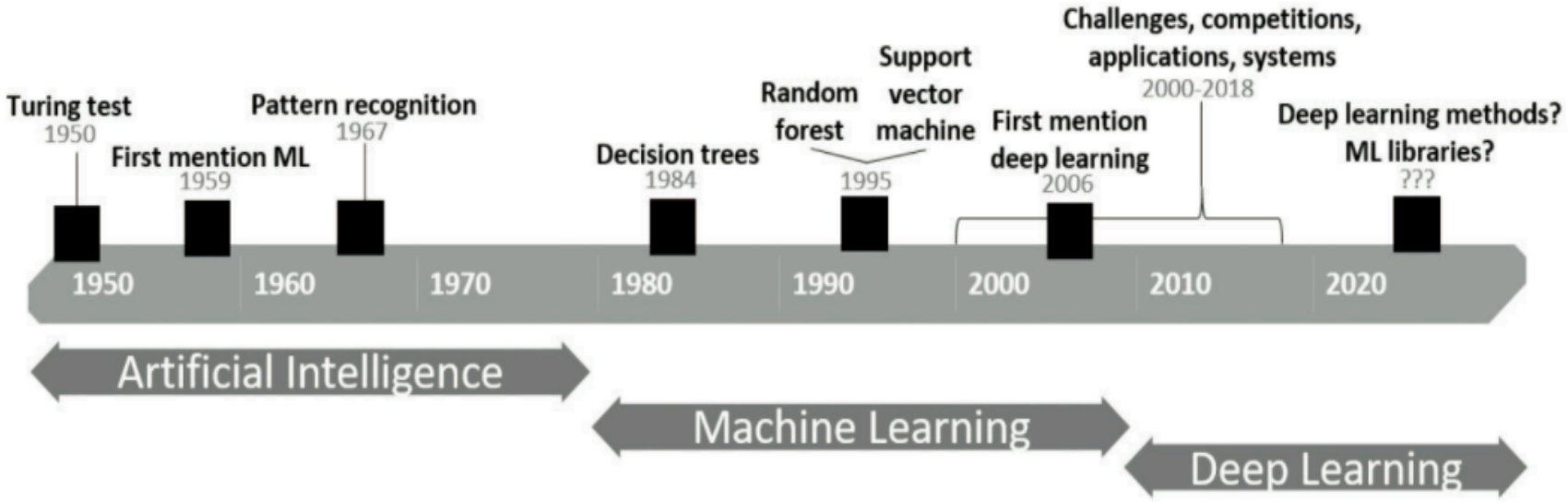
Processamento e Análise de Imagens

Deep Neural Network Architectures

Prof. Alexei Machado

PUC Minas

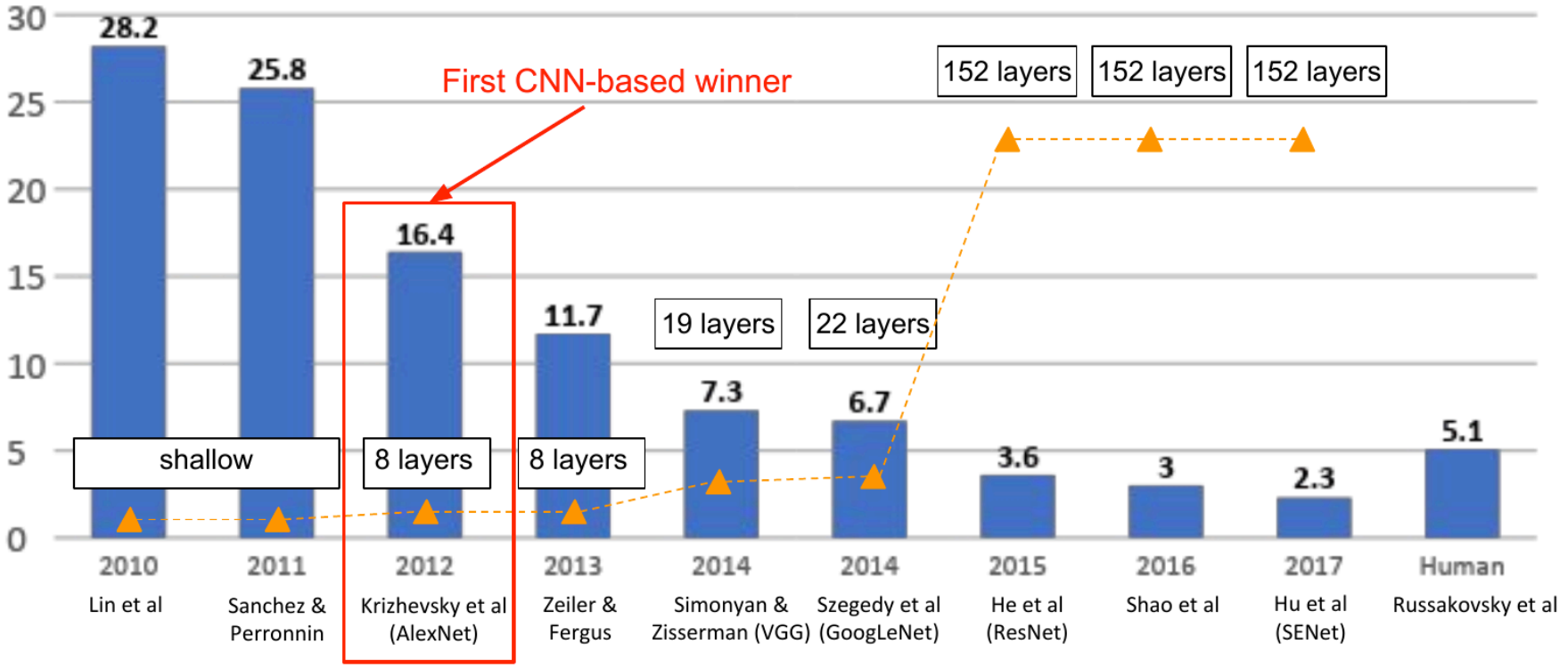
Machine Learning X Deep Learning



nekst-online.nl

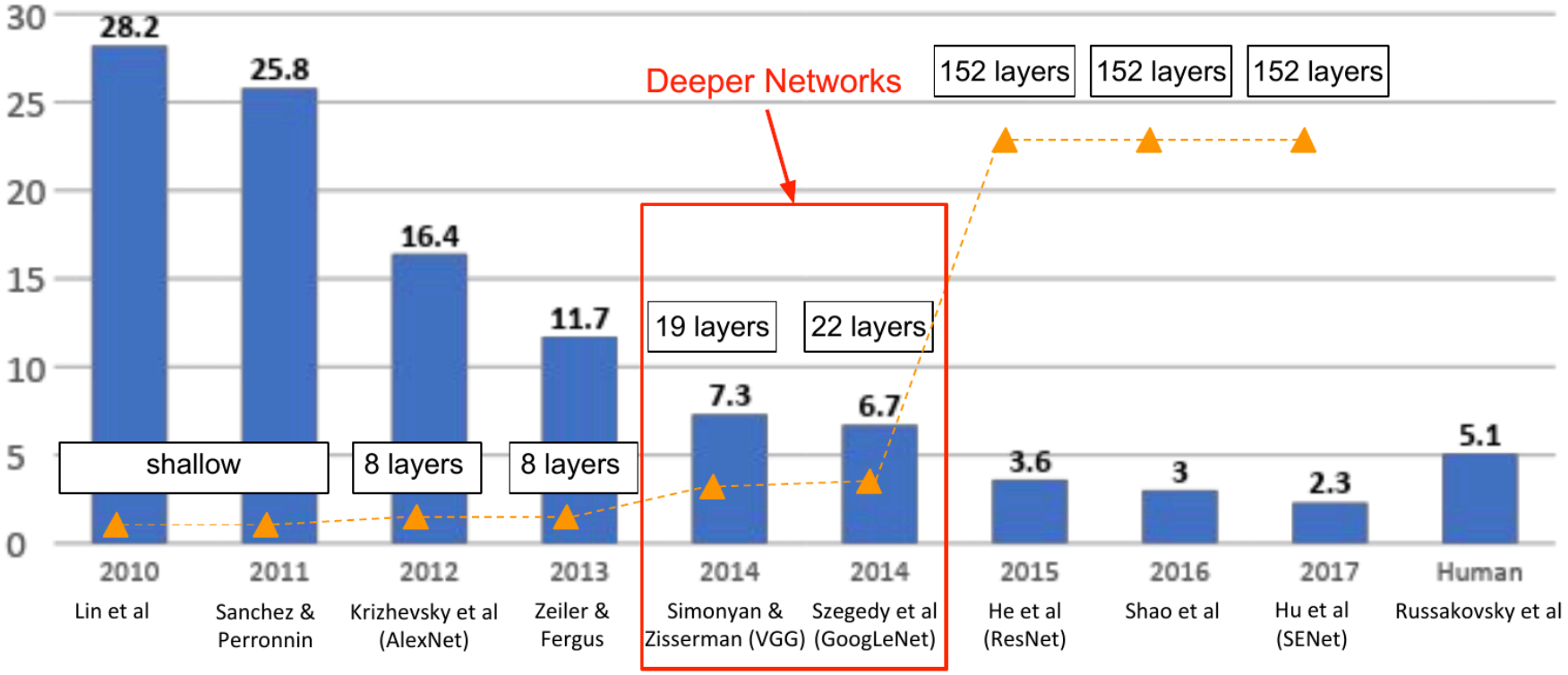
Machine Learning X Deep Learning

ImageNet Challenge



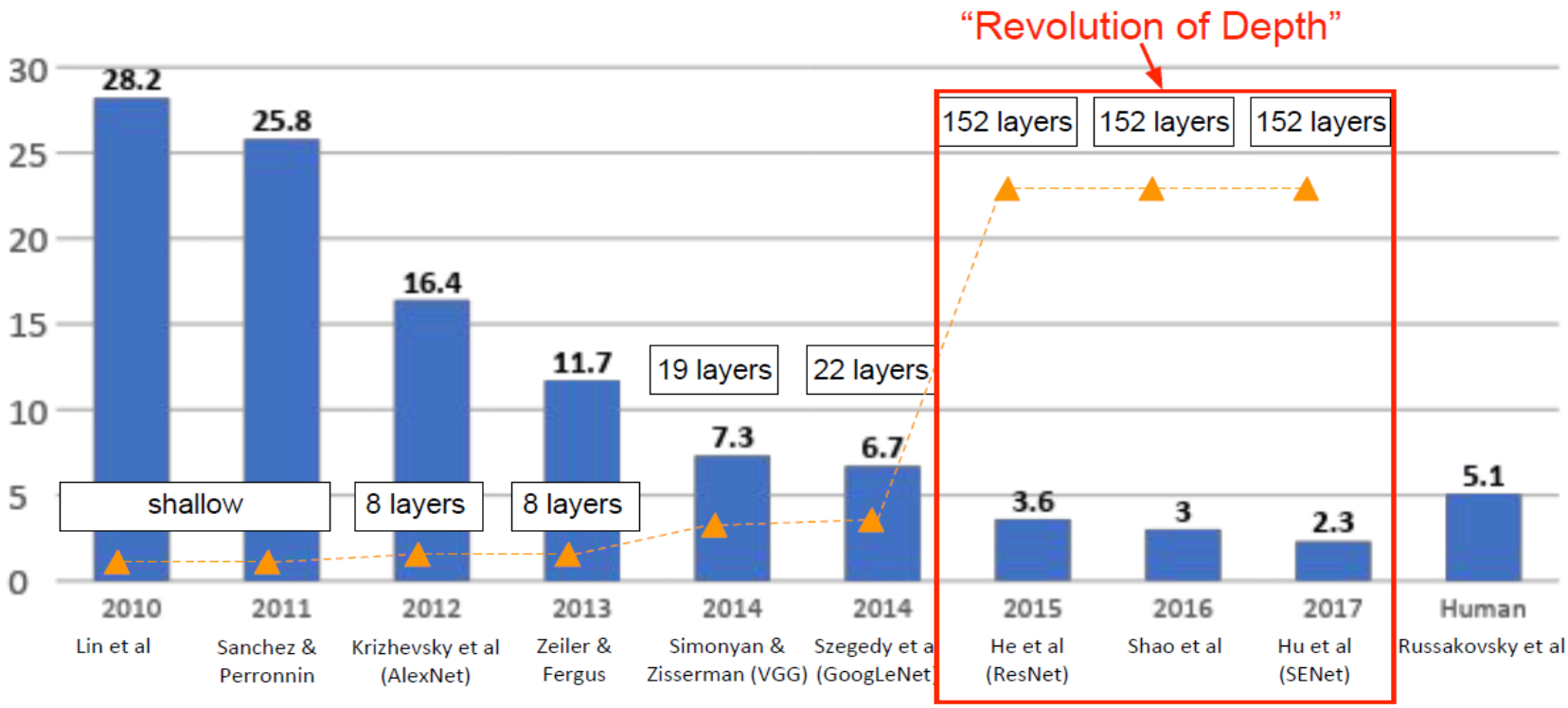
Machine Learning X Deep Learning

ImageNet Challenge

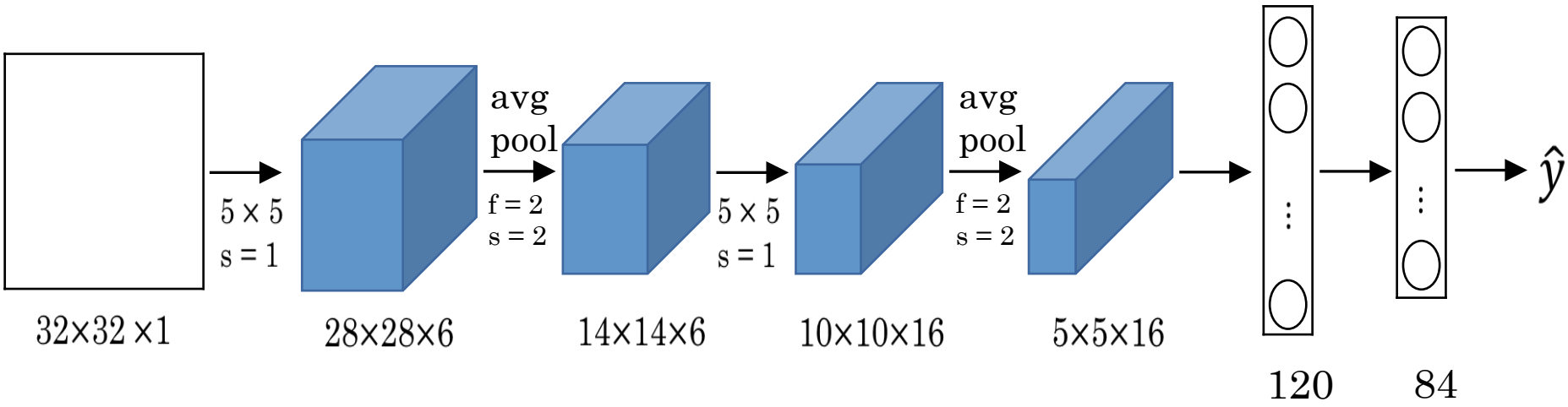


Machine Learning X Deep Learning

ImageNet Challenge



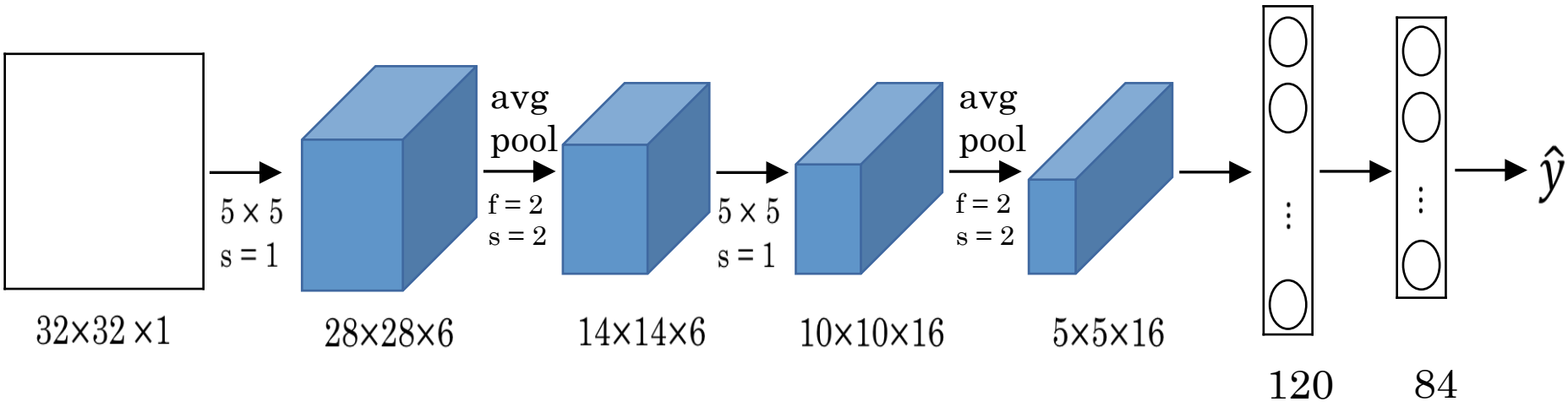
LeNet - 5



- Average pooling
- Sigmoid or tanh nonlinearity
- Fully connected layers at the end
- Trained on MNIST digit dataset with 60K training examples

[LeCun et al., 1998. Gradient-based learning applied to document recognition]

LeNet - 5



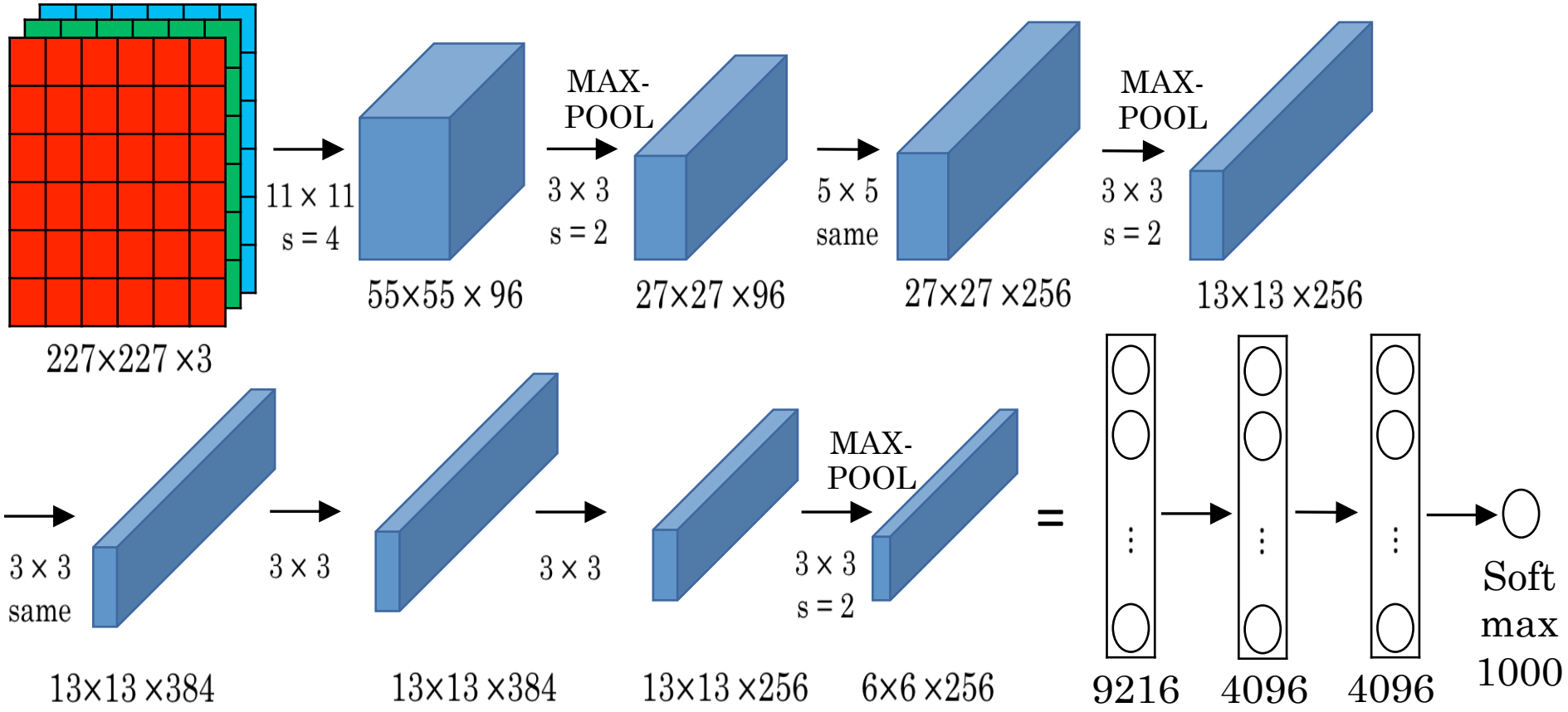
- This model was published in 1998 and the last layer was not using softmax at that time
- It has 60k parameters.
- The dimensions of the image decrease as the number of channels increases
- The modern implementation uses RELU in most cases

ImageNet Challenge

- ~14 million labeled images, 20k classes
- Images gathered from Internet
- Human labels via Amazon MTurk
- ImageNet Large-Scale Visual Recognition Challenge (ILSVRC):
  1.2 million training images, 1000 classes

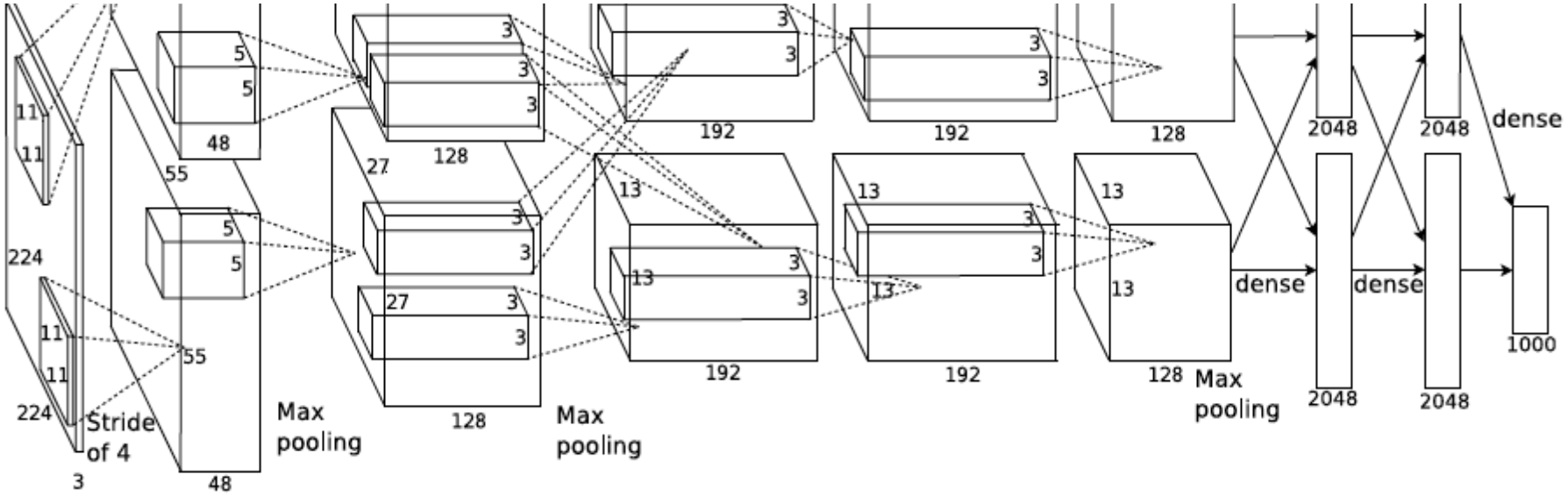
www.image-net.org/challenges/LSVRC/

AlexNet



[Krizhevsky et al., 2012. ImageNet classification with deep convolutional neural networks]

