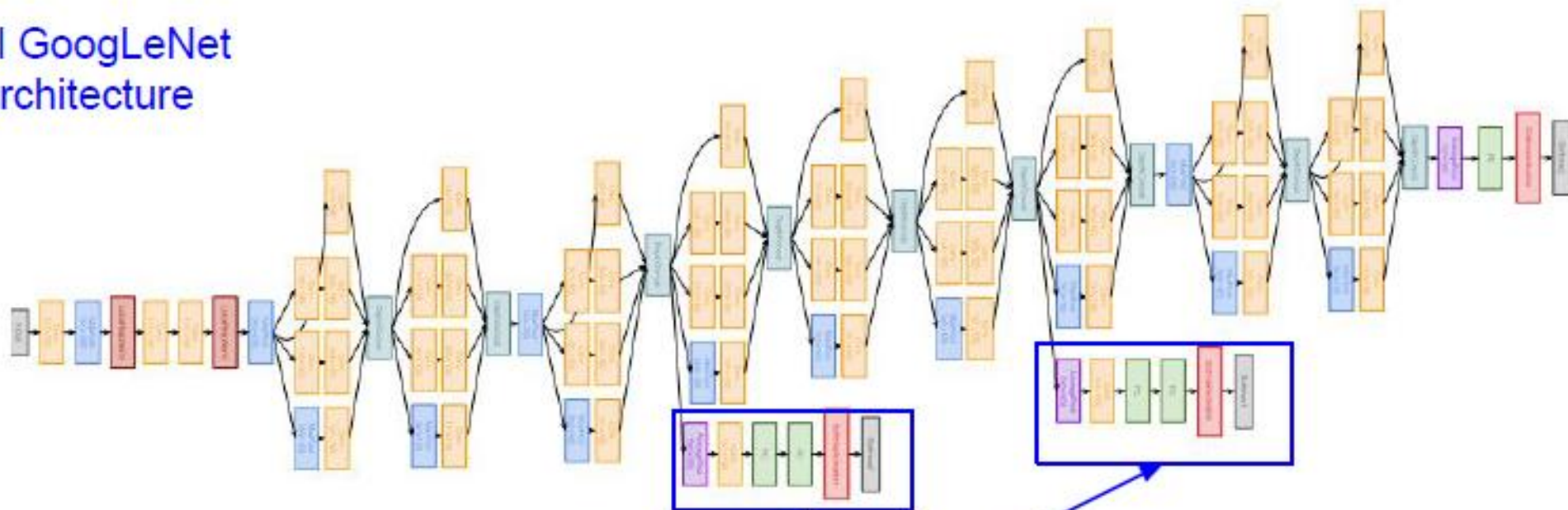
Full GoogLeNet
architecture



Case Study: GoogLeNet

[Szegedy et al., 2014]

Full GoogLeNet
architecture



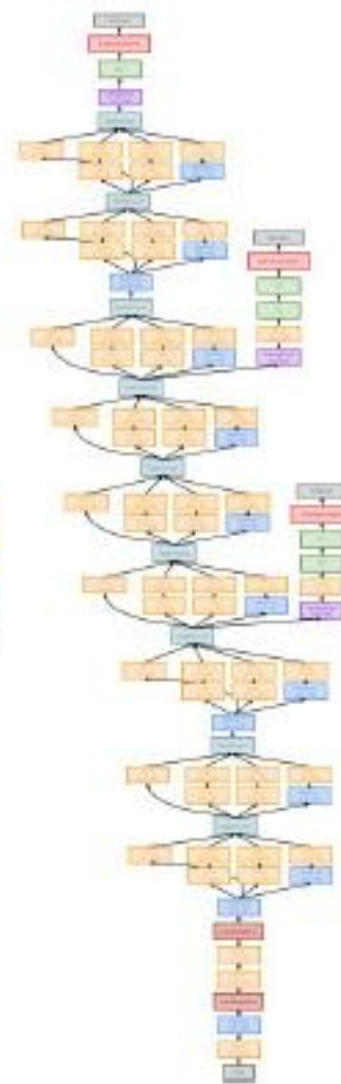
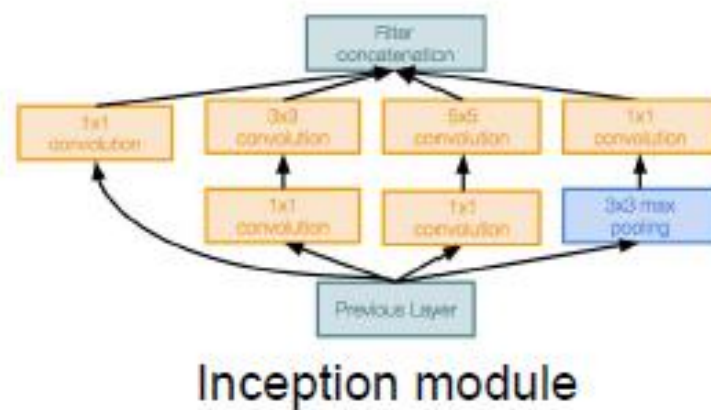
Auxiliary classification outputs to inject additional gradient at lower layers
(AvgPool-1x1Conv-FC-FC-Softmax)

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)



