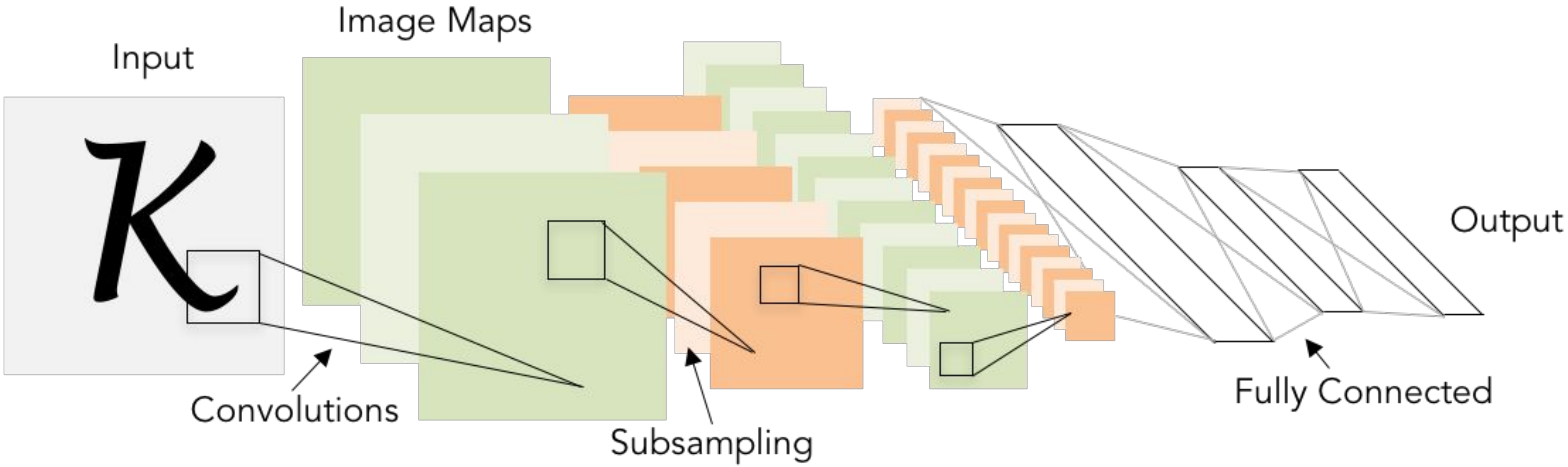
Lecture 9: CNN Architectures

Review: LeNet-5

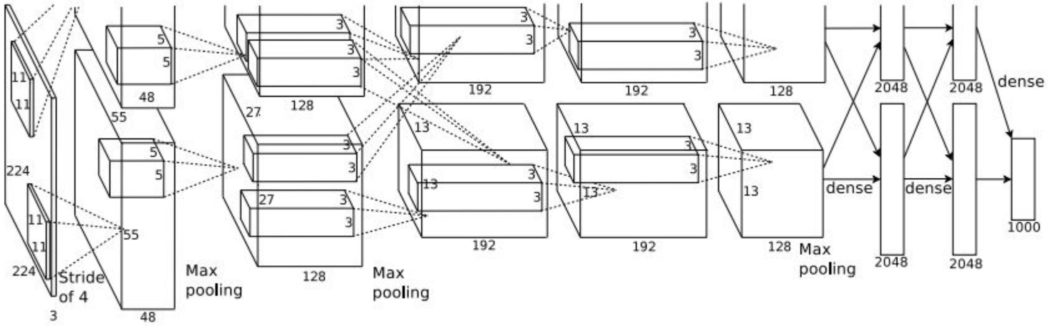
[LeCun et al., 1998]



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

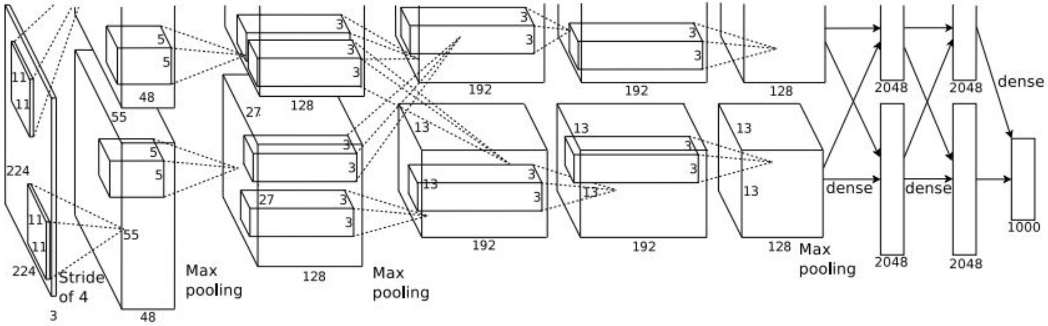
[Krizhevsky et al. 2012]



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

Case Study: AlexNet

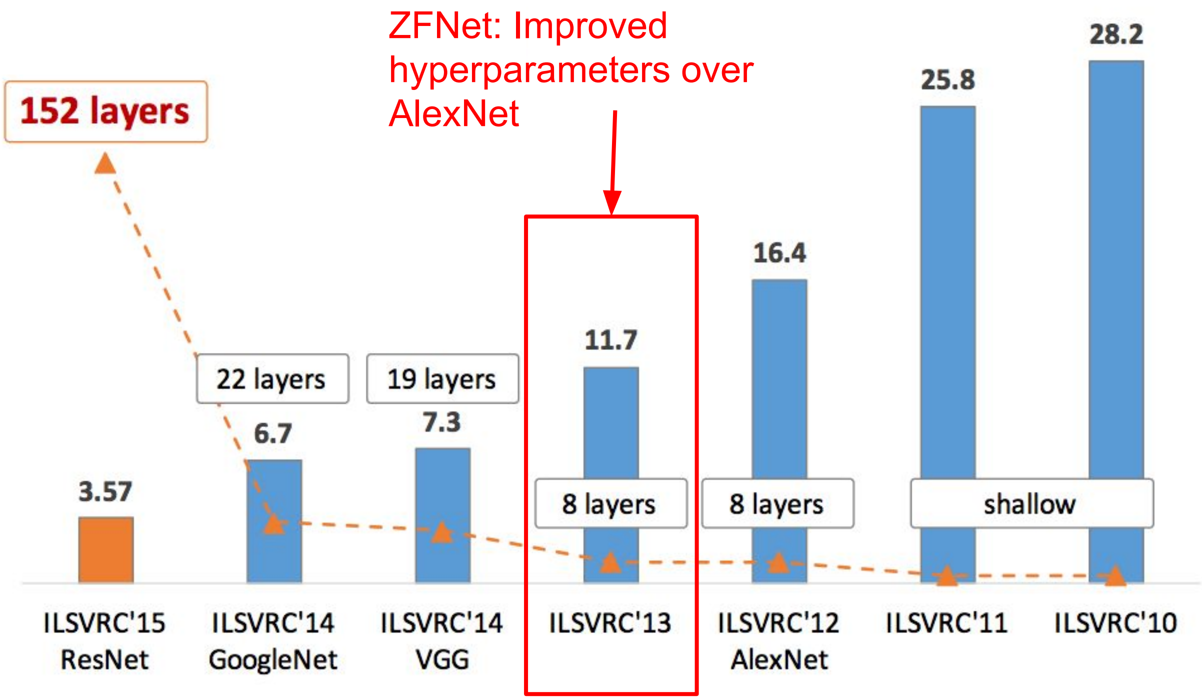
[Krizhevsky et al. 2012]



Input: 227x227x3 images

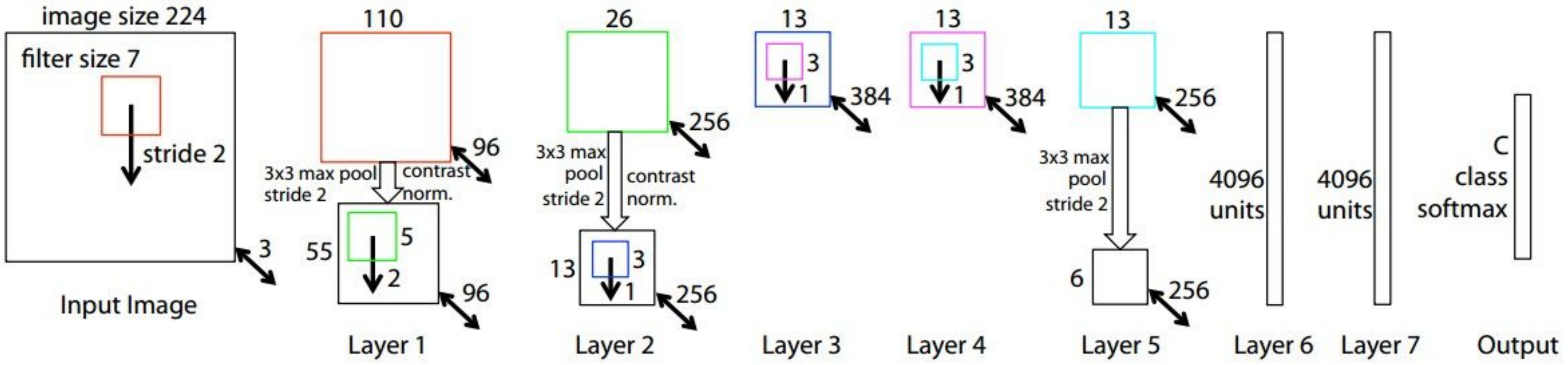
First layer (CONV1): 96 11x11 filters applied at stride 4
=>
Output volume **[55x55x96]**
Parameters: (11*11*3)*96 = **35K**

ImageNet Large Scale Visual Recognition Challenge (ILSVRC) winners



ZFNet

[Zeiler and Fergus, 2013]

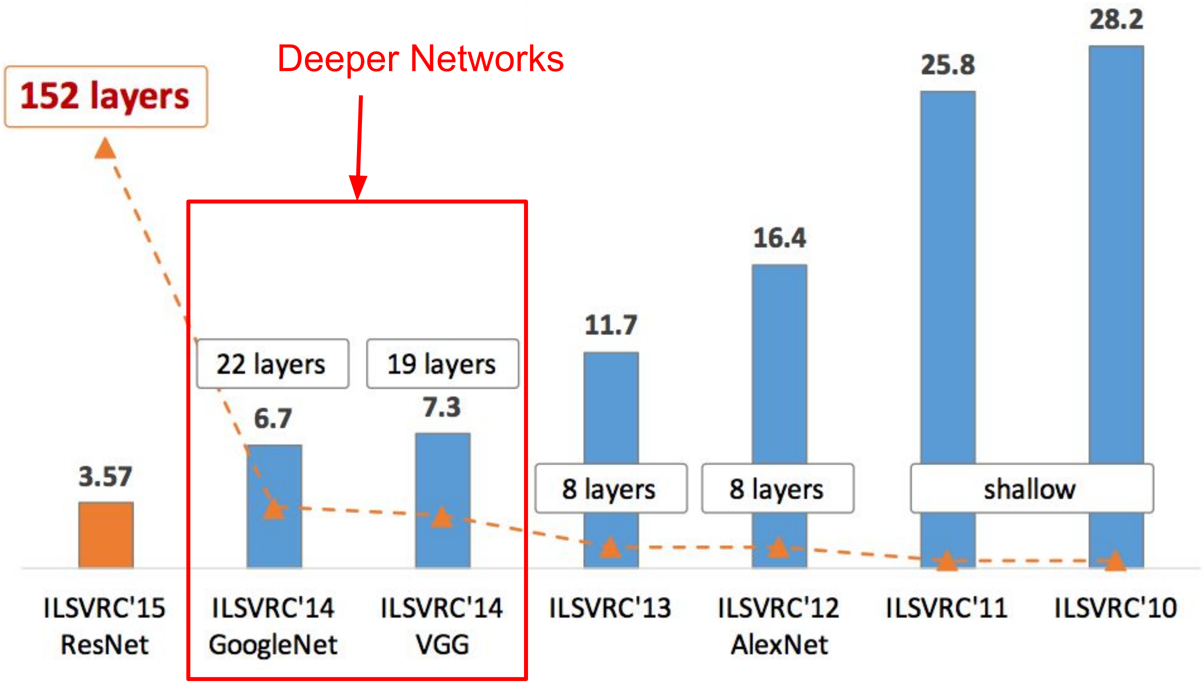


TODO: remake figure

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



Case Study: VGGNet

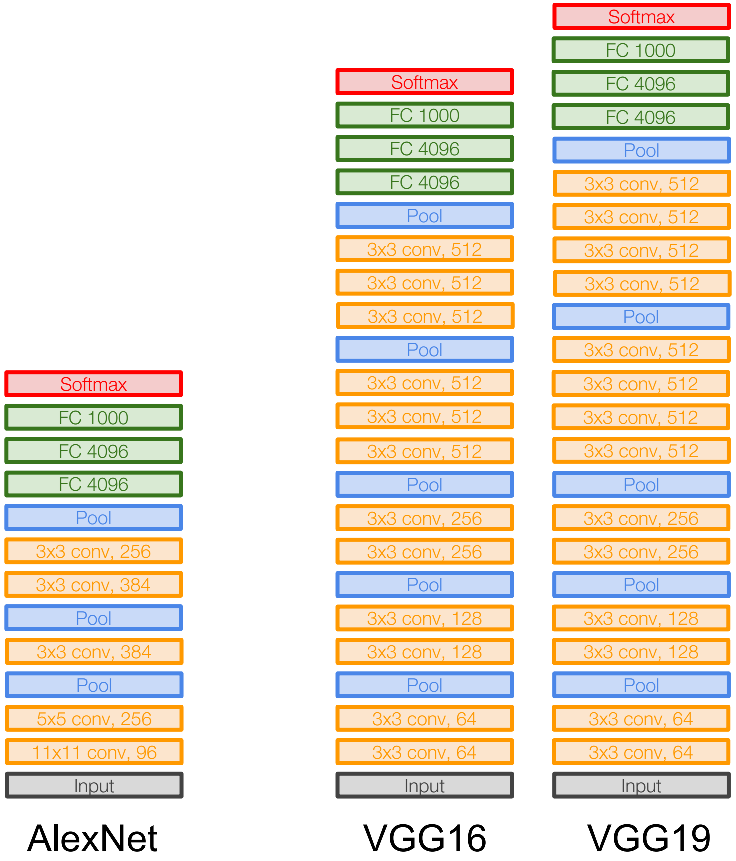
[Simonyan and Zisserman, 2014]

