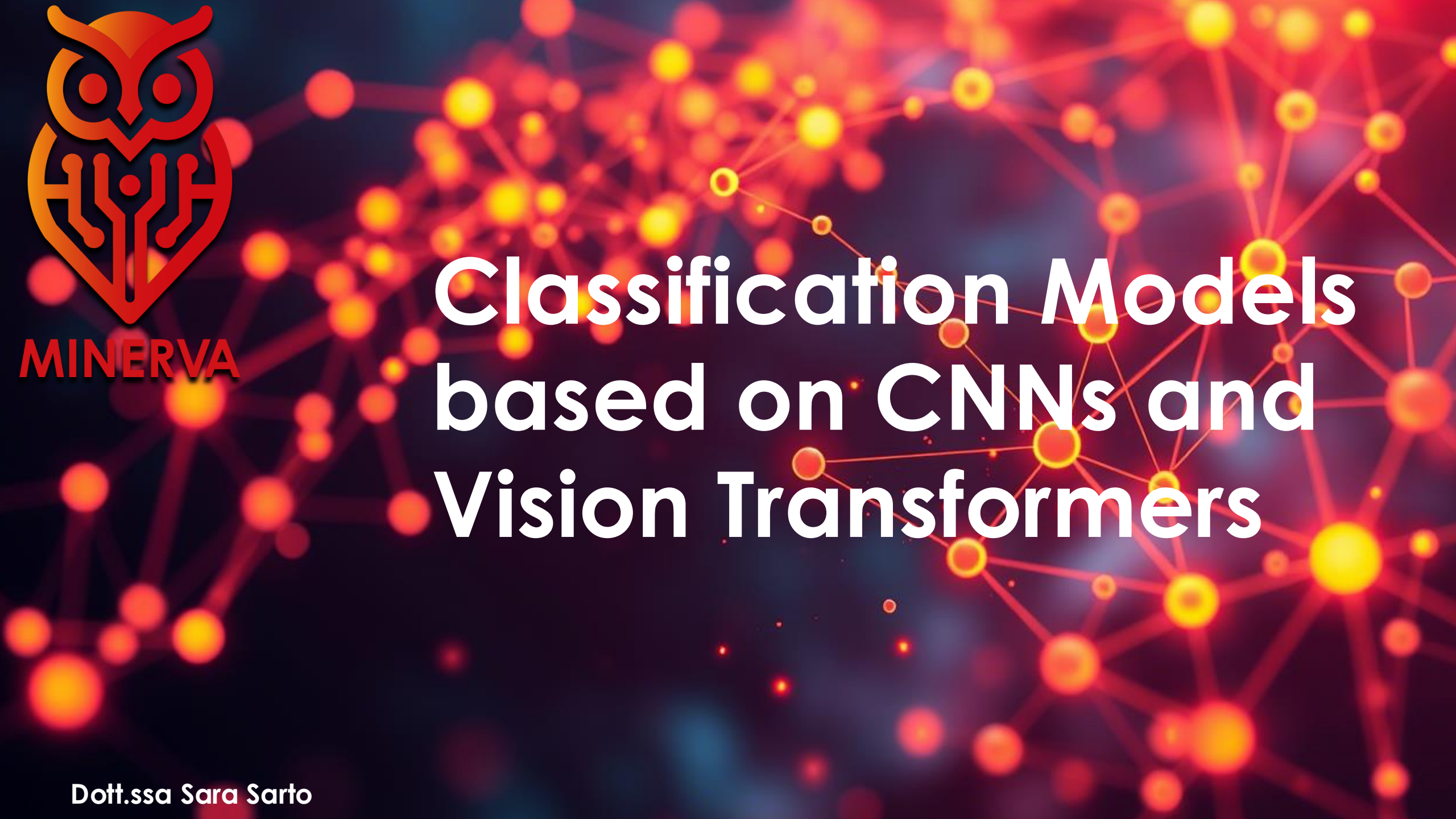



# Classification Models based on CNNs and Vision Transformers

Dott.ssa Sara Sarto


MINERVA
IMAGE
CLASSIFICATION

# Image Classification - Preliminaries

UNIMORE
UNIVERSITÀ DEGLI STUDI DI MODENA E REGGIO EMILIA

## Definition

Image Classification is a computer vision task where a model assigns **one label** to an entire image from a predefined set of categories.

## Goal

Given an input image → Predict its class.

## Key Idea

The model learns visual patterns (edges, textures, shapes, objects) using training data.

## Typical Approach

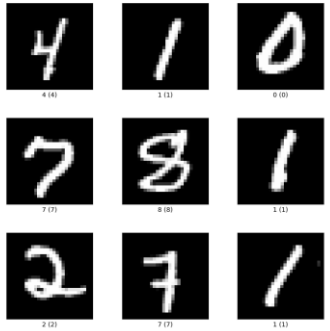
- Convolutional Neural Networks (CNNs)
- Vision Transformers (ViTs)
- Pretrained models + fine-tuning

# Famous Image Classification Datasets

UNIMORE
UNIVERSITÀ DEGLI STUDI DI MODENA E REGGIO EMILIA

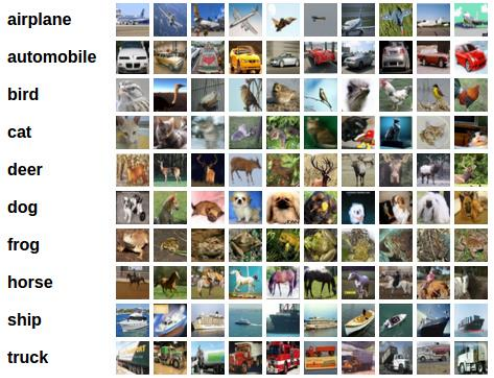
## MNIST (1998)

- 70,000 grayscale images
- Handwritten digits (0–9)
- Classic beginner dataset



## CIFAR-10 / CIFAR-100

- 60,000 color images (32×32)
- 10 or 100 object categories
- Small but challenging




## ImageNet

- 1+ million images
- 1,000 classes
- Benchmark that revolutionized deep learning (AlexNet, 2012)

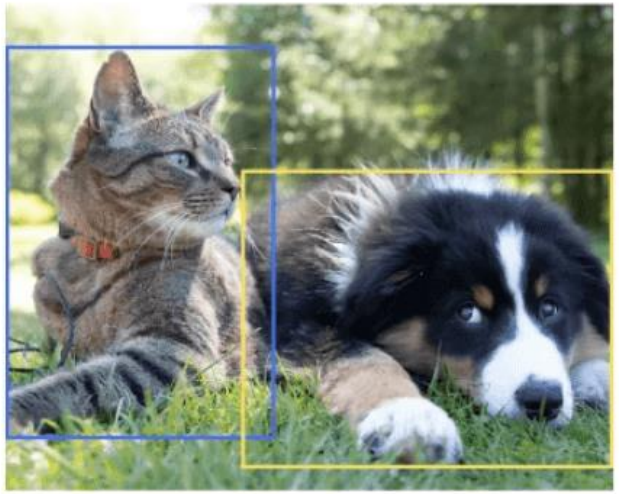
# Difference from Other Vision Tasks



Is this a cat?

Image Classification

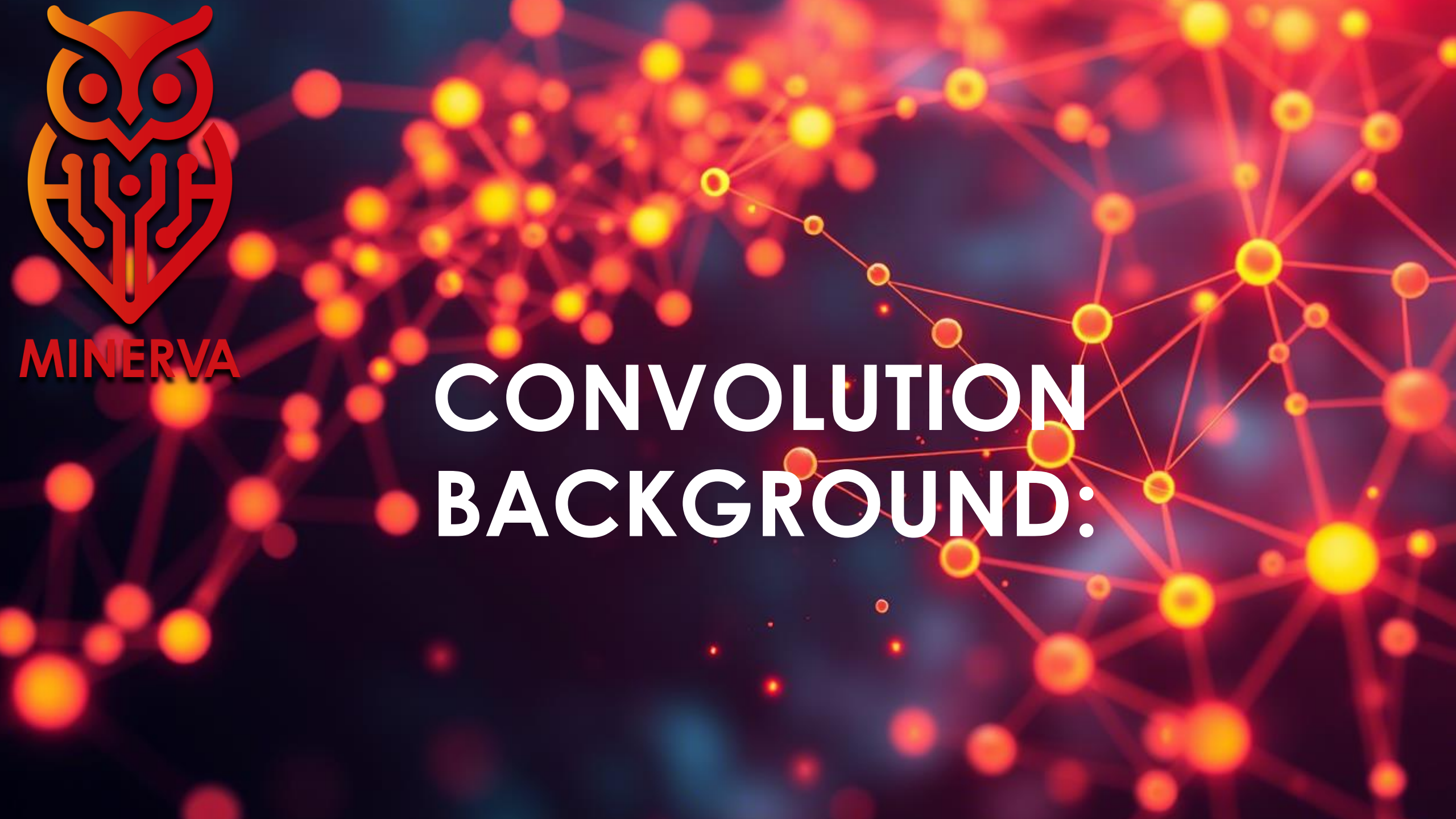
What is there in the image and where?



Object Detection

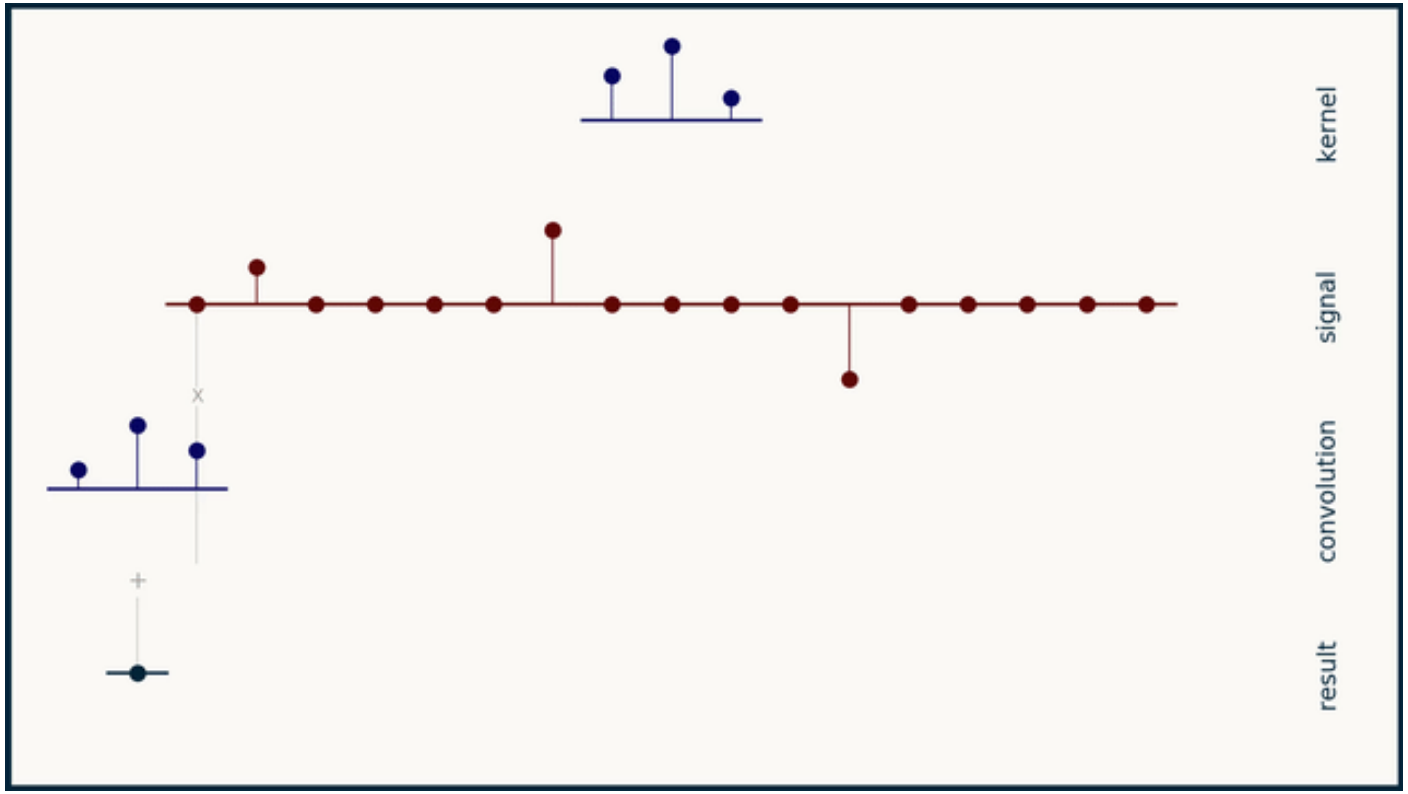
Which pixels belong to which object

Image Segmentation



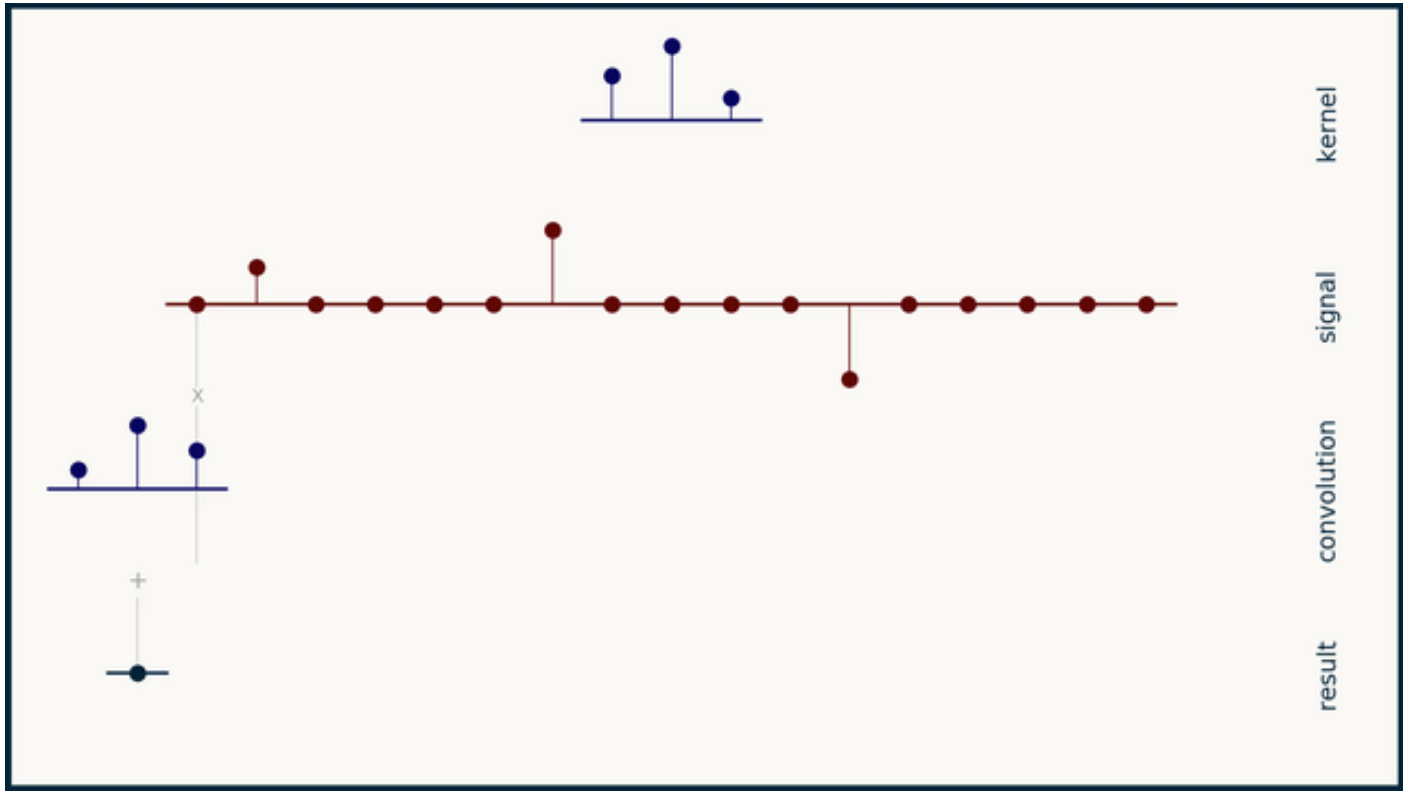

# CONVOLUTION BACKGROUND:

# Convolution 1-D



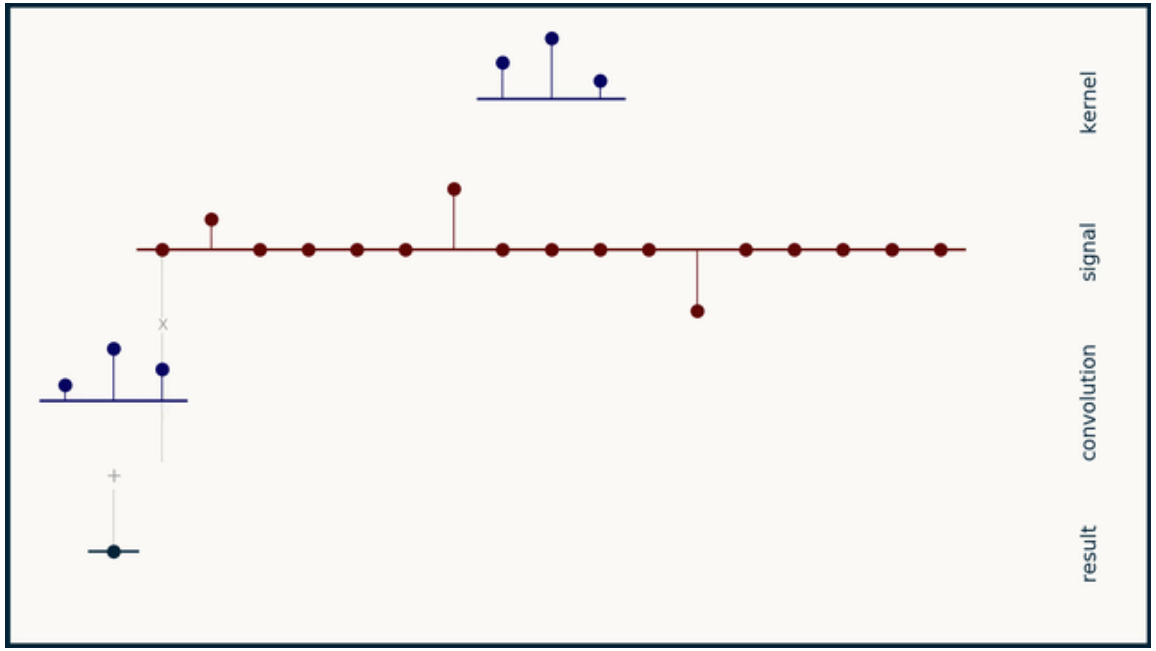

- The math behind convolution is a combination of multiplication and addition.
- Main components:
  - Input Signal (image or text)
  - Kernel (sliding window)

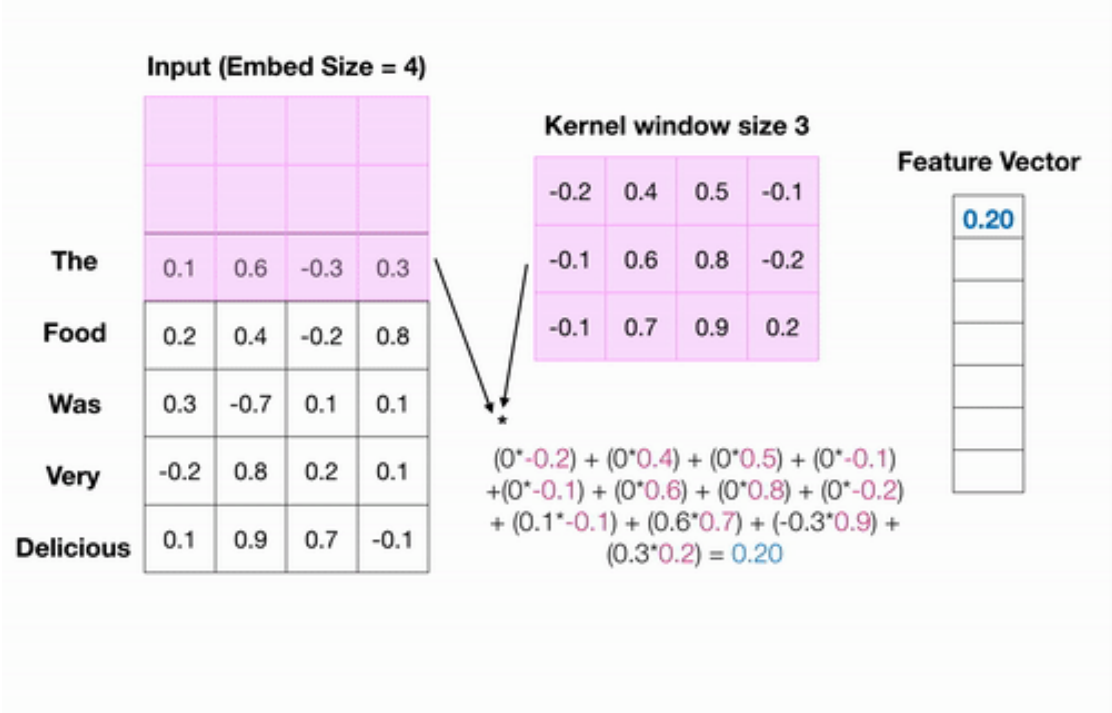
# Convolution 1-D




- Flip the kernel left to right
- Step the kernel along the signal one data point at a time
- At each position, calculate the dot product of the two
  - Multiply each pair of aligned values together
  - Add up those products
- The resulting sequence of dot products is the convolution of the kernel with the signal

# Convolution 1-D



# 2D Cross-Correlation on a 2D input

| | | | | | |
|---|---|---|---|---|---|
| $i_{11}$ | $i_{12}$ | $i_{13}$ | $i_{14}$ | $i_{15}$ | $i_{16}$ |
| $i_{21}$ | $i_{22}$ | $i_{23}$ | $i_{24}$ | $i_{25}$ | $i_{26}$ |
| $i_{31}$ | $i_{32}$ | $i_{33}$ | $i_{34}$ | $i_{35}$ | $i_{36}$ |
| $i_{41}$ | $i_{42}$ | $i_{43}$ | $i_{44}$ | $i_{45}$ | $i_{46}$ |
| $i_{51}$ | $i_{52}$ | $i_{53}$ | $i_{54}$ | $i_{55}$ | $i_{56}$ |
| $i_{61}$ | $i_{62}$ | $i_{63}$ | $i_{64}$ | $i_{65}$ | $i_{66}$ |

Input image

| | | |
|---|---|---|
| $k_{11}$ | $k_{12}$ | $k_{13}$ |
| $k_{21}$ | $k_{22}$ | $k_{23}$ |
| $k_{31}$ | $k_{32}$ | $k_{33}$ |

Kernel

| | | | |
|---|---|---|---|
| $o_{11}$ | $o_{12}$ | $o_{13}$ | $o_{14}$ |
| $o_{21}$ | $o_{22}$ | $o_{23}$ | $o_{24}$ |
| $o_{31}$ | $o_{32}$ | $o_{33}$ | $o_{34}$ |
| $o_{41}$ | $o_{42}$ | $o_{43}$ | $o_{44}$ |

Result

$O_{11} = i_{11} * k_{11} + i_{12} * k_{12} + i_{13} * k_{13} + i_{21} * k_{21} + i_{22} * k_{22} + i_{23} * k_{23} + i_{31} * k_{31} + i_{32} * k_{32} + i_{33} * k_{33} + b$

In short: place the kernel in the right position, element-wise multiplication, then sum and apply bias.

Shape of the input: (H, W), shape of the kernel (kH, kW)

Shape of the output: (H – (kH-1), W – (kW – 1))

# 2D Cross-Correlation on a 2D input

| | | | | | |
|---|---|---|---|---|---|
| $i_{11}$ | $i_{12}$ | $i_{13}$ | $i_{14}$ | $i_{15}$ | $i_{16}$ |
| $i_{21}$ | $i_{22}$ | $i_{23}$ | $i_{24}$ | $i_{25}$ | $i_{26}$ |
| $i_{31}$ | $i_{32}$ | $i_{33}$ | $i_{34}$ | $i_{35}$ | $i_{36}$ |
| $i_{41}$ | $i_{42}$ | $i_{43}$ | $i_{44}$ | $i_{45}$ | $i_{46}$ |
| $i_{51}$ | $i_{52}$ | $i_{53}$ | $i_{54}$ | $i_{55}$ | $i_{56}$ |
| $i_{61}$ | $i_{62}$ | $i_{63}$ | $i_{64}$ | $i_{65}$ | $i_{66}$ |

Input image

| | | |
|---|---|---|
| $k_{11}$ | $k_{12}$ | $k_{13}$ |
| $k_{21}$ | $k_{22}$ | $k_{23}$ |
| $k_{31}$ | $k_{32}$ | $k_{33}$ |

Kernel

| | | | |
|---|---|---|---|
| $o_{11}$ | $o_{12}$ | $o_{13}$ | $o_{14}$ |
| $o_{21}$ | $o_{22}$ | $o_{23}$ | $o_{24}$ |
| $o_{31}$ | $o_{32}$ | $o_{33}$ | $o_{34}$ |
| $o_{41}$ | $o_{42}$ | $o_{43}$ | $o_{44}$ |

Result

$O_{12} = i_{12} * k_{11} + i_{13} * k_{12} + i_{14} * k_{13} + i_{22} * k_{21} + i_{23} * k_{22} + i_{24} * k_{23} + i_{32} * k_{31} + i_{33} * k_{32} + i_{34} * k_{33} + b$

In short: place the kernel in the right position, element-wise multiplication, then sum and apply bias.

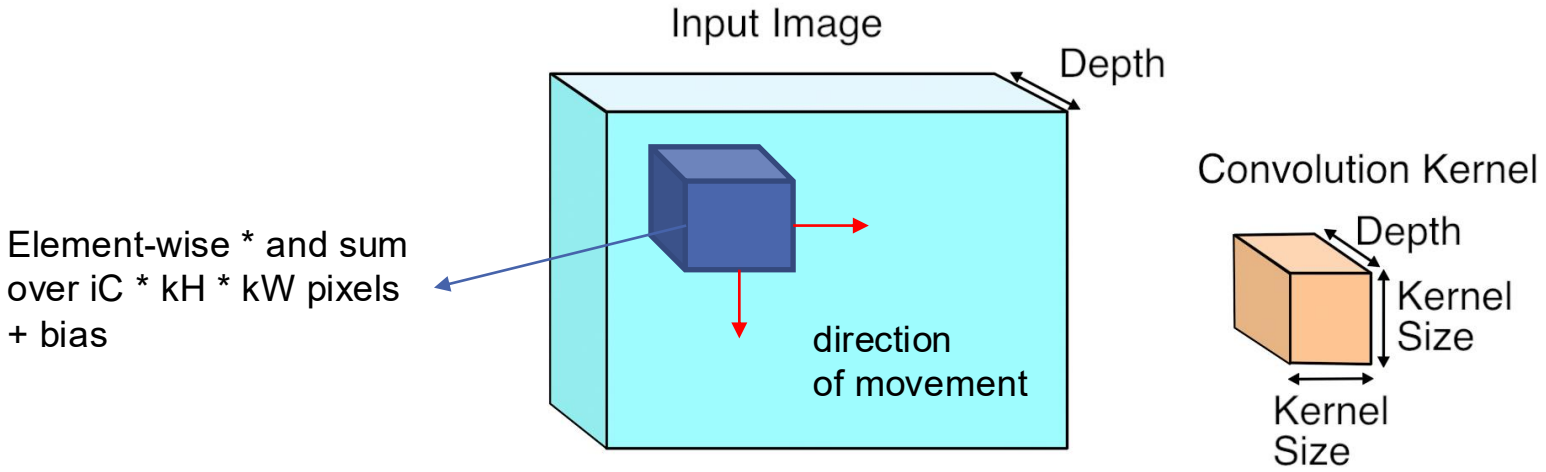
Shape of the input: (H, W), shape of the kernel (kH, kW)

Shape of the output: (H – (kH-1), W – (kW – 1))

# 2D Cross-Correlation on a 3D input

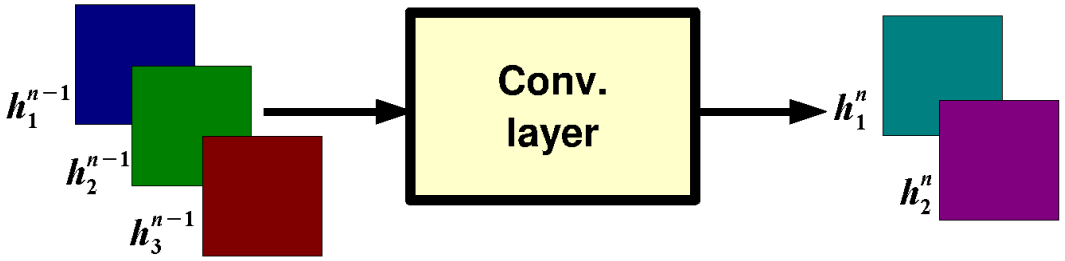
- With a (H, W) input, we have a (kH, kW) kernel
  - And we apply the usual correlation formula.
  - I.e.: element-wise *, then sum over a (kH, kW) matrix, plus bias.
  - Yes, the kernel might be rectangular! ☺

- What if we have a (iC, H, W) input instead of a (H, W) input?
  - Kernel will be (**iC**, kH, kW), bias will still be a scalar
  - And we apply the usual formula (element-wise *, then sum + bias) …
  - … over a (**iC**, kH, kW) tensor!
  - Note: this is STILL **2D "Convolution"**, because the kernel still moves over 2 axes!

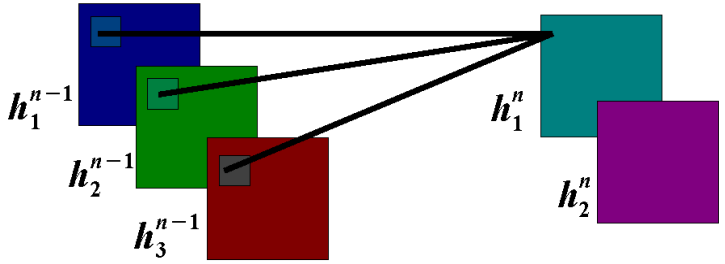


# Convolutional Layers in CNNs

- A convolutional layer usually takes iC input feature maps
- But also produces oC output maps!

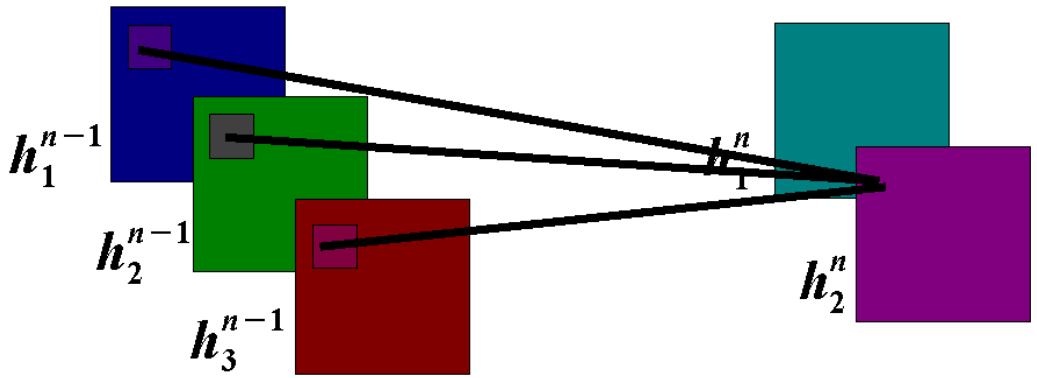



- Each output feature map considers all input feature map
- Basically, equivalent to performing the operation of the previous slides oC times with different kernels and biases.



# Convolutional Layers in CNNs

- The input shape is (iC, H, W)
- Kernel shape is (oC, iC, kH, kW), bias shape is (oC, ) → it's like having oC different kernels and biases
- Output shape is (oC, oH, oW) → it's like performing oC different cross-correlations




- Plus, we do this for a batch of n images in parallel, so:
  - The input shape is (n, iC, H, W)
  - Kernel shape is (oC, iC, kH, kW), bias shape is (oC, ) → same kernel and biases for all images! ☺
  - Output shape is (n, oC, oH, oW), where oH and oW are computed as before

# Convolutional Layers in CNNs

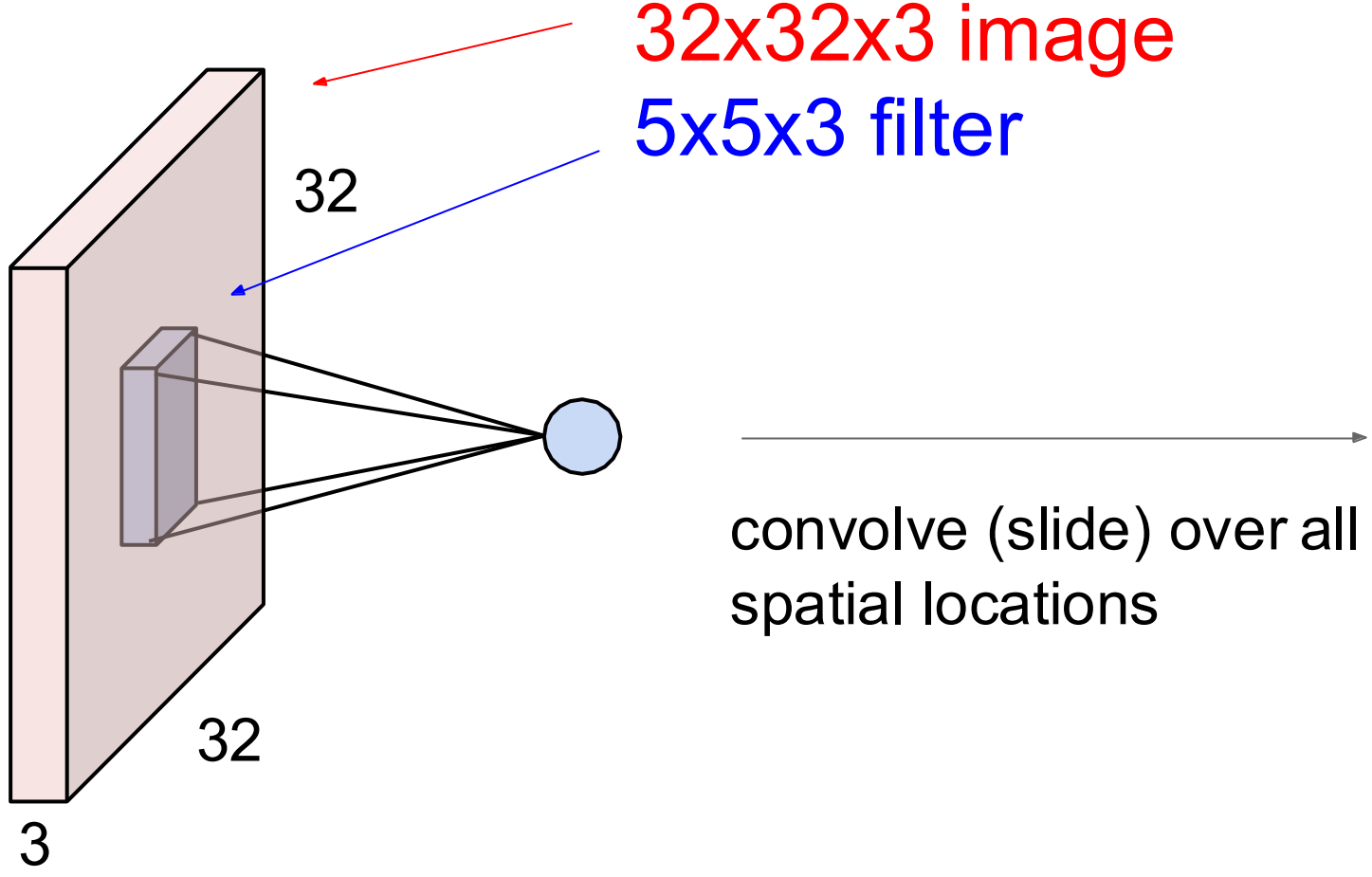
UNIMORE
UNIVERSITÀ DEGLI STUDI DI MODENA E REGGIO EMILIA

- Other “parameters”:
  - **Stride**: controls the stride for the cross-correlation (might be different for the two axes on which the kernel slides).
    - We assume 1, if we don’t specify otherwise.
  - **Padding**: the amount of implicit paddings on both sides for padding number of points for each dimension.
    - We assume 0, if we don’t specify otherwise.
    - A typical choice is k-1 (where k is the kernel size), so that the output shape matches the input shape on spatial axes
    - Usually, we pad with 0s, even if other options are on the table (e.g. reflection, constant padding)

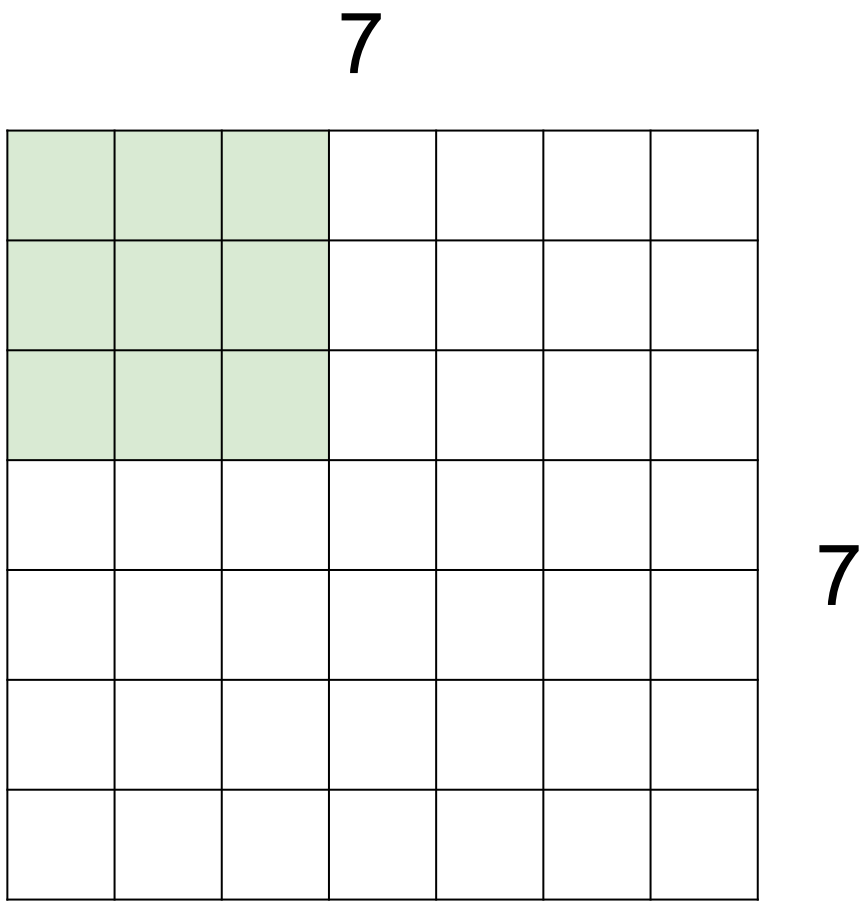
    https://ezyang.github.io/convolution-visualizer/

# A closer look at spatial dimensions:



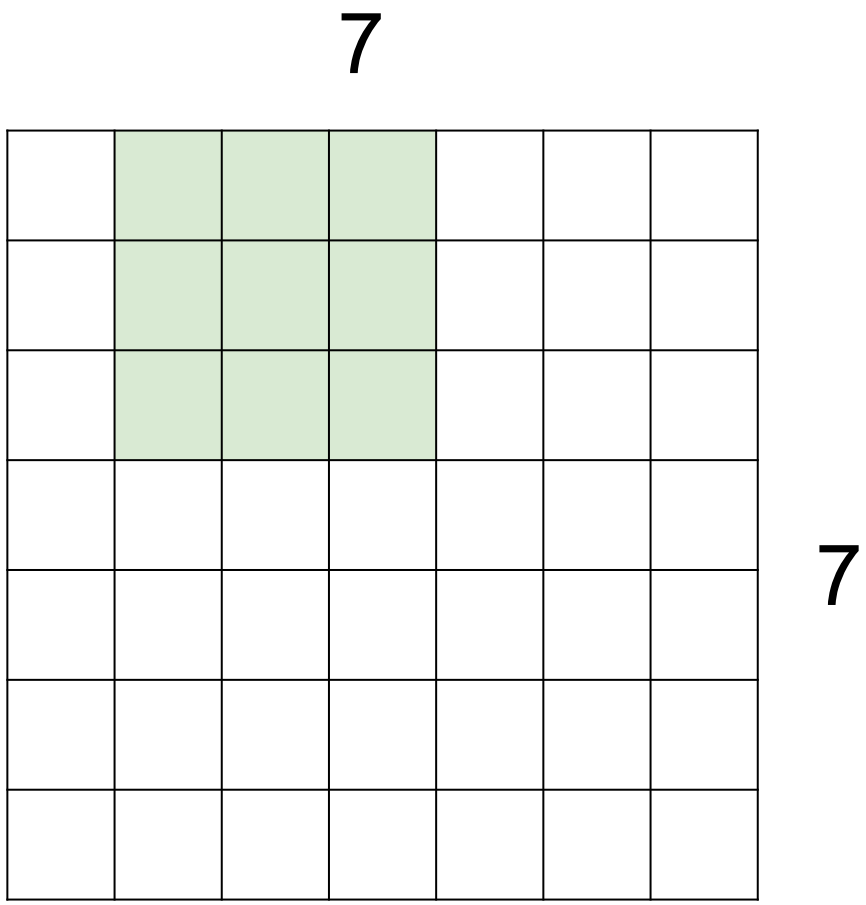


# A closer look at spatial dimensions:



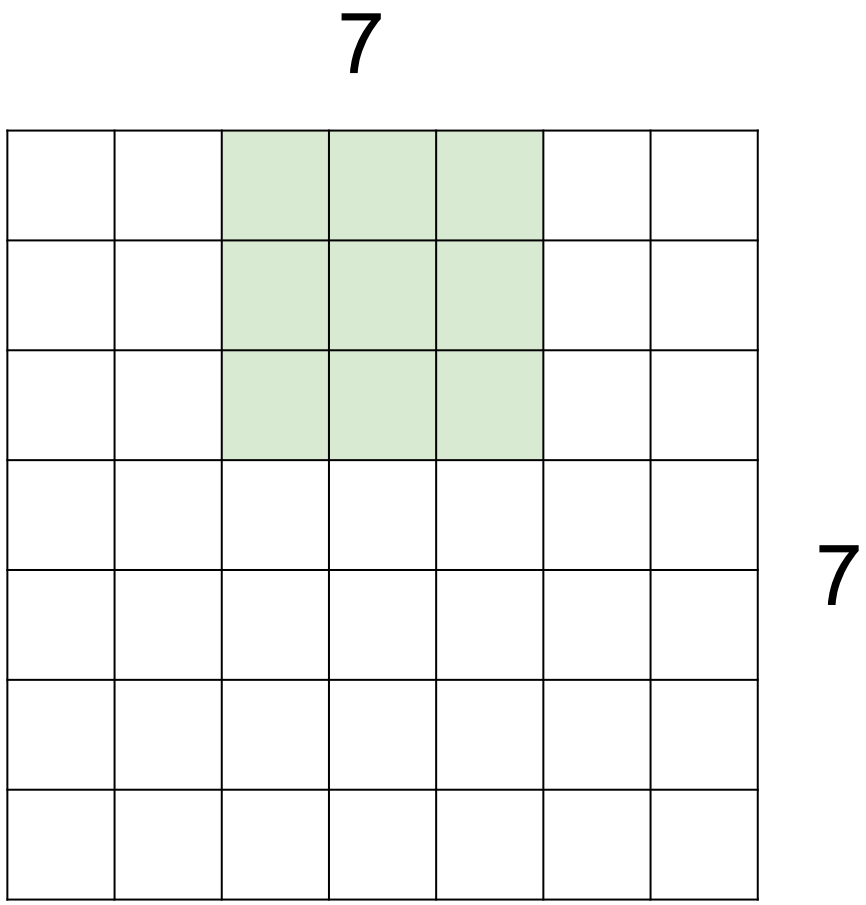

7x7 input (spatially)
assume 3x3 filter

# A closer look at spatial dimensions:



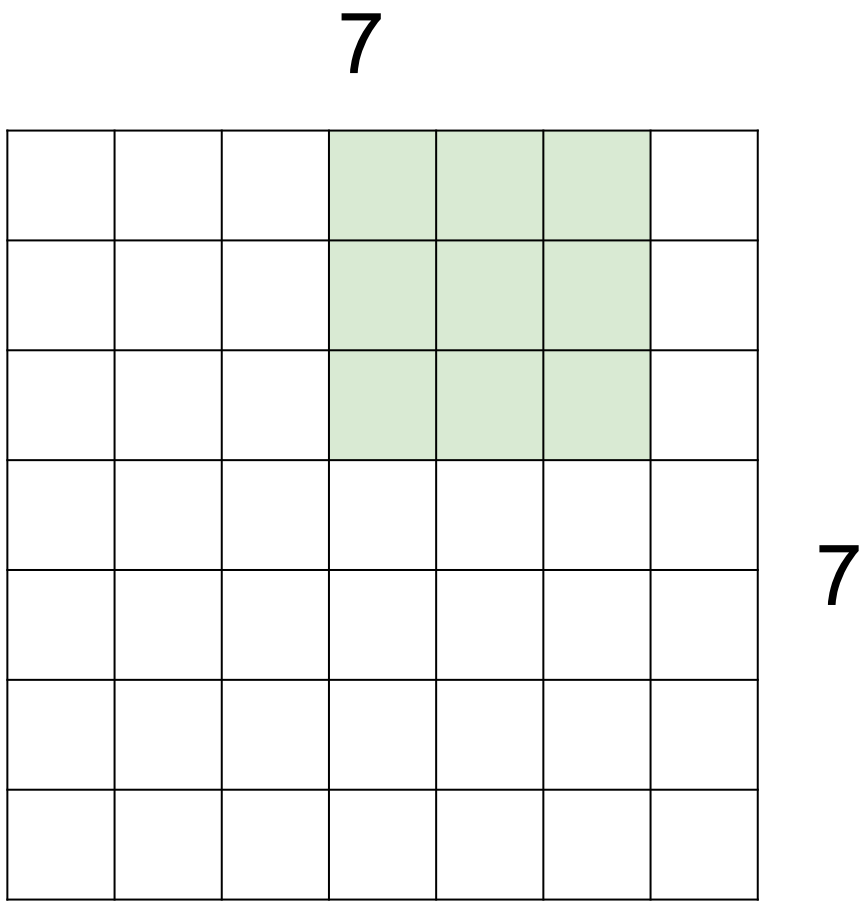

7x7 input (spatially)
assume 3x3 filter

# A closer look at spatial dimensions:



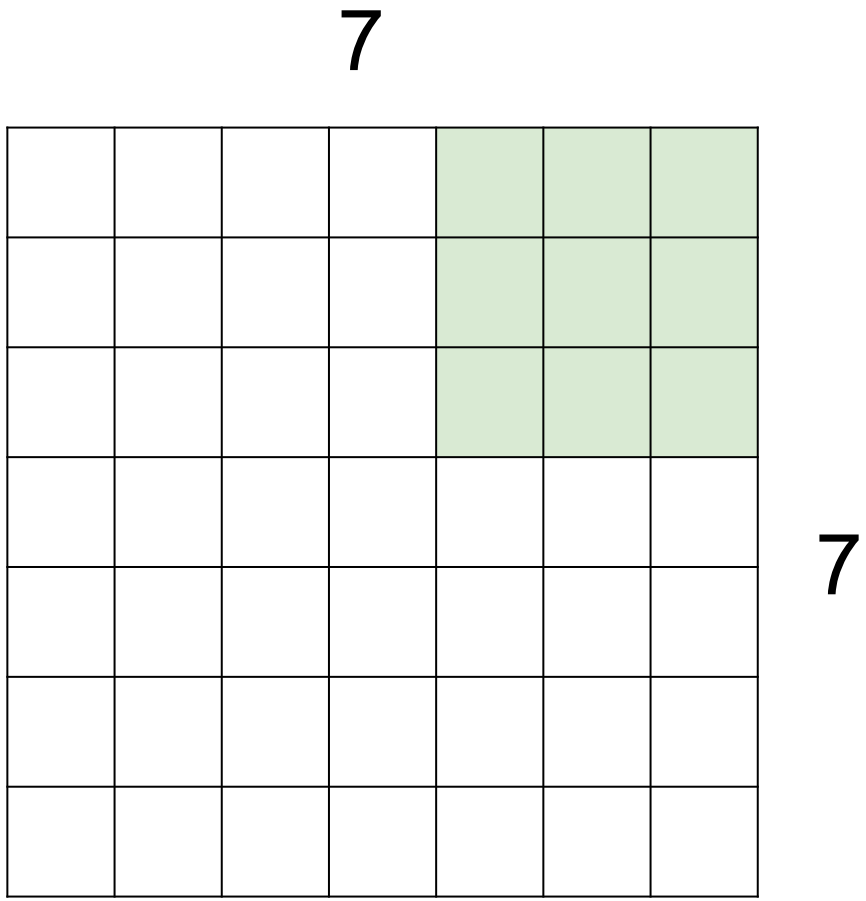

7x7 input (spatially)
assume 3x3 filter

# A closer look at spatial dimensions:




7x7 input (spatially)
assume 3x3 filter

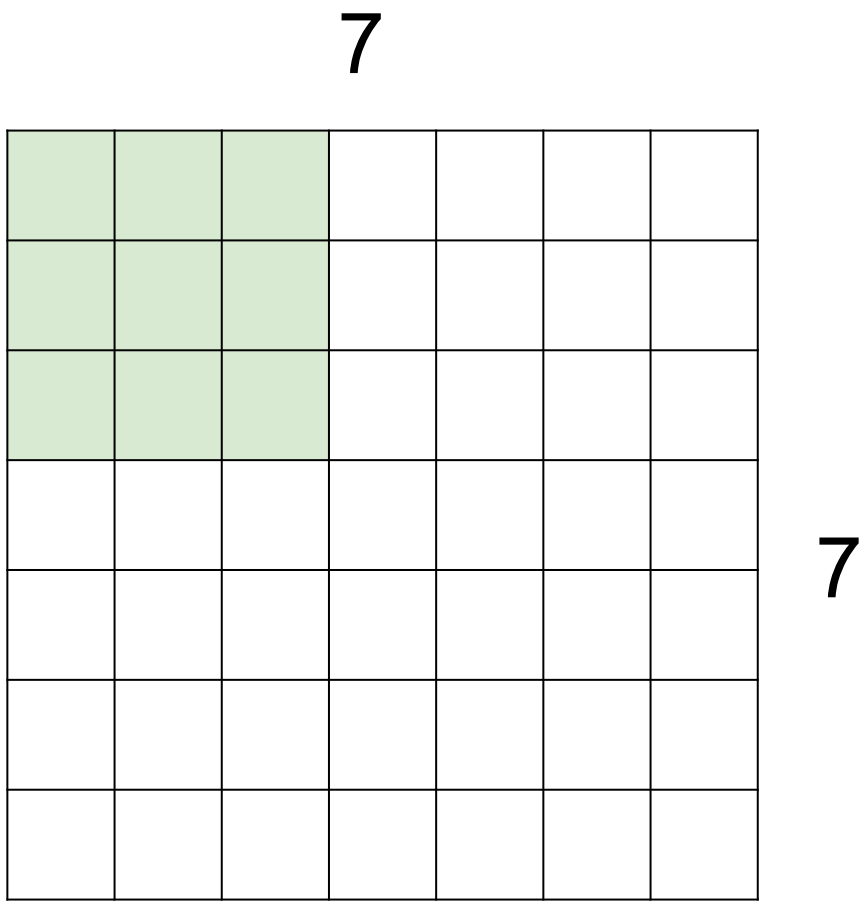
# A closer look at spatial dimensions:




7x7 input (spatially)
assume 3x3 filter

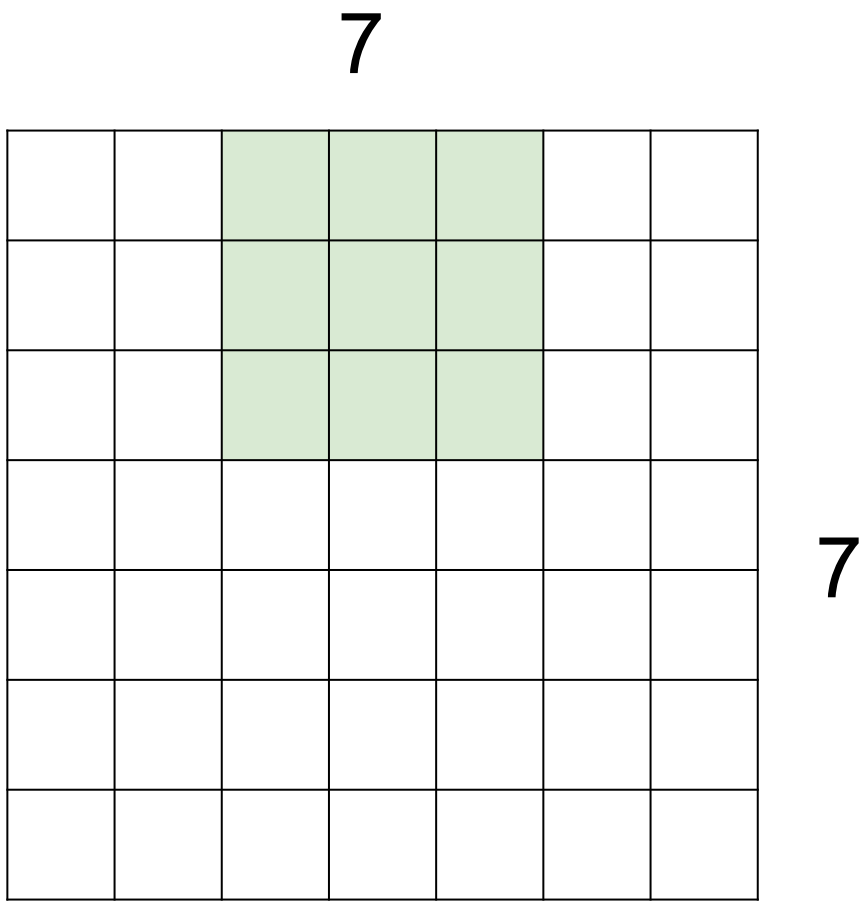
**=> 5x5 output**

# A closer look at spatial dimensions:



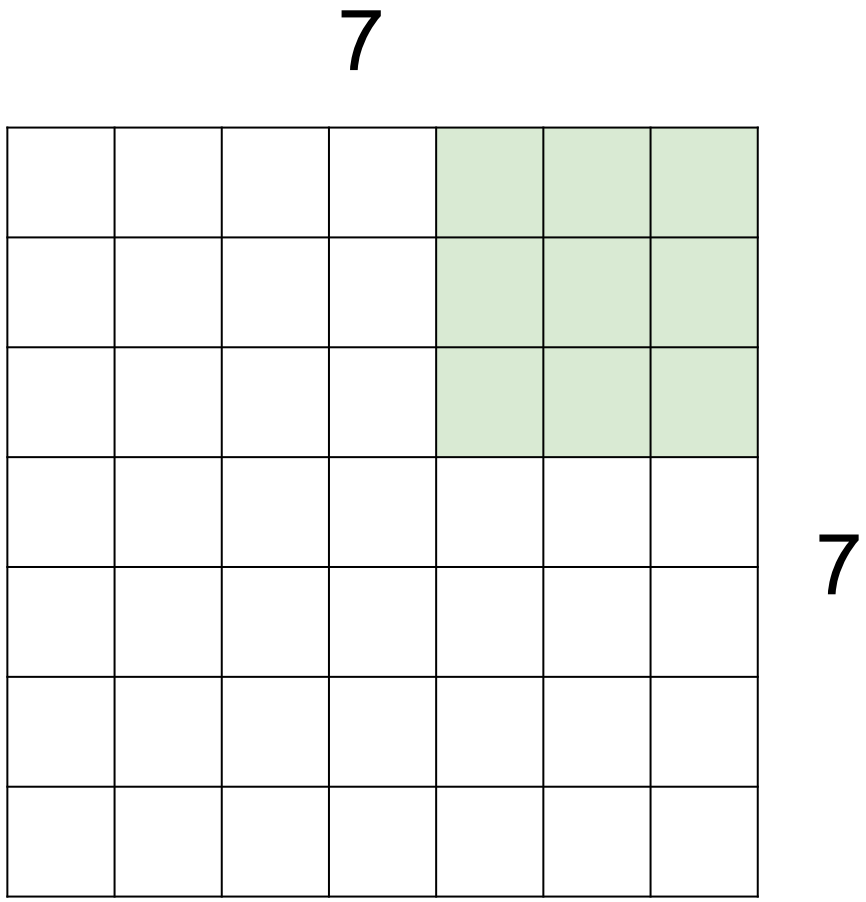

7x7 input (spatially)
assume 3x3 filter
applied **with stride 2**

# A closer look at spatial dimensions:

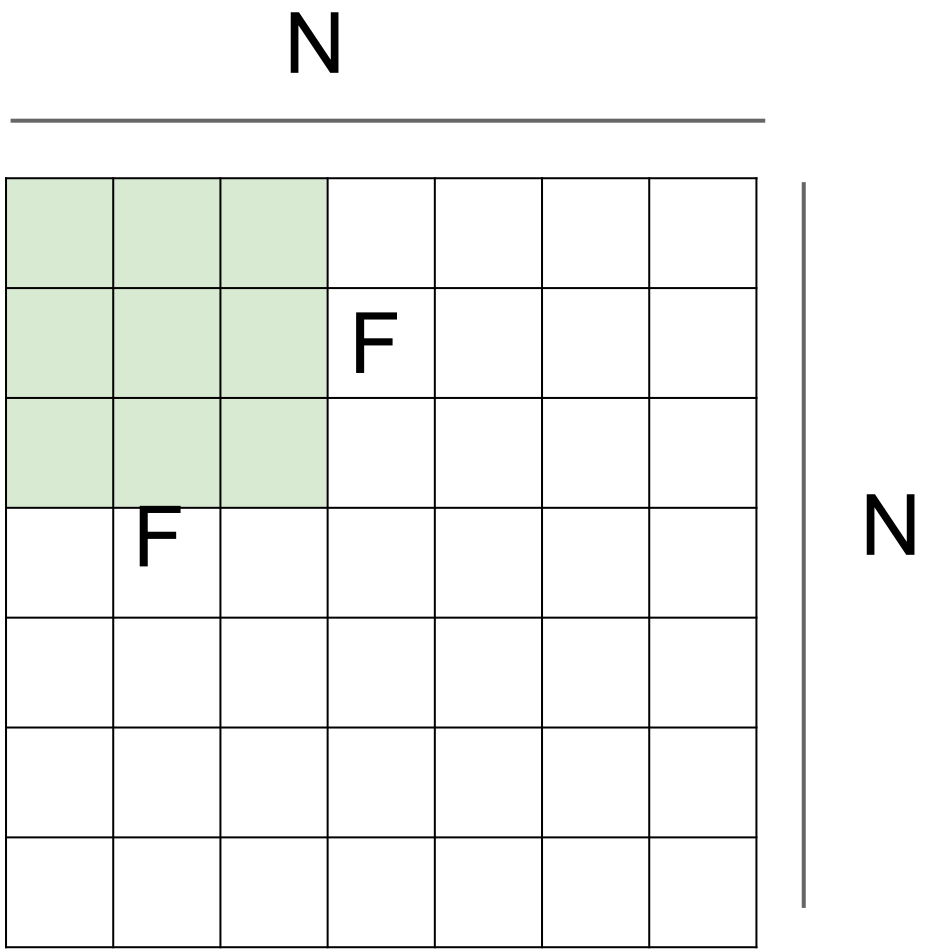



7x7 input (spatially)
assume 3x3 filter
applied **with stride 2**

# A closer look at spatial dimensions:




7x7 input (spatially)
assume 3x3 filter
applied **with stride 2**
**=> 3x3 output!**




Output size:

**(N - F) / stride + 1**

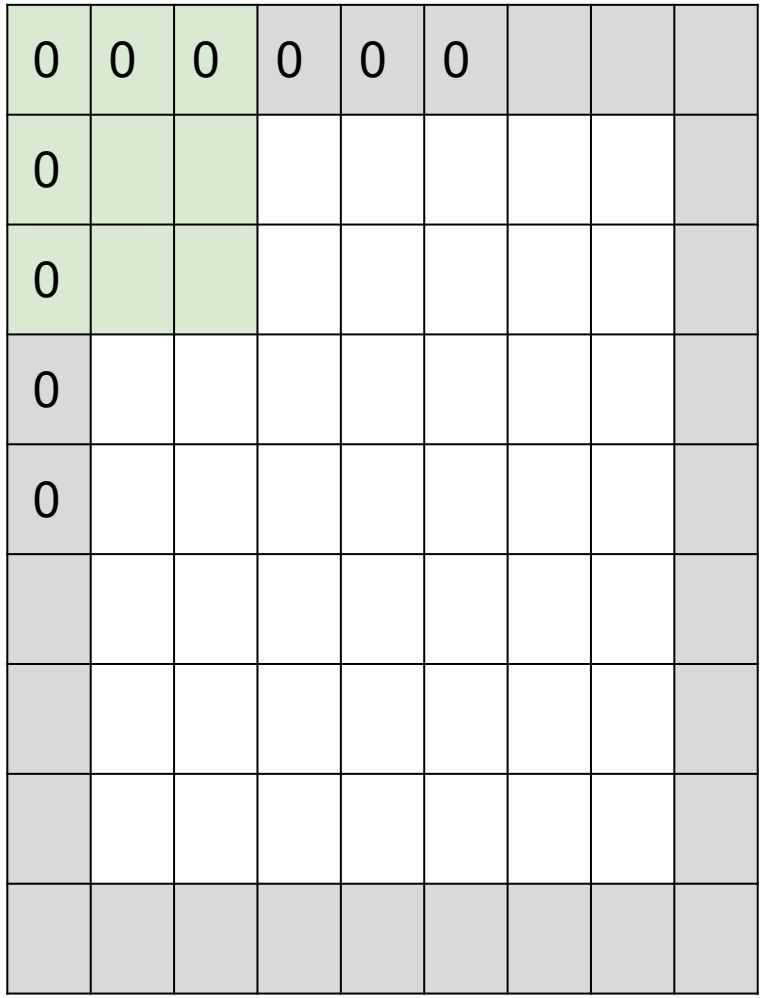
e.g. N = 7, F = 3:

stride 1 => (7 - 3)/1 + 1 = 5

stride 2 => (7 - 3)/2 + 1 = 3

# It’s common to zero pad the border



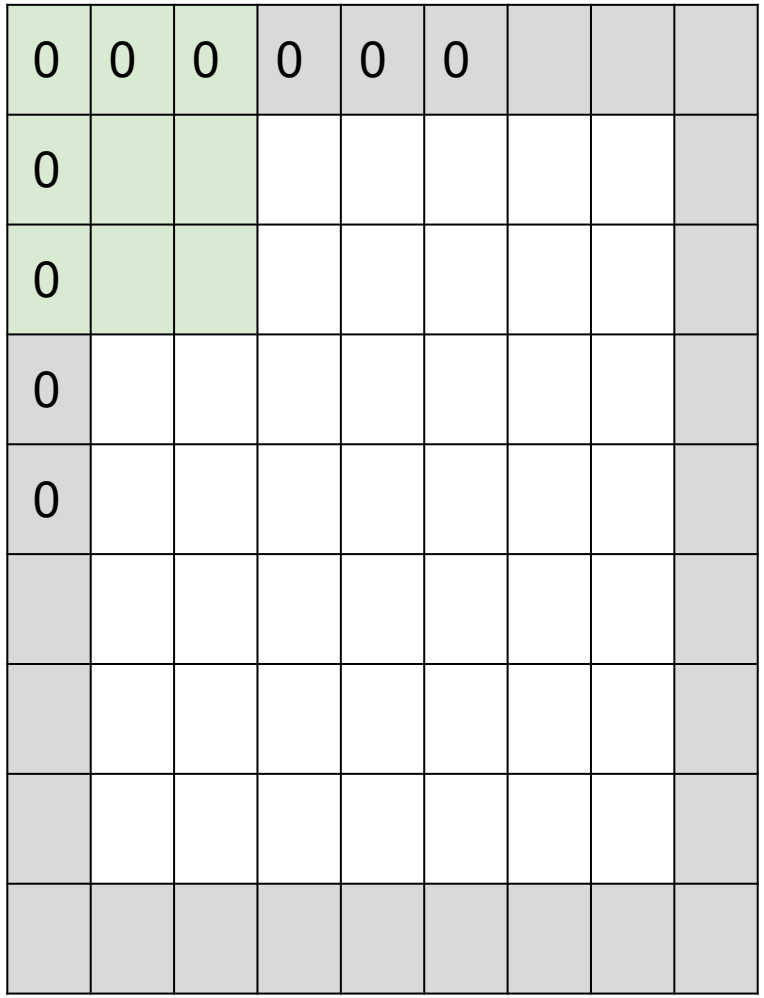



e.g. input 7x7
**3x3** filter, applied with **stride 1**
**pad with 1 pixel** border => what is the output?

(recall:)
(N - F) / stride + 1

# In practice: Common to zero pad the border

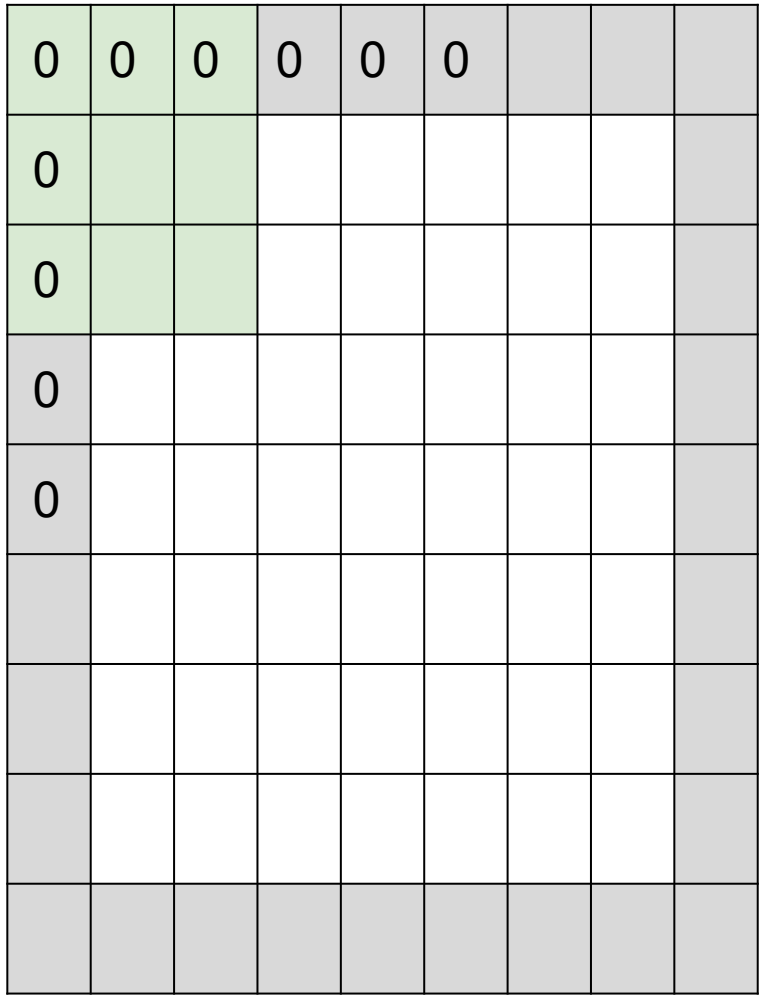





e.g. input 7x7
**3x3** filter, applied with **stride 1**
**pad with 1 pixel** border => what is the output?

**7x7 output!**

# In practice: Common to zero pad the border

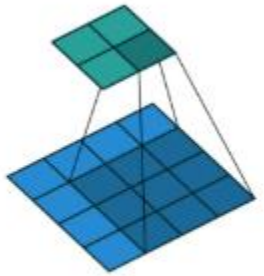
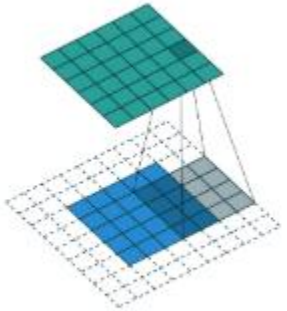
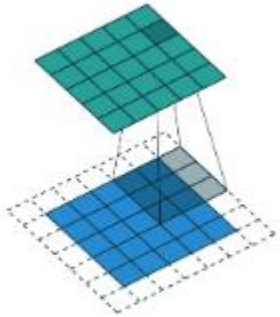
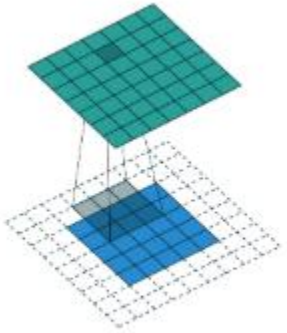
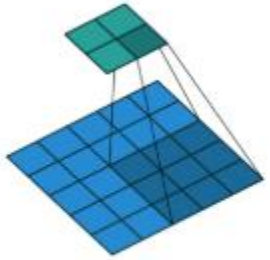
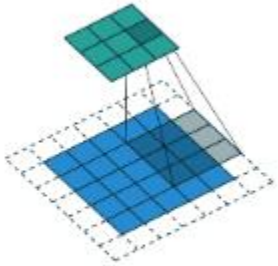
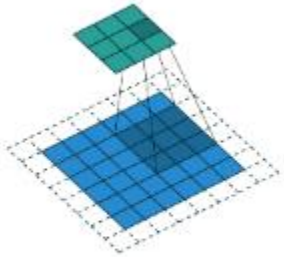



e.g. input 7x7
**3x3** filter, applied with **stride 1**
**pad with 1 pixel** border => what is the output?

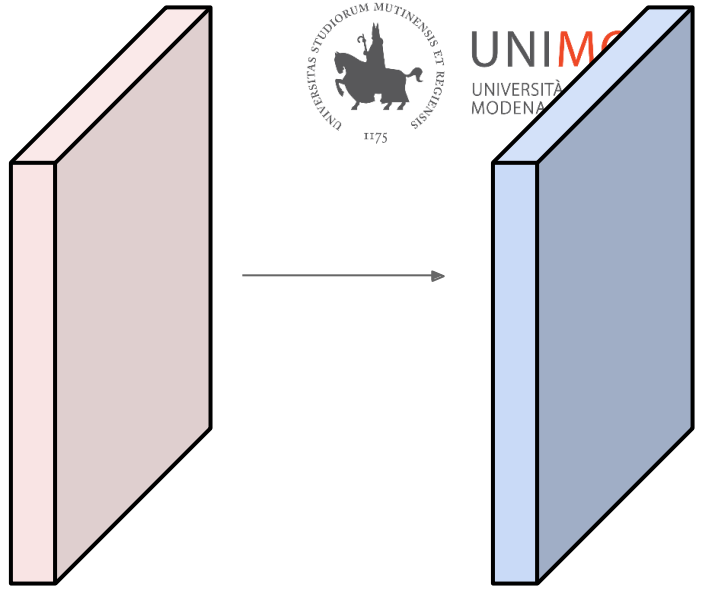
**7x7 output!**
in general, common to see CONV layers with stride 1, filters of size FxF, and zero-padding with (F-1)/2. (will preserve size spatially)
e.g. F = 3 => zero pad with 1
F = 5 => zero pad with 2
F = 7 => zero pad with 3

# Examples



| | | | |
|---|---|---|---|
|  |  |  |  |
| No padding, no strides | Arbitrary padding, no strides | Half padding, no strides | Full padding, no strides |
|  |  |  | |
| No padding, strides | Padding, strides | Padding, strides (odd) | |

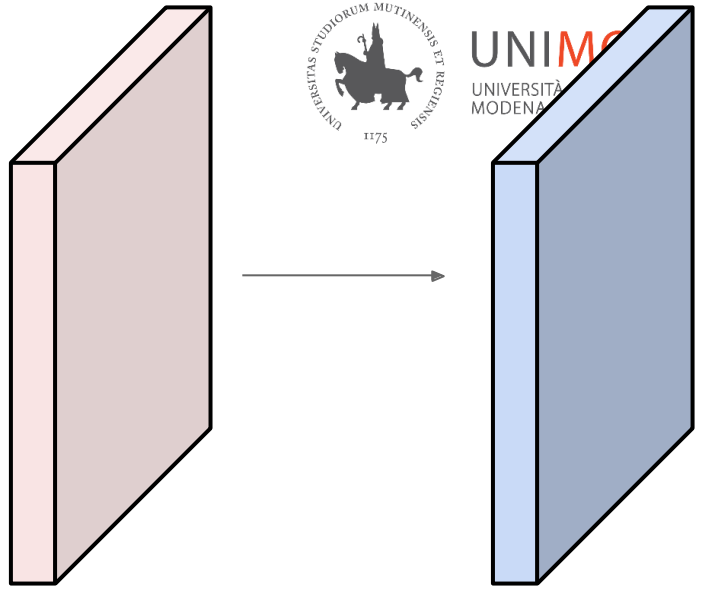
# Examples time:



Input volume: **32x32x3**
10 5x5 filters with stride 1, pad 2

Output volume size: ?

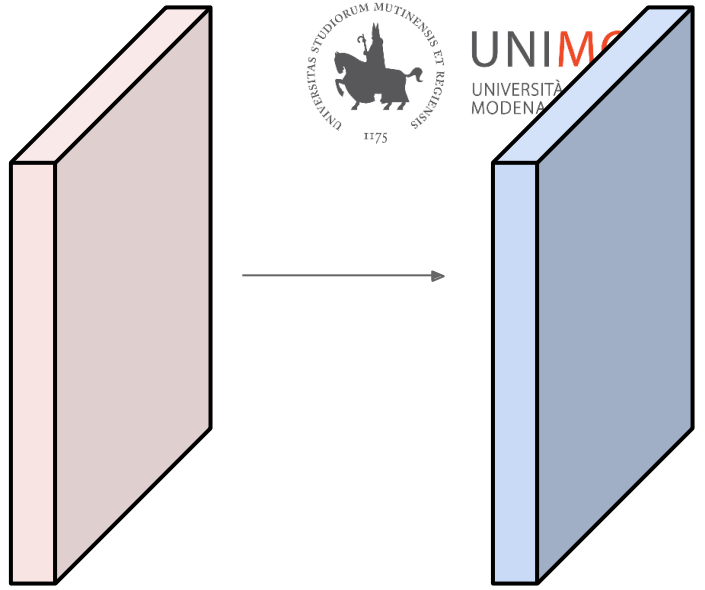
# Examples time:



Input volume: **32x32x3**
10 5x5 filters with stride 1, pad 2

Output volume size:
(32+2*2-5)/1+1 = 32 spatially, so
**32x32x10**

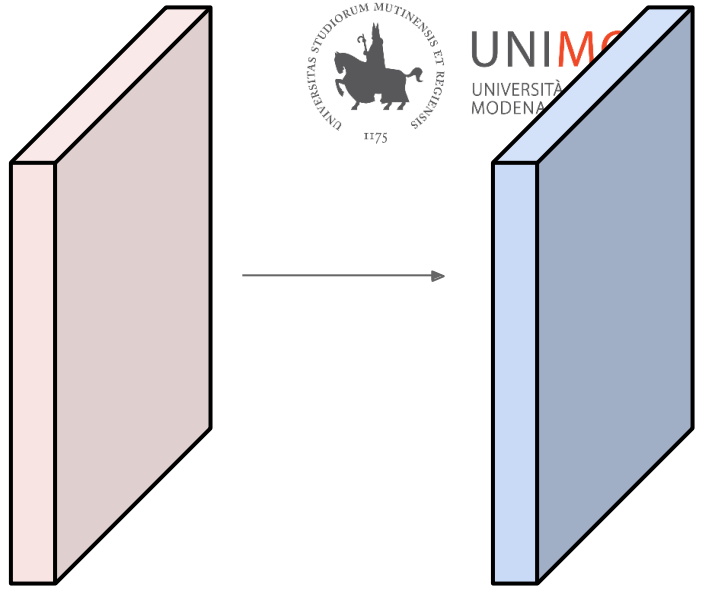
# Examples time:



Input volume: **32x32x3**
10 5x5 filters with stride 1, pad 2

Number of parameters in this layer?

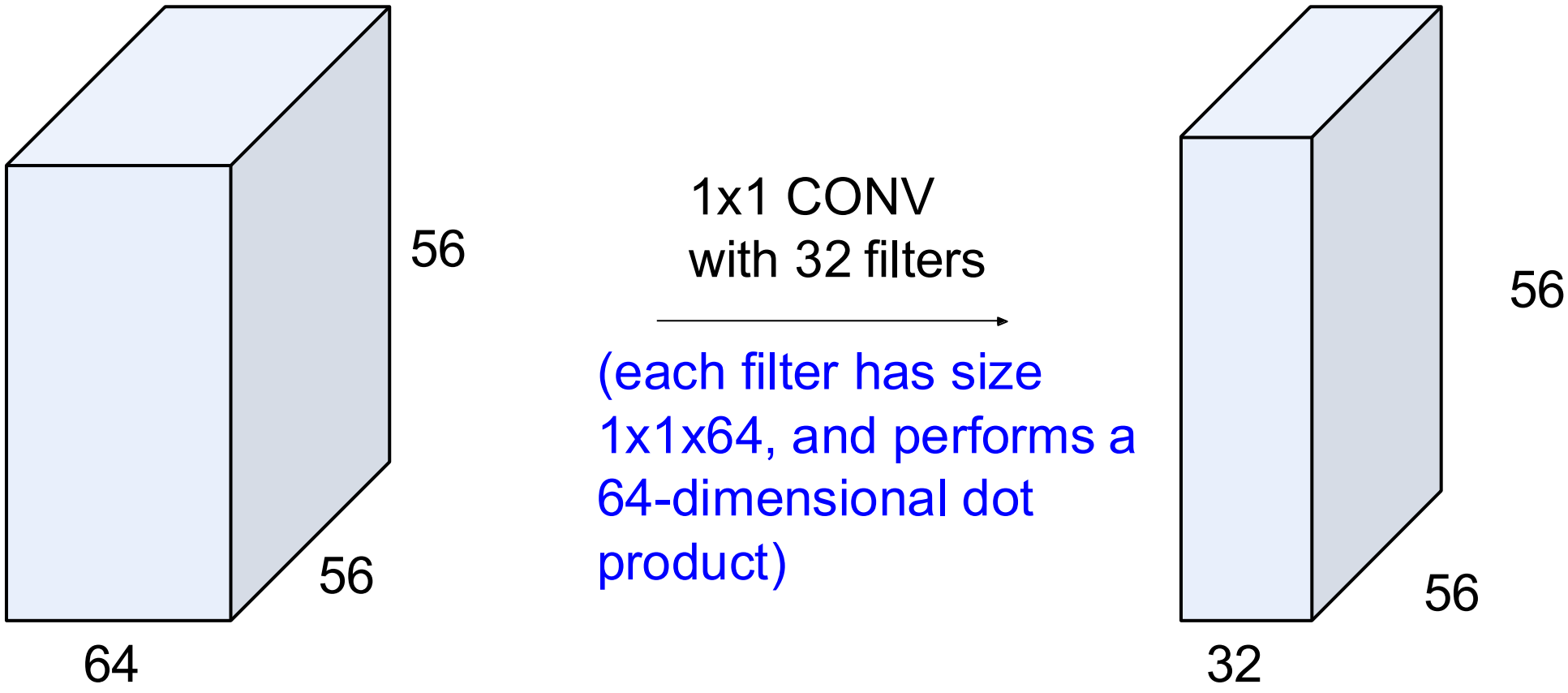
# Examples time:



Input volume: **32x32x3**
10 5x5 filters with stride 1, pad 2

Number of parameters in this layer?
each filter has 5*5*3 + 1 = 76 params (+1 for bias)
=> 76*10 = **760**

# (btw, 1x1 convolution layers make perfect sense)



# Dilation (à trous)

- An extension of the regular 2D convolution, in which the kernel is «dilated»
  - Bigger receptive field, same number of parameters ☺
- Given a dilation factor, it is defined as follows:

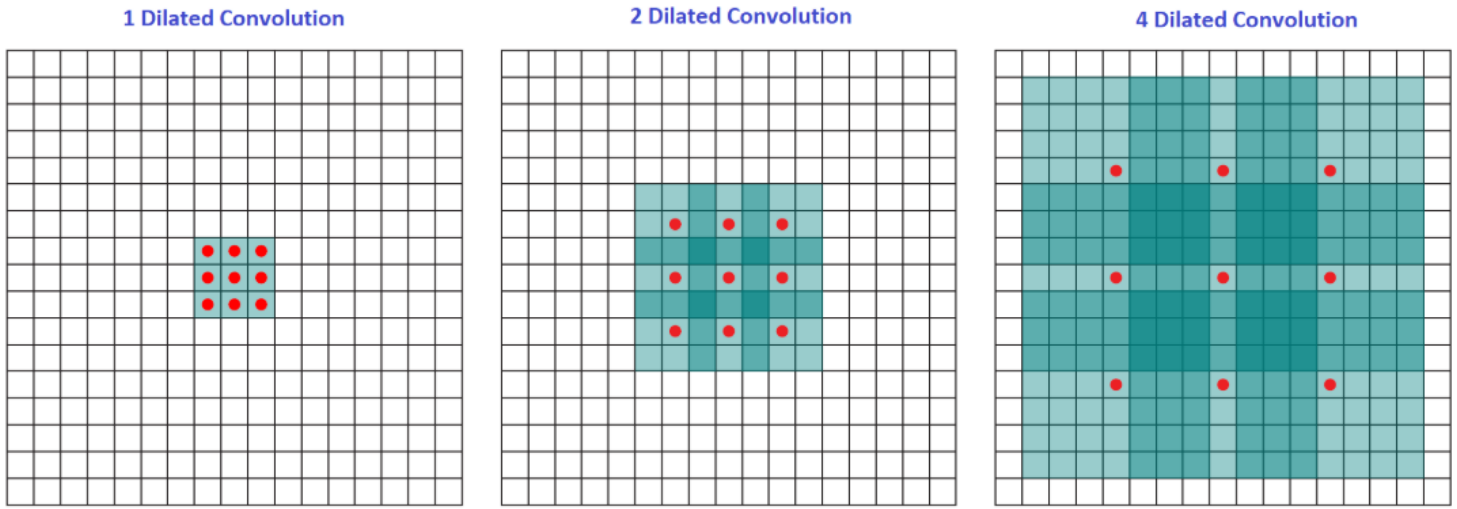
We now generalize this operator. Let $l$ be a dilation factor and let $*_l$ be defined as

$$(F *_l k)(\mathbf{p}) = \sum_{\mathbf{s}+l\mathbf{t}=\mathbf{p}} F(\mathbf{s})\, k(\mathbf{t}). \qquad (2)$$

We will refer to $*_l$ as a dilated convolution or an $l$-dilated convolution. The familiar discrete convolution $*$ is simply the 1-dilated convolution.

- *If we apply a sequence of increasingly dilated Convolutional layers, we get an exponentially growing receptive field*. In the figure below, red dots are the application points of increasingly dilated kernels at each layer, green areas represent the exponentially increasing receptive field of the sequence of convolutions.

- https://ezyang.github.io/convolution-visualizer/



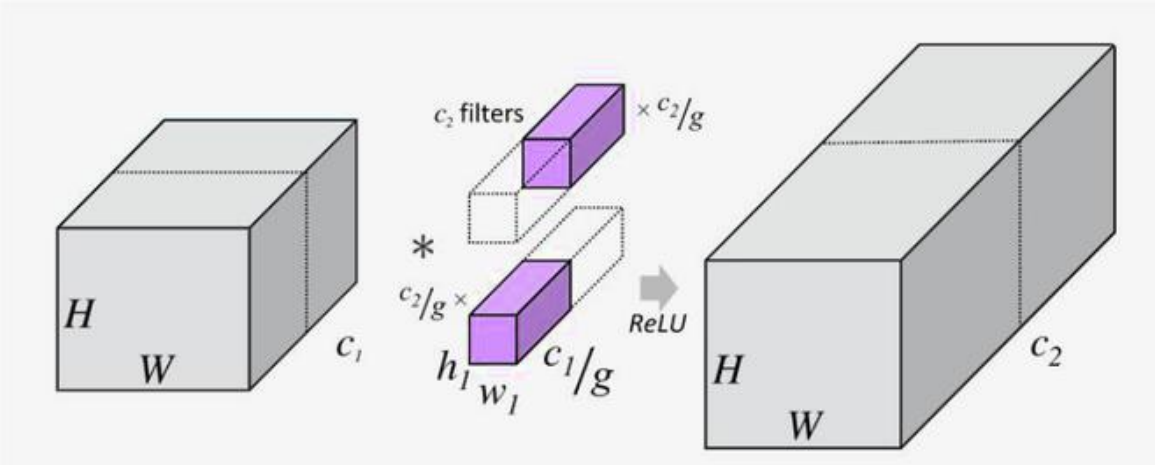
# Grouping

UNIMORE
UNIVERSITÀ DEGLI STUDI DI MODENA E REGGIO EMILIA

- In the regular 2D Convolution, all input channels are convolved using all filters
  - Which implies that all output channels “have seen” all input channels, each through its own filter
- **Changing this “all-to-all” connectivity schema we could connect each input channel to a specific set of filters.**
- For instance, we could say that the first half of the input channels is connected to the first half of the filters, while the second half of the input channels is connected to the second half of the filters. It’s like having two convolutional layers in parallel, or, better to say, two ***groups*** running inside the same layer.

# Grouping

- Following this schema, we could define *n* groups: input channels are divided into *n* groups with equal size, and so are the filters. Each *i*-th group of input channels is connected to the *i*-th group of filters.
- Examples:
  - At groups=1, all inputs are convolved to all outputs.
  - At groups=2, the operation becomes equivalent to having two conv layers side by side, each seeing half the input channels, and producing half the output channels, and both subsequently concatenated.
  - At groups=in_channels, each input channel is connected to its own set of filters, of size $\left\lfloor \frac{out\_channels}{in\_channels} \right\rfloor$



# Output shape



- Padding, dilation and stride all have an impact on the output shape of a 2D Convolutional Layer.
- *Putting this all together, we get the following formulas*

Input: $(N, C_{in}, D_{in}, H_{in}, W_{in})$
Output: $(N, C_{out}, D_{out}, H_{out}, W_{out})$ where

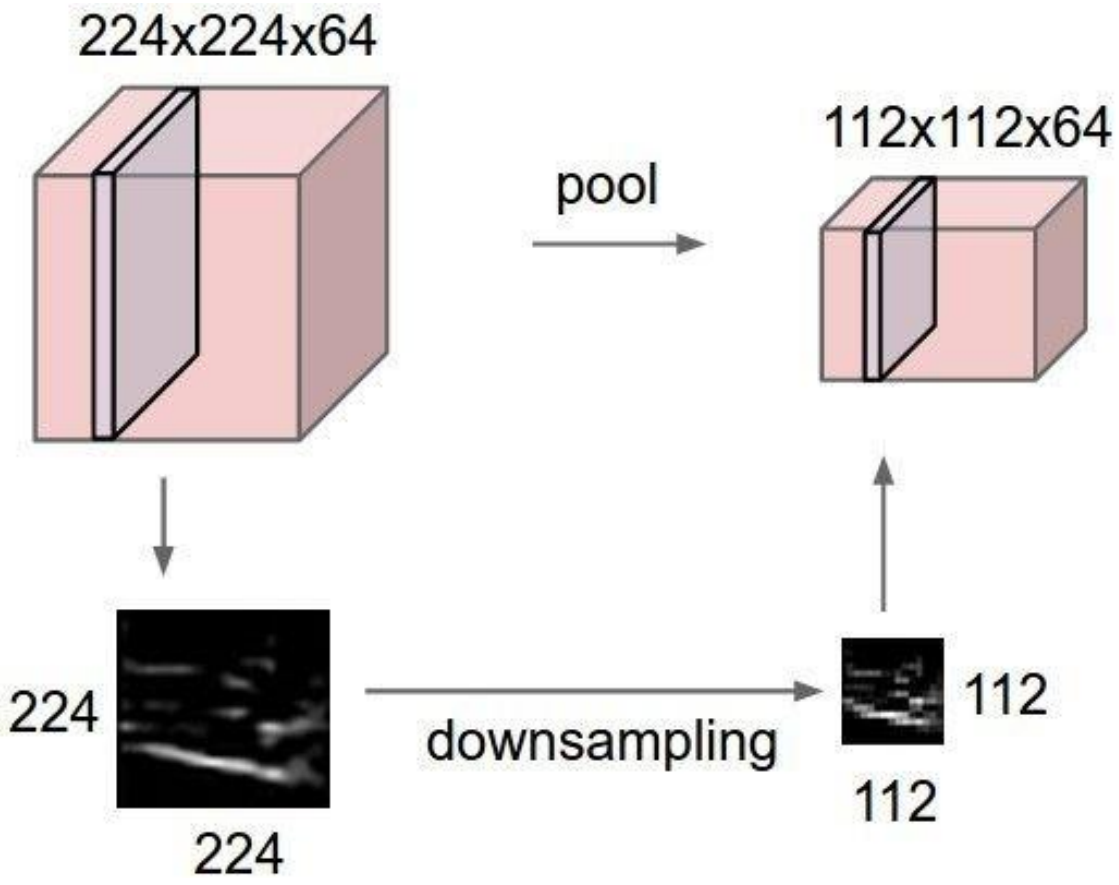
$$D_{out} = \left\lfloor \frac{D_{in} + 2 \times \text{padding}[0] - \text{dilation}[0] \times (\text{kernel\_size}[0] - 1) - 1}{\text{stride}[0]} + 1 \right\rfloor$$

$$H_{out} = \left\lfloor \frac{H_{in} + 2 \times \text{padding}[1] - \text{dilation}[1] \times (\text{kernel\_size}[1] - 1) - 1}{\text{stride}[1]} + 1 \right\rfloor$$

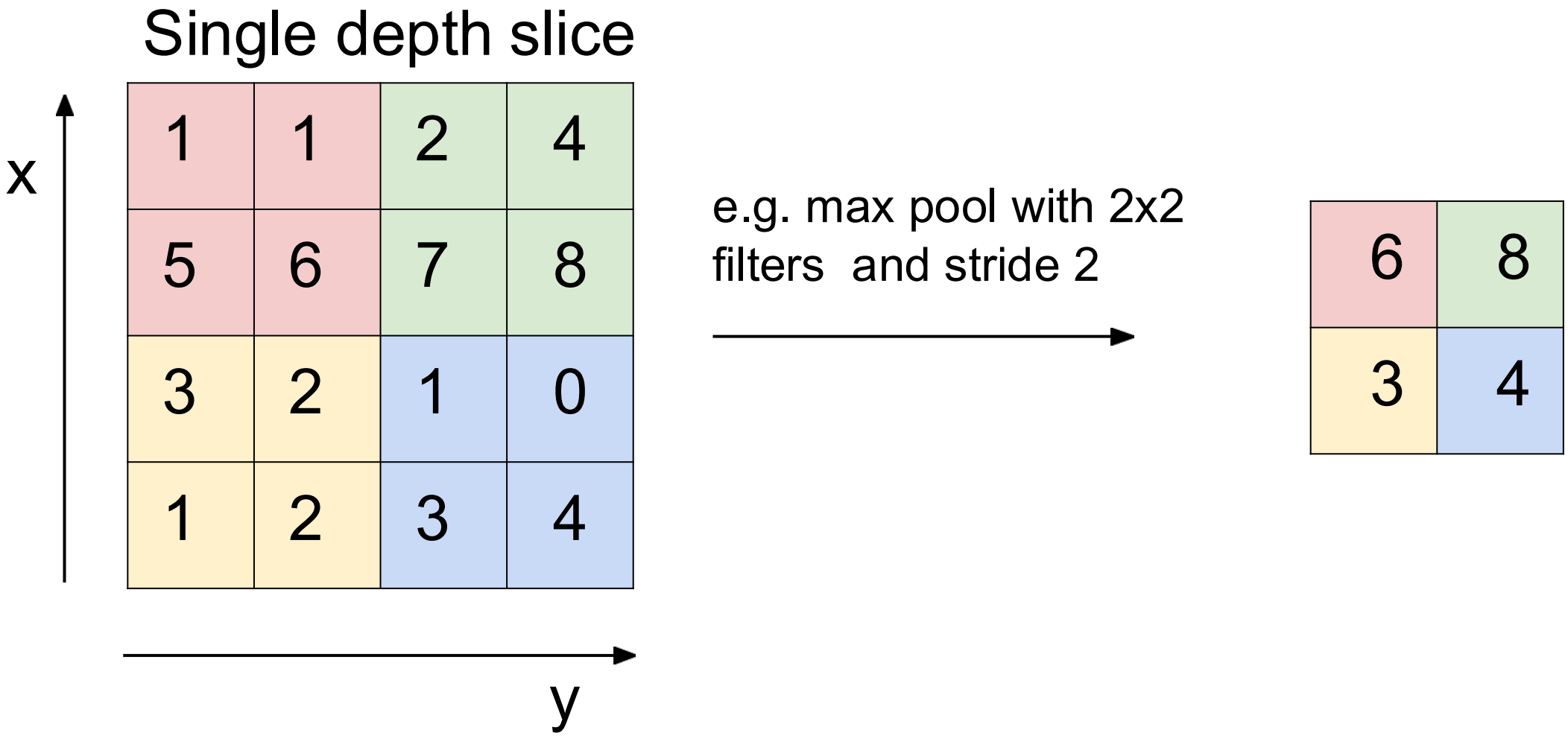
$$W_{out} = \left\lfloor \frac{W_{in} + 2 \times \text{padding}[2] - \text{dilation}[2] \times (\text{kernel\_size}[2] - 1) - 1}{\text{stride}[2]} + 1 \right\rfloor$$

# 2D Pooling

- operates over each activation map independently:
- Often used to make representations smaller and more manageable