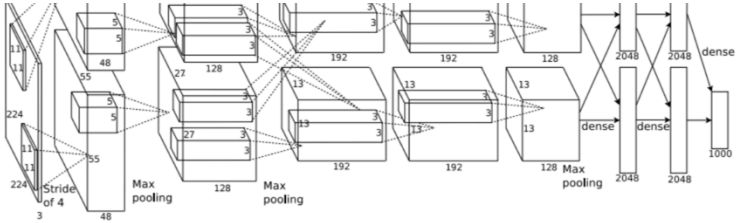
AlexNet



- ILSVRC 2012 winner
- Similar framework to LeNet but:
  - Max pooling, ReLU nonlinearity
  - More data and bigger model (7 hidden layers, 650K units, 60M params)
  - GPU implementation (50x speedup over CPU)
  - Trained on two GPUs for a week
  - Dropout regularization

AlexNet

AlexNet



	Input size			Layer				Output size	
Layer	C	H / W		filters	kernel	stride	pad	C	H / W
conv1	3	227		64	11	4	2	64	56

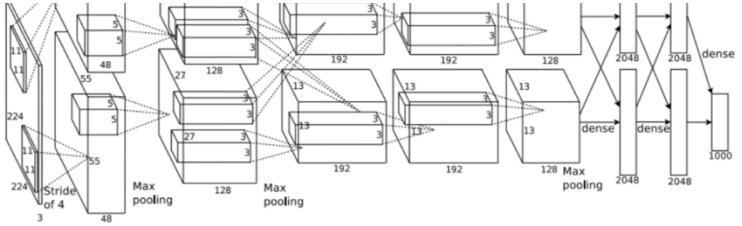
Recall: W' = (W – K + 2P) / S + 1

= 227 – 11 + 2*2) / 4 + 1

= 220/4 + 1 = 56

AlexNet

AlexNet



	Input size			Layer				Output size		
Layer	C	H / W		filters	kernel	stride	pad	C	H / W	memory (KB)
conv1	3	227		64	11	4	2	64	56	784

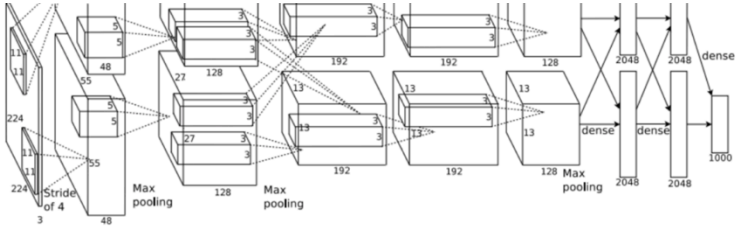
Number of output elements = C * H' * W'
= 64*56*56 = 200,704

Bytes per element = 4 (for 32-bit floating point)

KB = (number of elements) * (bytes per elem) / 1024
= 200704 * 4 / 1024
= **784**

AlexNet

AlexNet



	Input size			Layer				Output size				
Layer	C	H / W		filters	kernel	stride	pad	C	H / W		memory (KB)	params (k)
conv1	3	227		64	11	4	2	64	56		784	23

Weight shape = C_{out} x C_{in} x K x K

= 64 x 3 x 11 x 11

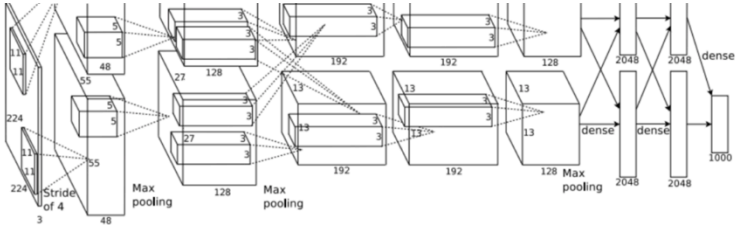
Bias shape = C_{out} = 64

Number of weights = 64*3*11*11 + 64

= **23,296**

AlexNet

AlexNet

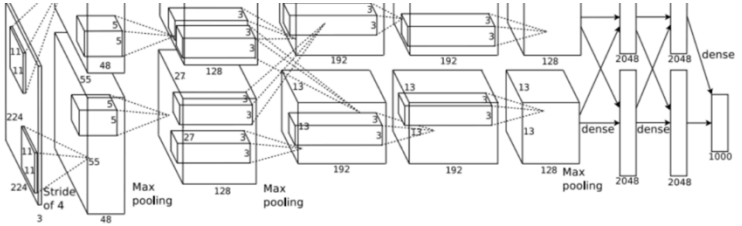


	Input size			Layer				Output size					
Layer	C	H / W		filters	kernel	stride	pad	C	H / W		memory (KB)	params (k)	flop (M)
conv1	3	227		64	11	4	2	64	56		784	23	73

Number of floating point operations (multiply+add)
= (number of output elements) * (ops per output elem)
= (C_{out} x H' x W') * (C_{in} x K x K)
= (64 * 56 * 56) * (3 * 11 * 11)
= 200,704 * 363
= **72,855,552**

AlexNet

AlexNet



	Input size			Layer				Output size				
Layer	C	H / W		filters	kernel	stride	pad	C	H / W	memory (KB)	params (k)	flop (M)
conv1	3	227		64	11	4	2	64	56	784	23	73
pool1	64	56			3	2	0	64	27			

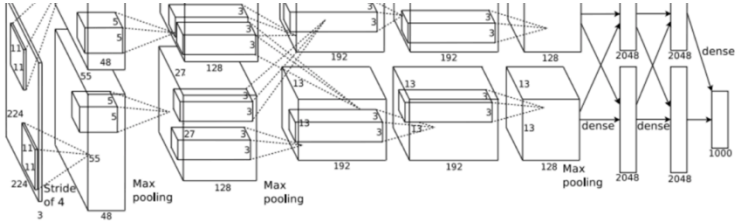
For pooling layer:

#output channels = #input channels = 64

W' = floor((W – K) / S + 1)
= floor(53 / 2 + 1) = floor(27.5) = **27**

AlexNet

AlexNet

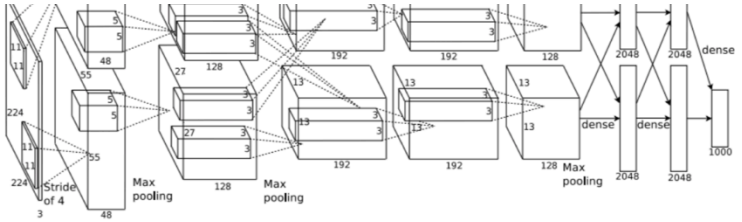


	Input size			Layer				Output size					
Layer	C	H / W		filters	kernel	stride	pad	C	H / W		memory (KB)	params (k)	flop (M)
conv1	3	227		64	11	4	2	64	56		784	23	73
pool1	64	56			3	2	0	64	27		182	?	

#output elems = C_{out} x H' x W'
Bytes per elem = 4
KB = C_{out} * H' * W' * 4 / 1024
= 64 * 27 * 27 * 4 / 1024
= **182.25**

AlexNet

AlexNet

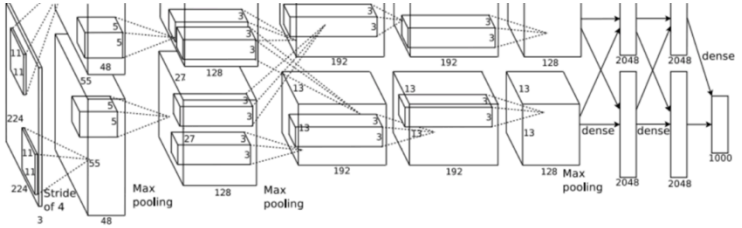


	Input size			Layer				Output size				
Layer	C	H / W	filters	kernel	stride	pad	C	H / W	memory (KB)	params (k)	flop (M)
conv1	3	227	64	11	4	2	64	56	784	23	73
pool1	64	56		3	2	0	64	27	182	0	?

Pooling layers have no learnable parameters!

AlexNet

AlexNet

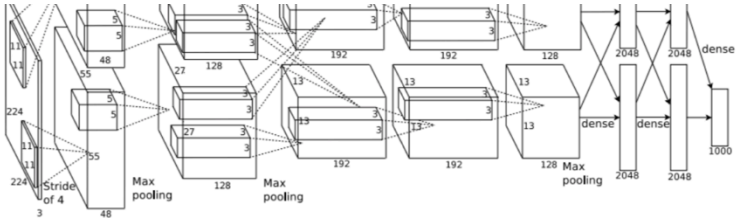


	Input size			Layer				Output size				
Layer	C	H / W	filters	kernel	stride	pad	C	H / W	memory (KB)	params (k)	flop (M)
conv1	3	227	64	11	4	2	64	56	784	23	73
pool1	64	56		3	2	0	64	27	182	0	0

Floating-point ops for pooling layer
= (number of output positions) * (flops per output position)
= (C_{out} * H' * W') * (K * K)
= (64 * 27 * 27) * (3 * 3)
= 419,904
= **0.4 MFLOP**

AlexNet

AlexNet

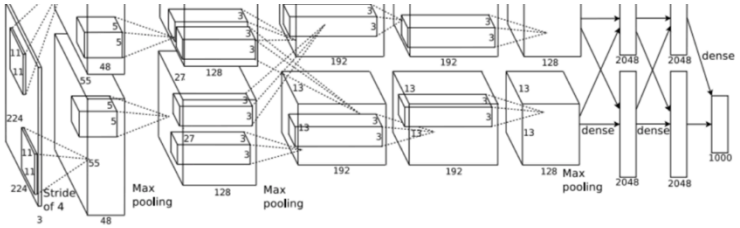


	Input size			Layer				Output size				
Layer	C	H / W	filters	kernel	stride	pad	C	H / W	memory (KB)	params (k)	flop (M)
conv1	3	227	64	11	4	2	64	56	784	23	73
pool1	64	56		3	2	0	64	27	182	0	0
conv2	64	27	192	5	1	2	192	27	547	307	224
pool2	192	27		3	2	0	192	13	127	0	0
conv3	192	13	384	3	1	1	384	13	254	664	112
conv4	384	13	256	3	1	1	256	13	169	885	145
conv5	256	13	256	3	1	1	256	13	169	590	100
pool5	256	13		3	2	0	256	6	36	0	0
flatten	256	6					9216		36	0	0

Flatten output size = C_{in} x H x W
= 256 * 6 * 6
= **9216**

AlexNet

AlexNet



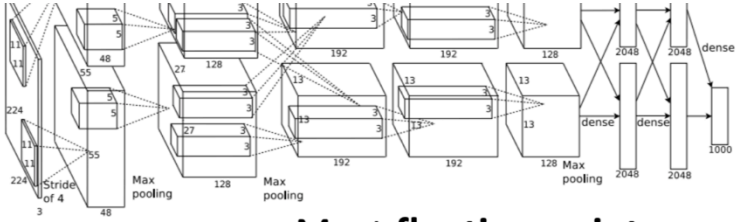
	Input size			Layer				Output size				
Layer	C	H / W	filters	kernel	stride	pad	C	H / W	memory (KB)	params (k)	flop (M)
conv1	3	227	64	11	4	2	64	56	784	23	73
pool1	64	56		3	2	0	64	27	182	0	0
conv2	64	27	192	5	1	2	192	27	547	307	224
pool2	192	27		3	2	0	192	13	127	0	0
conv3	192	13	384	3	1	1	384	13	254	664	112
conv4	384	13	256	3	1	1	256	13	169	885	145
conv5	256	13	256	3	1	1	256	13	169	590	100
pool5	256	13		3	2	0	256	6	36	0	0
flatten	256	6					9216		36	0	0
fc6	9216		4096				4096		16	37,749	38

FC params = $C_{in} * C_{out} + C_{out}$
= 9216 * 4096 + 4096
= 37,725,832

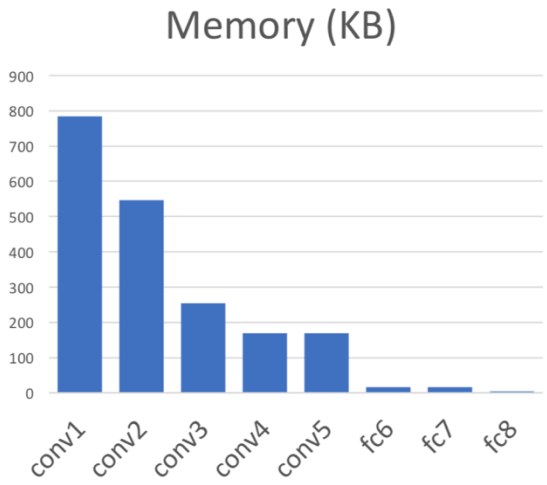
FC flops = $C_{in} * C_{out}$
= 9216 * 4096
= 37,748,736

AlexNet

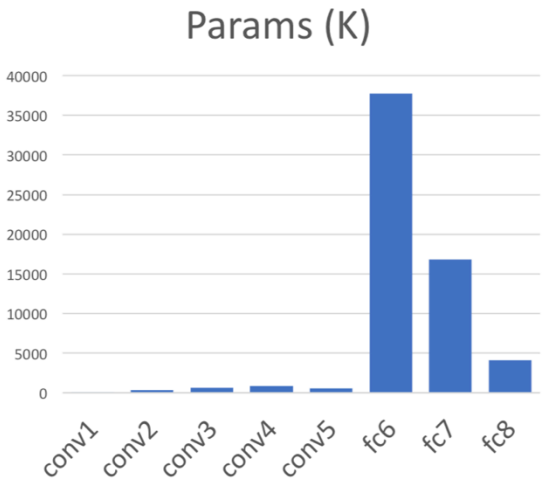
AlexNet



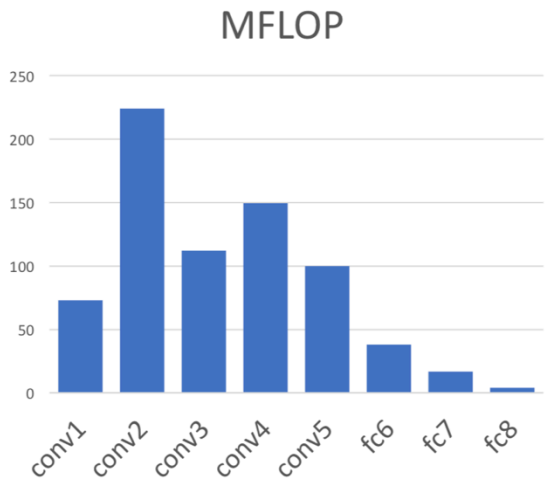
Most of the **memory usage** is in the early convolution layers



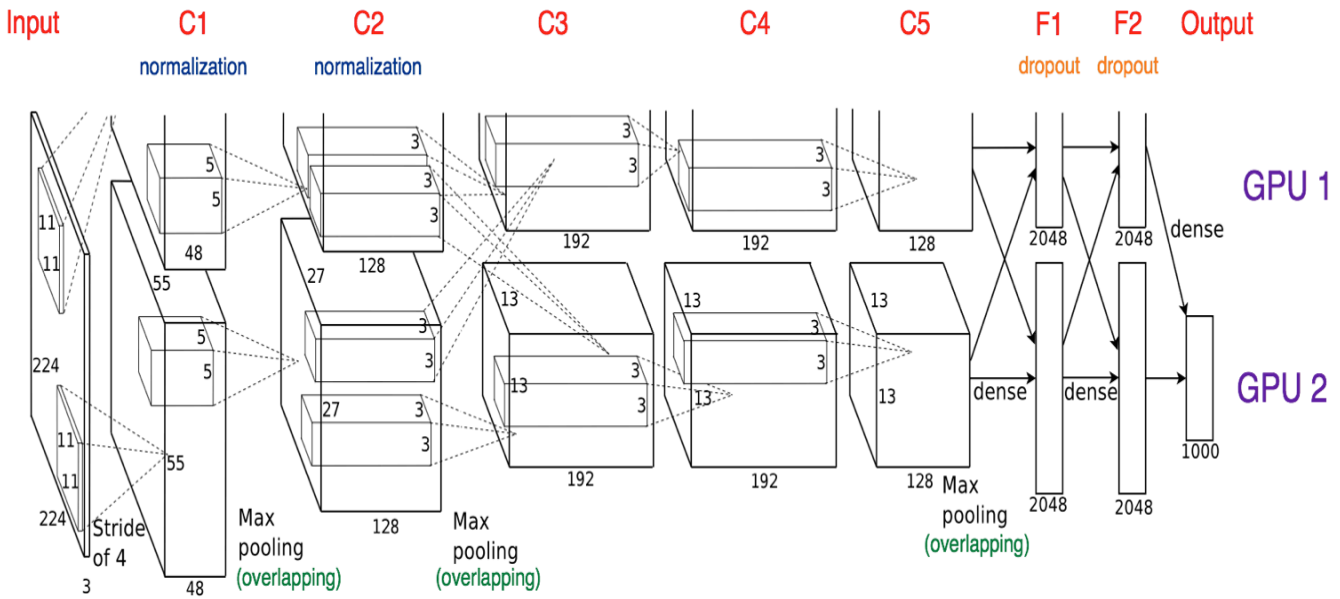
Nearly all **parameters** are in the fully-connected layers



Most **floating-point ops** occur in the convolution layers



AlexNet



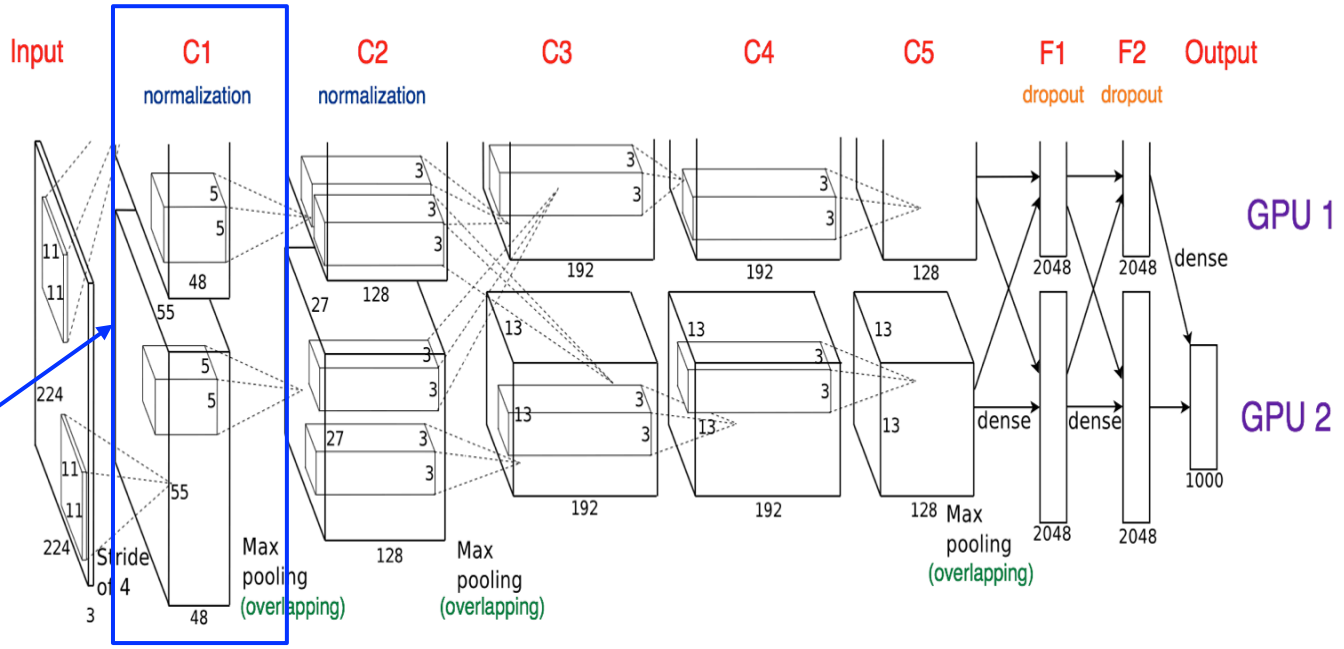
Complete architecture:

[227x227x3] Input
[55x55x96]CONV1: 96 **filters** 11x11, ***stride*** 4, ***pad*** 0
[27x27x96]MAX POOL1: 3x3 **filters**, ***stride*** 2
[27x27x96]NORM1: normalization layer
[27x27x256]CONV2: 256 **filters** 5x5, ***stride*** 1, ***pad*** 2
[13x13x256]MAX POOL2: 3x3 **filters**, ***stride*** 2
[13x13x256]NORM2: normalization layer
[13x13x384]CONV3: 384 **filters** 3x3, ***stride*** 1, ***pad*** 1
[13x13x384]CONV4: 384 **filters** 3x3, ***stride*** 1, ***pad*** 1
[13x13x256]CONV5: 256 **filters** 3x3, ***stride*** 1, ***pad*** 1
[6x6x256]MAX POOL3: 3x3 **filters**, ***stride*** 2
[4096]FC6: 4096 neurons
[4096]FC7: 4096 neurons
[1000]FC8: 1000 neurons (class values)

Remarks:

- First use of ReLU
- NORM layers (not used anymore)
- Dropout 0.5
- Batch size: 128
- SGD Momentum 0.9
- Learning rate of 0.01, manually reduced after learning curve gets stable
- L2 weight decay 0.0005

AlexNet

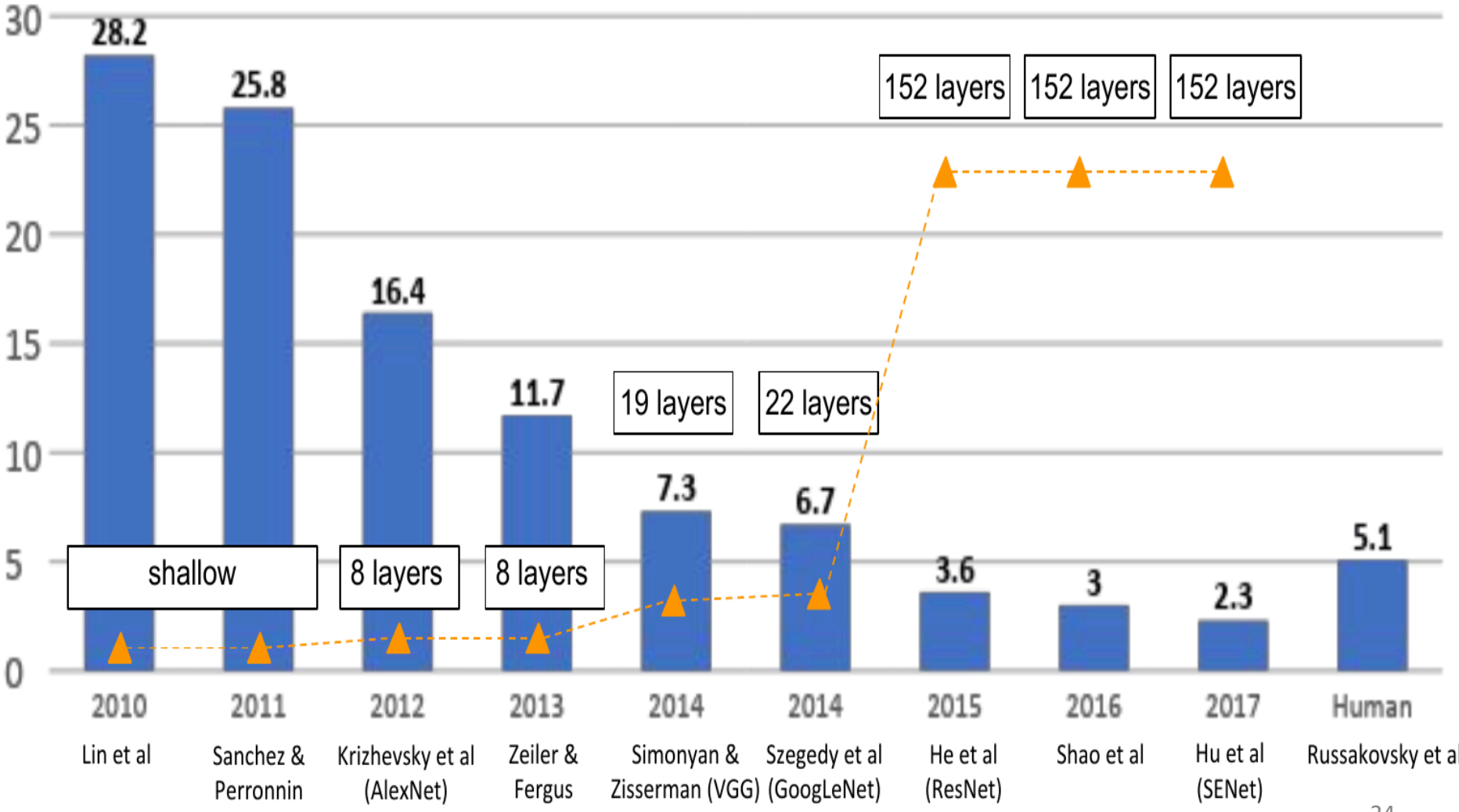


Complete architecture:

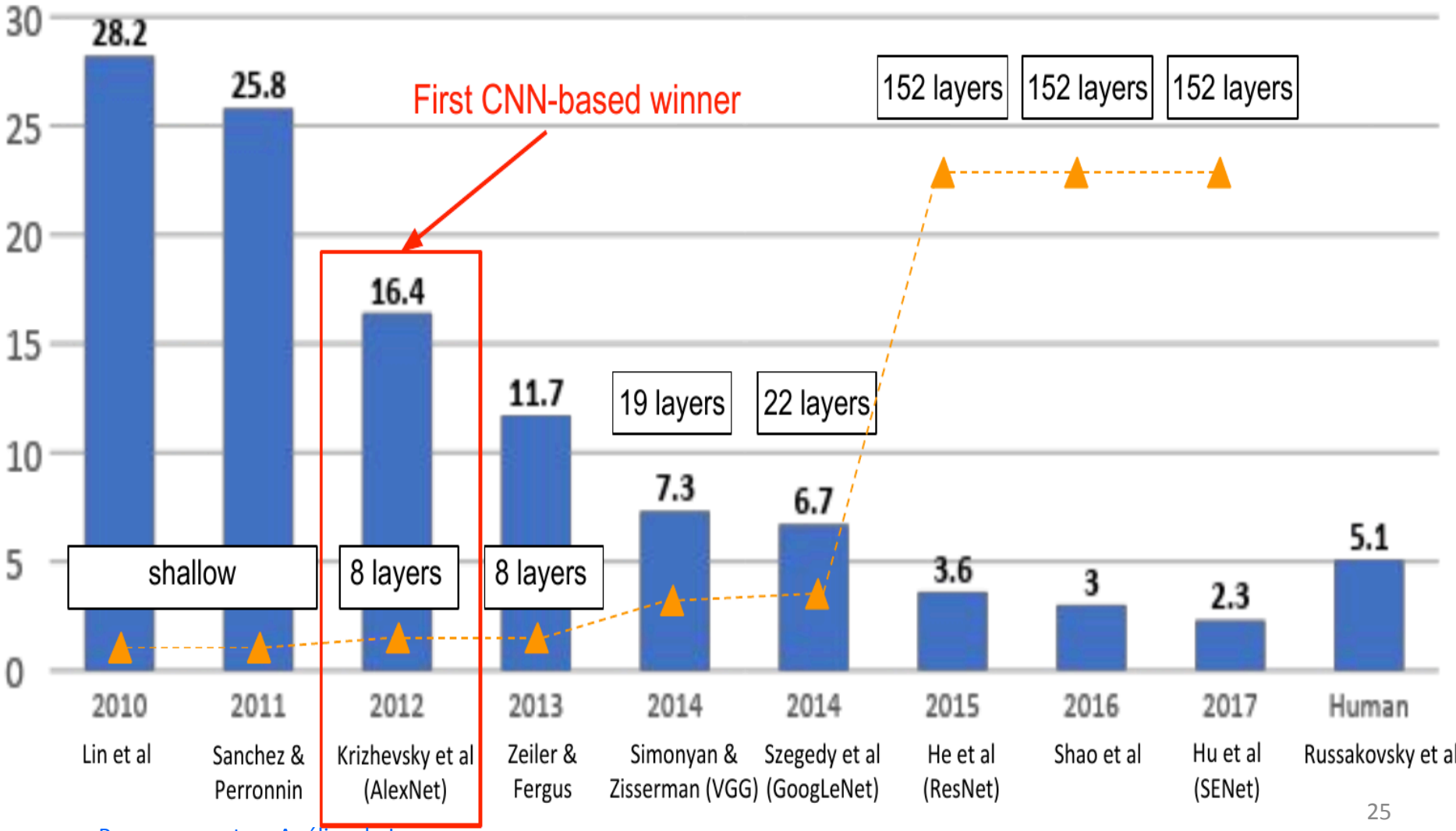
[227x227x3] Input
[55x55x96]CONV1: 96 **filters** 11x11, ***stride*** 4, ***pad*** 0
[27x27x96]MAX POOL1: 3x3 **filters**, ***stride*** 2
[27x27x96]NORM1: normalization layer
[27x27x256]CONV2: 256 **filters** 5x5, ***stride*** 1, ***pad*** 2
[13x13x256]MAX POOL2: 3x3 **filters**, ***stride*** 2
[13x13x256]NORM2: normalization layer
[13x13x384]CONV3: 384 **filters** 3x3, ***stride*** 1, ***pad*** 1
[13x13x384]CONV4: 384 **filters** 3x3, ***stride*** 1, ***pad*** 1
[13x13x256]CONV5: 256 **filters** 3x3, ***stride*** 1, ***pad*** 1
[6x6x256]MAX POOL3: 3x3 **filters**, ***stride*** 2
[4096]FC6: 4096 neurons
[4096]FC7: 4096 neurons
[1000]FC8: 1000 neurons (class values)

Remarks: trained on a GTX 580 GPU with only 3GB of RAM. As the memory was insufficient, 2 GPUs were needed, half of the neurons (feature map) in each GPU.

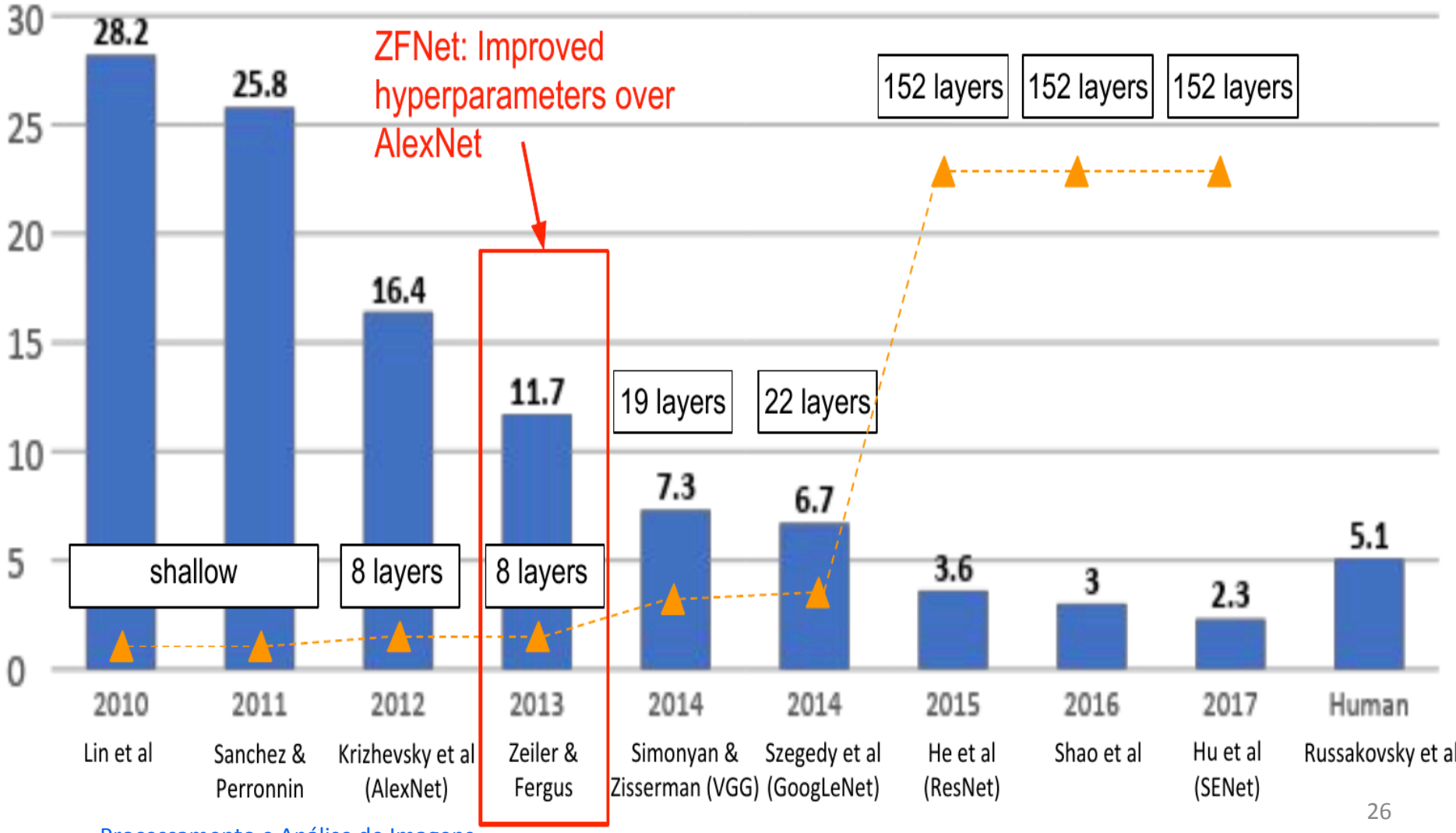
ImageNet Challenge (winners)



ImageNet Challenge (winners)



ImageNet Challenge (winners)



ZFNet

- Refinement of AlexNet
- Still trial and error
- ILSVRC 2013 winner
- CONV1: changed filters (11x11 stride 4) to (7x7 stride 2)
- CONV3,4,5: instead of 384, 384, 256 filters, they used 512, 1024, 512
- ImageNet top 5 error: 16.4% -> 11.7%

ZFNet

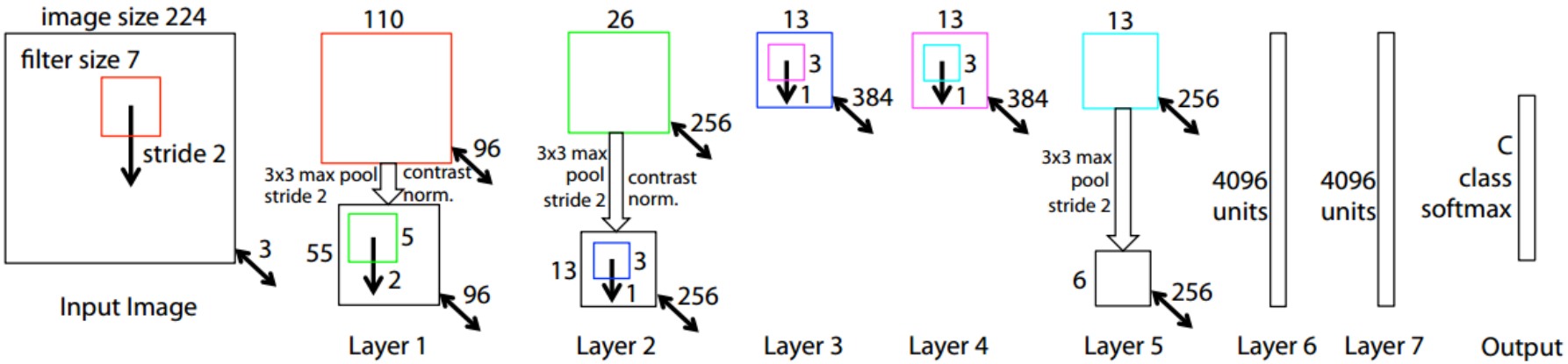
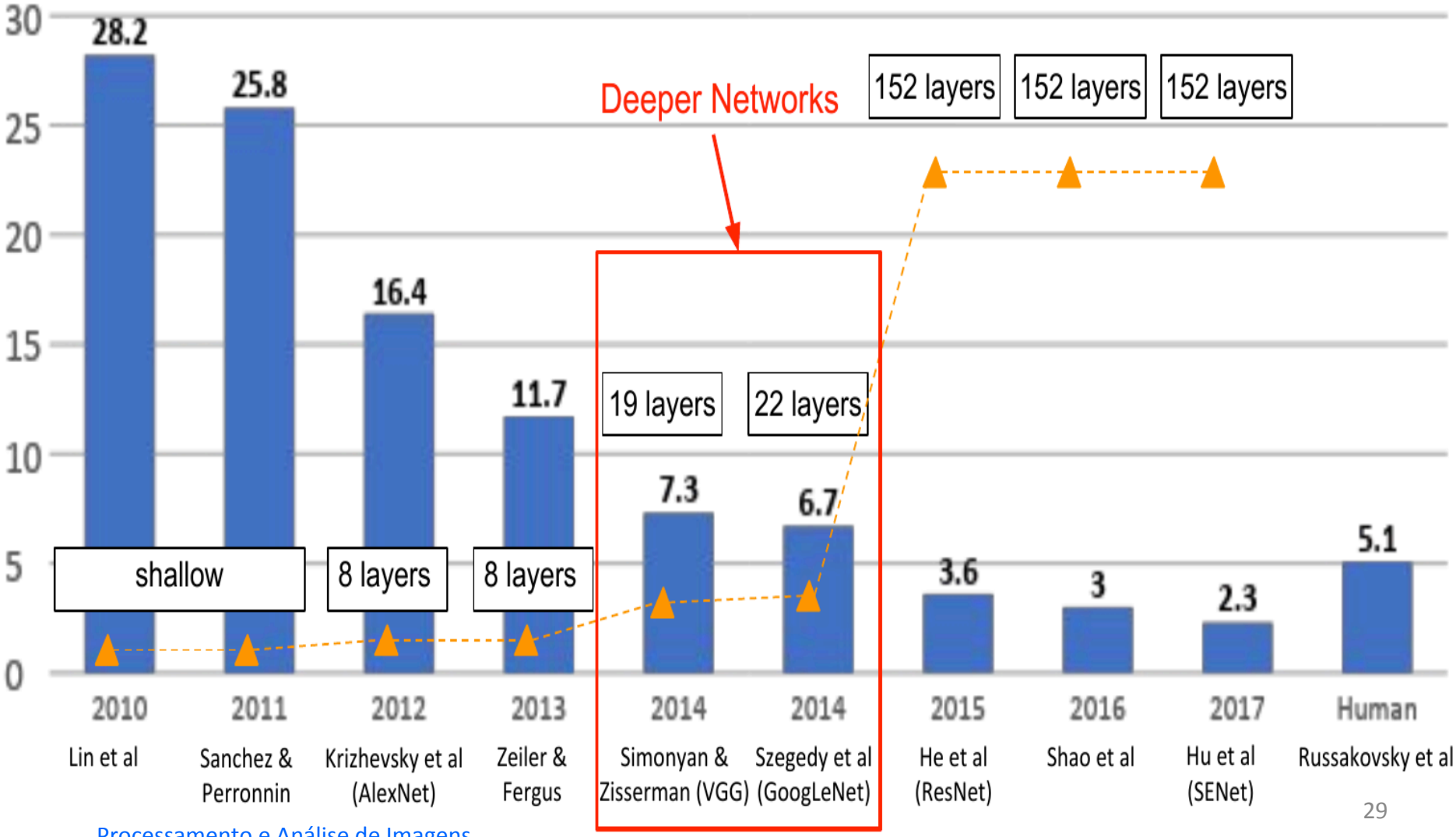


Figure 3. Architecture of our 8 layer convnet model. A 224 by 224 crop of an image (with 3 color planes) is presented as the input. This is convolved with 96 different 1st layer filters (red), each of size 7 by 7, using a stride of 2 in both x and y. The resulting feature maps are then: (i) passed through a rectified linear function (not shown), (ii) pooled (max within 3x3 regions, using stride 2) and (iii) contrast normalized across feature maps to give 96 different 55 by 55 element feature maps. Similar operations are repeated in layers 2,3,4,5. The last two layers are fully connected, taking features from the top convolutional layer as input in vector form ($6 \cdot 6 \cdot 256 = 9216$ dimensions). The final layer is a C-way softmax function, C being the number of classes. All filters and feature maps are square in shape.

M. Zeiler and R. Fergus, Visualizing and Understanding Convolutional Networks, ECCV 2014 (Best Paper Award winner)

ImageNet Challenge (winners)



VGG -16

- A modification of AlexNet
- The first to really have principled design
- Instead of having a lot of hyperparameters, we have a simpler network
- It focuses on having only the following blocks:

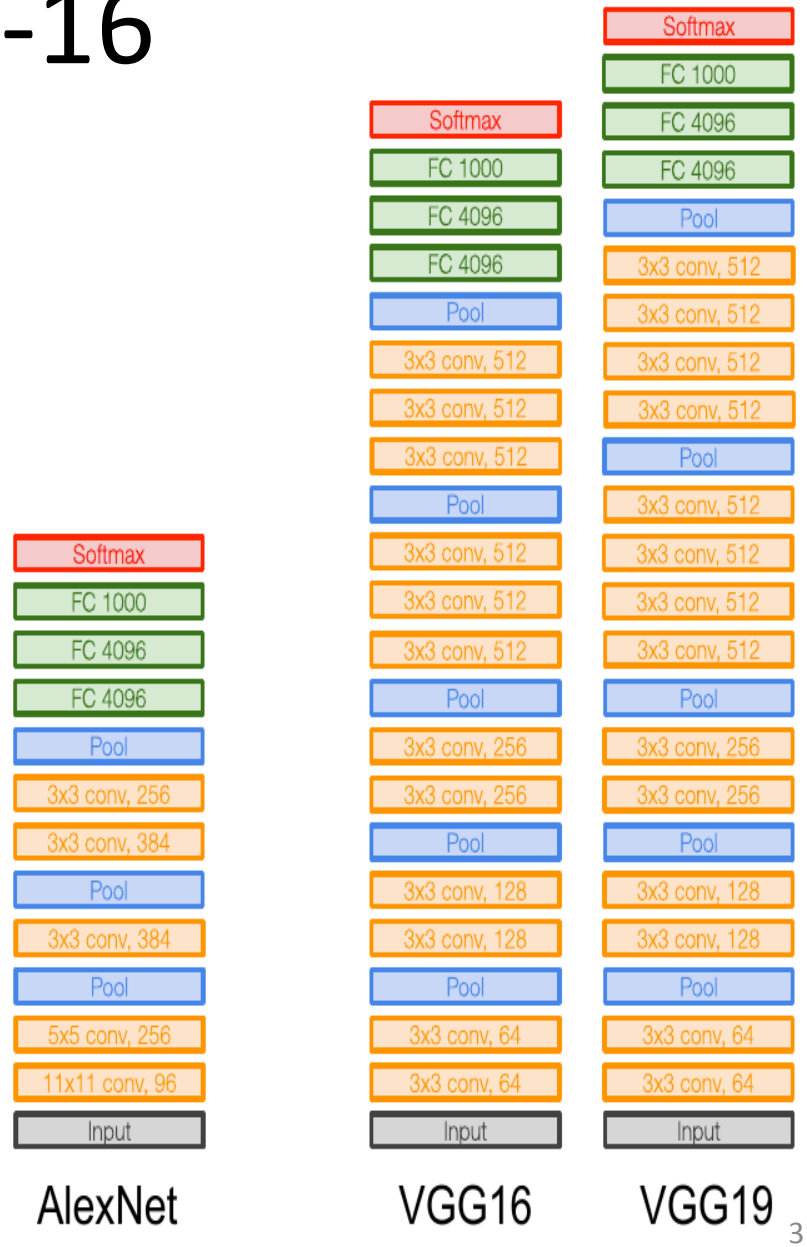
CONV = 3 X 3 filters, stride = 1, "same"

MAX-POOL = 2 X 2, stride = 2

[Simonyan & Zisserman 2015. Very deep convolutional networks for large-scale image recognition]

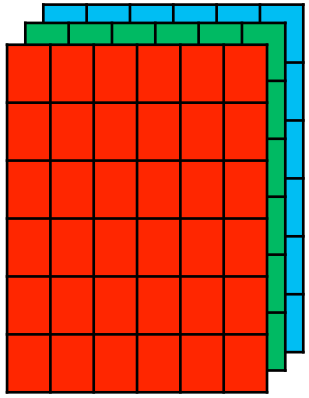
VGG -16

- Smaller filters, deeper networks
- AlexNet: 8 layers
- VGG: 16 to 19 layers
- Total memory: 24M * 4bytes = 96MB / imagem (forward pass)
- parameters: 138M
- ImageNet top 5 error: From 11.7% (ZFNet) to 7.3%



VGG - 16

CONV = 3×3 filter, s = 1, same MAX-POOL = 2×2 , s = 2



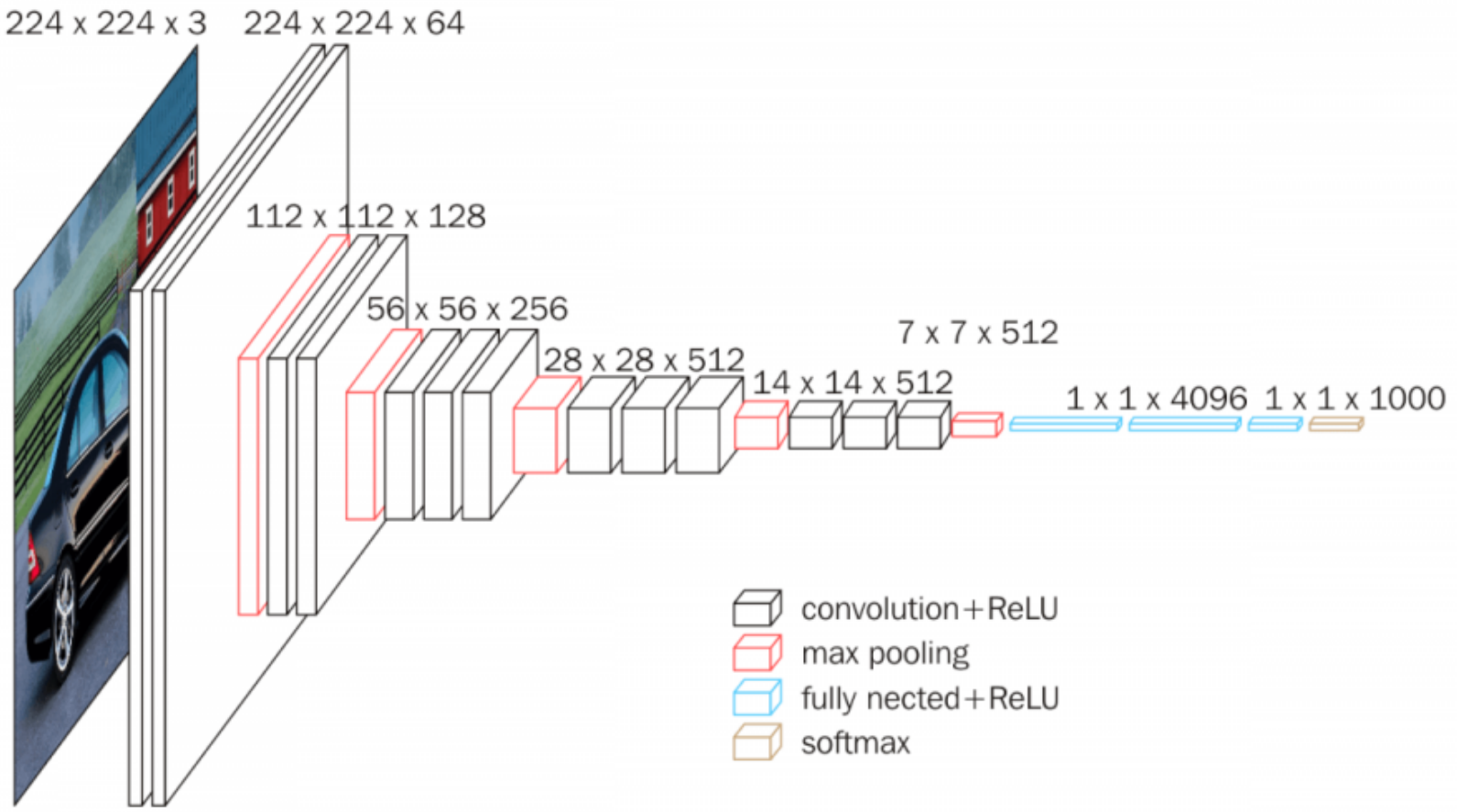
224×224 ×3

→ [CONV 64] ×2 → 224×224×64 → POOL → 112×112 ×64 → [CONV 128] ×2 → 112×112 ×128 → POOL → 56×56 ×128

→ [CONV 256] ×3 → 56×56 ×256 → POOL → 28×28 ×256 → [CONV 512] ×3 → 28×28 ×512 → POOL → 14×14×512

→ [CONV 512] ×3 → 14×14 ×512 → POOL → 7×7×512 → FC 4096 → FC 4096 → Softmax 1000

VGG - 16

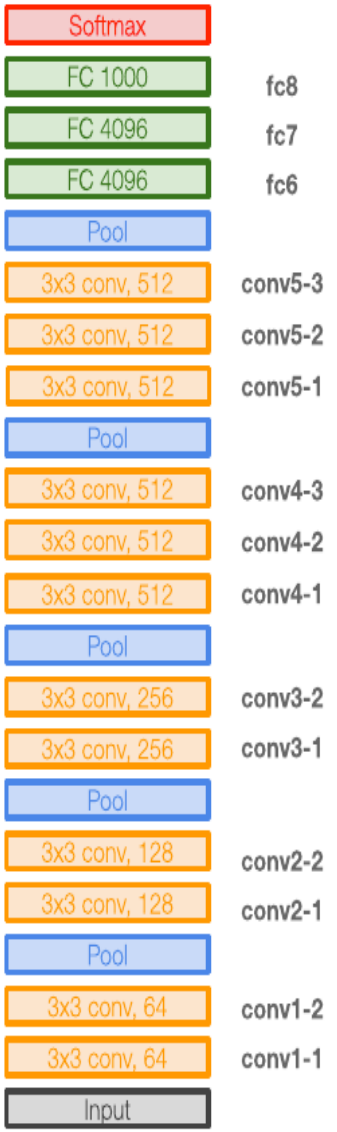


https://towardsdatascience.com

VGG -16

(not counting biases)