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

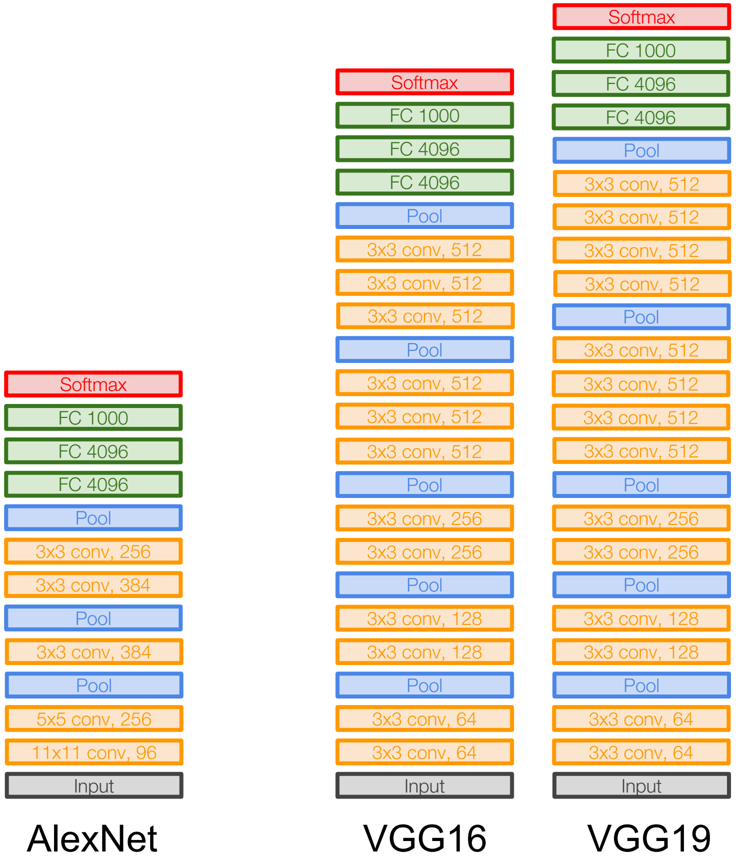
[Simonyan and Zisserman, 2014]

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 * (3^2C^2) vs. 7^2C^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

VGG16

TOTAL memory: 24M * 4 bytes ~= 96MB / image (only forward! ~*2 for bwd)
TOTAL params: 138M parameters

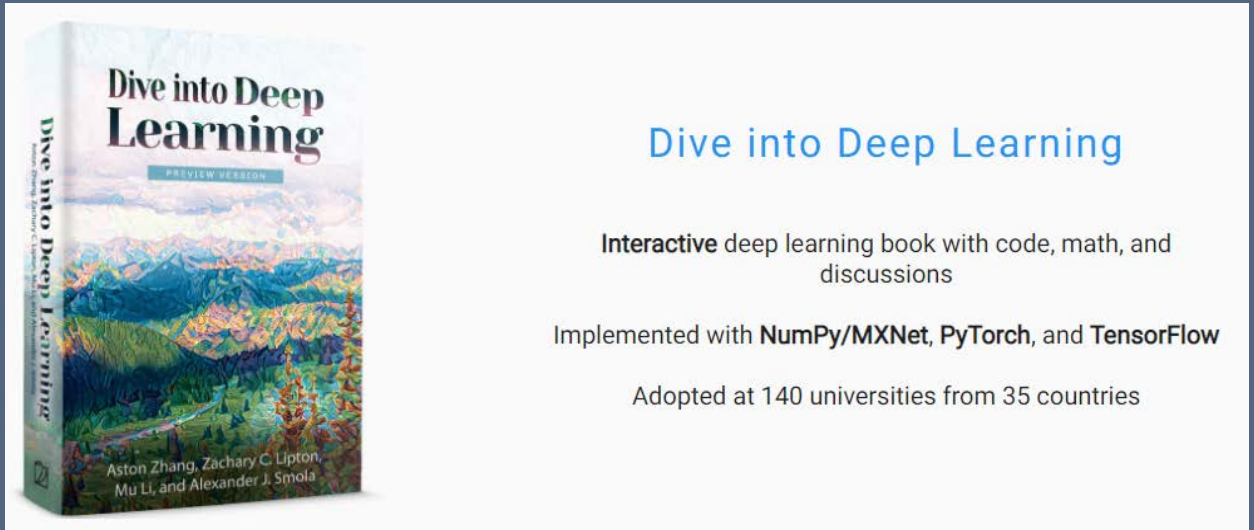
Linear Neural Networks/

LeeSaeBom

- *Network in Network*
- *Batch Normalixation*

Dive into Deep Learning

Linear Neural Networks



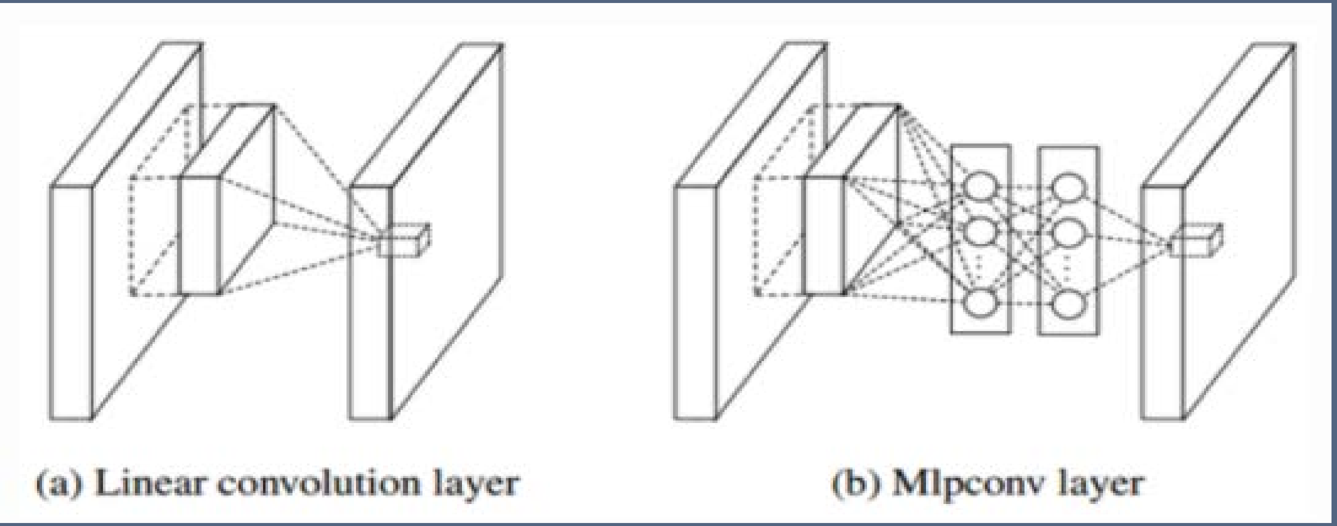
Dive into Deep Learning

Network in Network

- *Network in Network*

- *CNN :* Filter을 이용하여 Stride만큼 이동하면서 CONV으로 Feature 추출함

- *NiN :* Filter대신에 MLP를 사용 = Mlpconv layer



Network in Network

Batch Normalization

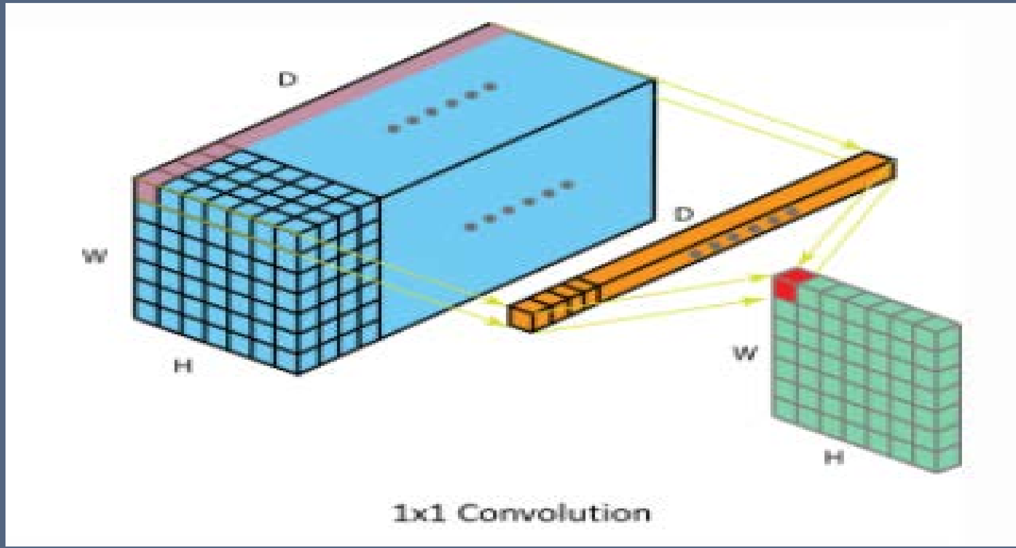
Dive into Deep Learning

Network in Network

- *Network in Network*

- *Why use the NiN?* 1x1 Conv를 통해 Feature map 개수를 줄일 수 있다
  = Parameter 수를 줄일 수 있다

- **NiN은 Mlpconv layer를 여러 개 쌓아 사용했으므로 네트워크 안에 네트워크가 있다는 개념을 NIN으로 불린다.**



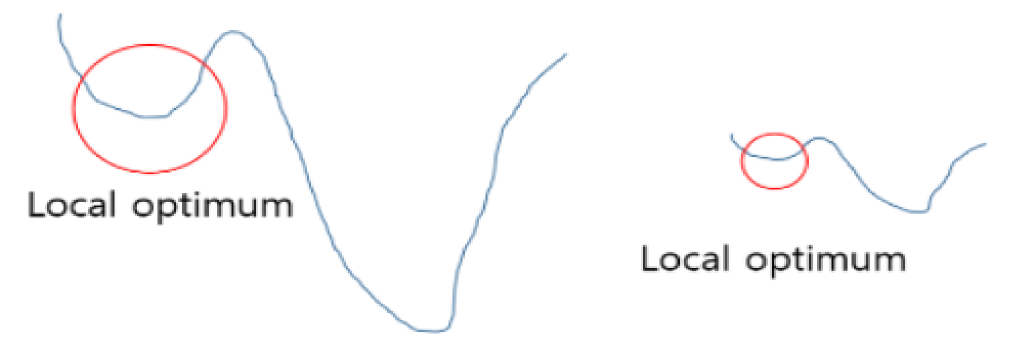
Network in Network

Batch Normalization

Dive into Deep Learning

Batch Normalization

- *Batch :* **신경망을 학습시킬 때, 한 번에 학습시키지 않고 조그만 단위로 분할해서 학습을 시키는데 이 때의 조그만 단위가 배치**
- *Batch Normalization :* **배치별로 구분하고 각각의 출력값들의 정규화**



(좌) Normalization 적용 전 / (우) Normalization 적용 후

- *Internal Covariance Shift*

- *Covariate Shift :* 이전 레이어의 파라미터 변화로 인하여 현재 레이어의 입력의 분포가 바뀌는 현상
- *Internal Covariance Shift :* 레이어를 통과할 때 마다 Covariate Shift가 일어나면서 입력의 분포가 약간씩 변하는 현상

Network in Network

Batch Normalization

Dive into Deep Learning

Batch Normalization

- *Batch Normalization Algorithm*

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1 \ldots m}\}$;
Parameters to be learned: γ, β

Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma_{\mathcal{B}}^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m} (x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma \widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

Algorithm 1: Batch Normalizing Transform, applied to activation x over a mini-batch.

- mini-batch의 평균과 분산을 구하고 입력 데이터를 평균이 0, 분산이 1로 되게 정규화를 진행함.
- scale(확대) and shift(이동)를 거쳐 학습 가능한 변수를 γ, β 통해 실행
- γ, β 는 역전파에 의해 학습된 변수