# 2D Pooling

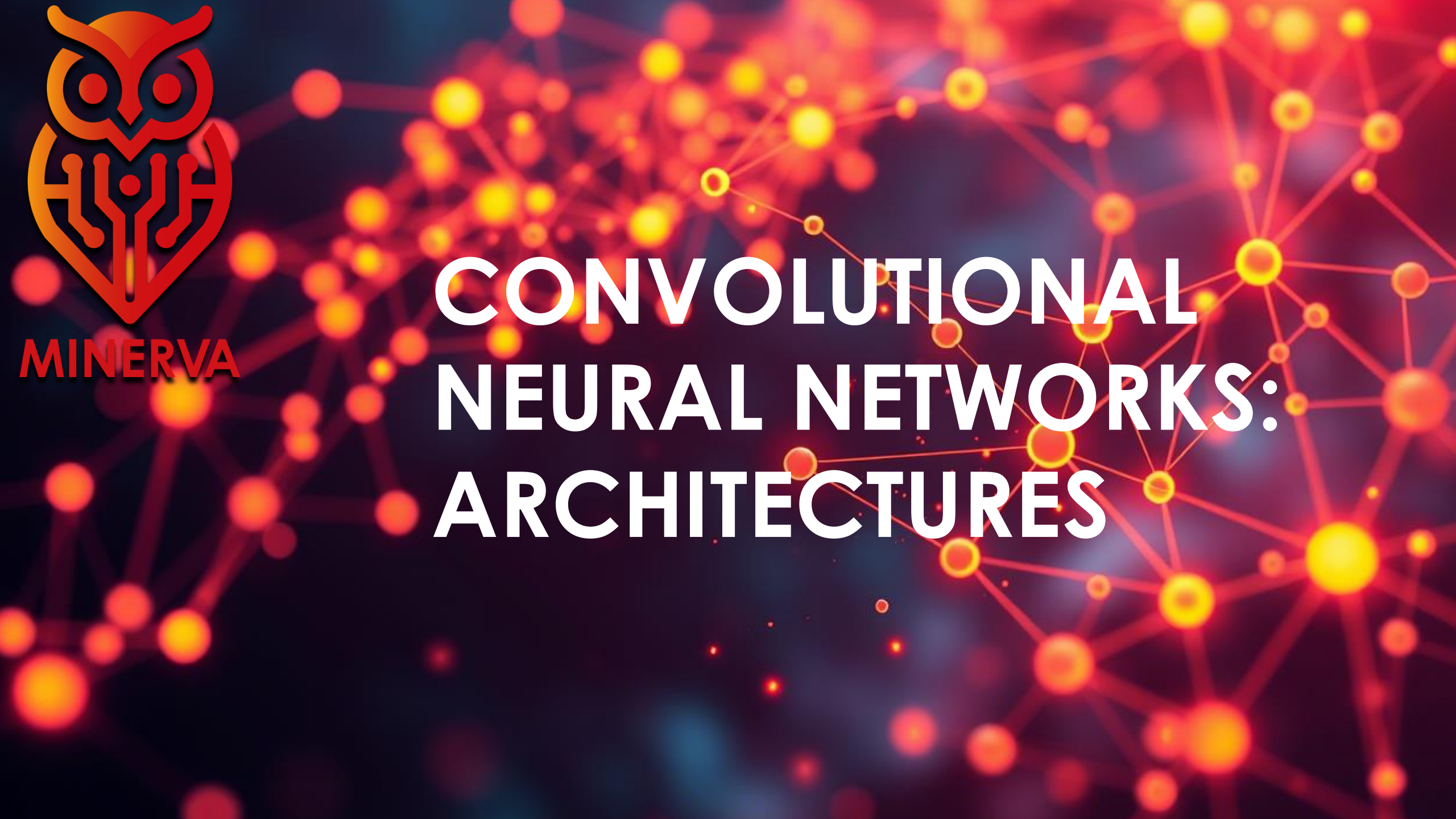



*The same operation is applied over all input channels independently.*

# 2D Pooling

UNIMORE
UNIVERSITÀ DEGLI STUDI DI MODENA E REGGIO EMILIA

- Accepts a tensor of size (n, iC, iH, iW)
- **Performs an operation with zero parameters** (e.g. max, mean) inside a neighbourhood, which slides with a given stride. Differently from convolution, we do not perform a reduction (i.e. sum) over the input channel axis; instead, the operation is applied independently on all input channels.
- **Requires three hyperparameters**:
  - The kernel size (kH, kW)
  - The stride s
- The output size is (n, **iC**, oH, oW), with:
  - oH = (iH – kH)/s + 1
  - oW = (iW – kW)/s + 1
- As for the convolution, we could apply:
  - A padding;
  - A dilation factor




# CONVOLUTIONAL NEURAL NETWORKS: ARCHITECTURES

# Convolutional Neural Networks

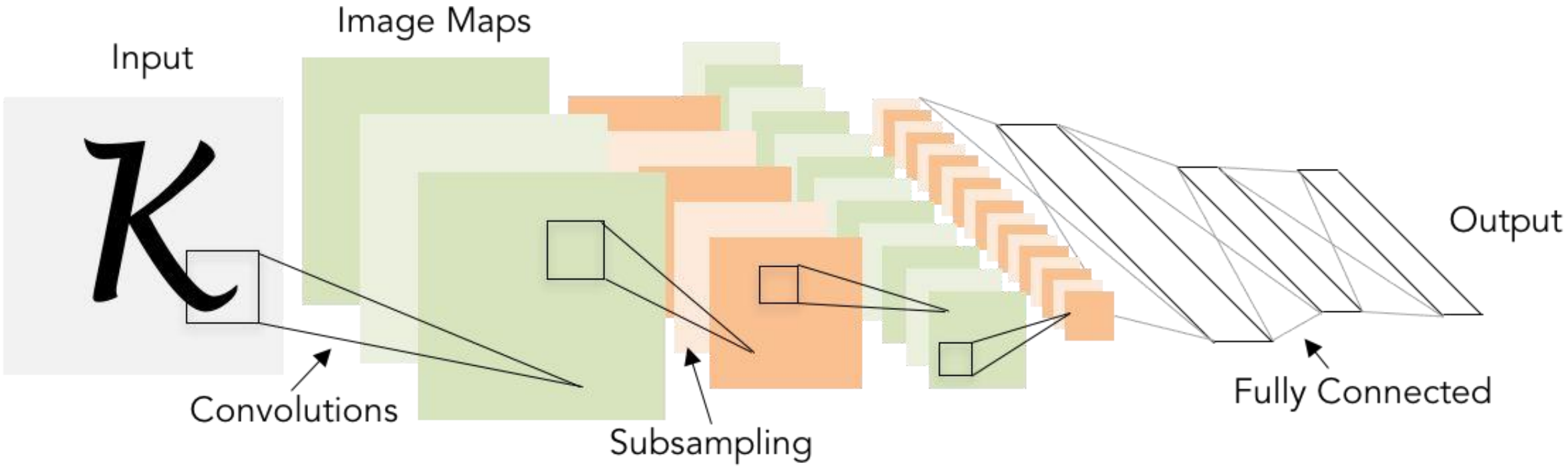
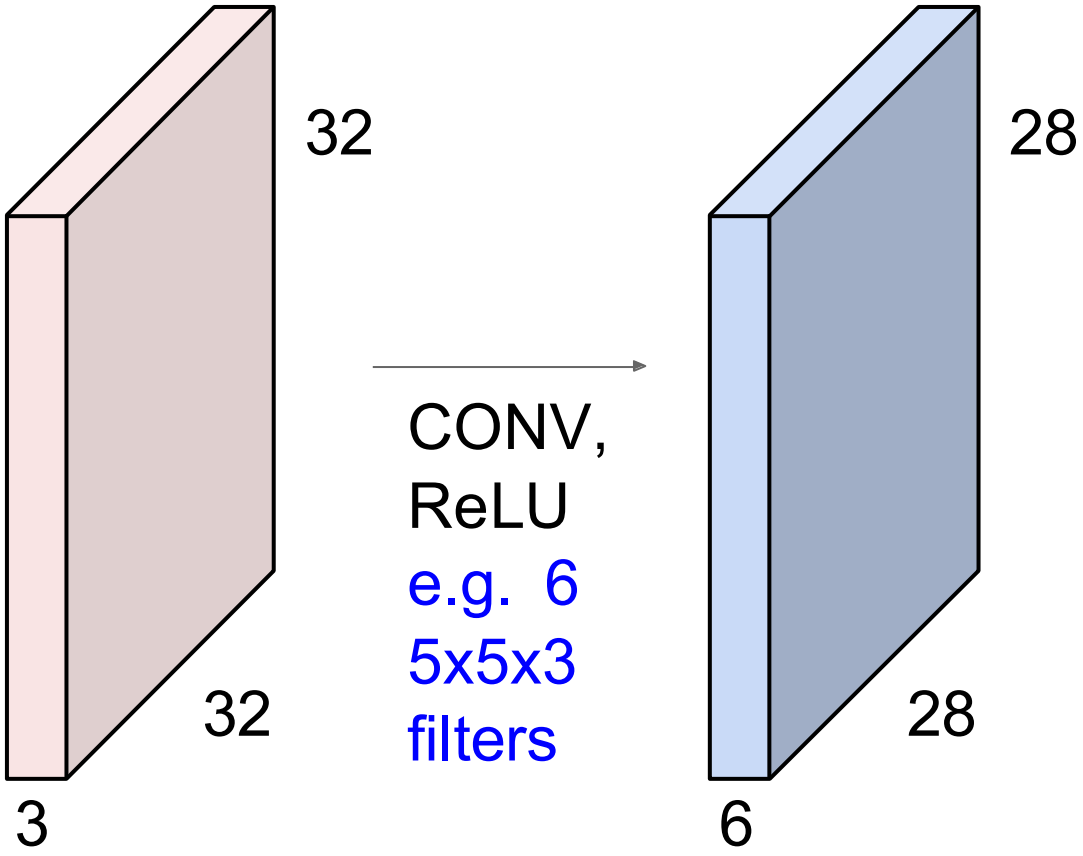
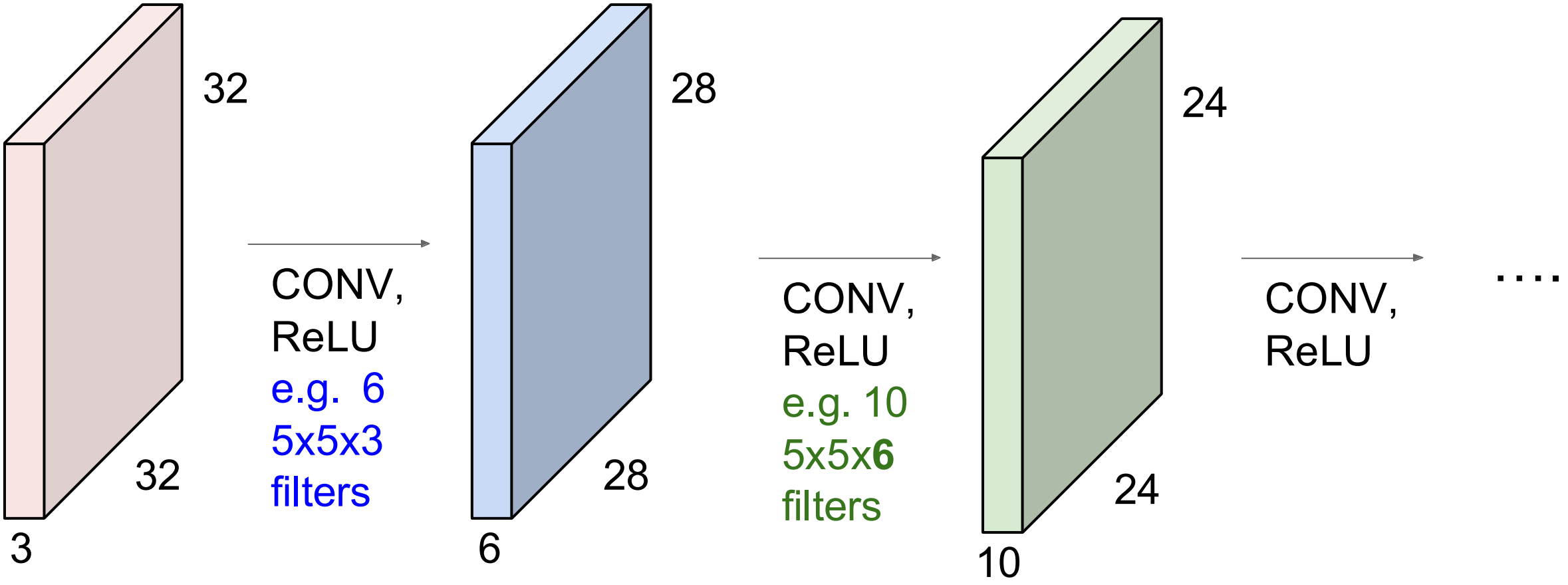



Illustration of LeCun et al. 1998 from CS231n 2017 Lecture 1

**Preview:** ConvNet is a sequence of Convolution Layers, interspersed with activation functions



**Preview:** ConvNet is a sequence of Convolution Layers, interspersed with activation functions

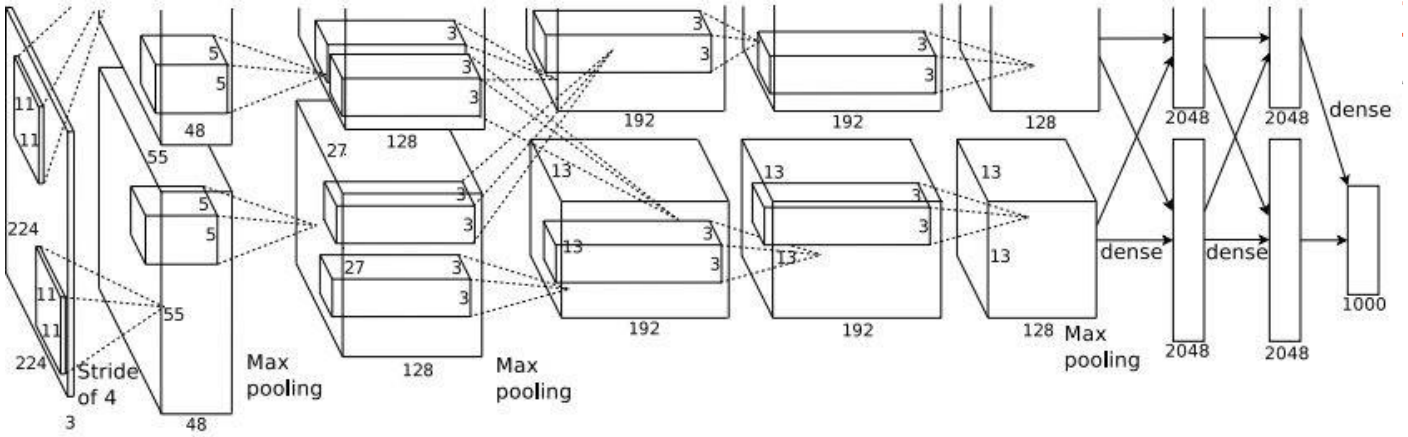


# Case Study: AlexNet

## *[Krizhevsky et al. 2012]*

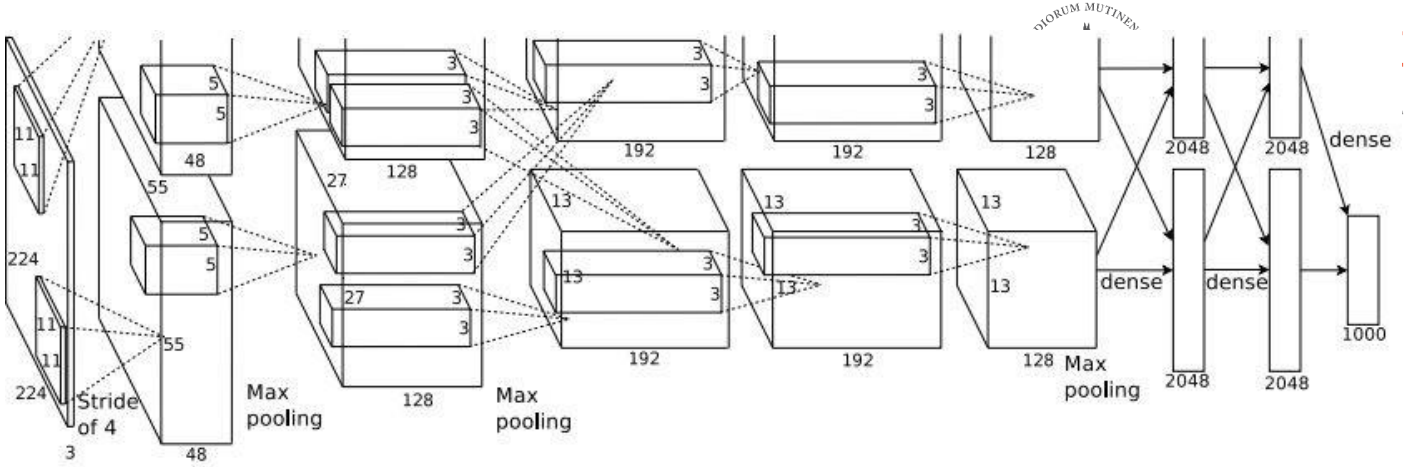
**Architecture:**

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



# Case Study: AlexNet

*[Krizhevsky et al. 2012]*




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)

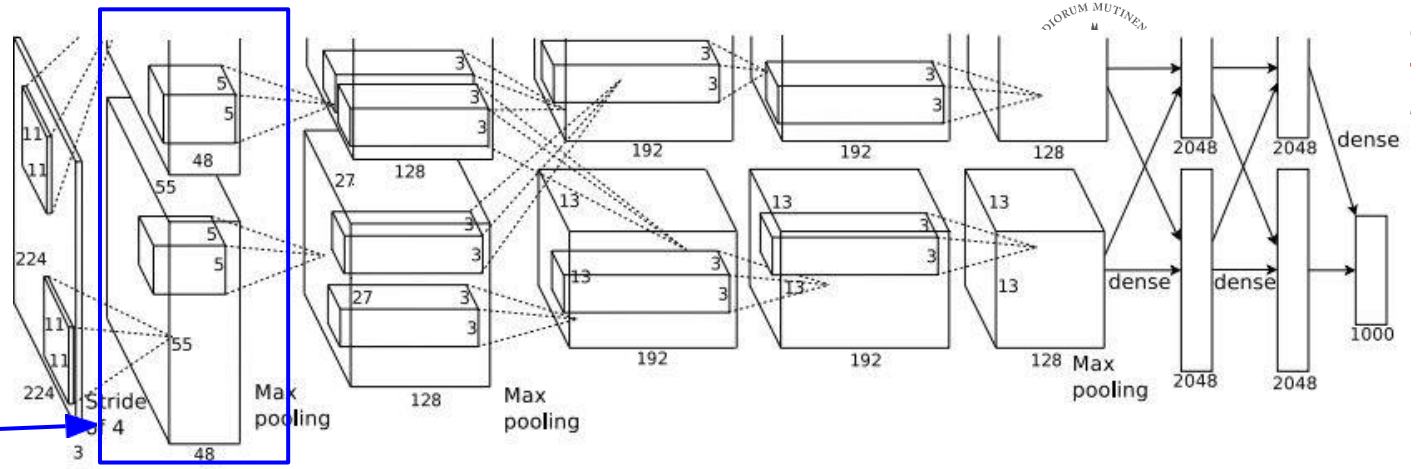
**Details/Retrospectives:**

-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%

# 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)



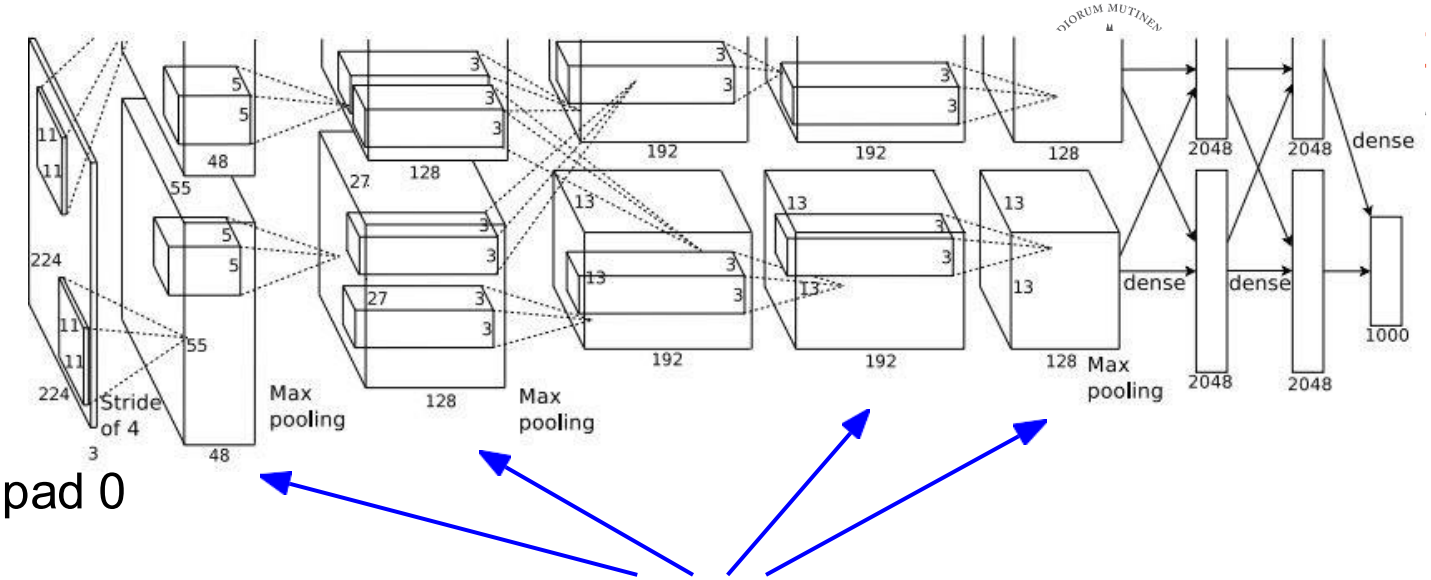

[55x55x48] x 2

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.

# 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)

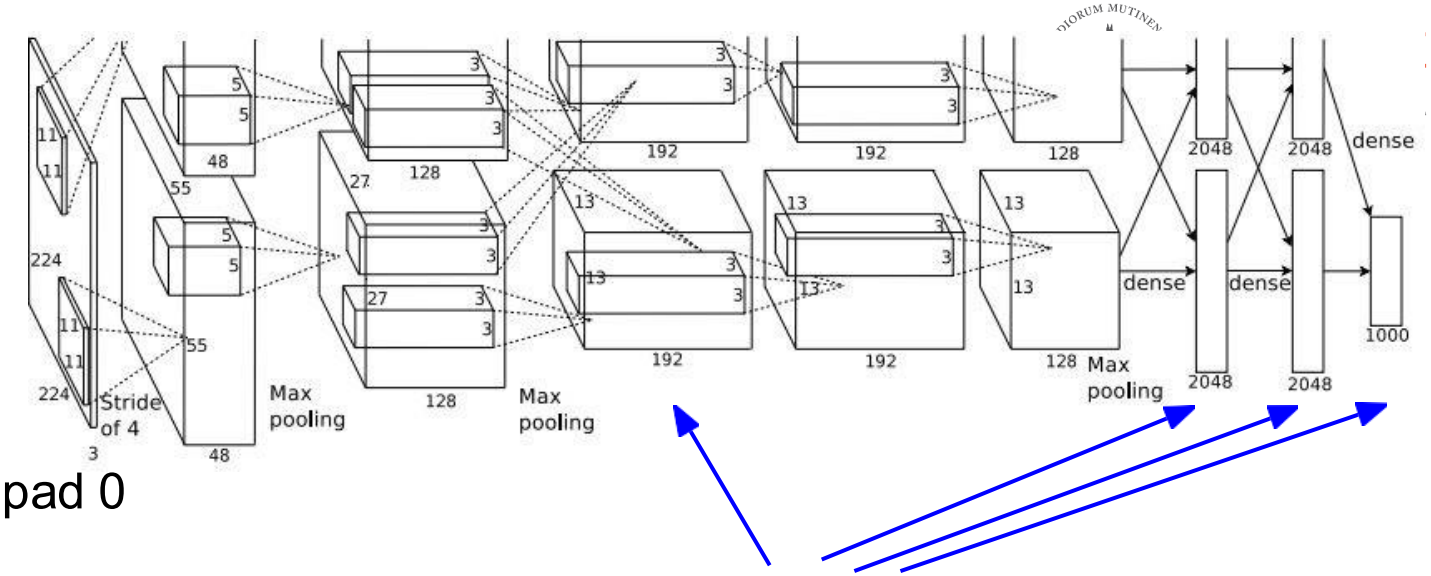



CONV1, CONV2, CONV4, CONV5:
Connections only with feature maps on same GPU

# Case Study: AlexNet

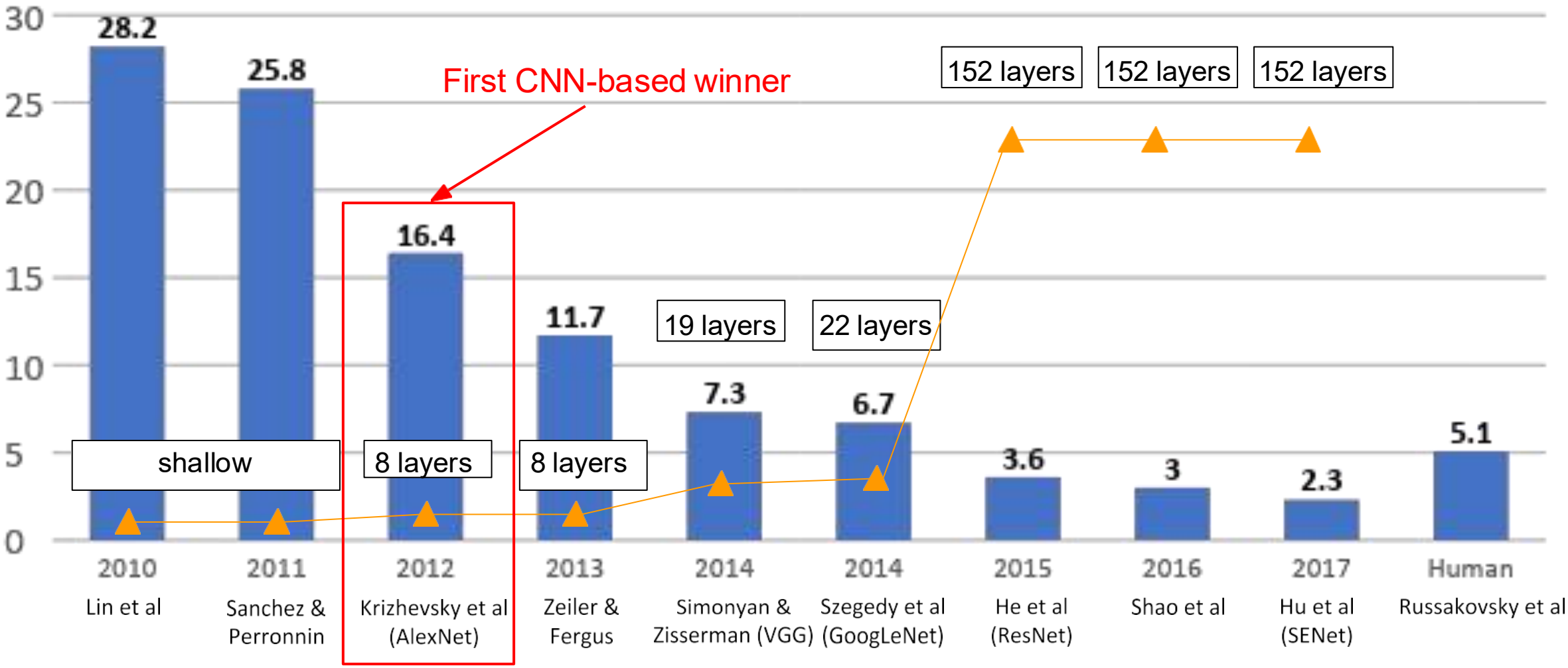
*[Krizhevsky et al. 2012]*

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)



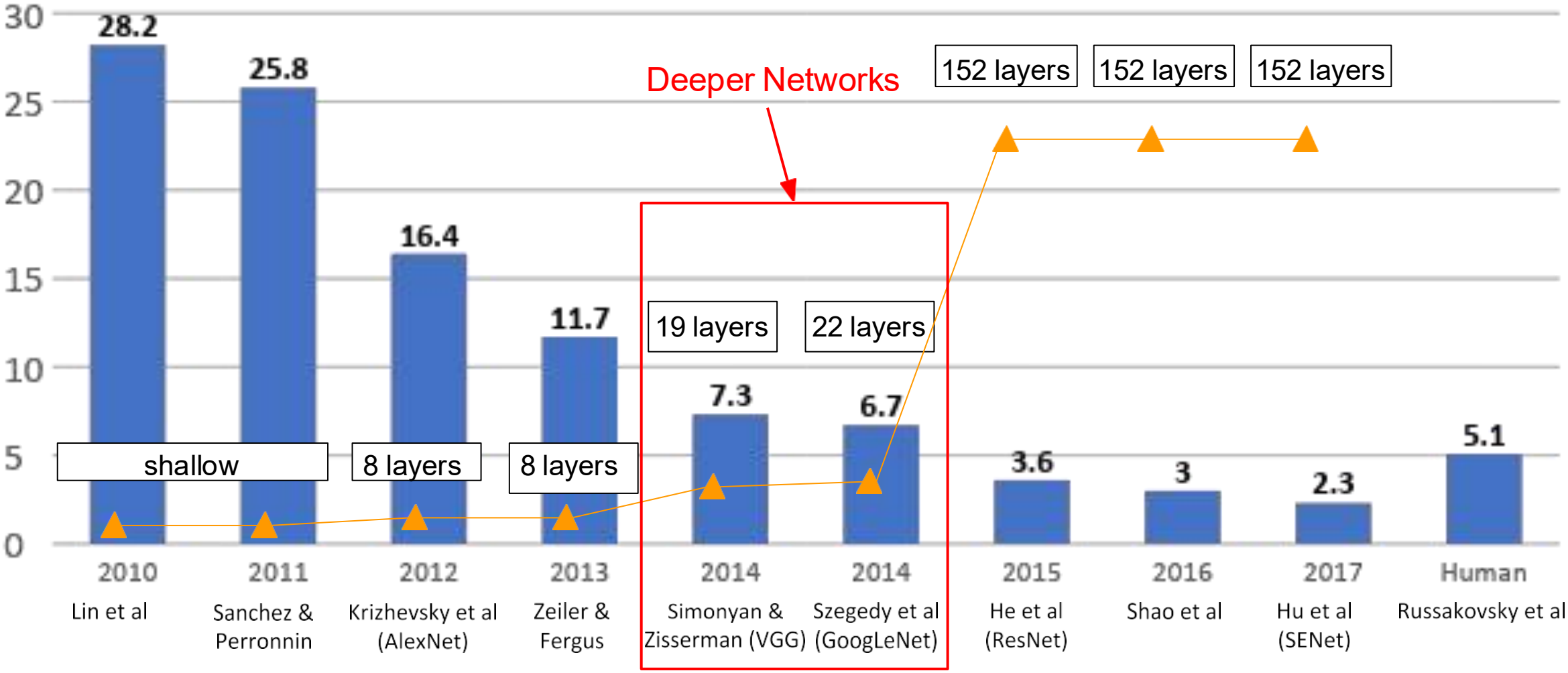

CONV3, FC6, FC7, FC8:
Connections with all feature maps in preceding layer, communication across GPUs

# ImageNet Large Scale Visual Recognition Challenge (ILSVRC) winners



# ImageNet Large Scale Visual Recognition Challenge (ILSVRC) winners





# Case Study: VGGNet

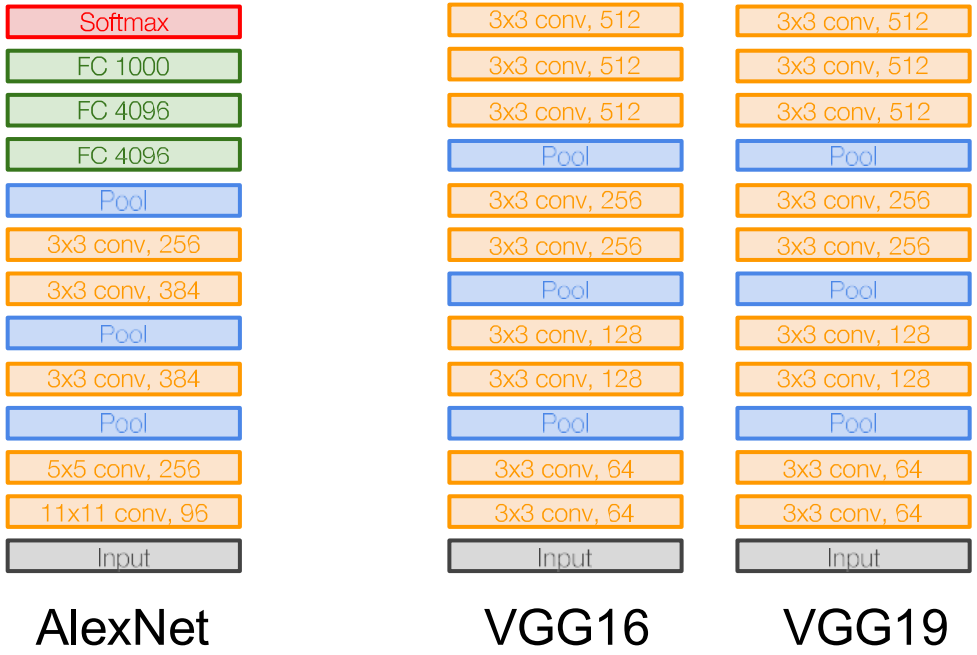
## *[Simonyan and Zisserman, 2014]*

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

Q: What is the effective receptive field of three 3x3 conv (stride 1) layers?


AlexNet


VGG16


VGG19

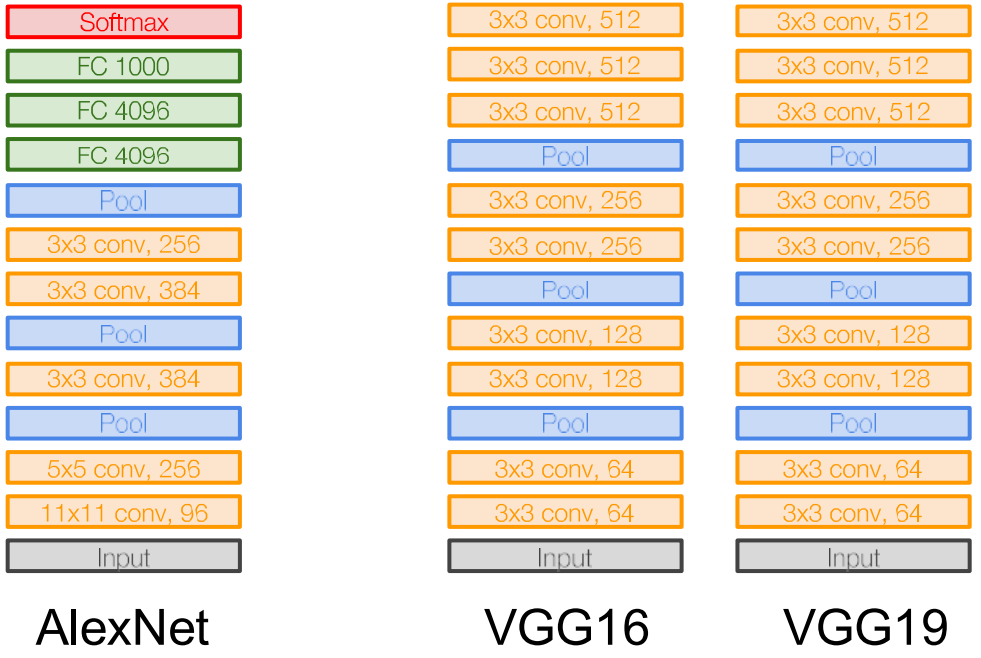
# Case Study: VGGNet

## *[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

[7x7]



# Case Study: VGGNet

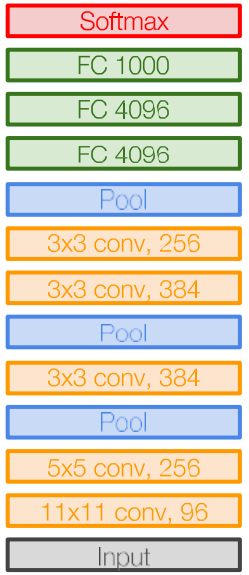
## *[Simonyan and Zisserman, 2014]*

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



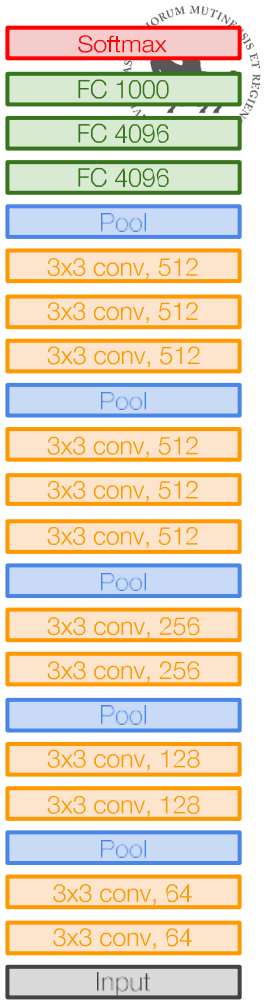

AlexNet


VGG16


VGG19

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

TOTAL memory: 24M * 4 bytes ~= 96MB / image (for a forward pass)
TOTAL params: 138M parameters




VGG16

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

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

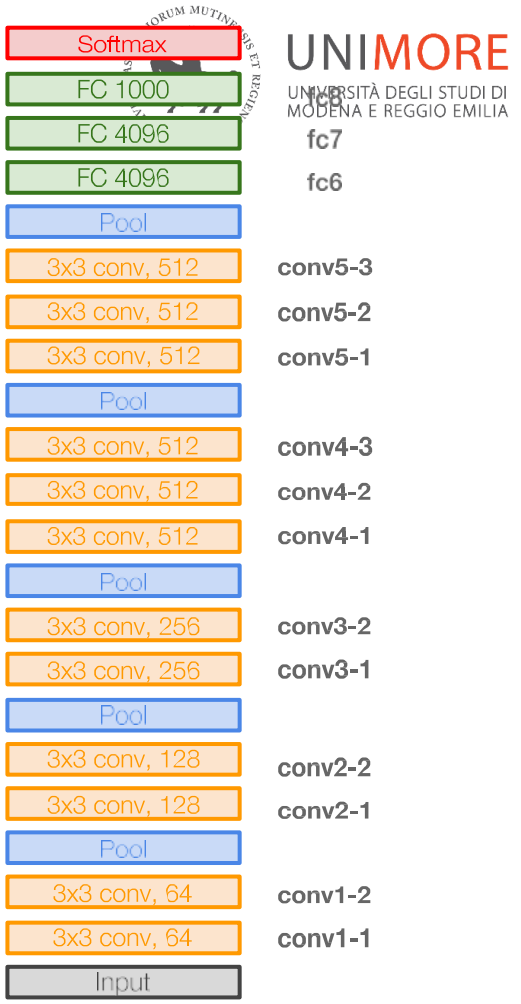
Note:

Most memory is in early CONV

Most params are in late FC

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

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




VGG16

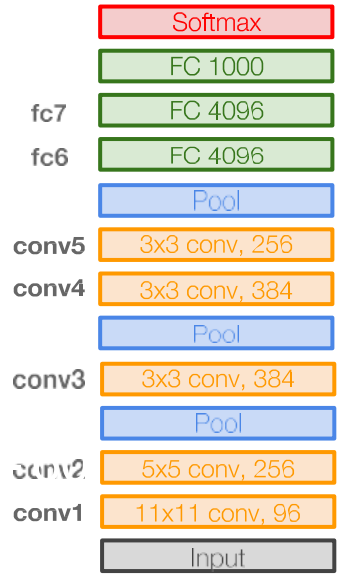
Common names

# Case Study: VGGNet

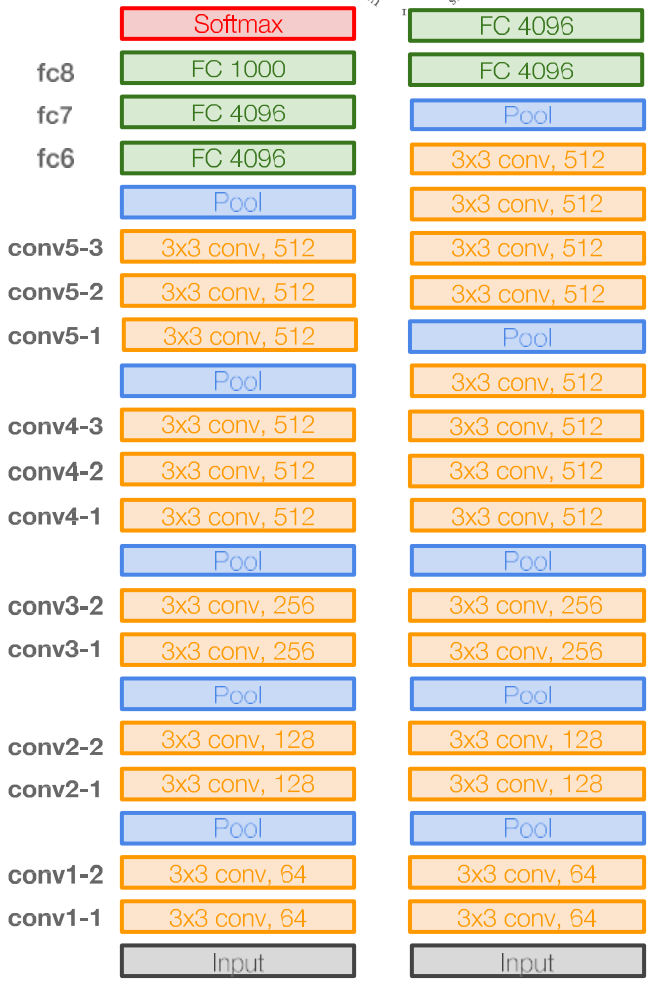
## *[Simonyan and Zisserman, 2014]*

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




AlexNet



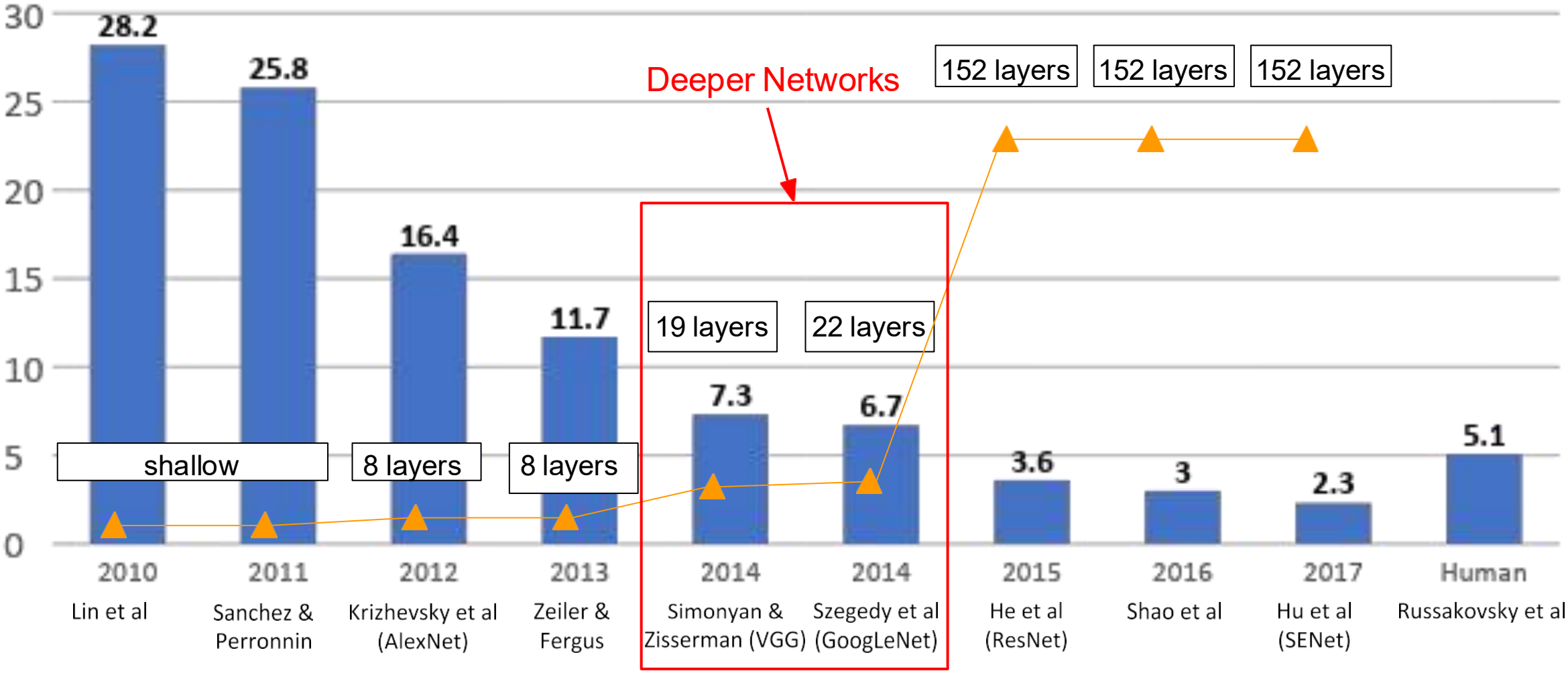

VGG16




VGG19

# ImageNet Large Scale Visual Recognition Challenge (ILSVRC) winners



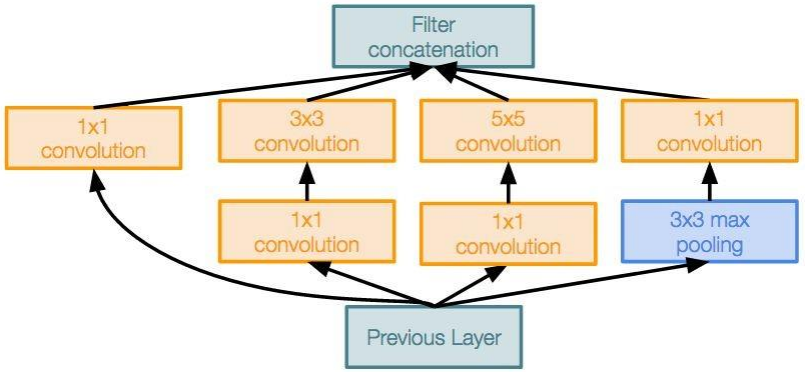


# Case Study: GoogLeNet

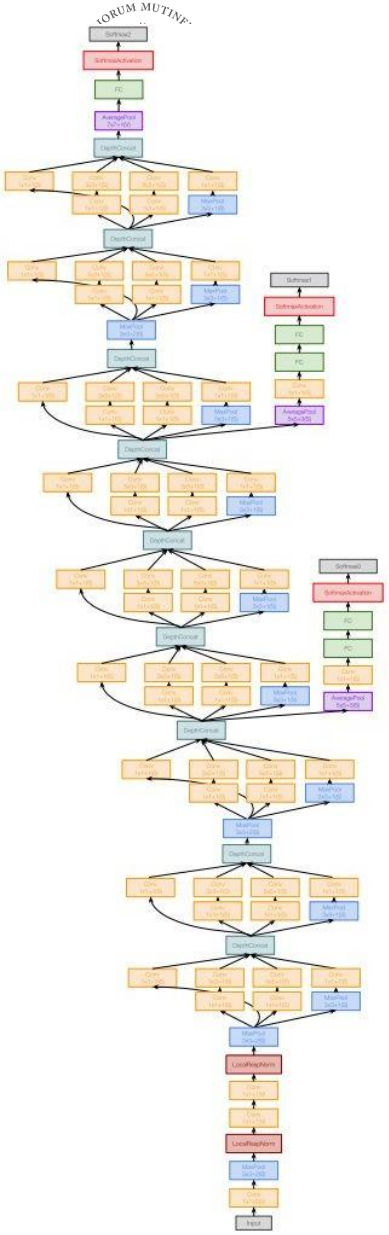
## *[Szegedy et al., 2014]*

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)



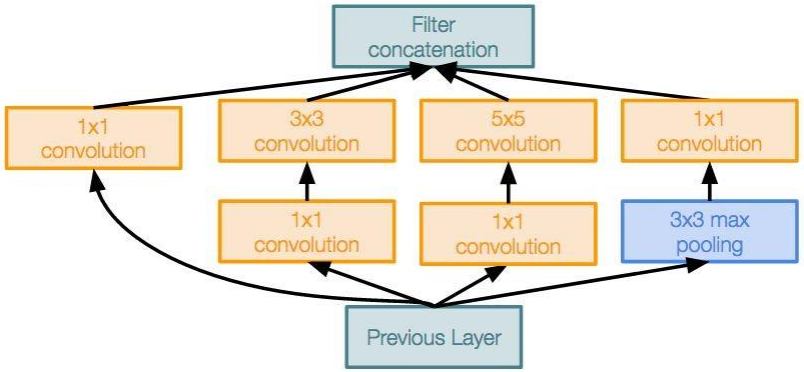

Inception module



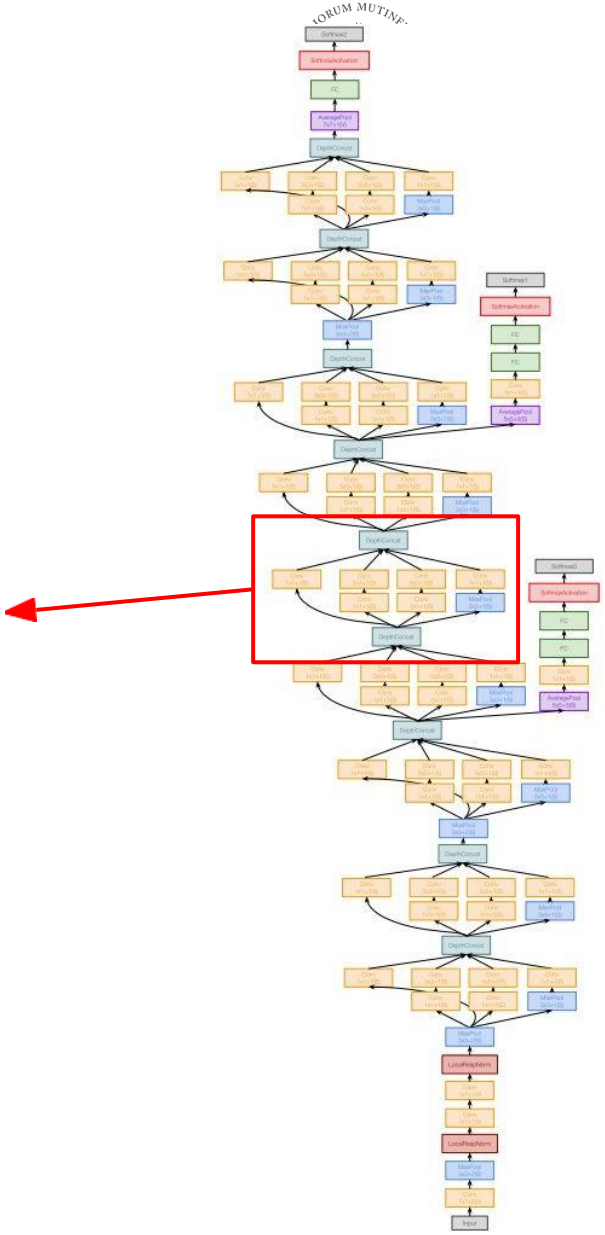
# Case Study: GoogLeNet

***[Szegedy et al., 2014]***

"Inception module": design a good local network topology (network within a network) and then stack these modules on top of each other

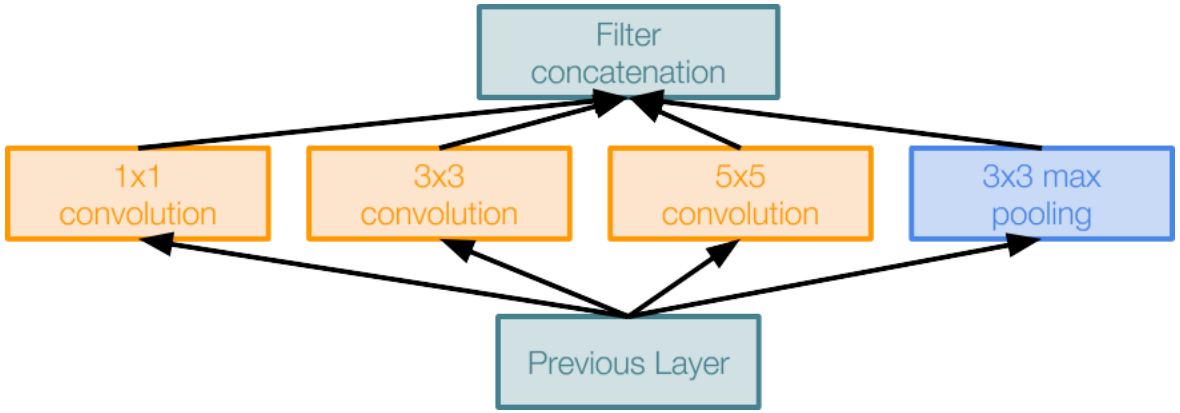



Inception module



# Case Study: GoogLeNet

***[Szegedy et al., 2014]***




Naive Inception module

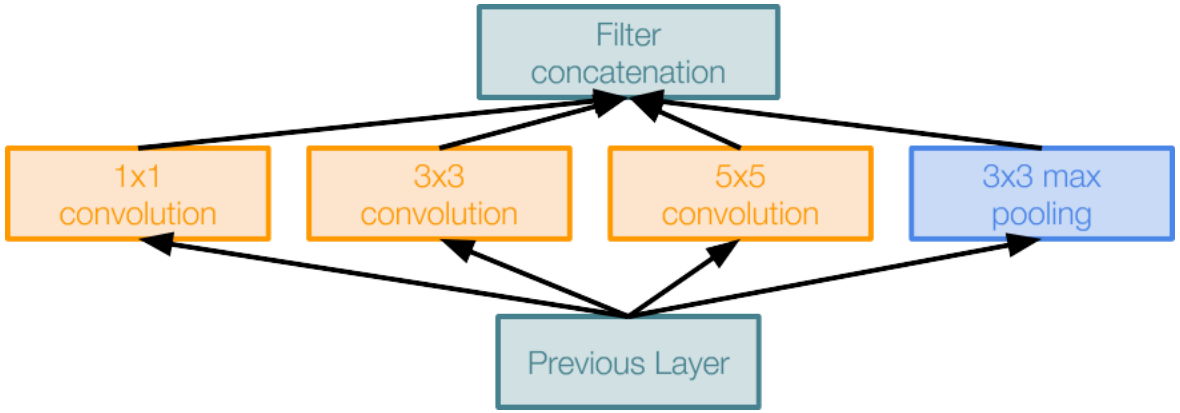
Apply parallel filter operations on the input from previous layer:

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

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)
- Each conv is conv+BN

Concatenate all filter outputs together depth-wise

Q: What is the problem with this?

