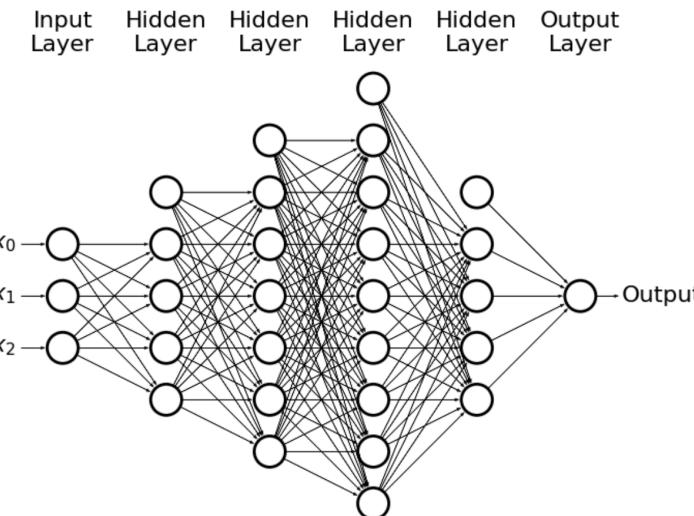


Convolutional neural networks

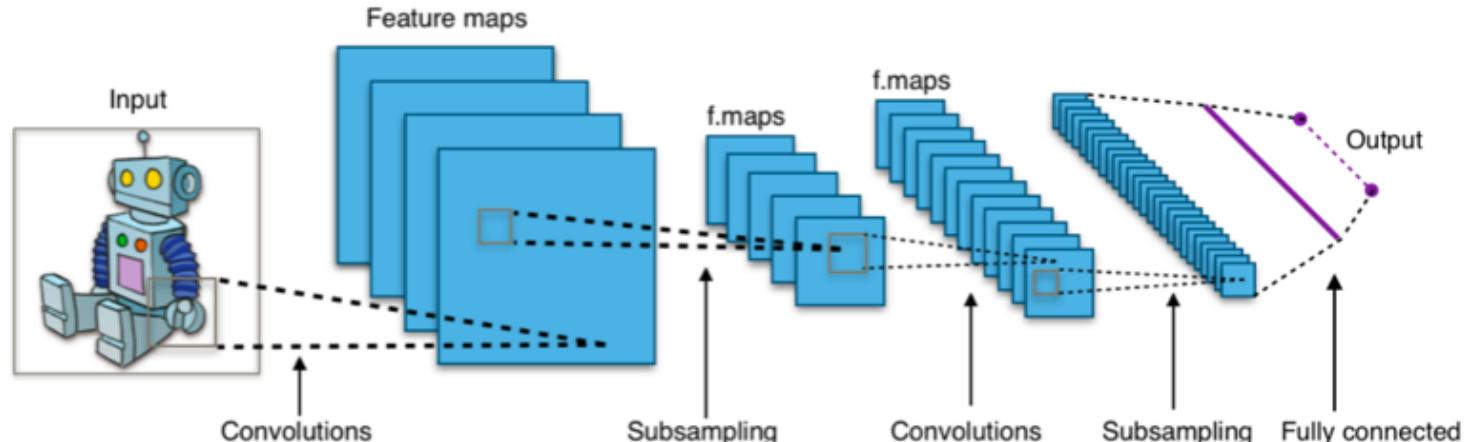
Materials for this section:

- <https://cs231n.github.io/convolutional-networks>
- simple review: <https://arxiv.org/abs/1901.06032>
- <https://arxiv.org/pdf/1603.07285.pdf>

Convolutional neural networks



vs.