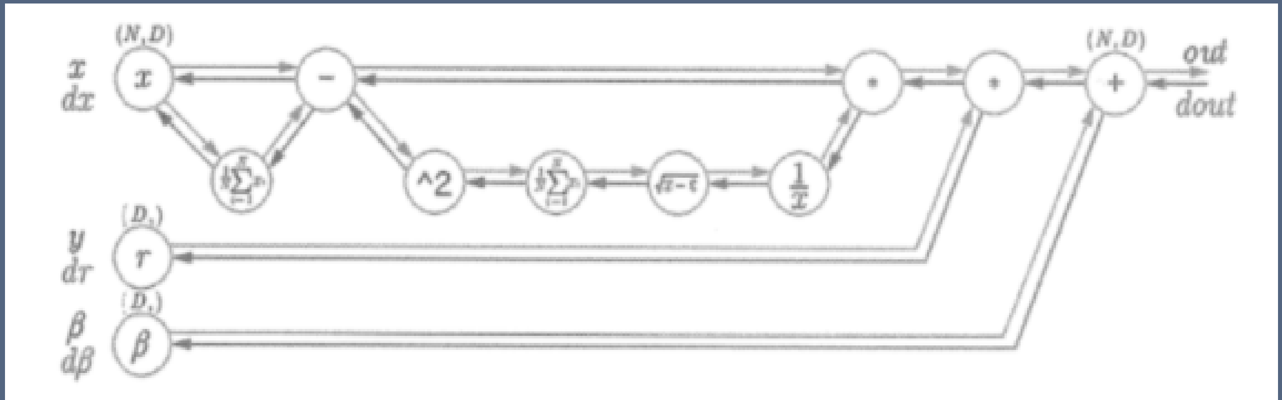
Network in Network

Batch Normalization

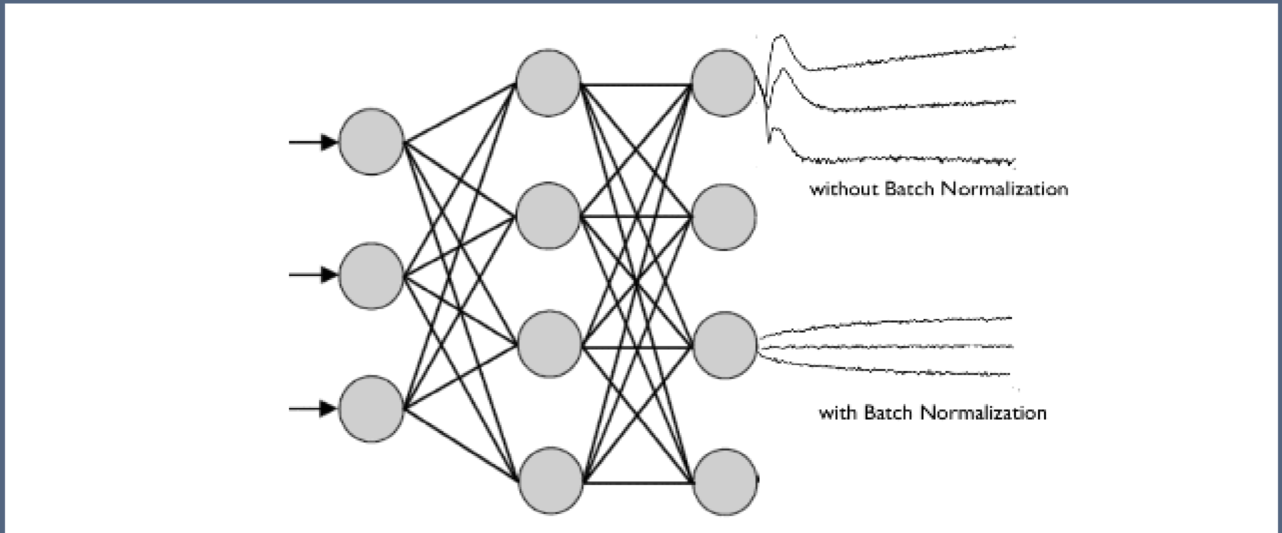
Dive into Deep Learning

Batch Normalization

- *배치정규화 계산 그래프*



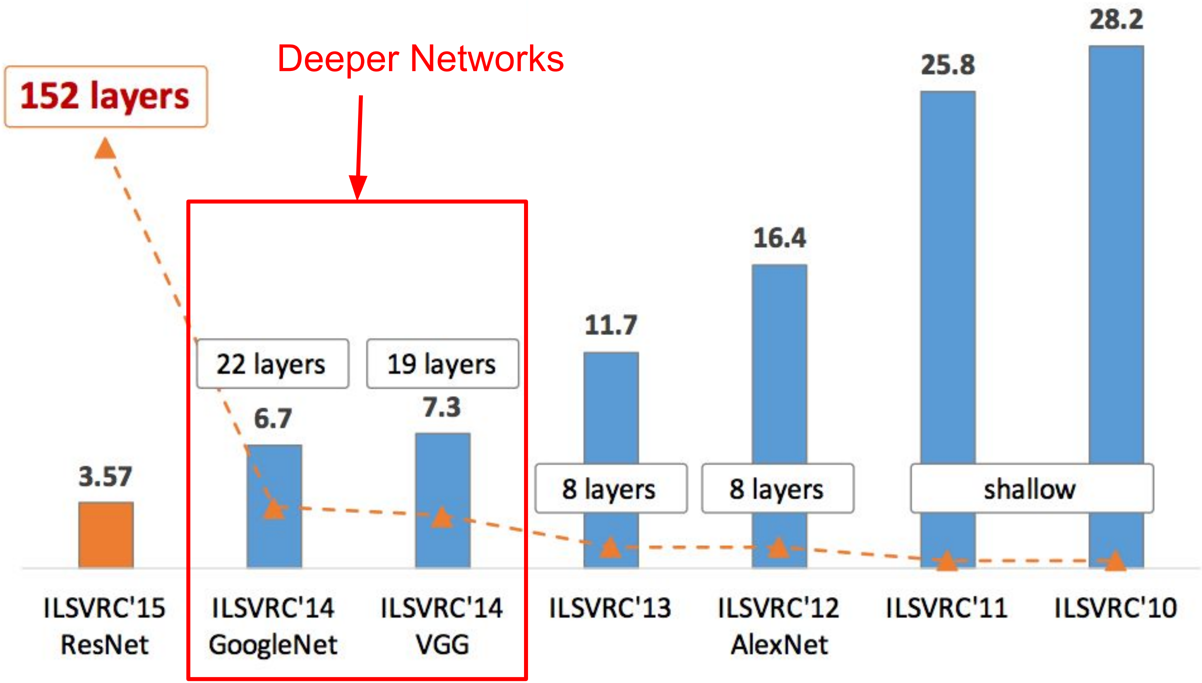
- *신경망에서의 배치 정규화*



Network in Network

Batch Normalization

ImageNet Large Scale Visual Recognition Challenge (ILSVRC) winners

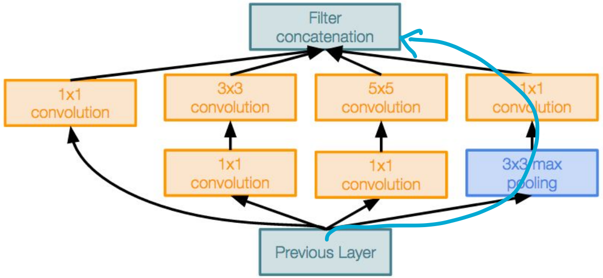


Case Study: GoogLeNet

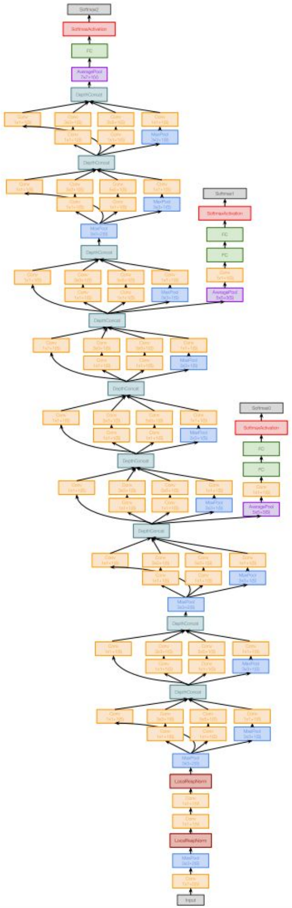
[Szegedy et al., 2014]

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)

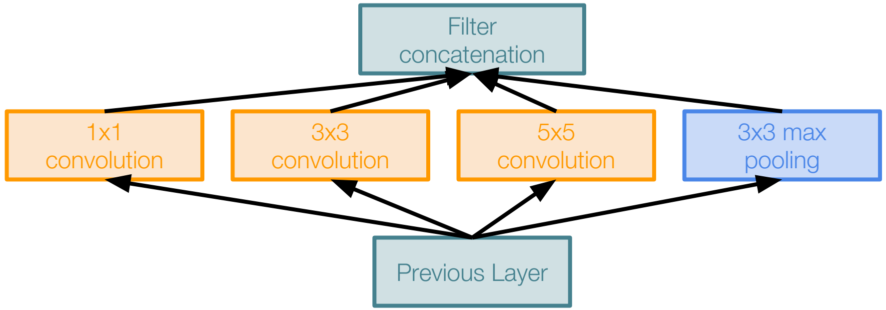


Inception module



Case Study: GoogLeNet

[Szegedy et al., 2014]



Naive Inception module

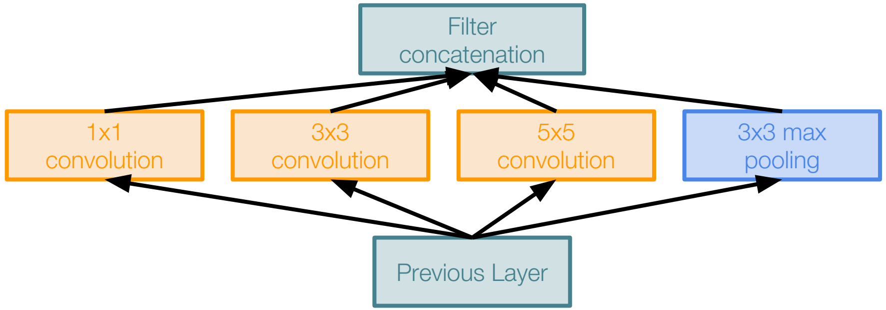
Apply parallel filter operations on the input from previous layer:

- 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]



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

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

Case Study: GoogLeNet

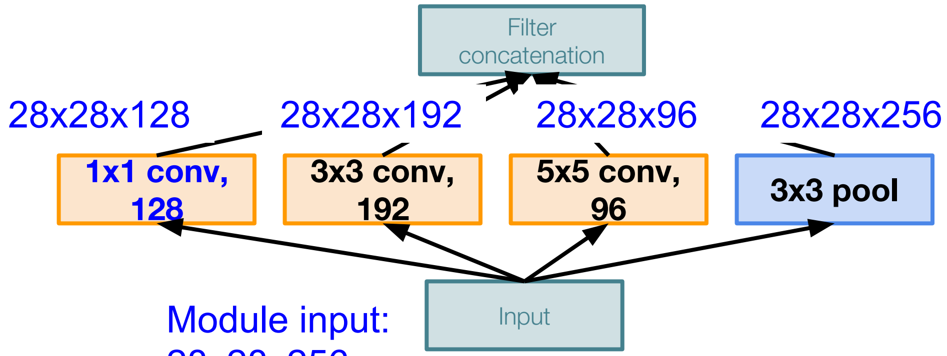
[Szegedy et al., 2014]

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

Example:

Q3:What is output size after filter concatenation?

28x28x(128+192+96+256) = 28x28x672



Module input: 28x28x256

Naive Inception module

Conv Ops:
[1x1 conv, 128] 28x28x128x1x1x256
[3x3 conv, 192] 28x28x192x3x3x256
[5x5 conv, 96] 28x28x96x5x5x256
Total: 854M ops

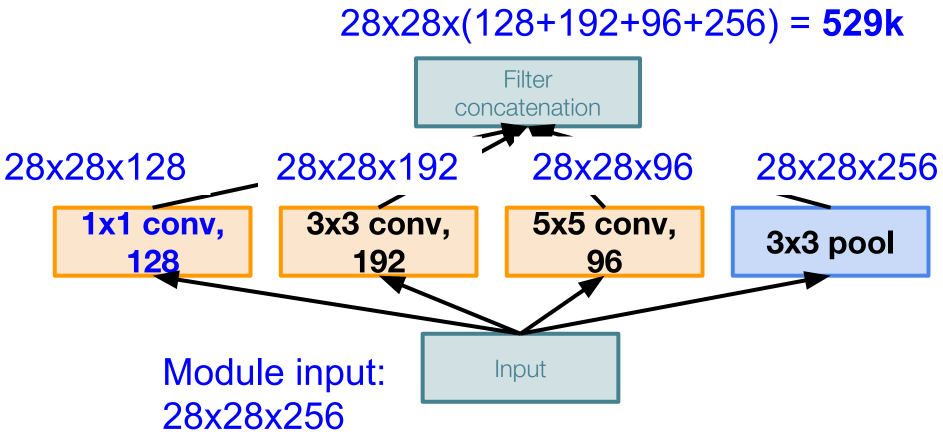
Case Study: GoogLeNet

[Szegedy et al., 2014]

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

Example:

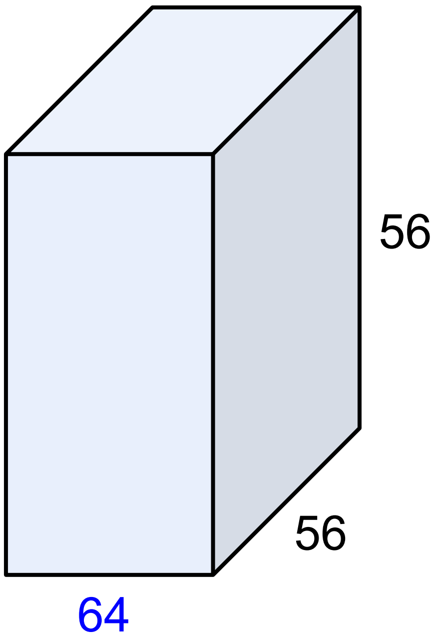
Q3:What is output size after filter concatenation?



Naive Inception module

Solution: "bottleneck" layers that use 1x1 convolutions to reduce feature depth

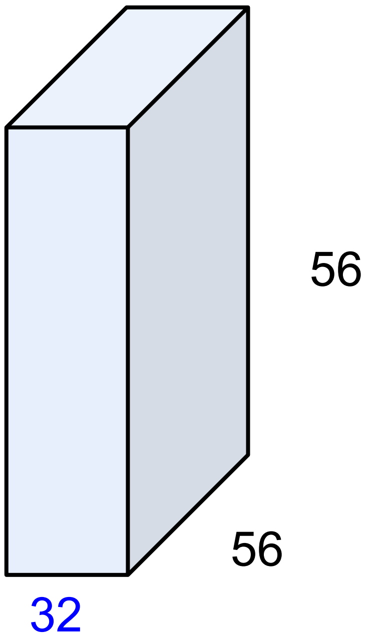
Reminder: 1x1 convolutions



1x1 CONV
with 32 filters

preserves spatial dimensions, reduces depth!

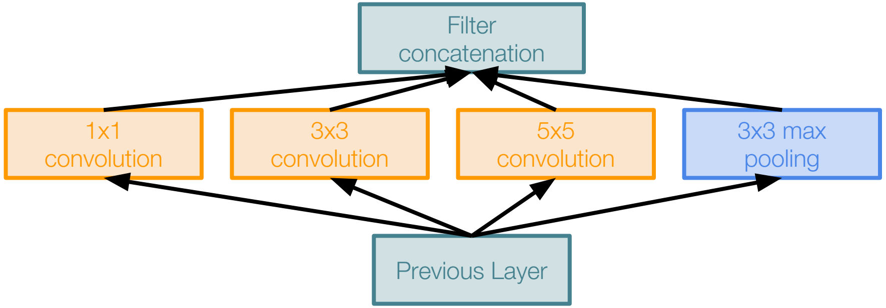
Projects depth to lower dimension (combination of feature maps)



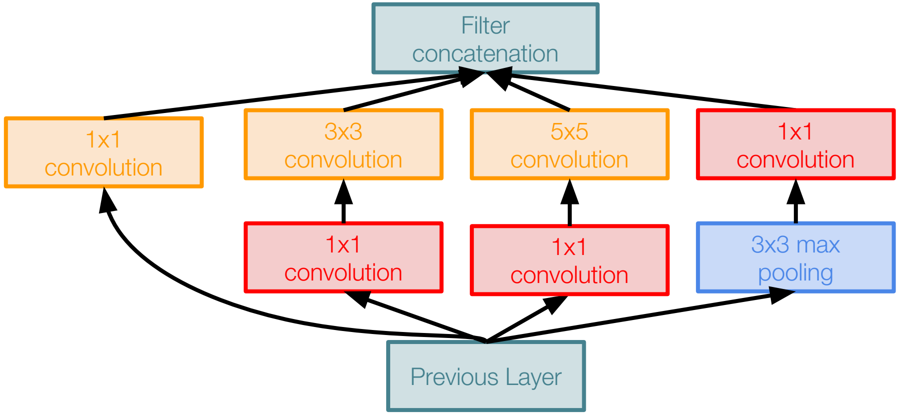
Case Study: GoogLeNet

[Szegedy et al., 2014]

1x1 conv "bottleneck" layers



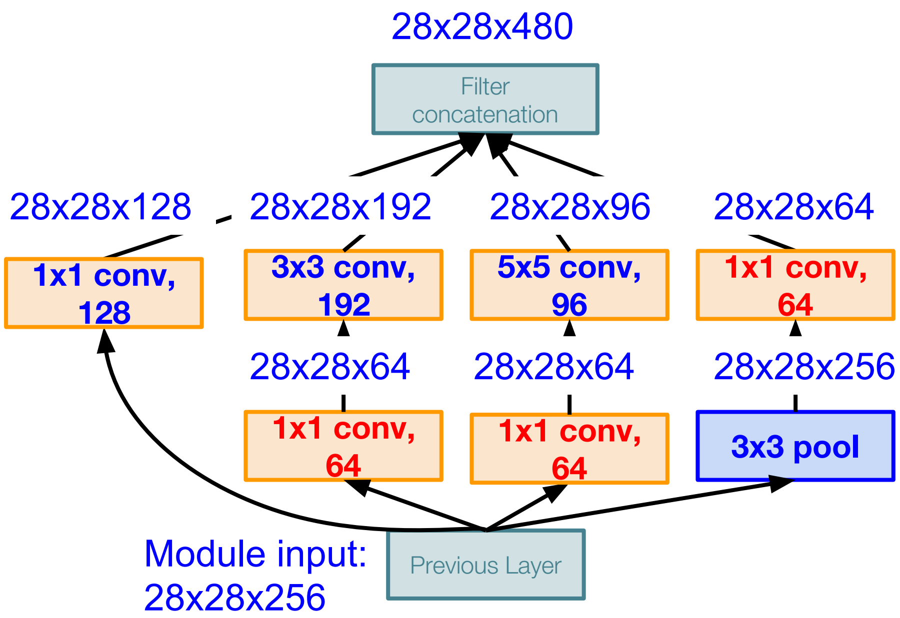
Naive Inception module



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:

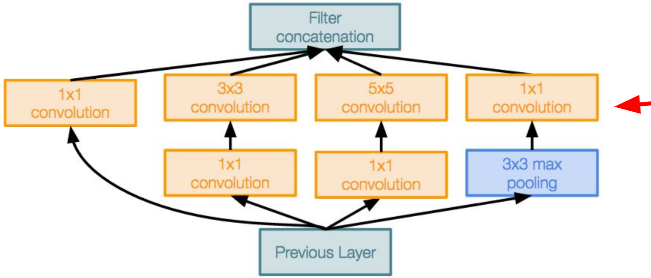
[1x1 conv, 64] 28x28x64x1x1x256
[1x1 conv, 64] 28x28x64x1x1x256
[1x1 conv, 128] 28x28x128x1x1x256
[3x3 conv, 192] 28x28x192x3x3x64
[5x5 conv, 96] 28x28x96x5x5x64
[1x1 conv, 64] 28x28x64x1x1x256
Total: 358M ops

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

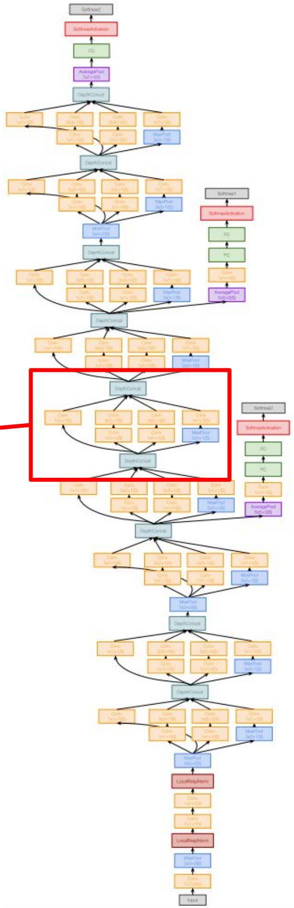
Case Study: GoogLeNet

[Szegedy et al., 2014]

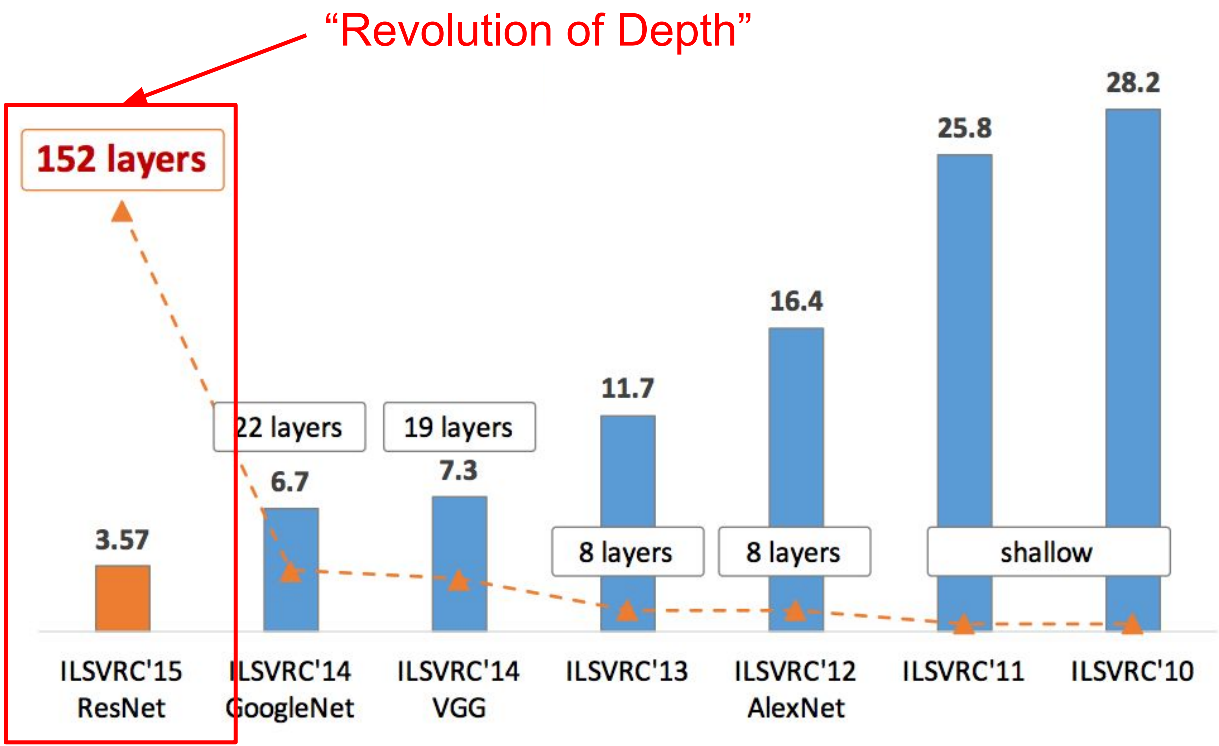
Stack Inception modules with dimension reduction on top of each other



Inception module



ImageNet Large Scale Visual Recognition Challenge (ILSVRC) winners

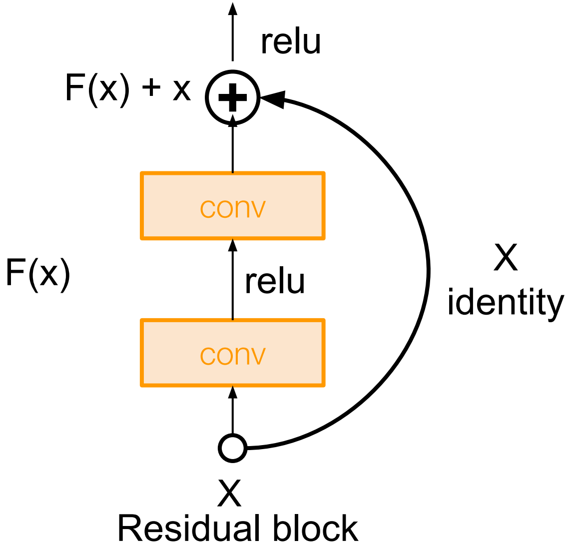


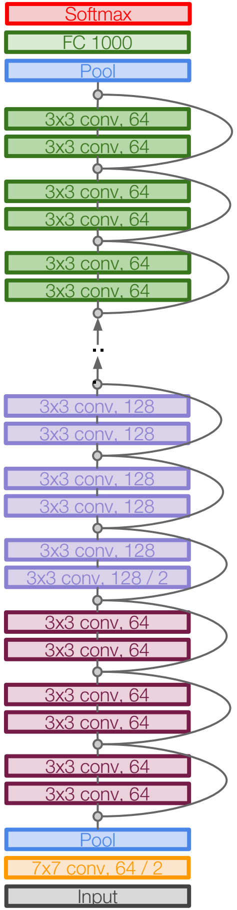
Case Study: ResNet

[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!

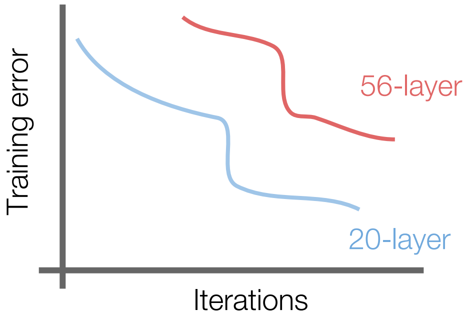


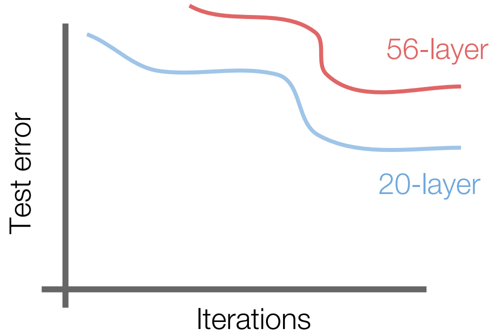


Case Study: ResNet

[He et al., 2015]

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



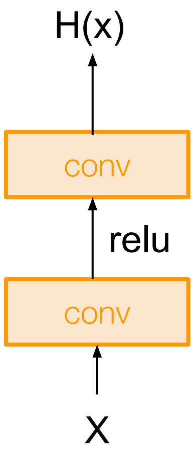


Q: What’s strange about these training and test curves?
[Hint: look at the order of the curves]

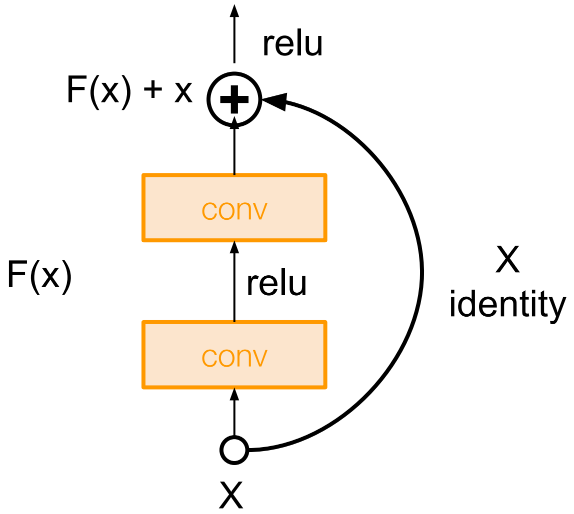
Case Study: ResNet

[He et al., 2015]