ImageNet Large Scale Visual Recognition Challenge (ILSVRC) winners

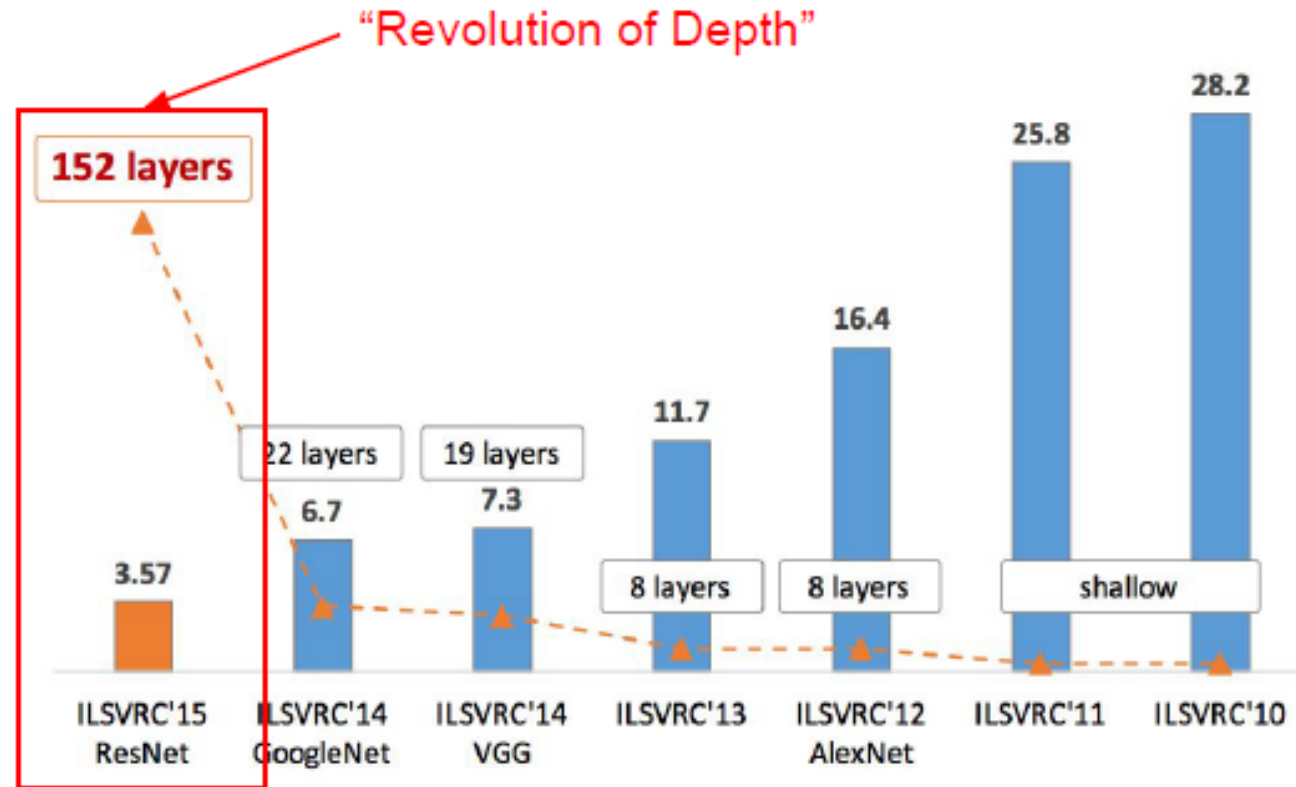


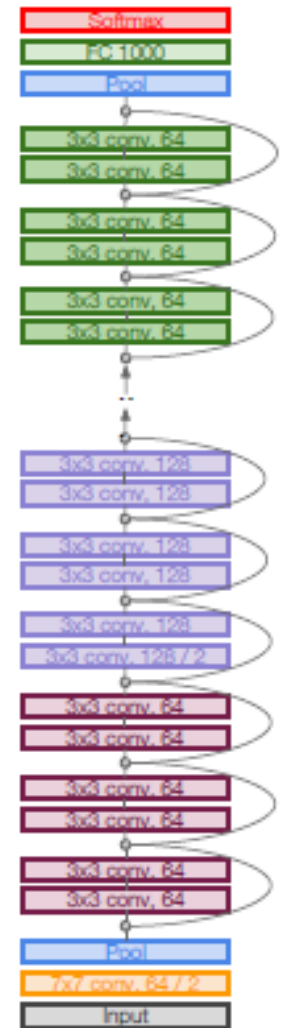
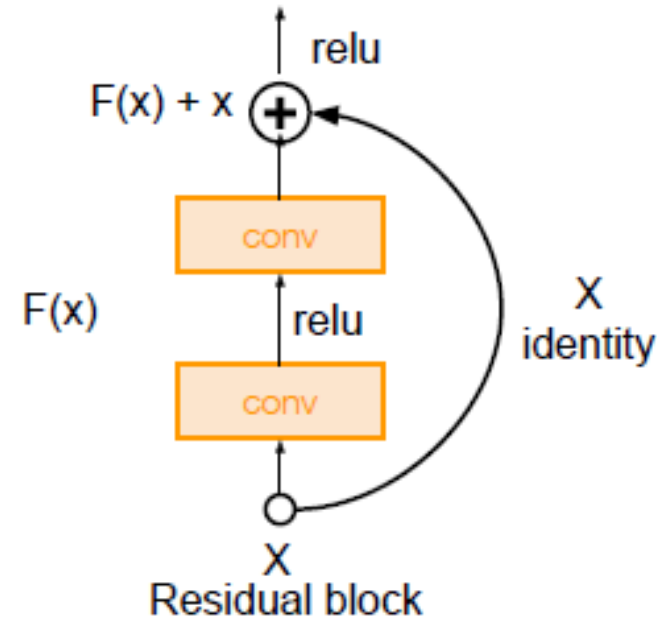
Figure copyright Kaiming He, 2016. Reproduced with permission.