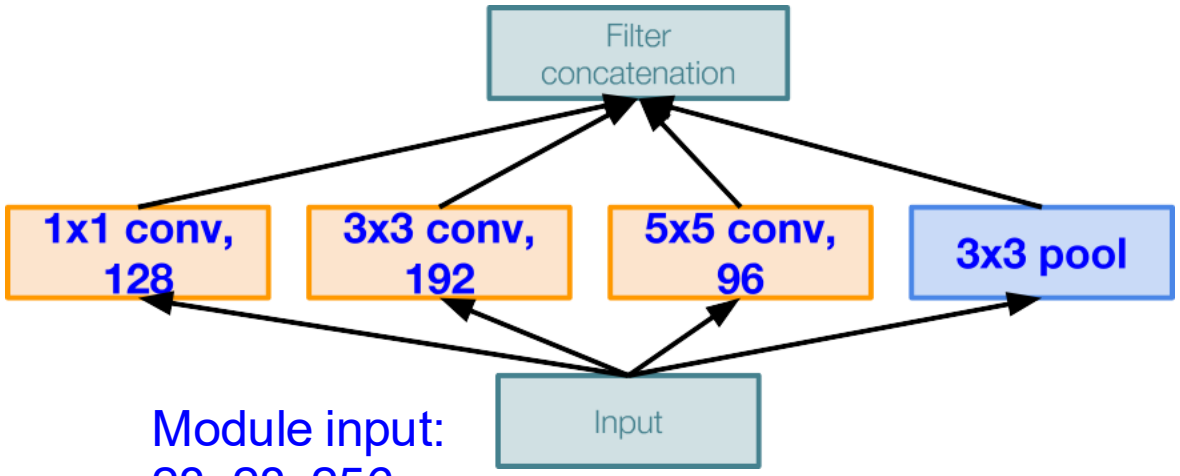
[Hint: Computational complexity]

# Case Study: GoogLeNet

***[Szegedy et al., 2014]***

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

Example:




Naive Inception module

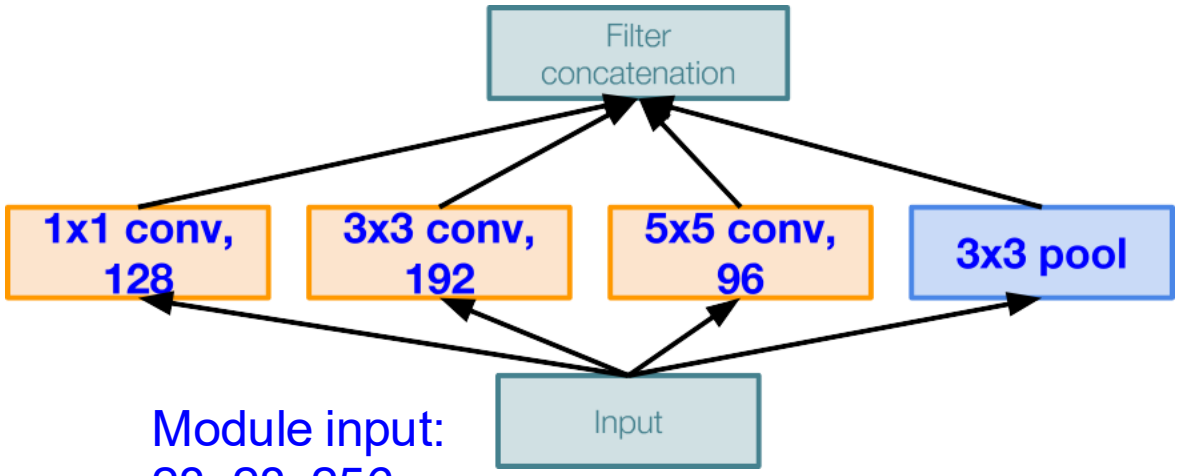
# Case Study: GoogLeNet

*[Szegedy et al., 2014]*

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

Example:

Q1: What is the output size of the 1x1 conv, with 128 filters?




Module input:
28x28x256

Naive Inception module

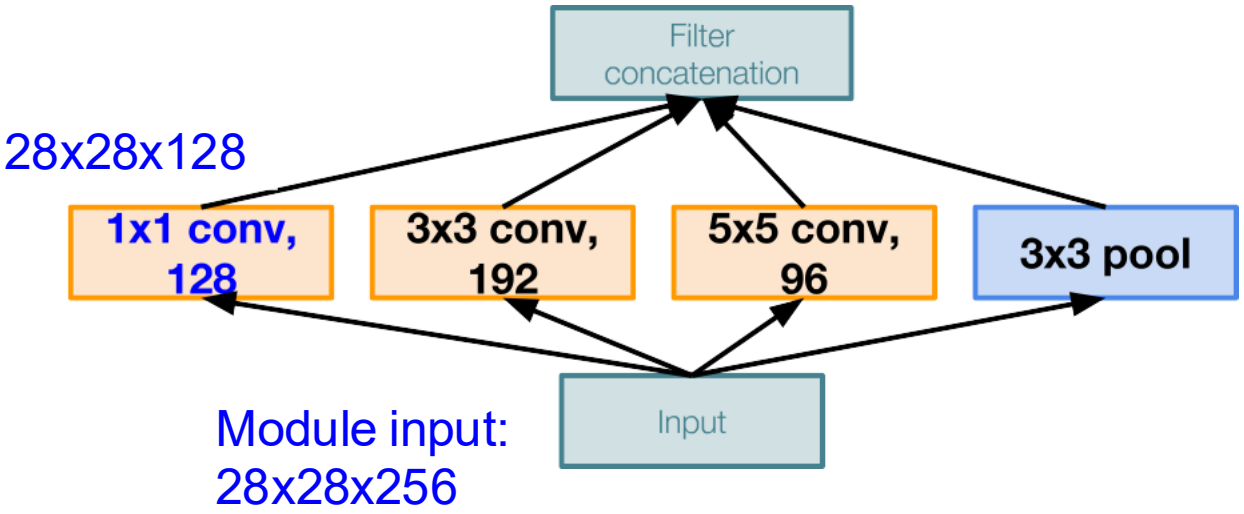
# Case Study: GoogLeNet

*[Szegedy et al., 2014]*

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

Example:

Q1: What is the output size of the 1x1 conv, with 128 filters?




Naive Inception module

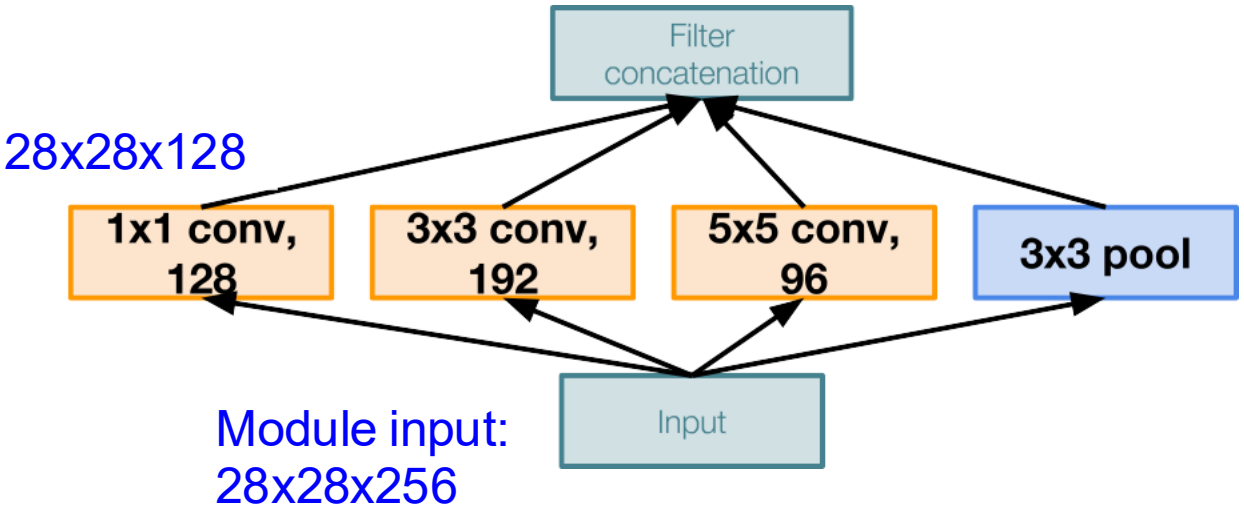
# Case Study: GoogLeNet

*[Szegedy et al., 2014]*

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

Example:

Q2: What are the output sizes of all different filter operations?




Naive Inception module

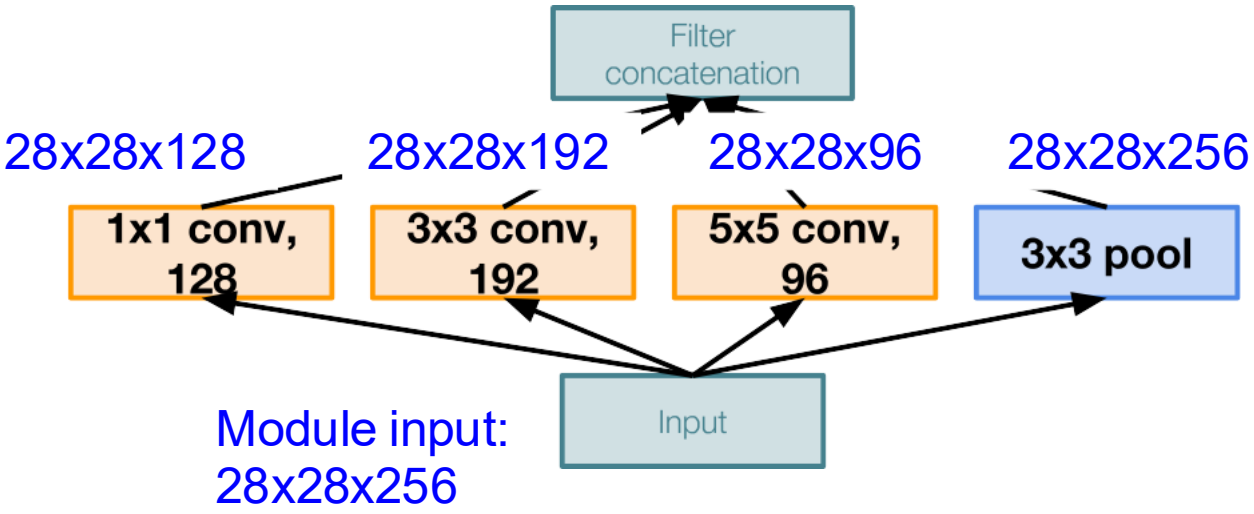
# Case Study: GoogLeNet

*[Szegedy et al., 2014]*

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

Example:

Q2: What are the output sizes of all different filter operations?




Naive Inception module

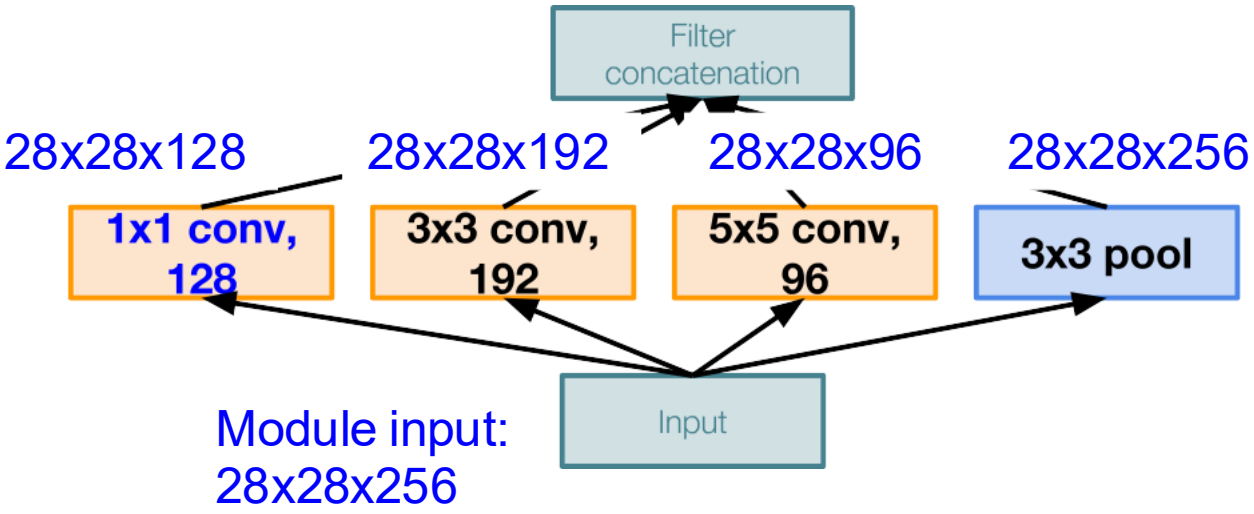
# Case Study: GoogLeNet

*[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

# Case Study: GoogLeNet

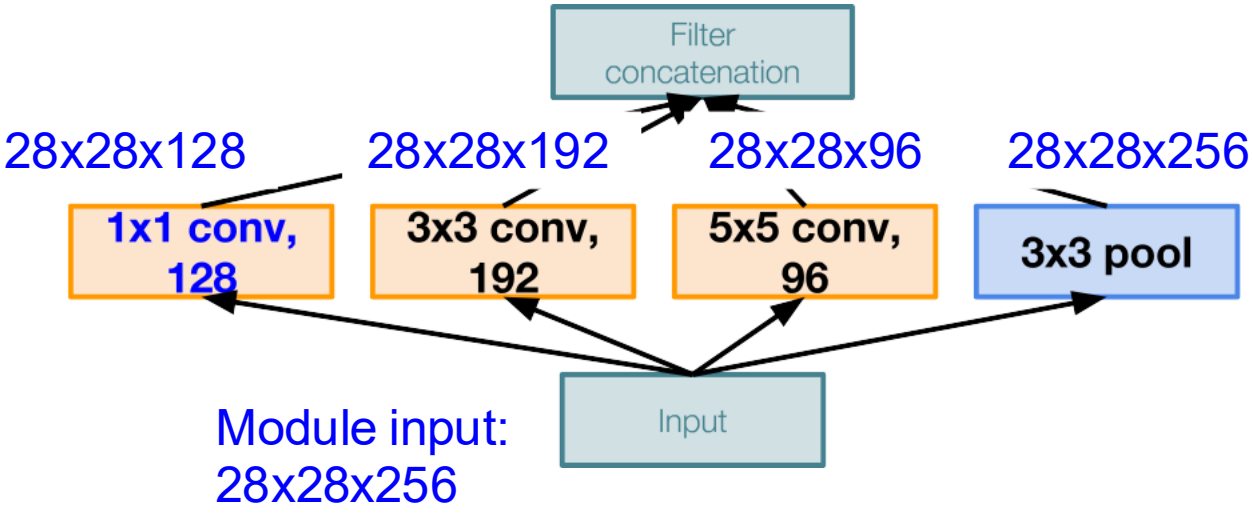
*[Szegedy et al., 2014]*

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




Naive Inception module

# Case Study: GoogLeNet

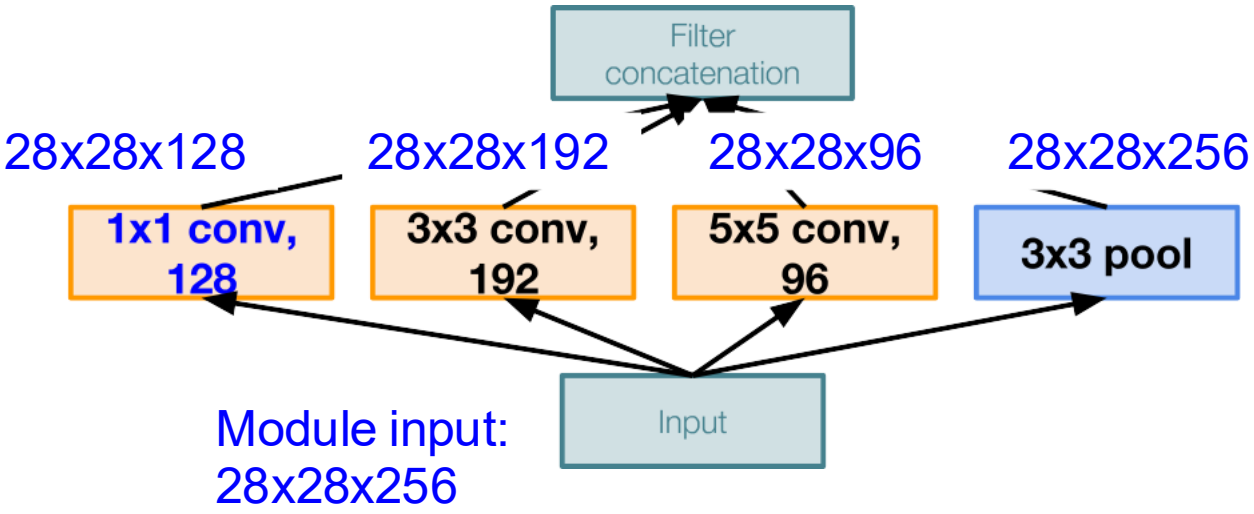
*[Szegedy et al., 2014]*

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




Naive Inception module

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

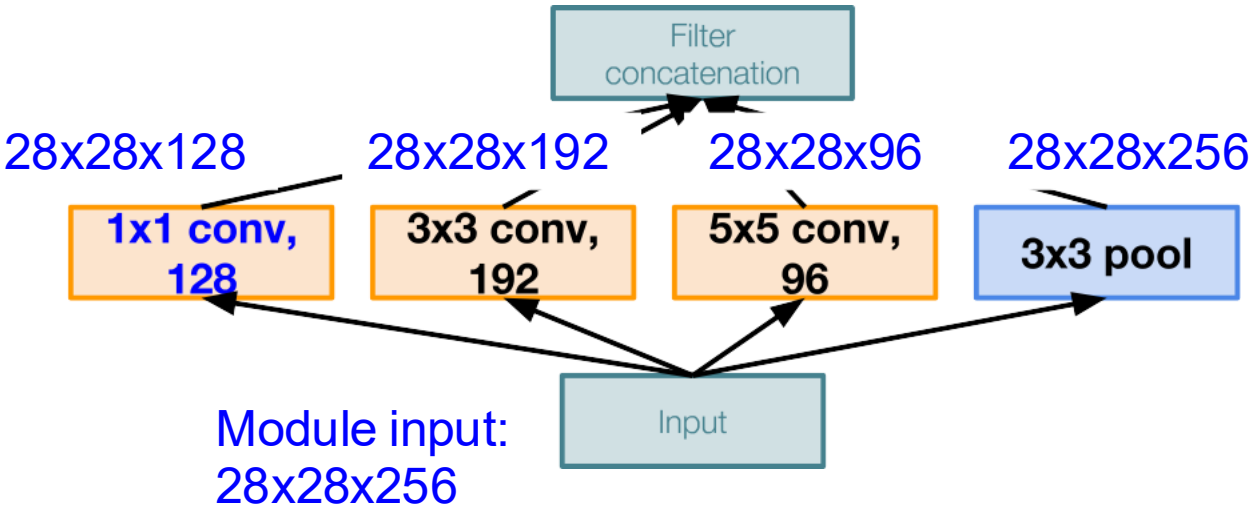
# Case Study: GoogLeNet

*[Szegedy et al., 2014]*

Example:

Q3:What is output size after filter concatenation?

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




Naive Inception module

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

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

Very expensive compute

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

# Case Study: GoogLeNet

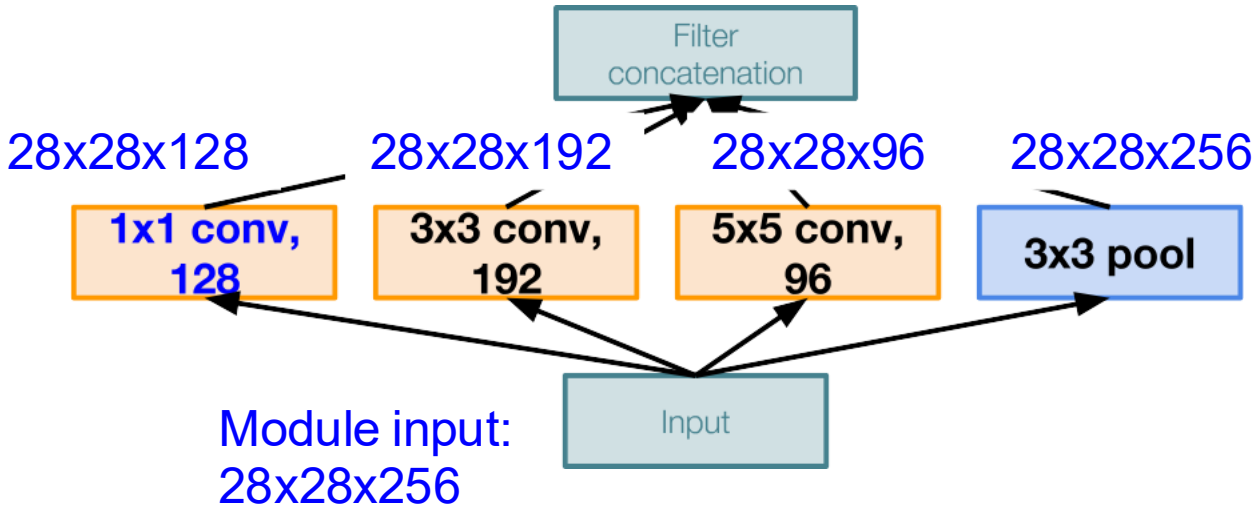
*[Szegedy et al., 2014]*

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

Example:

Q3:What is output size after filter concatenation?

28x28x(128+192+96+256) = **529k**

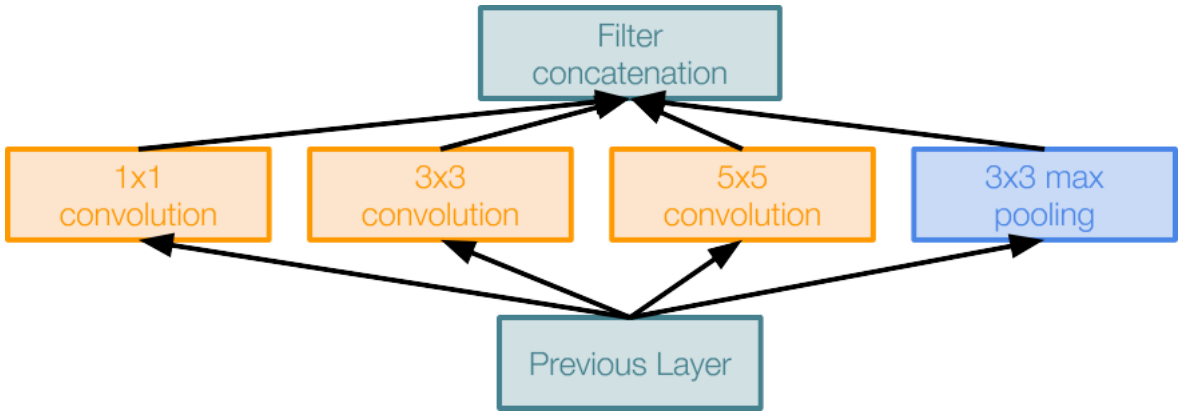



Naive Inception module

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

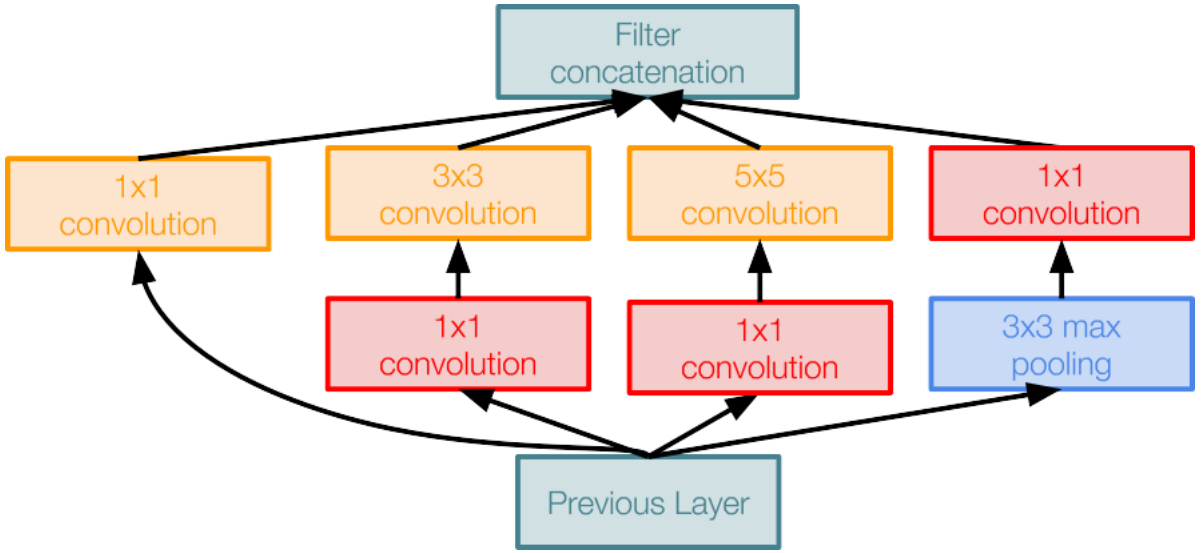
# Case Study: GoogLeNet

***[Szegedy et al., 2014]***




Naive Inception module

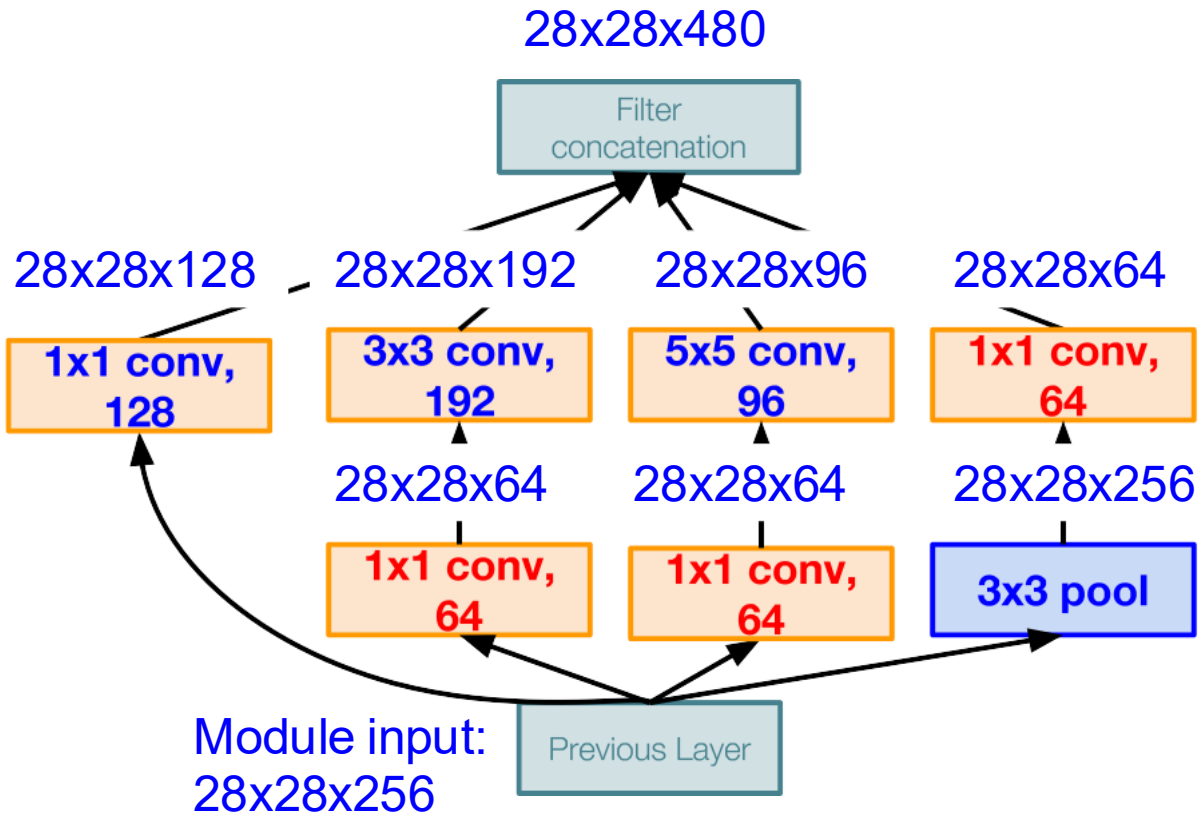
1x1 conv "bottleneck" layers




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:

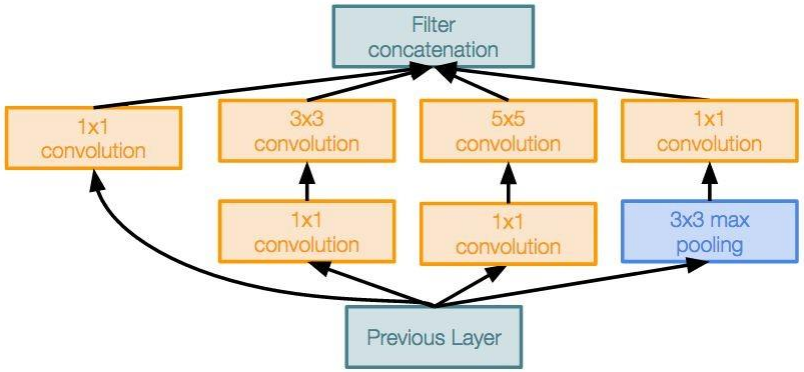
**Conv Ops:**
[1x1 conv, 64] 28x28x64x1x1x256
[1x1 conv, 64] 28x28x64x1x1x256
[1x1 conv, 128] 28x28x128x1x1x256
[3x3 conv, 192] 28x28x192x3x3x64
[5x5 conv, 96] 28x28x96x5x5x64
[1x1 conv, 64] 28x28x64x1x1x256
**Total: 358M ops**

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

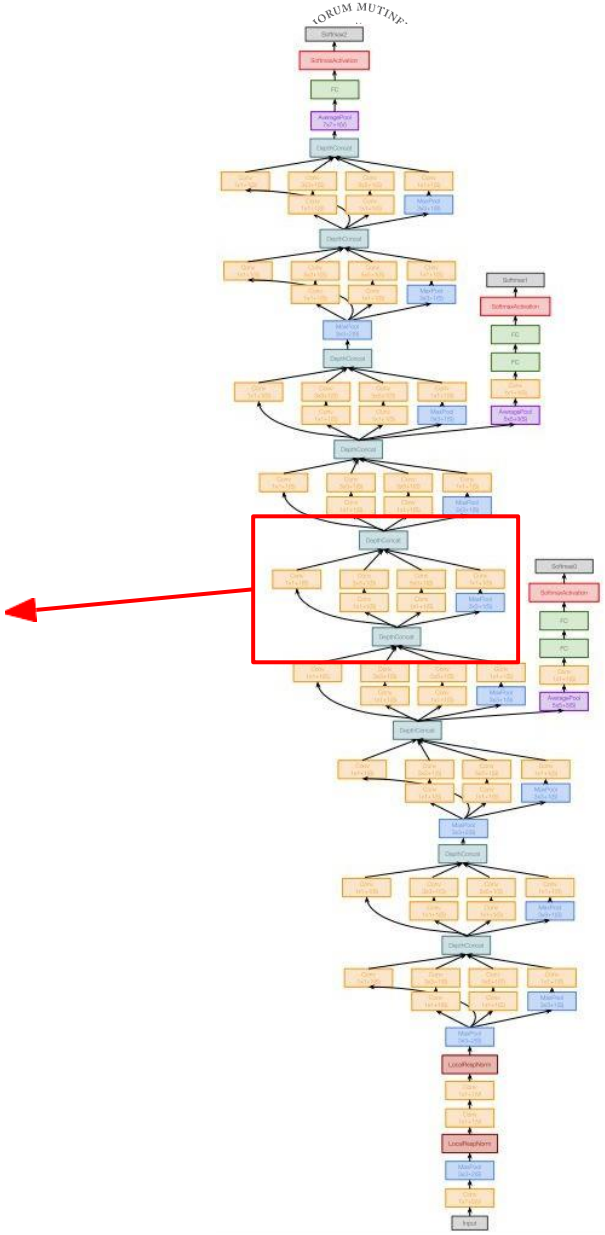
# Case Study: GoogLeNet

***[Szegedy et al., 2014]***

Stack Inception modules with dimension reduction on top of each other



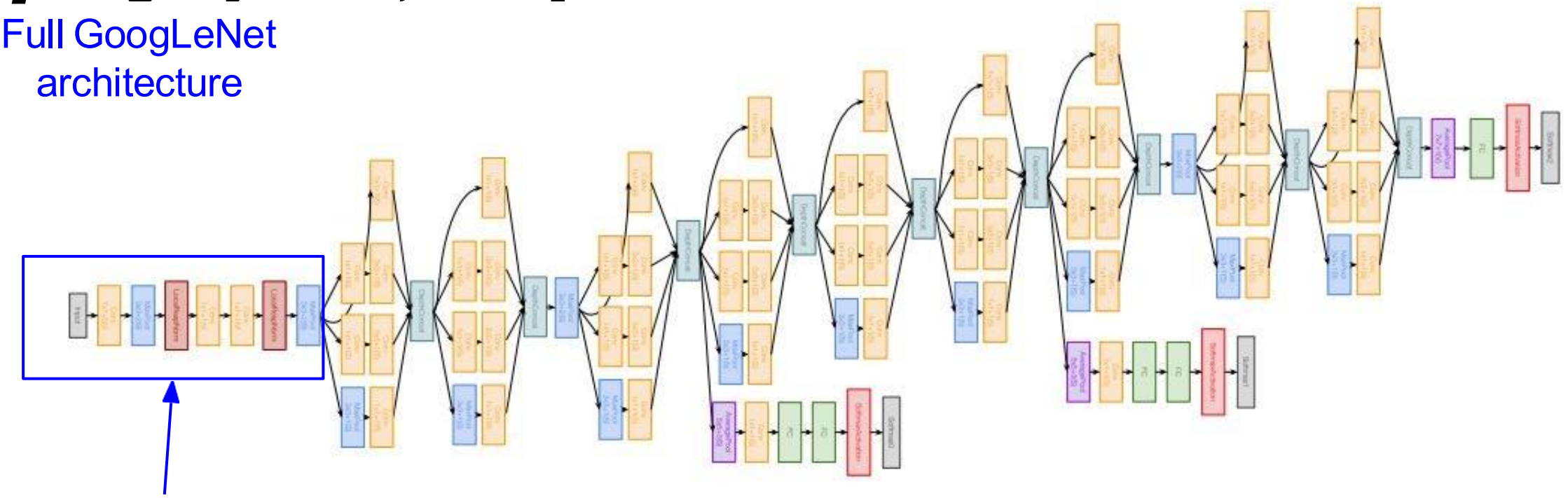

Inception module



# Case Study: GoogLeNet

***[Szegedy et al., 2014]***

Full GoogLeNet architecture

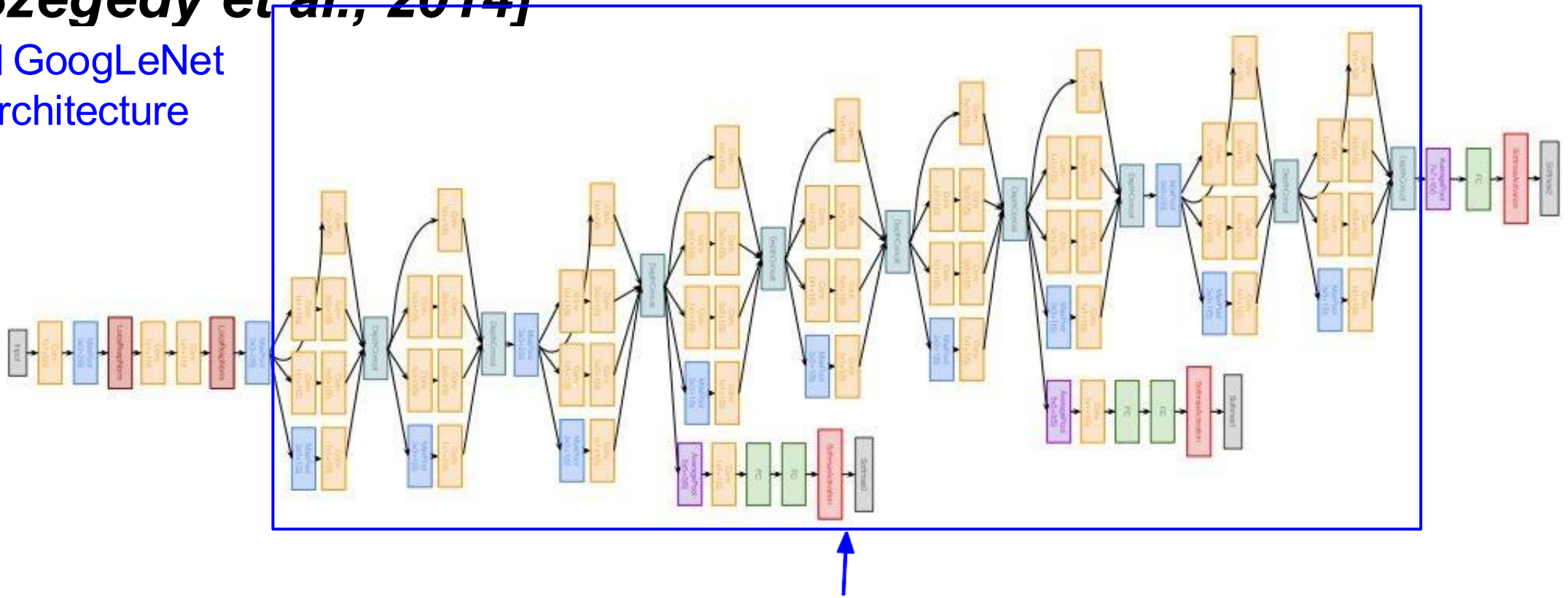


Stem Network:
Conv-Pool-
2x Conv-Pool

# Case Study: GoogLeNet

***[Szegedy et al., 2014]***

Full GoogLeNet architecture

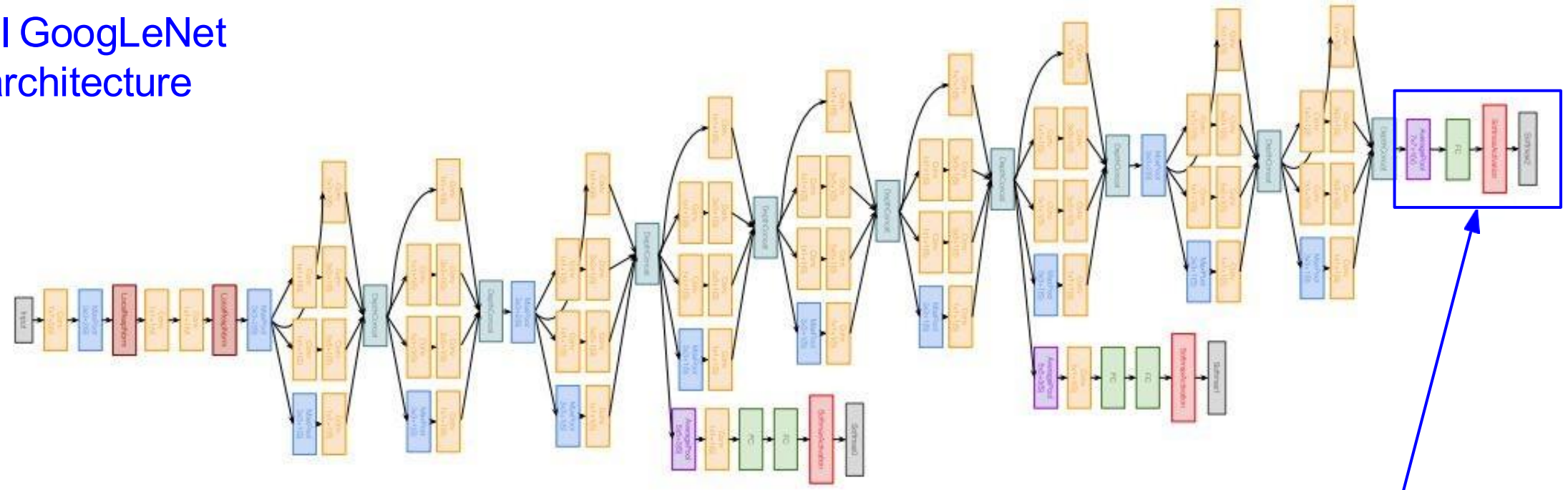


Stacked Inception Modules

# Case Study: GoogLeNet

***[Szegedy et al., 2014]***

Full GoogLeNet architecture

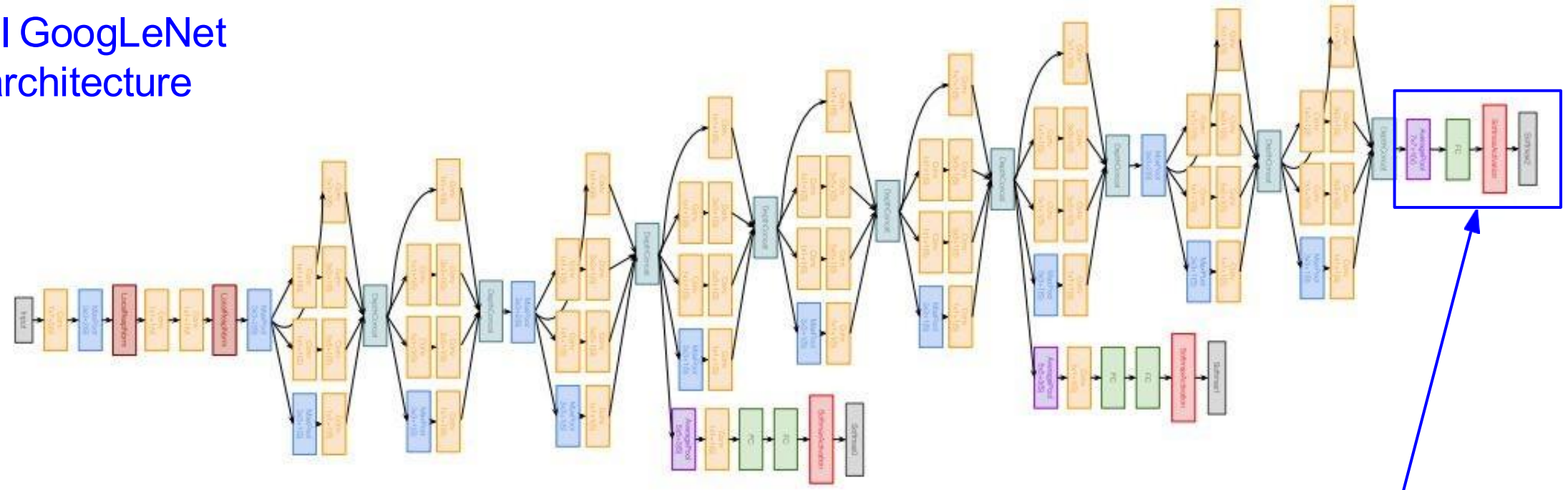


Classifier output

# Case Study: GoogLeNet

***[Szegedy et al., 2014]***

Full GoogLeNet architecture

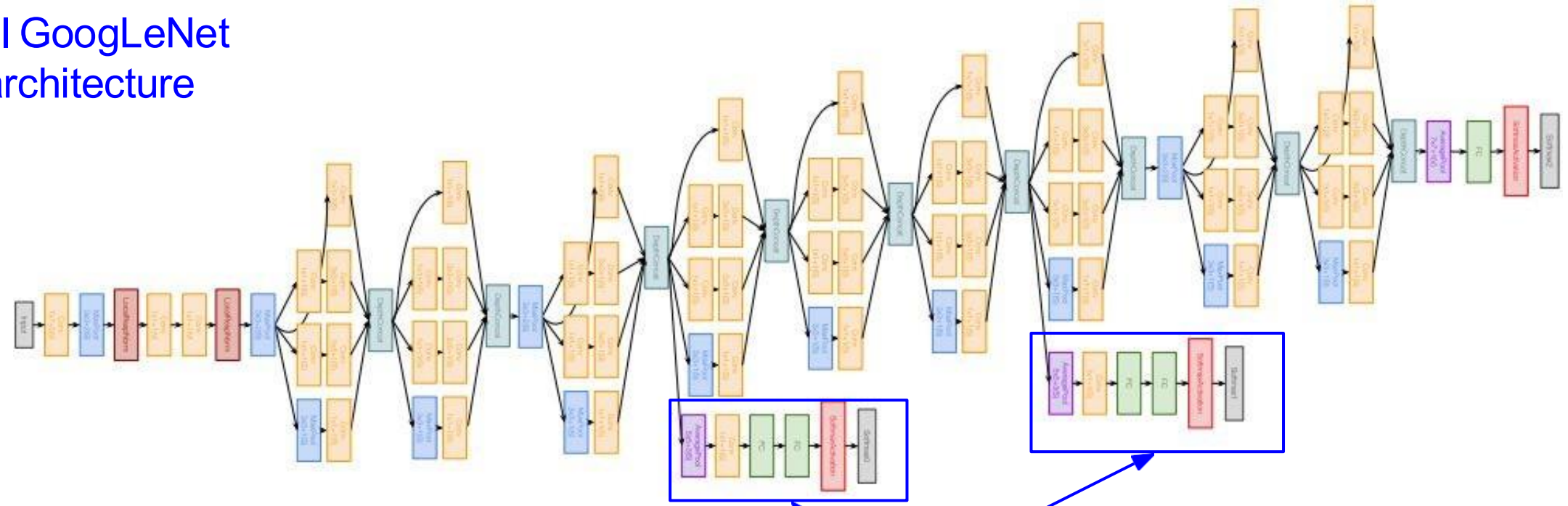


Classifier output
(removed expensive FC layers!)