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



"Plain" layers

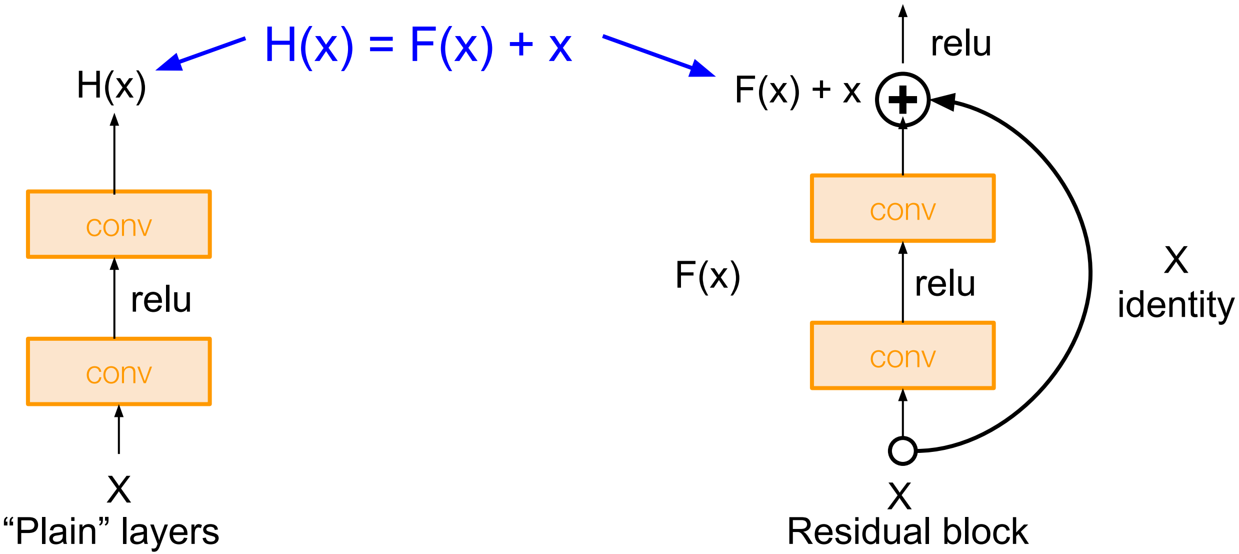


Residual block

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



Use layers to fit residual F(x) = H(x) - x instead of H(x) directly

Best paper award

DENSELY CONNECTED CONVOLUTIONAL NETWORKS

Gao Huang*, Zhuang Liu*, Laurens van der Maaten, Kilian Q. Weinberger

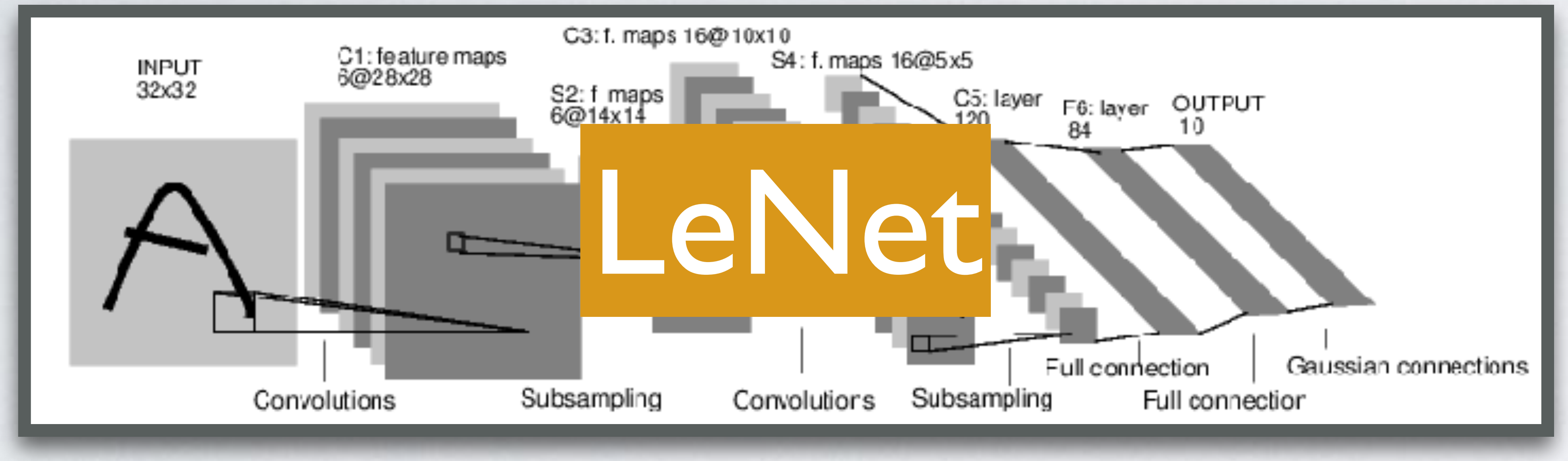
Cornell University

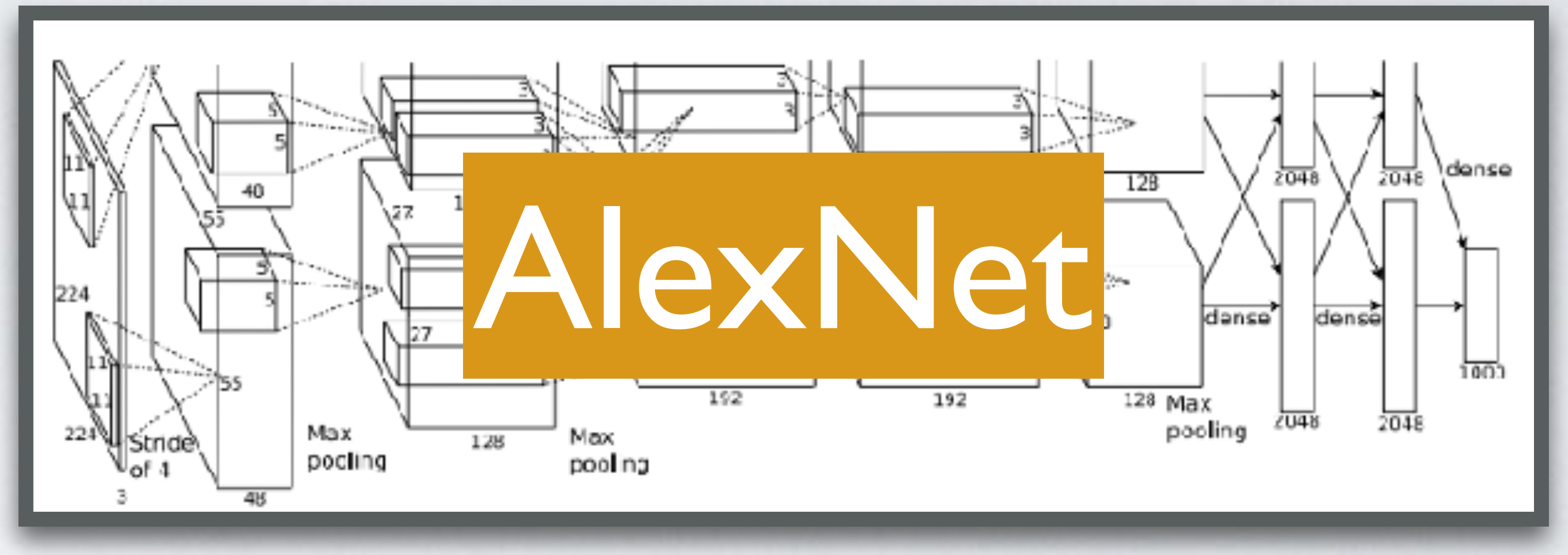
Tsinghua University

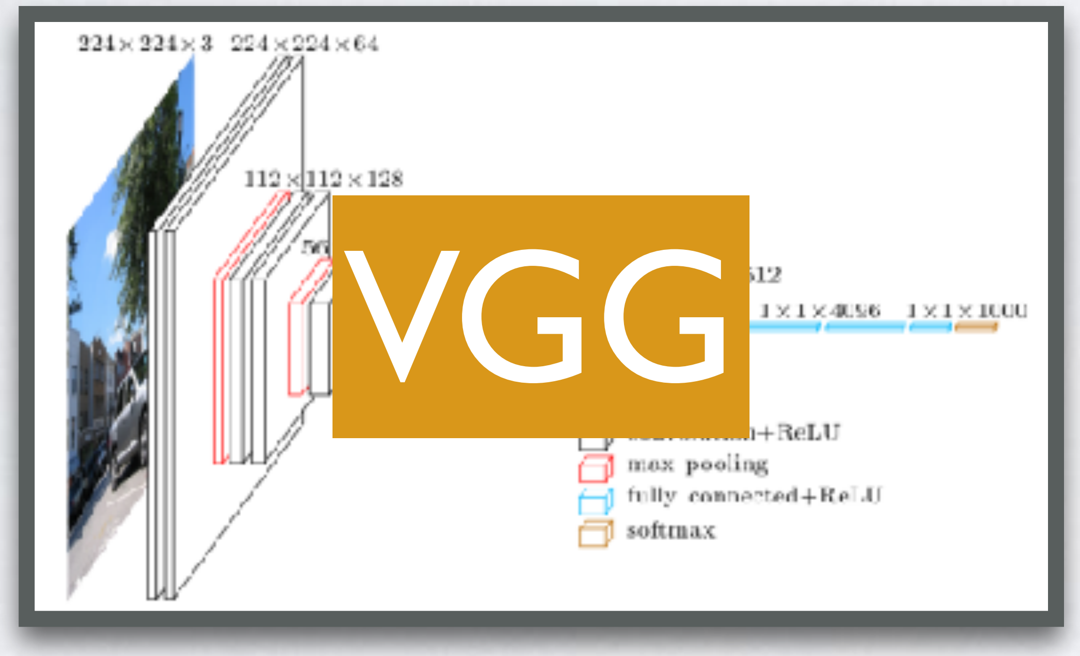
Facebook AI Research

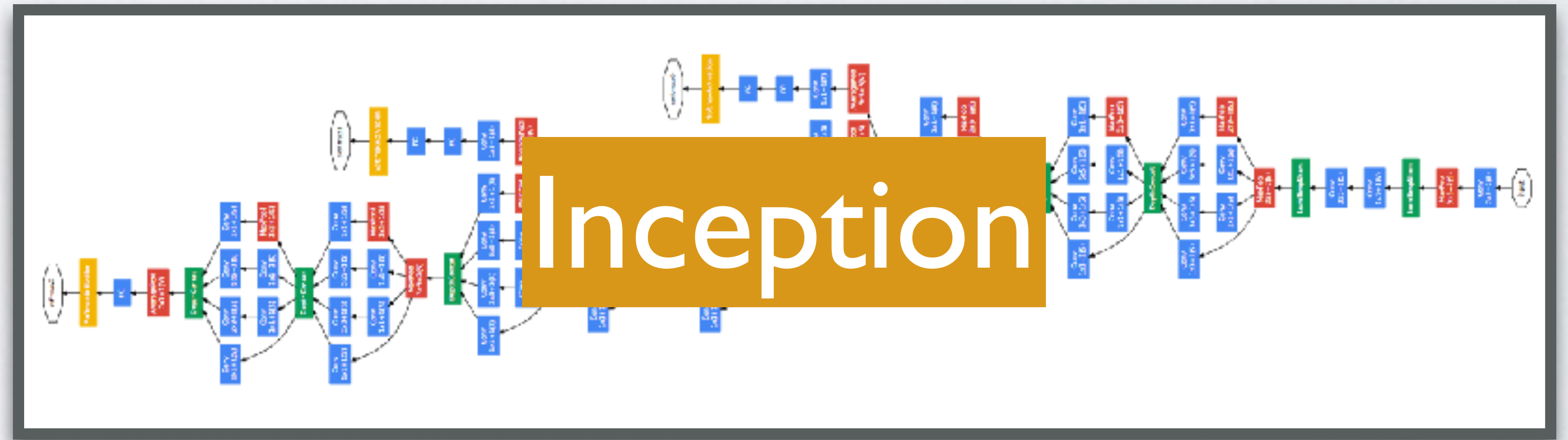
CVPR 2017

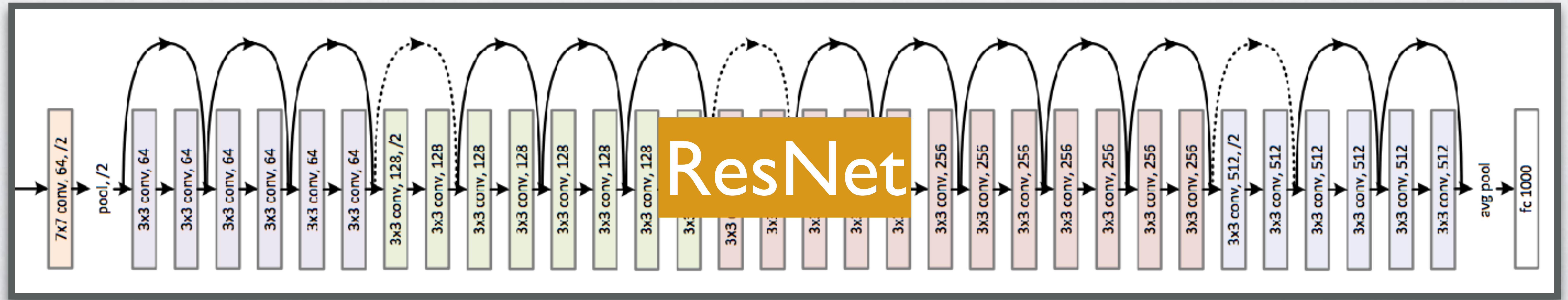
CONVOLUTIONAL NETWORKS



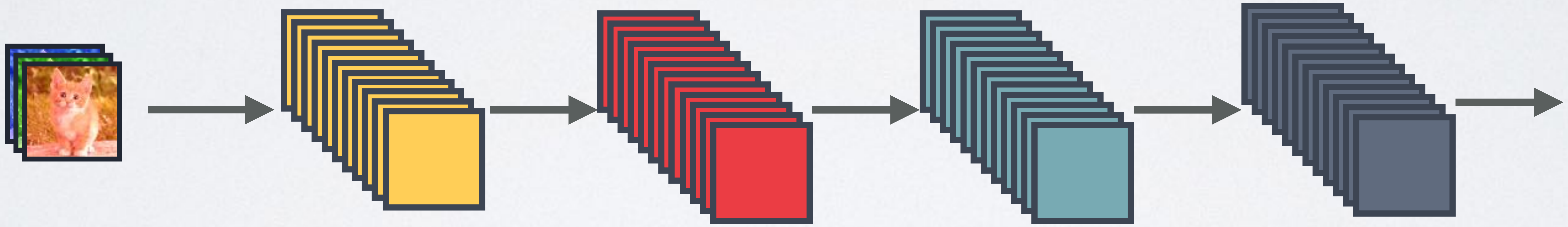






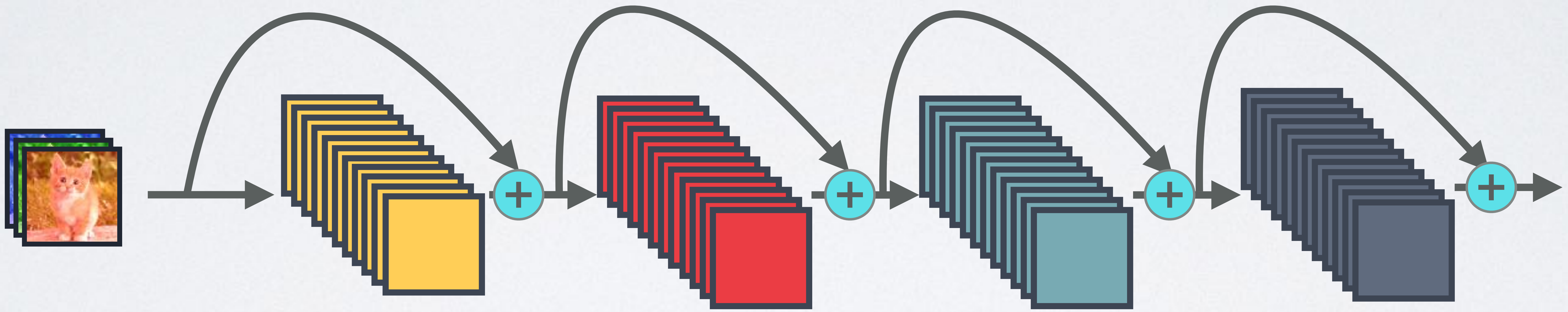


STANDARD CONNECTIVITY



RESNET CONNECTIVITY

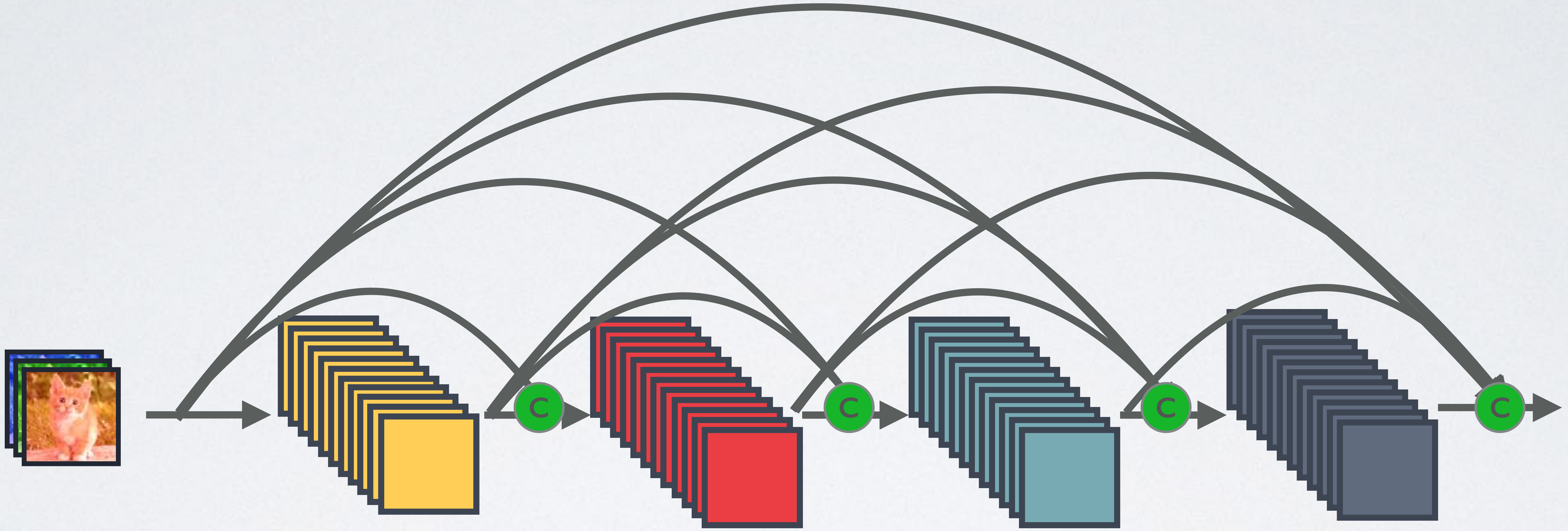
Identity mappings promote gradient propagation.



