Deep Learning

Convolutional Neural network (CNN)

Lionel Fillatre

2025-2026

Outline of Lecture 3

- Convolutional layers
- Convolution
- Convolution in practice
- Strides and padding
- Pooling
- A few famous CNN
- Conclusion

Convolutional layers

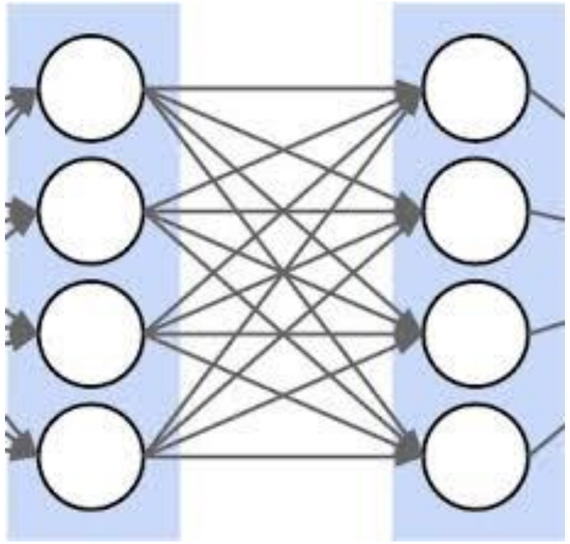
Why would we need them?

- If they were handled as normal "unstructured" vectors, large-dimension signals such as sound samples or images would require models of intractable size.
- For instance a linear layer taking a 256×256 RGB image as input, and producing an image of same size would require:

$$(256 \times 256 \times 3) \times (256 \times 256 \times 3) \simeq 3.87e^{+10}$$

parameters, with the corresponding memory footprint (≃150Gb with float32!), and excess of capacity.

16 connections



Why would we need them?

- Moreover, this requirement is inconsistent with the intuition that such large signals have some "invariance in translation". A representation meaningful at a certain location can / should be used everywhere.

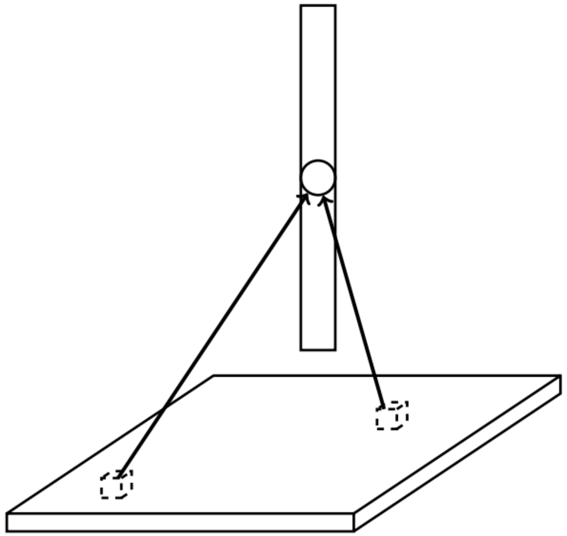
Cat

Cat

- A convolutional layer embodies this idea. It applies the same linear transformation locally, everywhere, and preserves the signal structure.

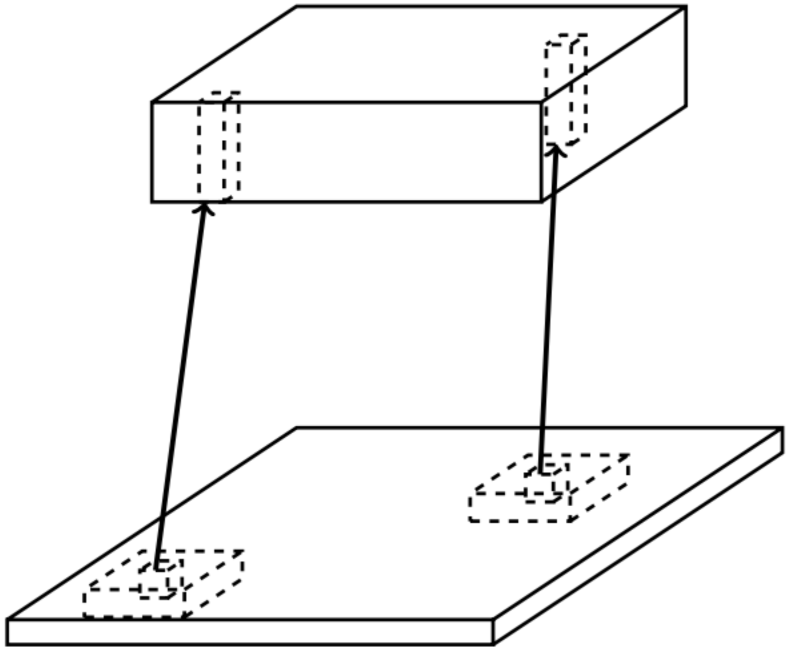
Fully connected layer

- In a fully connected layer, each hidden unit $h = \sigma(W^T x + \boldsymbol{b})$ is connected to the entire image.
- Looking for activations that depend on pixels that are spatially far away is supposedly a waste of time and ressources.
- Long range correlations can be dealt with in the higher layers.



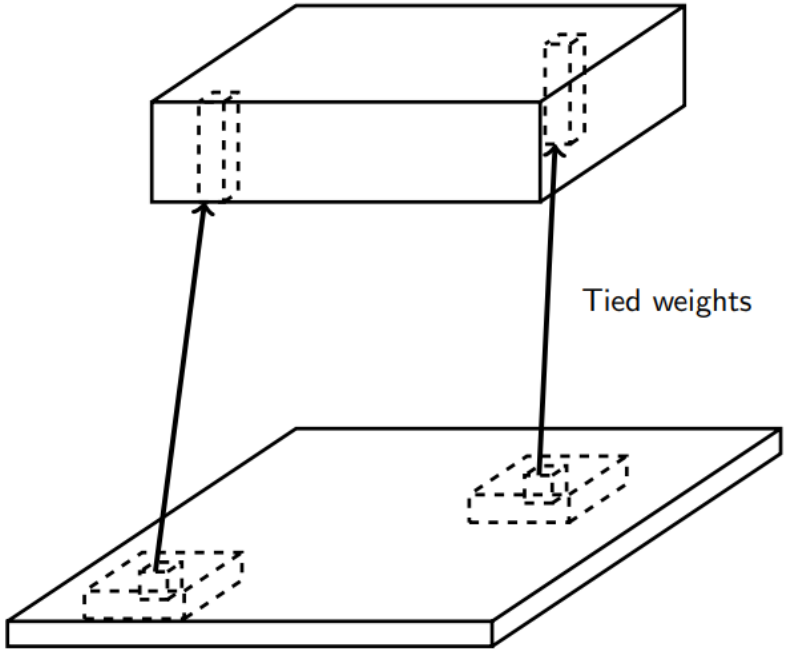
Locally connected layer

- In a locally connected layer, each hidden unit h is connected to only a patch of the input layer.
- Weights are specialized locally and functionally.
- Reduce the number of parameters.
- What if the object in the image shifts a little?



Convolutional layer

- In a convolutional layer, each hidden unit h_j is connected to only a patch of the input layer, and share its weights with the other units h_i.
- Weights are specialized functionally, regardless of spatial location.
- Reduce the number of parameters.



Convolution

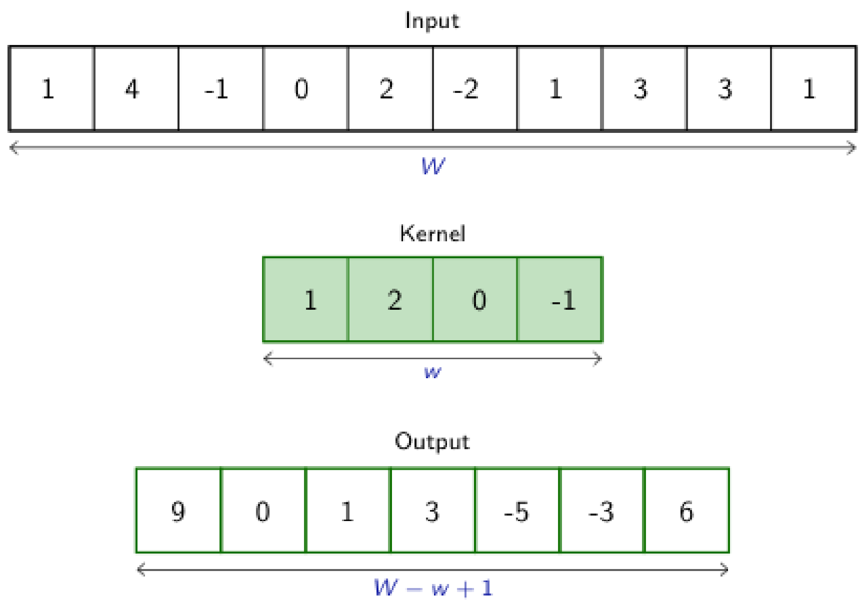
Convolution 1D

- For one-dimensional tensors, given an input vector $x \in \mathbb{R}^W$ and a convolutional kernel $u \in \mathbb{R}^w$, the discrete convolution $x * u$ is a vector of size $W - w + 1$ such that

$$(x * u)_i = \sum_{m=0}^{w-1} x_{i+m \bmod W} \, u_m$$

- Technically, $*$ denotes the cross-correlation operator. However, most machine learning libraries call it convolution.

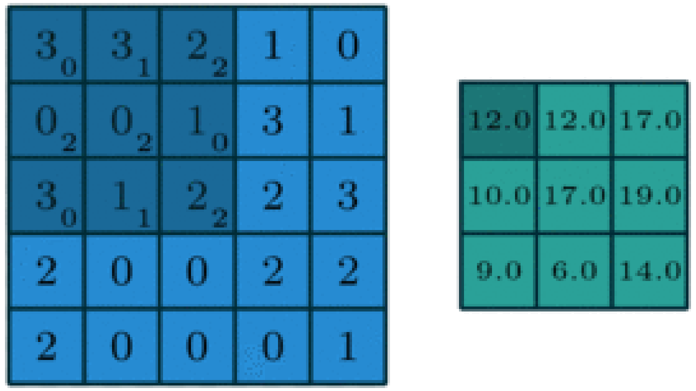
Convolution 1D



A convolution on an image

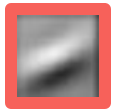
- Image: x of dimensions $H \times W$
- Kernel: u of dimensions $h \times w$

$$(x * u)_{i,j} = \sum_{n=0}^{h-1} \sum_{m=0}^{w-1} x_{i+n \bmod H, j+m \bmod W}\, u_{n,m}$$



Animation from V. Dumoulin https://github.com/vdumoulin/conv_arithmetic

Convolution as image filtering

- Filter  $\begin{pmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{pmatrix}$

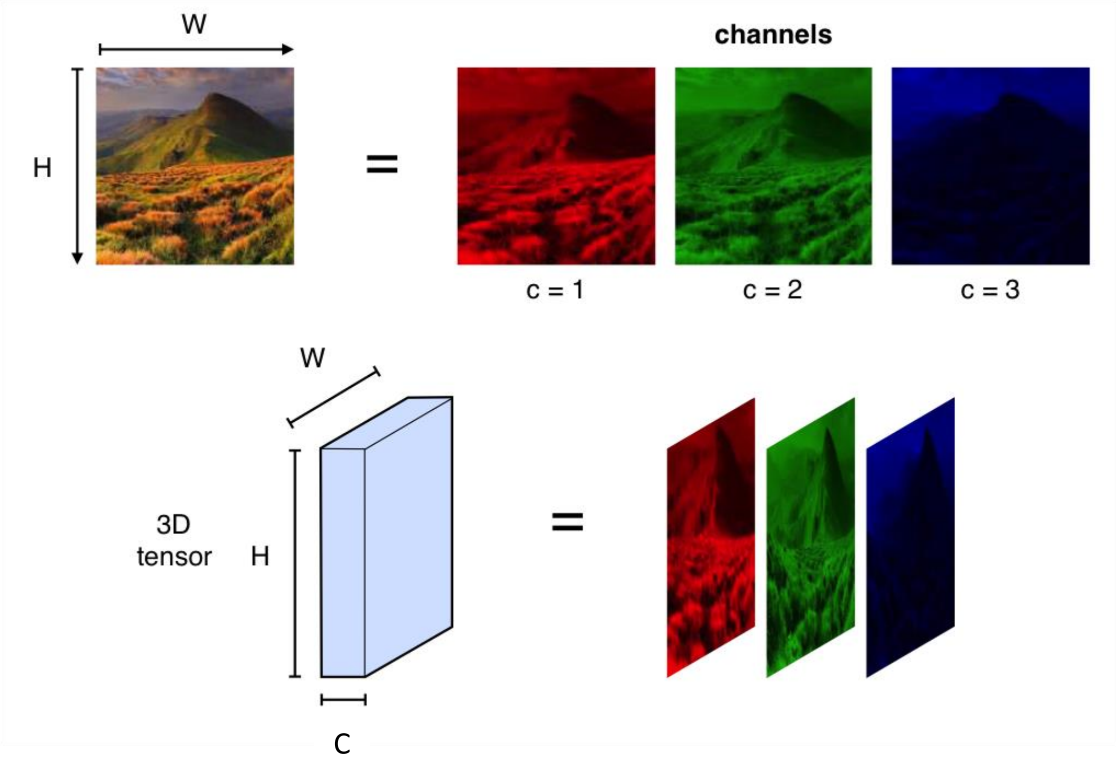
Input Image

Convoluted Image

- Inspired by the neurophysiological experiments conducted by Hubel and Wiesel 1962.