# Case Study: GoogLeNet

***[Szegedy et al., 2014]***

Full GoogLeNet architecture

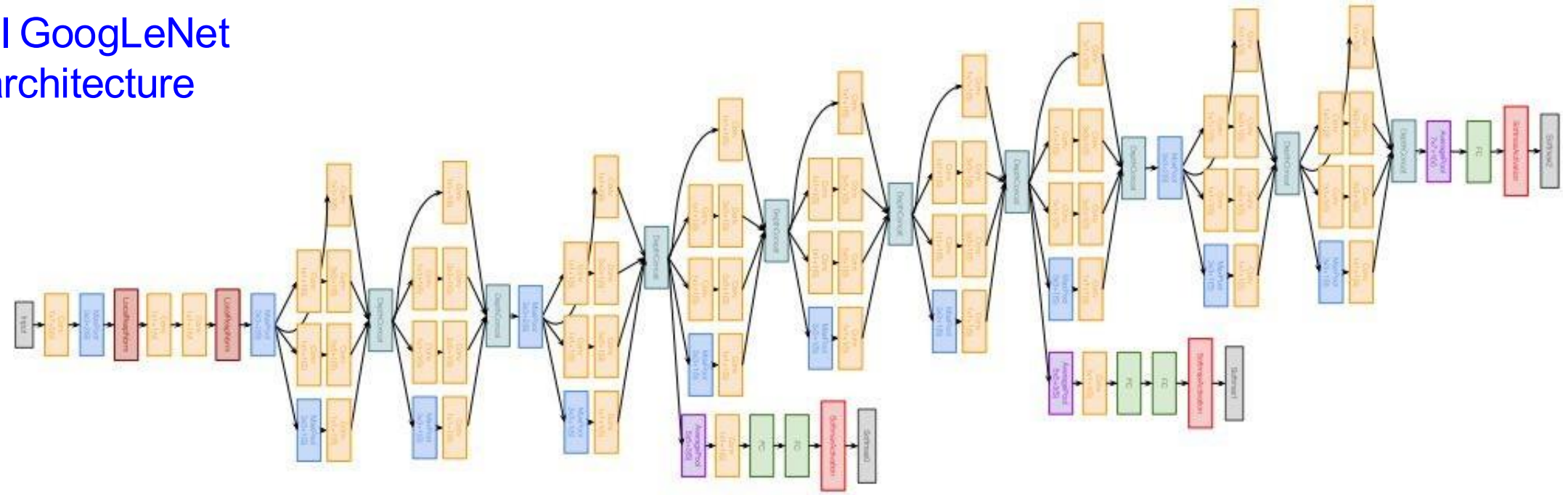


Auxiliary classification outputs to inject additional gradient at lower layers (AvgPool-1x1Conv-FC-FC-Softmax)

# Case Study: GoogLeNet

***[Szegedy et al., 2014]***

Full GoogLeNet architecture



22 total layers with weights

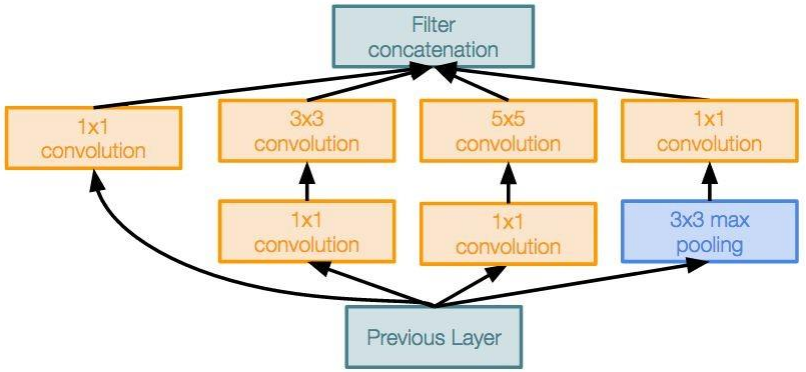
(parallel layers count as 1 layer => 2 layers per Inception module. Don't count auxiliary output layers)

# Case Study: GoogLeNet

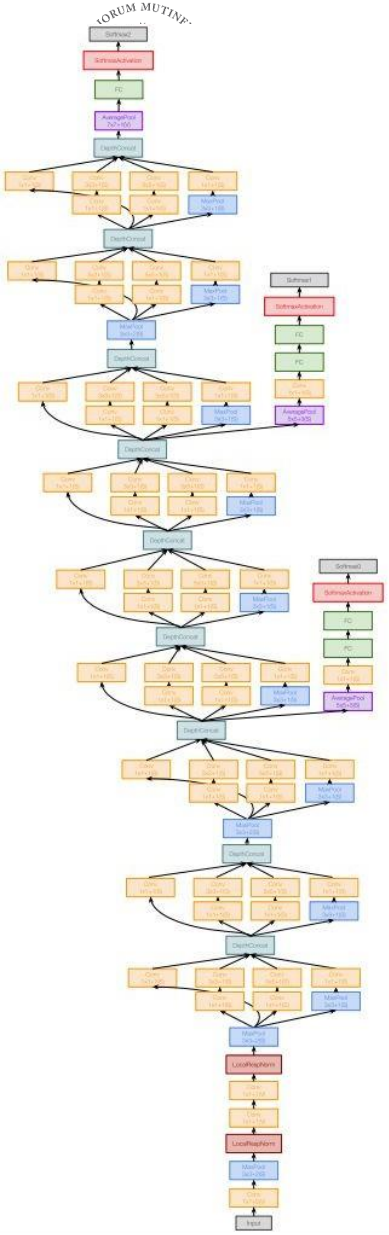
## *[Szegedy et al., 2014]*

Deeper networks, with computational efficiency

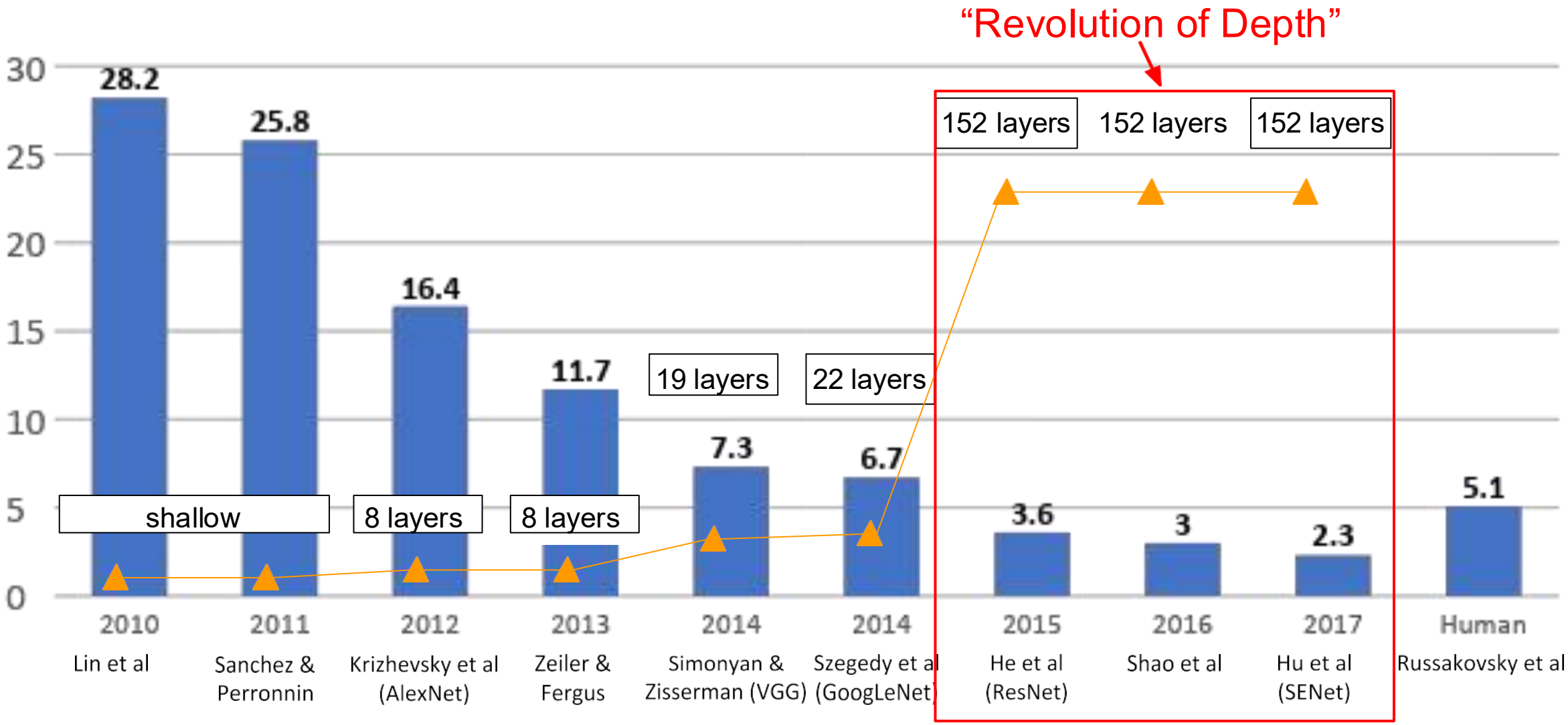
- 22 layers
- Efficient “Inception” module
- No FC layers
- 12x less params than AlexNet
- ILSVRC’14 classification winner (6.7% top 5 error)




Inception module



# ImageNet Large Scale Visual Recognition Challenge (ILSVRC) winners

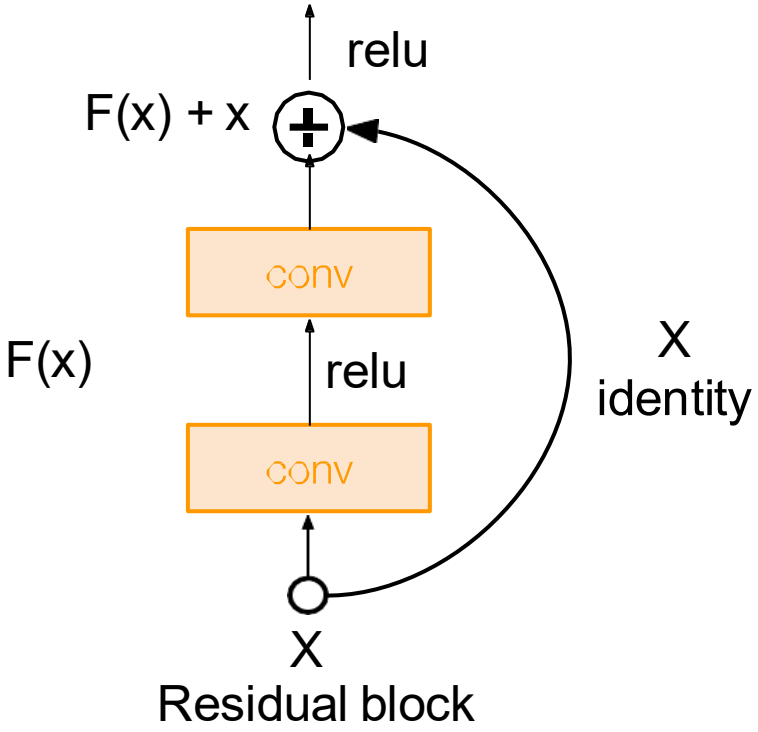


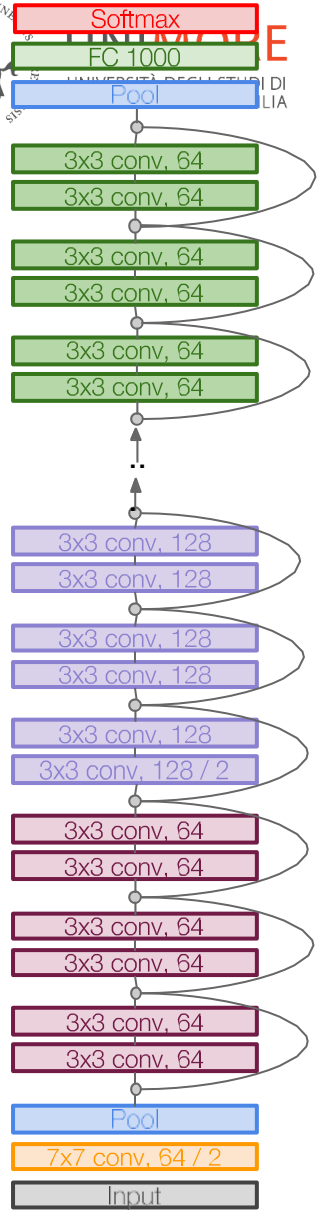
# Case Study: ResNet

## *[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!

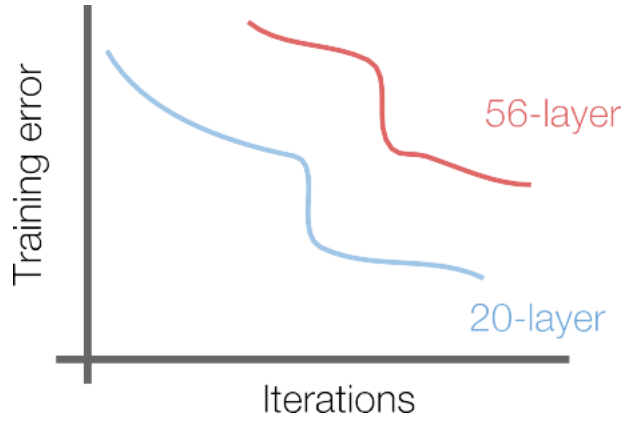


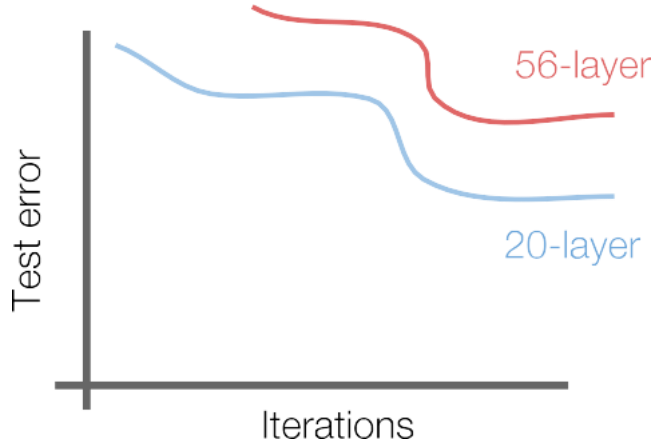



# Case Study: ResNet

***[He et al., 2015]***

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



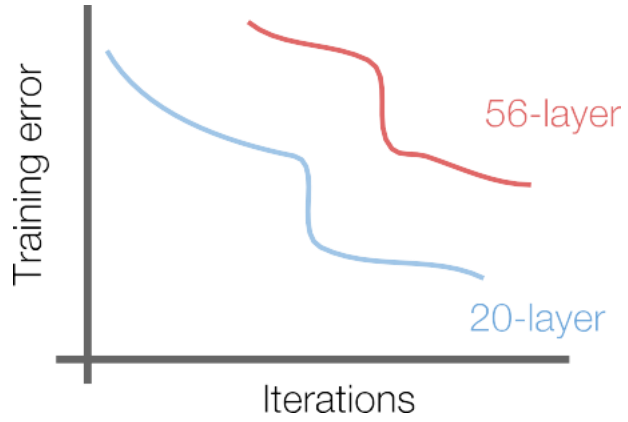



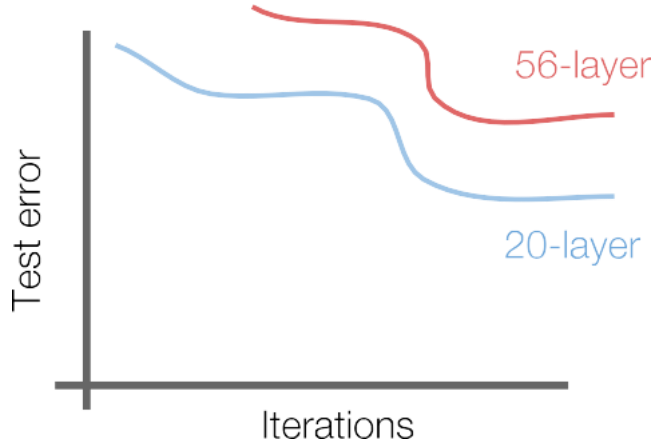

Q: What’s strange about these training and test curves?
[Hint: look at the order of the curves]

# Case Study: ResNet

***[He et al., 2015]***

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

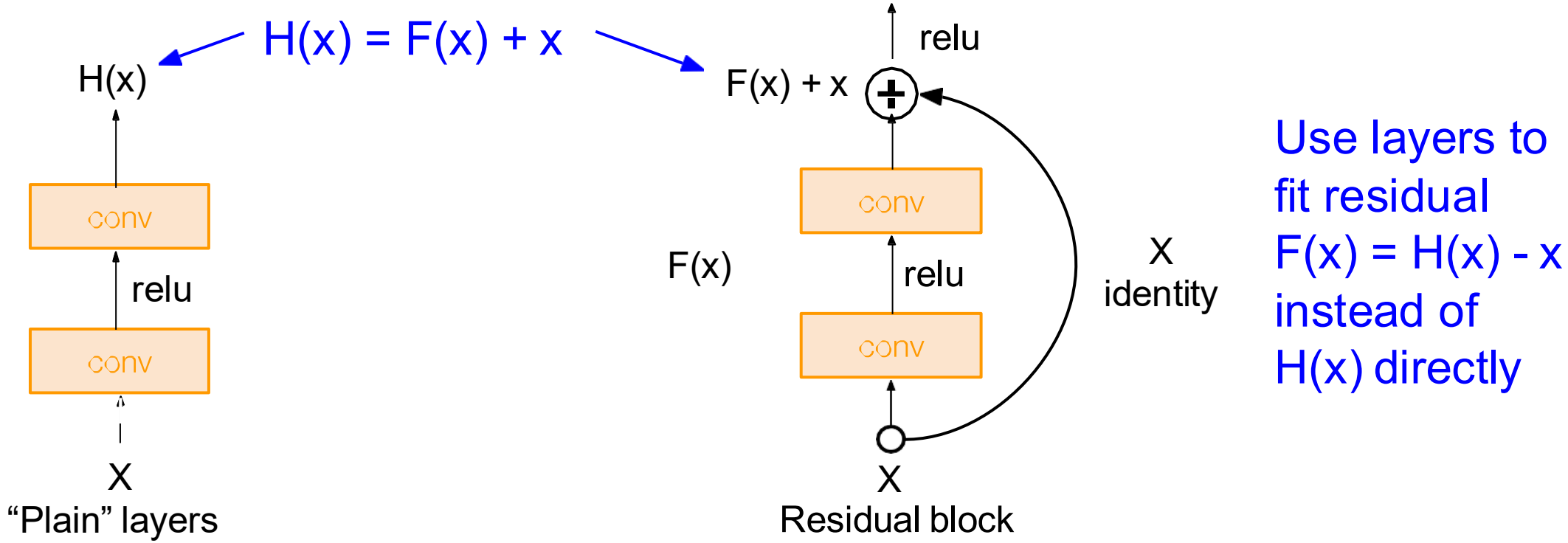
The deeper model should be able to perform at least as well as the shallower model.

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

# Case Study: ResNet

## *[He et al., 2015]*

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



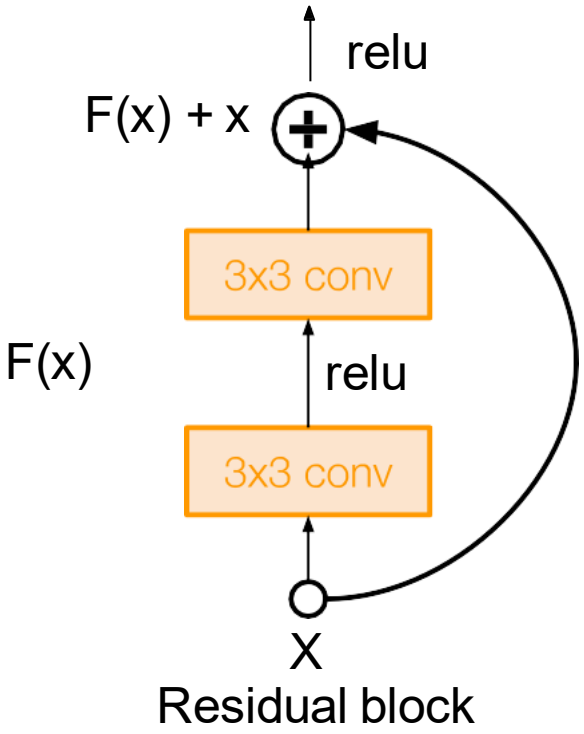

Use layers to fit residual $F(x) = H(x) - x$ instead of $H(x)$ directly

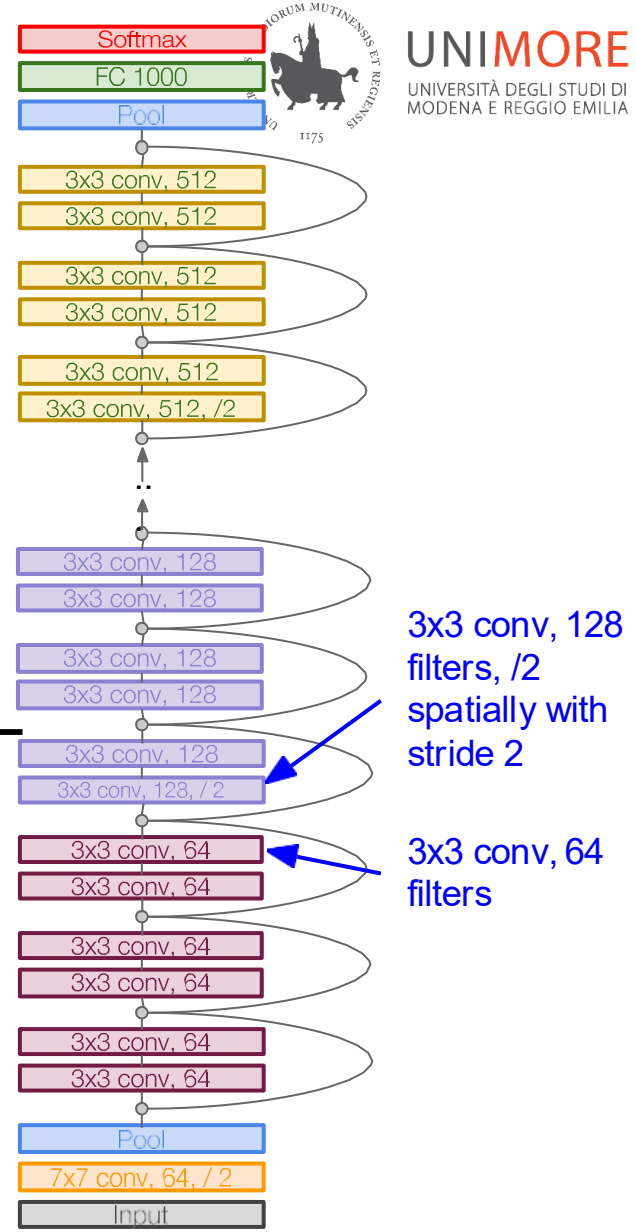
# Case Study: ResNet

## *[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)



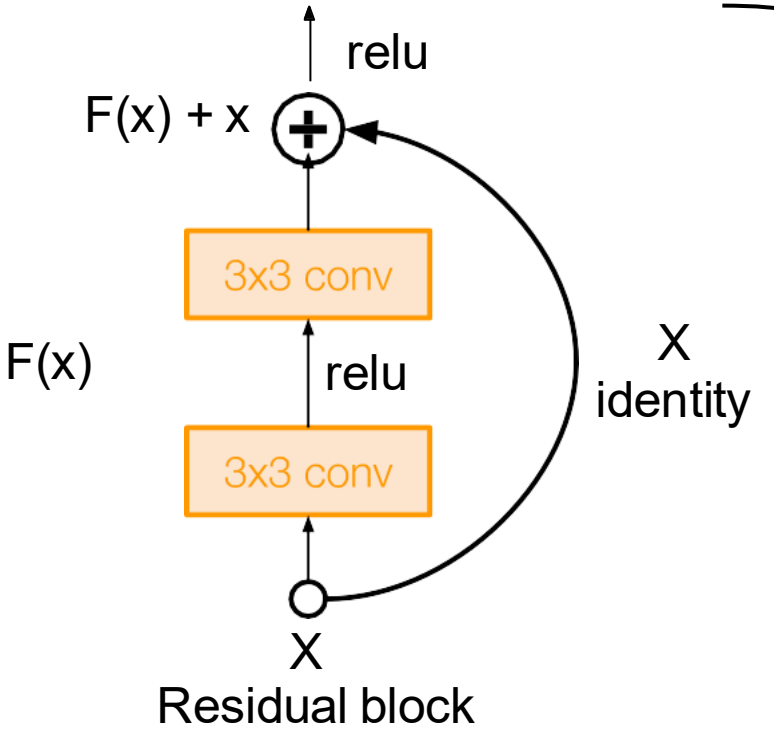

UNIMORE
UNIVERSITÀ DEGLI STUDI DI MODENA E REGGIO EMILIA

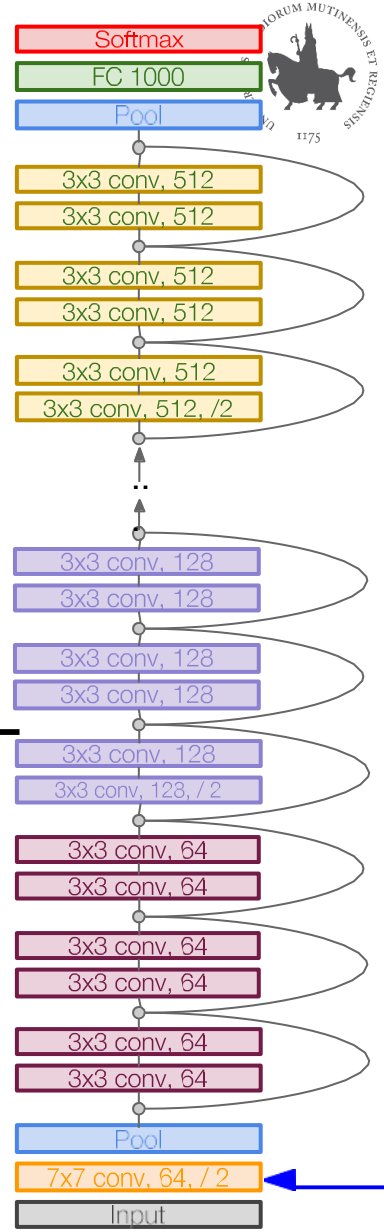
# Case Study: ResNet

## *[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



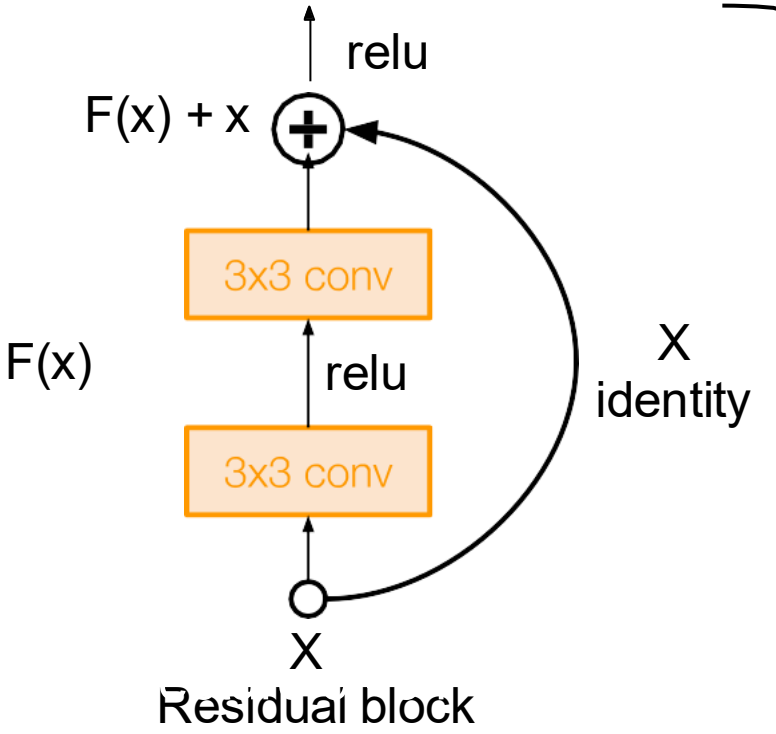



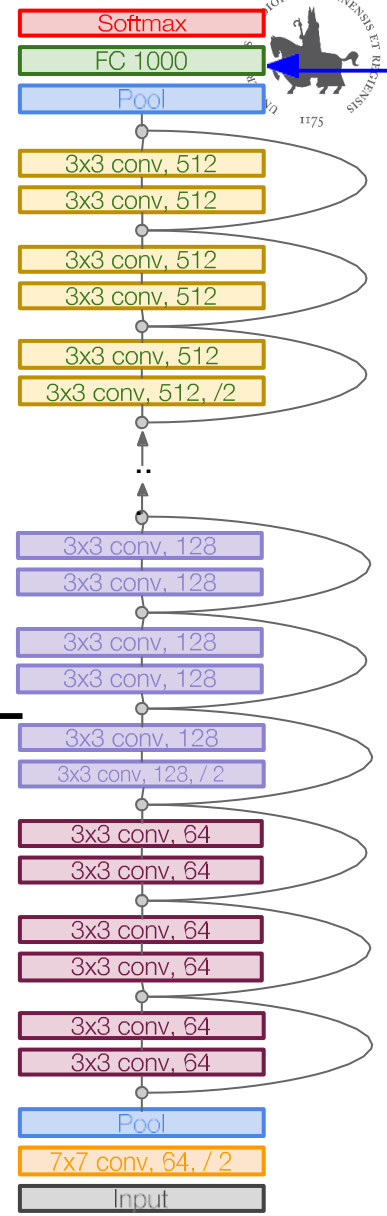
# Case Study: ResNet

## *[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)



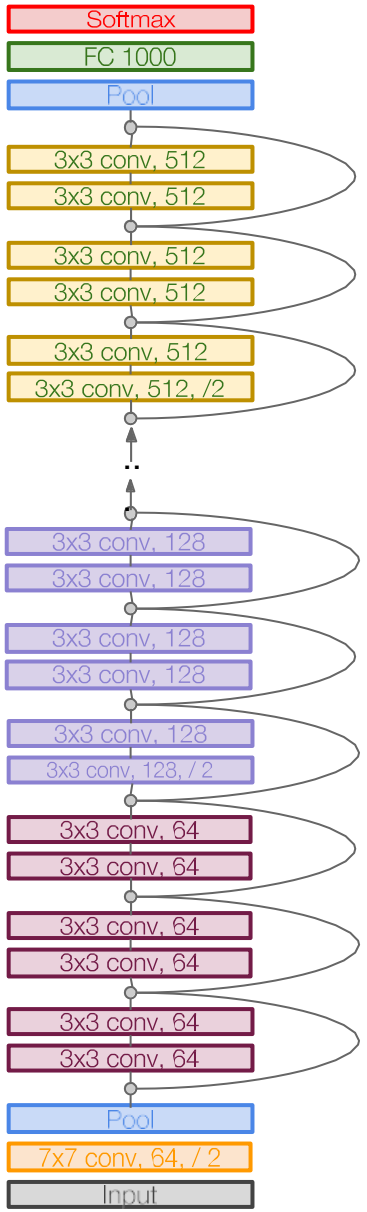

No FC layers besides FC 1000 to output classes

Global average pooling layer after last conv layer

# Case Study: ResNet

***[He et al., 2015]***

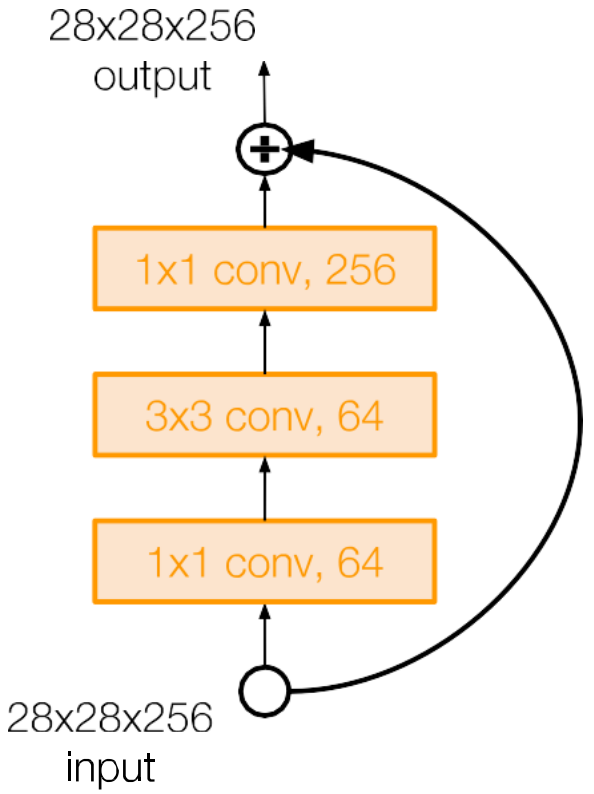
Total depths of 34, 50, 101, or
152 layers for ImageNet



# Case Study: ResNet

***[He et al., 2015]***

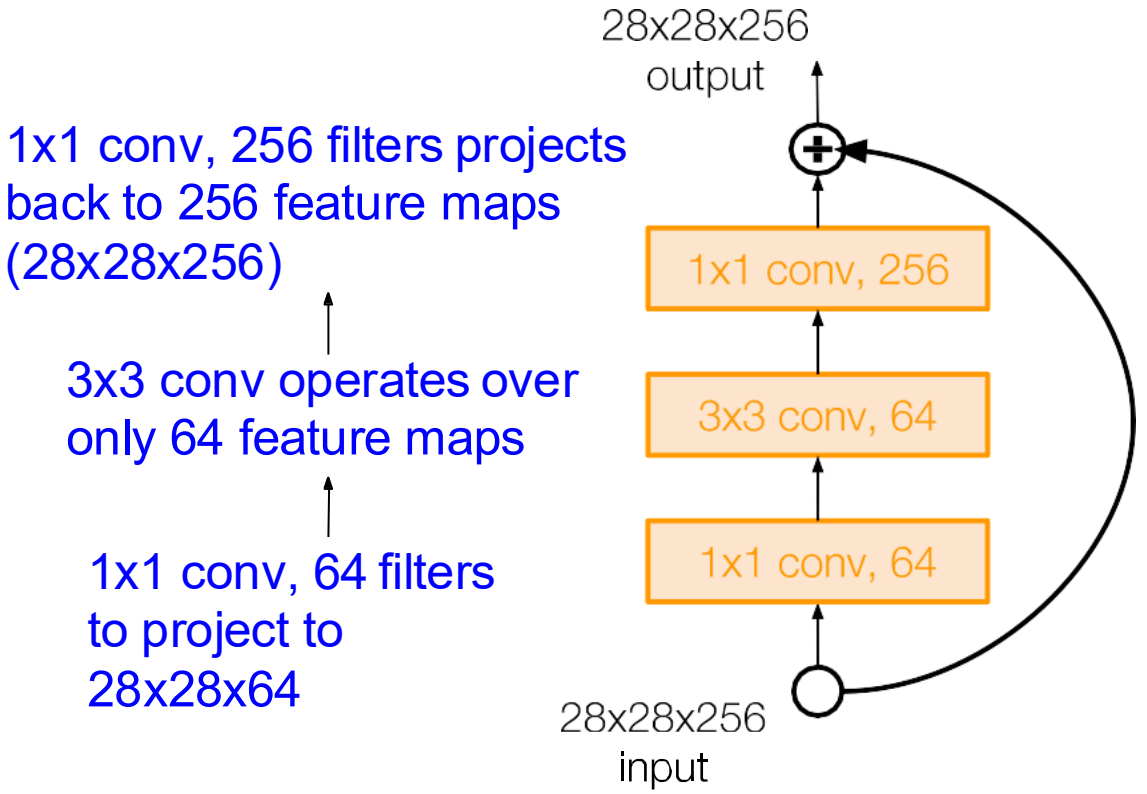
For deeper networks (ResNet-50+), use “bottleneck” layer to improve efficiency (similar to GoogLeNet)



# Case Study: ResNet

*[He et al., 2015]*

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



# Case Study: ResNet

## *[He et al., 2015]*

Experimental Results

- Able to train very deep networks without degrading (152 layers on ImageNet, 1202 on Cifar)
- Deeper networks now achieve lowing training error as expected
- Swept 1st place in all ILSVRC and COCO 2015 competitions

# Case Study: ResNet

## *[He et al., 2015]*

Experimental Results

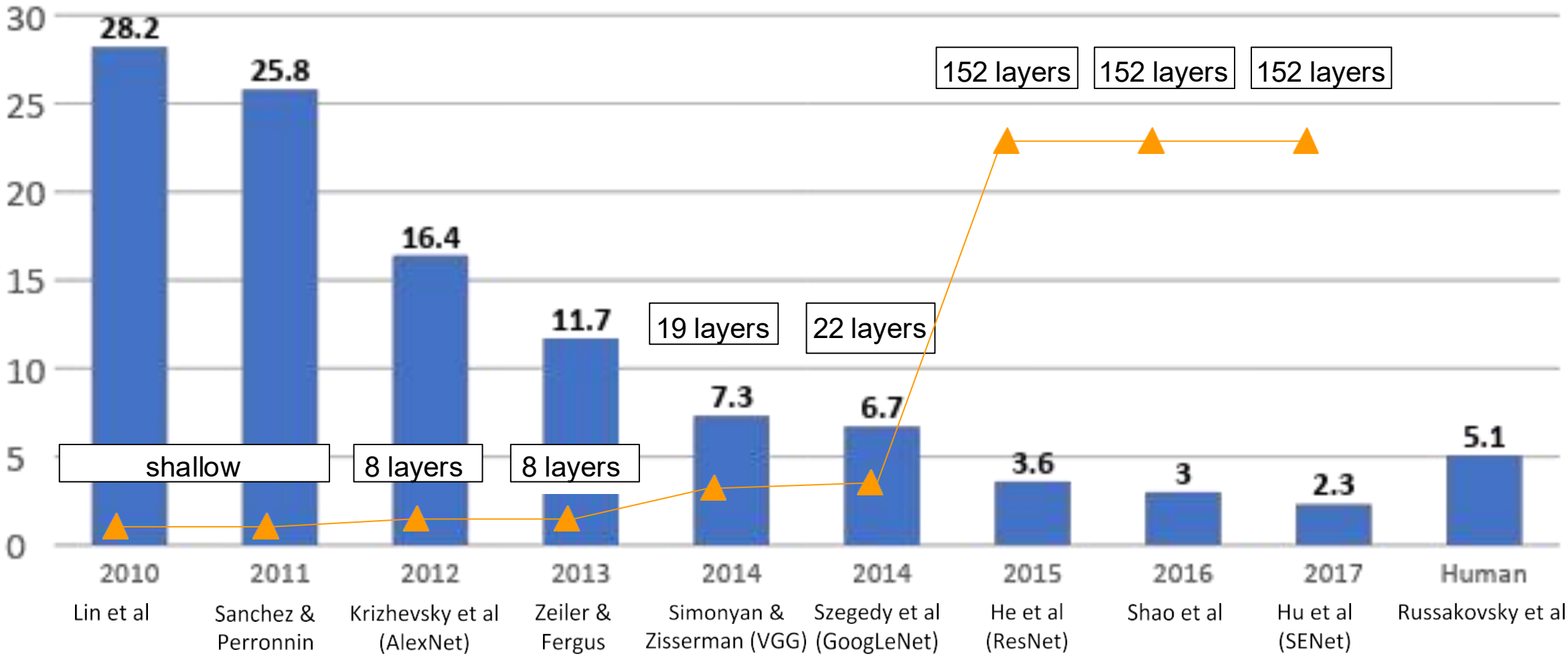
- Able to train very deep networks without degrading (152 layers on ImageNet, 1202 on Cifar)
- Deeper networks now achieve lowing training error as expected
- Swept 1st place in all ILSVRC and COCO 2015 competitions

MSRA @ ILSVRC & COCO 2015 Competitions

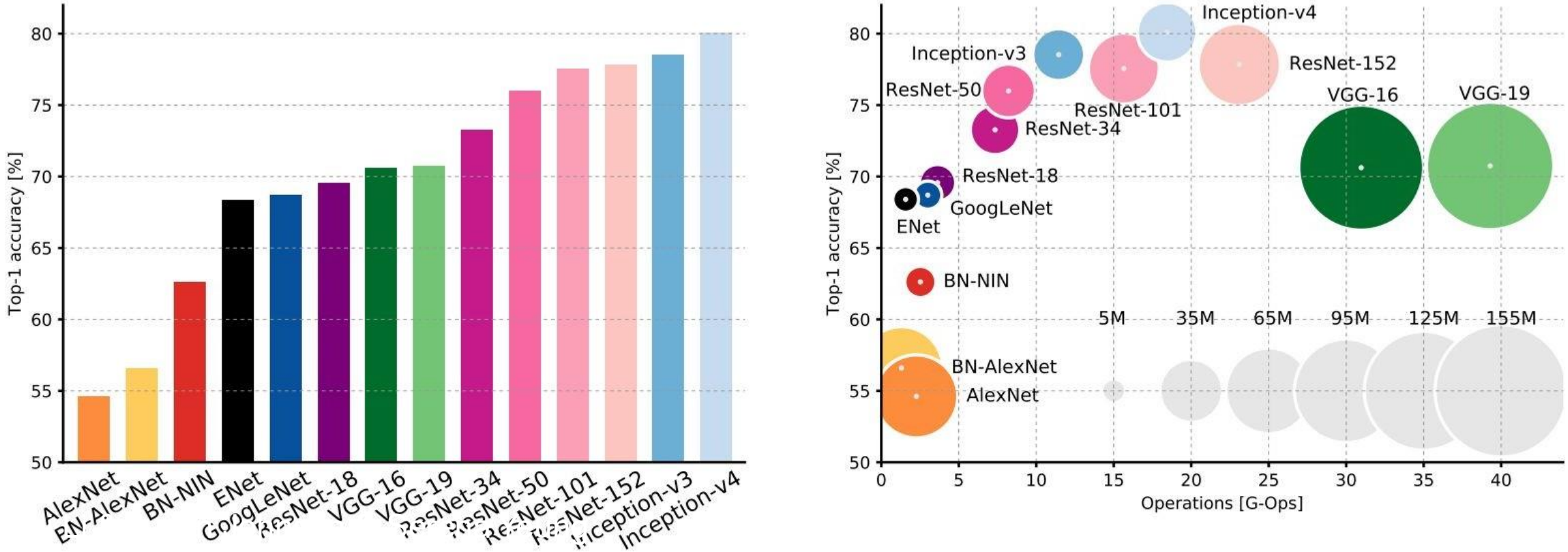
- **1st places in all five main tracks**
  - ImageNet Classification: *"Ultra-deep"* (quote Yann) 152-layer nets
  - ImageNet Detection: 16% better than 2nd
  - ImageNet Localization: 27% better than 2nd
  - COCO Detection: 11% better than 2nd
  - COCO Segmentation: 12% better than 2nd

ILSVRC 2015 classification winner (3.6% top 5 error) -- better than "human performance"! (Russakovsky 2014)

# ImageNet Large Scale Visual Recognition Challenge (ILSVRC) winners



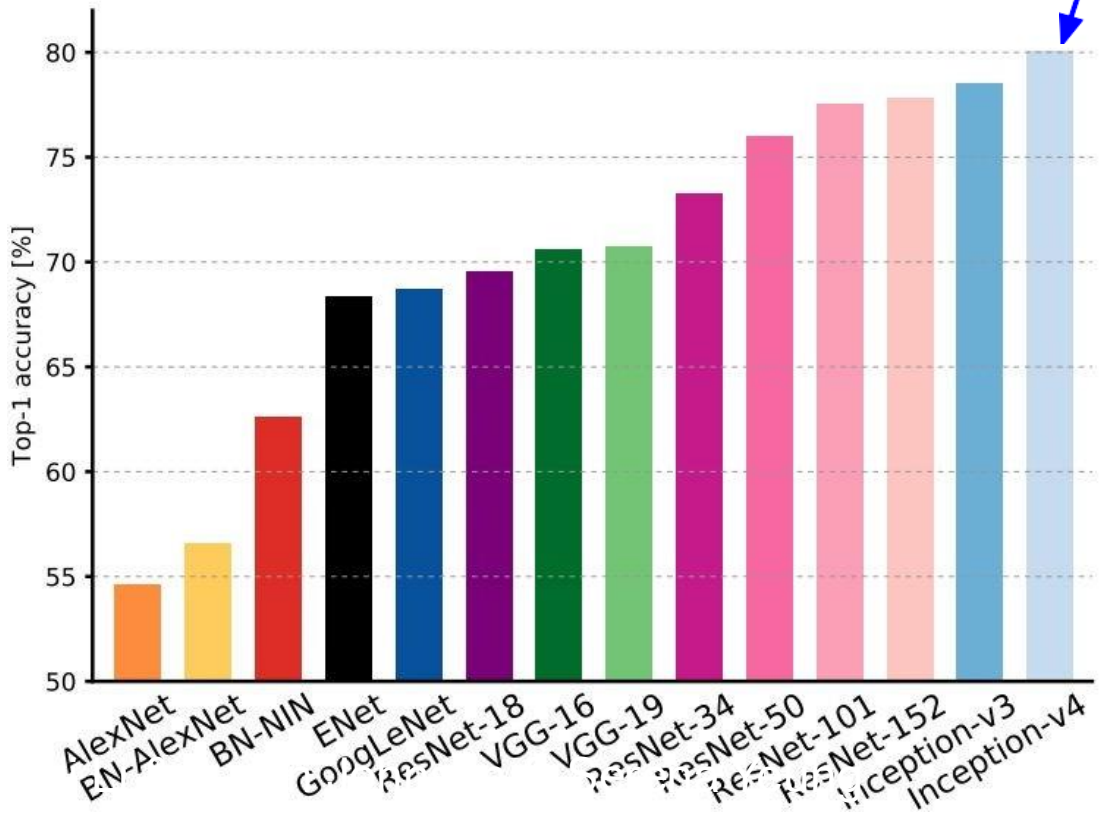
# Comparing complexity...

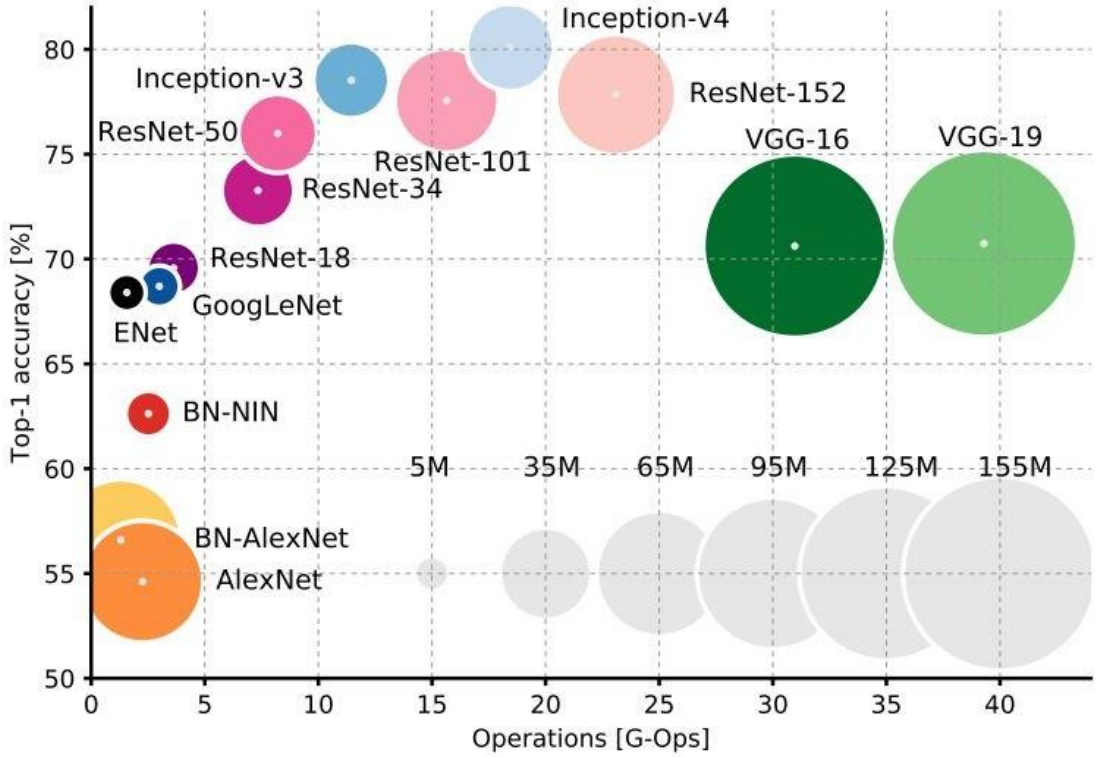



An Analysis of Deep Neural Network Models for Practical Applications, 2017.