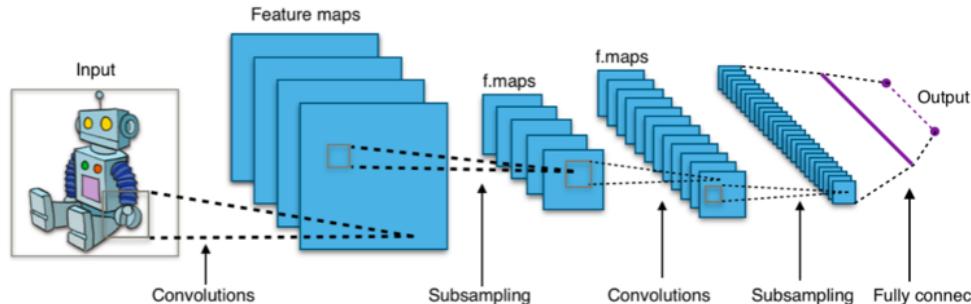
- fully-connected multi-layer perceptron uses (weights) matrix **multiplication** when passing between layers

- convolutional networks use **convolution** operation in place of matrix multiplication in at least one of their layers
- other operation used: pooling (subsampling), dropout (regularization)

Convolutional neural network = ConvNet = CNN

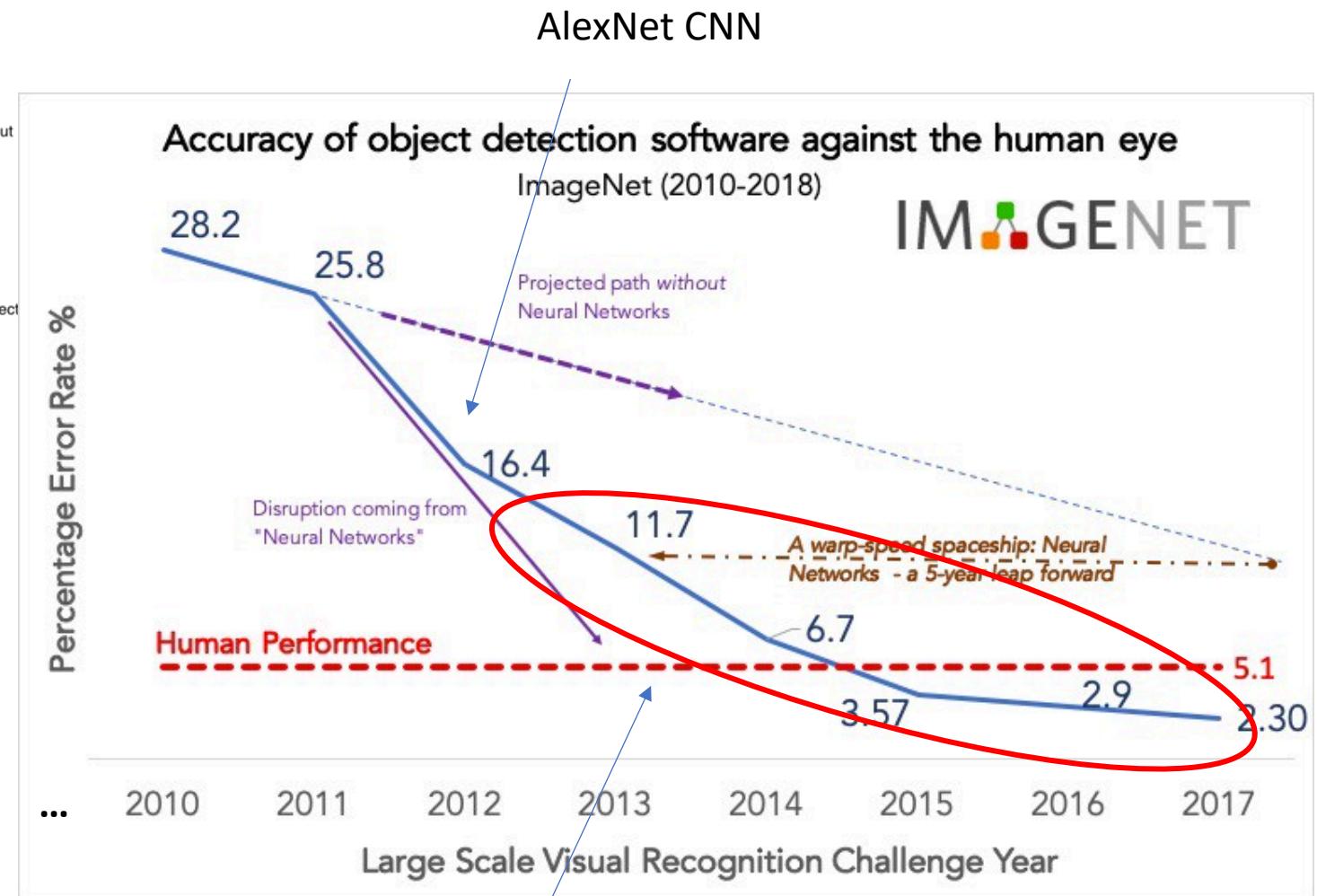
- CNNs start deep-learning revolution (AlexNet, 2012)
- widely used in *computer-vision*
- for processing data that has 1D, 2D, ... grid topology