Convolution 3D: Channels

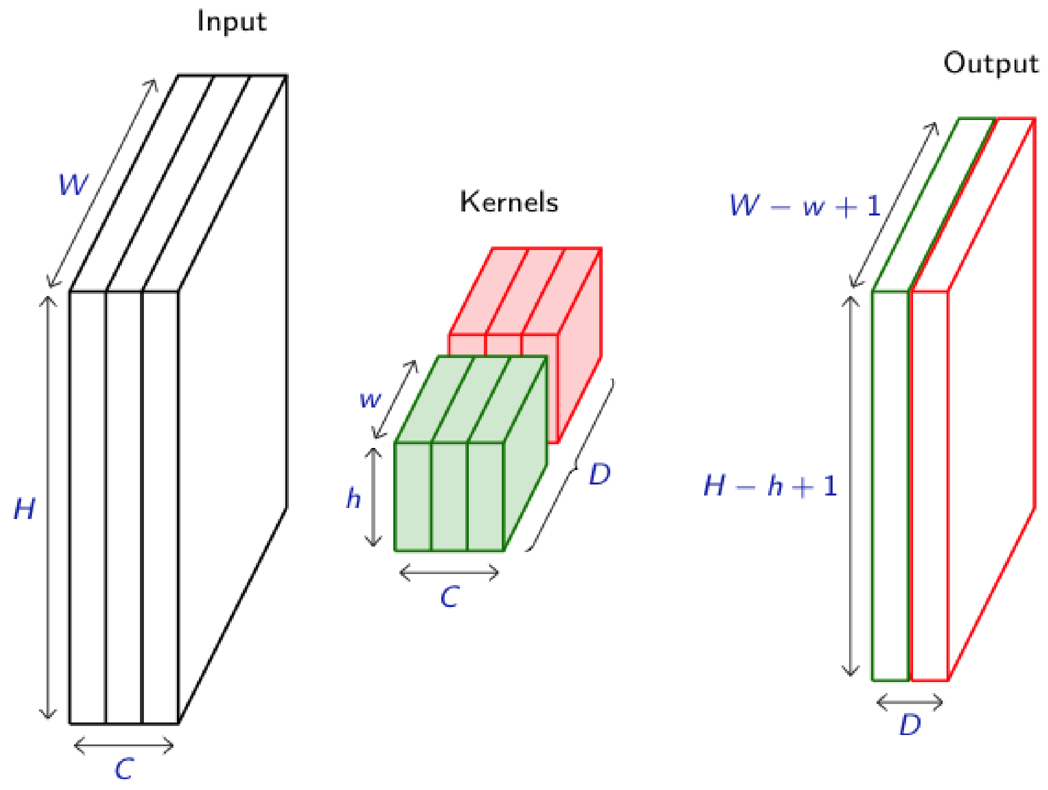
- For first layer, RGB channels of input image can be easily visualized
- Number of channels is typically increased at deeper levels of the network



Convolution of 3D tensors

- Convolutions generalize to multi-dimensional tensors
- In its most usual form, a convolution takes as input a 3D tensor $x \in \mathbb{R}^{C \times H \times W}$, called the input feature map.
- A kernel $u \in \mathbb{R}^{C \times h \times w}$ slides across the input feature map, along its height and width.
- The size $h \times w$ is called the receptive field.
- At each location, the element-wise product between the kernel and the input elements it overlaps is computed and the results are summed up.

Convolution of 3D tensors



Convolution 3D output

- The final output $o = \left(o_{i,j}\right)$ is a 2D tensor of size $(H - h + 1) \times (W - w + 1)$ called the output feature map and such that:

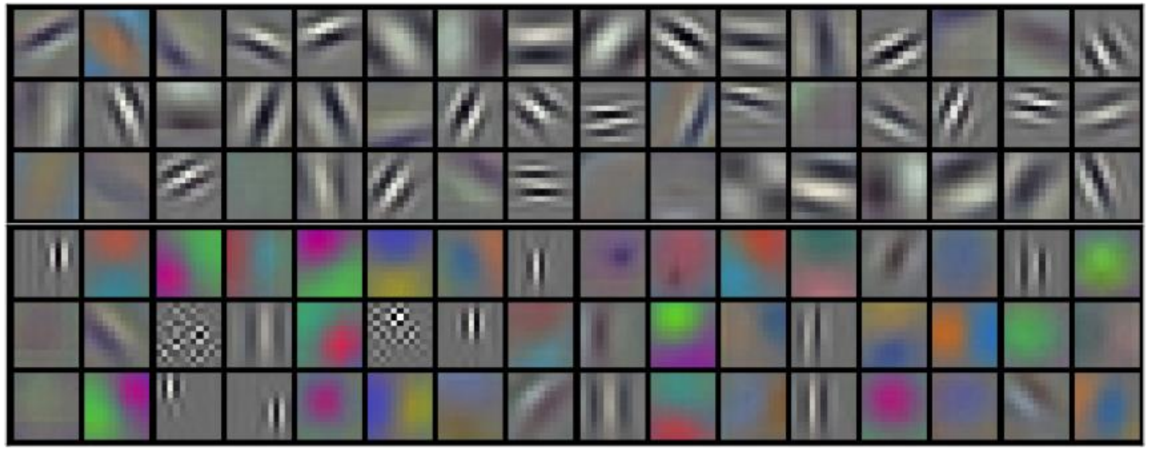
$$o_{i,j} = b_{i,j} + \sum_{c=0}^{C-1} (x_c * u_c)_{i,j} = b_{i,j} + \sum_{c=0}^{C-1} \sum_{n=0}^{h-1} \sum_{m=0}^{w-1} x_{c,i+n \bmod H, j+m \bmod W} \;\; u_{c,n,m}$$

where u and b are shared parameters to learn.

- D convolutions can be applied in the same way to produce a $D \times (H - h + 1) \times (W - w + 1)$ feature map, where D is the depth.

How to visualize the kernels?

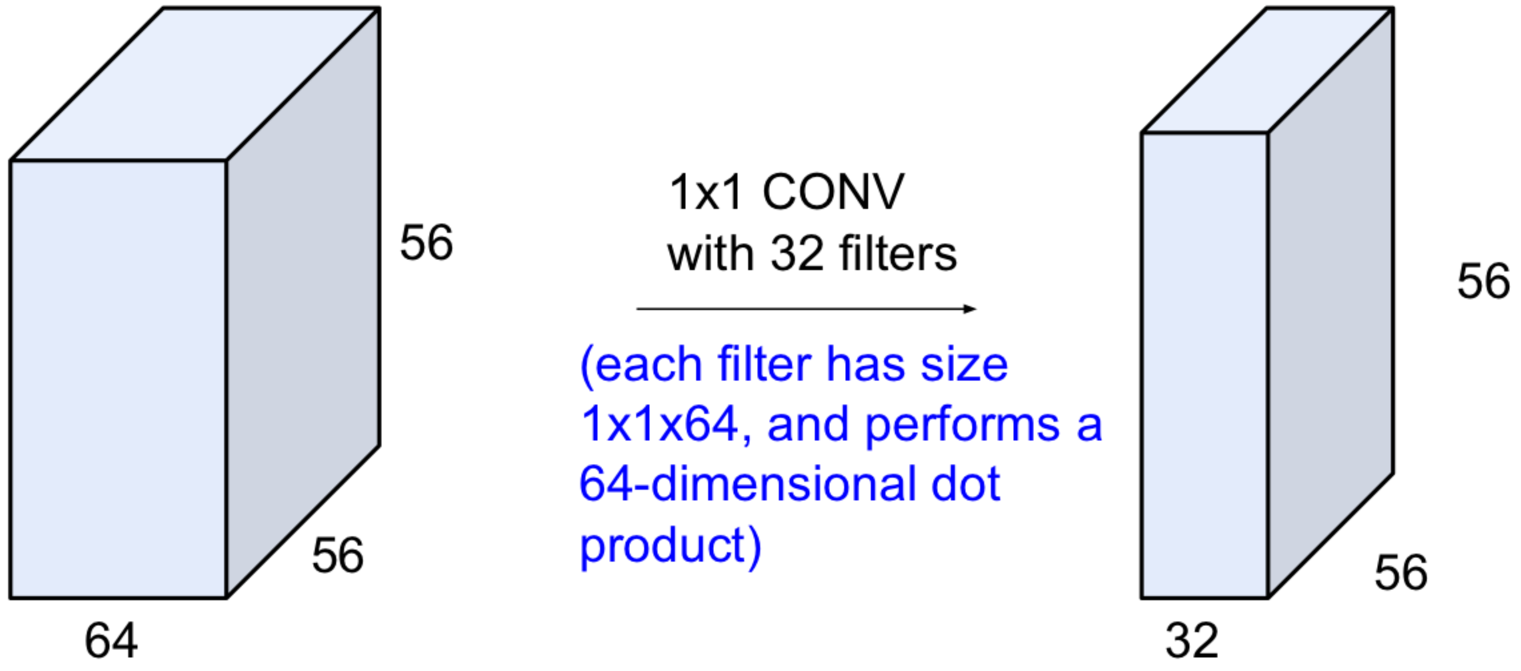
- A bank of 3D filters (learned from RGB inputs)
- Each of the 96 filters shown here is of size [11x11x3], and each one is shared by the 55x55 neurons in one depth slice.
- Notice that the parameter sharing assumption is relatively reasonable:
  - If detecting a horizontal edge is important at some location in the image, it should intuitively be useful at some other location as well due to the translationally-invariant structure of images.
  - There is therefore no need to relearn to detect a horizontal edge at every one of the 55x55 distinct locations in the convolutional layer output volume.



A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," *Commun. ACM*, vol. 60, pp. 84–90, May 2017.

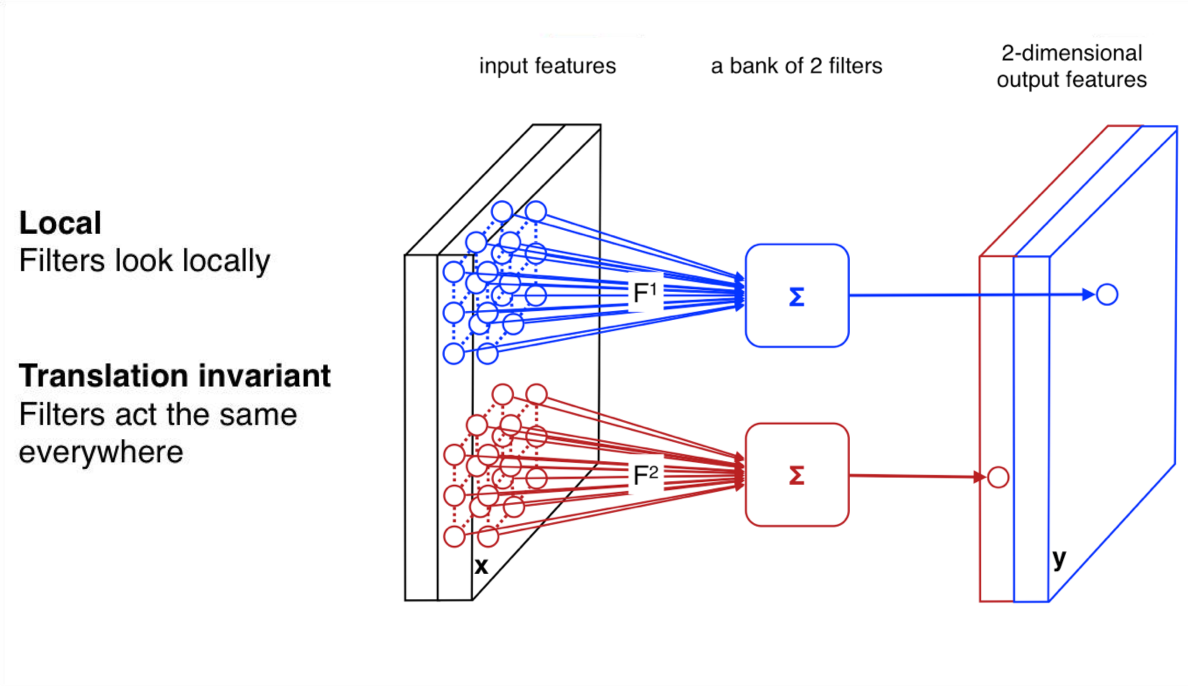
Convolutions 1×1

- 1×1 convolution layers: aggregating pixel information from all feature maps



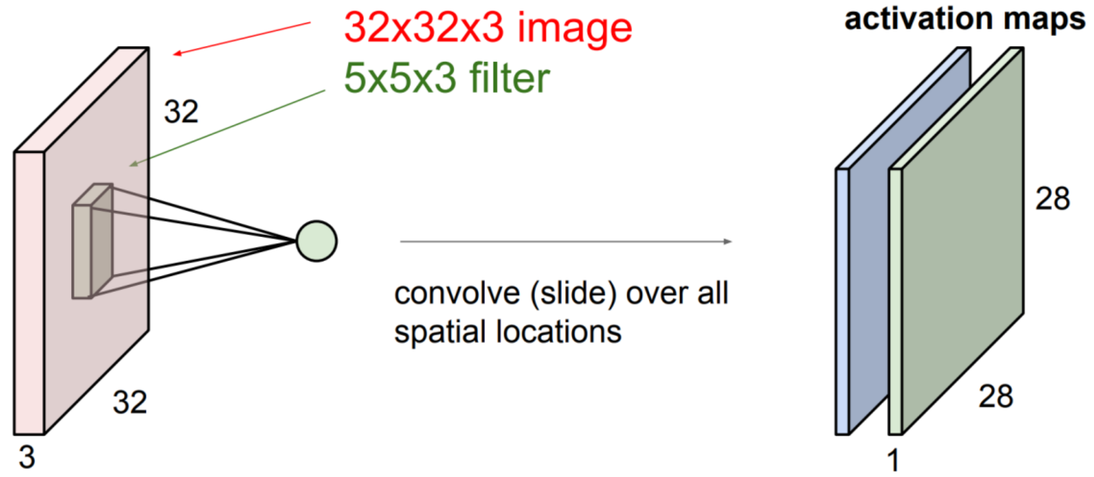
Convolutions as neurons

- Since convolutions output one scalar at a time, they can be seen as an individual neuron from a MLP with a receptive field limited to the dimensions of the kernel
- The same neuron is "fired" over multiple areas from the input.