# Comparing complexity...

Inception-v4: Resnet + Inception!

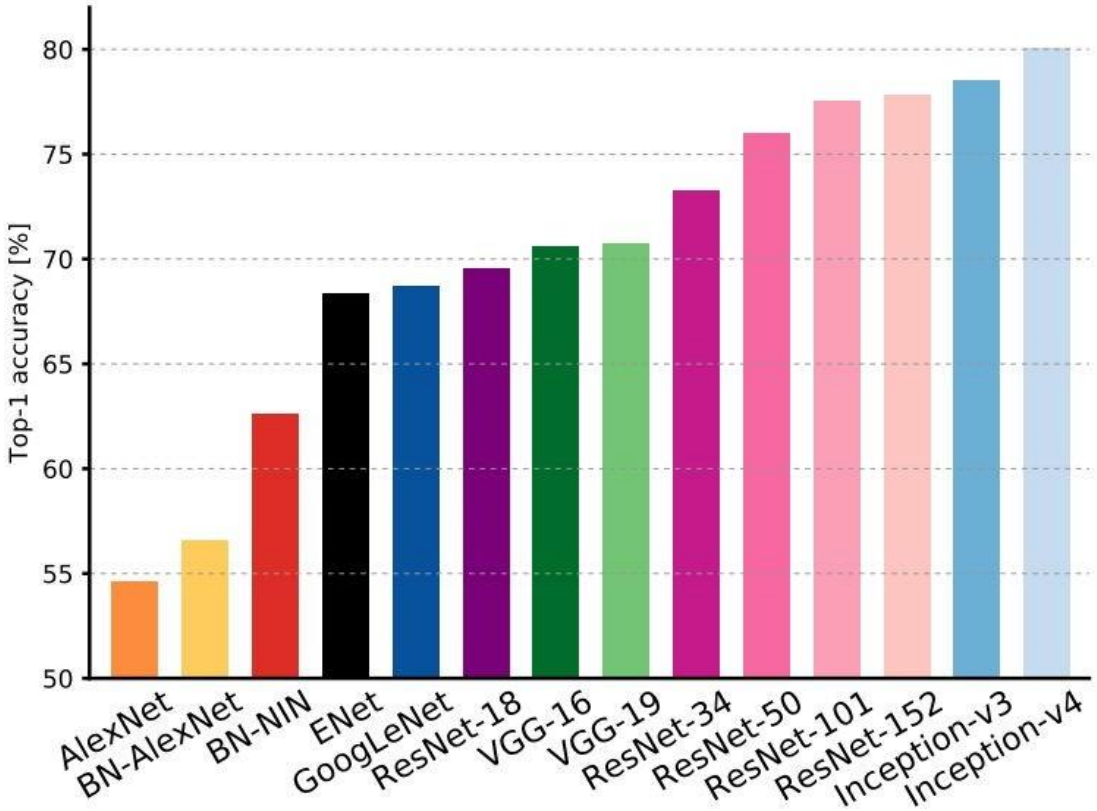


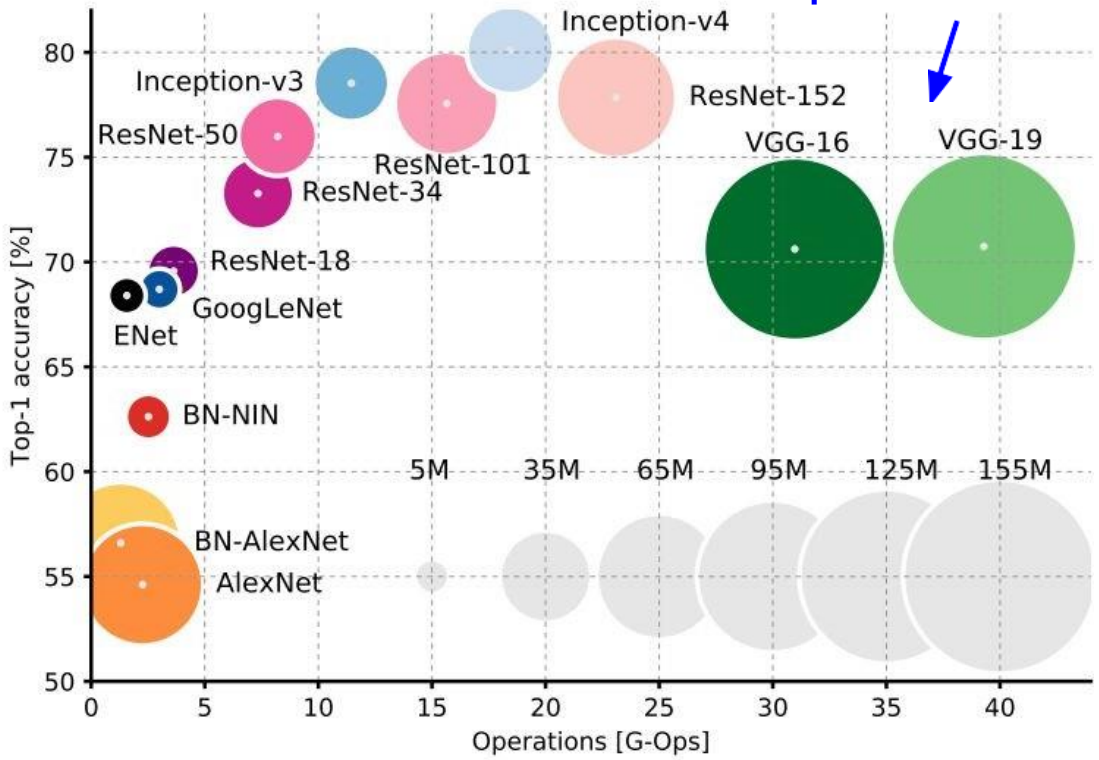




An Analysis of Deep Neural Network Models for Practical Applications, 2017.

# Comparing complexity...

VGG: Highest memory, most operations

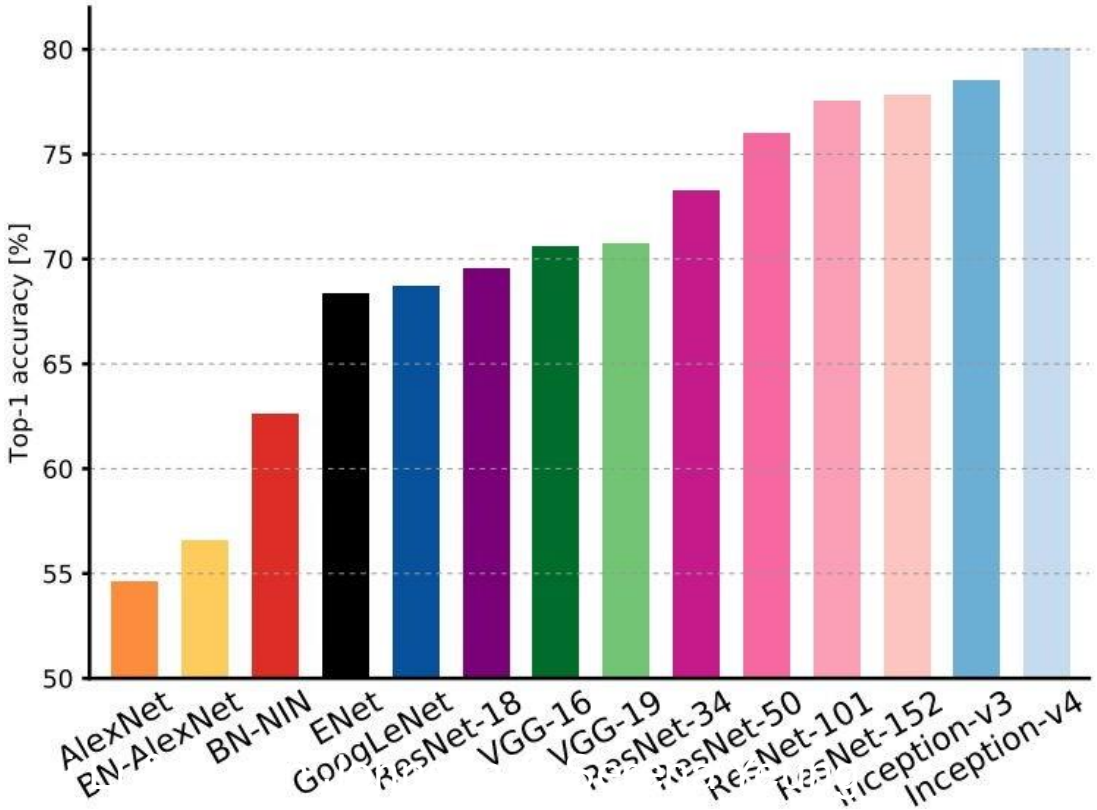


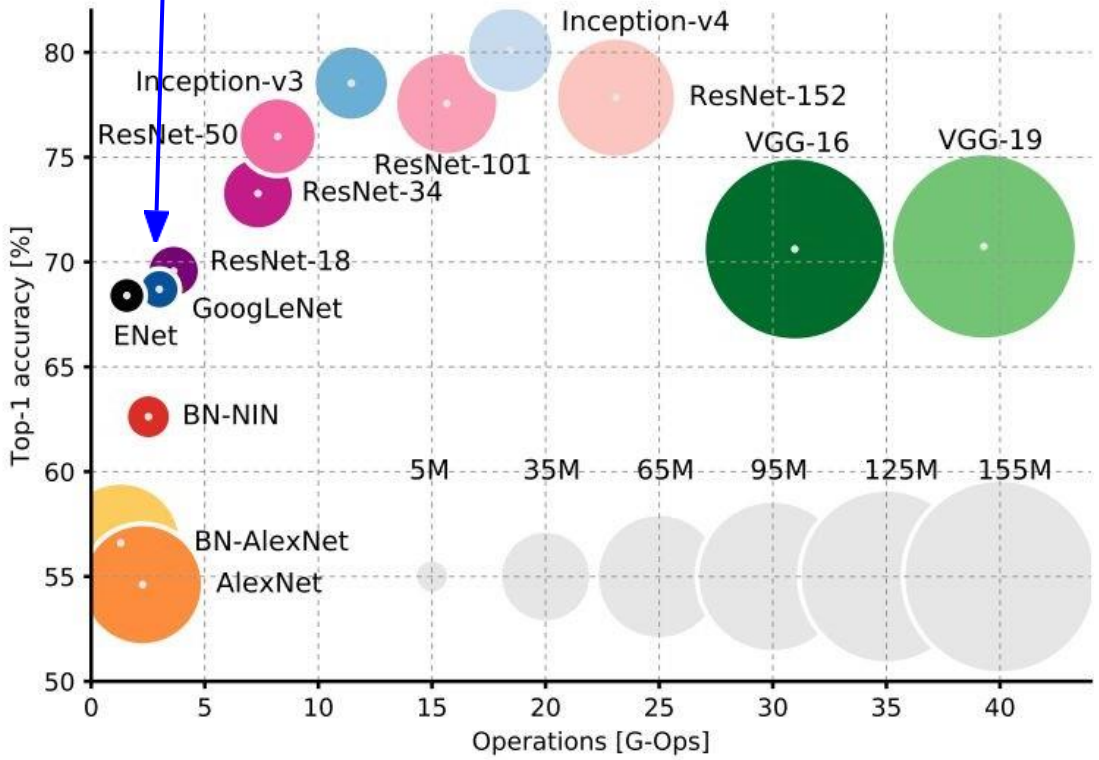




An Analysis of Deep Neural Network Models for Practical Applications, 2017.

# Comparing complexity...

GoogLeNet:
most efficient

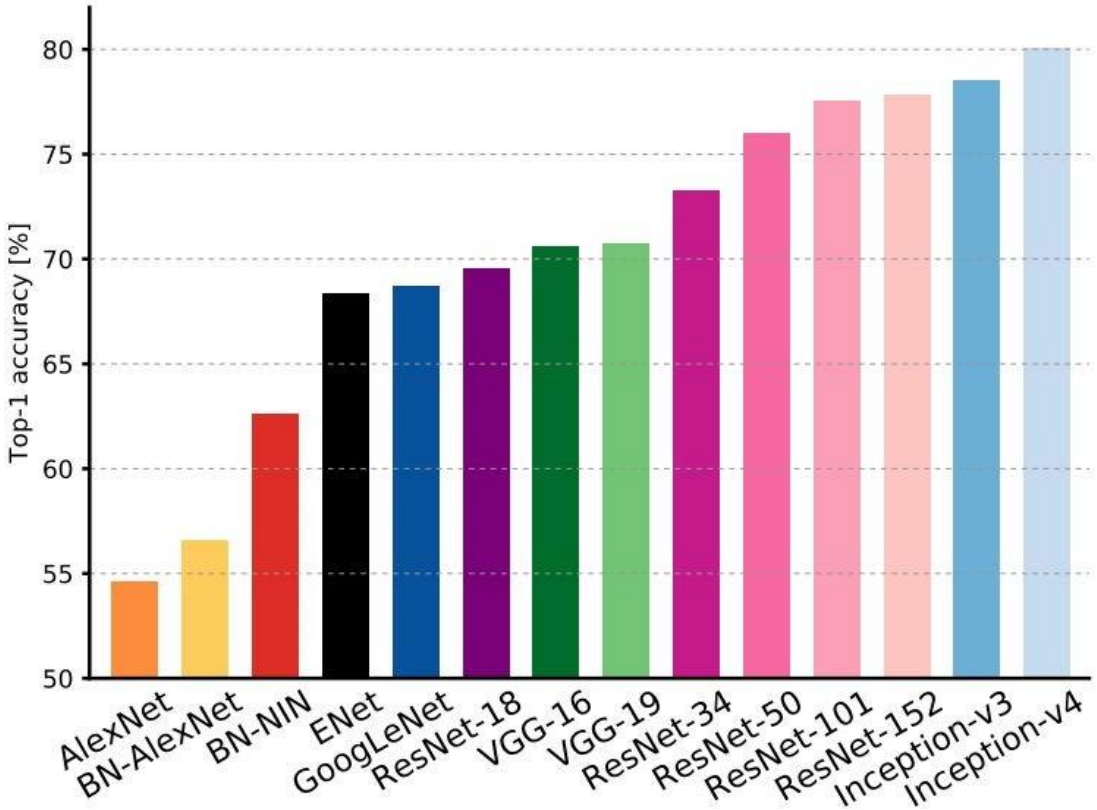


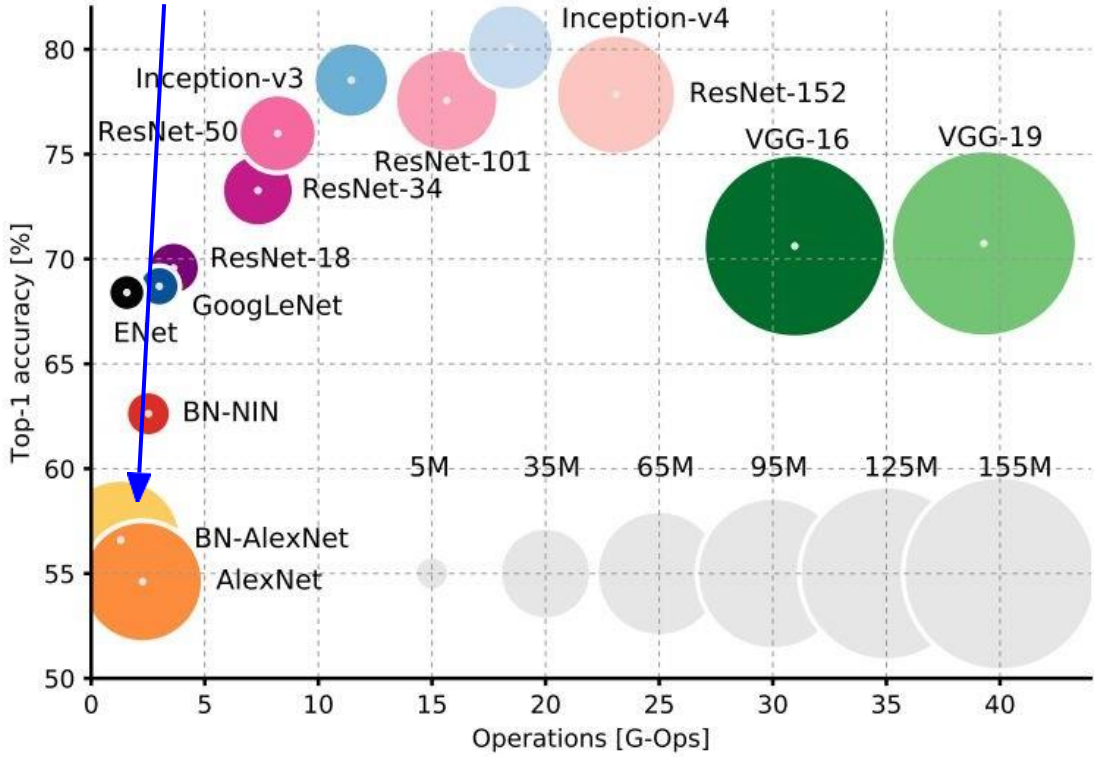




An Analysis of Deep Neural Network Models for Practical Applications, 2017.

# Comparing complexity...

AlexNet:
Smaller compute, still memory heavy, lower accuracy

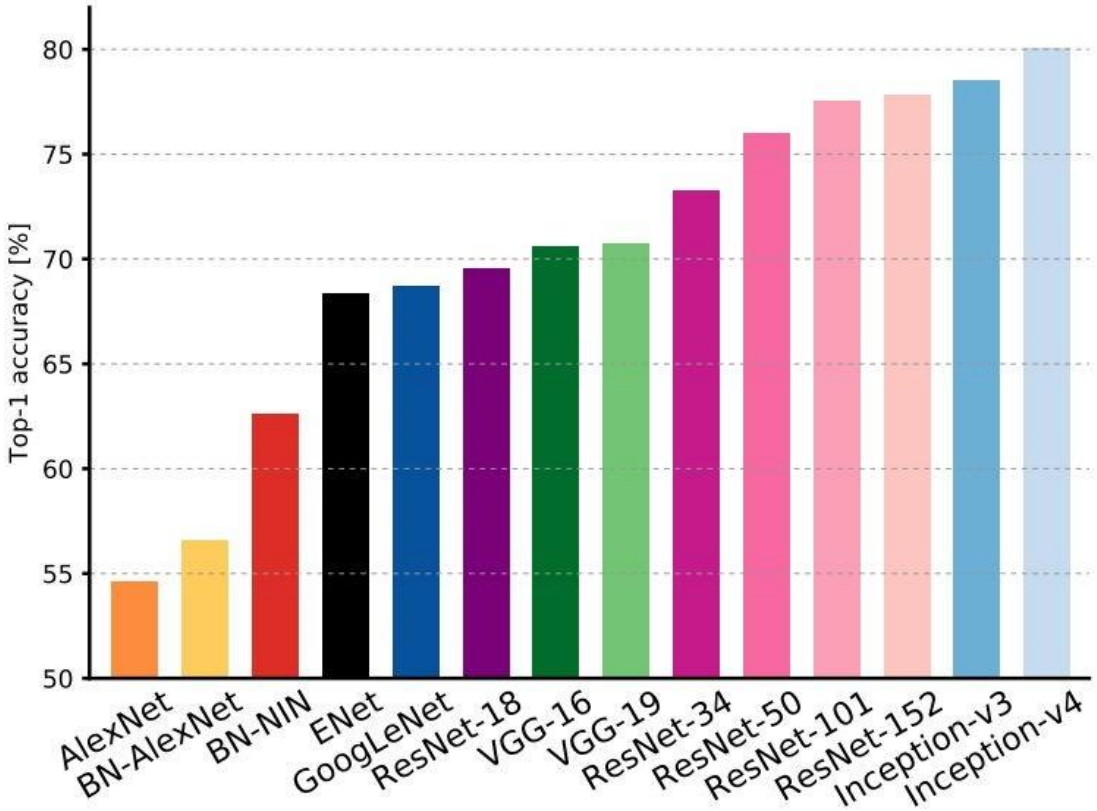


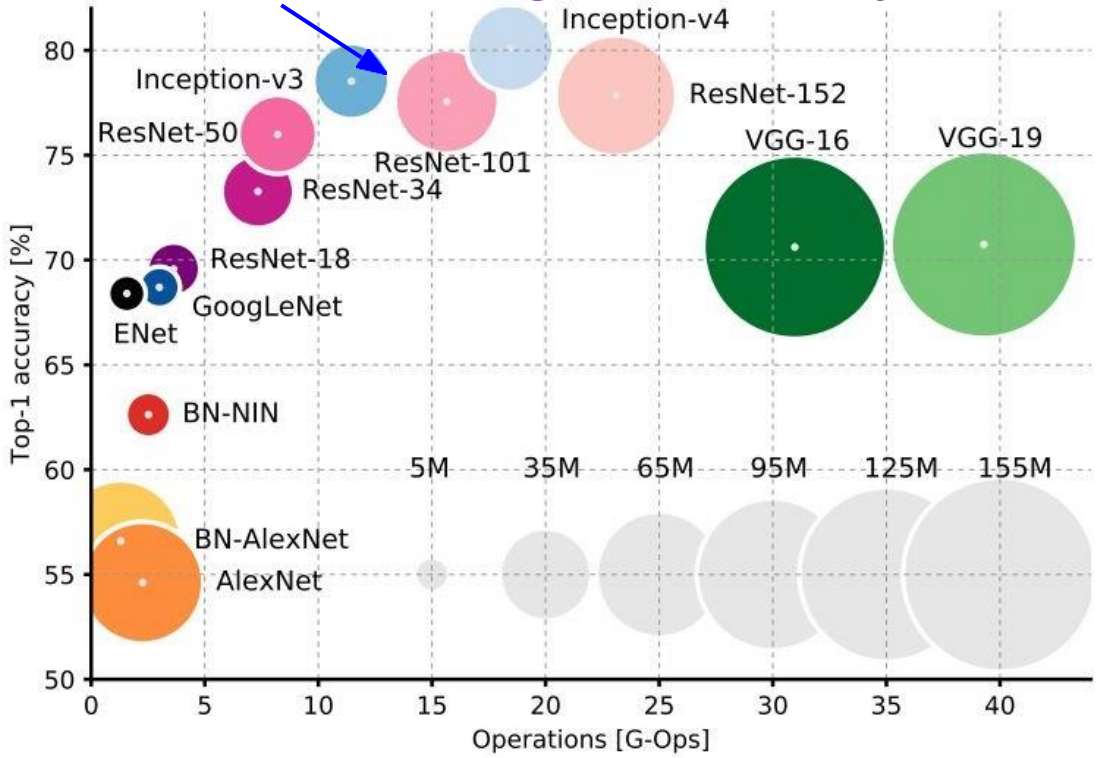




An Analysis of Deep Neural Network Models for Practical Applications, 2017.

# Comparing complexity...

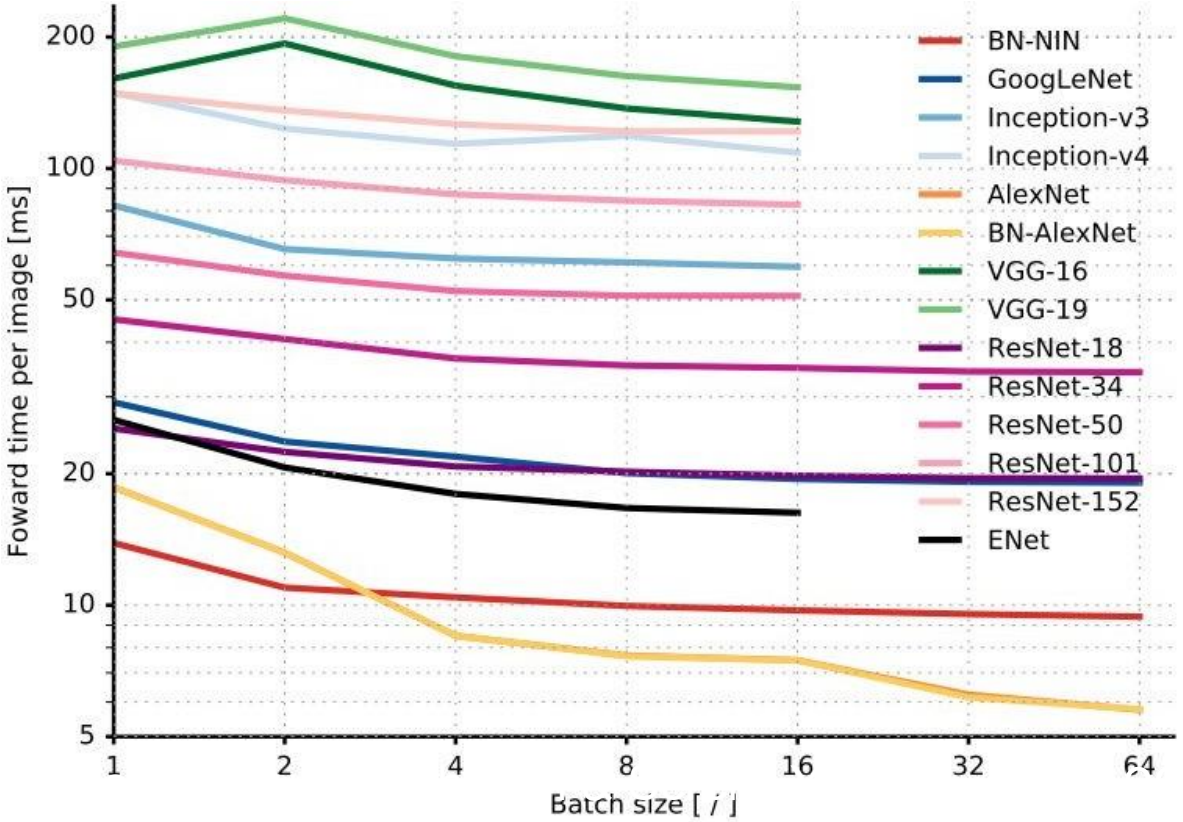
ResNet:
Moderate efficiency depending on model, highest accuracy

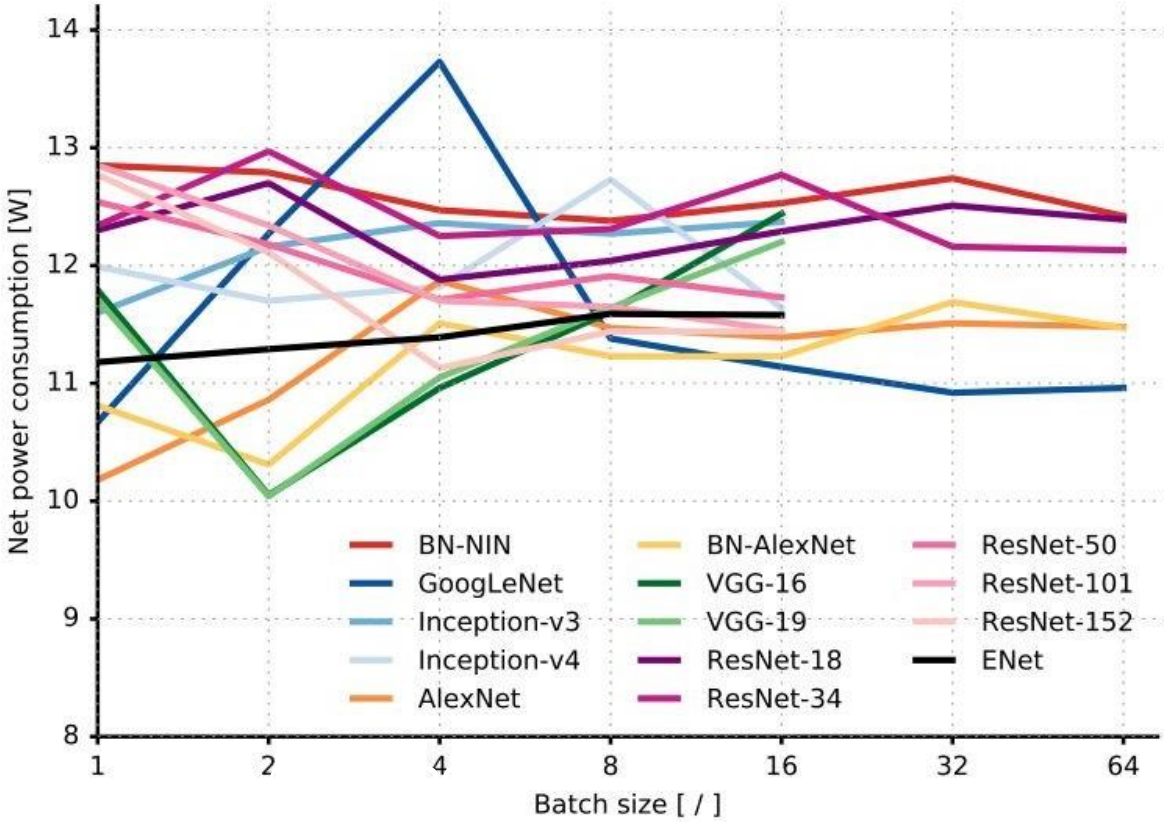



An Analysis of Deep Neural Network Models for Practical Applications, 2017.

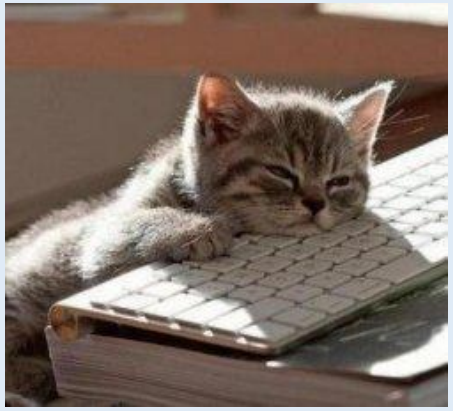
# Forward pass time and power consumption

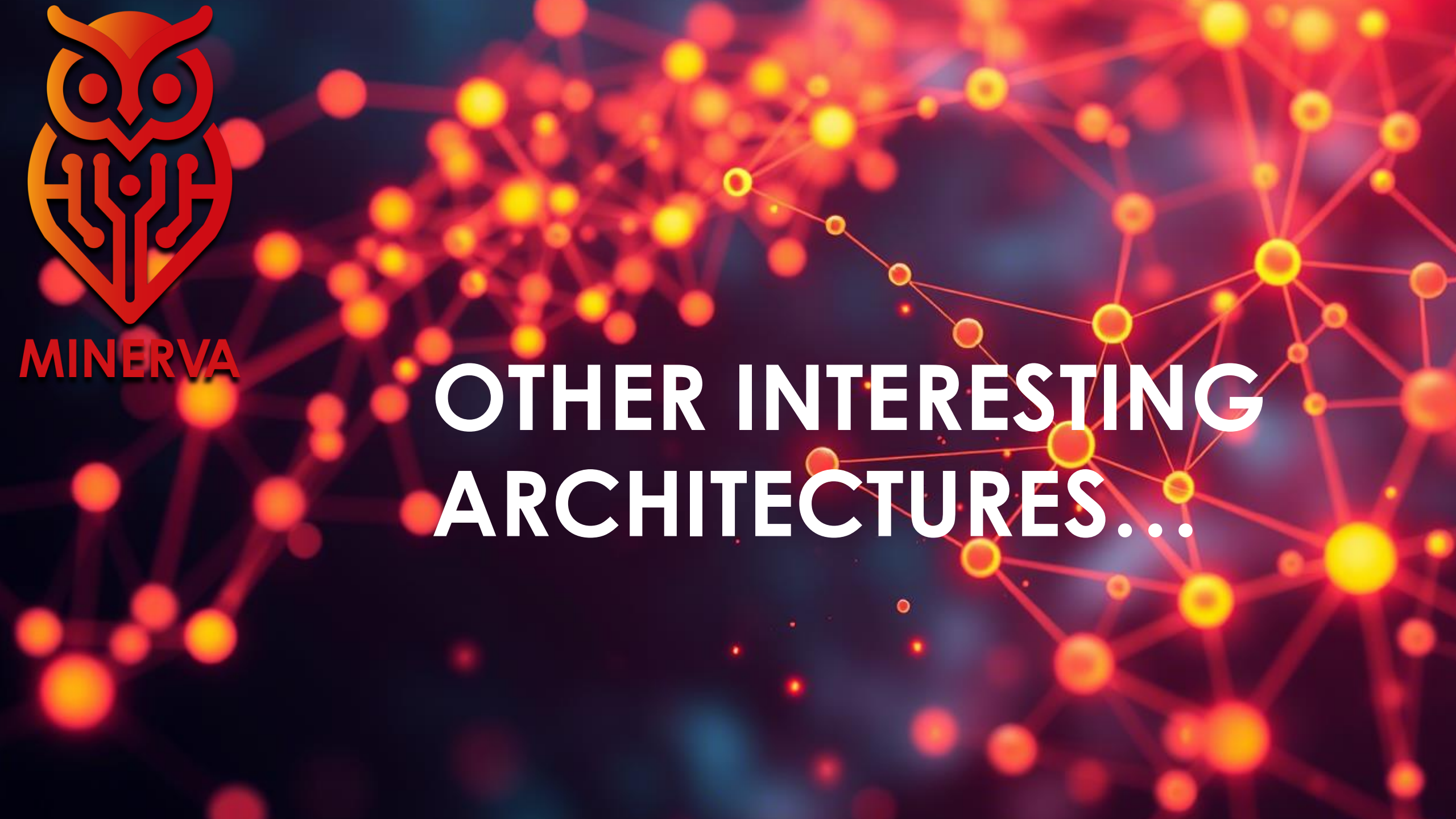



An Analysis of Deep Neural Network Models for Practical Applications, 2017.


# BREAK?



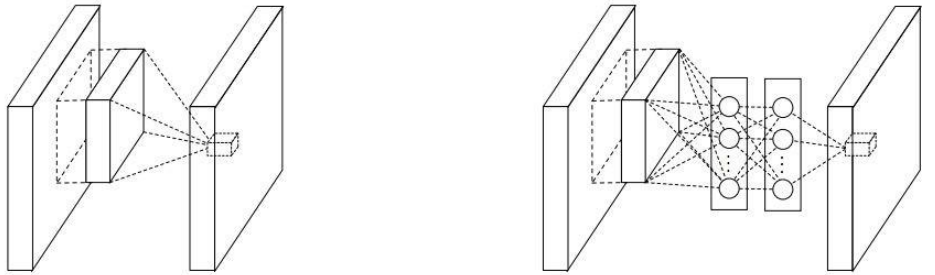



# OTHER INTERESTING ARCHITECTURES…

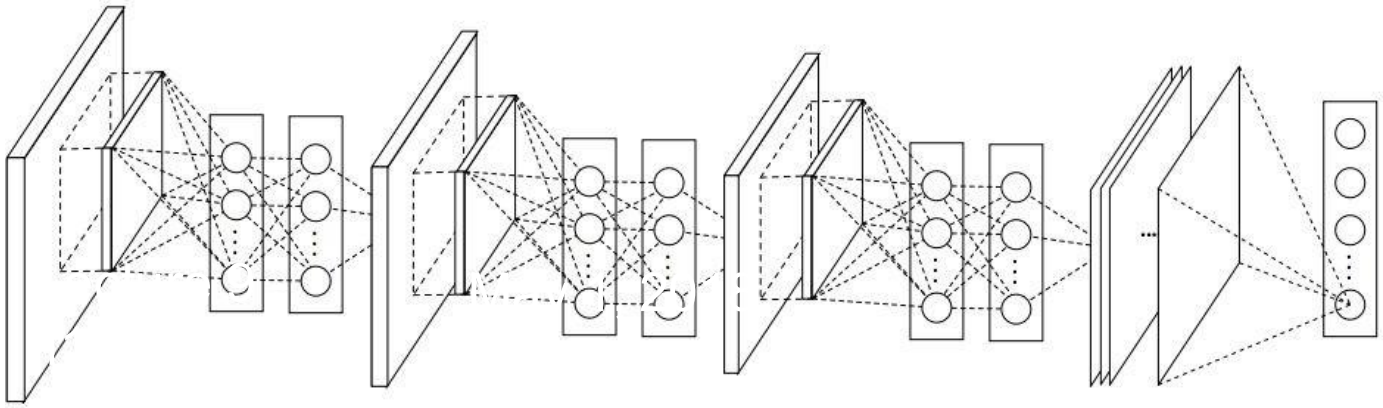
# Network in Network (NiN)

*[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



(a) Linear convolution layer (b) Mlpconv layer

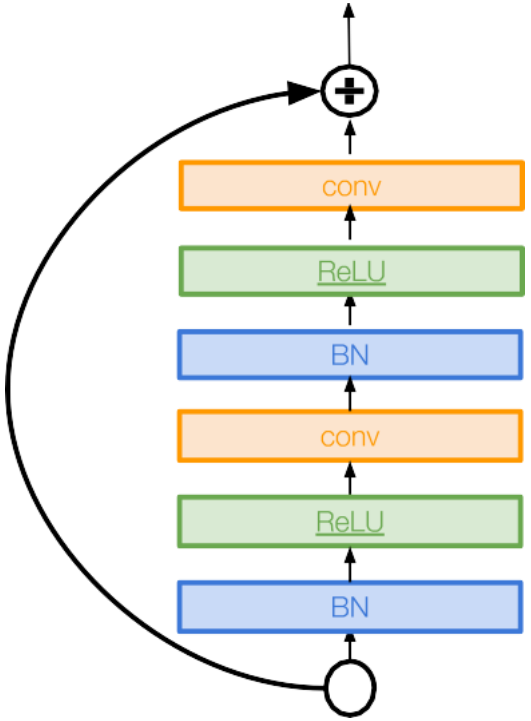


Figures copyright Lin et al., 2014. Reproduced with permission.

# Identity Mappings in Deep Residual Networks

*[He et al. 2016]*

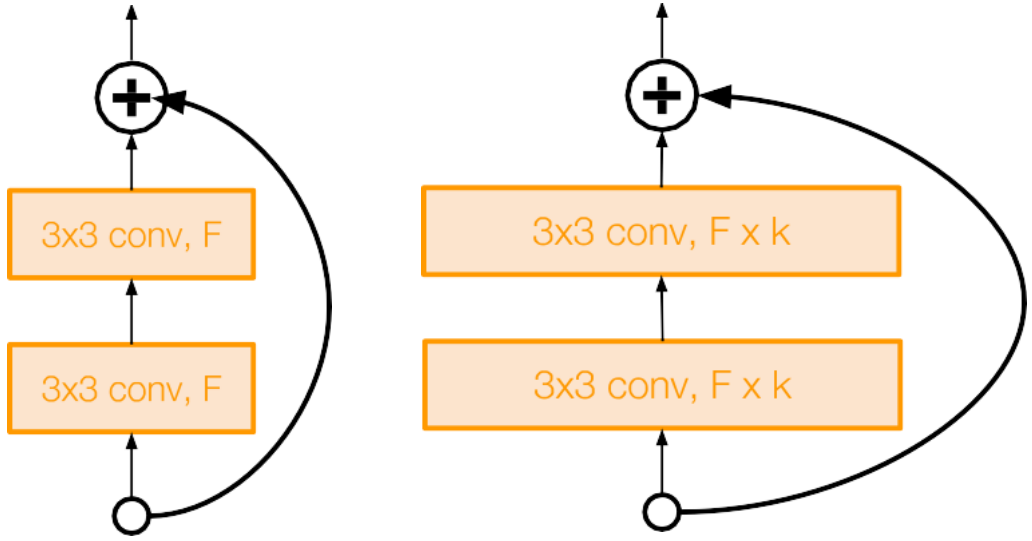
- Improved ResNet block design from creators of ResNet
- Creates a more direct path for propagating information throughout network (moves activation to residual mapping pathway)
- Gives better performance



# Wide Residual Networks

*[Zagoruyko et al. 2016]*

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




Basic residual block

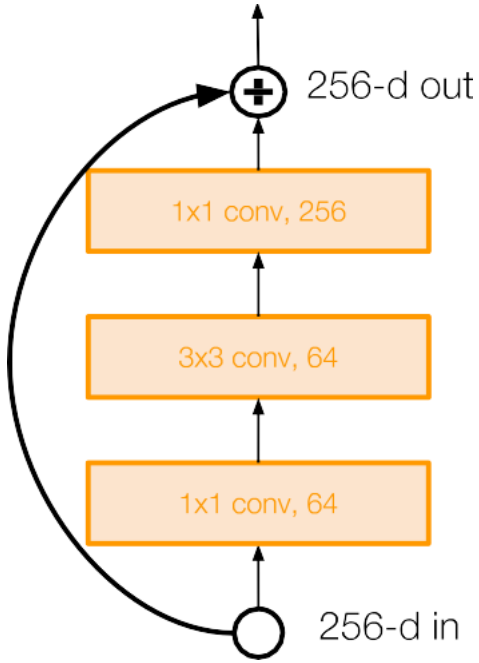
Wide residual block

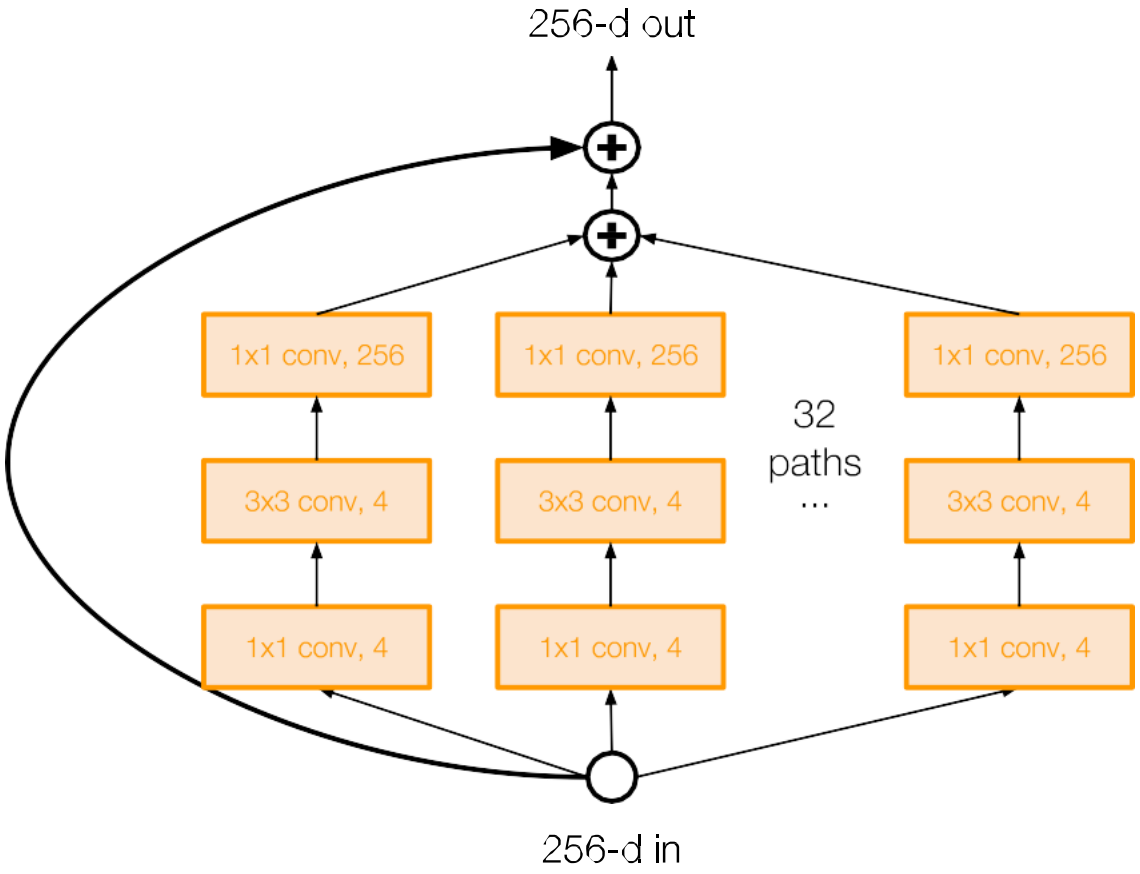
Improving ResNets...