Convolutional neural networks



- CNNs start deep-learning revolution: ImageNet competition
- are the most significant NN architecture type nowadays, also being a part of much complicated architectures

Yann LeCun (1989) used back-propagation to learn **shared** convolution kernel

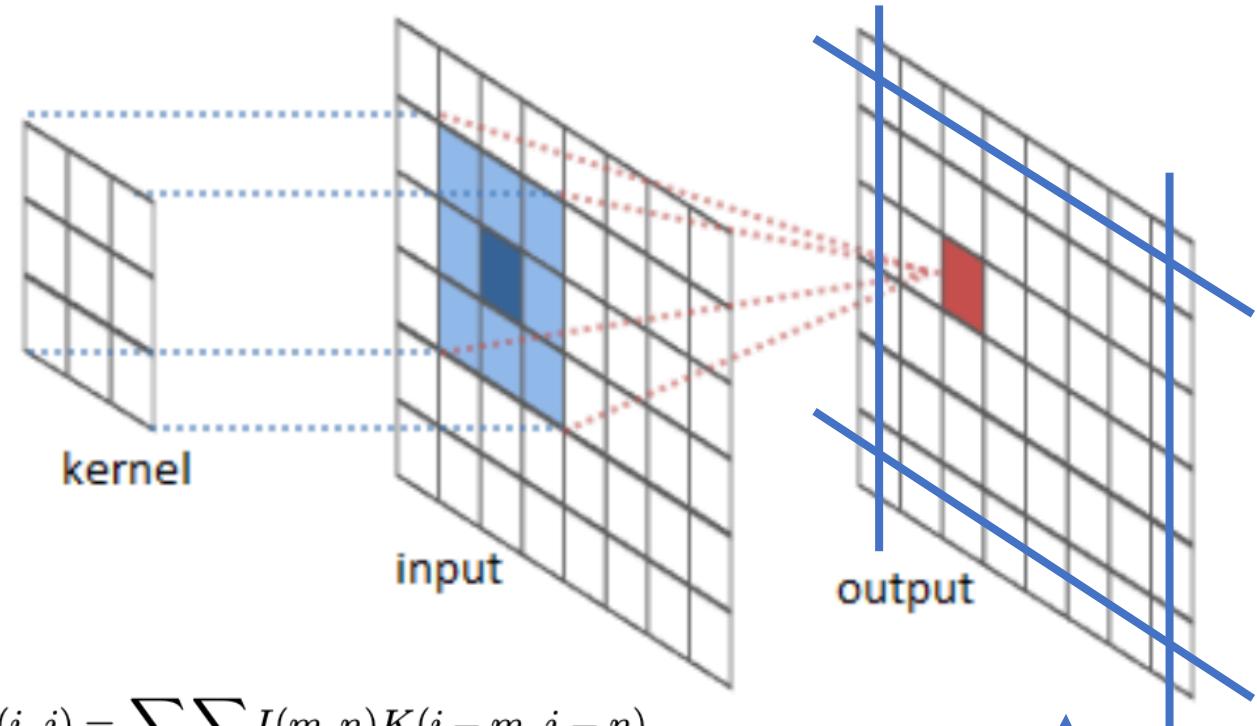
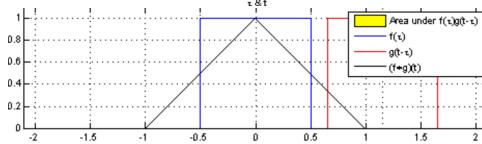
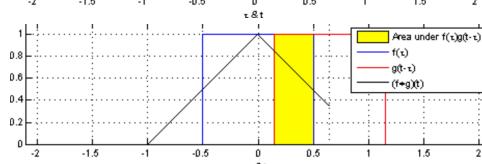
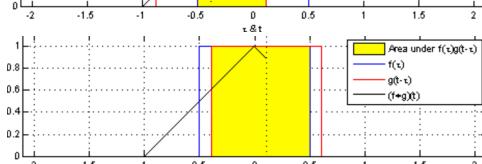
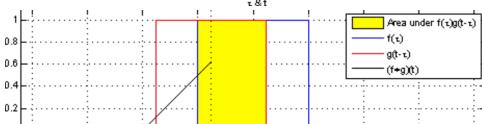
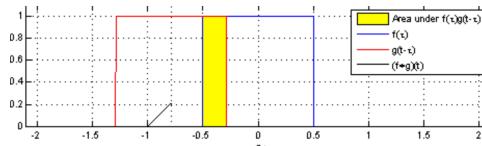