Activation Map

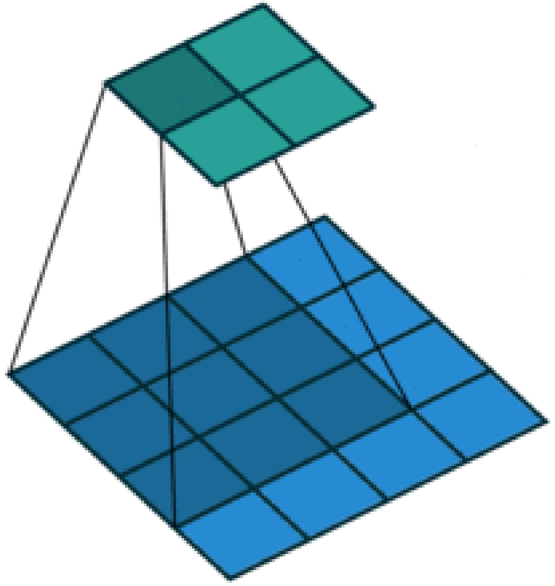
- We usually refer to one of the channels generated by a convolution layer as an activation map.
- The sub-area of an input map that influences a component of the output as the receptive field of the latter.
- In the context of convolutional networks, a standard linear layer is called a fully connected layer since every input influences every output.



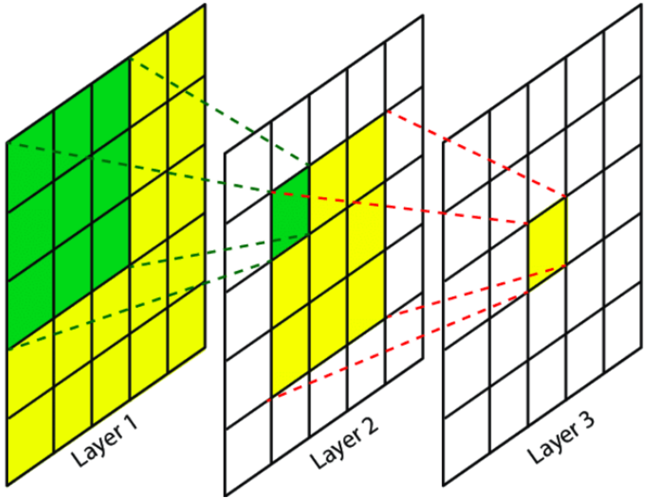
Receptive field

- The receptive field is defined as the region in the input space that a particular CNN's feature is looking at (i.e. be affected by).
- A receptive field of a feature can be fully described by its center location and its size
- Example: kernel size 3×3

Convolution without zero-padding and with stride of 1

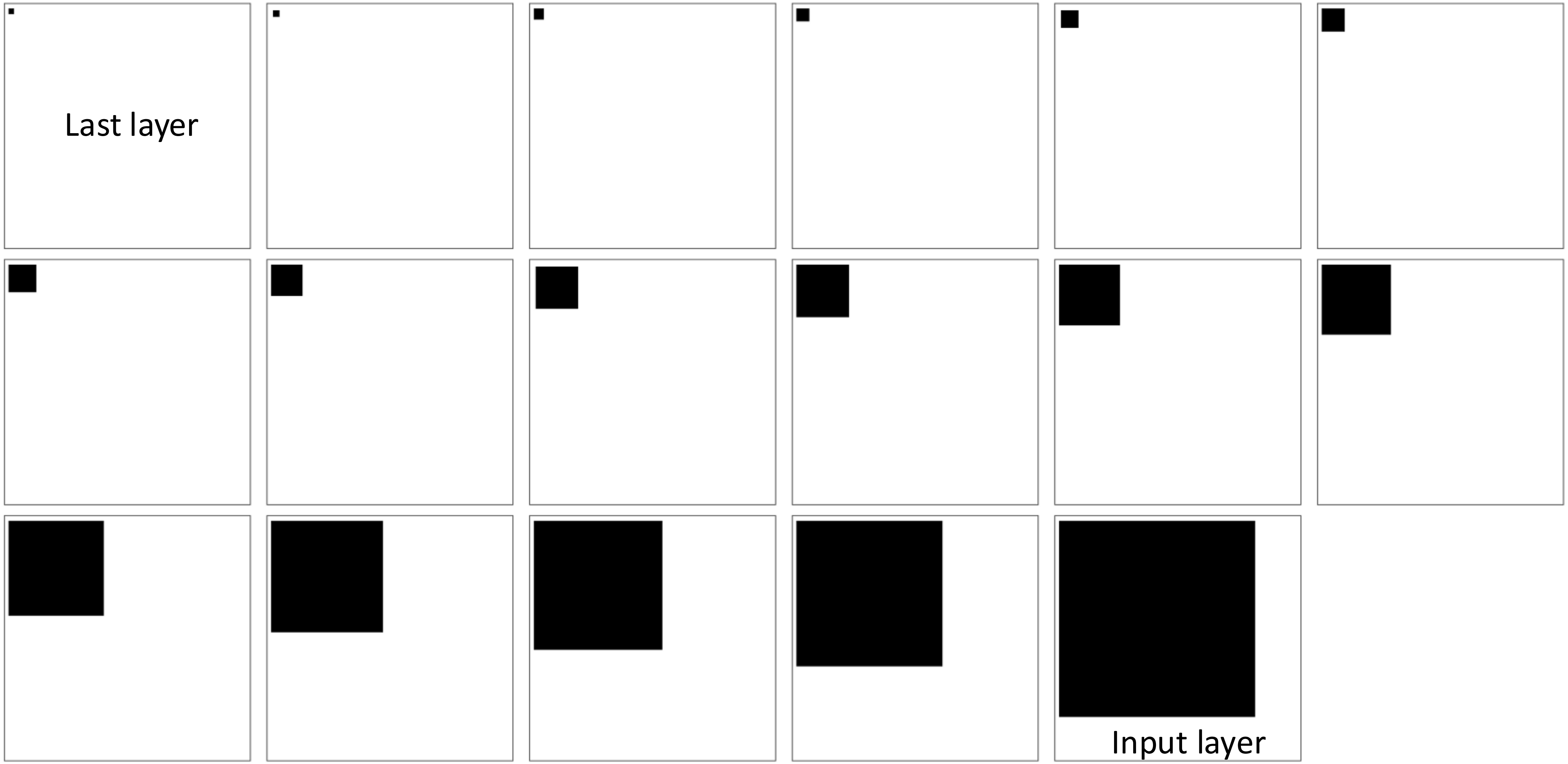


Receptive field 5×5 in Layer 1 for a feature in Layer 3



https://theaisummer.com/receptive-field/
https://medium.com/mlreview/a-guide-to-receptive-field-arithmetic-for-convolutional-neural-networks-e0f514068807

Receptive field



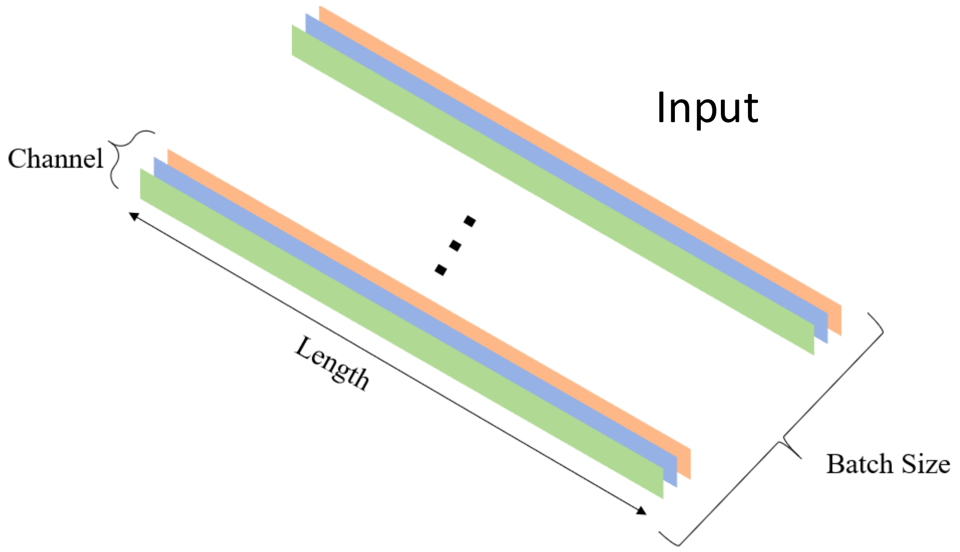
Receptive fields for convolutional and pooling layers of VGG-16

Convolution in Pytorch

1D Convolution in Pytorch

torch.nn.functional.conv1d(input, weight, bias=None, stride=1, padding=0, dilation=1, groups=1)

- Implements a 1D convolution, where
  - weight (contains the kernels) is of dimension $D \times C \times w$, where D is the number of output channels
  - bias is of dimension D,
  - input is of dimension $N \times C \times W$ where N is the batch size and C the number of input channels,
  - and the result is of dimension: $N \times D \times (W - w + 1)$



https://programmer.group/pytorch-learning-conv1d-conv2d-and-conv3d.html

Code in Pytorch

```
>>> import torch
>>> out_channel_number = 4 # number of out channels
>>> in_channel_number = 2 # number of in channels
>>> w = 3 # 1D filter length
>>> W = 5 # 1D input signal length
>>> batch_size = 3
>>> filters = torch.randn(out_channel_number, in_channel_number, w)
>>> inputs = torch.randn(batch_size, in_channel_number, W)
>>> bias = torch.empty(out_channel_number).normal_()
>>> outputs = torch.nn.functional.conv1d(inputs, filters,bias)
>>> print(inputs)
>>> print(outputs)
```

```
tensor([[[-0.2564, -1.4026, -2.6301,  0.4547, -0.5276],
         [-0.3589,  0.8100,  0.3939, -0.3209, -0.1702]],

        [[-0.5567, -0.7665,  0.3204,  1.1852,  1.3797],
         [-1.0769,  0.4855,  1.5382,  0.1565, -0.4238]],

        [[ 0.0183,  1.0366,  1.2764,  0.3992,  0.6236],
         [-1.4047, -0.3058,  0.1974, -1.0961, -0.1314]]])
```