Case Study: ResNet

[He et al., 2015]

Very deep networks using residual connections

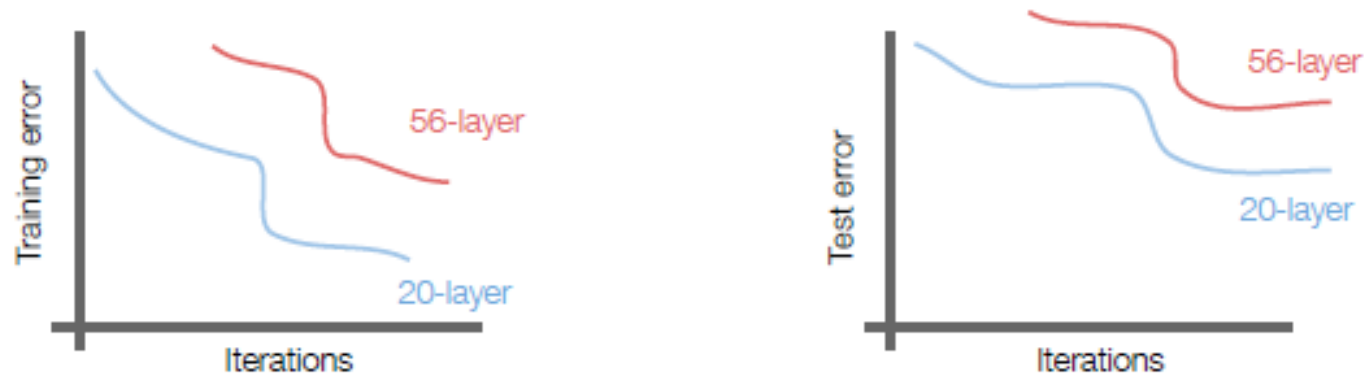
- 152-layer model for ImageNet
- ILSVRC'15 classification winner (3.57% top 5 error)
- Swept all classification and detection competitions in ILSVRC'15 and COCO'15!



Case Study: ResNet

[He et al., 2015]

What happens when we continue stacking deeper layers on a “plain” convolutional neural network?



56-layer model performs worse on both training and test error
-> The deeper model performs worse, but it's not caused by overfitting!

Case Study: ResNet

[He et al., 2015]

Hypothesis: the problem is an *optimization* problem, deeper models are harder to optimize

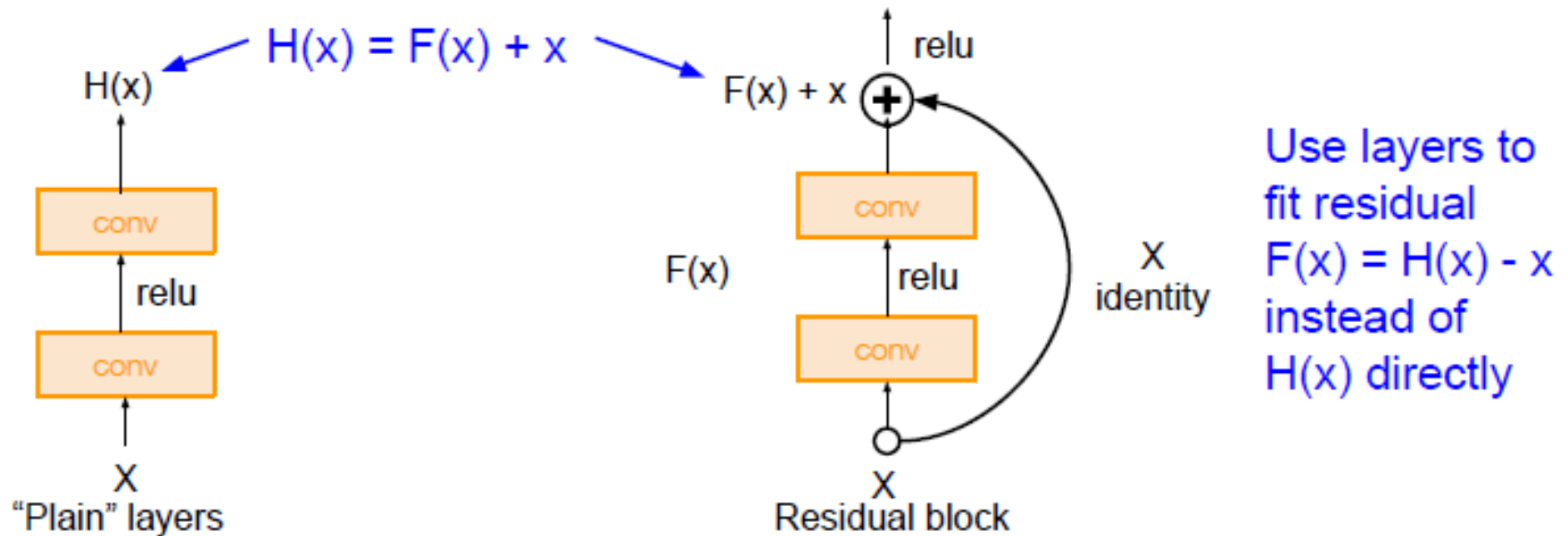
The deeper model should be able to perform at least as well as the shallower model.

A solution by construction is copying the learned layers from the shallower model and setting additional layers to identity mapping.