<http://image-net.org/>

Convolutional neural networks

Convolution is an operation where we multiply input by moving weight (kernel) function and obtaining subsequent outputs

$$s(t) = (x * w)(t) = \sum_{a=-\infty}^{\infty} x(a)w(t-a).$$



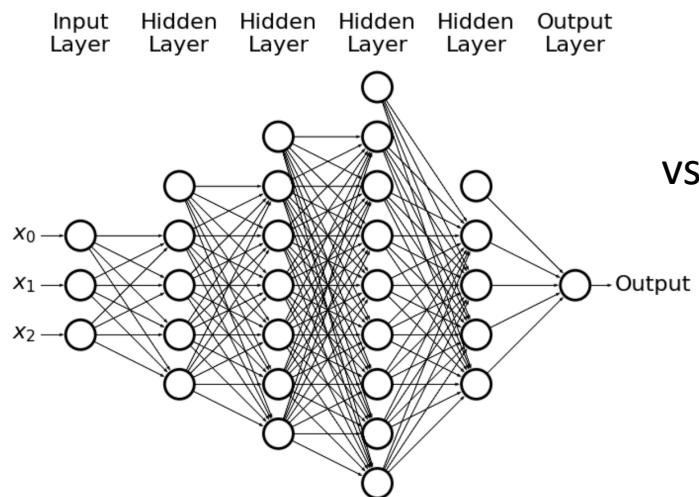
$$S(i, j) = (I * K)(i, j) = \sum_m \sum_n I(m, n)K(i - m, j - n).$$

- To get 1d or 2d **discrete** convolutions we move kernel over input grid, e.g. image,
- kernel is usually much smaller, e.g. 3x3, than input image, e.g. 512x512 pixels,
- and obtain reduced size output (510x510 in our example).

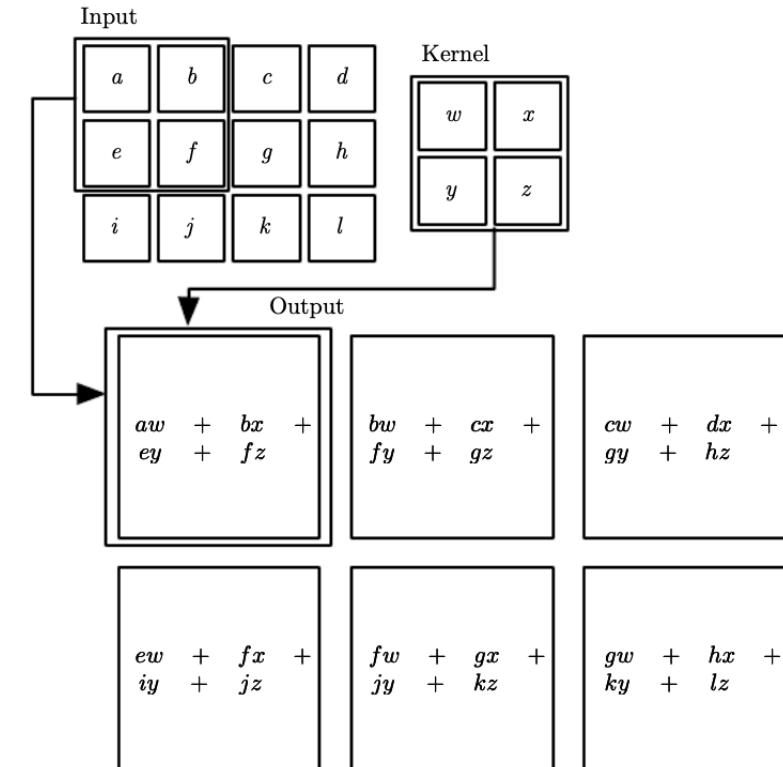
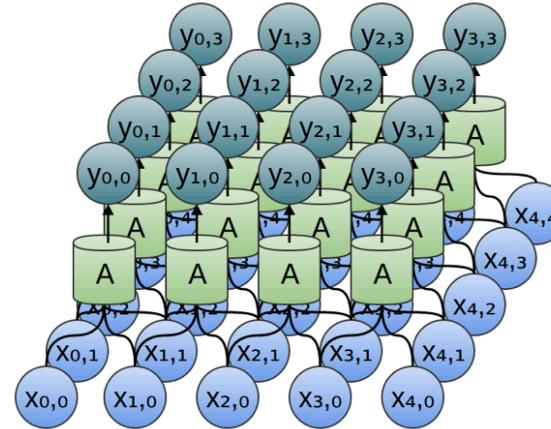
Convolutional neural networks