Input: $3 \times 2 \times 5$

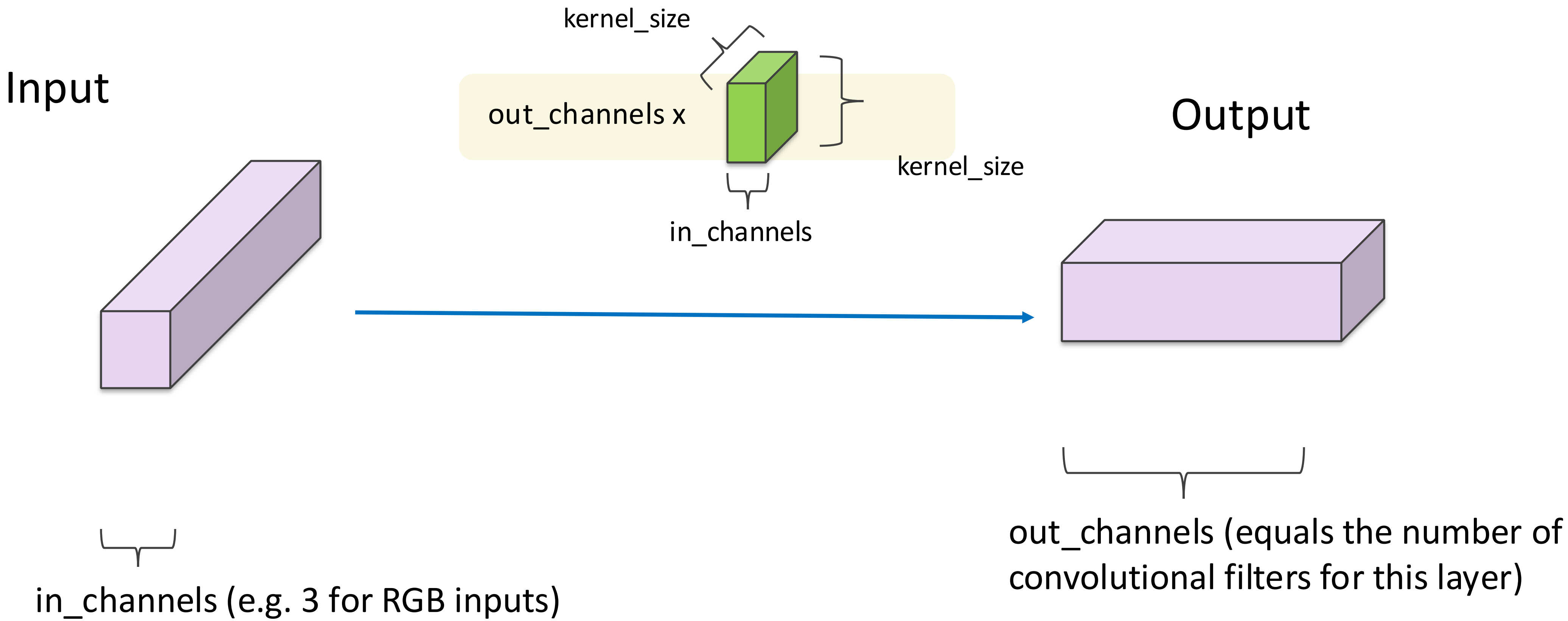
```
tensor([[[ 3.5633, -1.5768,  0.2834],
         [ 4.4160,  2.7037,  4.8476],
         [ 0.7849, -1.0373,  1.7854],
         [-2.8596, -6.1889, -5.9252]],

        [[-3.4493, -1.6813, -2.4046],
         [ 2.8996,  0.1544, -4.4264],
         [ 1.9660, -3.2219, -6.8773],
         [-0.7422, -4.1237, -3.3361]],

        [[-3.8895, -0.0725, -3.1728],
         [-0.8058, -3.6168, -4.2987],
         [-0.6345, -5.3933, -4.5297],
         [ 2.1737,  1.1122,  1.0123]]])
```

Output: $3 \times 4 \times (5 - 3 + 1)$

2D Convolutional Layer in Pytorch



2D Convolution with Pytorch

torch.nn.functional.conv2d(input, weight, bias=None, stride=1, padding=0, dilation=1, groups=1)

- Implements a 2D convolution, where weight contains the kernels, and is of dimension $D \times C \times h \times w$, bias is of dimension D, input is of dimension

$$N \times C \times H \times W$$

and the result is of dimension:

$$N \times D \times (H - h + 1) \times (W - w + 1)$$

```
>>> weight = torch.empty(5, 4, 2, 3).normal_()
>>> bias = torch.empty(5).normal_()
>>> input = torch.empty(117, 4, 10, 3).normal_()
>>> output = torch.nn.functional.conv2d(input, weight, bias)
>>> output.size()
torch.Size([117, 5, 9, 1])
```

- A similar function implements 3d convolution.

Convolution inside a module

class torch.**nn**.**Conv2d**(in_channels, out_channels, kernel_size, stride=1, padding=0, dilation=1, groups=1, bias=True)

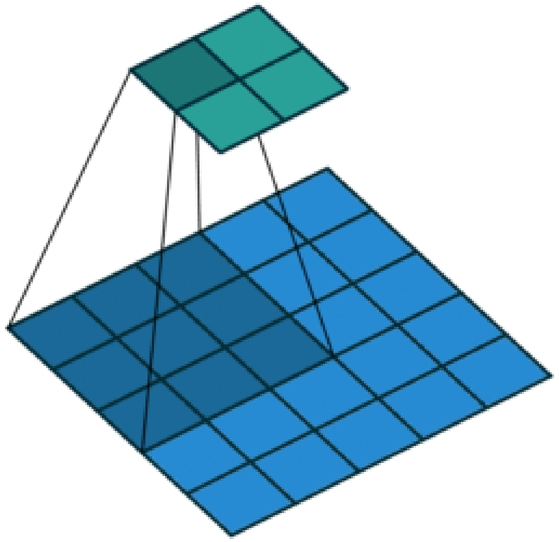
- Wraps the convolution into a Module, with the kernel and biases as parameter properly randomized at creation.
- The kernel size is either a pair (h, w) or a single value k interpreted as (k, k).

```
>>> f = nn.Conv2d(in_channels = 4, out_channels = 5, kernel_size = (2, 3))
>>> for n, p in f.named_parameters(): print(n, p.size())
weight torch.Size([5, 4, 2, 3])
bias torch.Size([5])
>>> x = torch.empty(117, 4, 10, 3).normal_()
>>> y = f(x)
>>> y.size()
torch.Size([117, 5, 9, 1])
```

Strides and padding

Strides

- Strides: increment step size for the convolution operator
- Reduces the size of the ouput map

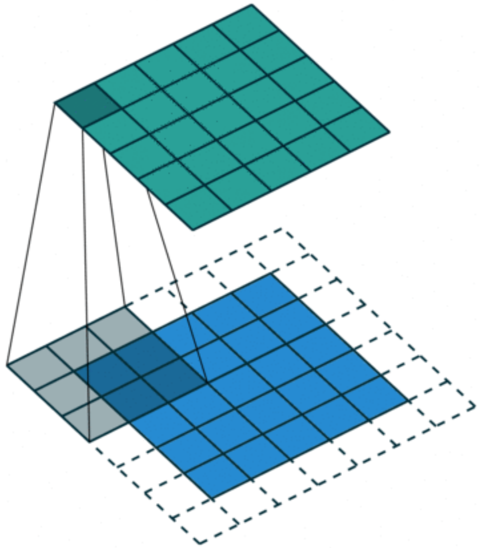


Example with kernel size 3×3 and a stride of 2 (image in blue)

https://syncedreview.com/2017/05/11/a-guide-to-receptive-field-arithmetic-for-convolutional-neural-networks/

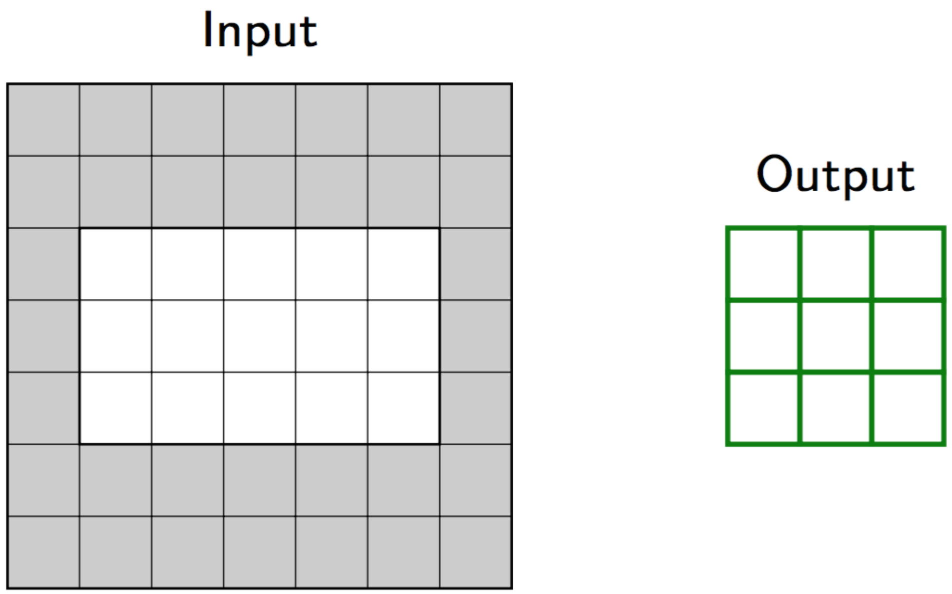
Padding

- Padding: artifically fill borders of image
- Useful to keep spatial dimension constant across filters
- Useful with strides and large receptive fields
- Usually: fill with 0s



Padding

- Example: input $C \times 3 \times 5$, padding of $(2{,}1)$, a stride of $(2{,}2)$, kernel of size $C \times 3 \times 3$



- Pooling operations (see later) have a default stride equal to their kernel size, and convolutions have a default stride of 1.
- Padding can be useful to generate an output of same size as the input.

Number of output features

$$n_{out} = \left\lfloor \frac{n_{in} + 2p - k}{s} \right\rfloor + 1$$

- n_{in}: number of input features
- n_{out}: number of output features
- k: convolution kernel size
- p: convolution padding size
- s: convolution stride size

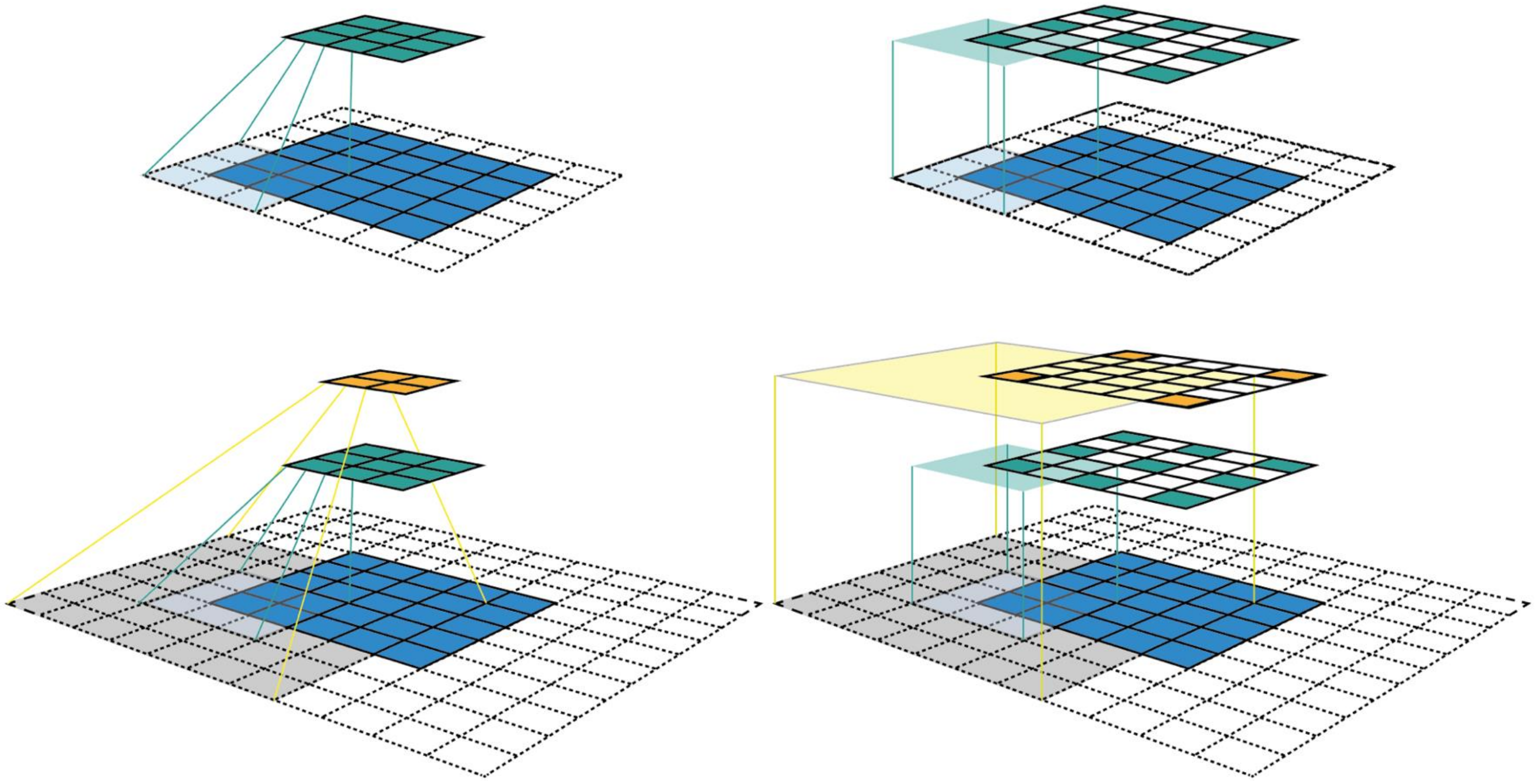
Receptive field with stride and padding

- Example: kernel size $k = 3 \times 3$; padding size $p = 1 \times 1$; stride $s = 2 \times 2$; input=5×5

Left: Non fixed-sized CNN feature map visualization

Right: Fixed-sized CNN feature map visualization, where the size of each feature map is fixed, and the **feature is located at the center** of its receptive field.

Note: a white pixel is a non-computed pixel.