Case Study: ResNet = “Full Resident Architecture”

[He et al., 2015]

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

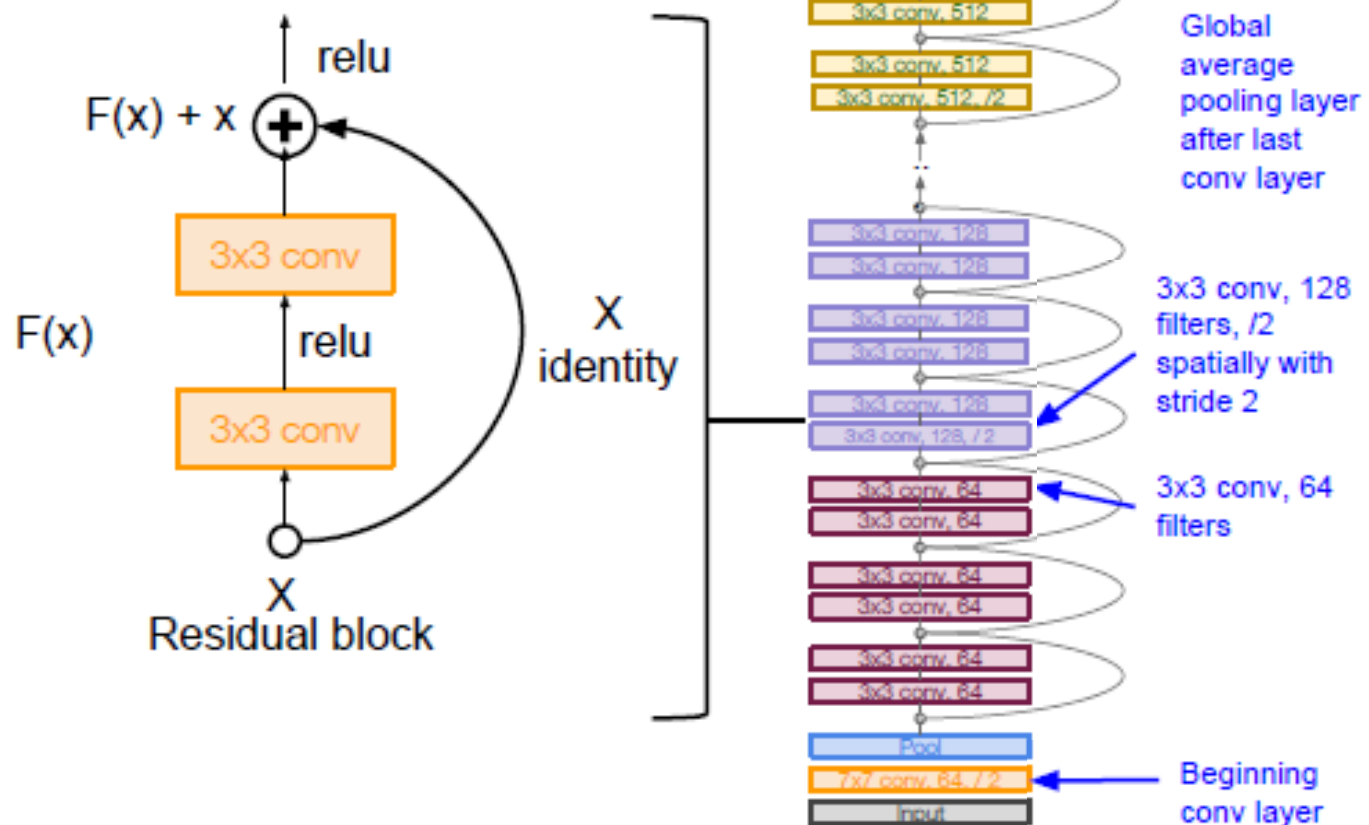


Case Study: ResNet

[He et al., 2015]

Full ResNet architecture:

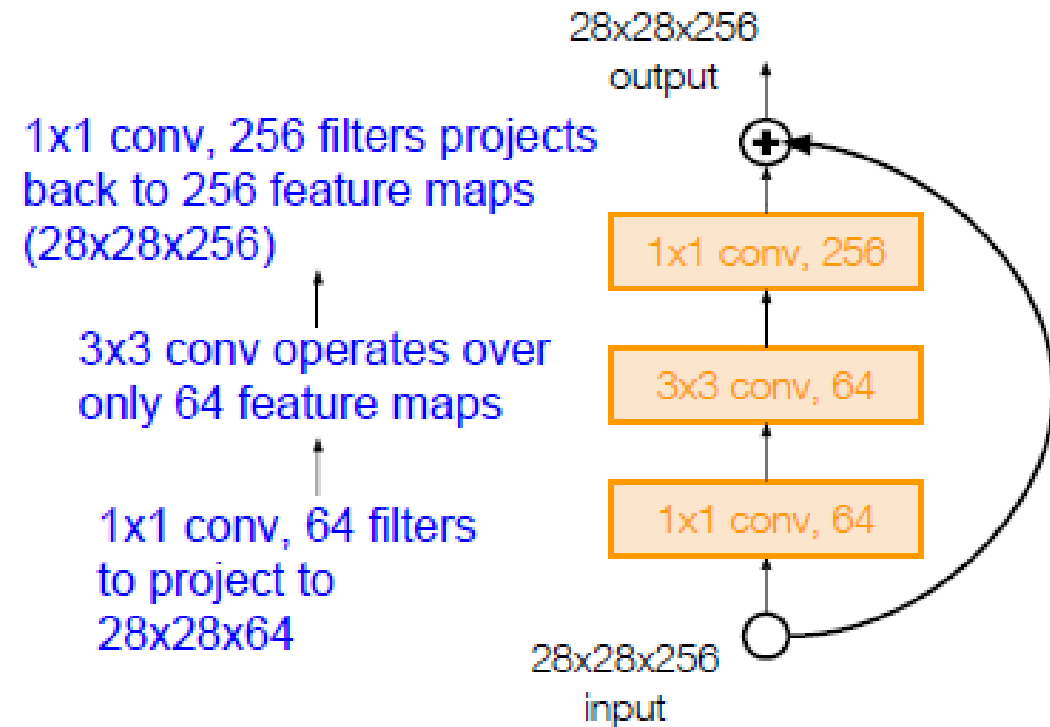
- Stack residual blocks
- Every residual block has two 3x3 conv layers
- Periodically, double # of filters and downsample spatially using stride 2 (/2 in each dimension)
- Additional conv layer at the beginning
- No FC layers at the end (only FC 1000 to output classes)