Idea why to use convolutions: Sparse-connectivity

- CNNs are regularized version of fully-connected MLPs
- A single element in the feature map is connected to only a small patch of pixels.
- This is very different from connecting to the whole input image, in the case of multi-layer perceptrons.



VS.



- CNNs are also much faster to train than fully MLPs.

Convolutional neural networks

- Please note, that the output size ($W_{out} \times H_{out}$) is determined by the input size ($W_{in} \times H_{in}$) and the kernel size ($W_f \times H_f$):

$$W_{out} = W_{in} - W_f + 1$$

$$H_{out} = H_{in} - H_f + 1$$

Padding

- Padding (zero padding) - adding zeros around an image

0 1 0 1 1	0 0 0 0 0 0 0
1 1 0 1 0	0 0 1 0 1 1 0
1 0 1 0 1 ->	0 1 1 0 1 0 0
1 1 0 1 0	0 1 0 1 0 1 0
0 0 1 1 0	0 1 1 0 1 0 0
0 0 0 0 0 0 0	0 0 0 1 1 0 0

- The output size ($W_{out} \times H_{out}$) is determined by the input size ($W_{in} \times H_{in}$), the kernel size ($W_f \times H_f$), and the padding size (P)

\$

$$W_{out} = W_{in} - W_f + 2P + 1$$

$$H_{out} = H_{in} - H_f + 2P + 1$$

Sometimes downsizing is unwanted, and information on the edges is of the same importance

Stride

- Stride controls the movement of the filter

Stride = 1

```
| [0 1 0] 1 1 |    | 0 [1 0 1] 1 |
| [1 1 0] 1 0 |    | 1 [1 0 1] 0 |
| [1 0 1] 0 1 | -> | 1 [0 1 0] 1 | -> ...
| 1 1 0 1 0 |    | 1 1 0 1 0 |
| 0 0 1 1 0 |    | 0 0 1 1 0 |
```

Stride = 2

```
| [0 1 0] 1 1 |    | 0 1 [0 1 1] |
| [1 1 0] 1 0 |    | 1 1 [0 1 0] |
| [1 0 1] 0 1 | -> | 1 0 [1 0 1] | -> ...
| 1 1 0 1 0 |    | 1 1 0 1 0 |
| 0 0 1 1 0 |    | 0 0 1 1 0 |
```

- The output size ($W_{out} \times H_{out}$) is determined by the input size ($W_{in} \times H_{in}$), the kernel size ($W_f \times H_f$), the padding size (P), and the stride size ($W_s \times H_s$)