INPUT: [224x224x3] memory: 224*224*3=150K params: 0
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

VGG

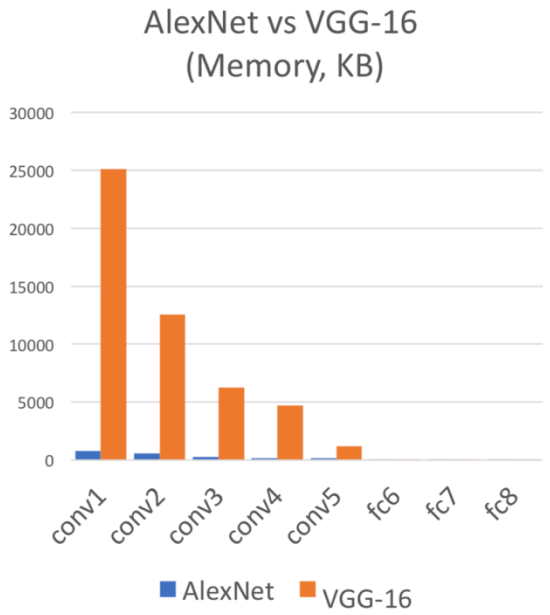
- Why use smaller filters? (3x3 conv)

  Stacking three 3x3 conv filters (stride 1) has the same effective receptive field as a 7x7 filter conv layer

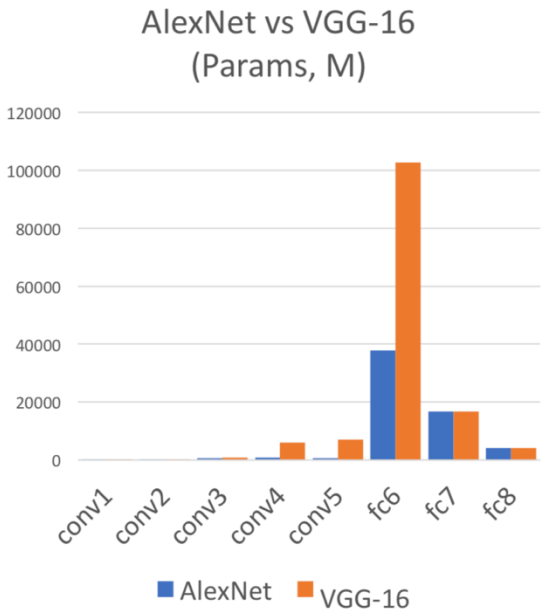
  - Deeper, more non-linearities (a ReLU after each 3x3 conv)
  - Fewer parameters: ~3/5

VGG x AlexNet

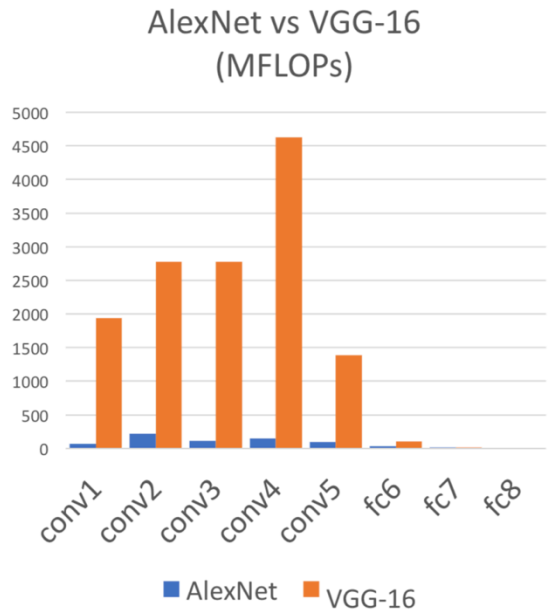
AlexNet vs VGG-16: Much bigger network!



AlexNet total: 1.9 MB
VGG-16 total: 48.6 MB (25x)

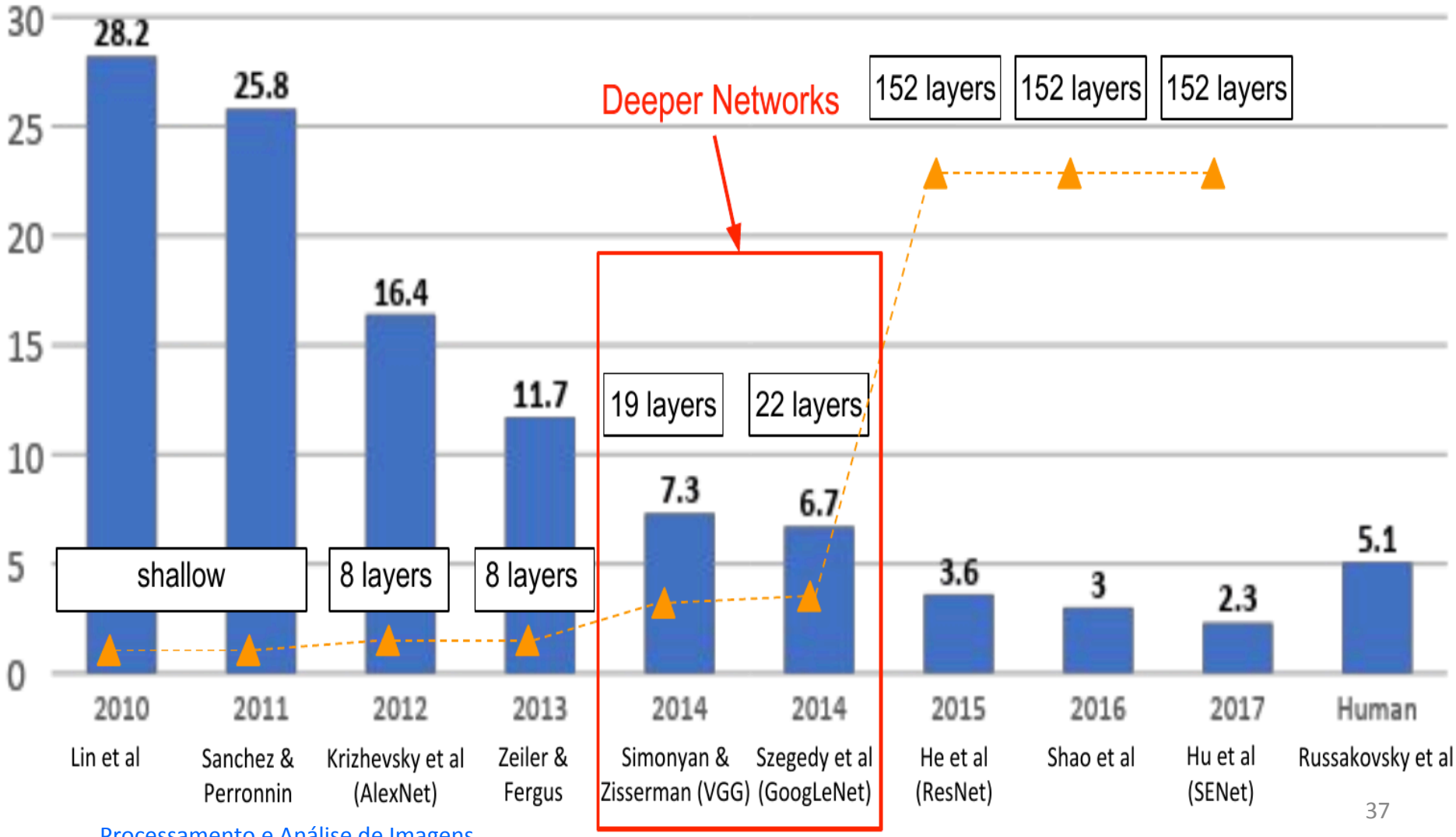


AlexNet total: 61M
VGG-16 total: 138M (2.3x)

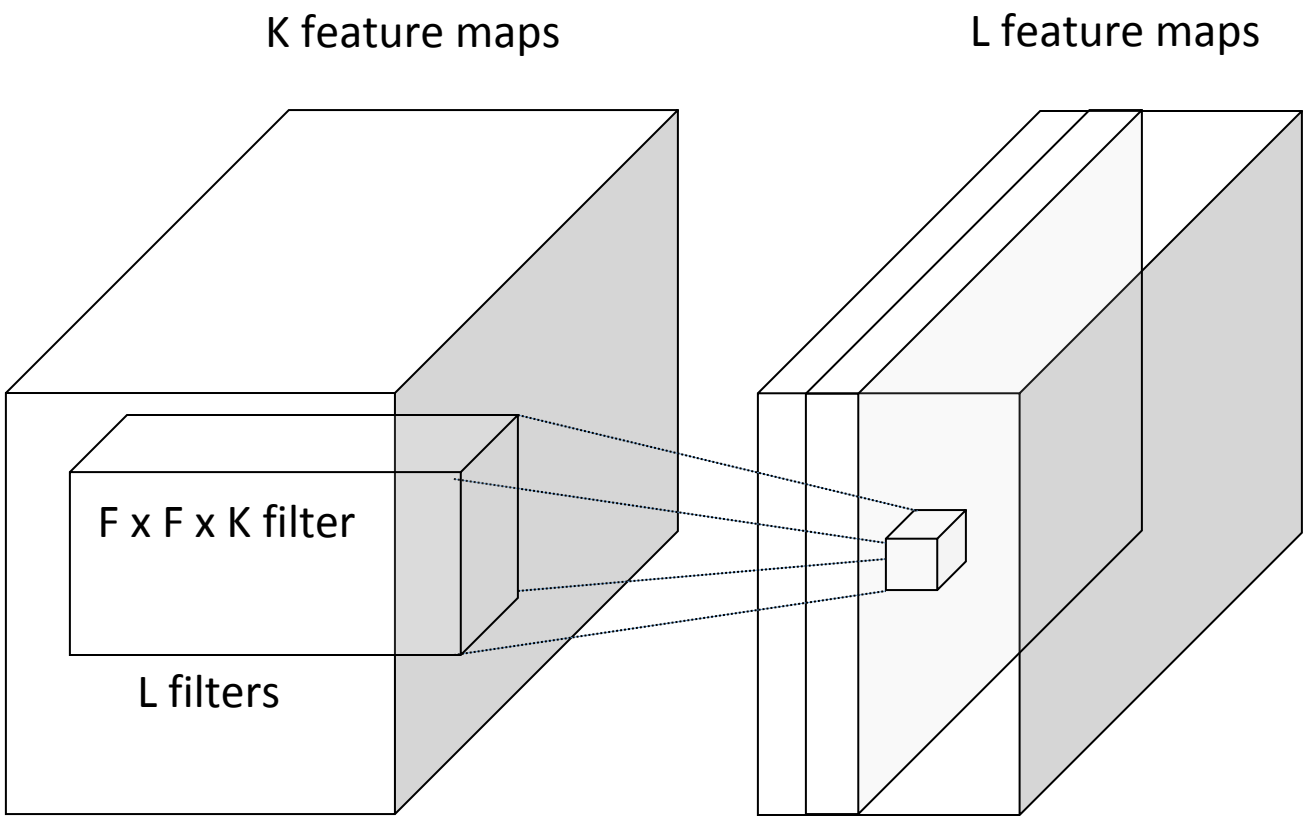


AlexNet total: 0.7 GFLOP
VGG-16 total: 13.6 GFLOP (19.4x)

ImageNet Challenge (winners)

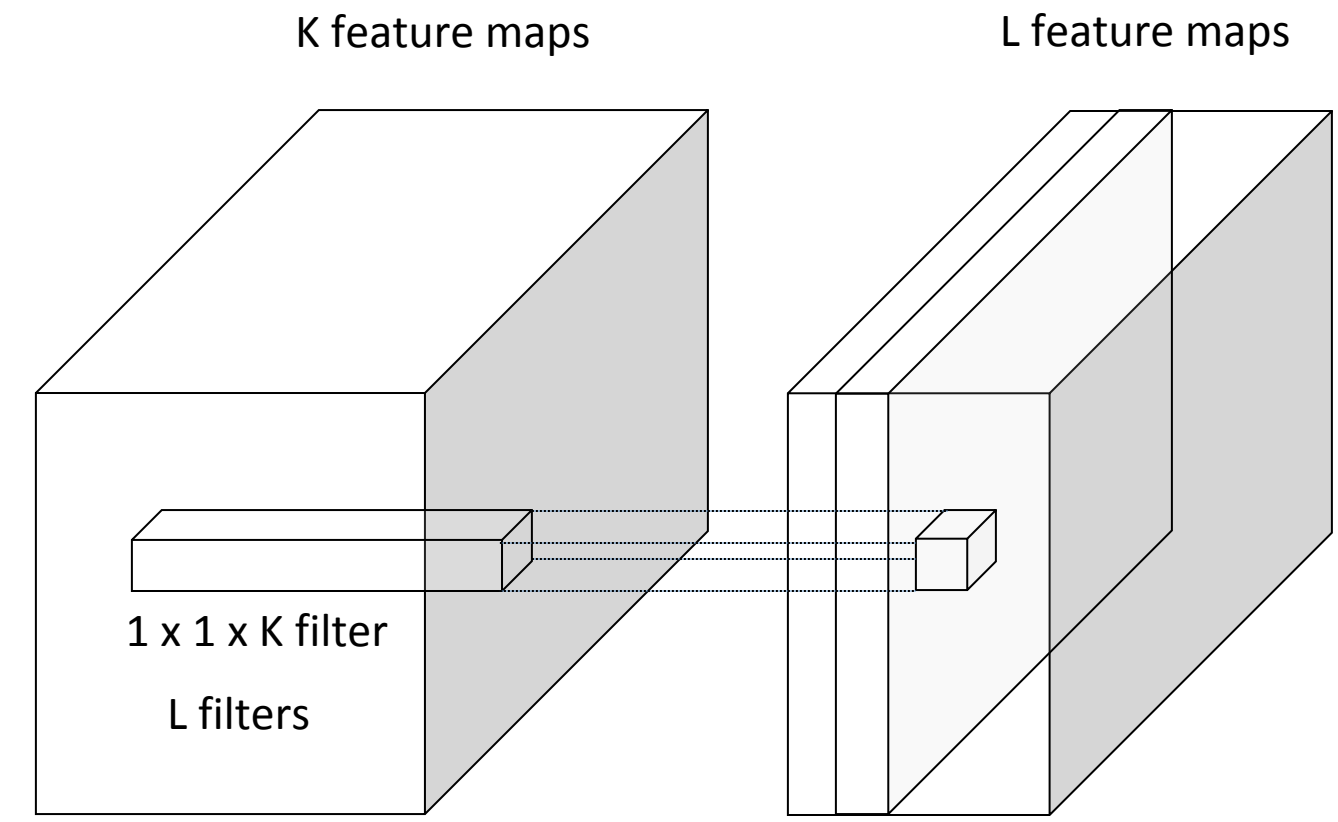


1x1 convolutions



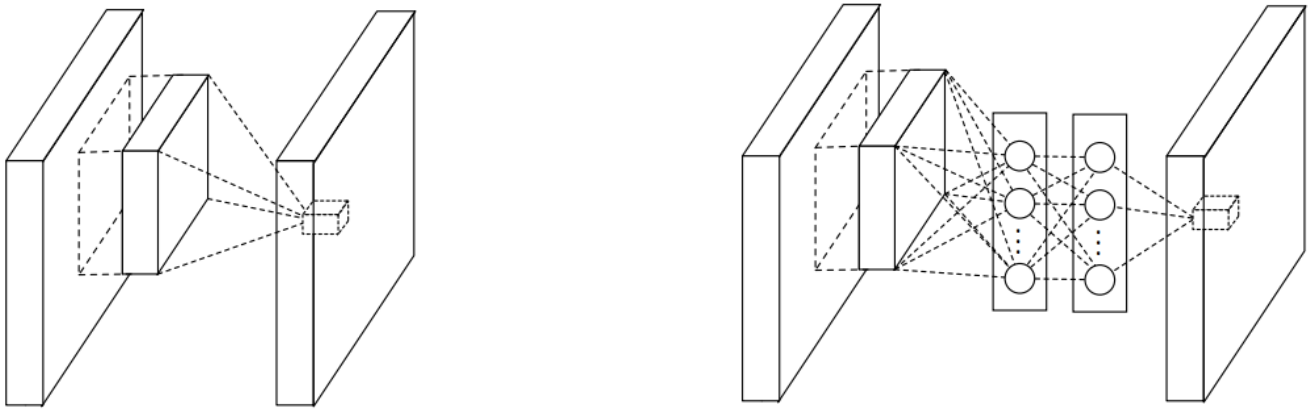
convolutional layer

1x1 convolutions



1 x 1 convolutional layer

Network in network



(a) Linear convolution layer

(b) Mlpconv layer

M. Lin, Q. Chen, and S. Yan, Network in network, ICLR 2014

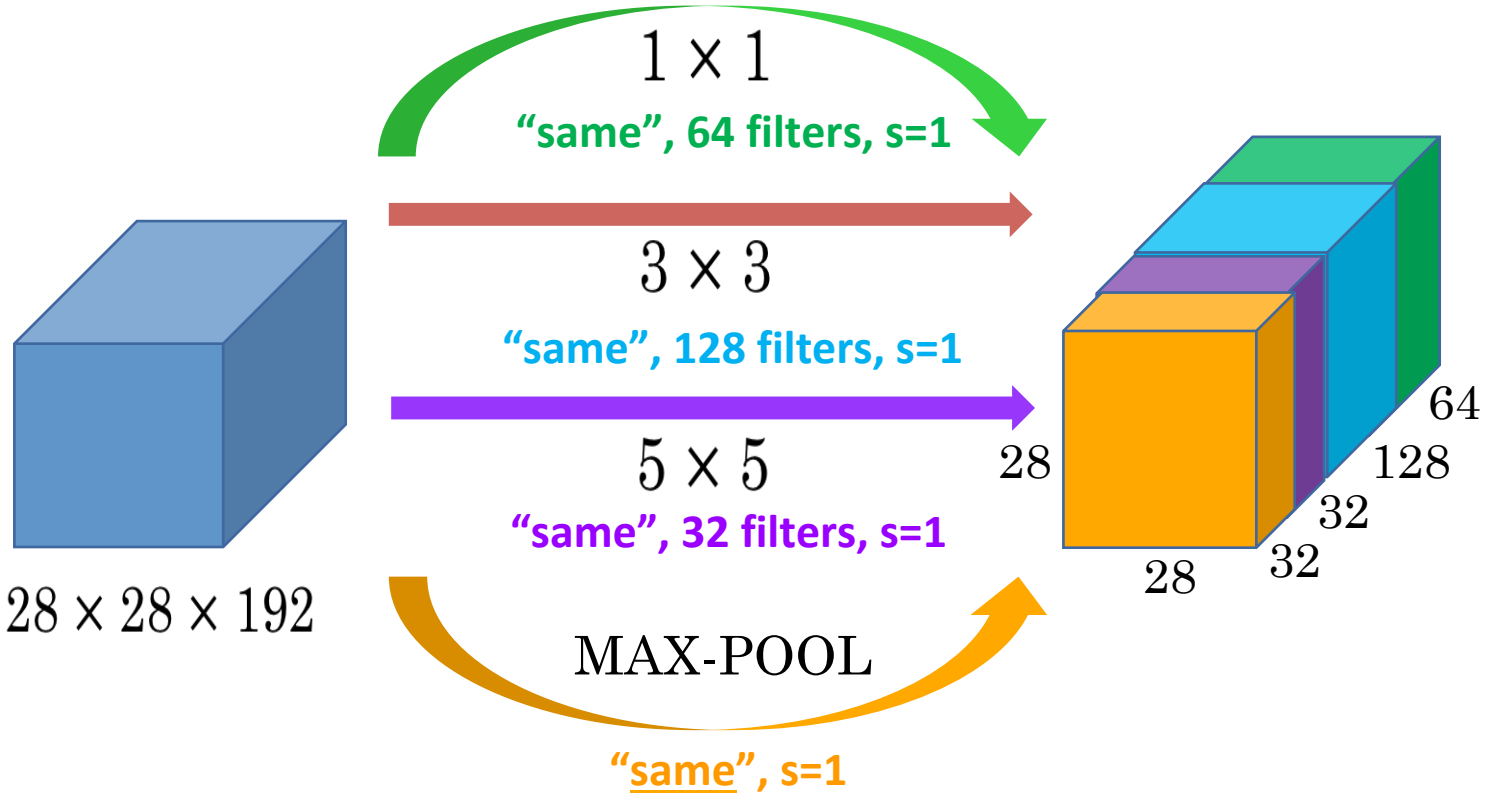
Network in network

- A 1x1 convolution, also called a Network in Network, is used in many CNNs such as the ResNet and Inception models
- A 1 x 1 convolution is useful when we want to decrease the number of channels (feature transformation)
- It reduces the amount of computation
- It behaves as a fully-connected layer
- If we have specified the number of 1x1 Conv filters to be the same as the number of input channels, the output will contain the same number of channels but act as an additional nonlinearity

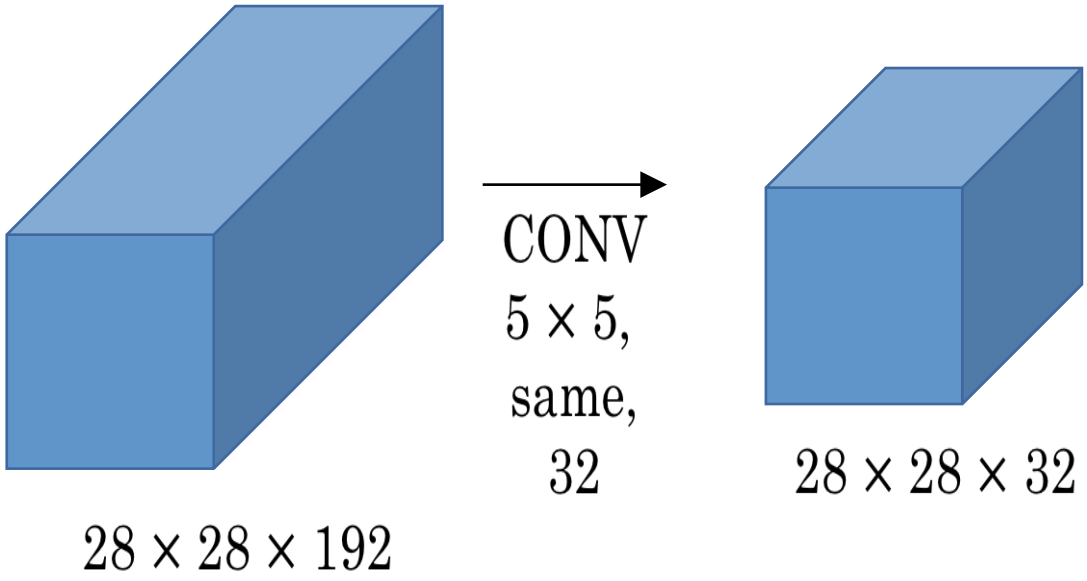
Inception network

- When you create a CNN, you have to decide on all the layers
- You will choose a 3 x 3 Conv or 5 x 5 Conv or maybe a max pooling layer
- You have so many choices!
- What inception tells us is: why not use them all at once and let the network decide which ones are important?