# Aggregated Residual Transformations for Deep Neural Networks (ResNeXt)

*[Xie et al. 2016]*

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

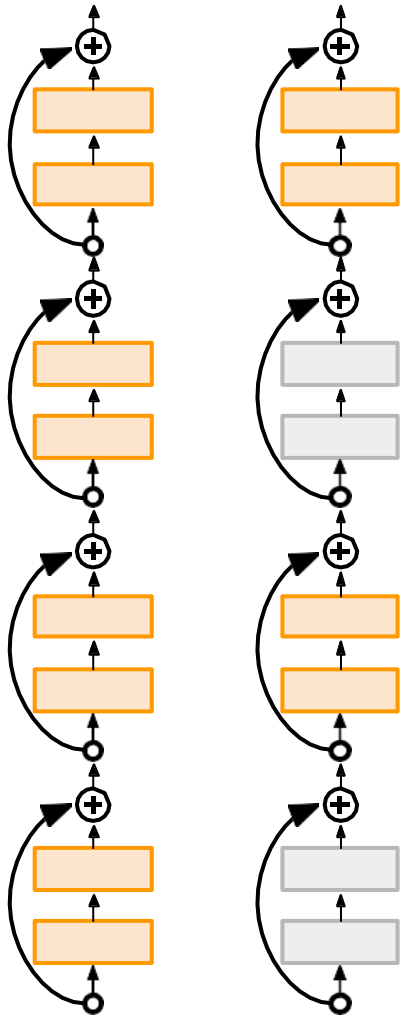


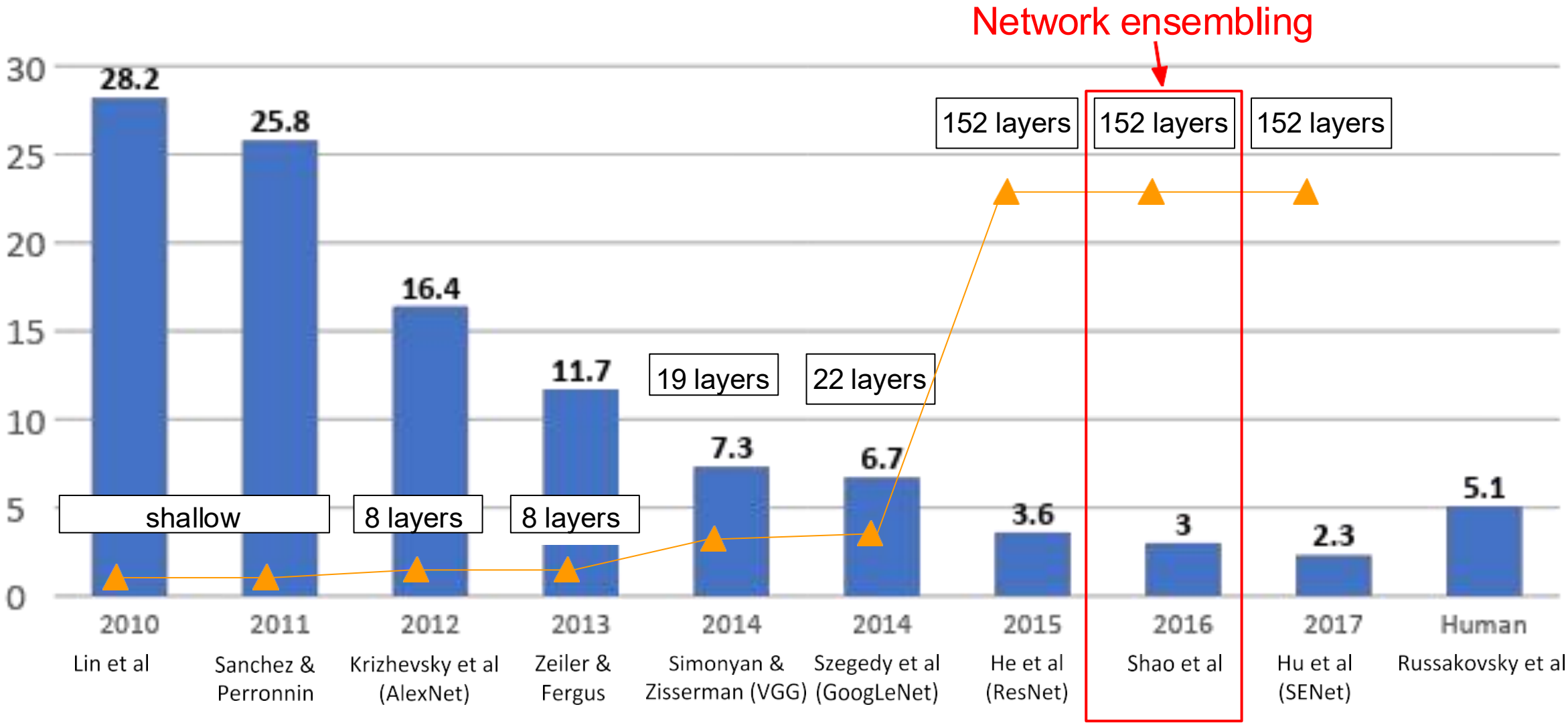
# Deep Networks with Stochastic Depth

*[Huang et al. 2016]*

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



# ImageNet Large Scale Visual Recognition Challenge (ILSVRC) winners



# Improving ResNets...

UNIMORE
UNIVERSITÀ DEGLI STUDI DI MODENA E REGGIO EMILIA

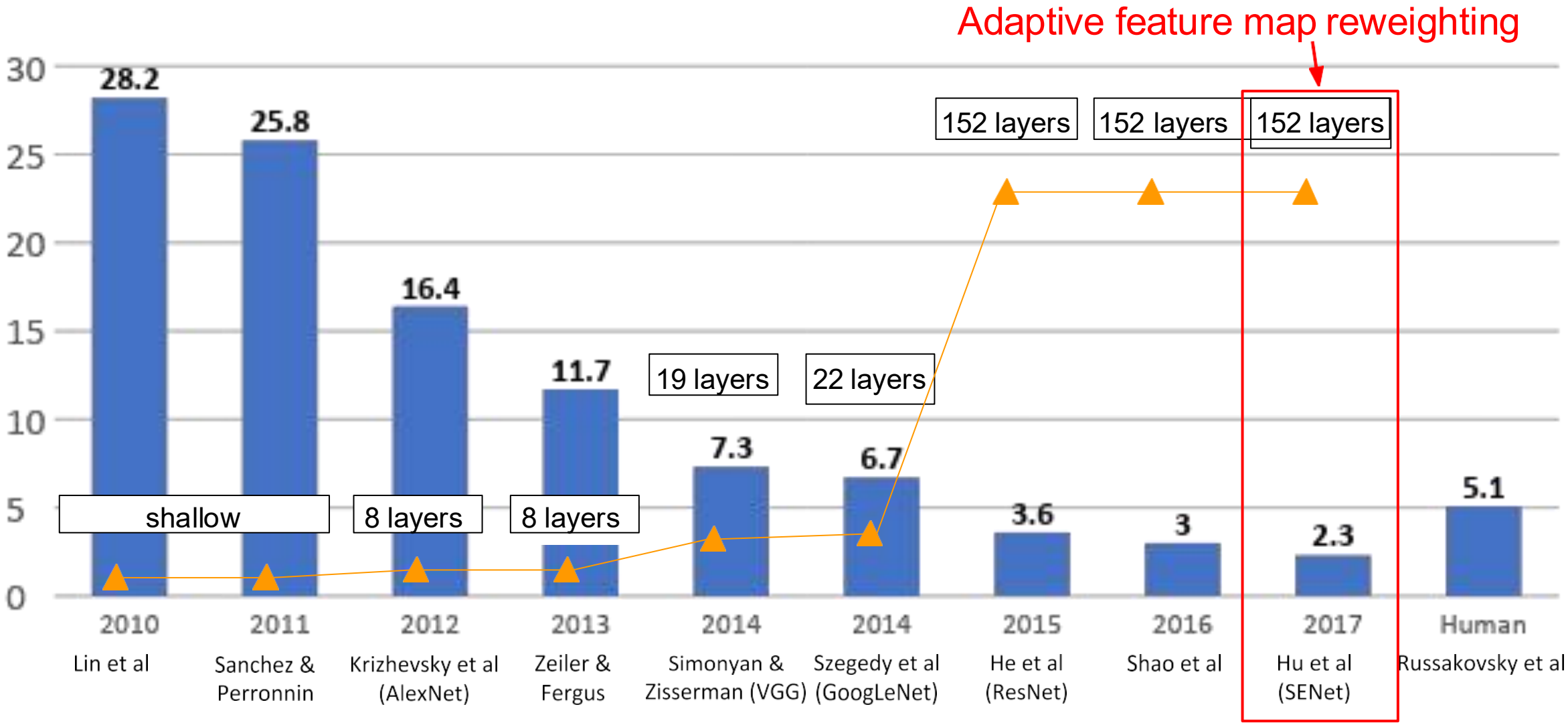
## "Good Practices for Deep Feature Fusion"

*[Shao et al. 2016]*

- Multi-scale ensembling of Inception, Inception-Resnet, Resnet, Wide Resnet models
- ILSVRC'16 classification winner

| | Inception-v3 | Inception-v4 | Inception-Resnet-v2 | Resnet-200 | Wrn-68-3 | Fusion（Val.） | Fusion（Test） |
|---|---|---|---|---|---|---|---|
| Err. (%) | 4.20 | 4.01 | 3.52 | 4.26 | 4.65 | 2.92 (-0.6) | 2.99 |

# ImageNet Large Scale Visual Recognition Challenge (ILSVRC) winners

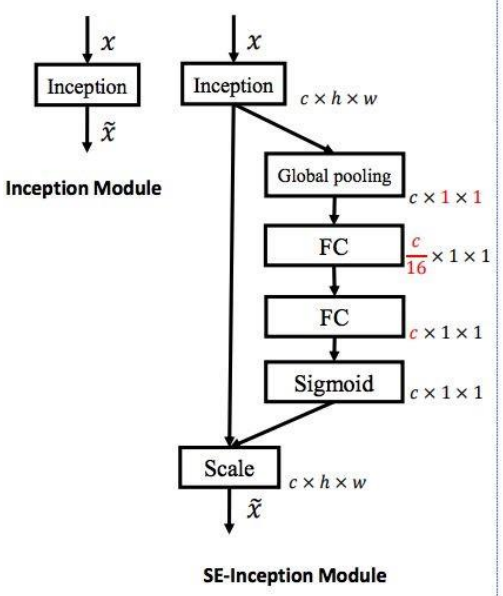


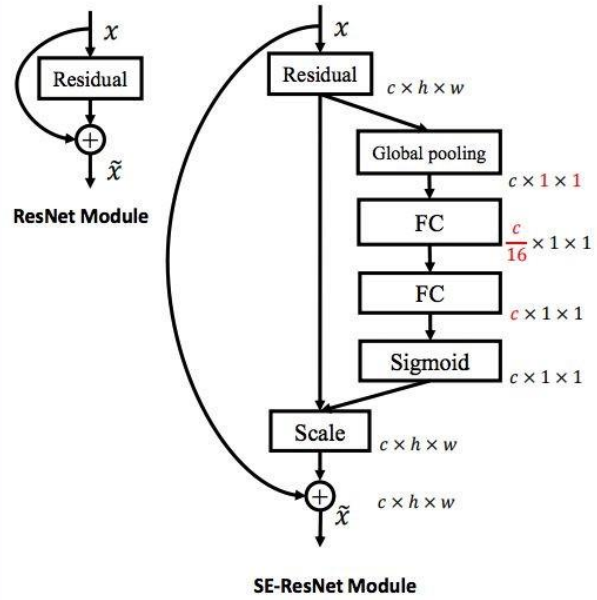
# Improving ResNets...

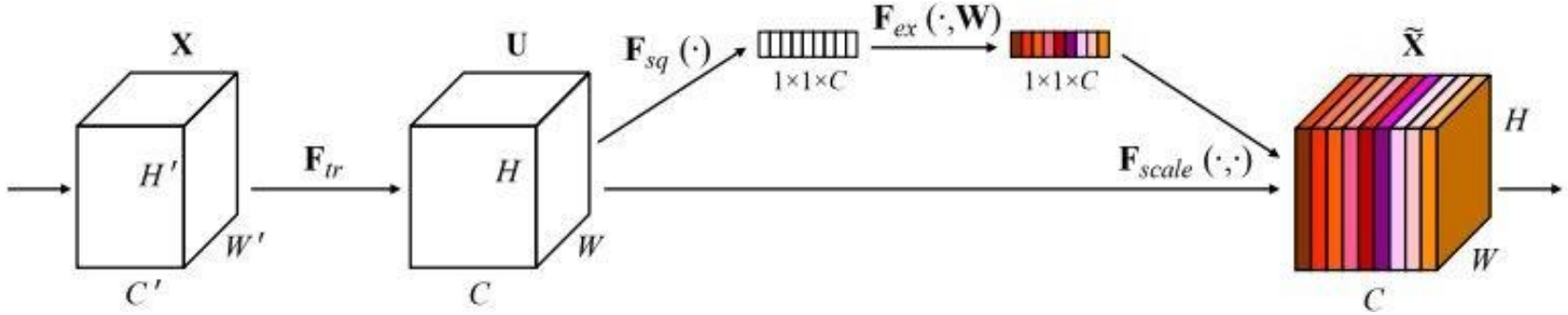
# Squeeze-and-Excitation Networks (SENet)

*[Hu et al. 2017]*

- Add a “feature recalibration” module that learns to **adaptively** reweight feature maps
- Global information (global avg. pooling layer) + 2 FC layers used to determine feature map weights
- ILSVRC’17 classification winner (using ResNeXt-152 as a base architecture)



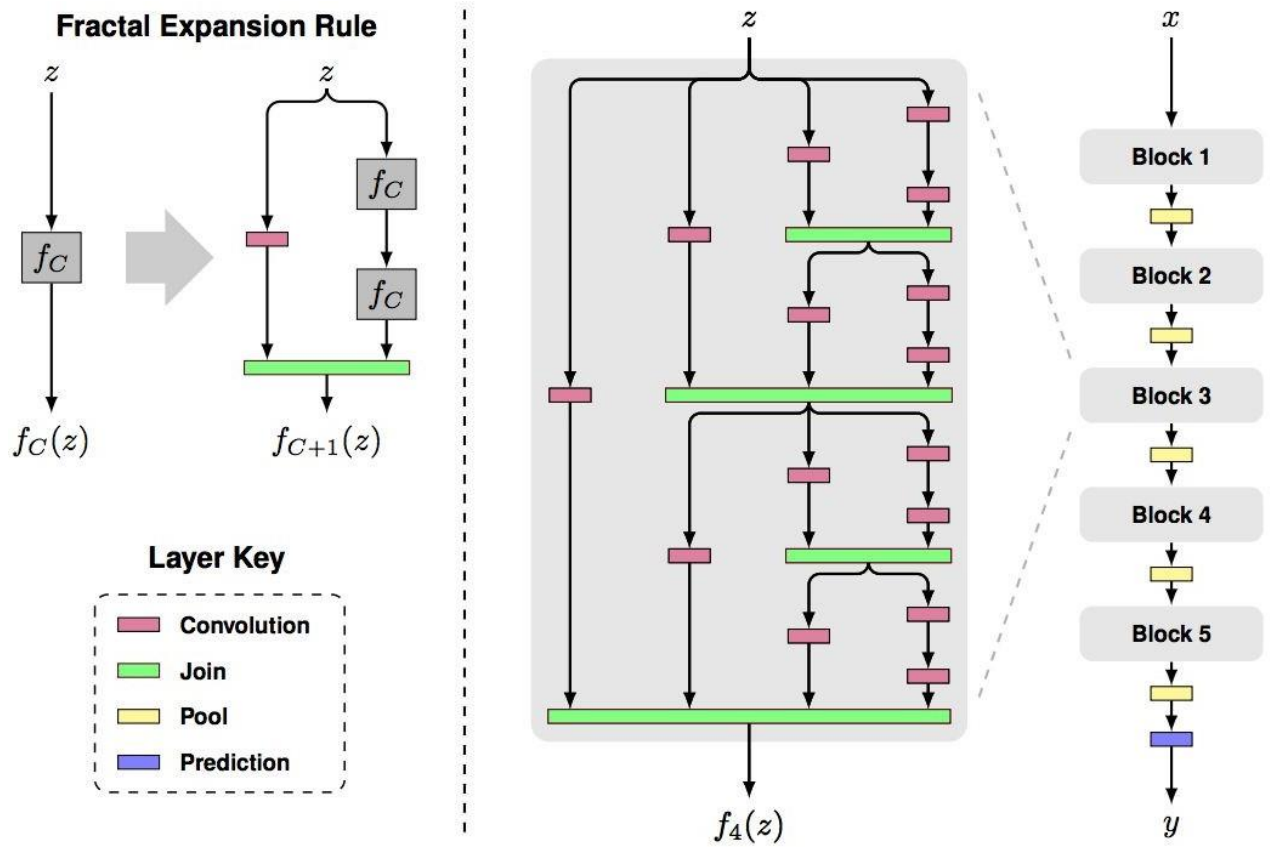
# Beyond ResNets...

UNIMORE
UNIVERSITÀ DEGLI STUDI DI MODENA E REGGIO EMILIA

## FractalNet: Ultra-Deep Neural Networks without Residuals

*[Larsson et al. 2017]*

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



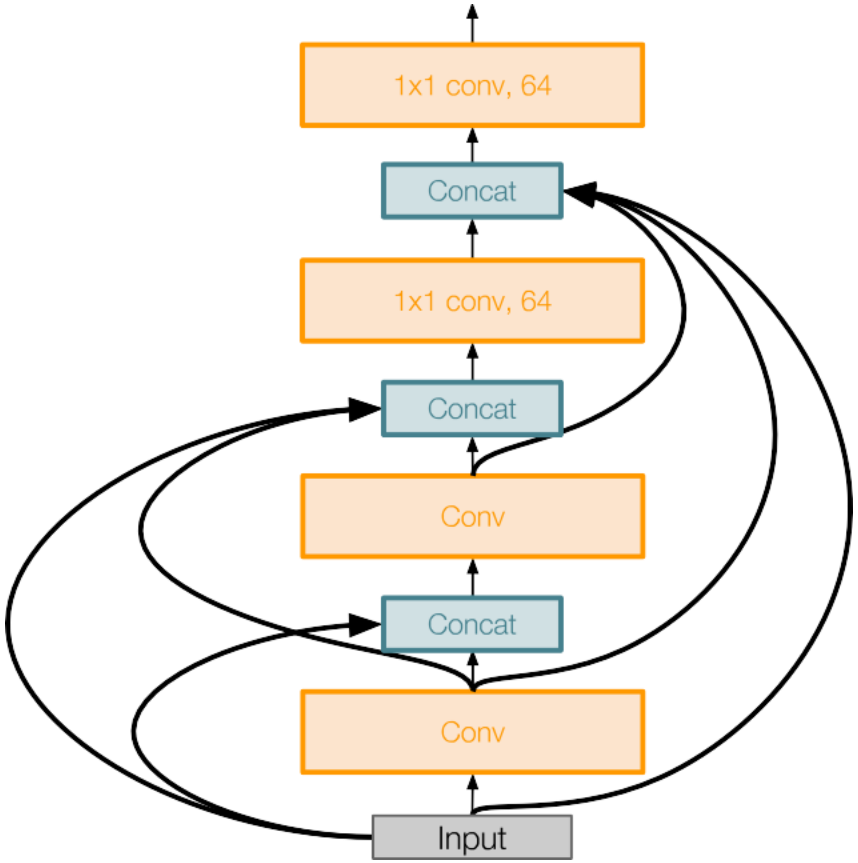

Figures copyright Larsson et al., 2017. Reproduced with permission.

# Beyond ResNets...

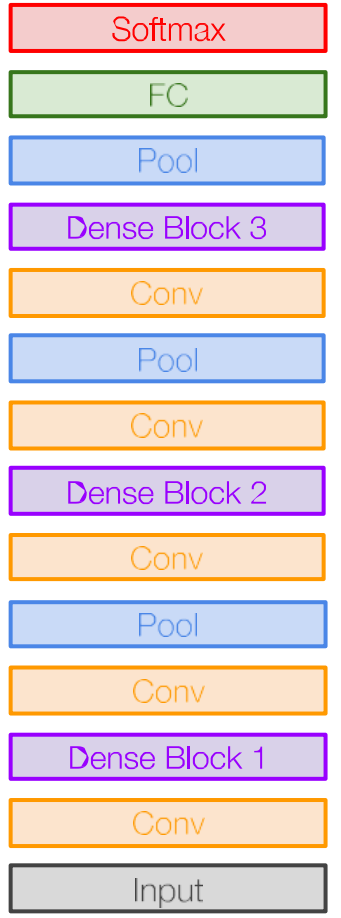
## Densely Connected Convolutional Networks

*[Huang et al. 2017]*

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




Dense Block



# Efficient networks...

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

*[Iandola et al. 2017]*

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

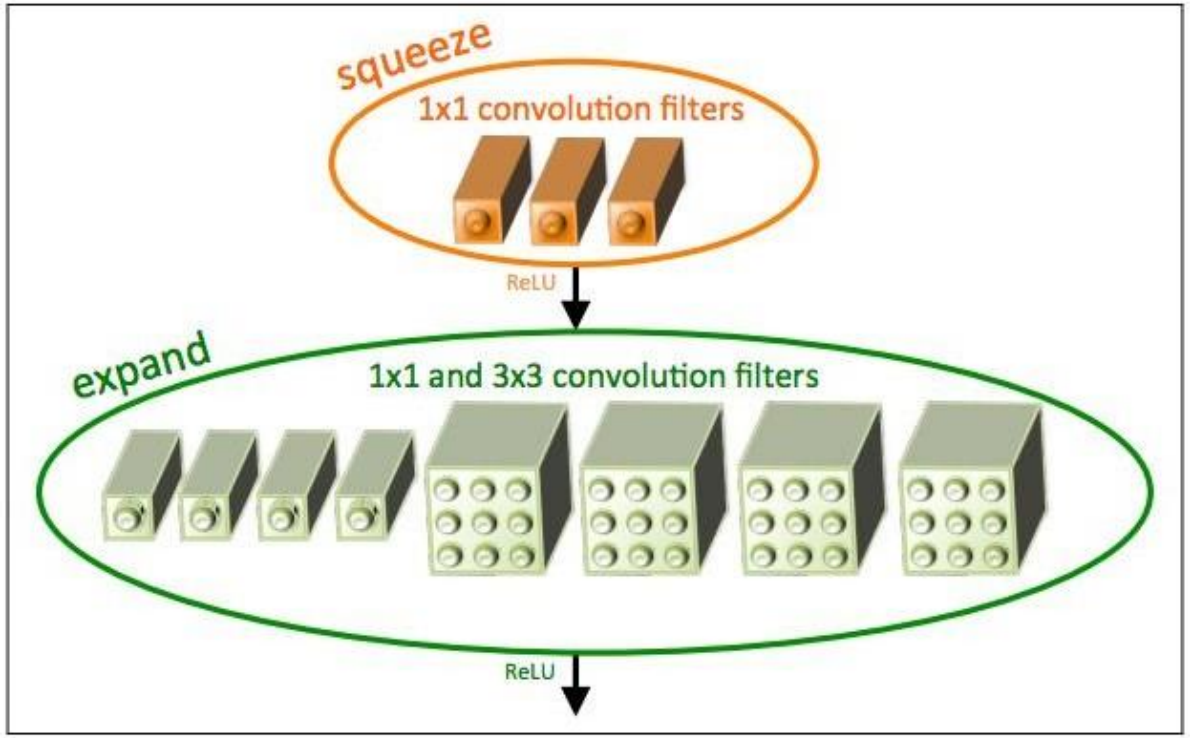



Figure copyright Iandola, Han, Moskewicz, Ashraf, Dally, Keutzer, 2017. Reproduced with permission.


MINERVA
SELF-ATTENTIVE
ARCHITECTURES

# A new operator

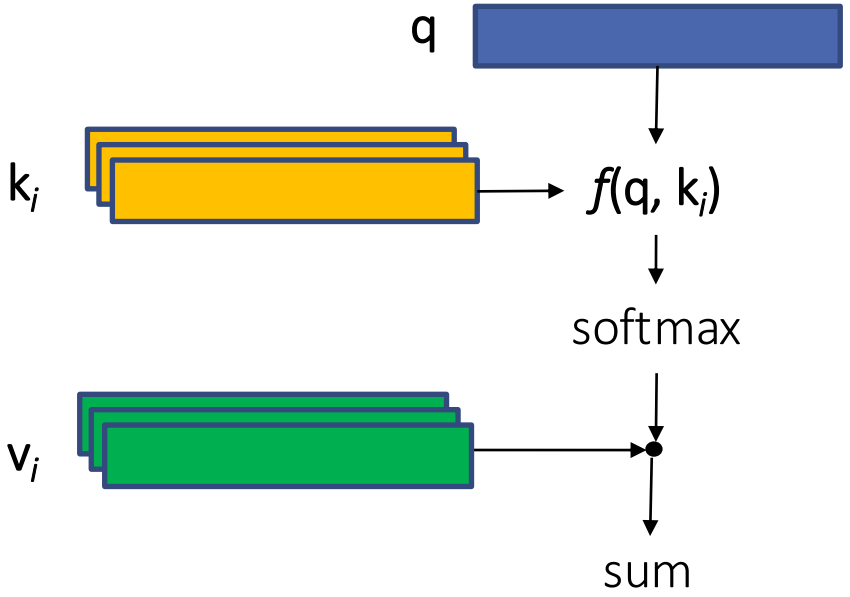
UNIMORE
UNIVERSITÀ DEGLI STUDI DI MODENA E REGGIO EMILIA

**Attention**

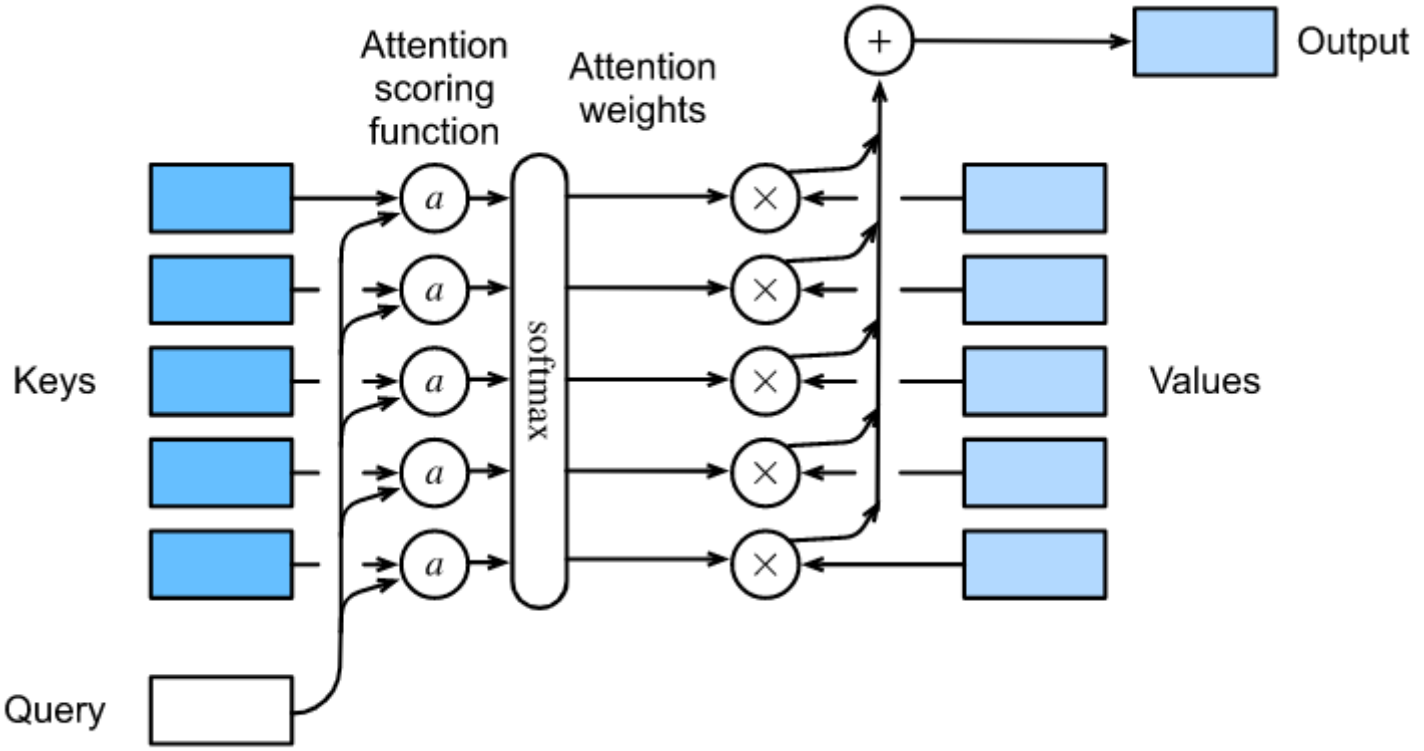
Provides a way to focus on part of an input set.

Given a query and pairs of keys and values,

- Compute similarities between queries and keys
- Normalizes similarities via softmax to obtain attention scores
- Multiplies values by the scores



# Attention




Mathematically, suppose that we have a query $\mathbf{q} \in \mathbb{R}^q$ and $m$ key-value pairs $(\mathbf{k}_1, \mathbf{v}_1), \ldots, (\mathbf{k}_m, \mathbf{v}_m)$, where any $\mathbf{k}_i \in \mathbb{R}^k$ and any $\mathbf{v}_i \in \mathbb{R}^v$. The attention pooling $f$ is instantiated as a weighted sum of the values:

$$f(\mathbf{q}, (\mathbf{k}_1, \mathbf{v}_1), \ldots, (\mathbf{k}_m, \mathbf{v}_m)) = \sum_{i=1}^{m} \alpha(\mathbf{q}, \mathbf{k}_i)\mathbf{v}_i \in \mathbb{R}^v, \tag{10.3.1}$$

where the attention weight (scalar) for the query $\mathbf{q}$ and key $\mathbf{k}_i$ is computed by the softmax operation of an attention scoring function $a$ that maps two vectors to a scalar:

$$\alpha(\mathbf{q}, \mathbf{k}_i) = \mathrm{softmax}(a(\mathbf{q}, \mathbf{k}_i)) = \frac{\exp(a(\mathbf{q}, \mathbf{k}_i))}{\sum_{j=1}^{m} \exp(a(\mathbf{q}, \mathbf{k}_j))} \in \mathbb{R}. \tag{10.3.2}$$

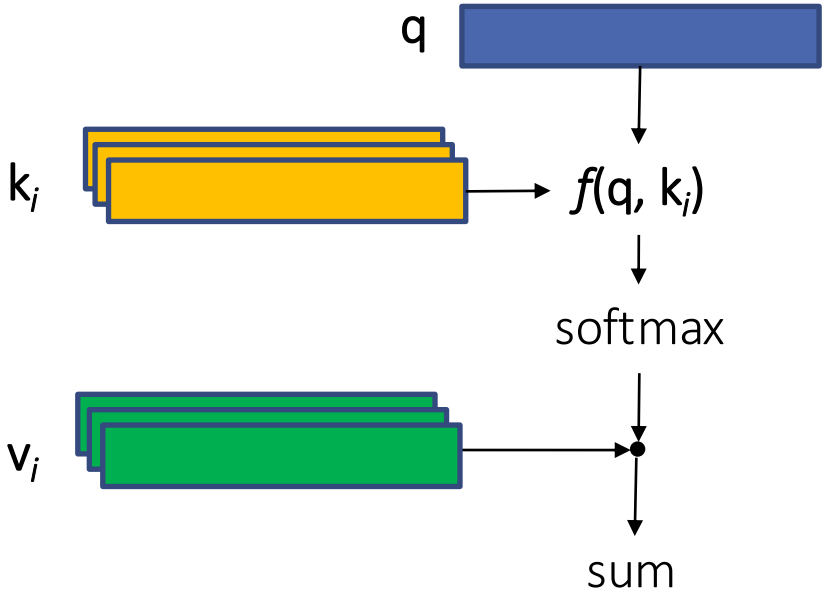
# Attention

## Attention

Provides a way to focus on part of an input set.

Given a query and pairs of keys and values,

- Compute similarities between queries and keys
- Normalizes similarities via softmax to obtain attention scores
- Multiplies values by the scores




*Similarity function*

- Additive attention, e.g. $w^T \tanh(w_q q + w_k k)$
- Dot-product attention

$$a(\mathbf{q}, \mathbf{k}) = \mathbf{q}^\top \mathbf{k} / \sqrt{d}$$

# Dot-product attention

UNIMORE
UNIVERSITÀ DEGLI STUDI DI MODENA E REGGIO EMILIA

```python
class DotProductAttention(nn.Module):
    """Scaled dot product attention."""
    def __init__(self, **kwargs):
        super(DotProductAttention, self).__init__(**kwargs)

    # Shape of `queries`: (`batch_size`, no. of queries, `d`)
    # Shape of `keys`: (`batch_size`, no. of key-value pairs, `d`)
    # Shape of `values`: (`batch_size`, no. of key-value pairs, value
    # dimension)
    # Shape of `valid_lens`: (`batch_size`,) or (`batch_size`, no. of queries)
    def forward(self, queries, keys, values, valid_lens=None):
        d = queries.shape[-1]
        # Use `transpose` to swap the last two dimensions of `keys`
        scores = torch.bmm(queries, keys.transpose(1, 2)) / math.sqrt(d)
        self.attention_weights = torch.softmax(scores)
        return torch.bmm(self.attention_weights, values)
```

# Convolution vs Self-attention

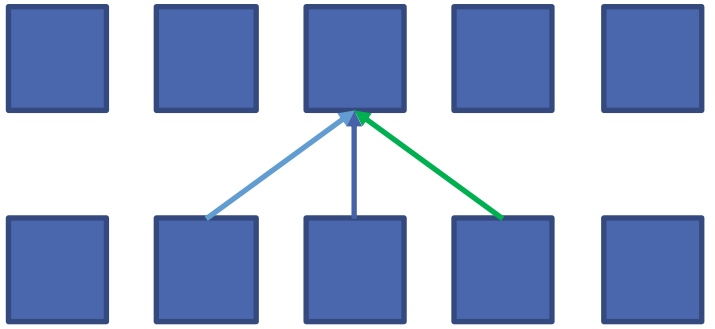
UNIMORE
UNIVERSITÀ DEGLI STUDI DI MODENA E REGGIO EMILIA

## Self Attention

"Refine" each element of the sequence by treating it as **query**, and the whole sequence as keys and values.

Actually: queries, keys and values are three different linear projections of each element of the input sequence.

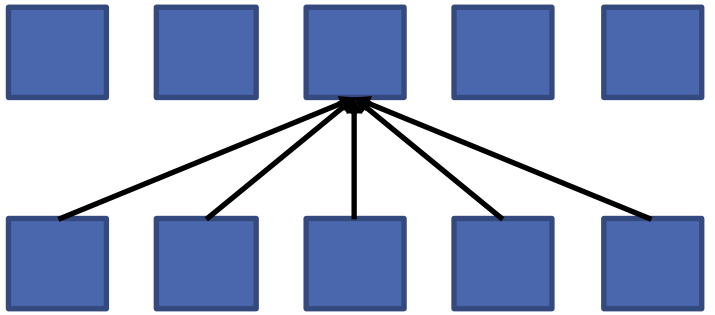
Receptive field is infinite!
Constant path length between two different positions
Trivial to parallelize during training!



Output sequence

Input sequence

**1D Convolution**



Output sequence

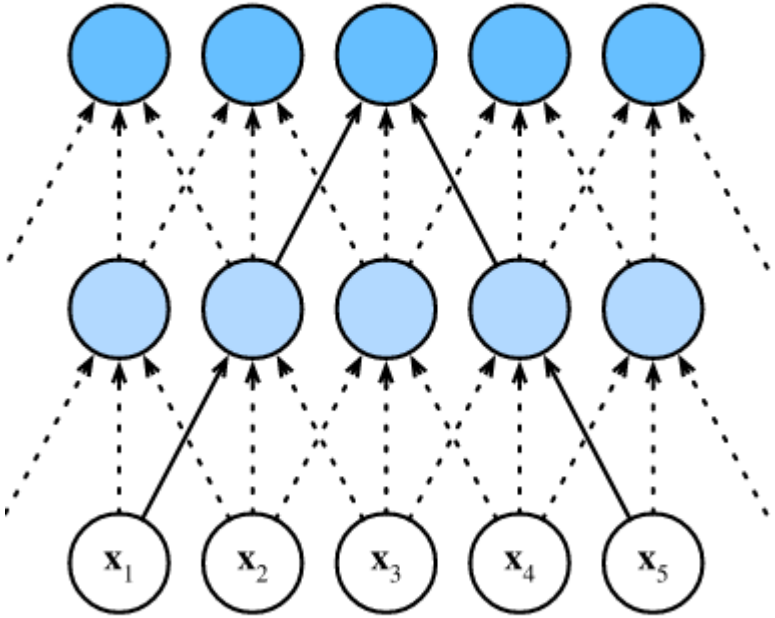
Input sequence (i.e. queries, keys, values)

**Self-Attention**

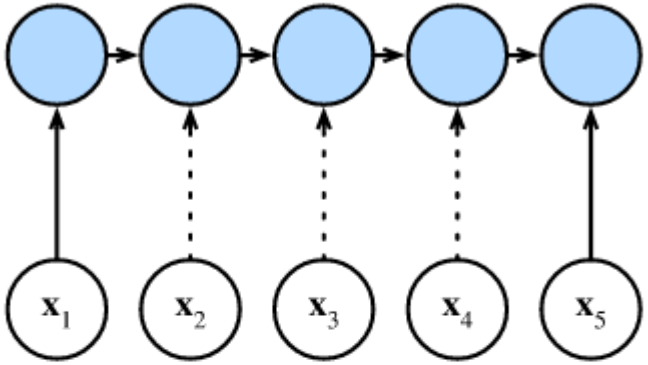
# CNNs vs RNNs vs Self-attention



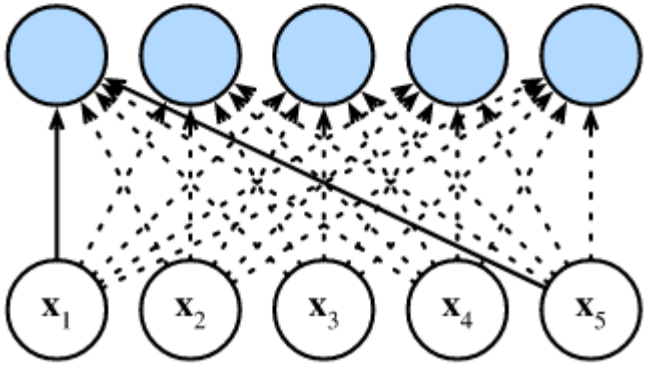
CNN




RNN




Self-attention



# Self-attention



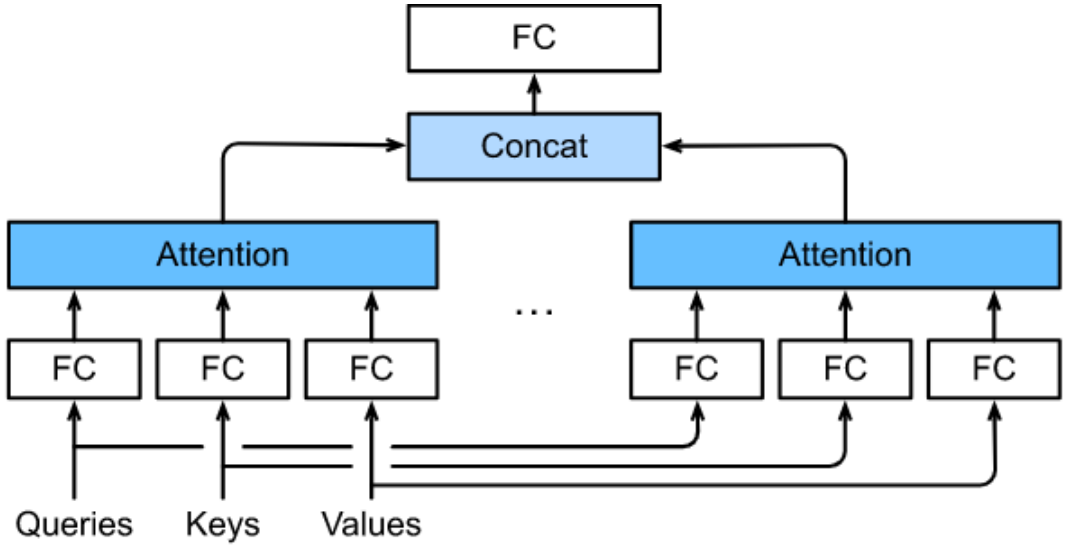
- Given the input sequence, extract queries, keys and values from each element by applying a linear transformation (i.e. a fully connected layer). The dimensionality of queries and keys need to match. From a sequence with length $T$, we obtain:
  - $Q$: matrix of queries, $(T, d_k)$
  - $K$: matrix of keys, $(T, d_k)$
  - $V$: matrix of values, $(T, d_v)$

- Apply scaled dot-product attention as:

$$\text{Attention}(Q, K, V) = \text{softmax}(\frac{QK^T}{\sqrt{d_k}})V$$

# Multi-head attention

- In practice, given the same set of queries, keys, and values we may want our model to combine knowledge from different behaviors of the same attention mechanism, such as capturing dependencies of various ranges in a sequence.
- To this end, instead of performing a single attention pooling, queries, keys, and values can be transformed with **h independently learned linear projections**. Then these h projected queries, keys, and values are fed into attention pooling **in parallel**. In the end, h attention pooling outputs are concatenated and transformed with another learned linear projection to produce the final output.




Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., ... & Polosukhin, I. Attention is all you need. In NIPS 2017.

# Multi-head Self-attention

- Linearly project the input sequence $h$ times with different weights, instead of doing this only once.
- To From a sequence with length $T$, we obtain:
  - $Q$: matrix of queries, $(h, T, d_k)$
  - $K$: matrix of keys, $(h, T , d_k)$
  - $V$: matrix of values, $(h, T, d_v)$

- Apply scaled dot-product attention over each “head” (i.e. over each element of axis 1)
- Concatenate the result and project back to a lower dimensionality

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \ldots, \text{head}_h)W^O$$
where $\text{head}_i = \text{Attention}(Q[i], K[i], V[i])$

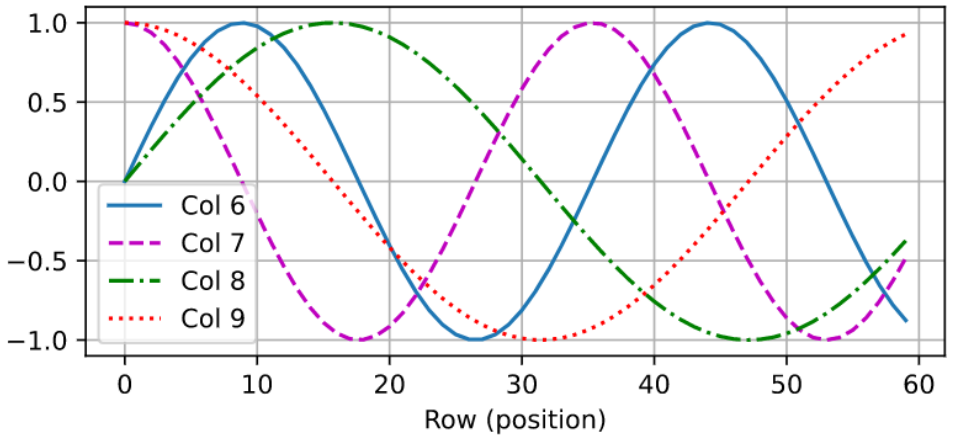
- Can be done in parallel, with batched matrix multiplication.

# Positional encoding

- Self-attention is permutation invariant
  - Given a query, if we change the order of keys and values, result does not change.
  - Ok for encoding sets. Not for sequences or images....☹
- To use **order information**, we can inject absolute or relative positional information by adding positional encoding to the input representations.
- Positional encodings can be learned (simple nn.Parameter) or fixed. In the original Transformer, they were defined as sinusoids. With this, attention can capture both absolute and relative positional information (i.e. distances between items!)

Suppose that the input representation $\mathbf{X} \in \mathbb{R}^{n \times d}$ contains the $d$-dimensional embeddings for $n$ tokens of a sequence. The positional encoding outputs $\mathbf{X} + \mathbf{P}$ using a positional embedding matrix $\mathbf{P} \in \mathbb{R}^{n \times d}$ of the same shape, whose element on the $i^{\text{th}}$ row and the $(2j)^{\text{th}}$ or the $(2j+1)^{\text{th}}$ column is

$$p_{i,2j} = \sin\left(\frac{i}{10000^{2j/d}}\right),$$
$$p_{i,2j+1} = \cos\left(\frac{i}{10000^{2j/d}}\right).$$



# A Self-attentive language model for translation

UNIMORE
UNIVERSITÀ DEGLI STUDI DI MODENA E REGGIO EMILIA

## Encoder

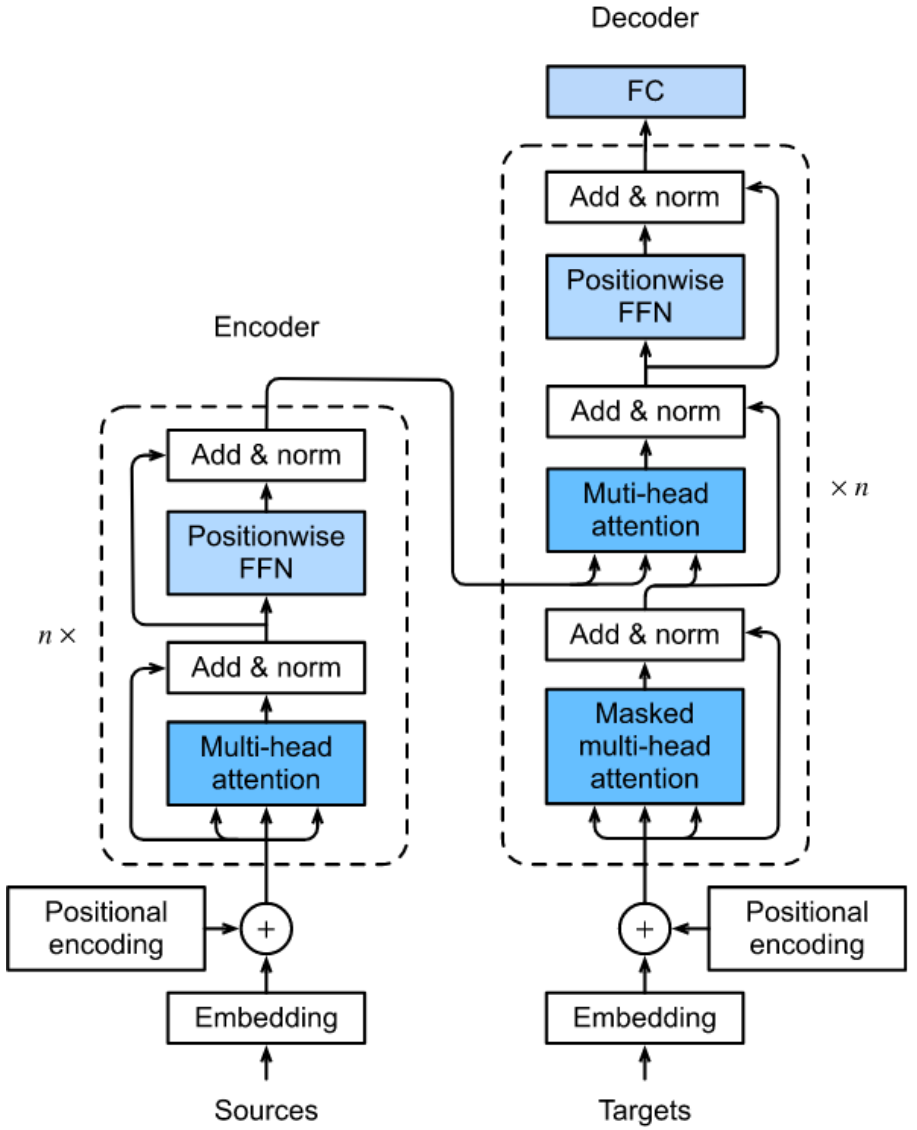
Uses self-attention on its input

Multiple attention layers stacked together (with add+norm) and feed-forward layers (linear layers applied timewise).

## Decoder

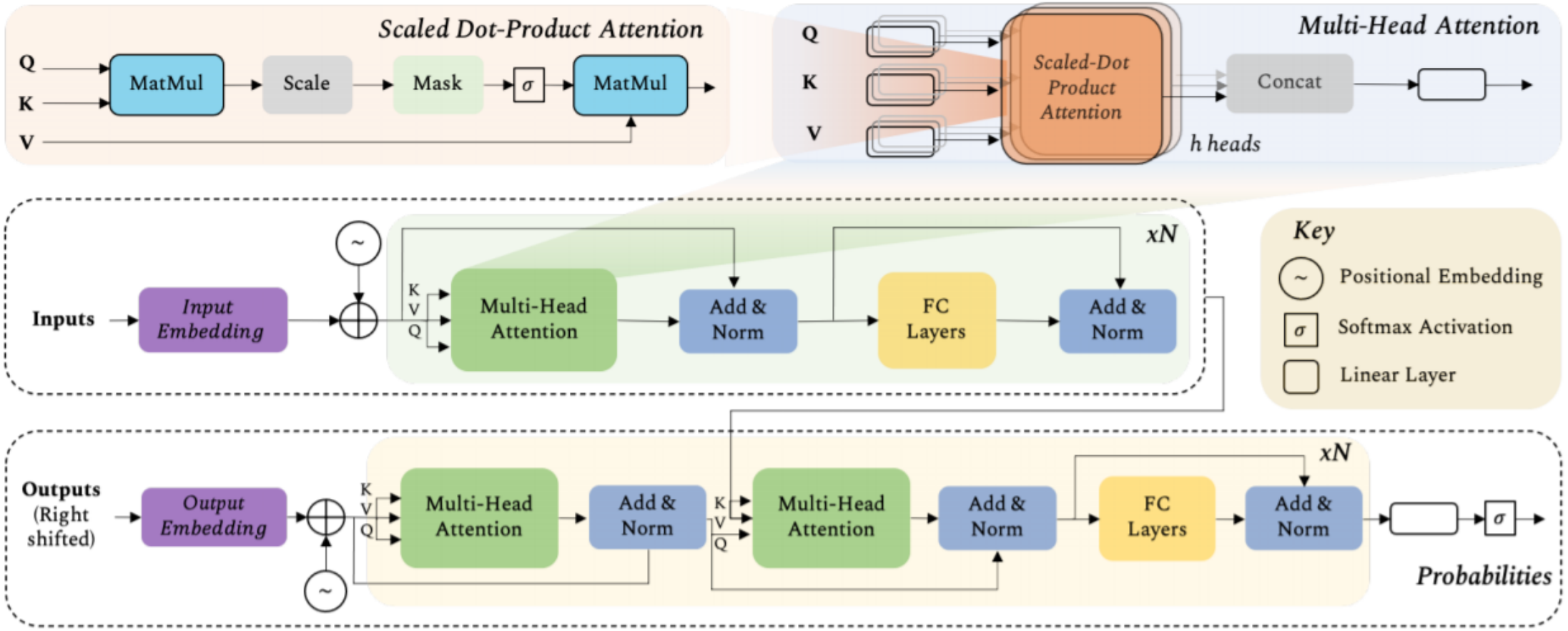
Self-attention on words

Cross-attention on encoder outputs: use decoder sequences as queries, encoder outputs as key/values.

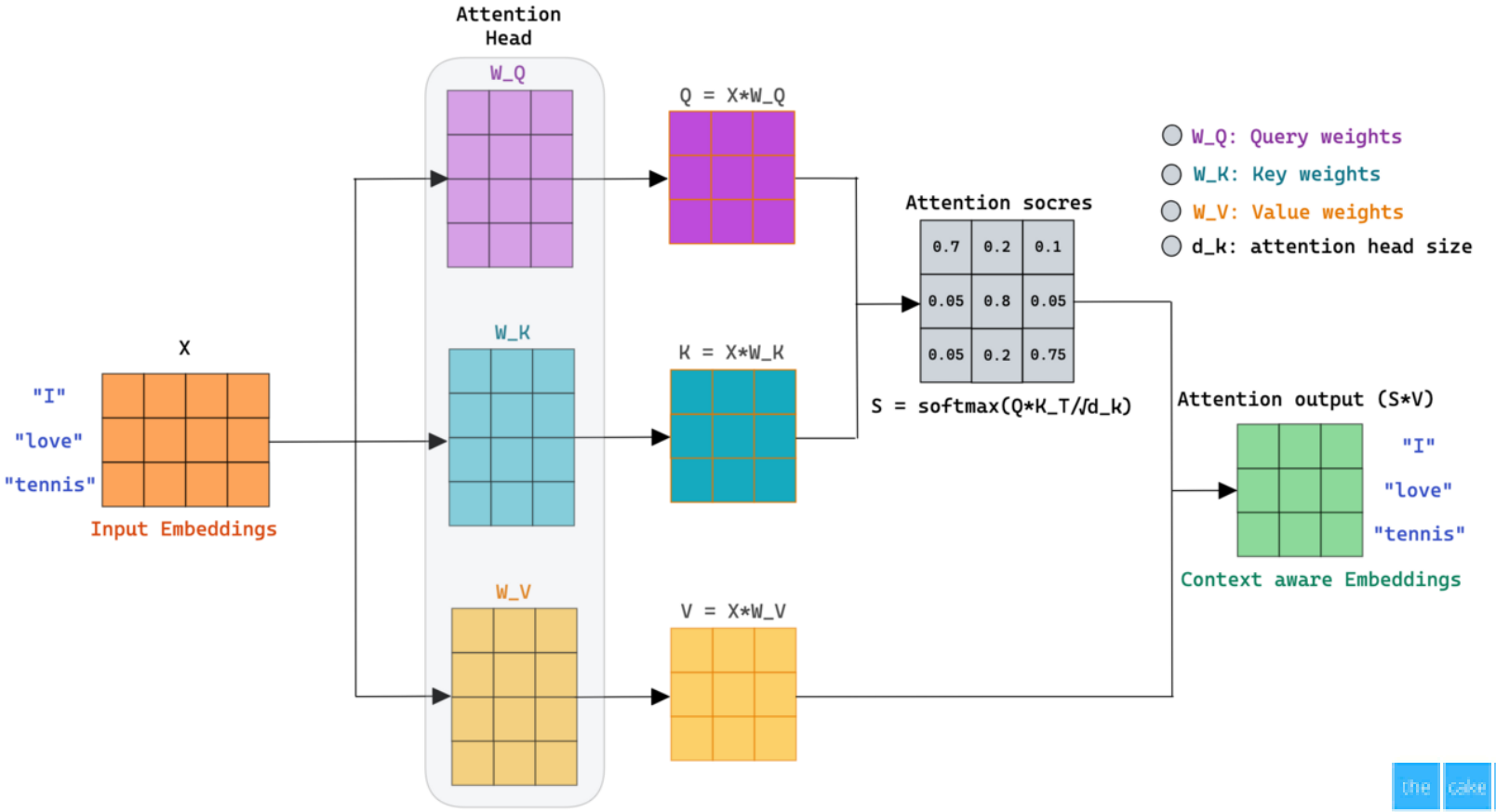



Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., ... & Polosukhin, I. Attention is all you need. In NIPS 2017.