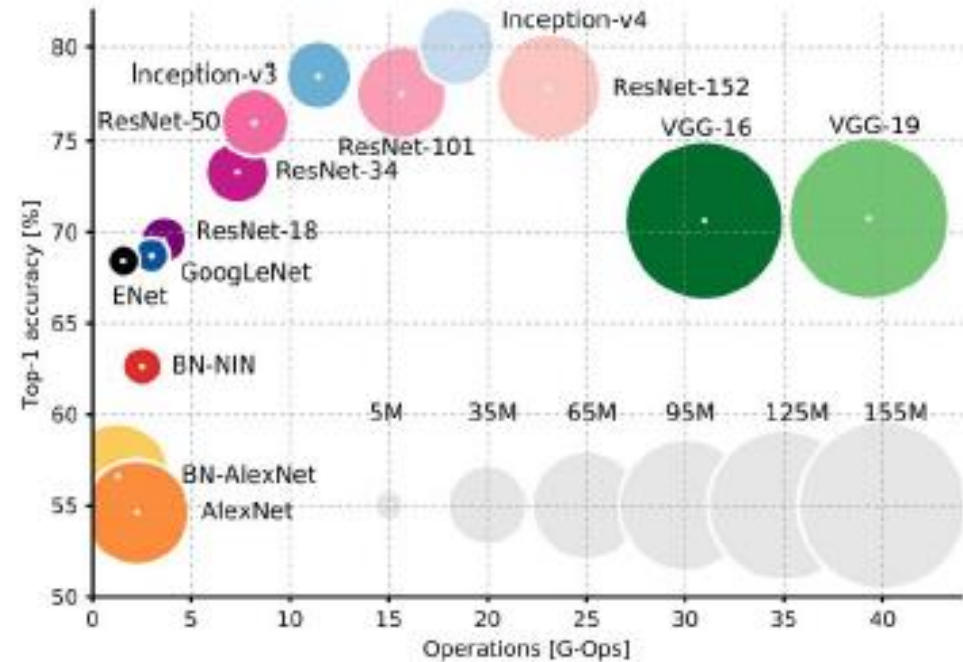
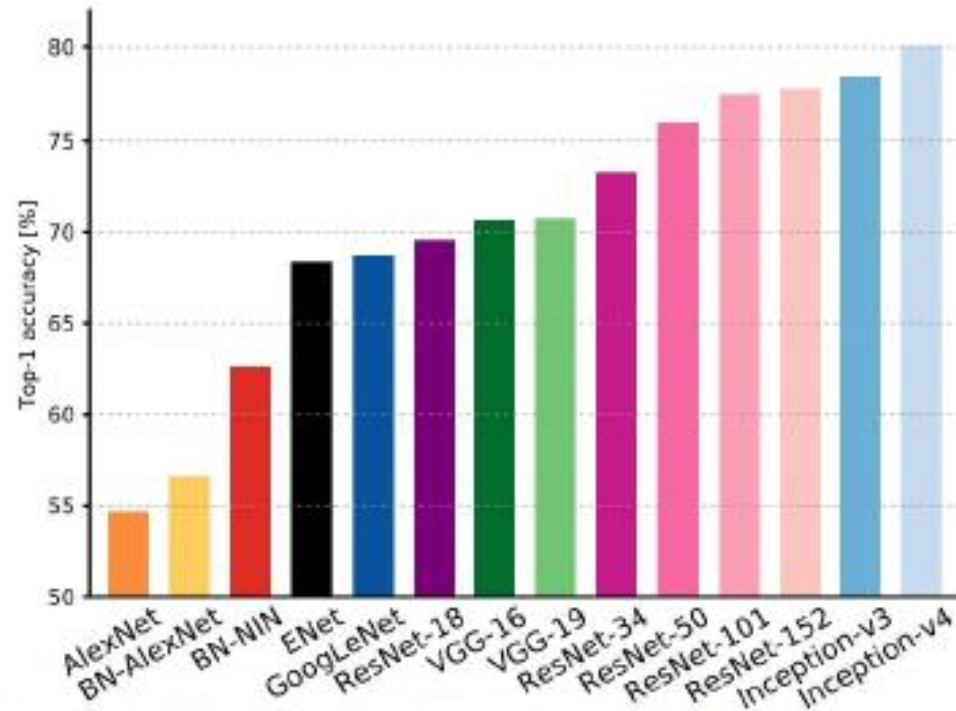
Case Study: ResNet

[He et al., 2015]

For deeper networks
(ResNet-50+), use “bottleneck”
layer to improve efficiency
(similar to GoogLeNet)



Comparing complexity...