Inception network

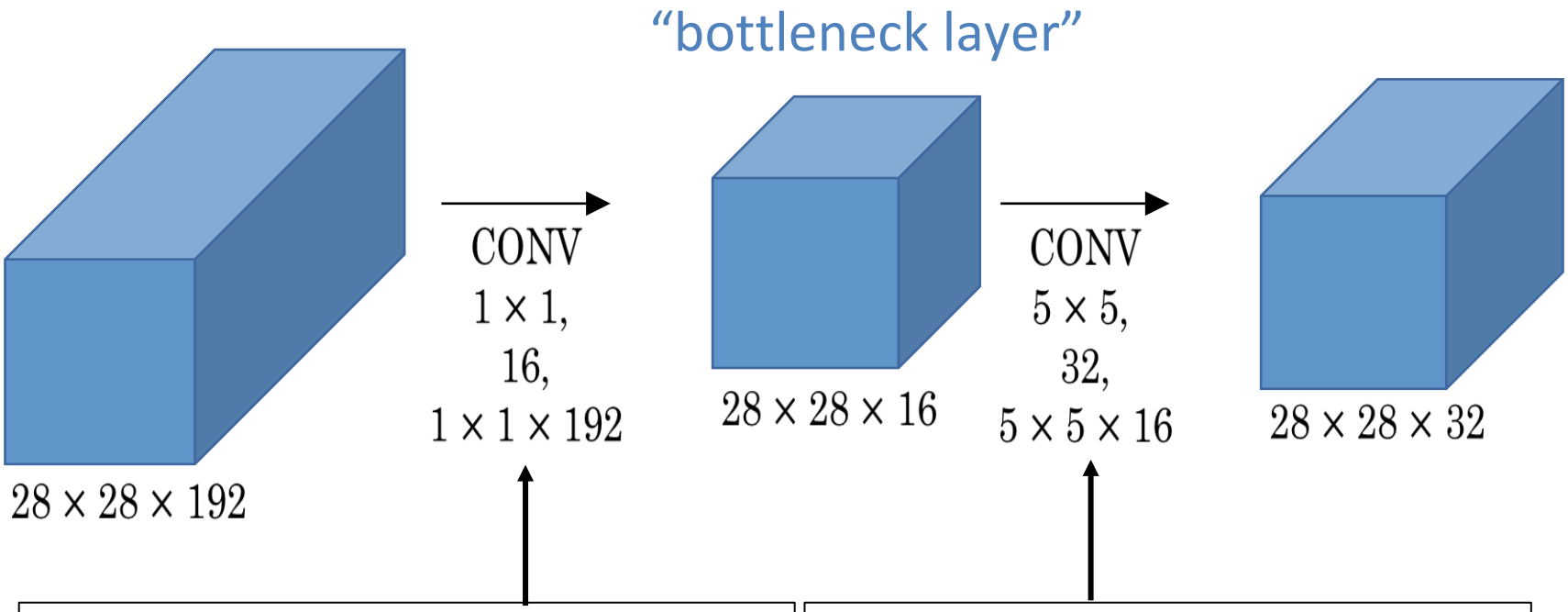


The computational cost problem



Number of multiplications:
(28 * 28 * 32) * (5 * 5 * 192) = 120M!

Using 1x1 convolutions



Number of multiplications:
(28 * 28 * 16) * (1 * 1 * 192) = 2.4M

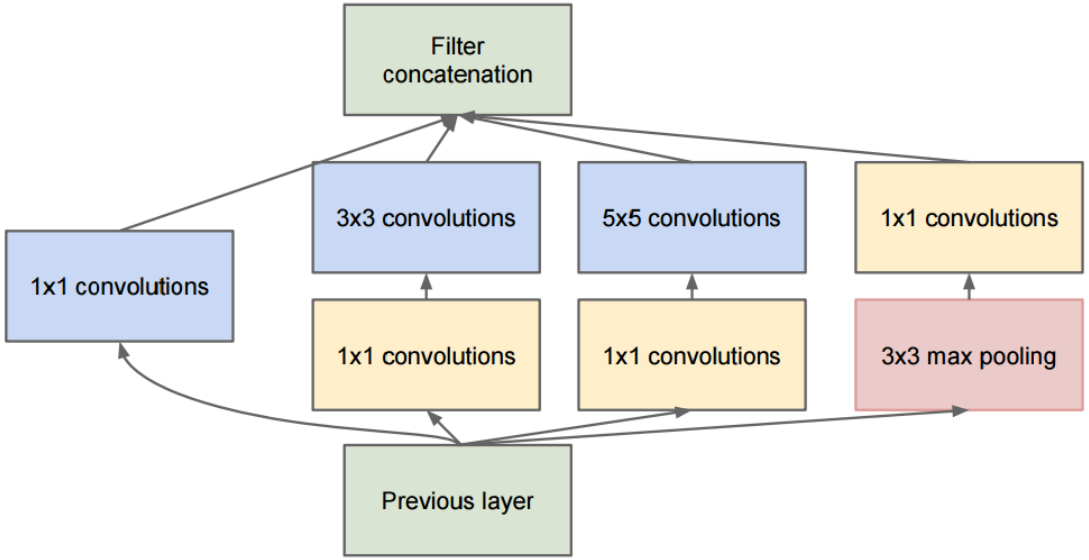
Number of multiplications:
(28 * 28 * 32) * (5 * 5 * 16) = 10M

12.4M << 120M

Inception network

The Inception Module

- Parallel paths with different receptive field sizes and operations are meant to capture sparse patterns of correlations in the stack of feature maps
- Use 1x1 convolutions for dimensionality reduction before expensive convolutions

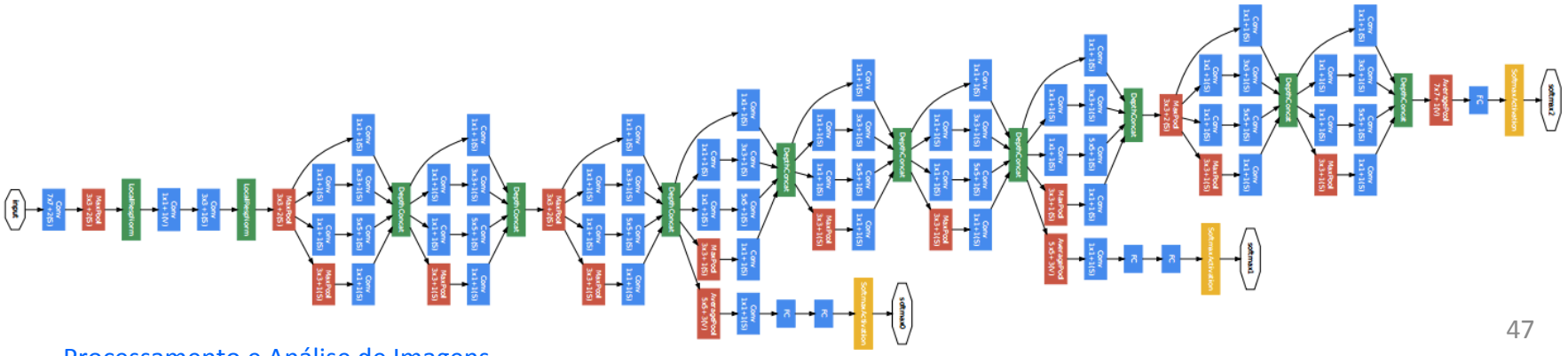


C. Szegedy et al., Going deeper with convolutions, CVPR 2015

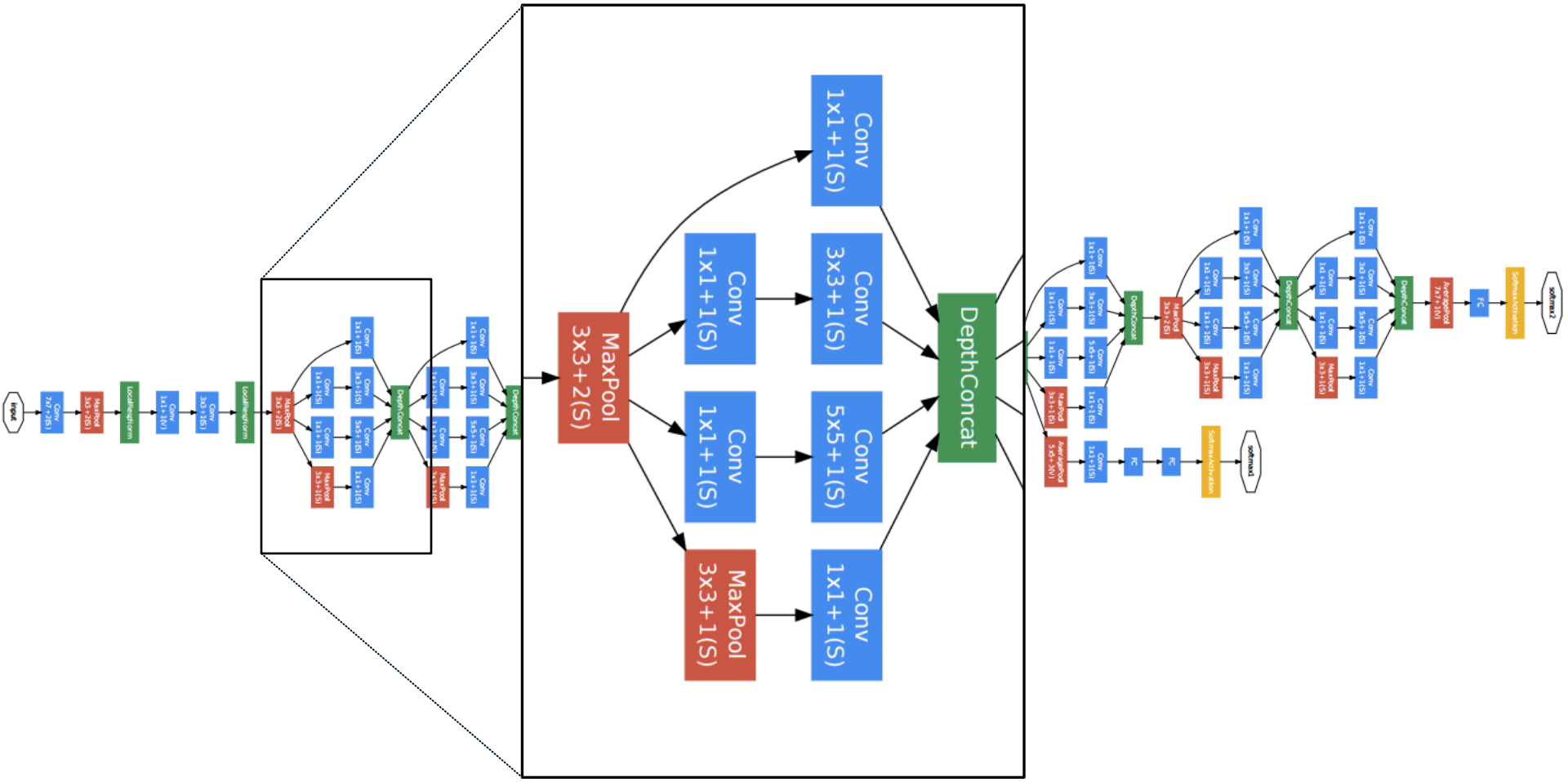
Inception network

The Inception Network

- Stack of Inception modules, extra maxpool layers
- Deeper, but computationally efficient networks
- 22 layers with parameters
- No multiple FC layers, instead use average pooling
- “Only” 6M parameters
- Winner of the ILSVRC´14 classification task (6.7% top 5 error)
- Newer versions have been developed



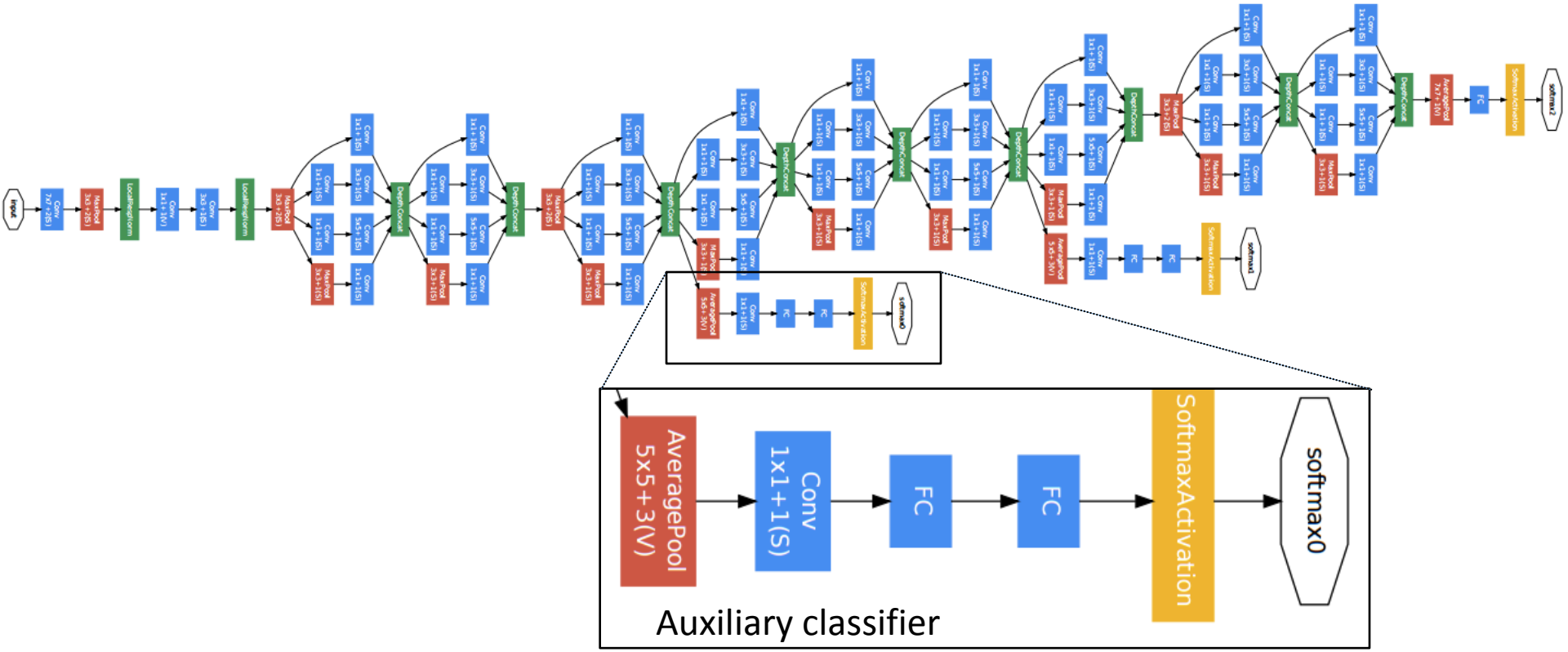
GoogLeNet



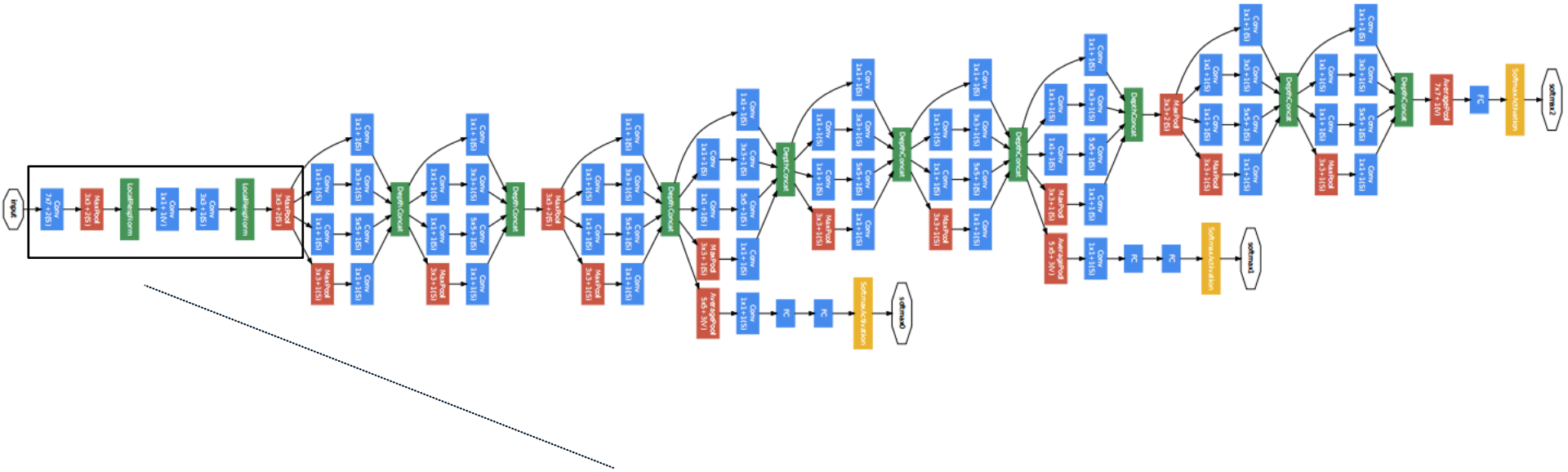
Inception module

GoogLeNet

Before Batch Normalization



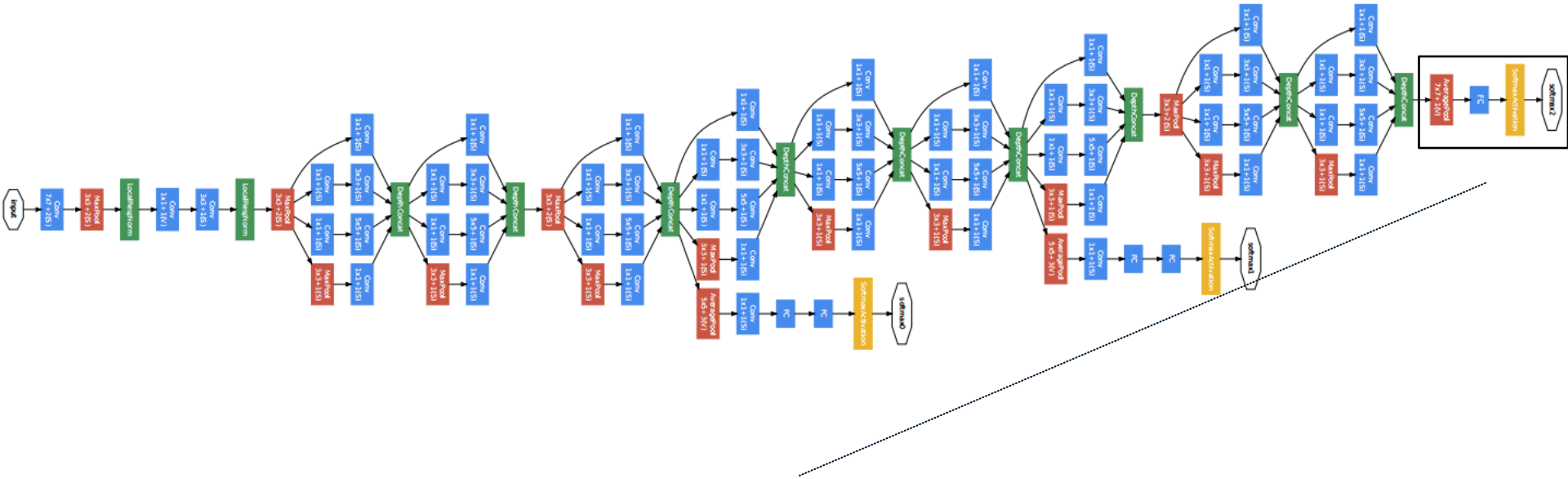
GoogLeNet



Traditional CNN:

CONV-Pool-CONV-CONV-Pool

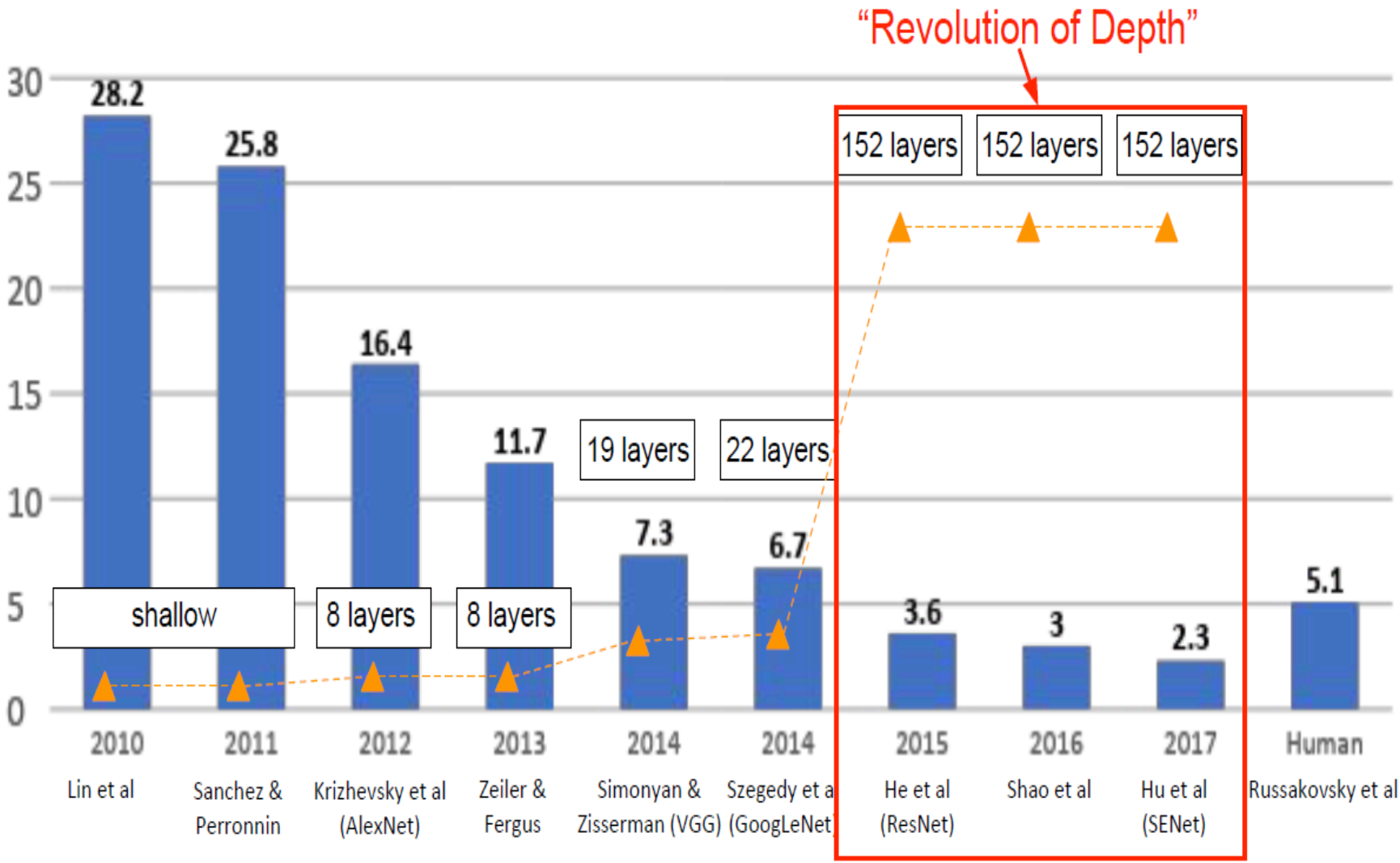
GoogLeNet



Output of the classifier:

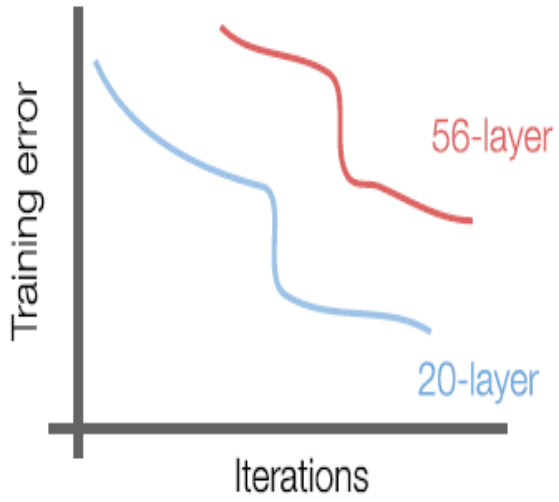
Pool-FC-Softmax

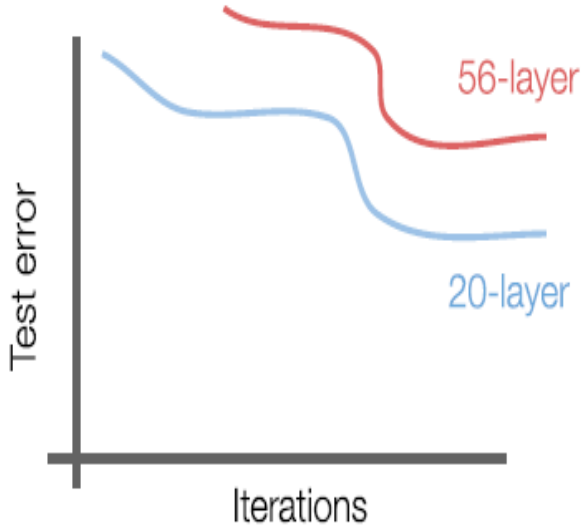
ImageNet Challenge (winners)



ResNets

- What happens when we continue to stack deeper layers on a CNN?