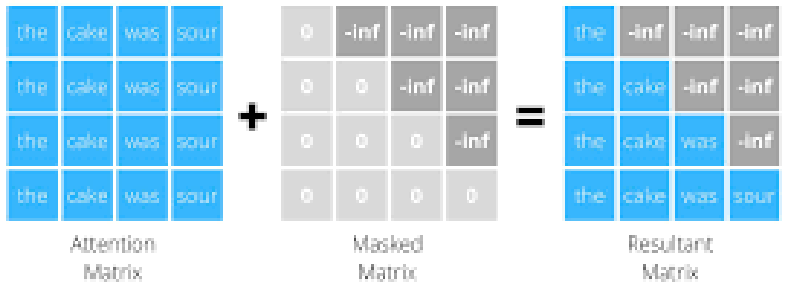
# Architecture of the Transformer



# Masked Self-Attention




## Masked Attention

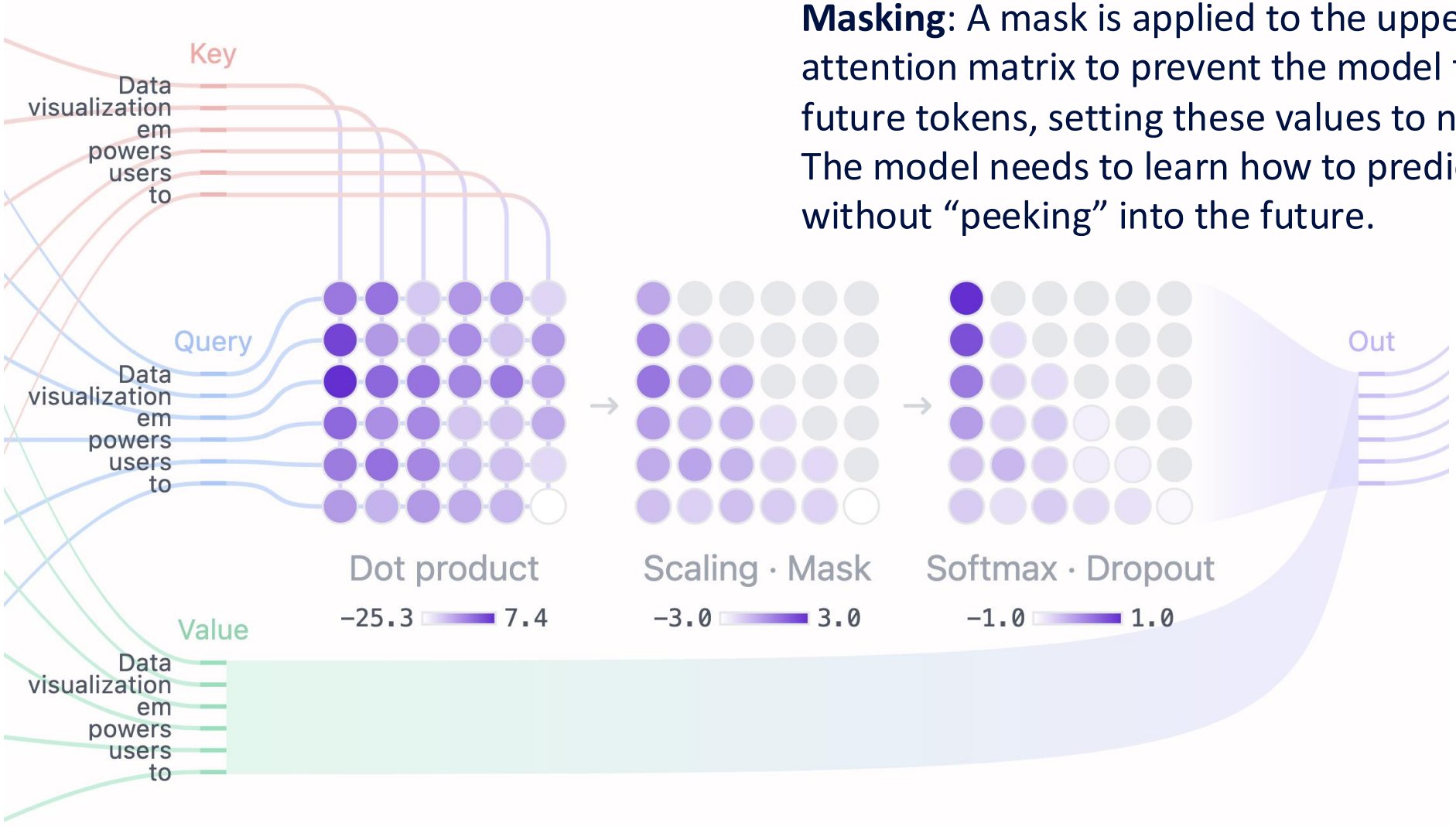



*instead of words there will be attention weight

# Masked Self-Attention



**Masking**: A mask is applied to the upper triangle of the attention matrix to prevent the model from accessing future tokens, setting these values to negative infinity. The model needs to learn how to predict the next token without “peeking” into the future.




https://poloclub.github.io/transformer-explainer/

# Transformer Architecture

Ways attention is used in the transformer:

- **Self-attention in the encoder**
  - Allows the model to attend to all positions in the previous encoder layer
  - Embeds context about how elements in the sequence relate to one another
- **Masked self-attention in the decoder**
  - Allows the model to attend to all positions in the previous decoder layer up to and including the current position (during auto-regressive process)
  - Prevents forward looking bias by stopping leftward information flow during training
  - Also embeds context about how elements in the sequence relate to one another
- **Encoder-decoder cross-attention**
  - Allows decoder layers to attend all parts of the latent representation produced by the encoder
  - Pulls context from the encoder sequence over to the decoder"

A. Vaswani *et al.*, "Attention is All you Need," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2017.

# Transformer

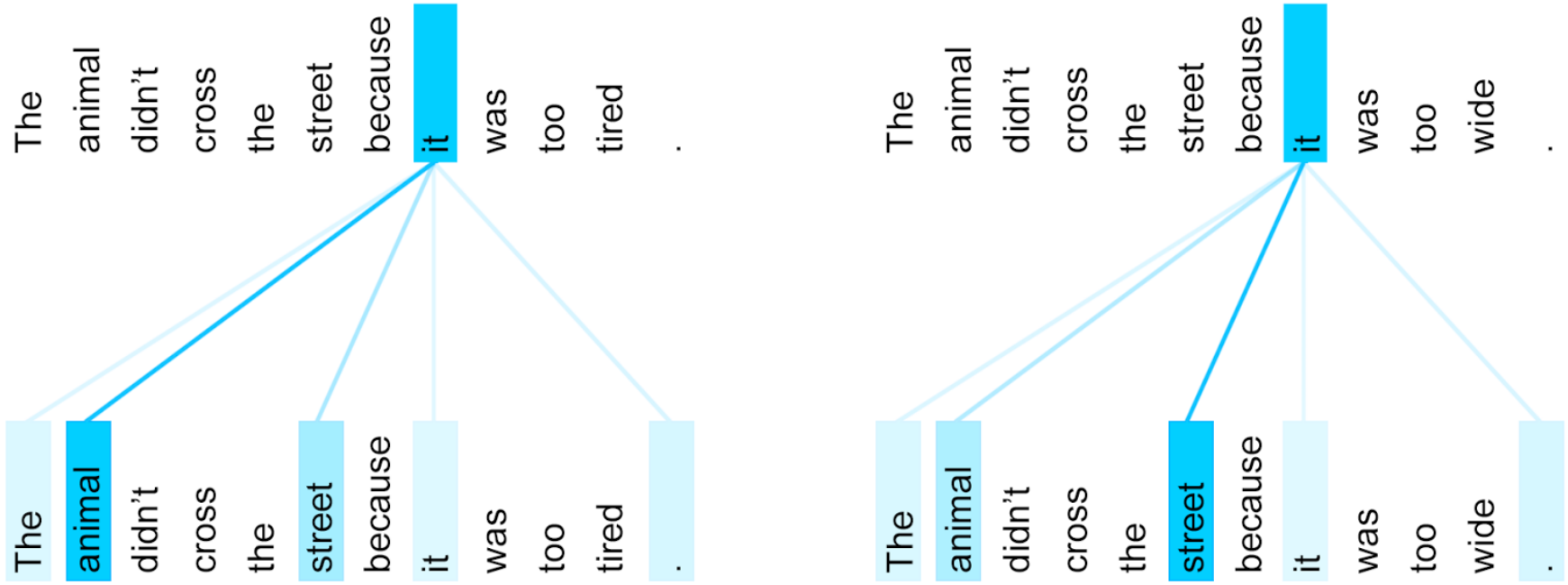
Encoder, using self-attention, it aggregates information from **all of the other words**, **generating a new representation per word informed by the entire context**, represented by the filled balls.
This step is then repeated multiple times in parallel for all words, successively generating new representations.

The decoder operates similarly, but generates **one word at a time**, from left to right. **It attends not only to the other previously generated words, but also to the final representations generated by the encoder.**

# Example

*The animal didn't cross the street because it was too tired.*
*L'animal n'a pas traversé la rue parce qu'il était trop fatigué.*

*The animal didn't cross the street because it was too wide.*
*L'animal n'a pas traversé la rue parce qu'elle était trop large.*



# Not only language: Vision Transformer (ViT)

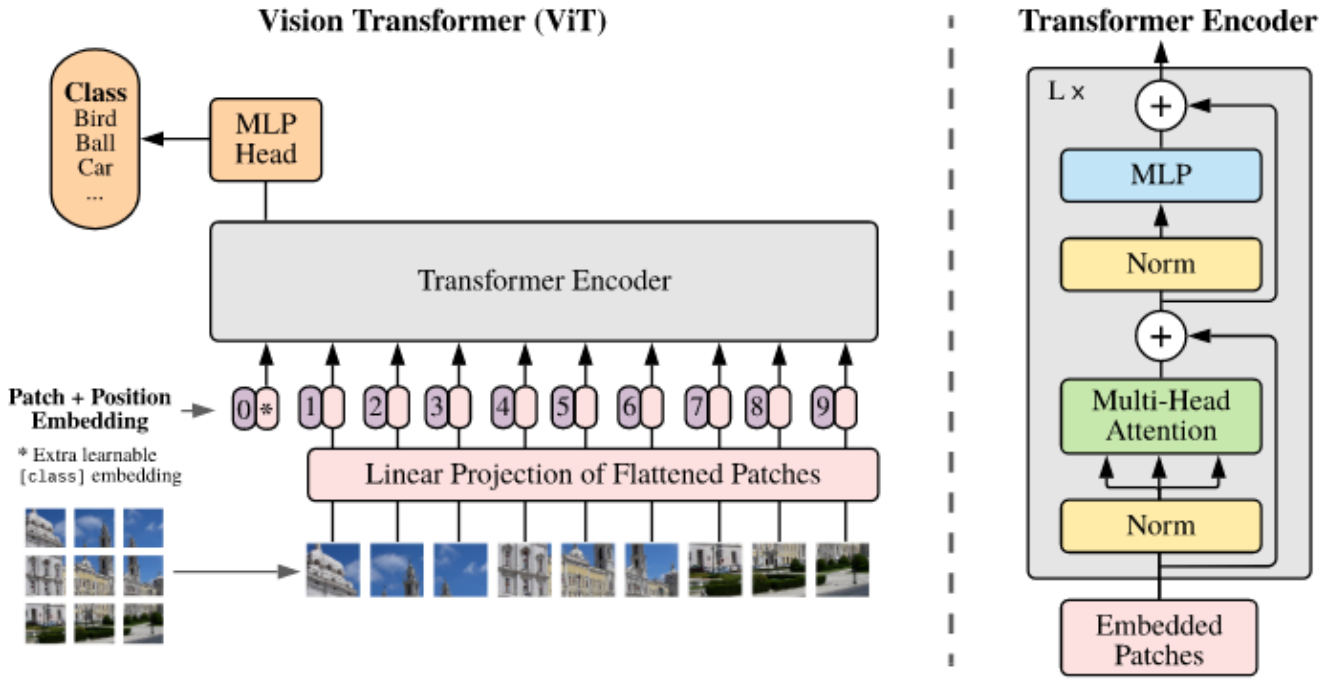
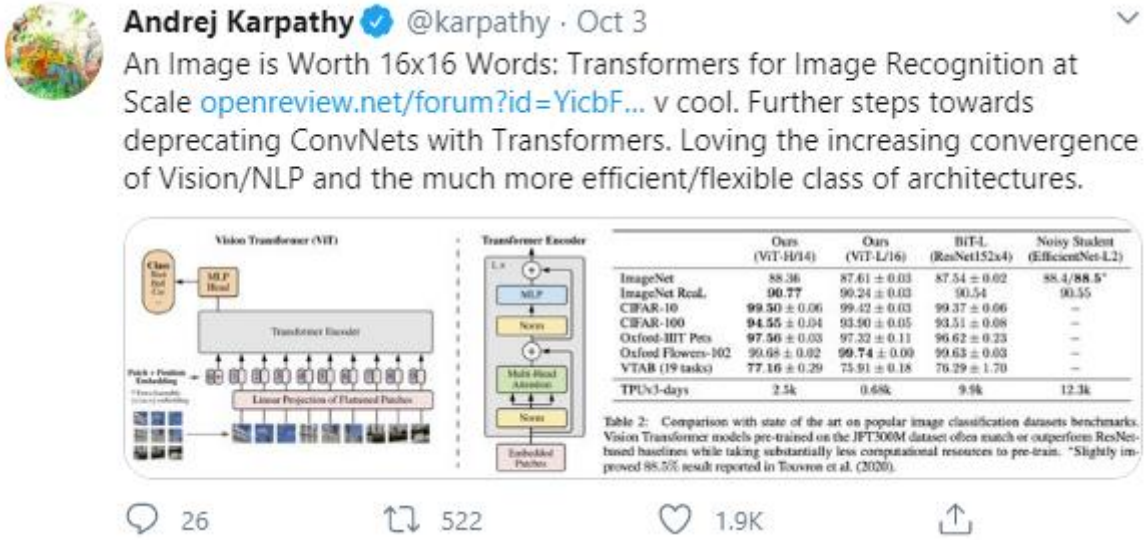



Figure 1: Model overview. We split an image into fixed-size patches, linearly embed each of them, add position embeddings, and feed the resulting sequence of vectors to a standard Transformer encoder. In order to perform classification, we use the standard approach of adding an extra learnable "classification token" to the sequence. The illustration of the Transformer encoder was inspired by Vaswani et al. (2017).

Dosovitskiy, A., Beyer, L., Kolesnikov, A., Weissenborn, D., Zhai, X., Unterthiner, T., ... & Houlsby, N. (2020). An image is worth 16x16 words: Transformers for image recognition at scale. arXiv preprint arXiv:2010.11929.

# Vision Transformer (ViT)






| Model | Layers | Hidden size $D$ | MLP size | Heads | Params |
|---|---|---|---|---|---|
| ViT-Base | 12 | 768 | 3072 | 12 | 86M |
| ViT-Large | 24 | 1024 | 4096 | 16 | 307M |
| ViT-Huge | 32 | 1280 | 5120 | 16 | 632M |

# Not only language: Vision Transformer (ViT)

UNIMORE
UNIVERSITÀ DEGLI STUDI DI MODENA E REGGIO EMILIA

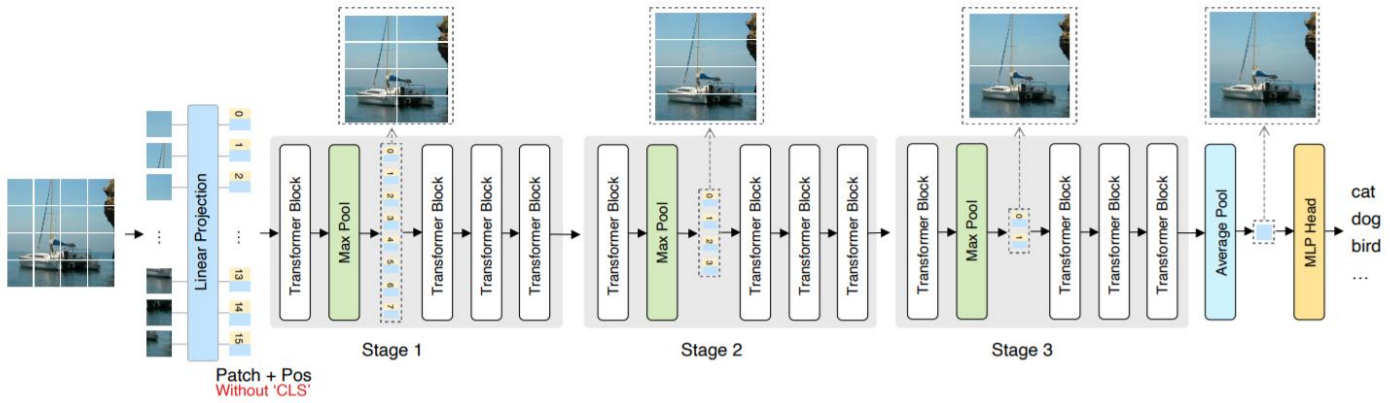
- Current ViT models maintain a **full-length patch** sequence during inference, which is redundant and lacks hierarchical representation

- Making predictions by taking the class token [CLS] as input in classification tasks relies solely on the single class token with limited capacity while discarding the remaining sequence that is capable of storing more discriminative information.

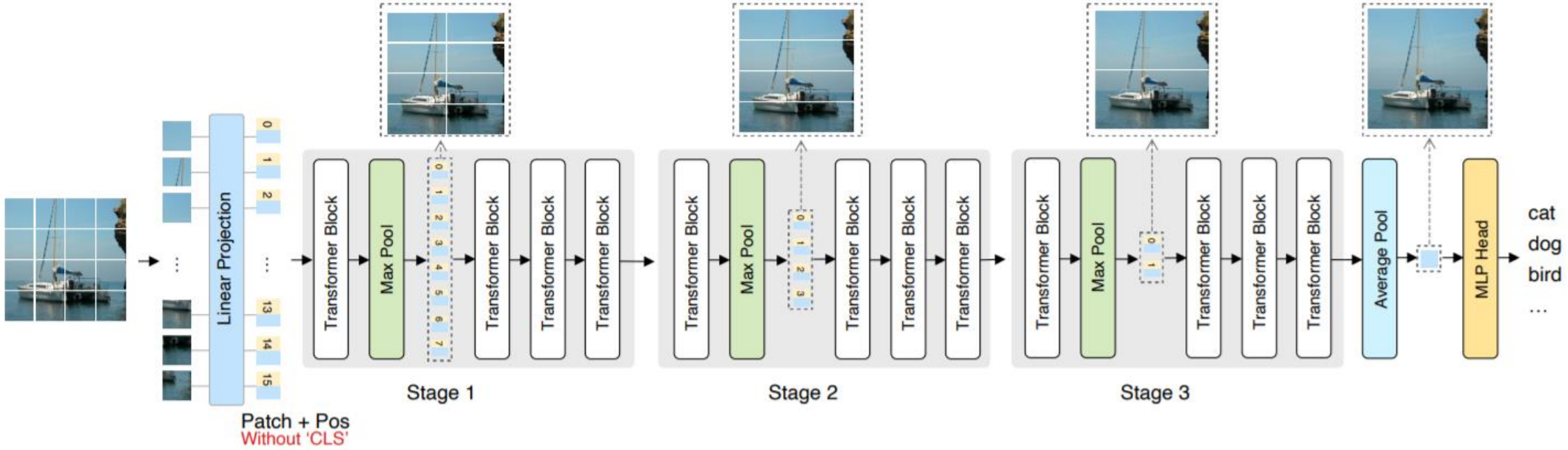
# Not only language: Vision Transformer (ViT)

- Current ViT models maintain a **full-length patch** sequence during inference, which is redundant and lacks hierarchical representation
  - They propose a Hierarchical Visual Transformer (HVT) which progressively pools visual tokens to shrink the sequence length and hence reduces the computational cost, analogous to the feature maps downsampling in Convolutional Neural Networks (CNNs).

- Making predictions by taking the class token [CLS] as input in classification tasks relies solely on the single class token with limited capacity while discarding the remaining sequence that is capable of storing more discriminative information.
  - To this end, they propose to remove the class token in the first place and predict with the remaining output sequence on the last stage.



# Not only language



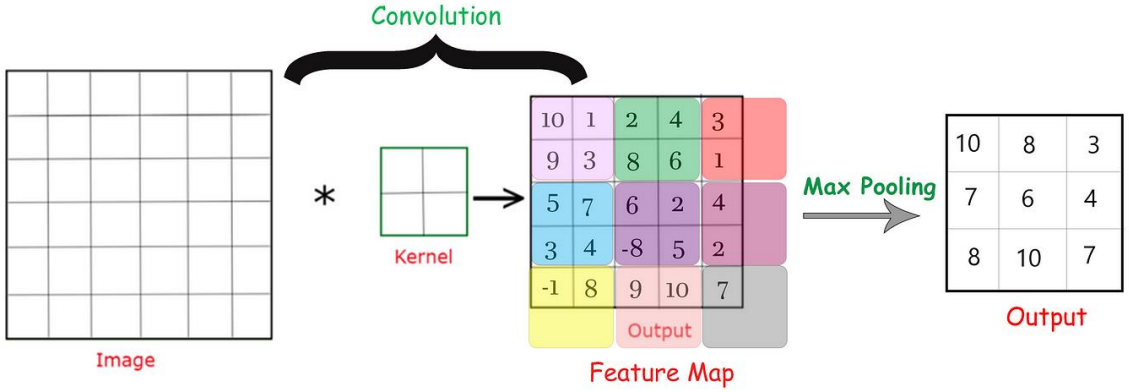

Pan, Z., Zhuang, B., Liu, J., He, H., & Cai, J. (2021). Scalable visual transformers with hierarchical pooling. arXiv preprint arXiv:2103.10619.

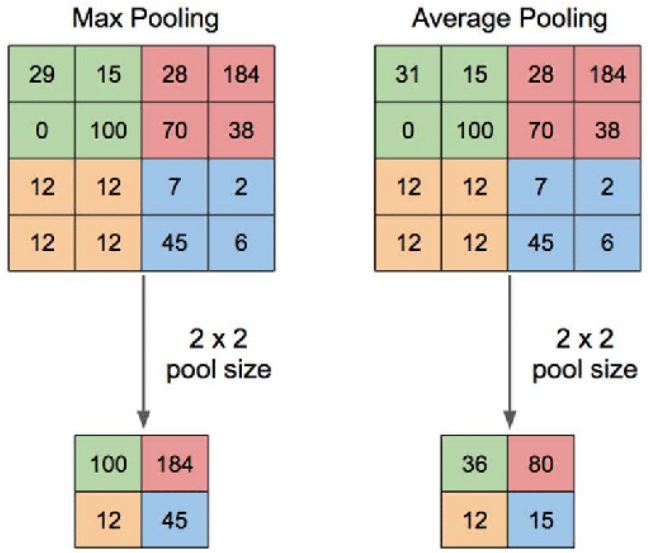
# Not only language

- **Max Pooling** is an operation that extract maximum value from the feature map according to filter and stride

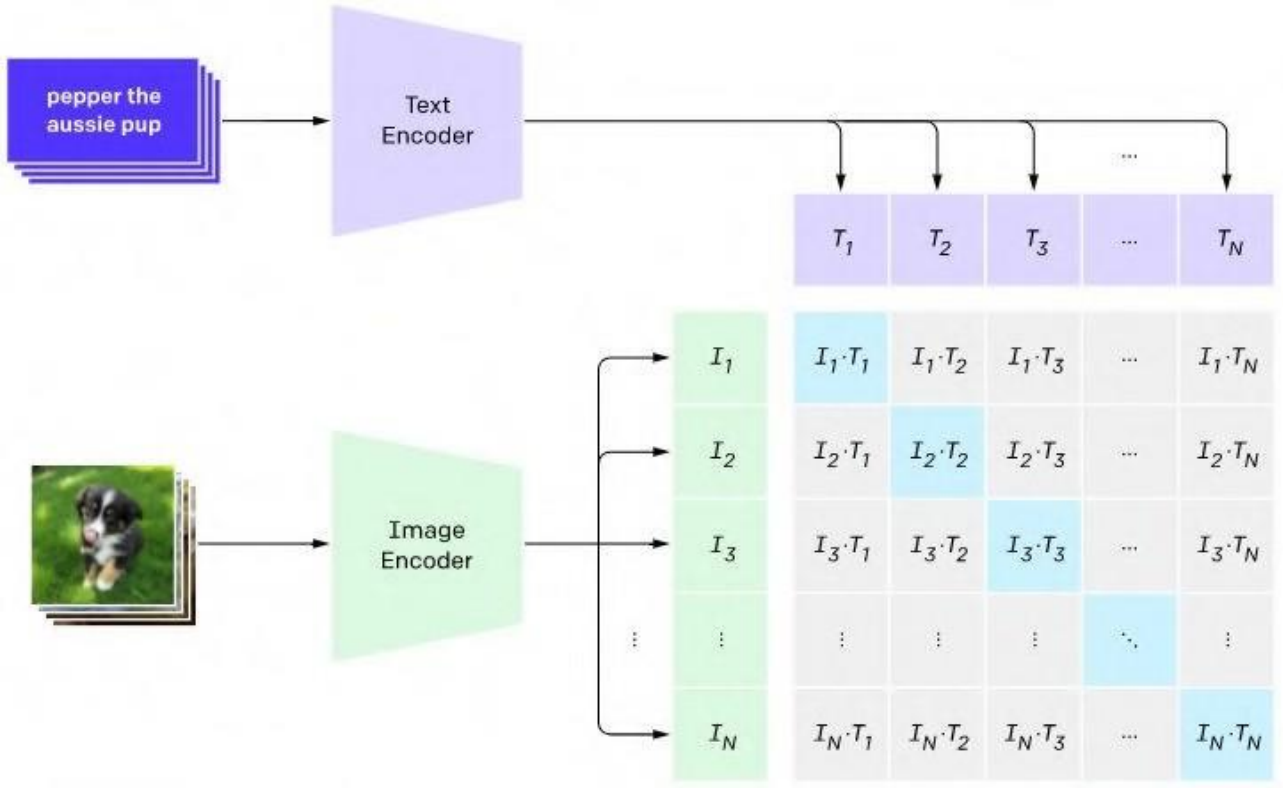
WHY??

- Dimension Reduction
- Invariant to Small Changes
- Feature Selection
  - By selecting the maximum values in each window, Max Pooling retains the most essential features while discarding less relevant information
- Increased Receptive Field
  - Max Pooling helps the network summarise information in larger regions and enables the network to capture more significant spatial patterns.



# CLIP: Zero-shot Capabilities




- Text encoder:
  - 12-layer Transformer with causal mask
- Image encoder:
  - ResNet families: RN50, RN101, RN50x4, RN50x16, RN50x64
  - ViT families: ViT-B/32, ViT-B/16, ViT-L/14

Radford et al. "Learning Transferable Visual Models From Natural Language Supervision", ICML 2021

# Vision-language models: Contrastive learning

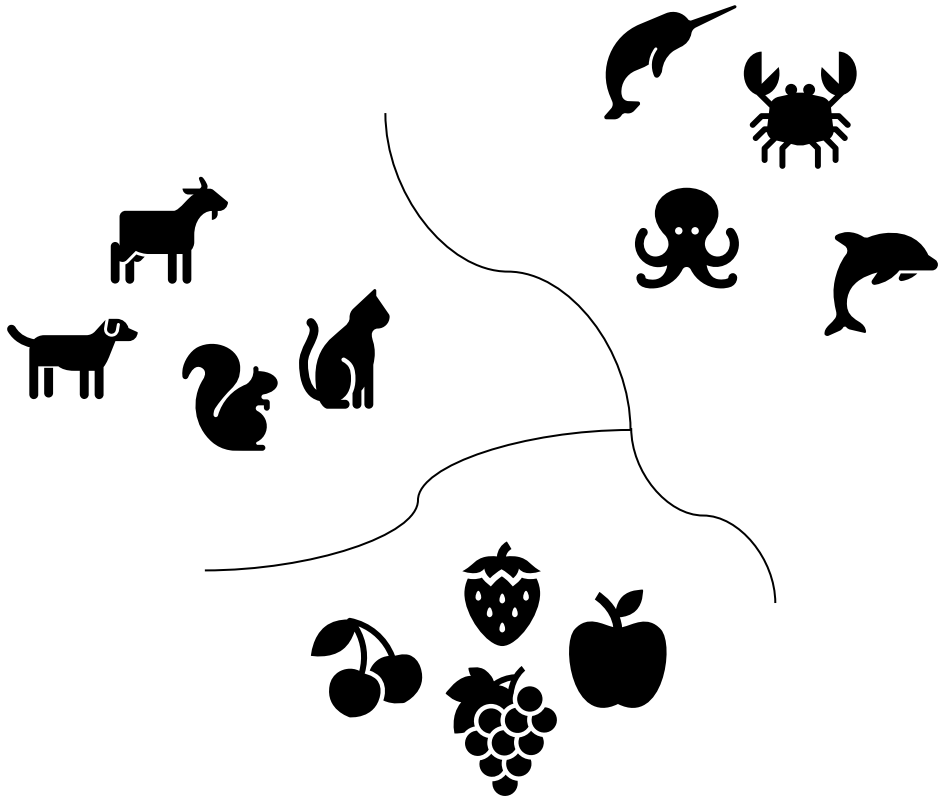


- **Contrastive training** to bridge the image and text embedding spaces
- Making embedding of (image, text) pairs similar and that of non-pairs dissimilar
- This embedding space is super helpful for performing **searches across modalities**
  - Can return the best caption given an image
  - Has impressive capabilities for zero-shot adaptation to unseen tasks, without the need for fine-tuning

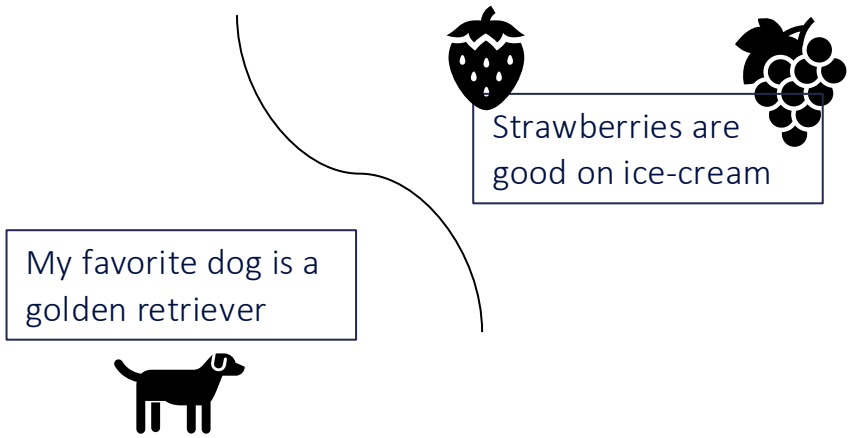
Radford et al. "Learning Transferable Visual Models From Natural Language Supervision", ICML 2021

# Vision-language models: Contrastive learning

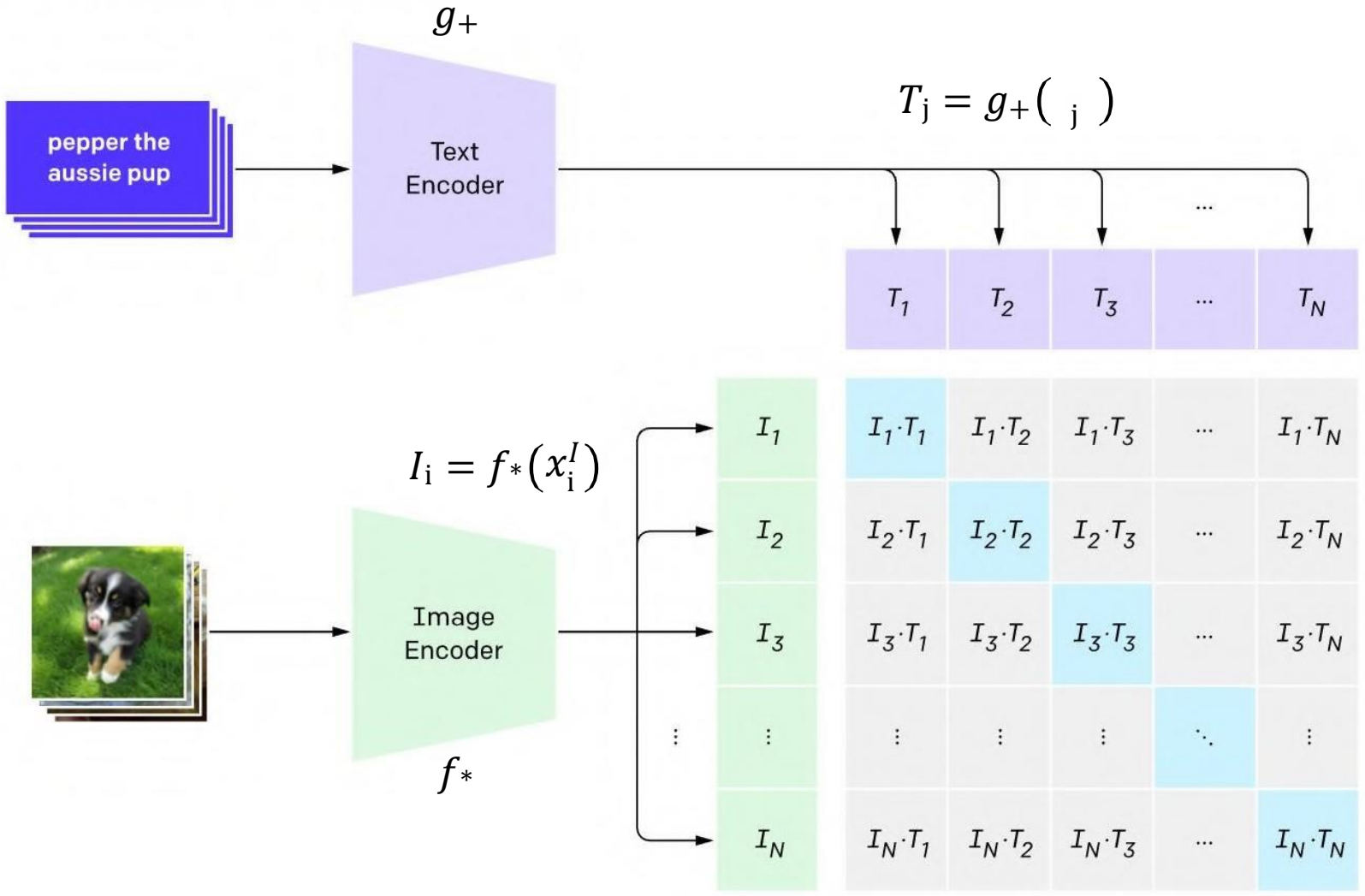
- The main idea is to learn concepts **without labels** → a self-supervised pretraining objective
- The hope was that the learned representations generalize to new instances



Can we generalize these representations beyond just images? To language perhaps?



## 1. Contrastive pre-training




$$s^T_{i,j} = s^I_{i,j} = I_i^T T_j$$

$$f_i^I = -\log \frac{e^{s^I_{i,i}}}{\sum_{j'(}^{N} e^{s^I_{i,\$}}}$$

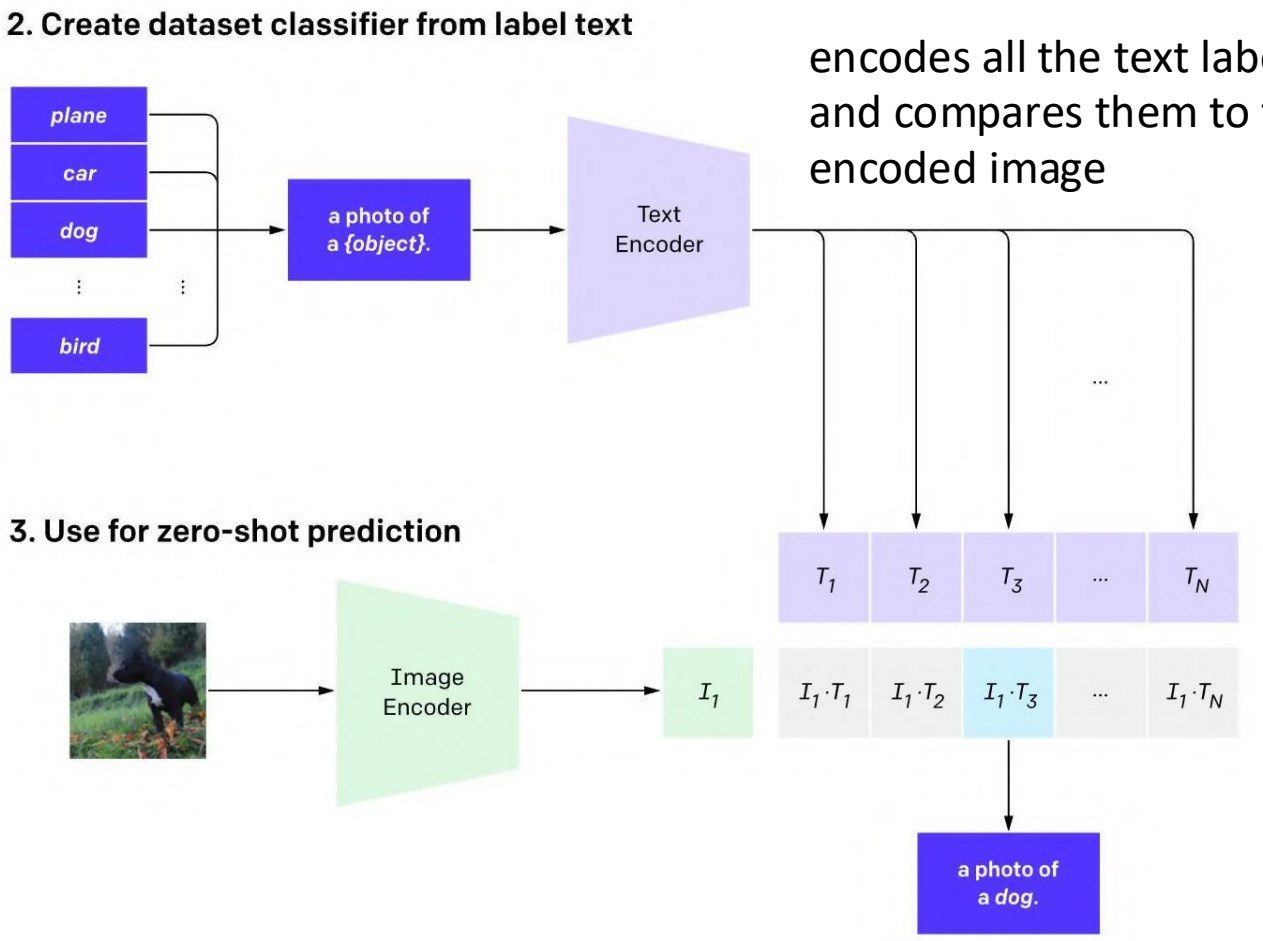
$$f_j^T = -\log \frac{e^{s^T_{i,i}}}{\sum_{i'(}^{N} e^{s^T_{i,\$}}}$$

$$f = \frac{1}{2N} \Big/ \sum_{i'(}^{N} (f_i^I + f_j^T)$$

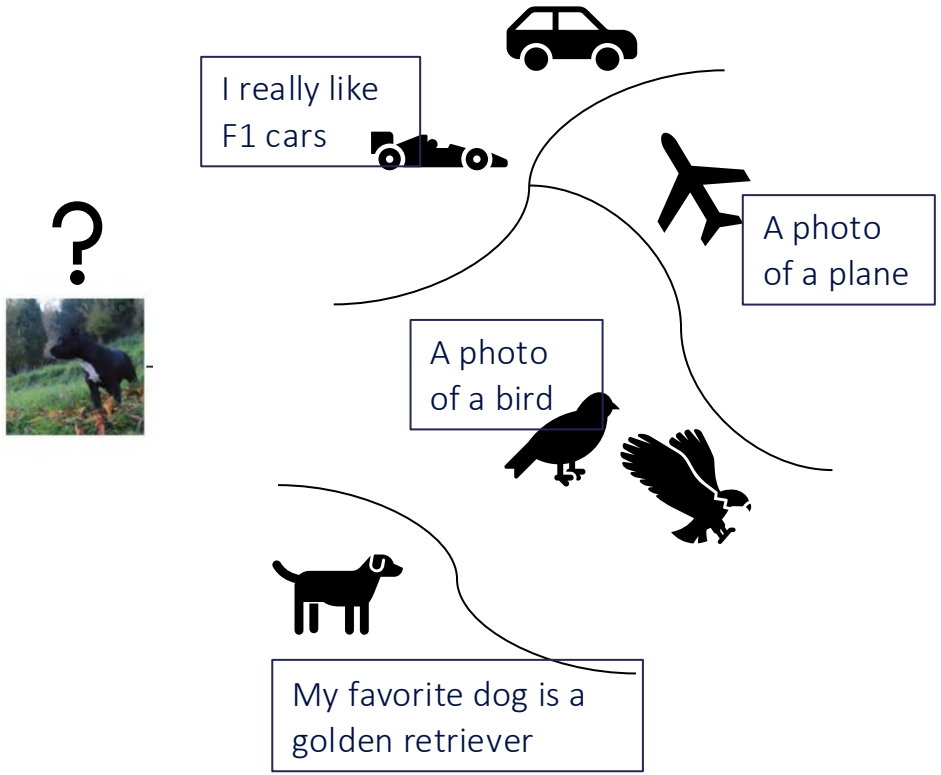
- Training batchsize: 32,768
- Training time:
  - RN50x64: 18 days on 592 V100 GPUs
  - ViT-L/14: 12 days on 256 V100 GPUs

https://openai.com/blog/clip/

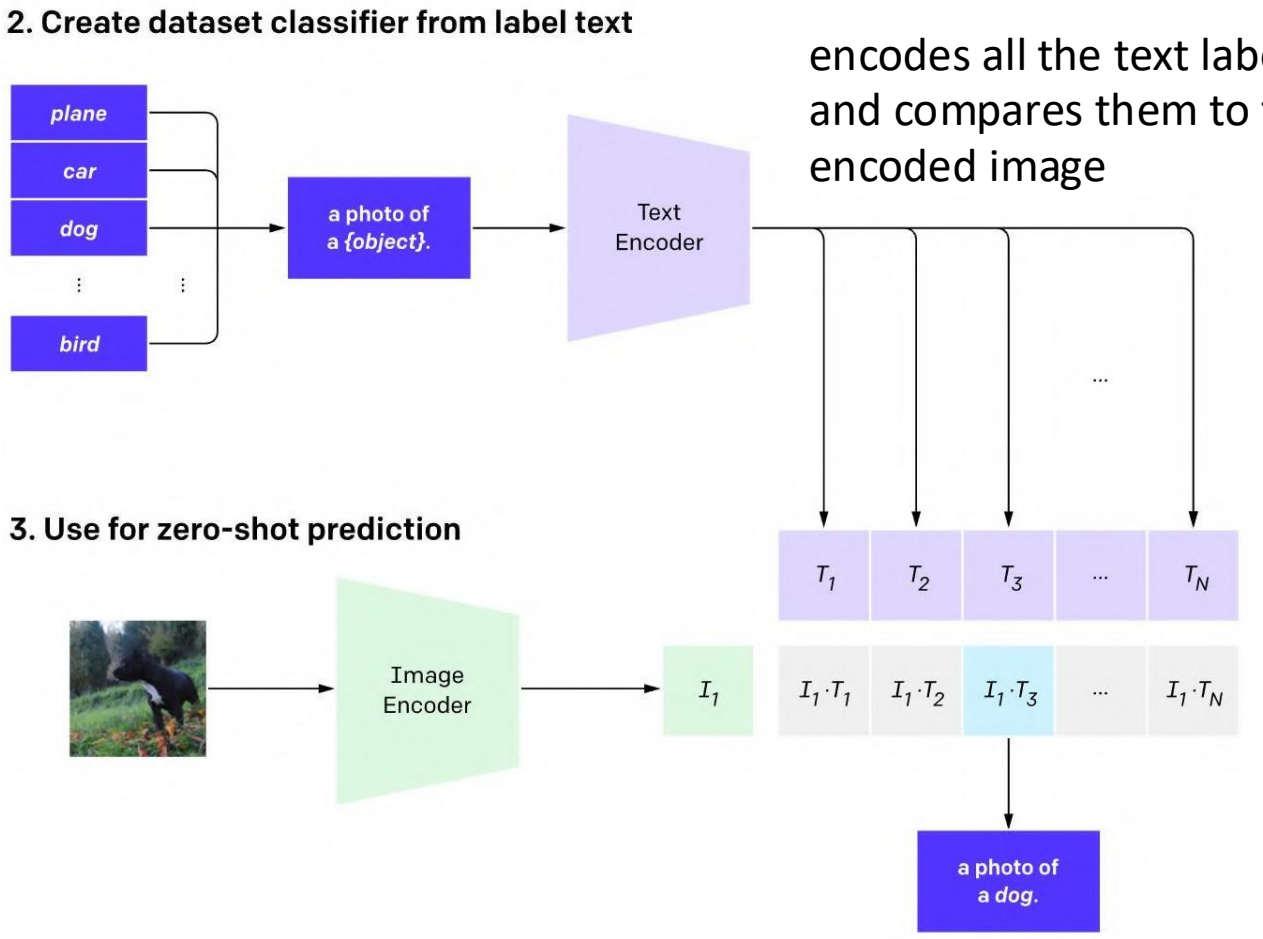
# CLIP for zero-shot learning



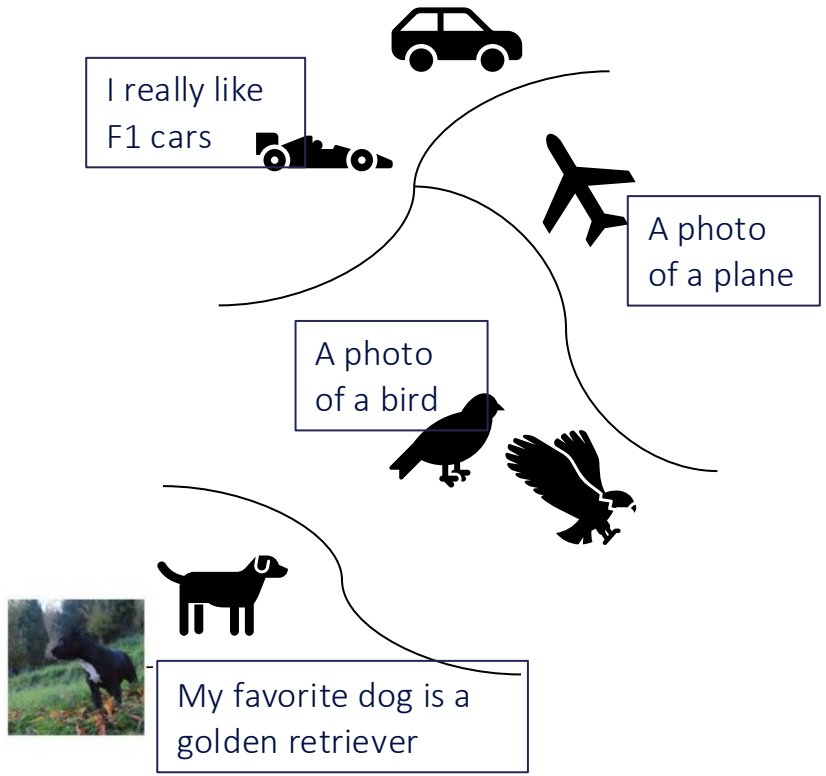

encodes all the text labels and compares them to the encoded image



# CLIP for zero-shot learning




encodes all the text labels and compares them to the encoded image




MINERVA
THANK YOU!
QUESTIONS?