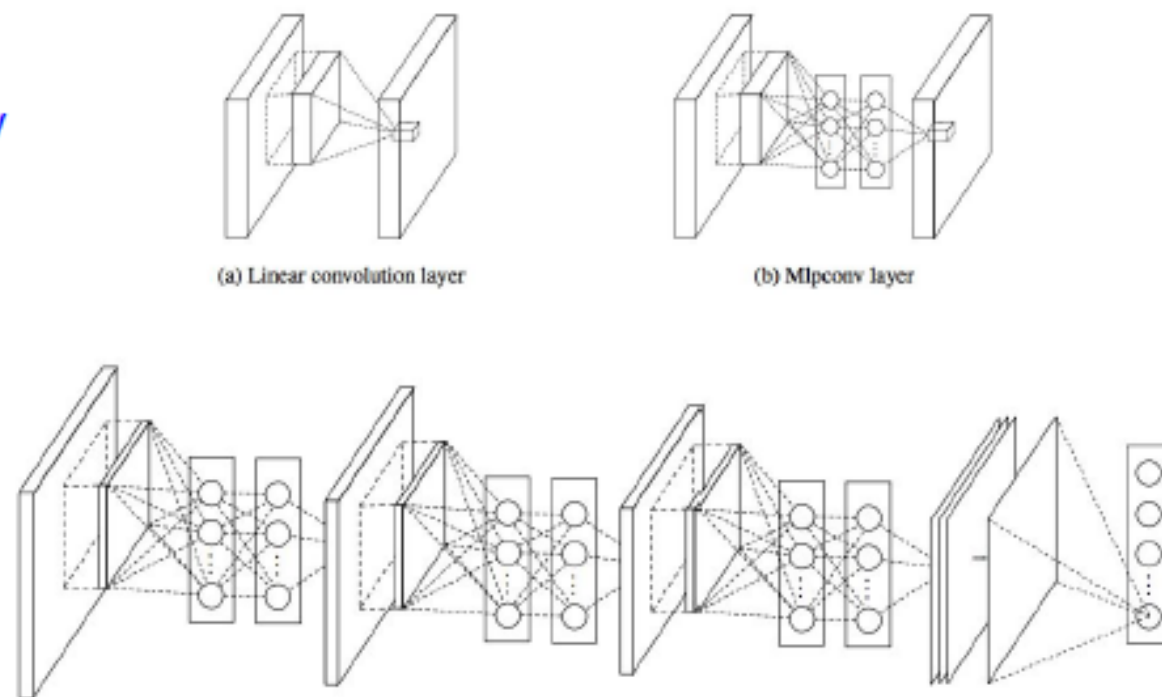
An Analysis of Deep Neural Network Models for Practical Applications, 2017.

Figures copyright Alfredo Canziani, Adam Paszke, Eugenio Culurciello, 2017. Reproduced with permission.

Network in Network (NiN)

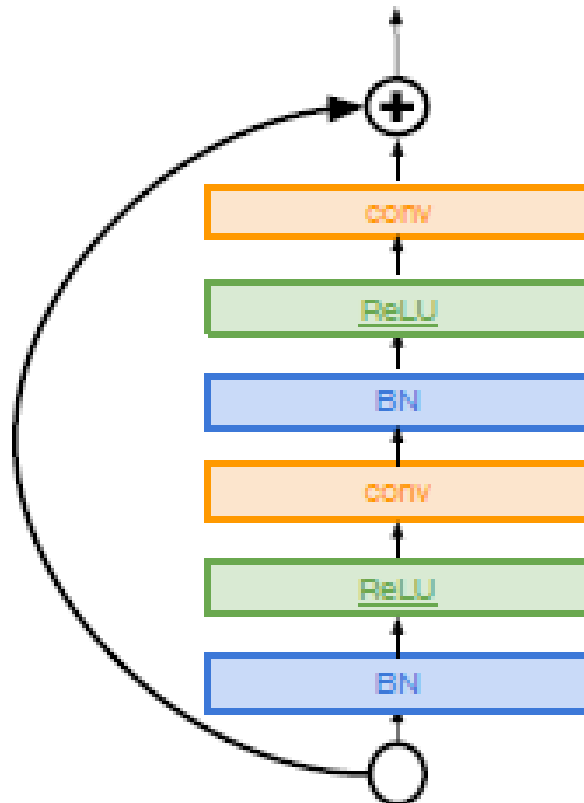
[Lin et al. 2014]

- 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



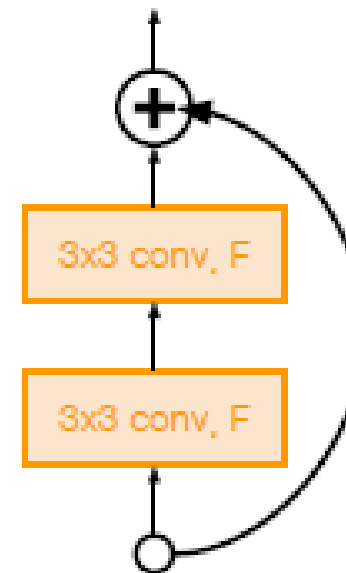
Figures copyright Lin et al., 2014. Reproduced with permission.

Improving ResNets...