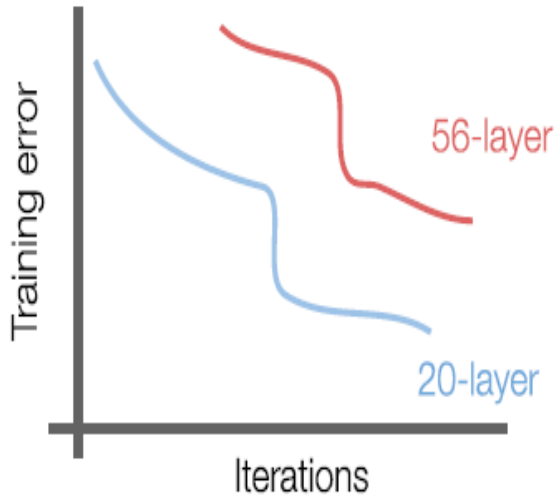


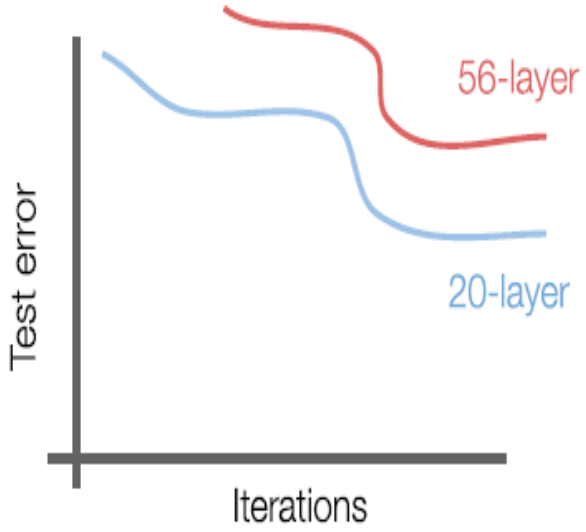
What's wrong with these curves?

The 56-layer model is worse in both training and testing!

ResNets

- What happens when we continue to stack deeper layers on a CNN?





What's wrong with these curves?

The problem is not overfitting!

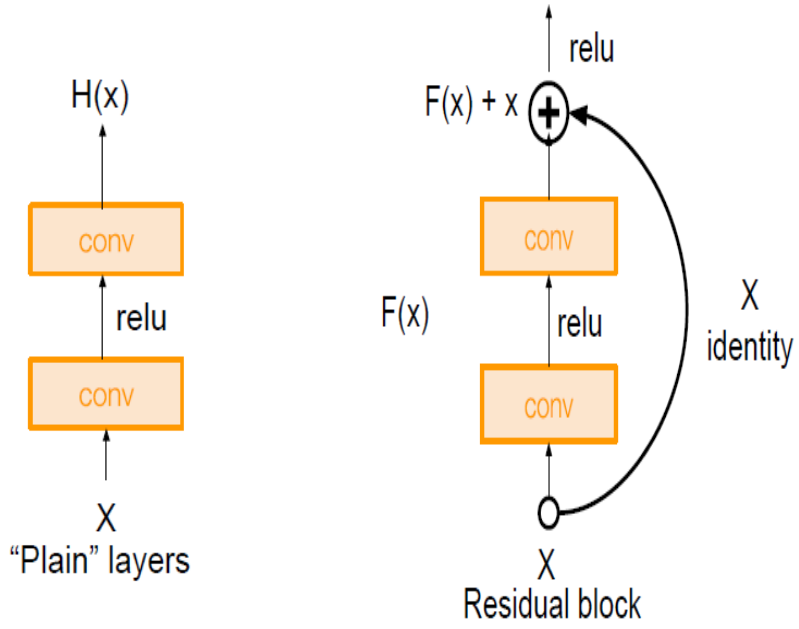
ResNets

Hypothesis: the problem is optimization

- Deeper models are more difficult to optimize
- Deeper models should be at least as good as shallow ones: consider a model with K layers, and add an extra layer that does just an identity mapping
- Very, very deep NNs are difficult to train because of the problem of gradient vanishing and explosion

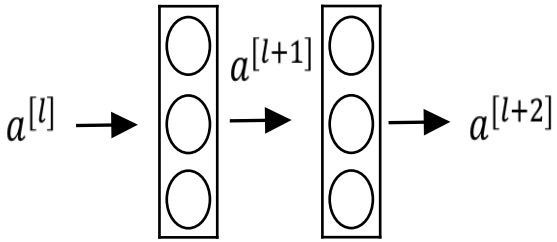
ResNets

- Solution: Use the network layers to fit a residual map instead of directly trying to adjust the desired underlying mapping
- To solve this, we will learn about the skip connection
- Take the activation of one layer and suddenly feed it to another layer
- Allows you to train large NNs, even with layers greater than 100



K. He, X. Zhang, S. Ren, and J. Sun, Deep Residual Learning for Image Recognition

Residual blocks



$a^{[l]}$ → Linear → ReLU → $a^{[l+1]}$ → Linear → ReLU → $a^{[l+2]}$

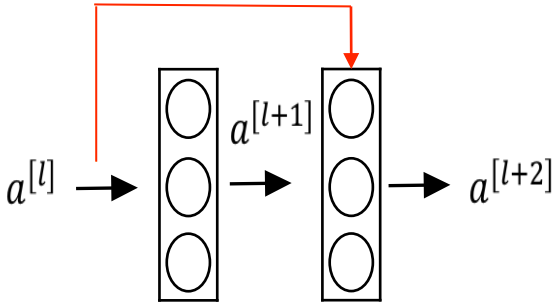
$$z^{[l+1]} = W^{[l+1]}\, a^{[l]} + b^{[l+1]}$$

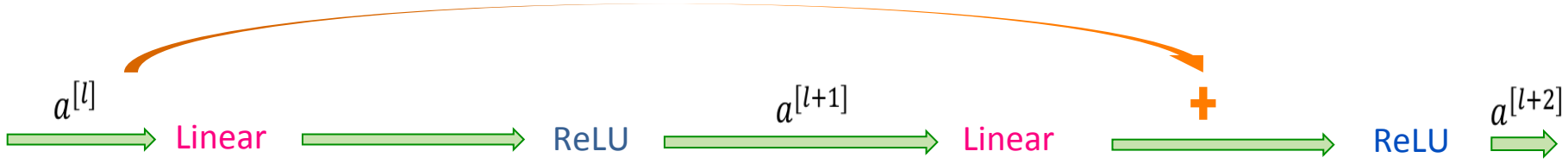
$$a^{[l+1]} = g(z^{[l+1]})$$

$$z^{[l+2]} = W^{[l+2]} a^{[l+1]} + b^{[l+2]}$$

$$a^{[l+2]} = g(z^{[l+2]})$$

Residual blocks





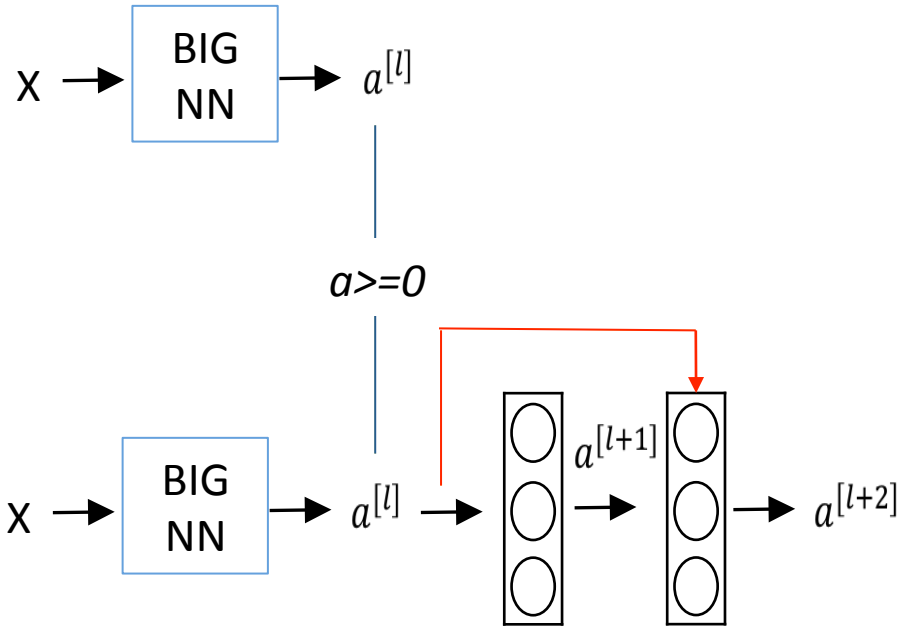
$$z^{[l+1]} = W^{[l+1]}\, a^{[l]} + b^{[l+1]}$$

$$a^{[l+1]} = g(z^{[l+1]})$$

$$z^{[l+2]} = W^{[l+2]} a^{[l+1]} + b^{[l+2]}$$

$$a^{[l+2]} = g(z^{[l+2]} + a^{[l]})$$

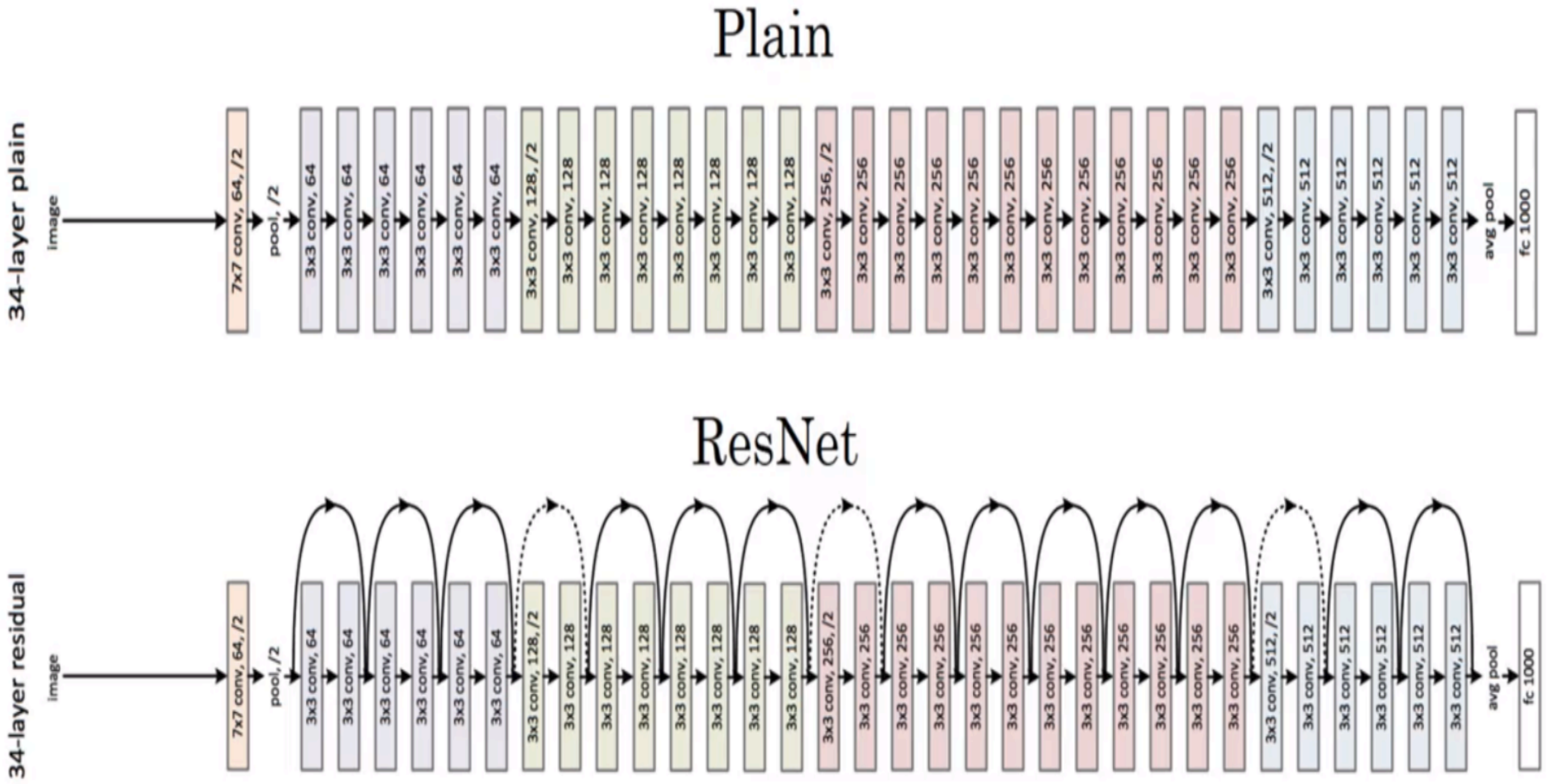
Residual blocks: Why do they work?



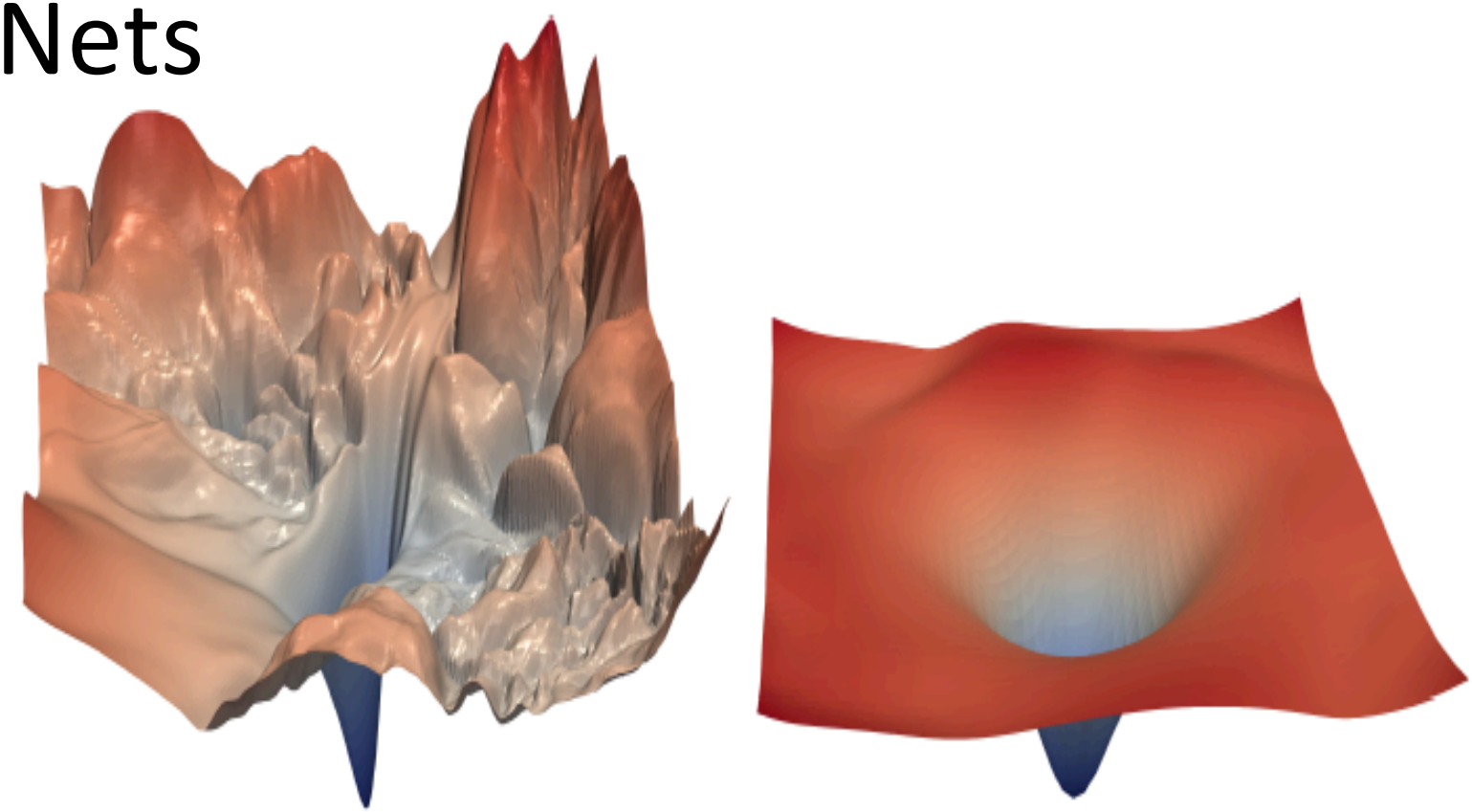
$$a^{[l+2]} = g(w^{[l+2]}a^{[l+1]} + b^{[l+2]} + a^{[l]})$$

If w,b->0, g needs only to be learnt as an identity function, which is easy! Otherwise, the network improves!

ResNet


Plain
34-layer plain
image
7x7 conv, 64, /2
pool, /2
3x3 conv, 64
3x3 conv, 128, /2
3x3 conv, 128
3x3 conv, 256, /2
3x3 conv, 256
3x3 conv, 512, /2
3x3 conv, 512
avg pool
fc 1000
ResNet
34-layer residual
image
7x7 conv, 64, /2
pool, /2
3x3 conv, 64
3x3 conv, 128, /2
3x3 conv, 128
3x3 conv, 256, /2
3x3 conv, 256
3x3 conv, 512, /2
3x3 conv, 512
avg pool
fc 1000

ResNets



(a) without skip connections

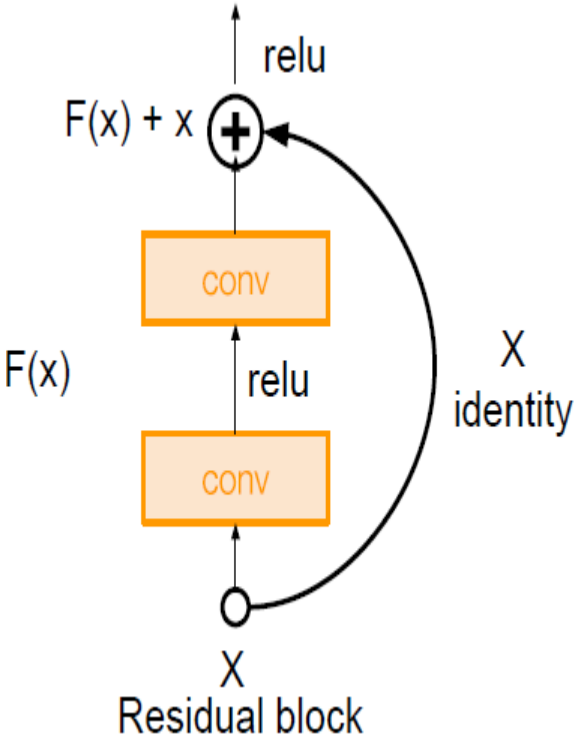
(b) with skip connections

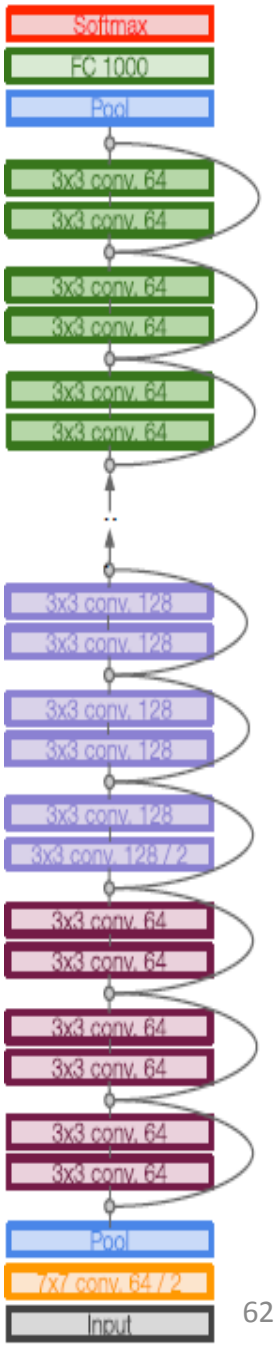
Surface of Loss Function

Li, Hao, et al. "Visualizing the loss landscape of neural nets." Advances in Neural Information Processing Systems. 2018.