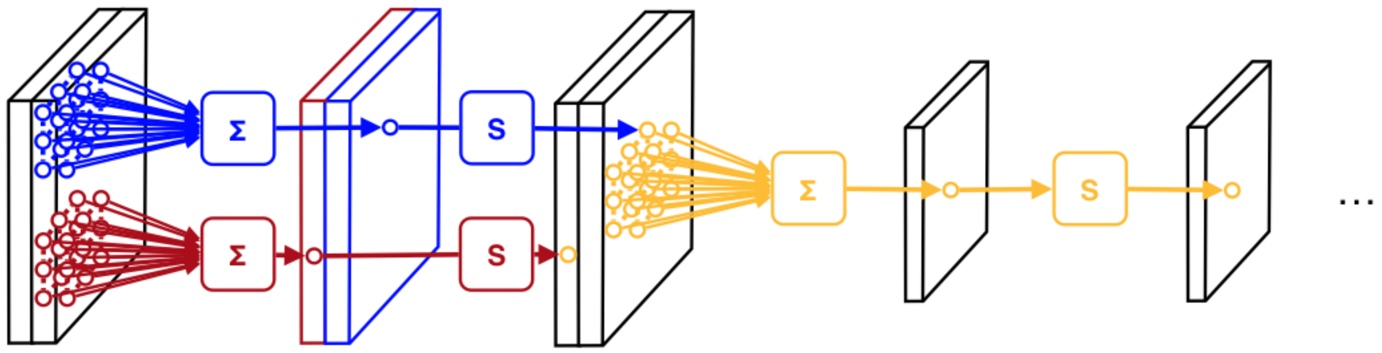
A common way to visualize a CNN feature map.

https://medium.com/mlreview/a-guide-to-receptive-field-arithmetic-for-convolutional-neural-networks-e0f514068807

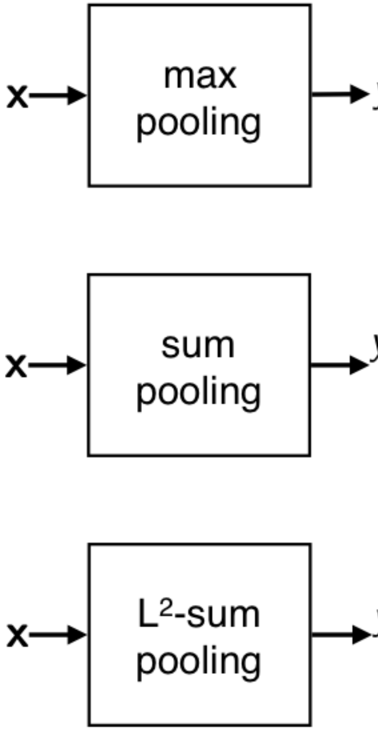
Pooling

Downsampling

- Downsampling by a factor S amount to keeping only one every S pixels, discarding others
- Filter banks often incorporate or are followed by 2x output downsampling
- Downsampling is often matched with an increase in the number of feature channels
- Overall the volume of the tensors decreases slowly



Spatial pooling



$$y_{ijk} = \max\{x_{uvk} : (u, v) \in \mathcal{N}(i, j)\}$$

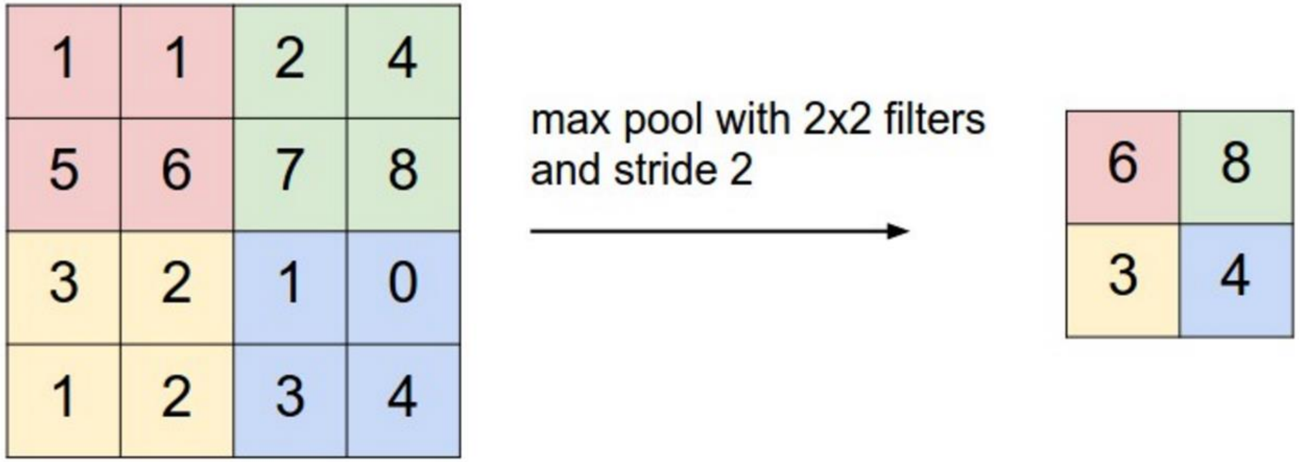
$$y_{ijk} = \sum_{(u,v)\in\mathcal{N}(i,j)} x_{uvk}$$

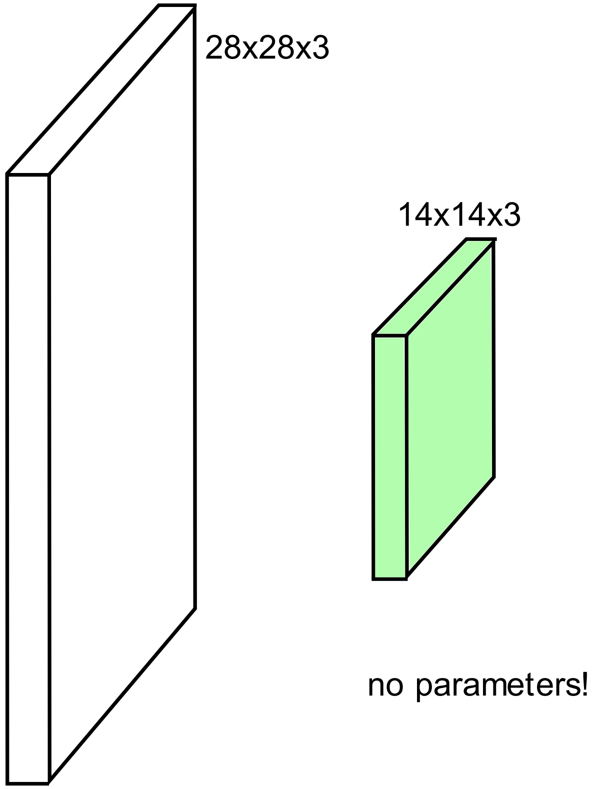
$$y_{ijk} = \sqrt{\sum_{(u,v)\in\mathcal{N}(i,j)} x_{uvk}^2}$$

By far, the most common variant is **max pooling**.

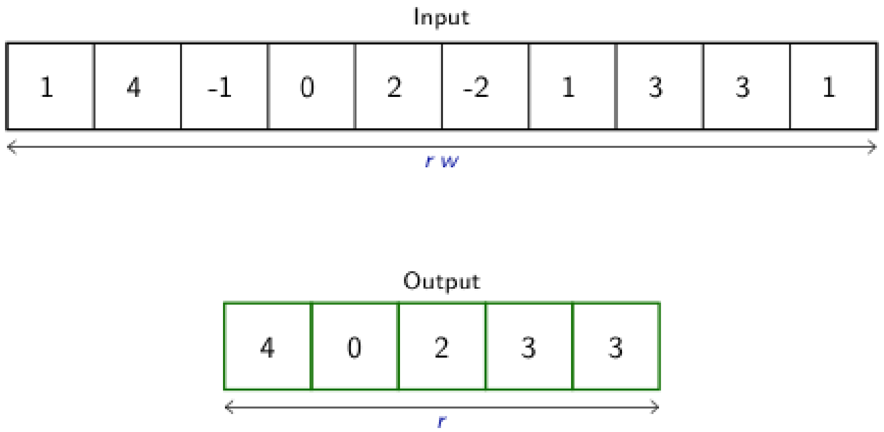
Pooling

- Spatial dimension reduction
- Local invariance
- No parameters: max or average of 2×2 units

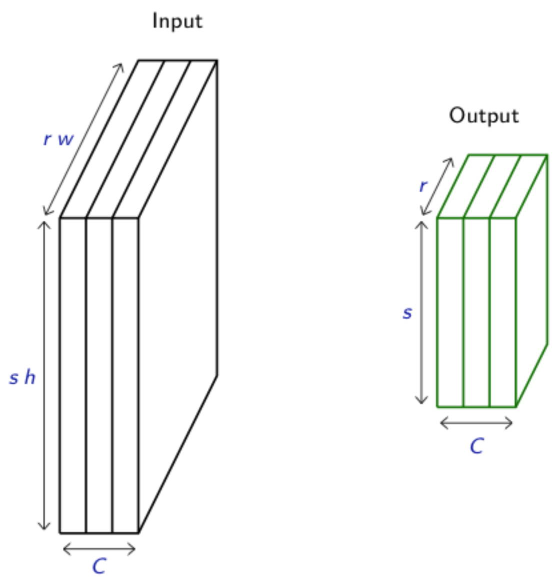




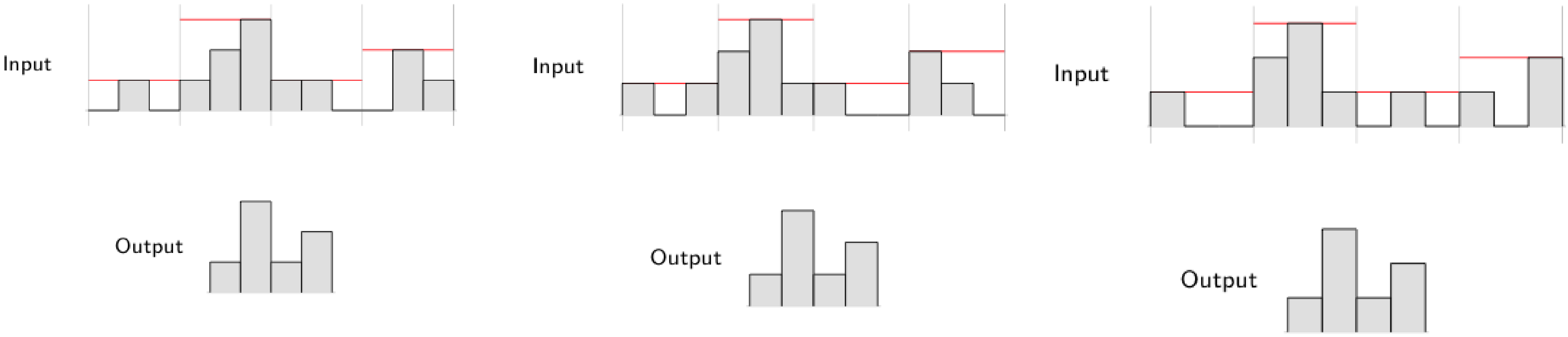
Max-Pooling 1D



Max-Pooling 2D



Translation invariance from pooling



Pooling in Pytorch

torch.nn.functional.max_pool2d(input, kernel_size, stride=**None**, padding=0, dilation=1, ceil_mode=**False**, return_indices=**False**)

- takes as input a $N \times C \times H \times W$ tensor, and a kernel size (h,w) or k interpreted as (k,k).
- applies the max-pooling on each channel of each sample separately, and produce if the padding is 0 a $N \times C \times \lfloor H/h \rfloor \times \lfloor W/w \rfloor$ output.

```
>>> x = torch.empty(2, 2, 6).random_(3)
>>> x
tensor([[[ 1., 2., 2., 1., 2., 1.],
[ 2., 0., 0., 0., 1., 0.]],
[[ 2., 0., 2., 1., 1., 1.],
[ 0., 0., 0., 1., 2., 1.]]])
>>> F.max_pool2d(x, (1, 2))
tensor([[[ 2., 2., 2.],
[ 2., 0., 1.]],
[[ 2., 2., 1.],
[ 0., 1., 2.]]])
```