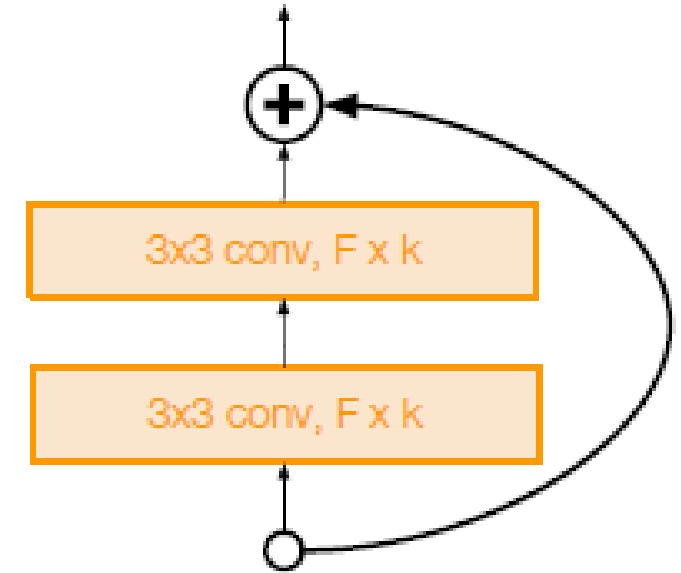


Identity Mappings in Deep Residual Networks

[He et al. 2016]



Basic residual block

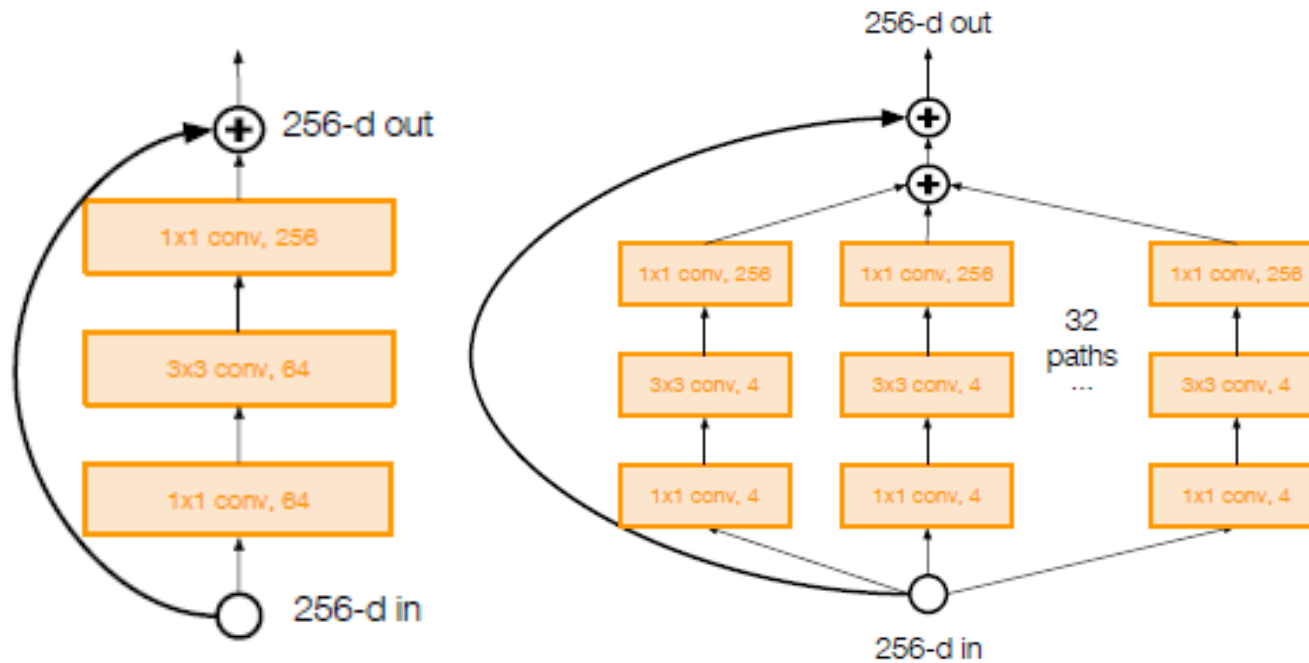


Wide residual block

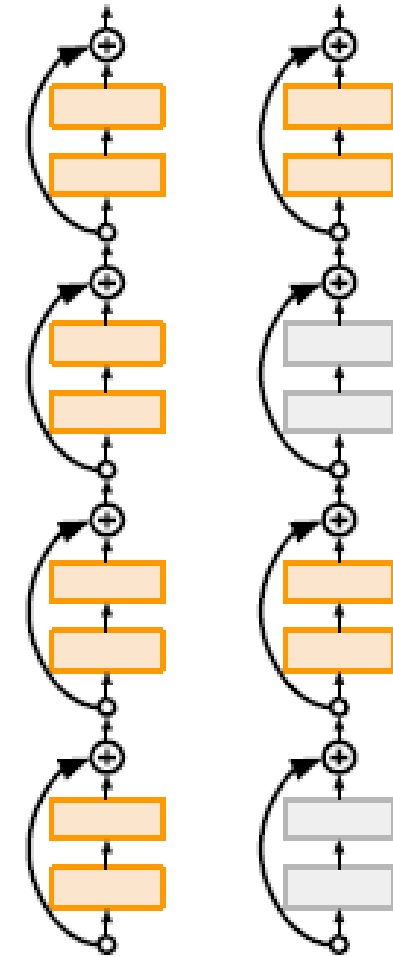
Wide Residual Networks

[Zagoruyko et al. 2016]

Improving ResNets...