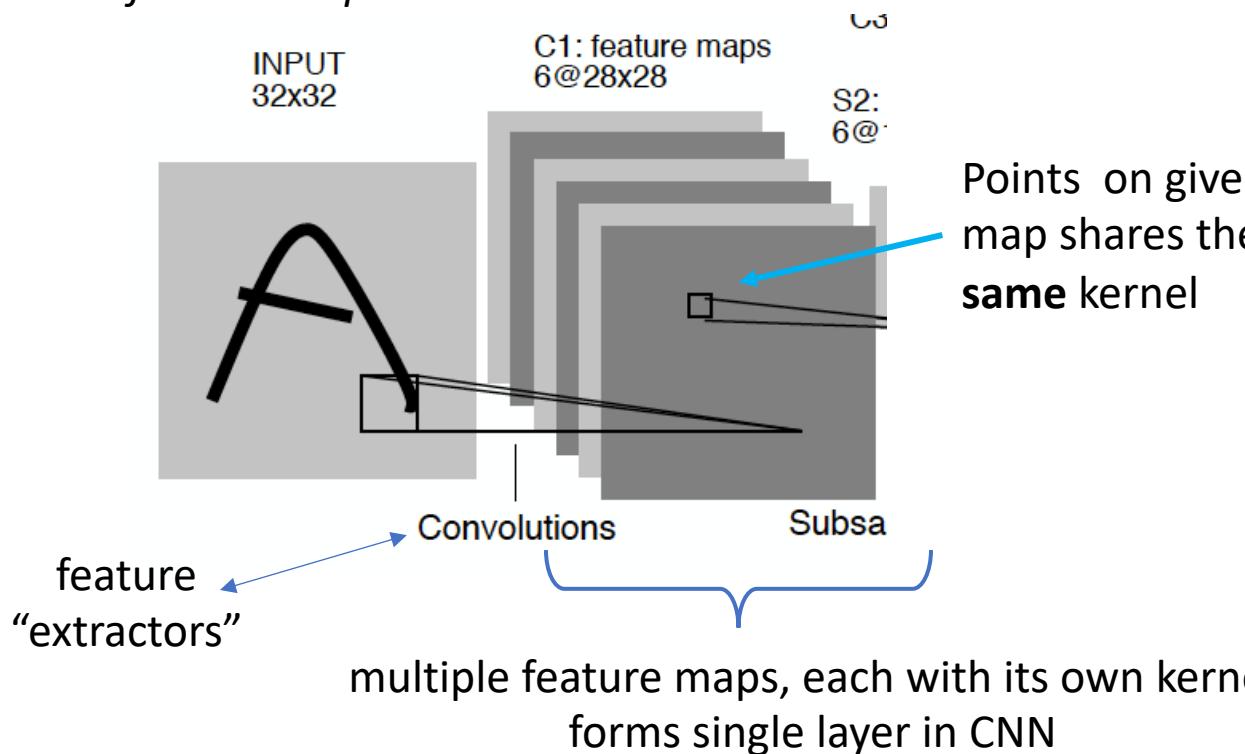
$$W_{out} = \frac{W_{in} - W_f + 2P}{W_s} + 1$$

$$H_{out} = \frac{H_{in} - H_f + 2P}{H_s} + 1$$

CNNs: feature maps

The idea is to build hierarchy of concepts using multiple layers each consisting feature maps:

- we need to extract **features** from images, time series, etc.
- each kernel extract different feature, producing so-called *feature map*



1	0	-1
2	0	-2
1	0	-1

Sobel filter

3	0	-3
10	0	-10
3	0	-3

Scharr filter

1	0	-1
1	0	-1
1	0	-1

Vertical

1	1	1
0	0	0
-1	-1	-1

Horizontal

example kernels

*

1	0	-1
1	0	-1
1	0	-1

=

-0	30	30	0
0	30	30	0
0	30	30	0
0	30	30	0

3 x 3

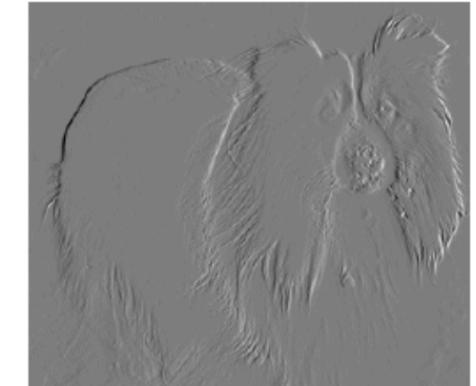
4 x 4

10	10	10	0	0	0
10	10	10	0	0	0
10	10	10	0	0	0
10	10	10	0	0	0
10	10	10	0	0	0

6 x 6



e.g. vertical edge detection kernel



CNNs: learning kernels

The coolest thing is that network learns kernel types automatically by itself!