ResNets

- Very deep networks using residual connections
- 152-layer model for ImageNet
- Won ILSVRC'15 (ranking) with 3.57% top 5 error

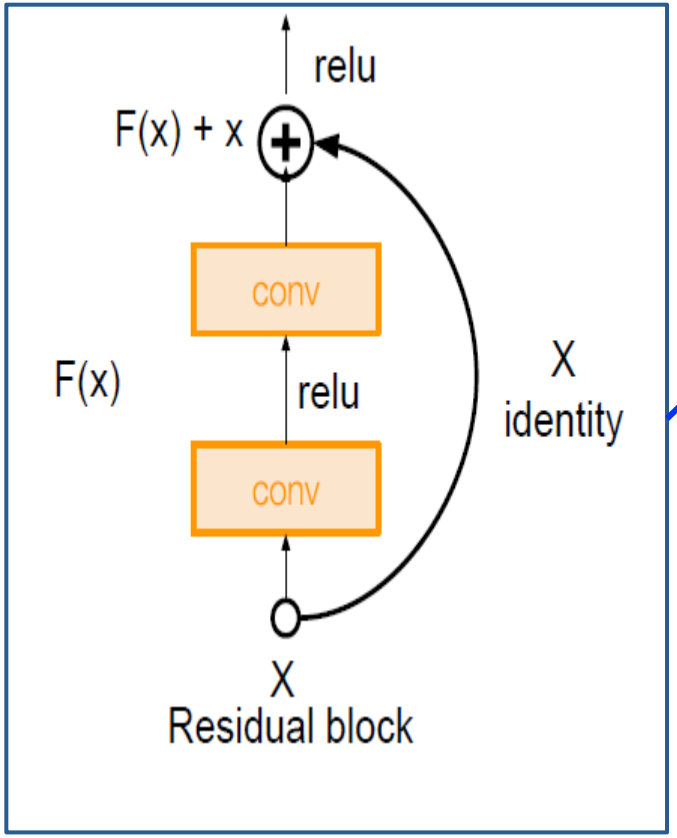


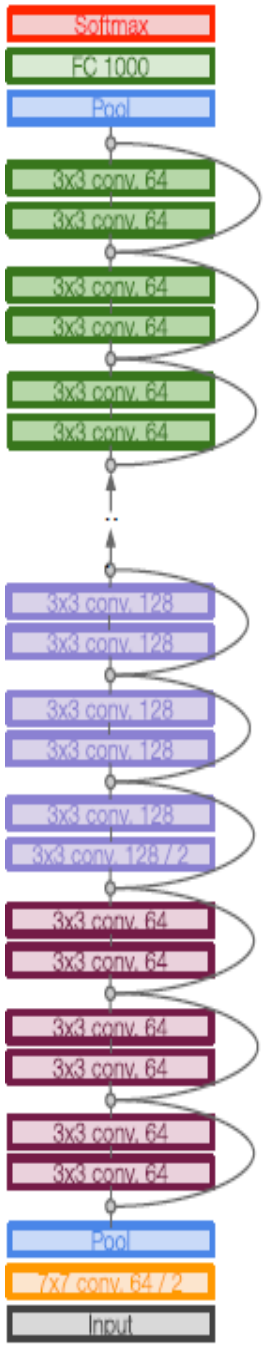


ResNets

Complete architecture:

- Stack residual blocks
- Each residual block has two 3x3 CONV layers

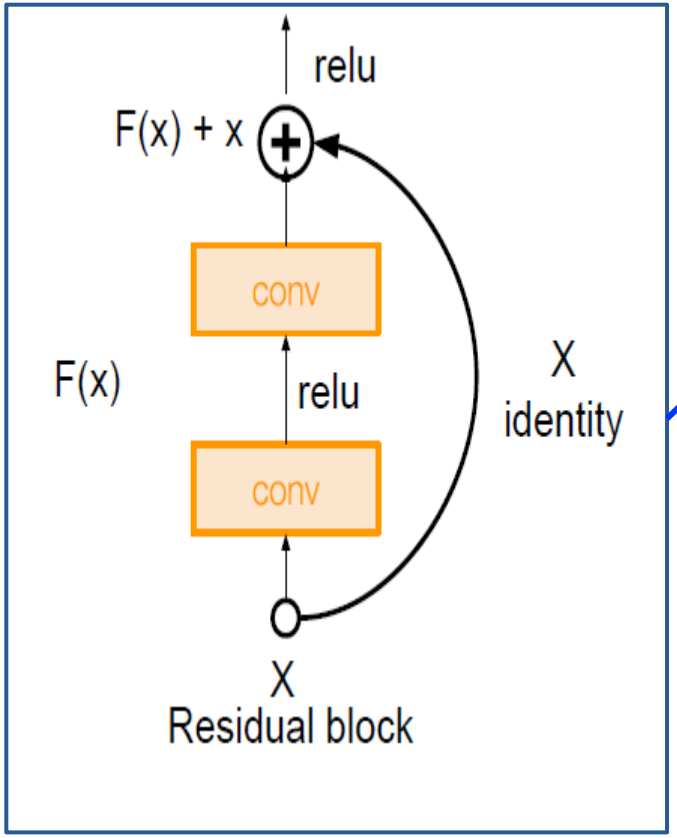


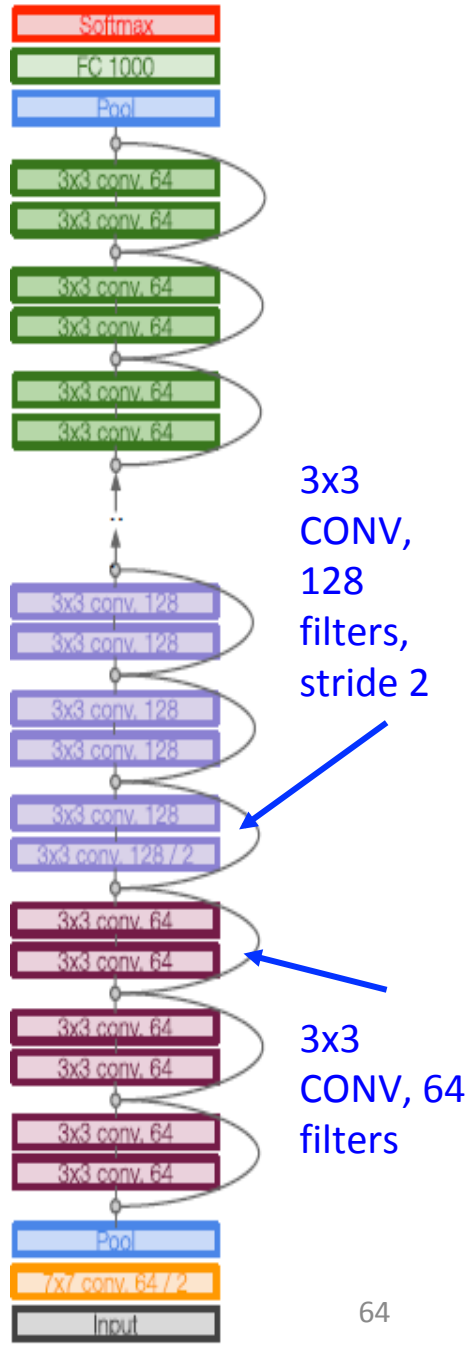


ResNets

Complete architecture:

- Stack residual blocks
- Each residual block has two 3x3 CONV layers
- Periodically double the # of filters and reduce the spatial size using stride 2

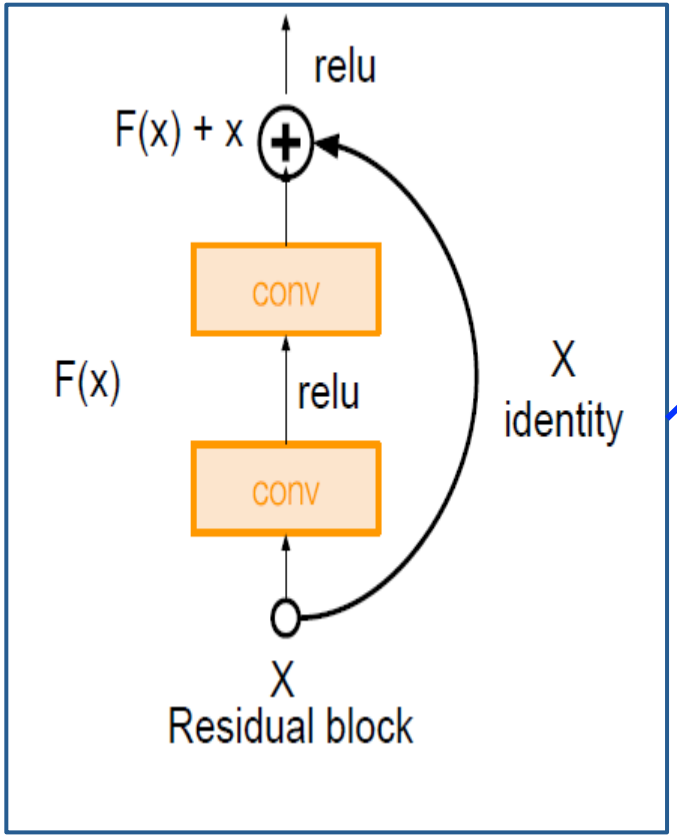


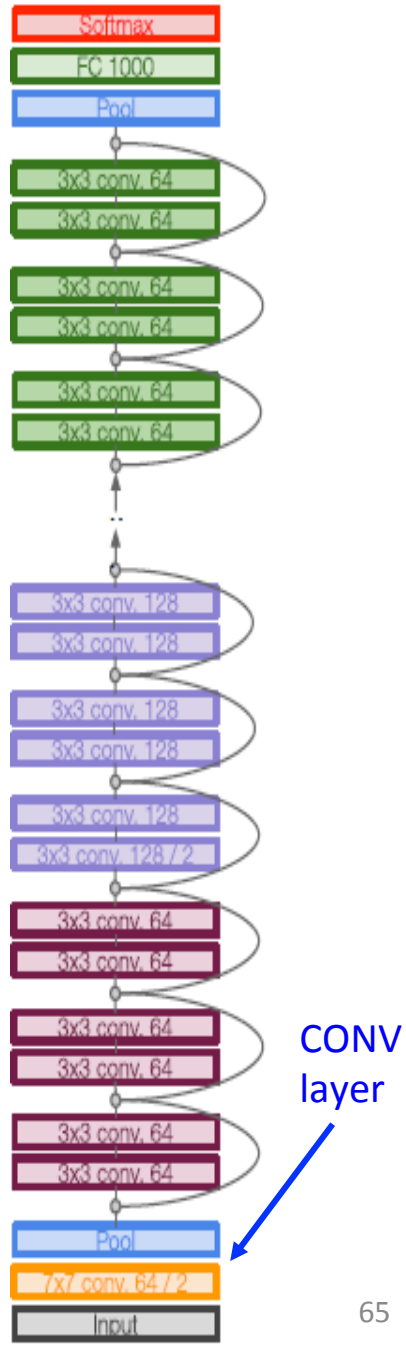


ResNets

Complete architecture:

- Stack residual blocks
- Each residual block has two 3x3 CONV layers
- Periodically double the # of filters and reduce the spatial size using stride 2
- Additional conv layer in the beginning

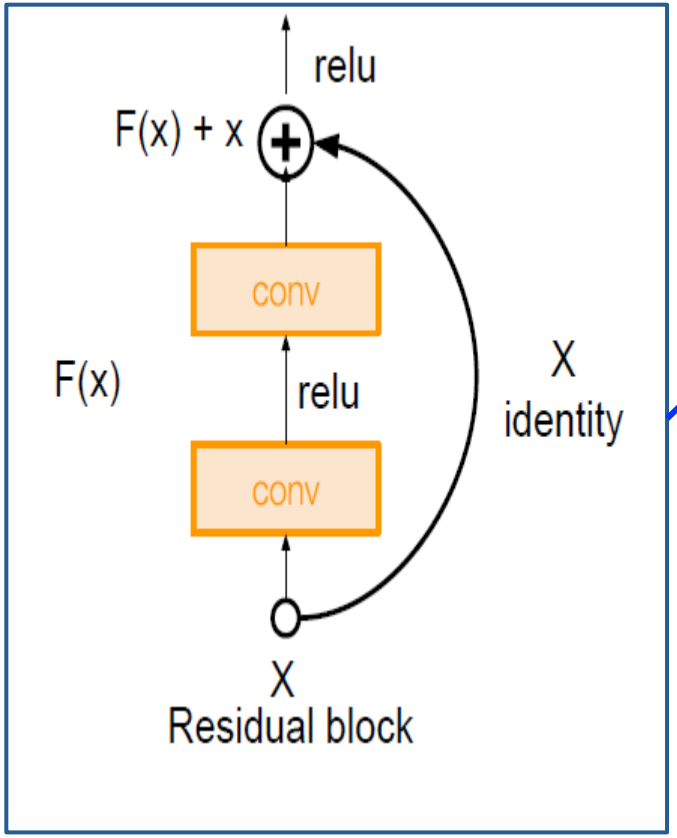


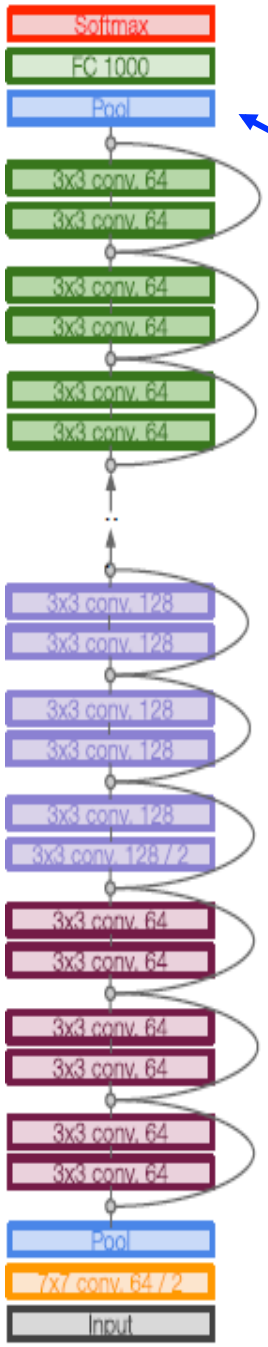


ResNets

Complete architecture:

- Stack residual blocks
- Each residual block has two 3x3 CONV layers
- Periodically double the # of filters and reduce the spatial size using stride 2
- Additional conv layer in the beginning
- No FC layers at the end (only FC 1000 for the final classes)



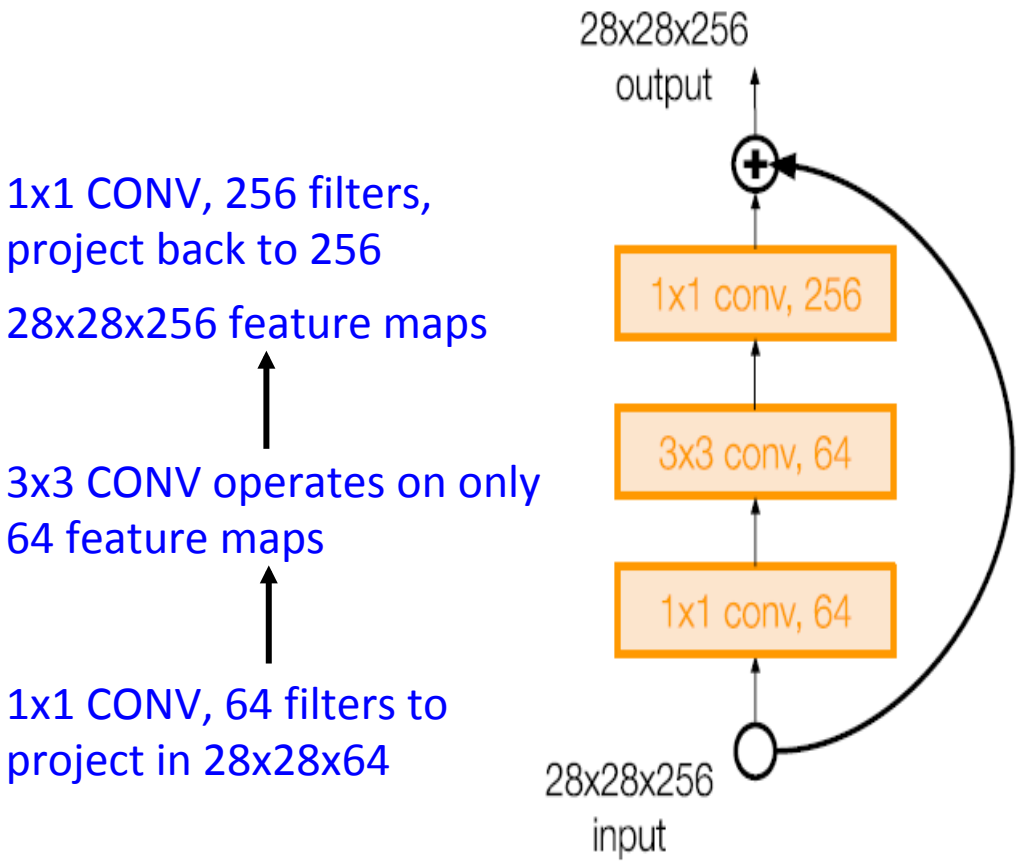


No FC layers, only 1000 FC for the final classes

Average pooling layer after the last CONV layer

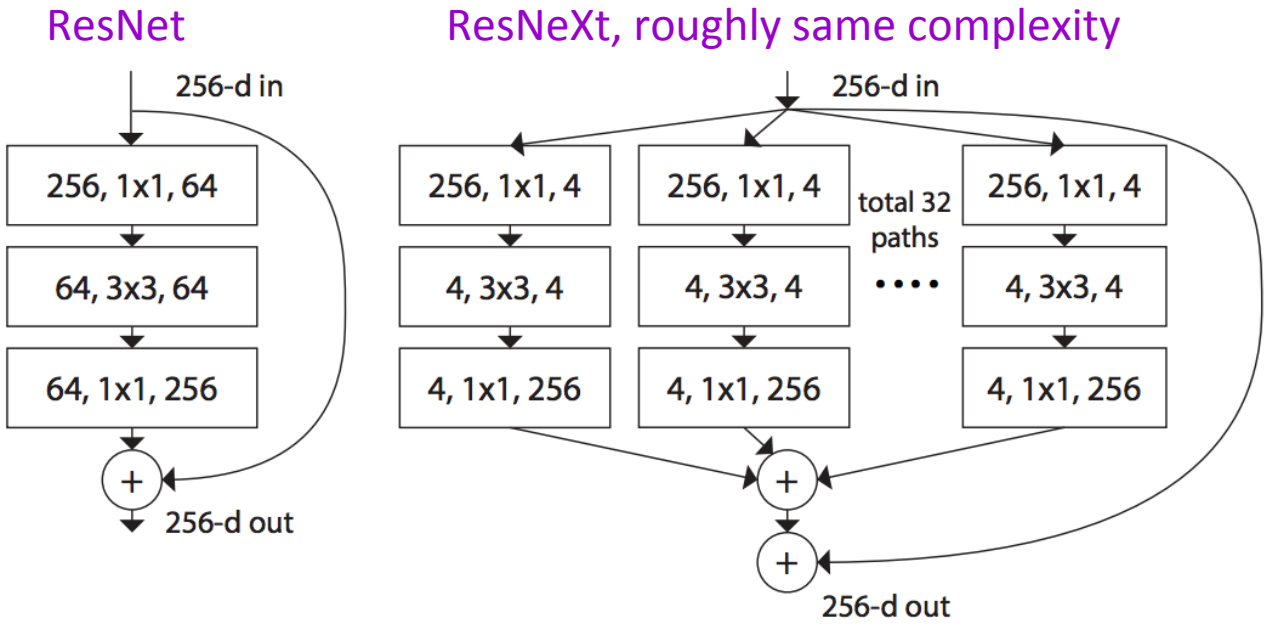
ResNets

- For ImageNet, depths of 34, 50, 101 and 152 layers were tested
- For deeper networks (ResNet-50 +), use a “bottleneck” layer to improve efficiency



ResNeXt

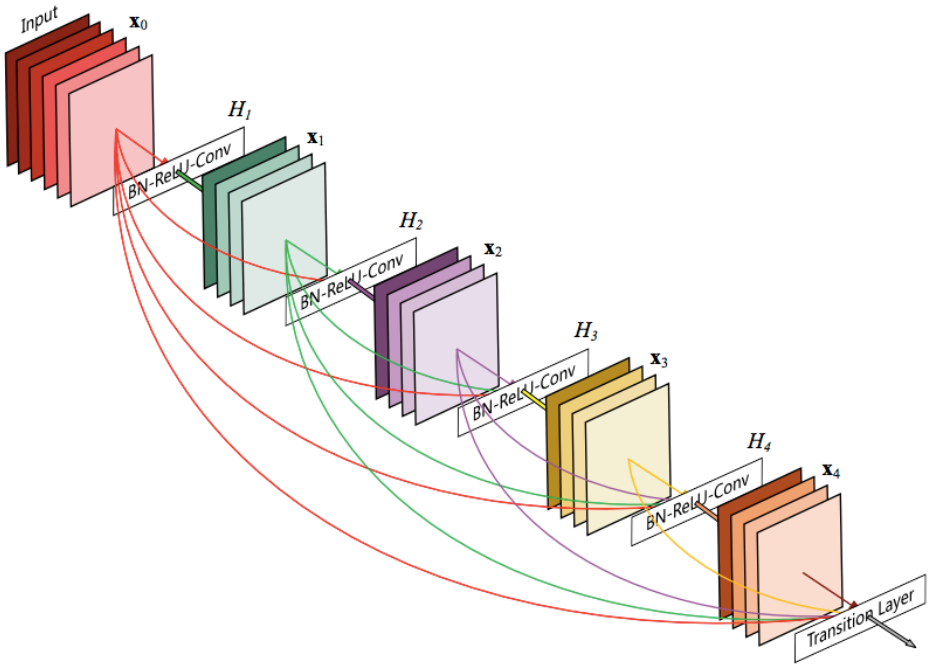
- Propose “cardinality” as a new factor in network design, apart from depth and width
- Claim that increasing cardinality is a better way to increase capacity than increasing depth or width

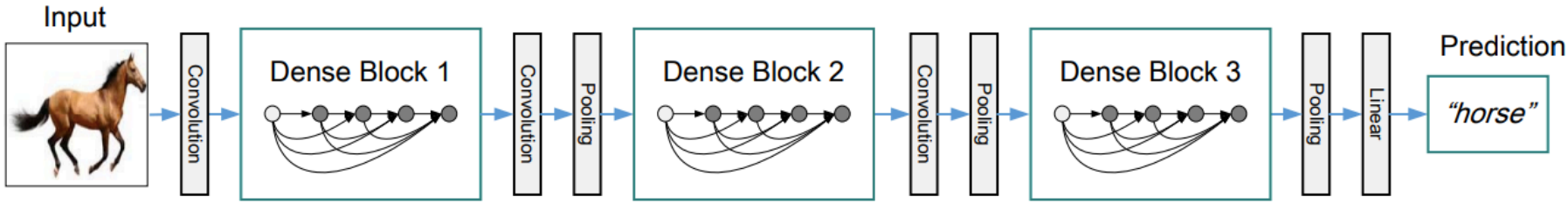


S. Xie, R. Girshick, P. Dollar, Z. Tu, and K. He, Aggregated Residual Transformations for Deep Neural Networks

DenseNet

- Each layer obtains additional inputs from all preceding layers and passes on its own feature-maps to all subsequent layers
- Concatenation is used
- Each layer is receiving a “collective knowledge”
- network can be thinner and compact (less # of channels)
- higher computational and memory efficiency





G. Huang, Z. Liu, and L. van der Maaten, Densely Connected Convolutional Networks

Comparison

A number of comparisons can be drawn:

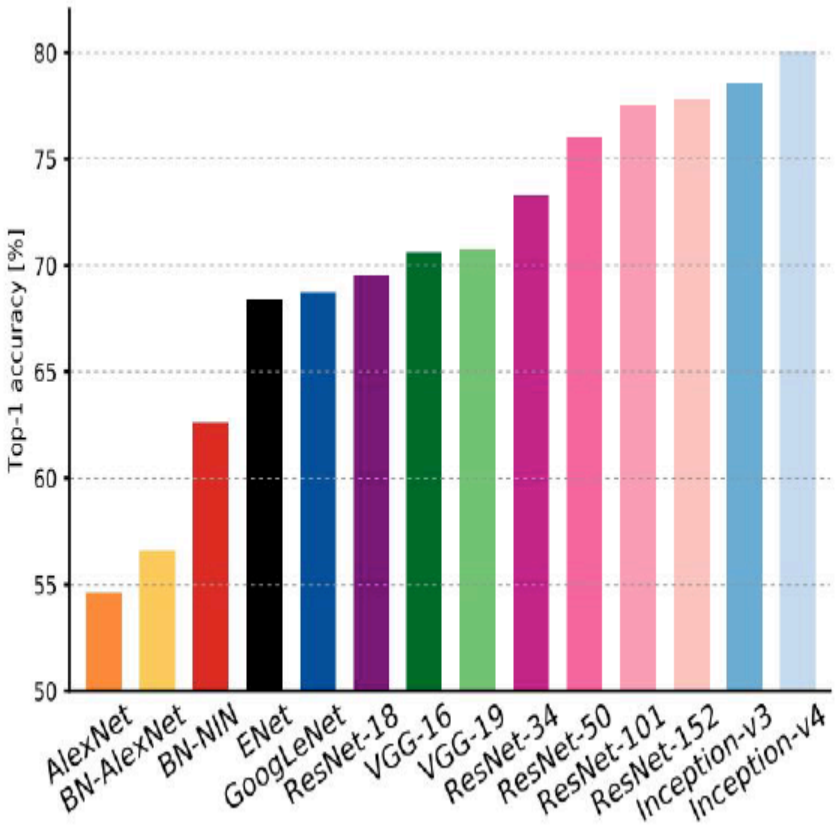
- AlexNet and ResNet-152, both have about 60M parameters but there is about 10% difference in their top-5 accuracy. But training a ResNet-152 requires a lot of computations (about 10 times more than that of AlexNet) which means more training time and energy required.
- VGGNet not only has a higher number of parameters and FLOP as compared to ResNet-152 but also has a decreased accuracy. It takes more time to train a VGGNet with reduced accuracy.
- Training an AlexNet takes about the same time as training Inception. The memory requirements are 10 times less with improved accuracy (about 9%)