- Similar function implements 1d and 3d max-pooling, and average pooling

Stride and Padding

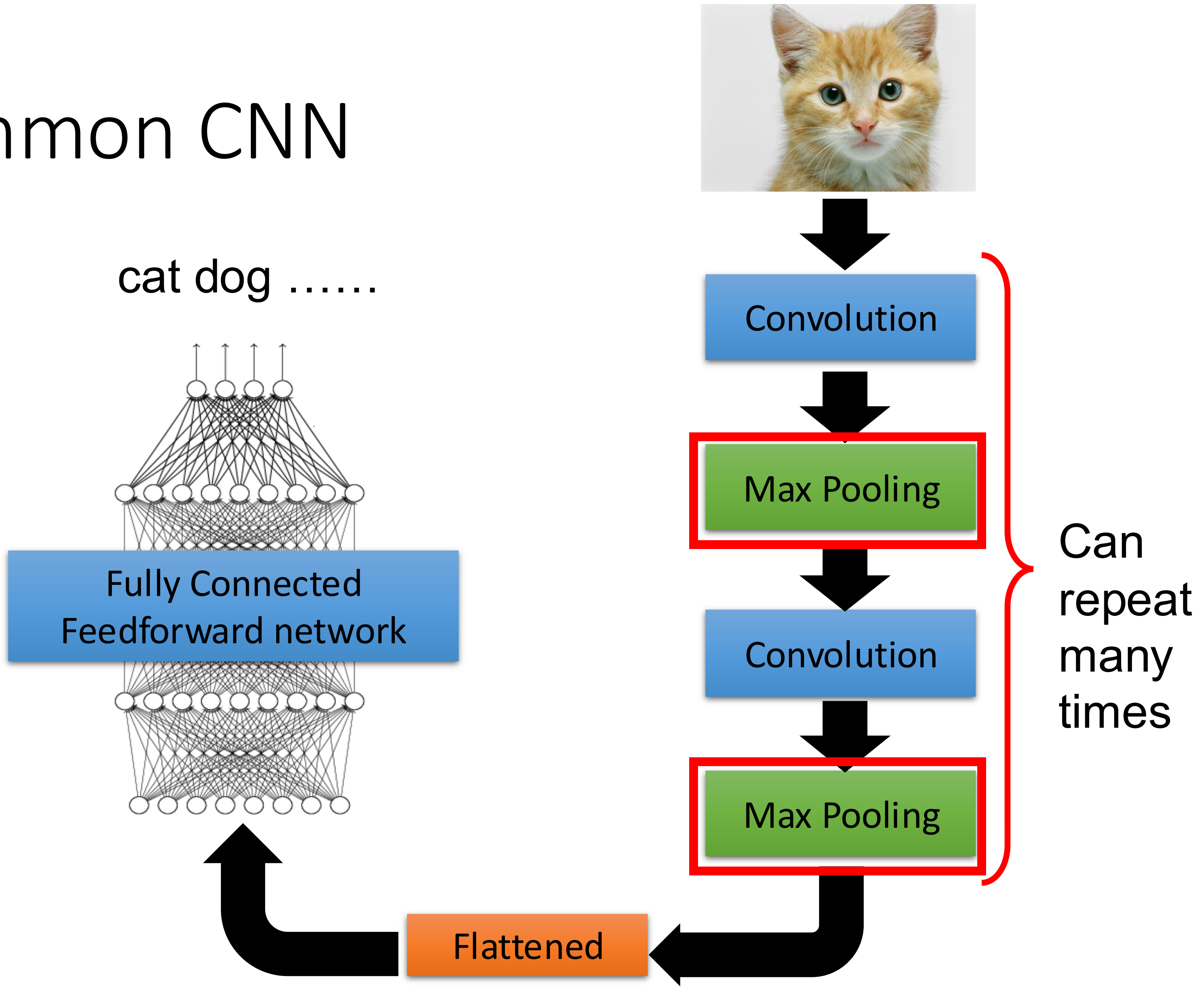
- As for convolution, pooling operations can be modulated through their stride and padding.
- While for convolution the default stride is 1, for pooling it is equal to the kernel size, but this not obligatory.
- Default padding is zero.

Pooling in a Module

- **class torch.nn.MaxPool2d**(kernel_size, stride=None, padding=0, dilation=1, return_indices=False, ceil_mode=False)
- Wraps the max-pooling operation into a Module.
- As for convolutions, the kernel size is either a pair (h, w) or a single value k interpreted as (k, k).

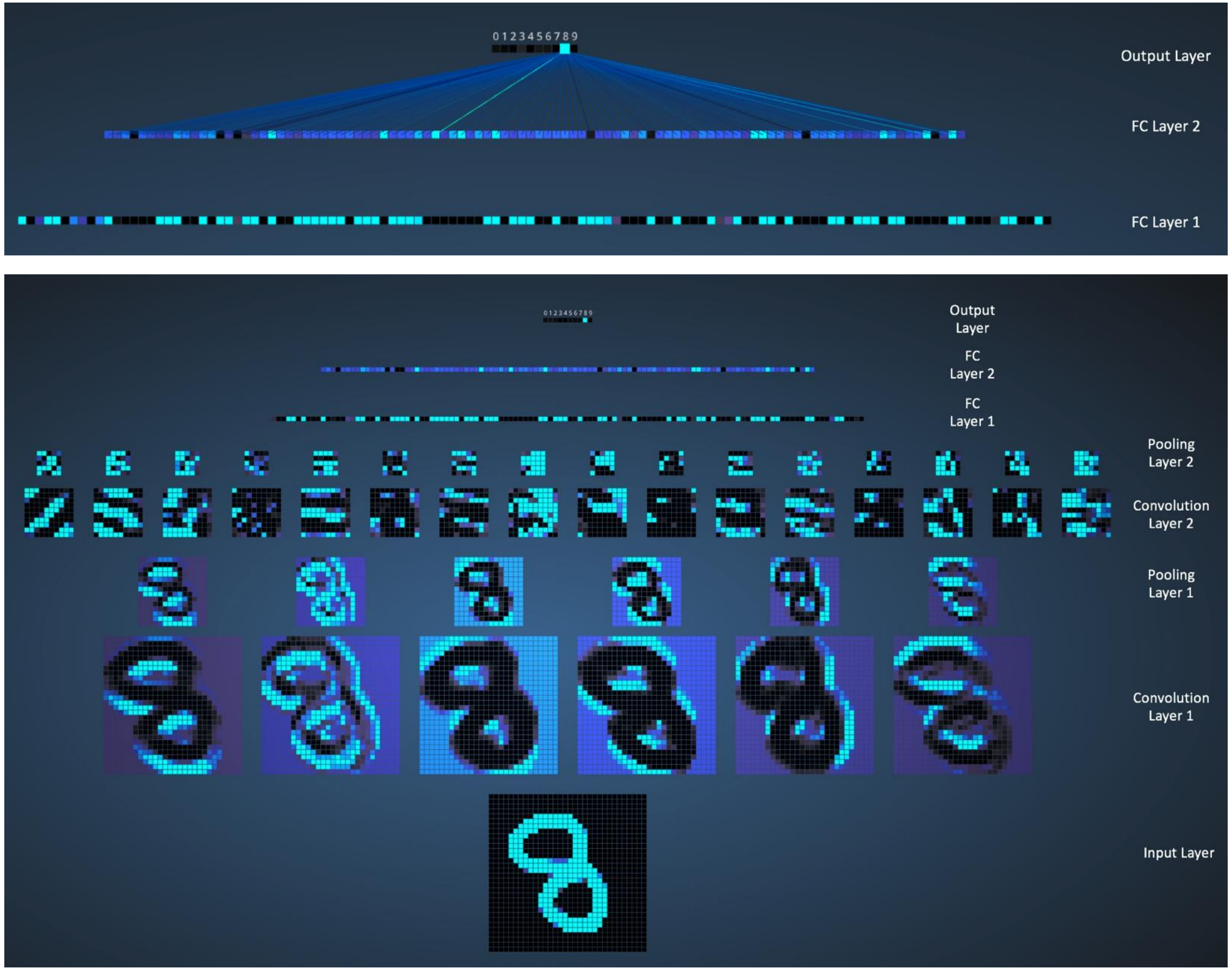
A few famous CNN

A common CNN



cat dog ……
Fully Connected
Feedforward network
Convolution
Max Pooling
Convolution
Max Pooling
Can
repeat
many
times
Flattened