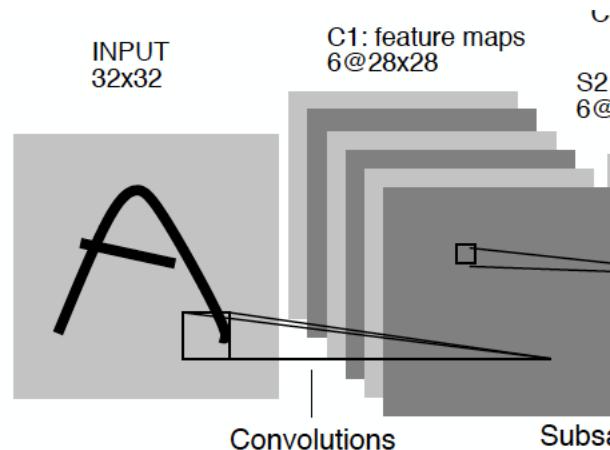
How to choose weights in the filter?

3	0	1	2	7	4
1	5	8	9	3	1
2	7	2	5	1	3
0	1	3	1	7	8
4	2	1	6	2	8
2	4	5	2	3	9

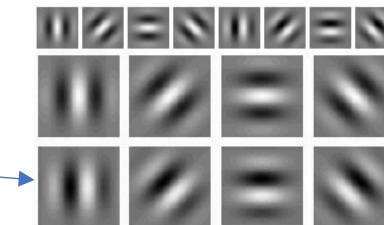
6 × 6

$$\begin{array}{c} \text{convolution} \\ \downarrow \\ * \\ \begin{array}{|c|c|c|}\hline W_1 & W_1 & W_1 \\ \hline W_1 & W_1 & W_1 \\ \hline W_1 & W_1 & W_1 \\ \hline \end{array} \end{array} = \begin{array}{|c|c|c|}\hline 0 & 0 & 0 & 0 \\ \hline 30 & 10 & -10 & -30 \\ \hline 30 & 10 & -10 & -30 \\ \hline 0 & 0 & 0 & 0 \\ \hline \end{array}$$

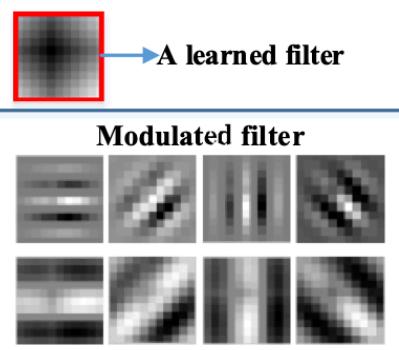
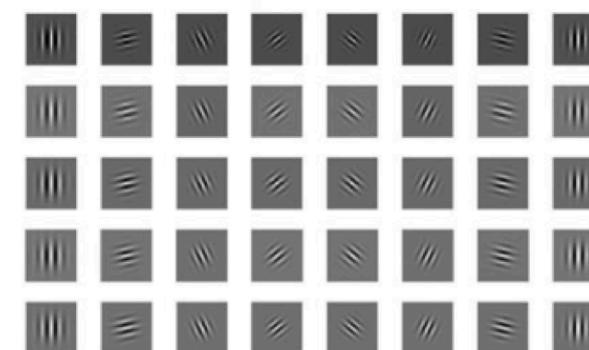
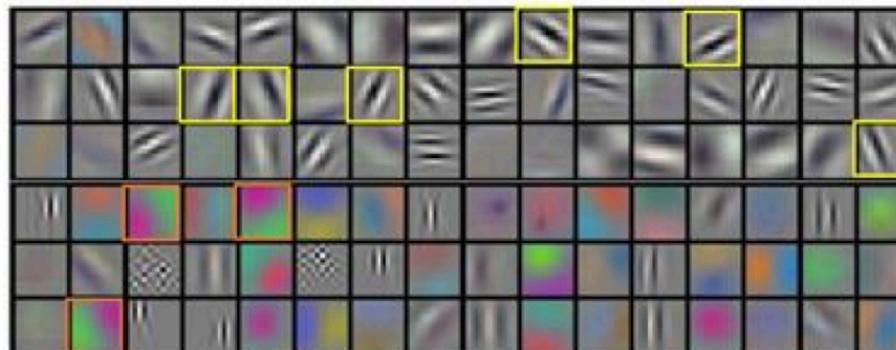
3 × 3 4 × 4