 : Element-wise addition

Deep residual learning for image recognition: [He, Zhang, Ren, Sun] (CVPR 2015)

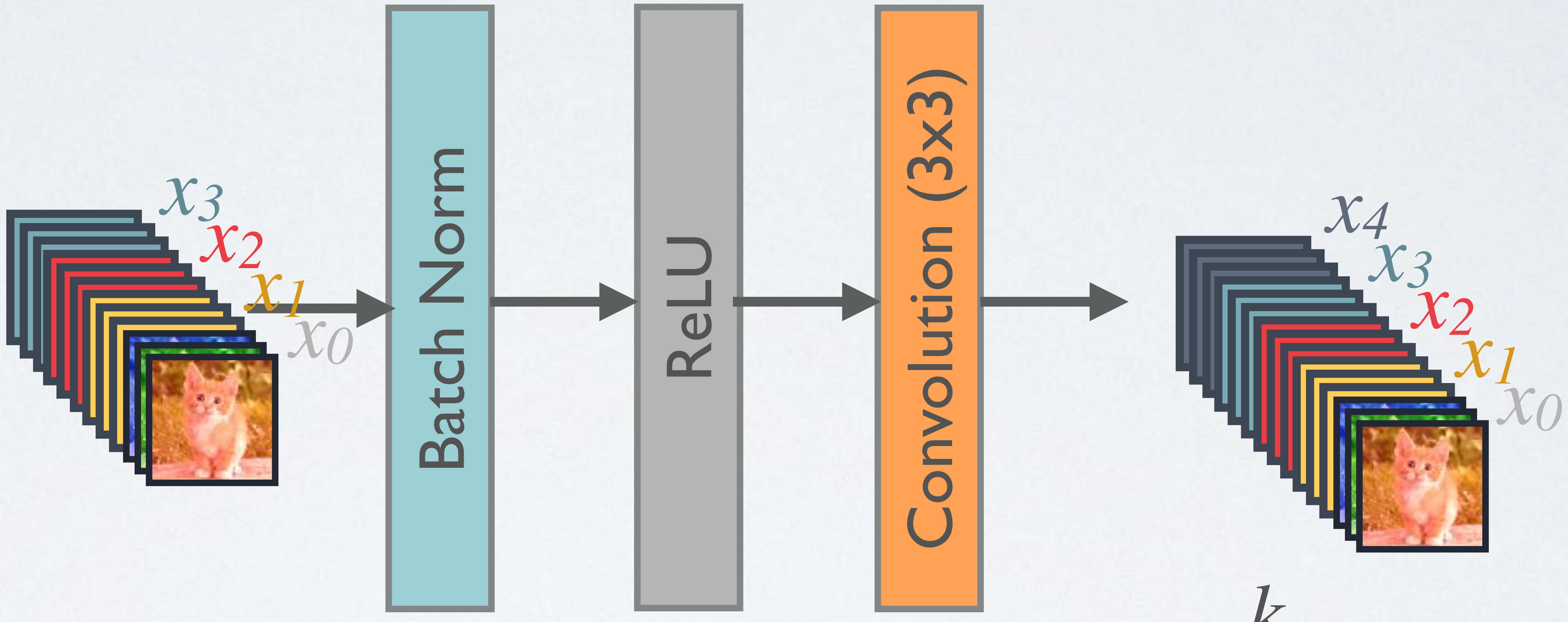
DENSE CONNECTIVITY



: Channel-wise concatenation

FORWARD PROPAGATION

COMPOSITE LAYER IN DENSENET

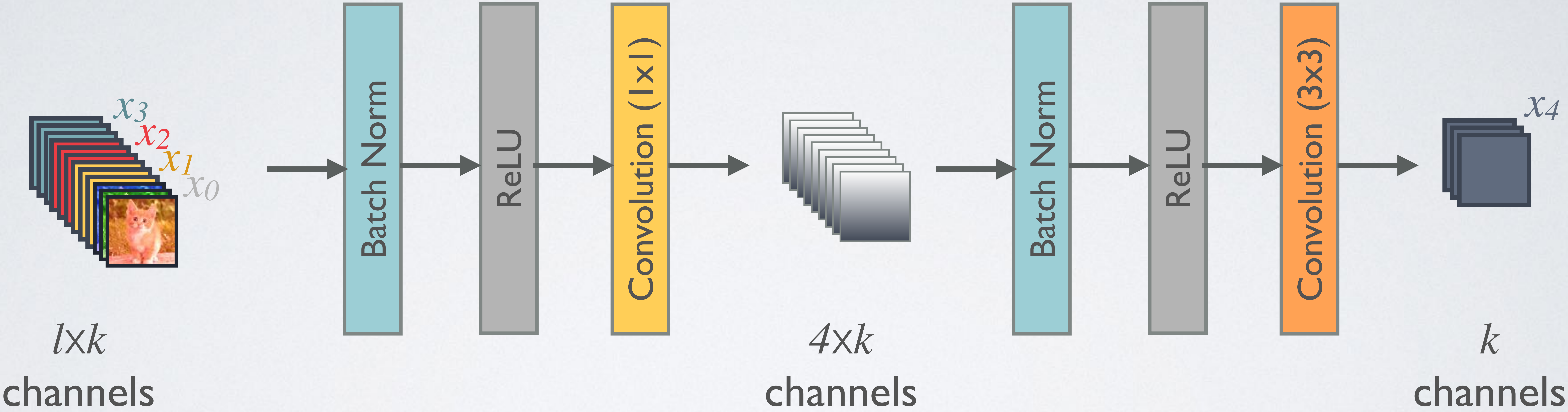


k
channels

$x_5 = h_5([x_0, \ldots, x_4])$

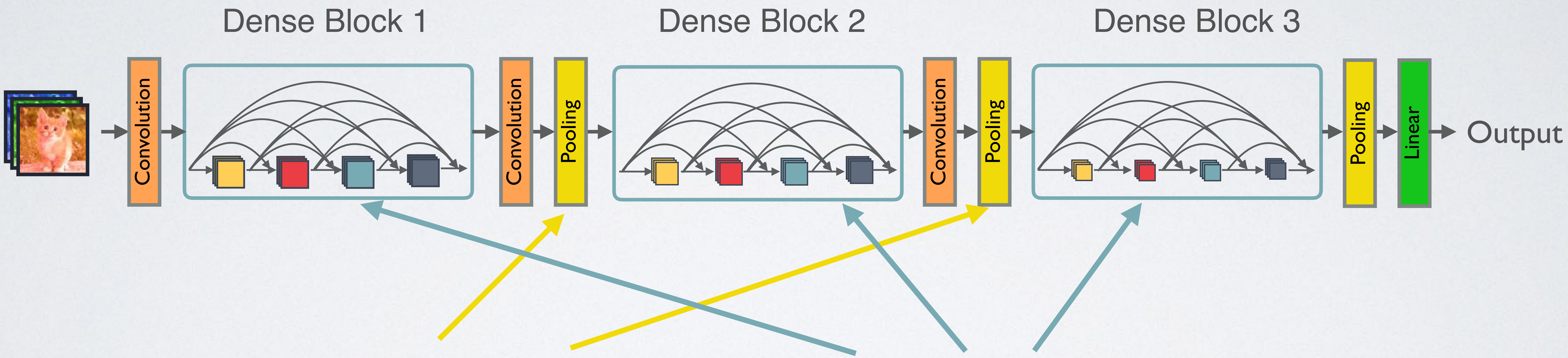
COMPOSITE LAYER IN DENSENET

WITH BOTTLENECK LAYER



Higher parameter and computational efficiency

DENSENET

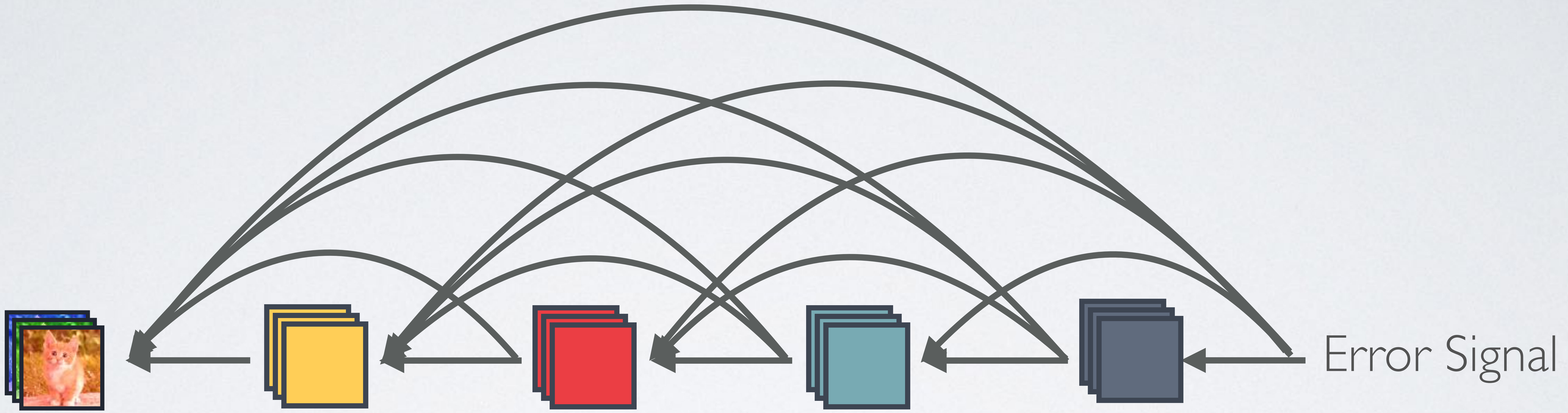


Pooling reduces feature map sizes

Feature map sizes match within each block

ADVANTAGES OF DENSE CONNECTIVITY

ADVANTAGE 1: STRONG GRADIENT FLOW

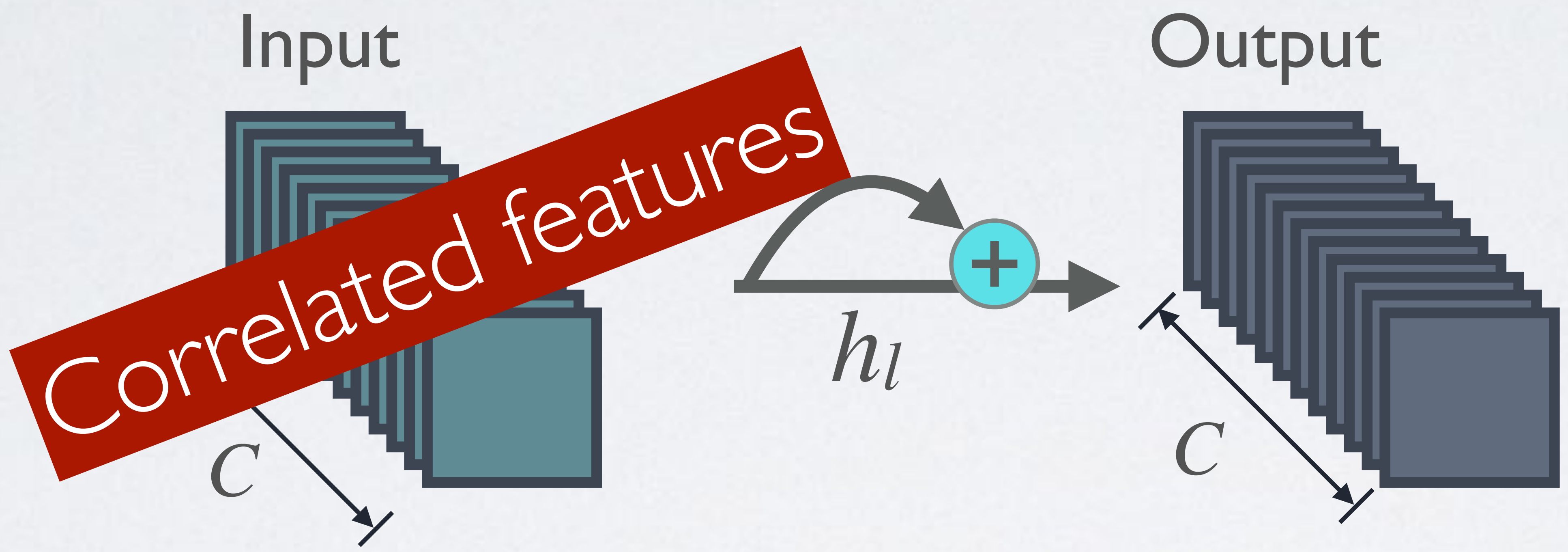


Implicit "deep supervision"

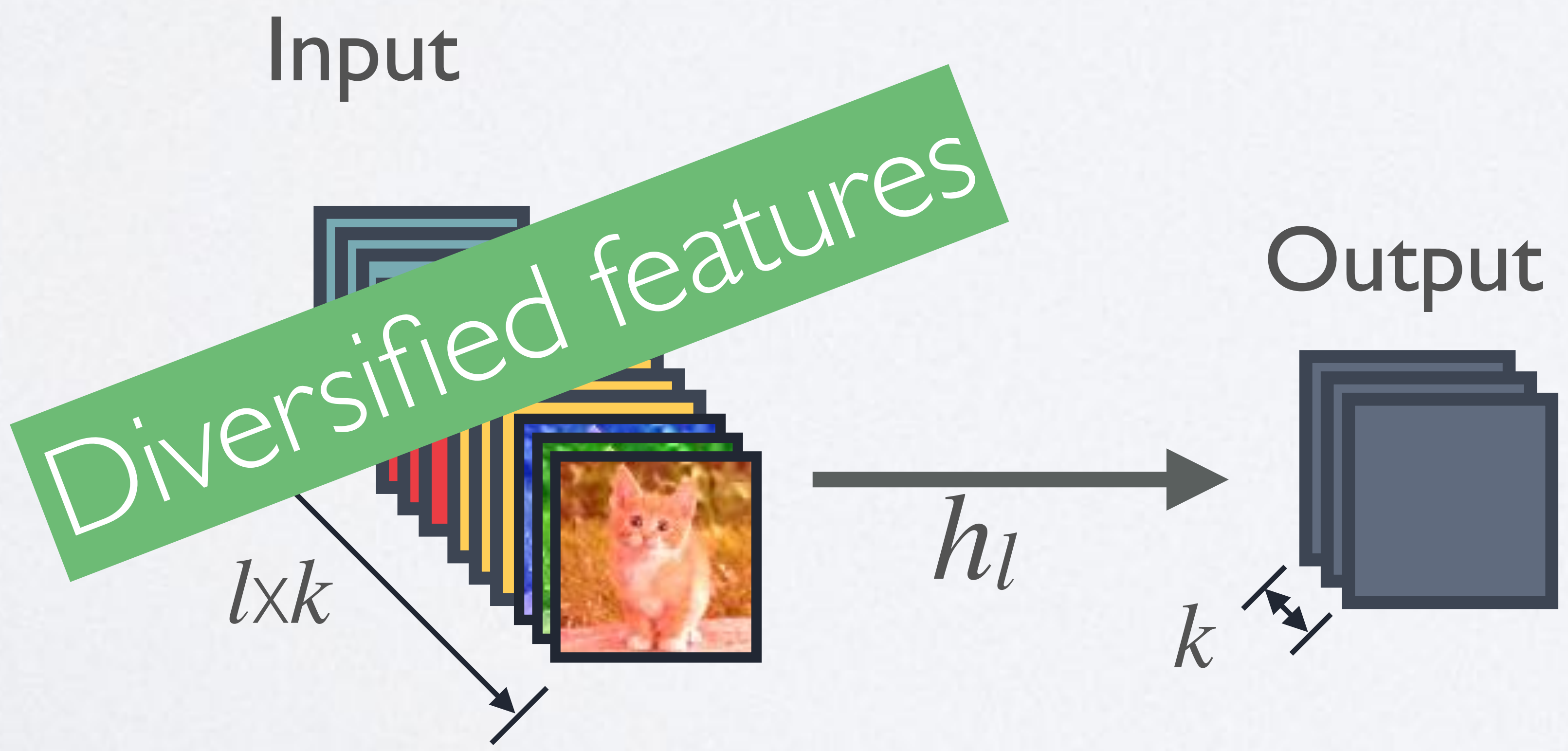
Deeply supervised Net: [Lee, Xie, Gallagher, Zhang, Tu] (2015)