Layer Visualization

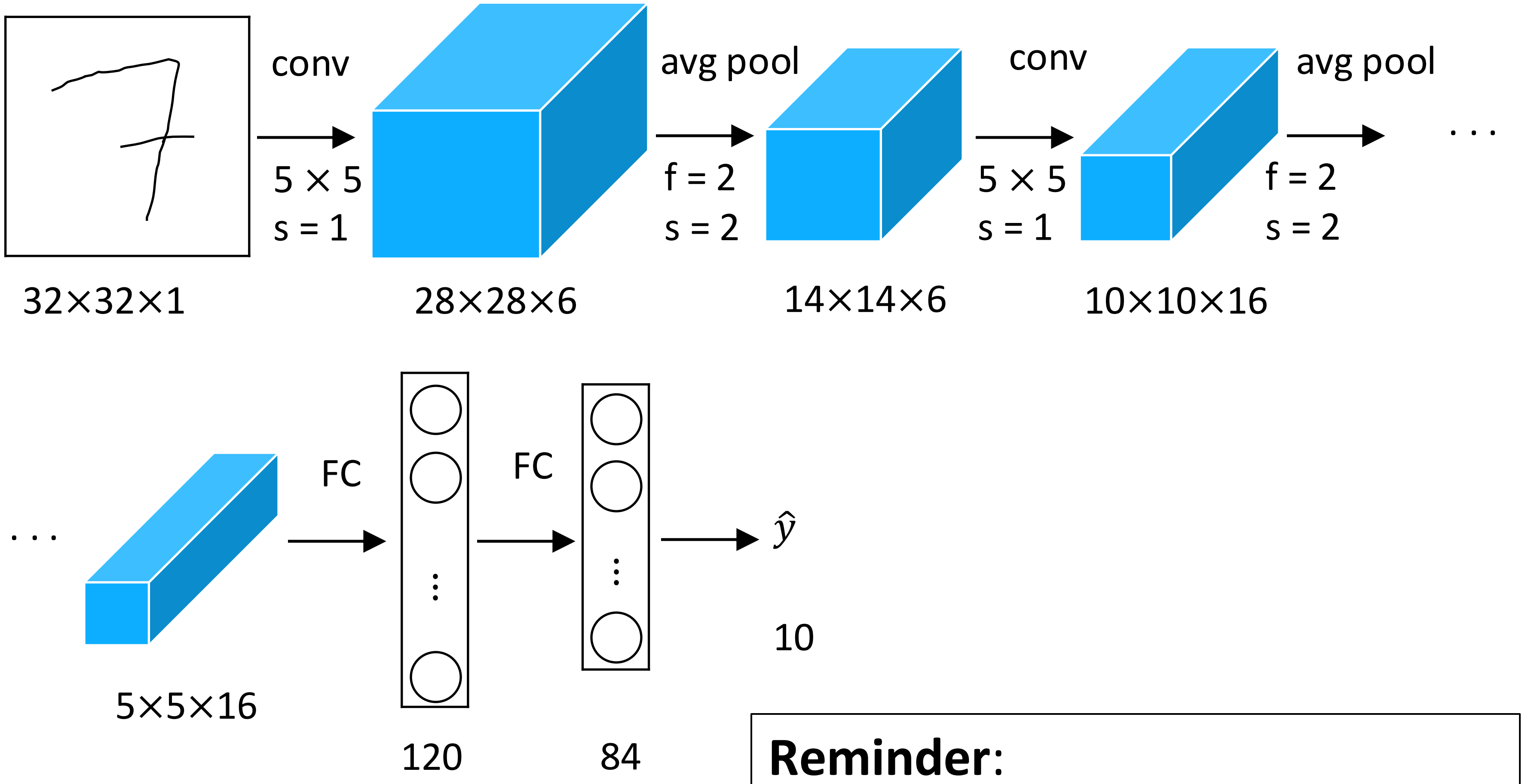


https://adamharley.com/nn_vis/cnn/2d.html

A large family of CNN
From 2012 – 2020: ConvNets dominate all vision tasks

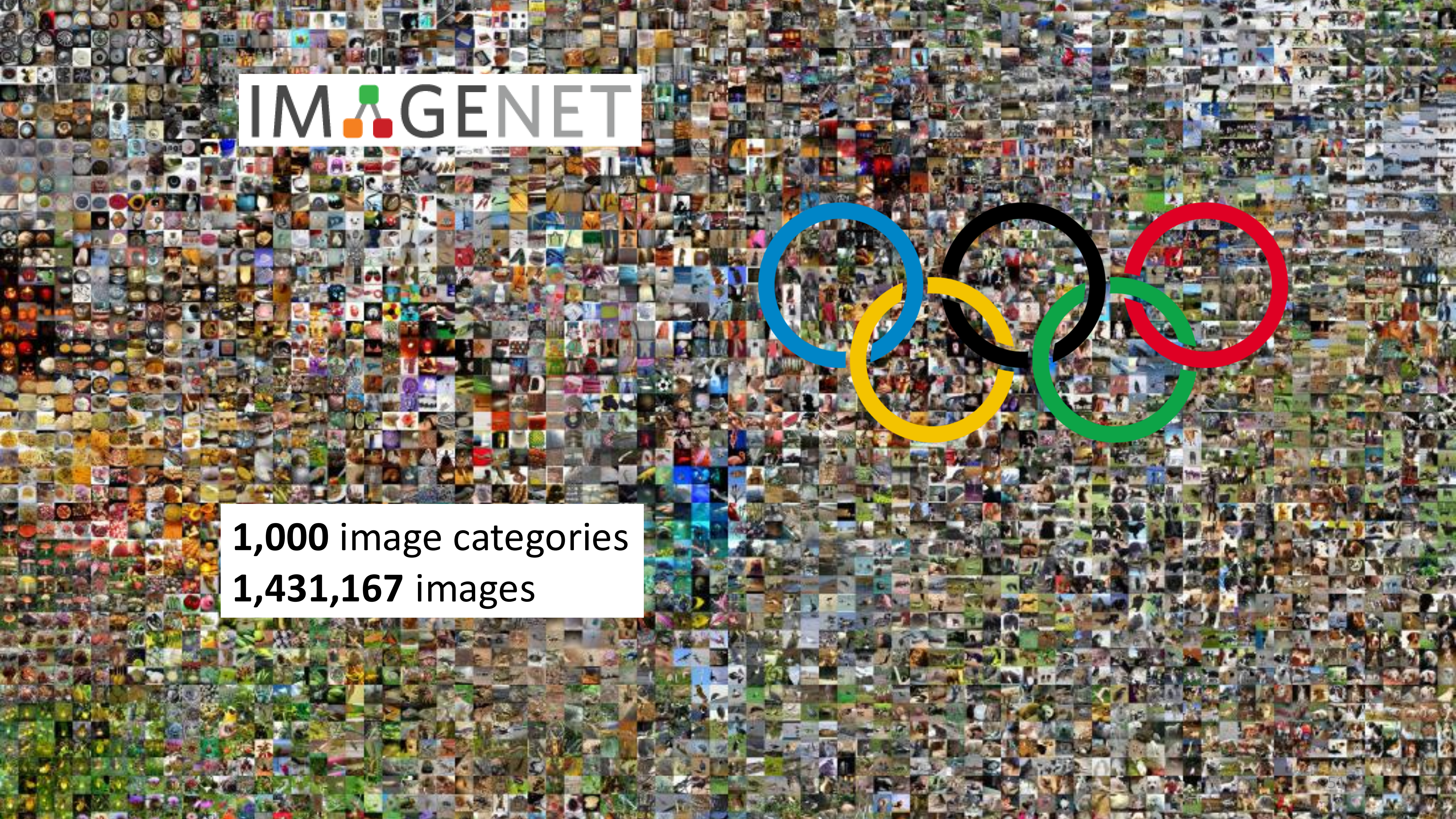
- **1989–1998**: [LeNet]
- **2012–2014**: [AlexNet & CaffeNet] [Maxout] [NIN] [ZFNet] [SPPNet]
- **2015**: [VGGNet] [Highway] [PReLU-Net] [STN] [DeepImage] [GoogLeNet / Inception-v1] [BN-Inception / Inception-v2]
- **2016**: [SqueezeNet] [Inception-v3] [ResNet] [Pre-Activation ResNet] [RiR] [Stochastic Depth] [WRN] [Trimps-Soushen]
- **2017**: [Inception-v4] [Xception] [MobileNetV1] [Shake-Shake] [Cutout] [FractalNet] [PolyNet] [ResNeXt] [DenseNet] [PyramidNet] [DRN] [DPN] [Residual Attention Network] [IGCNet / IGCV1] [Deep Roots]
- **2018**: [RoR] [DMRNet / DFN-MR] [MSDNet] [ShuffleNet V1] [SENet] [NASNet] [MobileNetV2] [CondenseNet] [IGCV2] [IGCV3] [FishNet] [SqueezeNext] [ENAS] [PNASNet] [ShuffleNet V2] [BAM] [CBAM] [MorphNet] [NetAdapt] [mixup] [DropBlock]
- **2019**: [ResNet-38] [AmoebaNet] [ESPNetv2] [MnasNet] [Single-Path NAS] [DARTS] [ProxylessNAS] [MobileNetV3] [FBNet] [ShakeDrop] [CutMix] [MixConv] [EfficientNet]
- **2020**: [Random Erasing (RE)]
- 2021 - Present: Transformers have taken over

LeNet-5

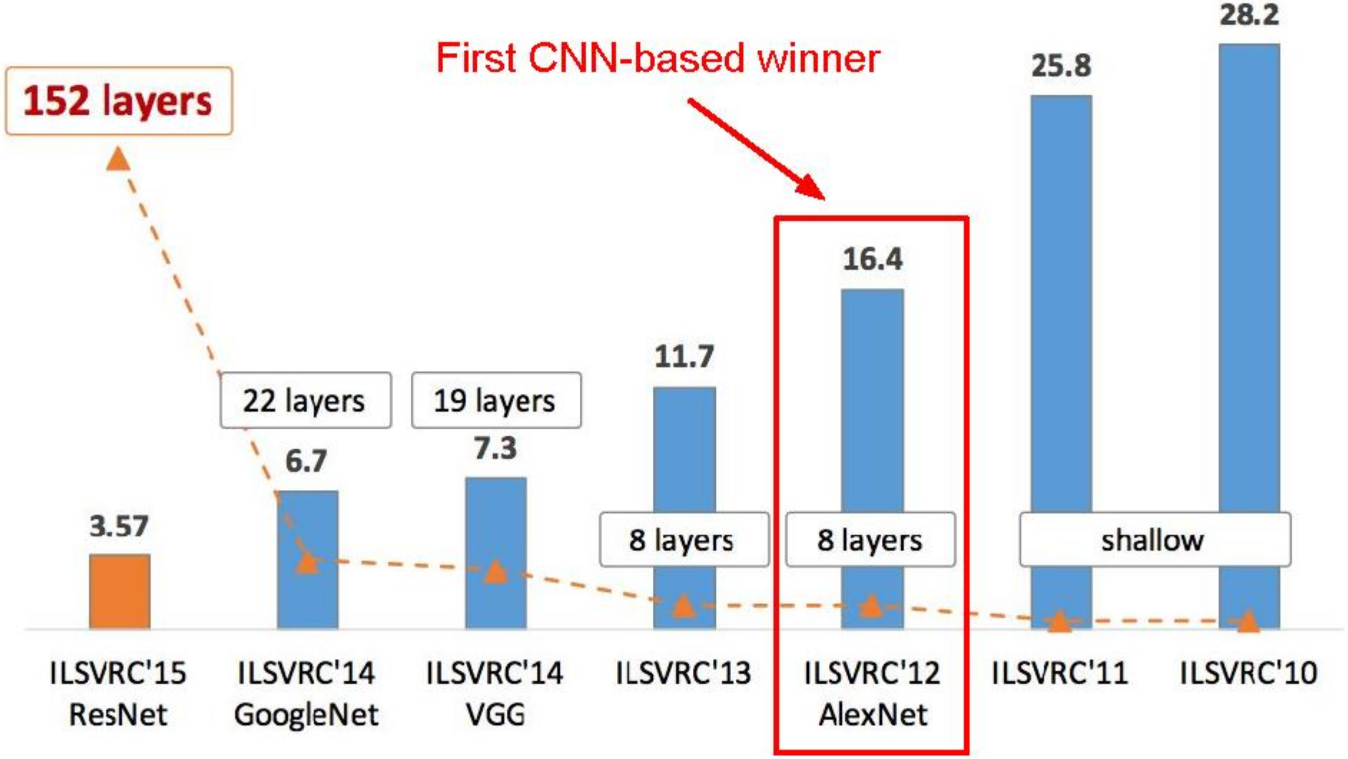


Reminder:
Output size = (N+2P-F)/stride + 1

Gradient Based Learning Applied To Document Recognition - Y. Lecun, L. Bottou, Y. Bengio, P. Haffner; 1998


IMAGENET
1,000 image categories
1,431,167 images

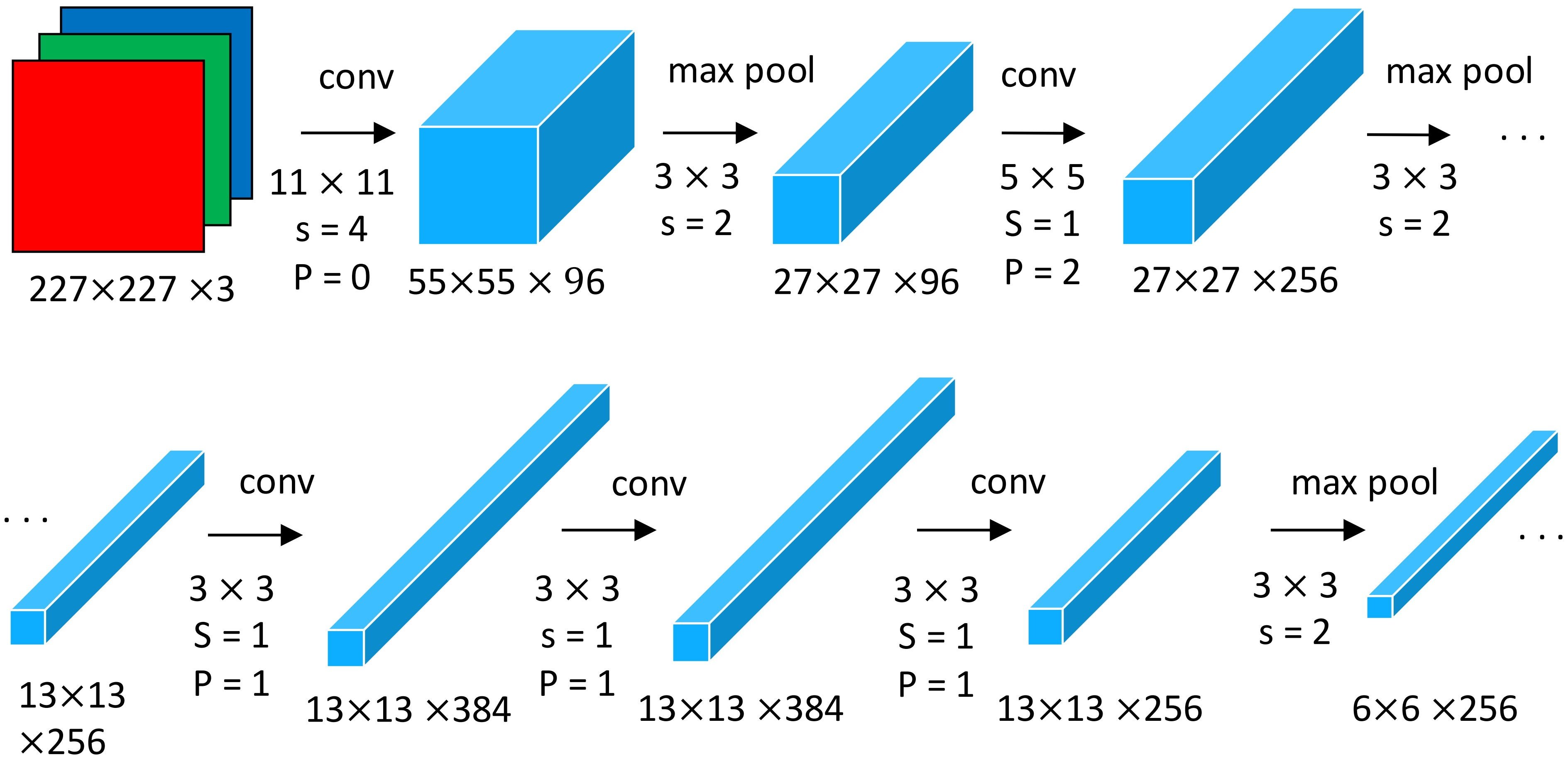
ImageNet Large Scale Visual Recognition Challenge (ILSVRC) winners



2013: All the 20 first ranking methods were using deep learning!!

Slide taken from Fei-Fei & Justin Johnson & Serena Yeung.

AlexNet



[Krizhevsky et al., 2012]

AlexNet

- Trained on GTX 580 GPU with only 3 GB of memory.

- Network spread across 2 GPUs, half the neurons (feature maps) on each GPU.

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

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

- Few details: first use of ReLU, not commun normalization layers, dropout, batch, SGD momentum, etc.

VGG Net (Oxford Net)

[Simonyan and Zisserman 2014]

- Runner-up of Imagenet 2014
- 7.3% top-5 error rate!
- Main idea:
    - Fix the filter size: 3x3
    - Fix the pooling 2x2 stride 2
    - Go deeper!
- Several model are proposed
- Trained on 4 Nvidia Titan Black GPUs for **two to three weeks**.

ConvNet Configuration					
A	A-LRN	B	C	D	E
11 weight layers	11 weight layers	13 weight layers	16 weight layers	16 weight layers	19 weight layers
input (224 × 224 RGB image)					
conv3-64	conv3-64 **LRN**	conv3-64 **conv3-64**	conv3-64 conv3-64	conv3-64 conv3-64	conv3-64 conv3-64
maxpool					
conv3-128	conv3-128	conv3-128 **conv3-128**	conv3-128 conv3-128	conv3-128 conv3-128	conv3-128 conv3-128
maxpool					
conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256 **conv1-256**	conv3-256 conv3-256 **conv3-256**	conv3-256 conv3-256 conv3-256 **conv3-256**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
FC-4096					
FC-4096					
FC-1000					
soft-max					

Best model

Table 2: **Number of parameters** (in millions).