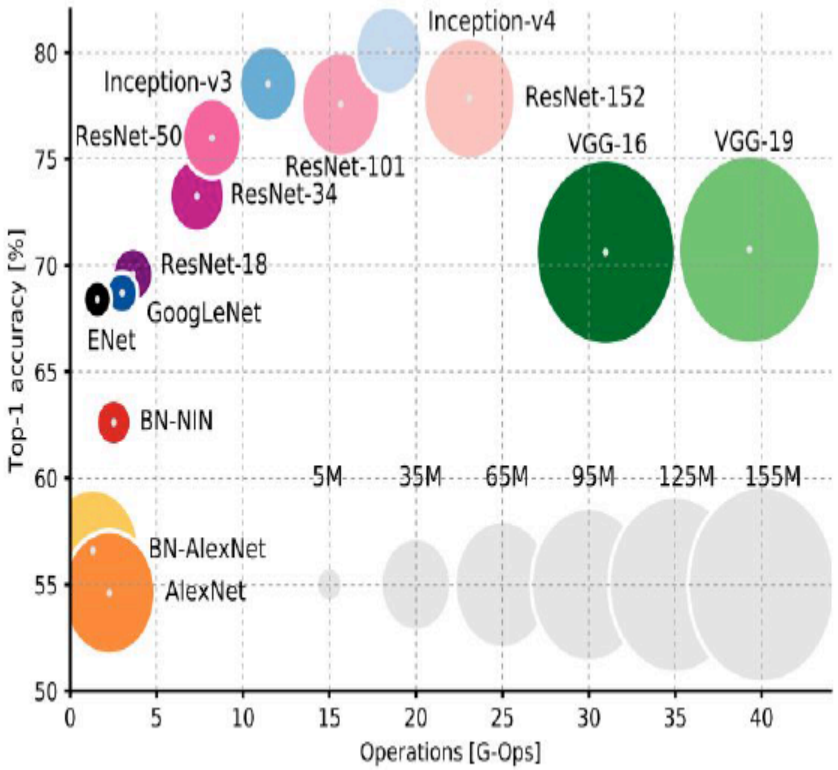
Comparison

Comparison					
Network	Year	Salient Feature	top5 accuracy	Parameters	FLOP
AlexNet	2012	Deeper	84.70%	62M	1.5B
VGGNet	2014	Fixed-size kernels	92.30%	138M	19.6B
Inception	2014	Wider - Parallel kernels	93.30%	6.4M	2B
ResNet-152	2015	Shortcut connections	95.51%	60.3M	11B

https://towardsdatascience.com

Comparison

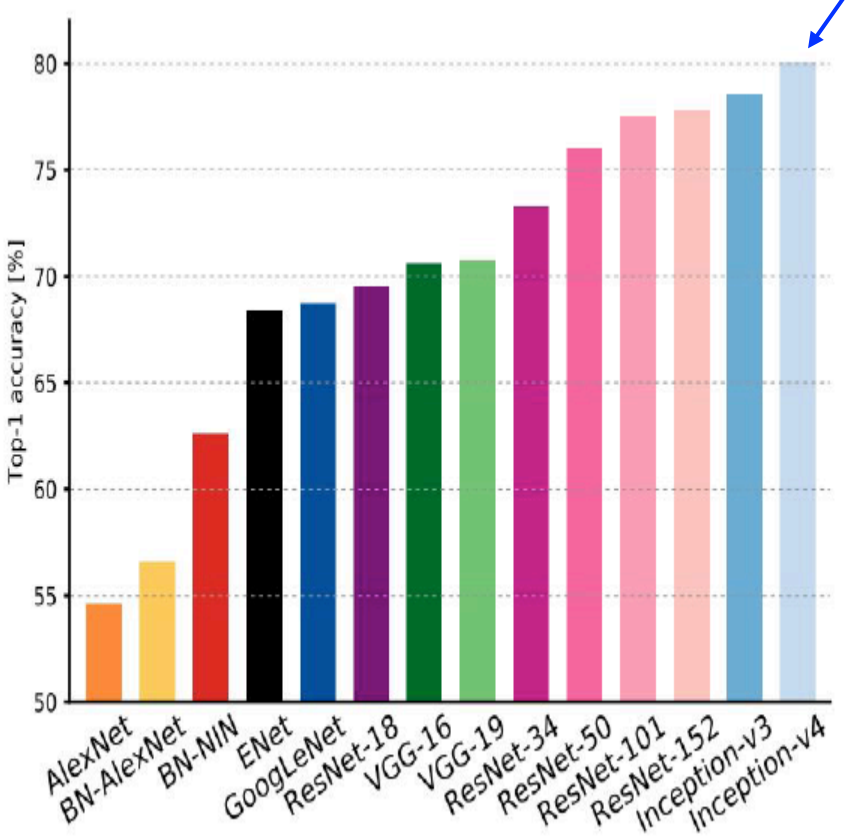


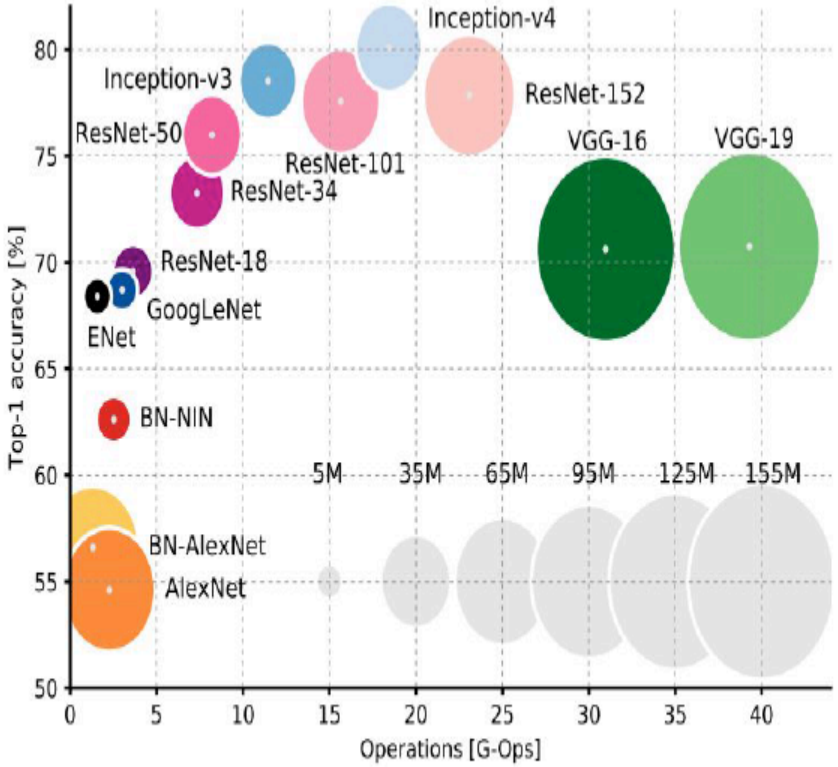


An Analysis of Deep Neural Network Models for Practical Applications, 2017.

Comparison

Inception-v4: ResNet + Inception

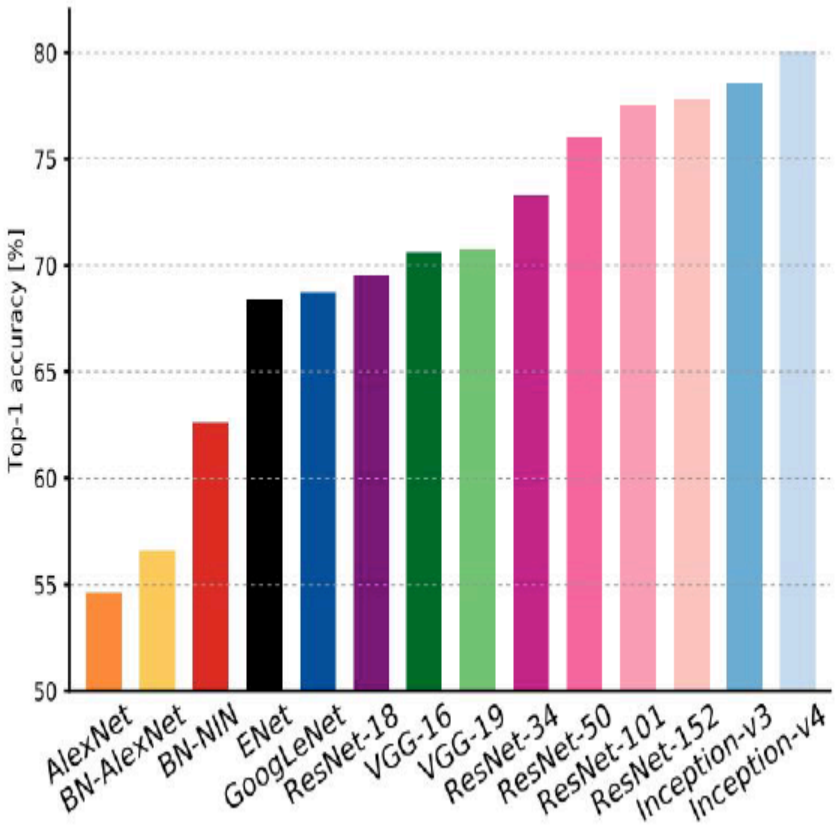


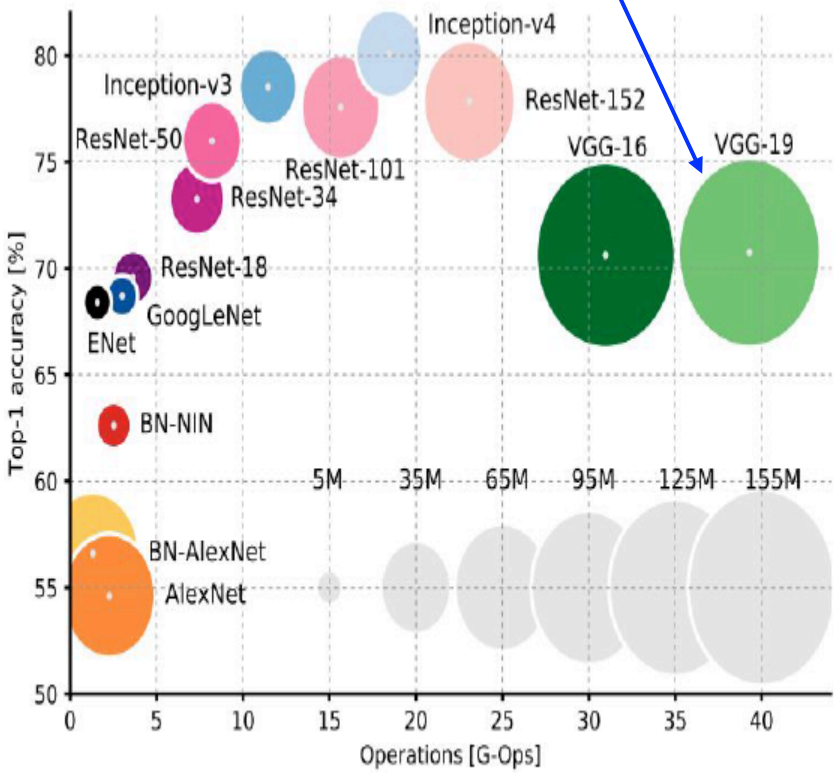


An Analysis of Deep Neural Network Models for Practical Applications, 2017.

Comparison

VGG: higher memory usage, more operations

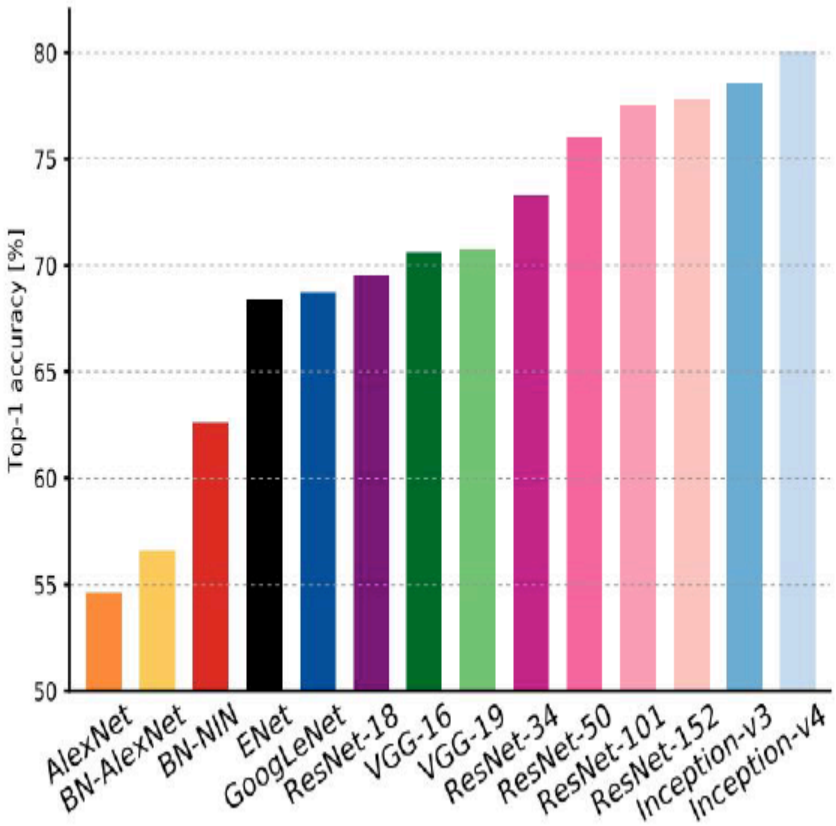


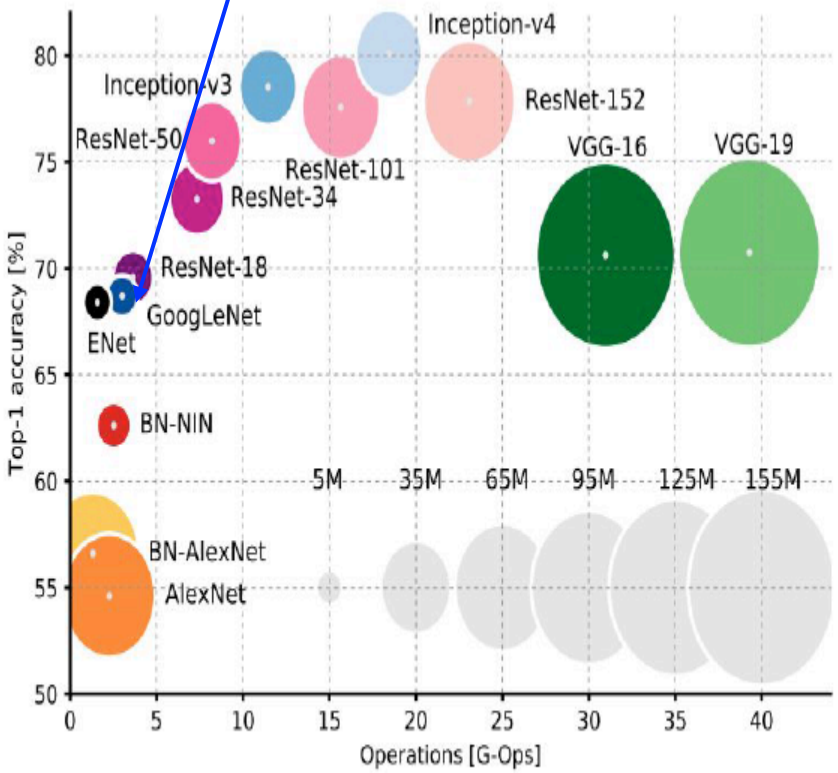


An Analysis of Deep Neural Network Models for Practical Applications, 2017.

Comparison

GoogLeNet: more efficient

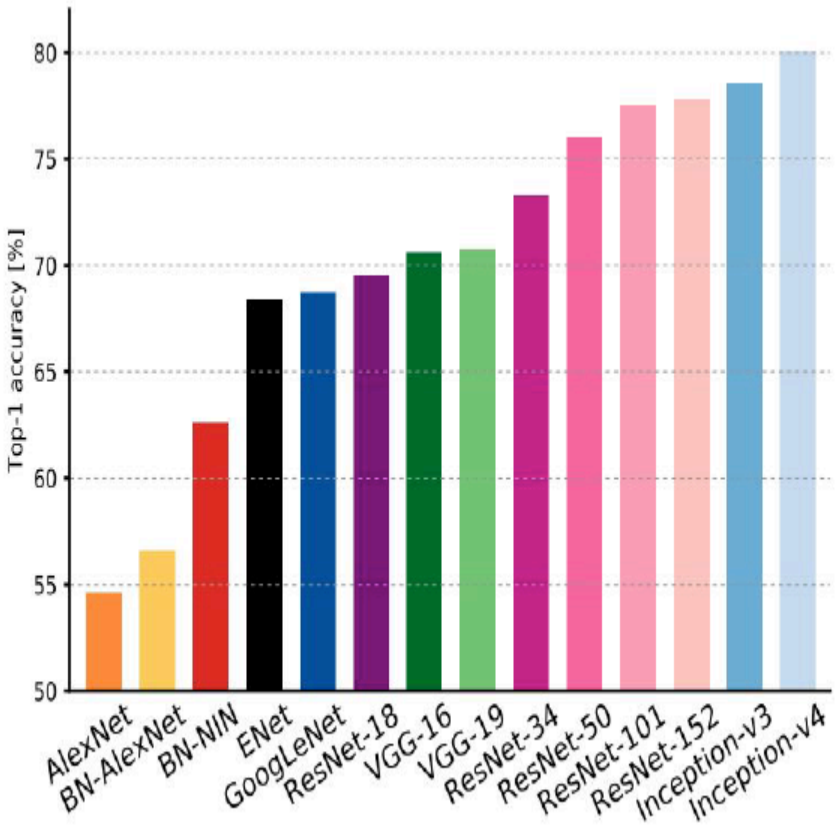


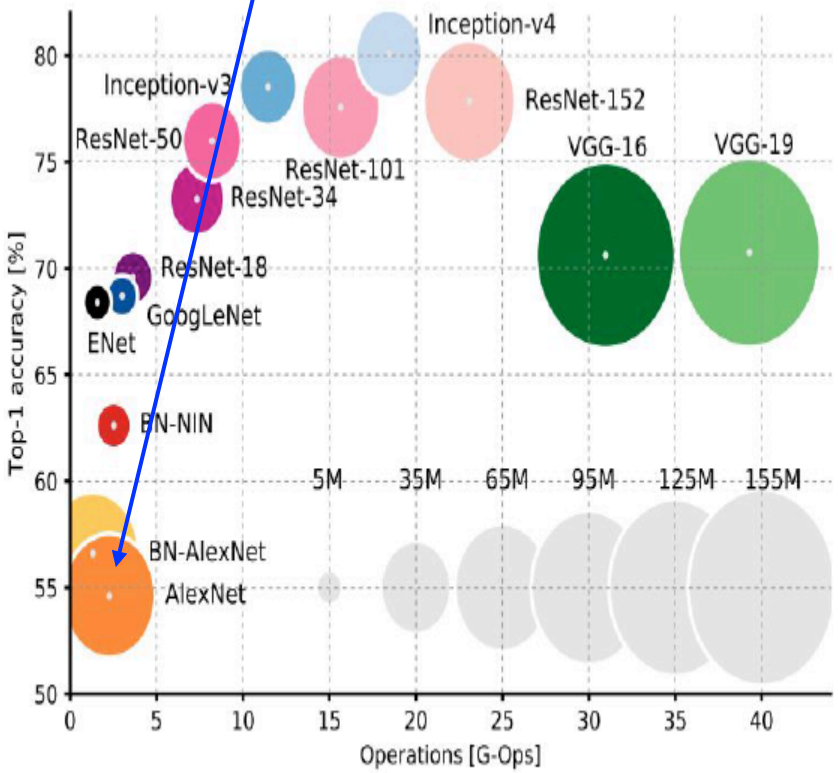


An Analysis of Deep Neural Network Models for Practical Applications, 2017.

Comparison

AlexNet: : low # of operations, but heavy on memory and low accuracy

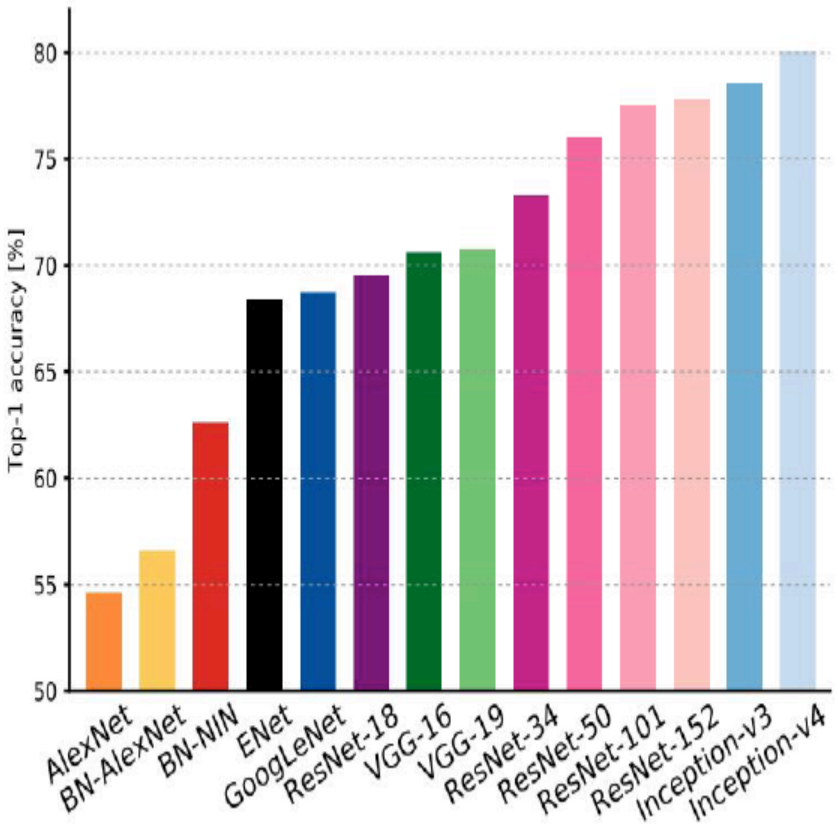


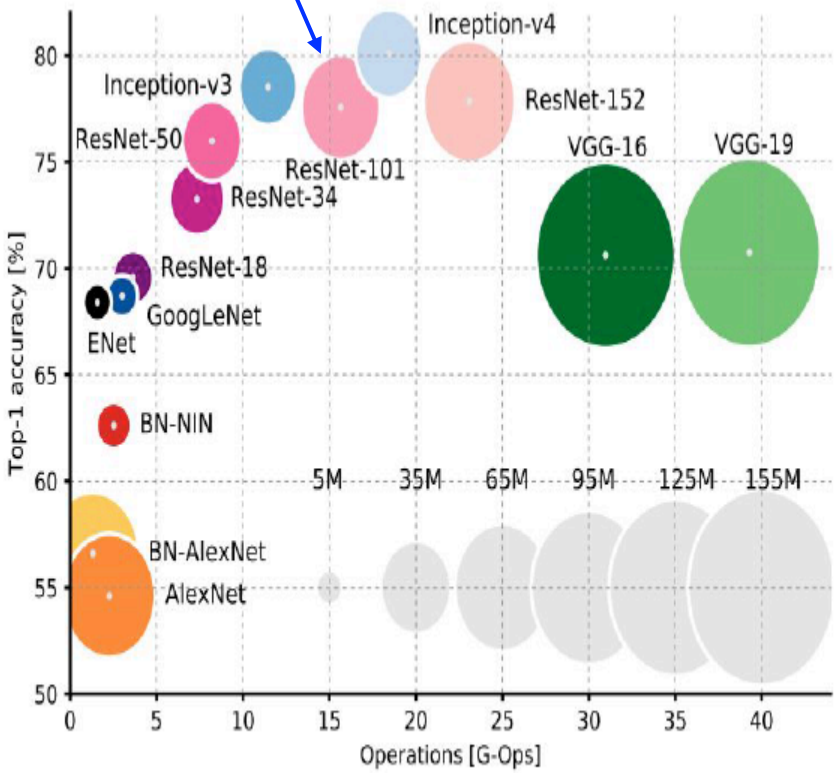


An Analysis of Deep Neural Network Models for Practical Applications, 2017.

Comparison

ResNet: moderate efficiency depending on model, high accuracy





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

References and acknowledgements

Some of these slides were inspired or adapted from courses and presentations given by Andrew Ng, Camila Laranjeira, Fei-Fei Li, Flávio Figueiredo, Hugo Oliveira, Jefersson dos Santos, Justin Johnson, Keiller Nogueira, Pedro Olmo, Renato Assunção, Serena Yeung.

Reference courses include *Machine Learning* and *Deep Learning* CS230 and CS231 from Stanford University, *Deep Learning* and *Hands-on Deep Learning* from UFMG, *Deep Learning* CS498 from Un. Of Illinois.