ADVANTAGE 2: PARAMETER & COMPUTATIONAL EFFICIENCY

ResNet connectivity:



DenseNet connectivity:



#parameters:

$\boxed{O(C \times C)}$

$k << C$

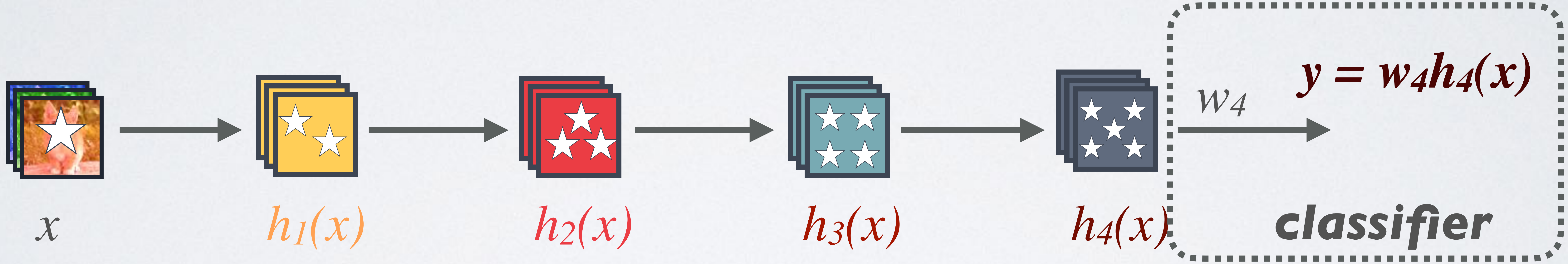
$\boxed{O(l \times k \times k)}$

k: *Growth rate*

ADVANTAGE 3: MAINTAINS LOW COMPLEXITY FEATURES

Standard Connectivity:

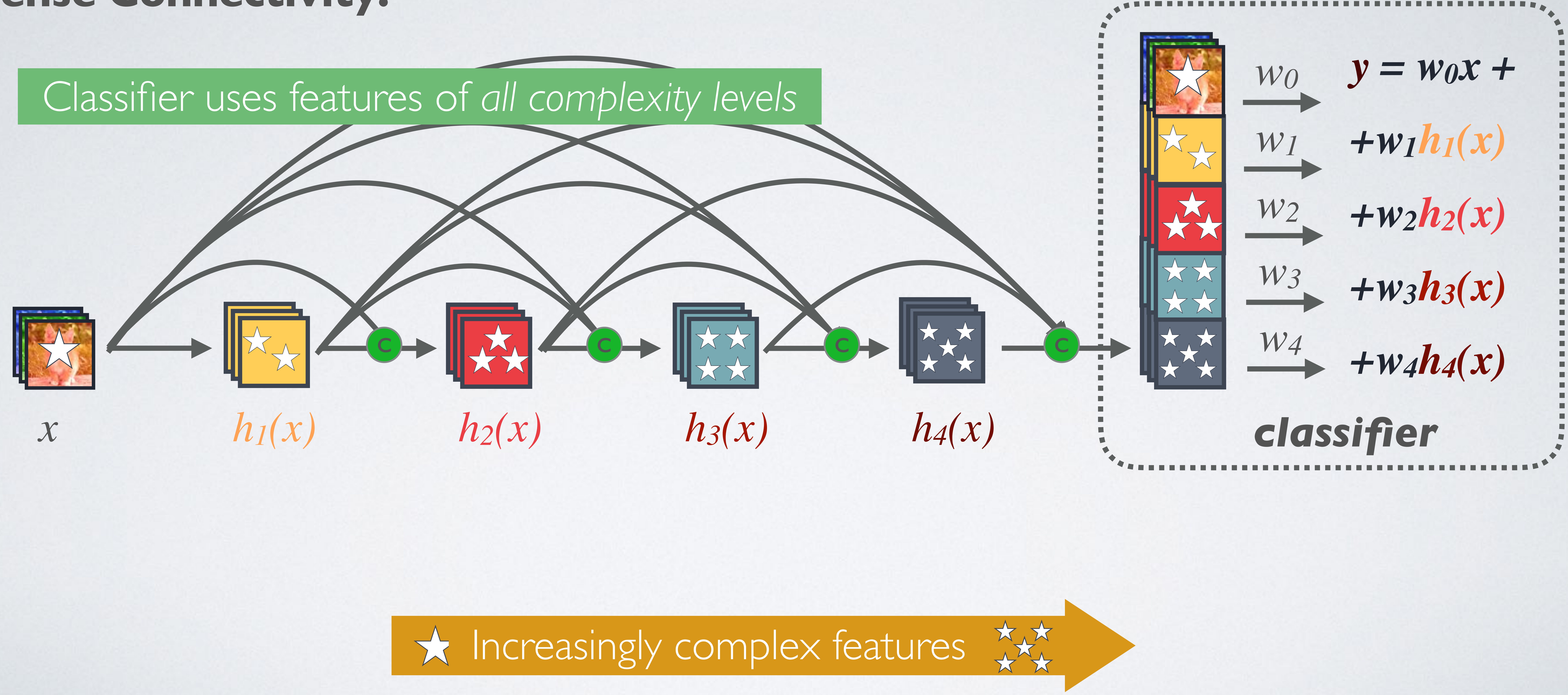
Classifier uses most complex (high level) features



Increasingly complex features

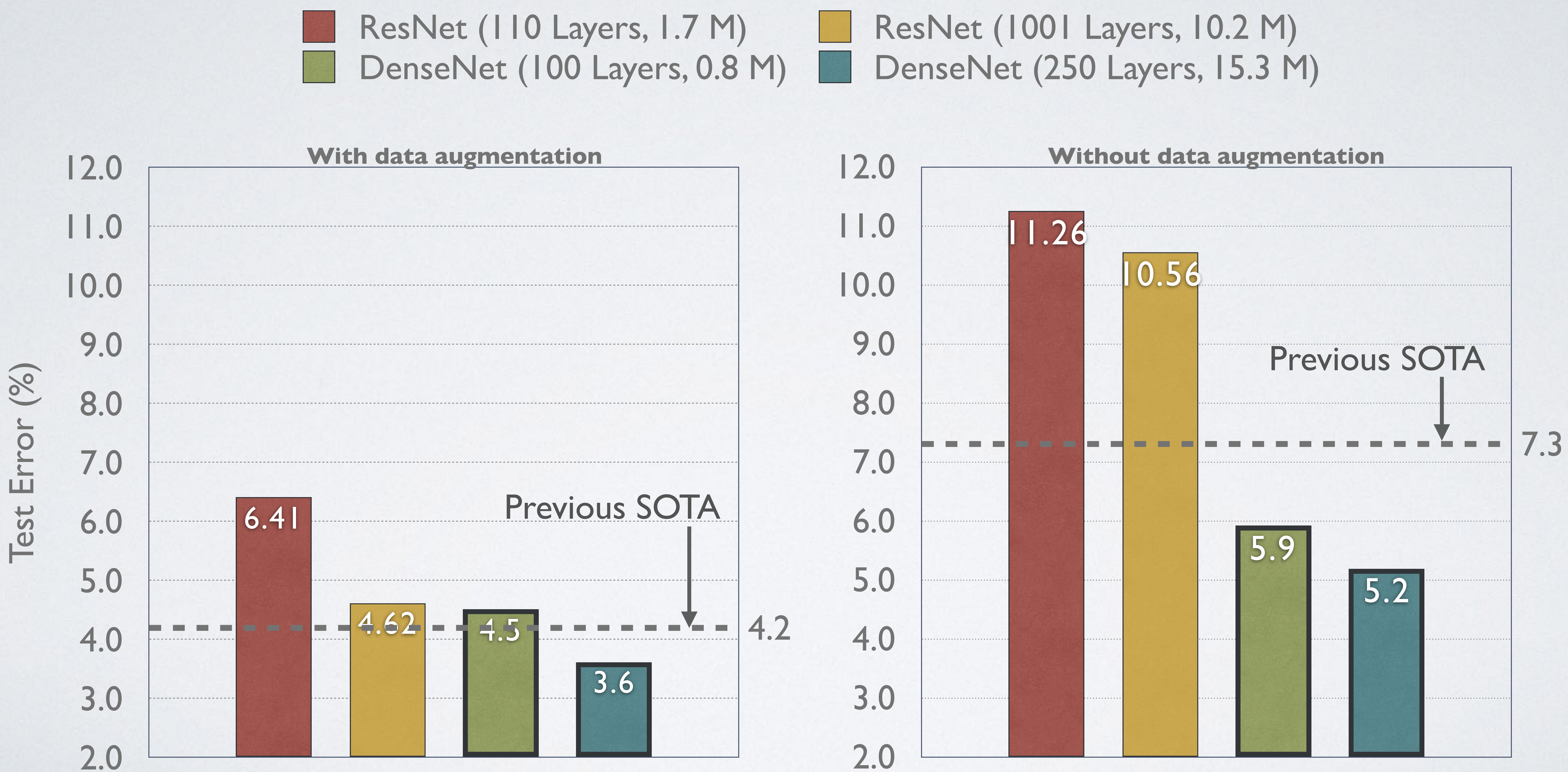
ADVANTAGE 3: MAINTAINS LOW COMPLEXITY FEATURES

Dense Connectivity:

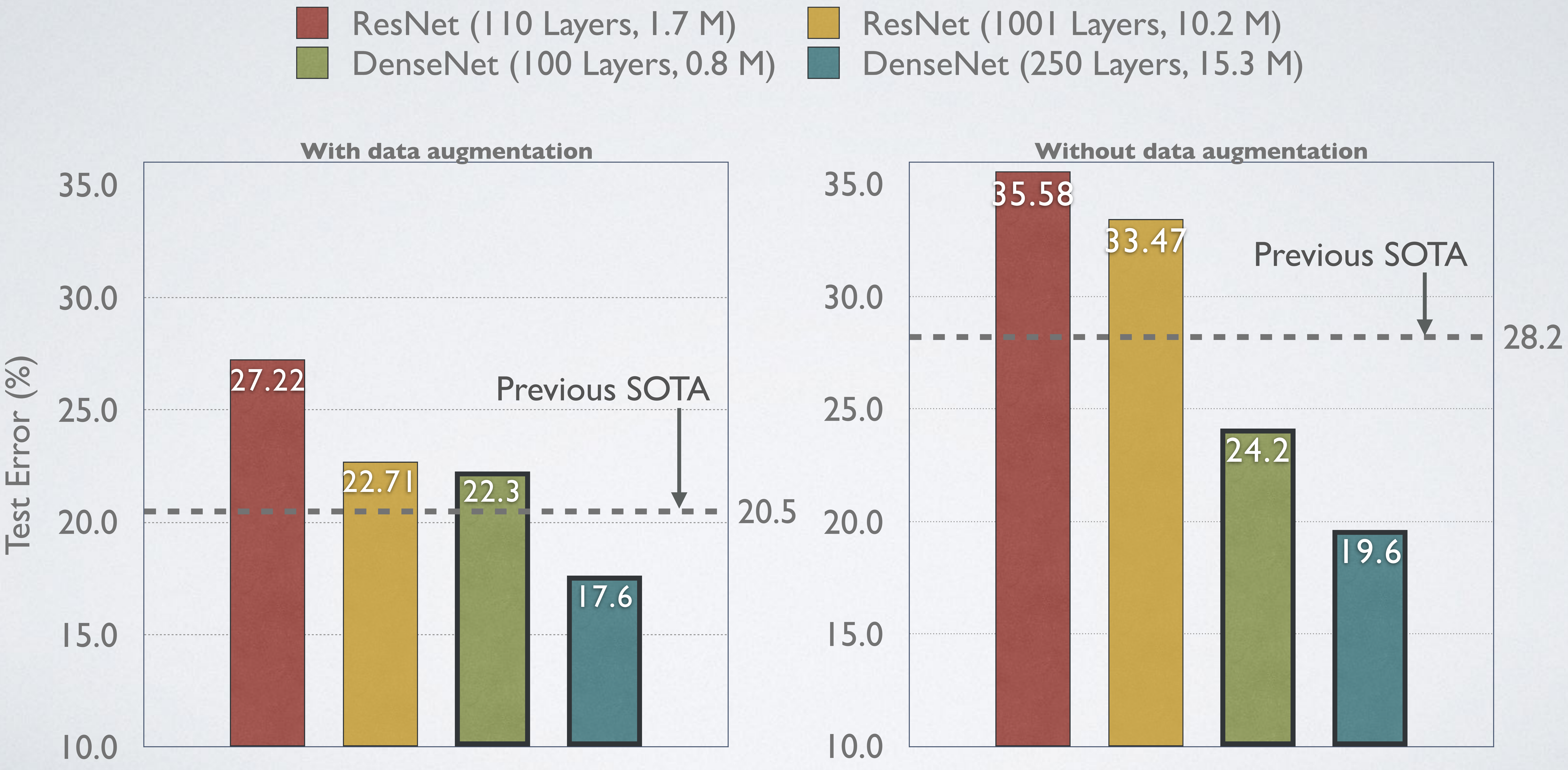


RESULTS

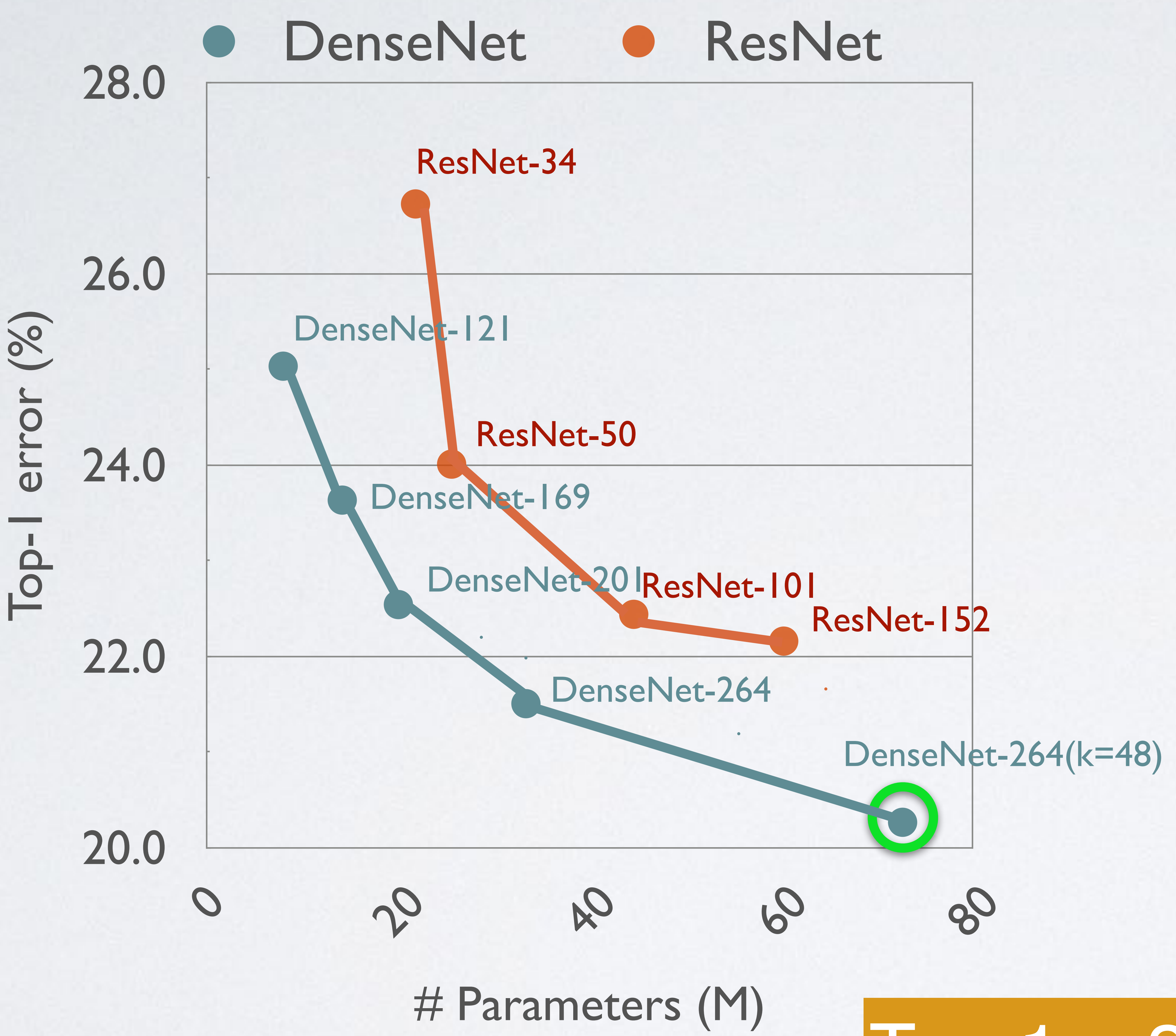
RESULTS ON **CIFAR-10**

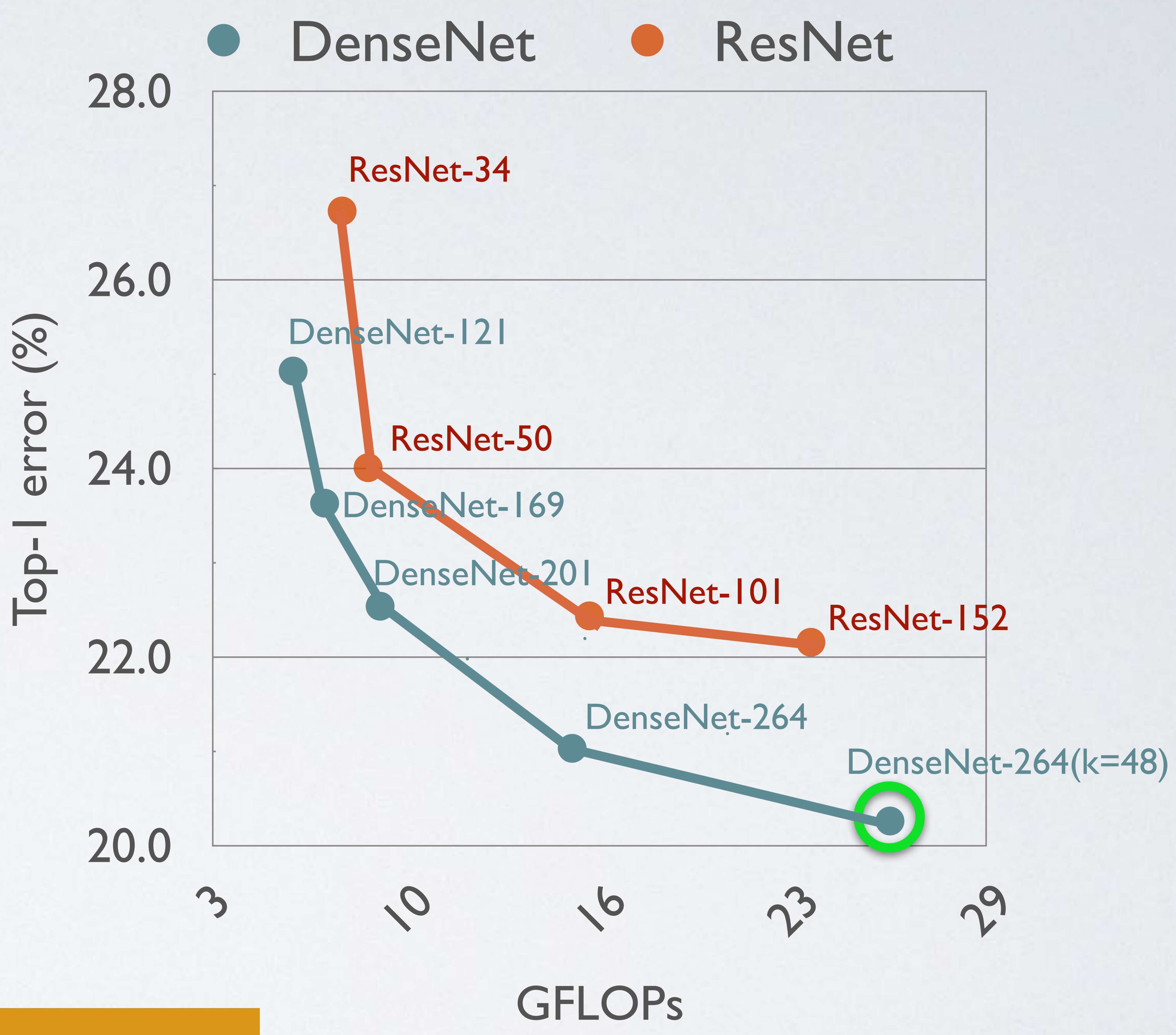


RESULTS ON **CIFAR-100**



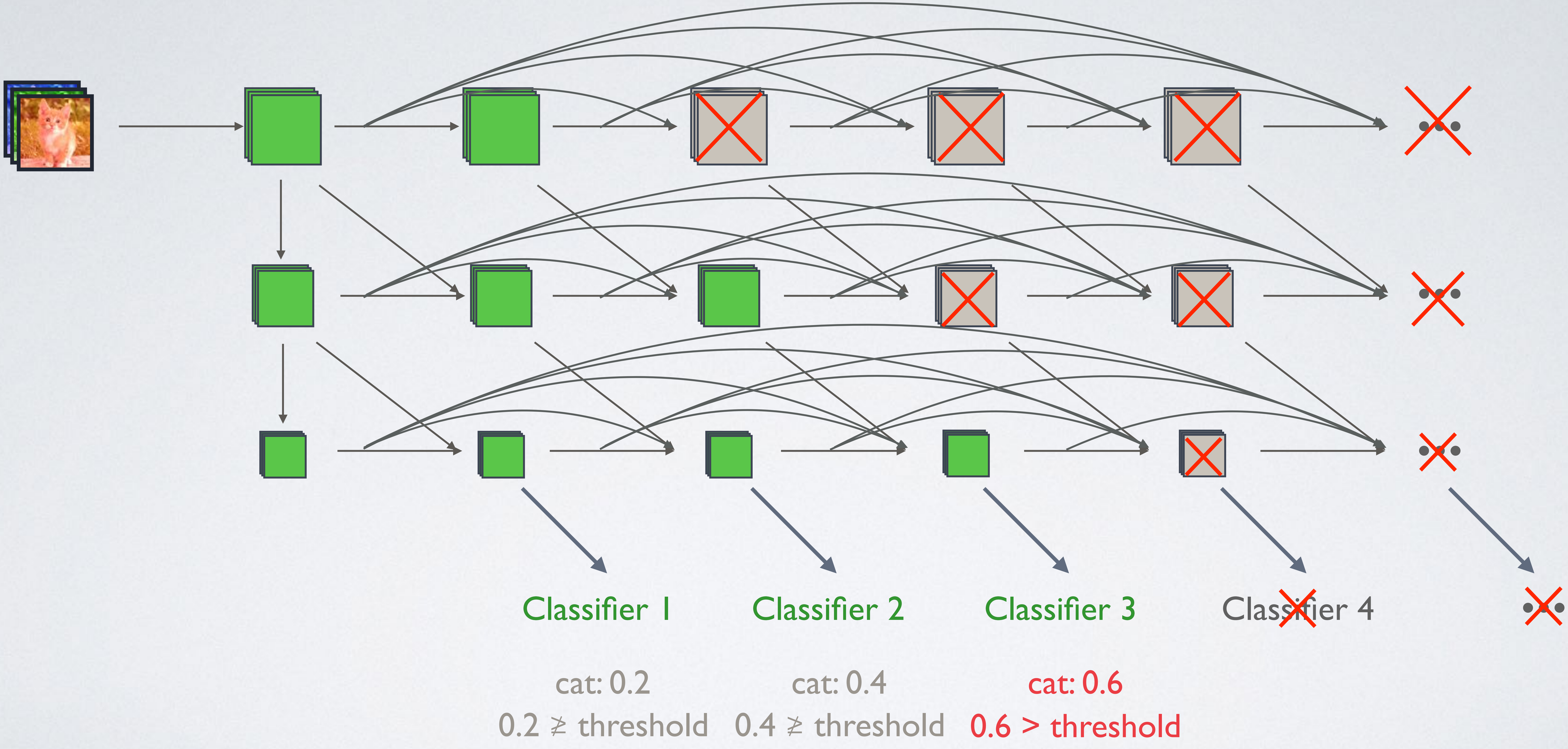
RESULTS ON **IMAGENET**





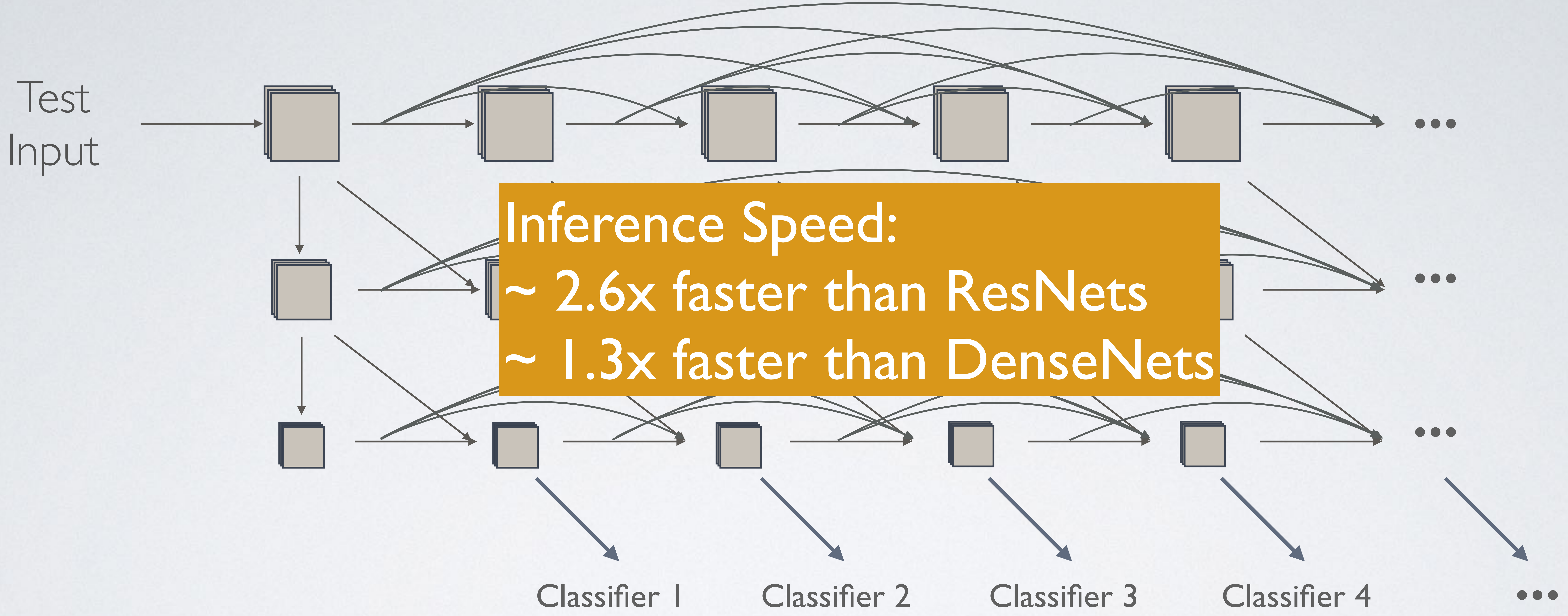
Top-1: 20.27%
Top-5: 5.17%

MULTI-SCALE DENSENET (Preview)



Multi-Scale DenseNet: [Huang, Chen, Li, Wu, van der Maaten, Weinberger] (arXiv Preprint: 1703.09844)

MULTI-SCALE DENSENET (Preview)



Inference Speed:
~ 2.6x faster than ResNets
~ 1.3x faster than DenseNets

"Easy" examples

"Hard" examples

NEW *Memory efficient Torch implementation:* NEW

https://github.com/liuzhuang13/DenseNet

Other implementations:

Our Caffe Implementation
Our memory-efficient Caffe Implementation.
Our memory-efficient PyTorch Implementation.
PyTorch Implementation by Andreas Veit.
PyTorch Implementation by Brandon Amos.
MXNet Implementation by Nicatio.
MXNet Implementation (supports ImageNet) by Xiong Lin.
Tensorflow Implementation by Yixuan Li.
Tensorflow Implementation by Laurent Mazare.
Tensorflow Implementation (with BC structure) by Illarion Khlestov.
Lasagne Implementation by Jan Schlüter.
Keras Implementation by tdeboissiere.
Keras Implementation by Roberto de Moura Estevão Filho.
Keras Implementation (with BC structure) by Somshubra Majumdar.
Chainer Implementation by Toshinori Hanya.
Chainer Implementation by Yasunori Kudo.

REFERENCES

- Kaiming He, et al. "Deep residual learning for image recognition" CVPR 2016
- Chen-Yu Lee, et al. "Deeply-supervised nets" *AISTATS* 2015
- Gao Huang, et al. "Deep networks with stochastic depth" *ECCV* 2016
- Gao Huang, et al. "Multi-Scale Dense Convolutional Networks for Efficient Prediction" *arXiv preprint arXiv:1703.09844* (2017)
- Geoff Pleiss, et al. "Memory-Efficient Implementation of DenseNets", *arXiv preprint arXiv:1707.06990* (2017)