Network	A,A-LRN	B	C	D	E
Number of parameters	133	133	134	138	144

VGG Net (Oxford Net)

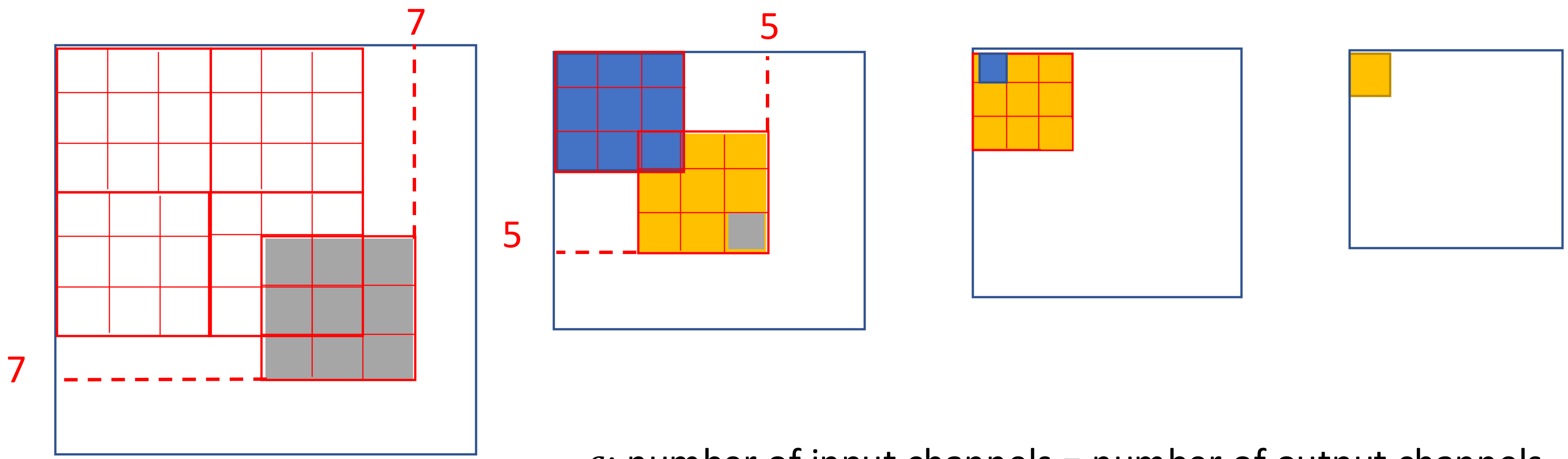
- Why 3x3 convolutions??
  1. Getting rid of a filter-size as a hyper-parameters

     Keep adding layers until they help!
  2. Similar receptive field, less parameters, more nonlinearities!

Comparison of a layer of 7x7 vs. 3 layers of 3x3

$((7 \times 7 \times c) \times c) \; = 49c^2$ vs $((3 \times 3 \times c) \times c) \times 3 \; = \; 27c^2$

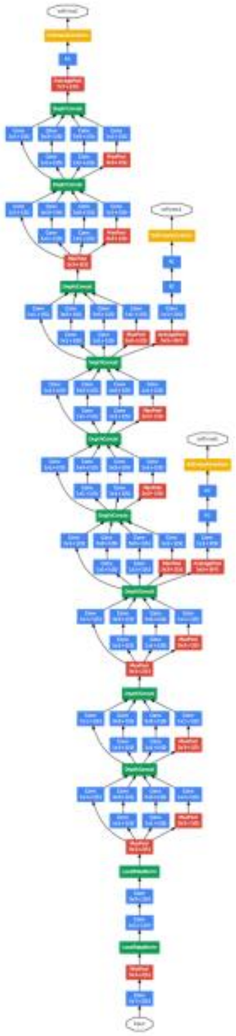
1 nonlinearity applied vs. 3 nonlinearities



c: number of input channels = number of output channels

GoogLeNet

[Szegedy et al. 2014]



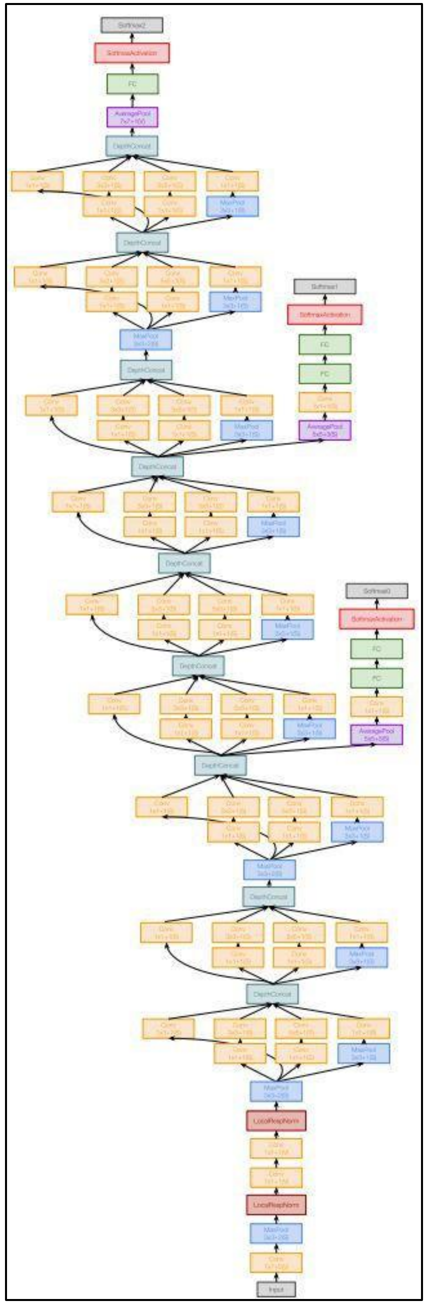
"Going deeper with convolutions"

Inception network

22 layers network
6.7% Top-5 error rate!
Winners of Imagenet 2014

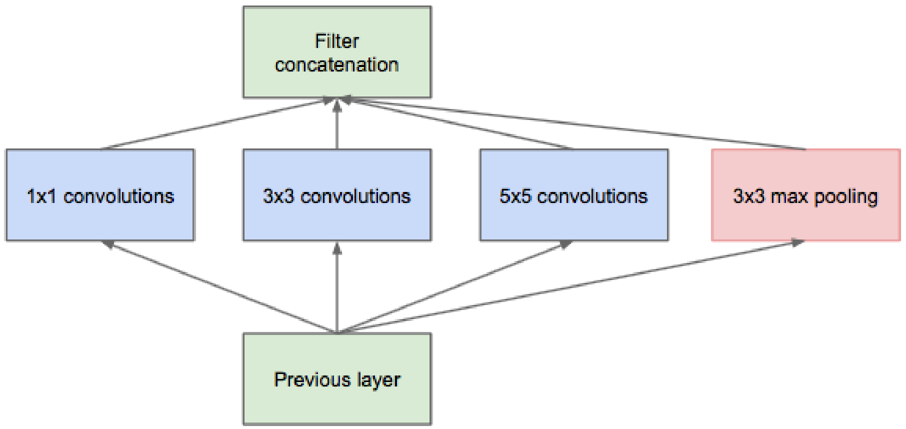
GoogleNet

- 22 layers
- Efficient **“Inception” module** - strayed from the general approach of simply stacking conv and pooling layers on top of each other in a sequential structure
- No FC layers
- Only 5 million parameters!
- ILSVRC’14 classification winner (6.7% top 5 error)

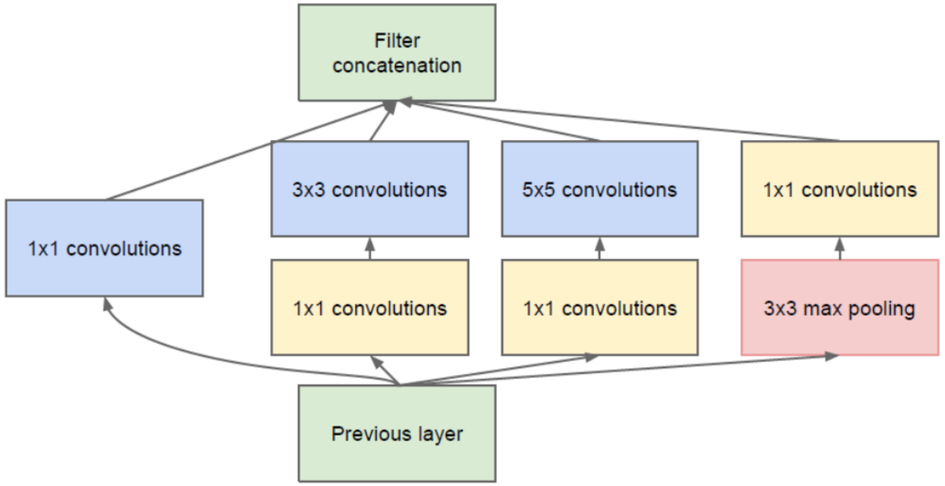


[Szegedy et al., 2014]

GoogLeNet: Inception module



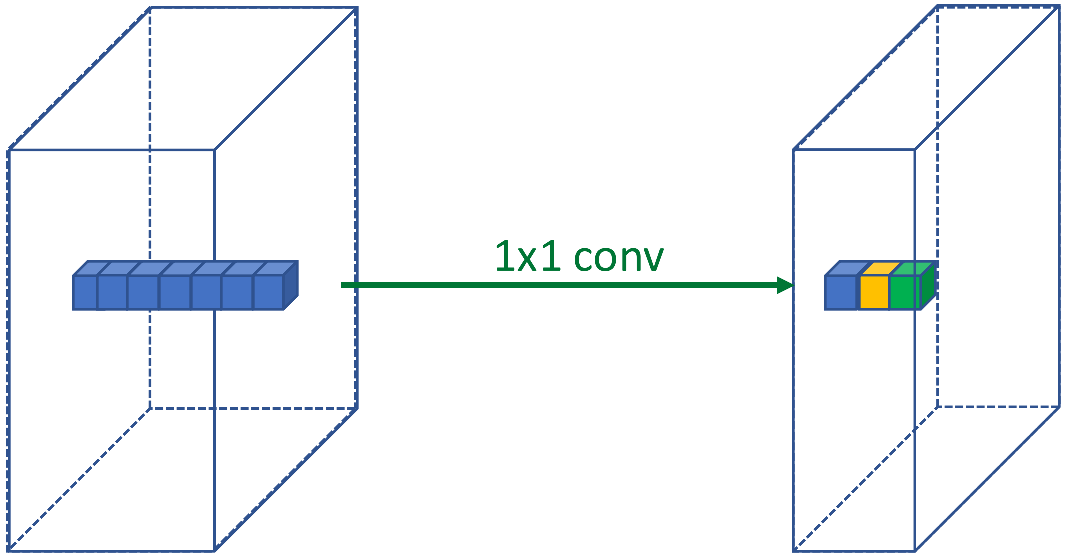
Inception module (naïve version)



Inception module with dimensionality reduction

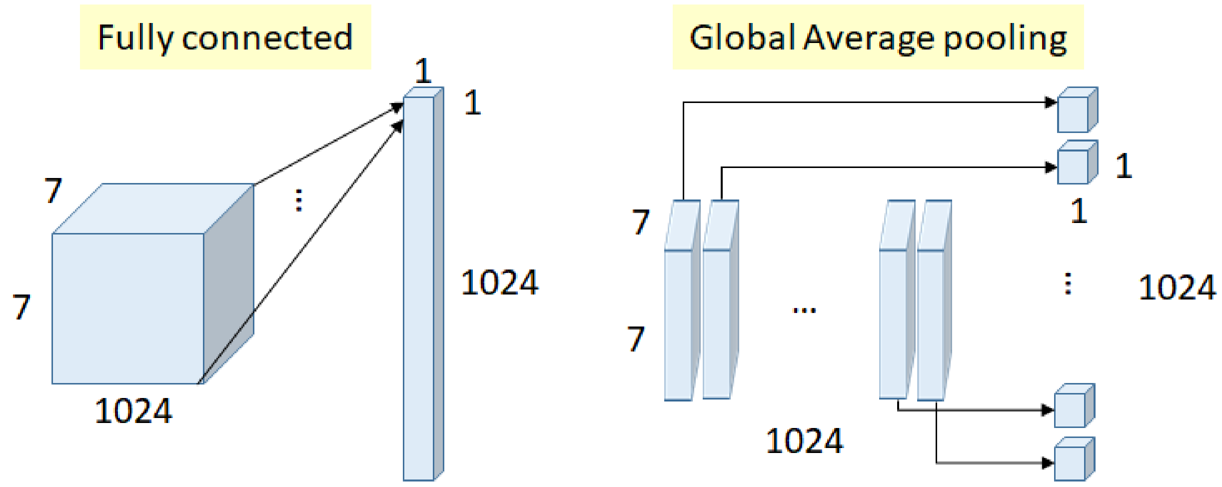
GoogLeNet

- Does it make any sense to do 1x1 convolutions?
- Can we do dimensionality reduction on the depth?



GoogLeNet

- Global average pooling is used at the end of the network instead of using fully connected layers.
- The key contribution is made by average pooling instead of fully connected layers
- Fully connected (FC) layers
    - Number of weights (connections) below = 7×7×1024×1024 = 51.3M
- In GoogLeNet, global average pooling is used nearly at the end of network by averaging each feature map from 7×7 to 1×1, as in the figure below.
    - Number of weights = 0



Conclusion

Conclusion

- Convolutional layer is one of the key of deep neural network efficiency
  - Sharing the parameters reduce the number of parameters
  - Convolution is good to extract patterns
- Other tools like padding and stride are useful to control the size of the layers
- Pooling is great to manage invariance
- It is challenging to manage a large number of layers
  - Many tuned neural network architectures have been proposed
  - Training of very deep neural networks is quite tricky