Aggregated Residual Transformations for Deep Neural Networks (ResNeXt)

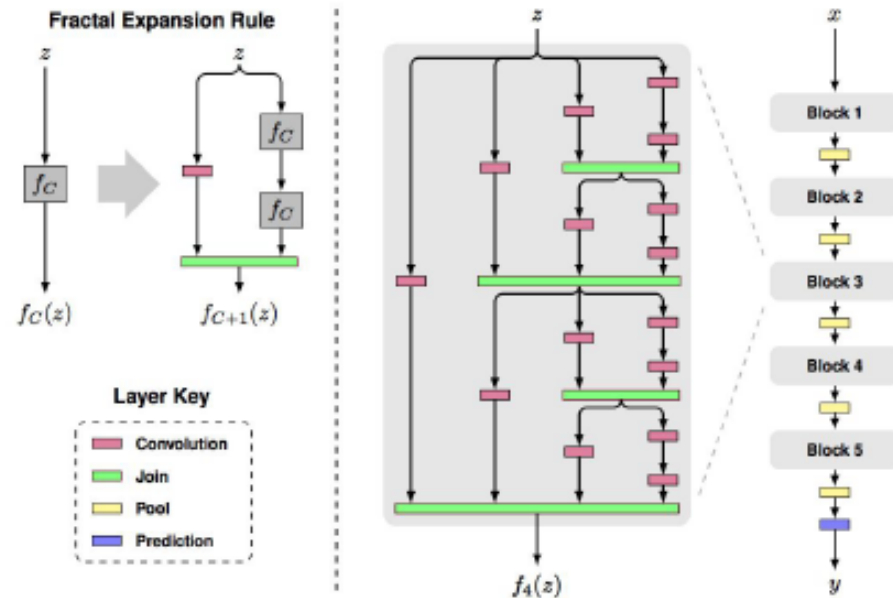


Deep Networks with Stochastic Depth

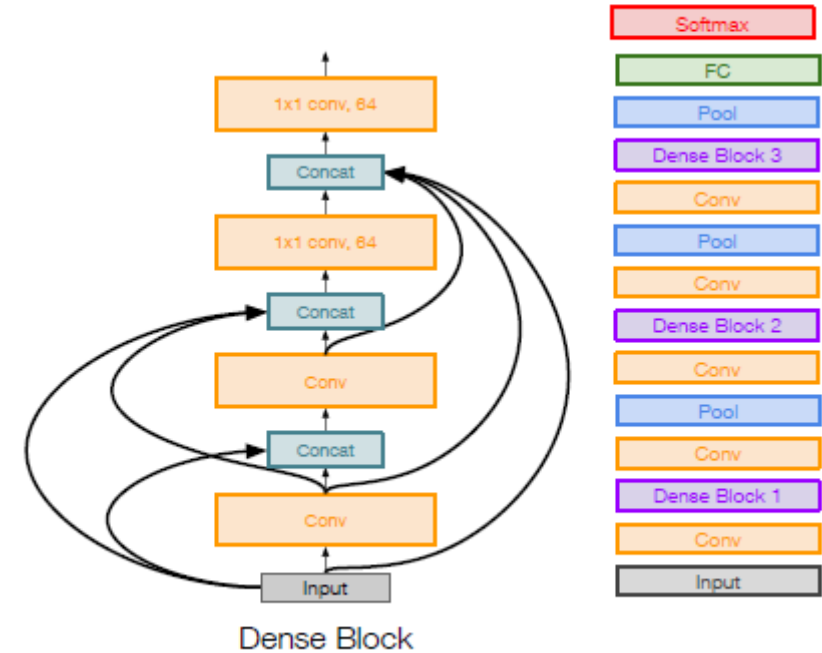
[Huang et al. 2016]

Beyond ResNets...

FractalNet: Ultra-Deep Neural Networks without Residuals



Figures copyright Larsson et al., 2017. Reproduced with permission.

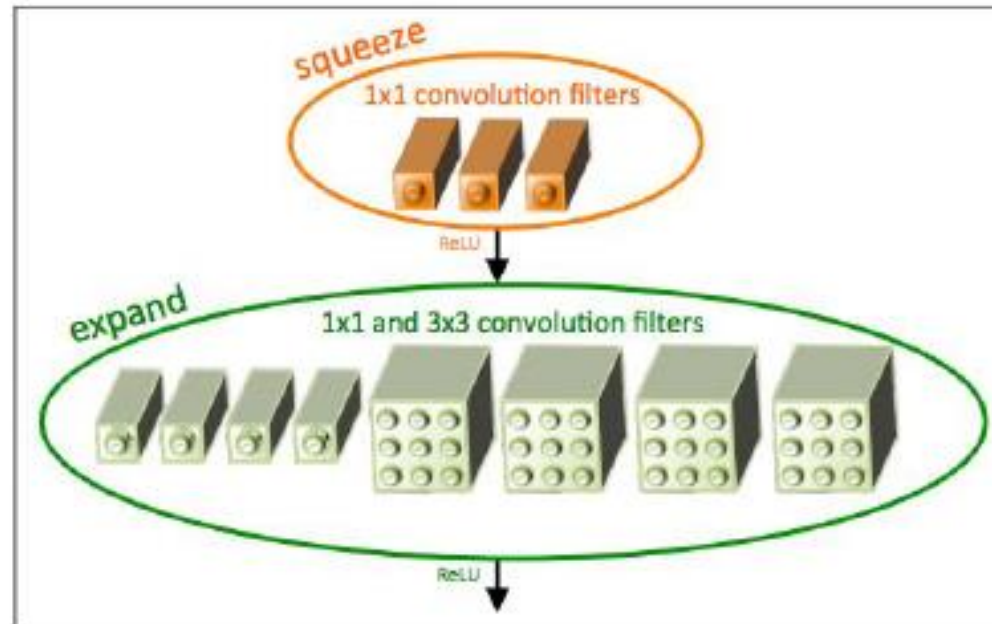


Densely Connected Convolutional Networks

Efficient networks...

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

[Iandola et al. 2017]



감사합니다!

발표 들어주셔서 감사합니다