CNNs: Feature extractors using kernels learned by itself



AlexNet learnt by itself filters that resemble *Gabor filters* known from classical image processing,
<https://arxiv.org/pdf/1705.01450.pdf>



Deep Convolutional neural networks

stanford.edu/~shervine/teaching/cs-230/cheatsheet-convolutional-neural-networks

□ **Dimensions of a filter** — A filter of size $F \times F$ applied to an input containing C channels is a $F \times F \times C$ volume that performs convolutions on an input of size $I \times I \times C$ and produces an output feature map (also called activation map) of size $O \times O \times 1$.

Convolutions with Color Channels

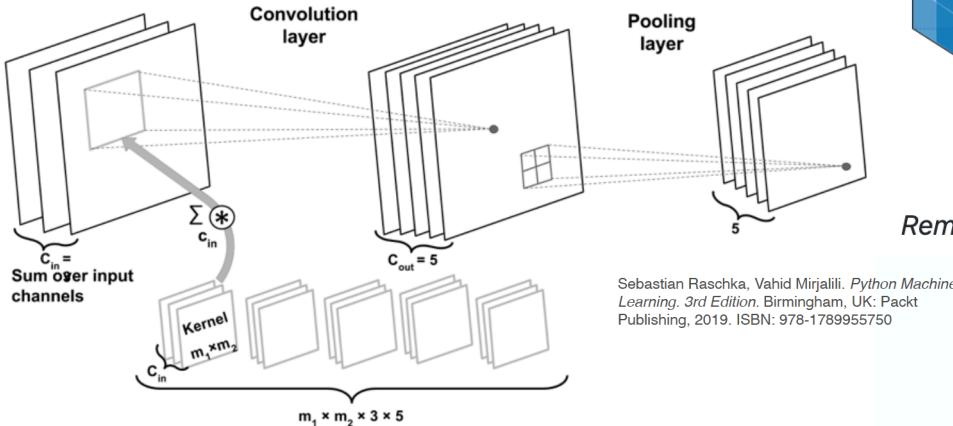
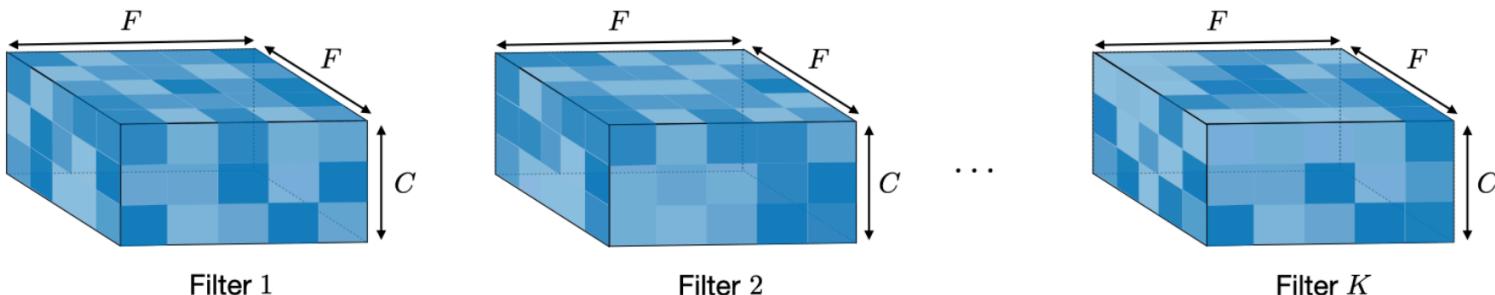


Image dimension: $\mathbf{X} \in \mathbb{R}^{n_1 \times n_2 \times c_{in}}$ in NWHC format,
CUDA & PyTorch use NCWH

In cases where 2-d data have multiple channels
kernel might be 3-dimensional



Remark: the application of K filters of size $F \times F$ results in an output feature map of size $O \times O \times K$.

Summary

- Convolutional layer consist of N filters, so the hyperparameters are:
 - the number of filters
 - the size of the filters
 - padding
 - stride

CNN full architecture

Let's go deeper into full CNN

Hidden Layers

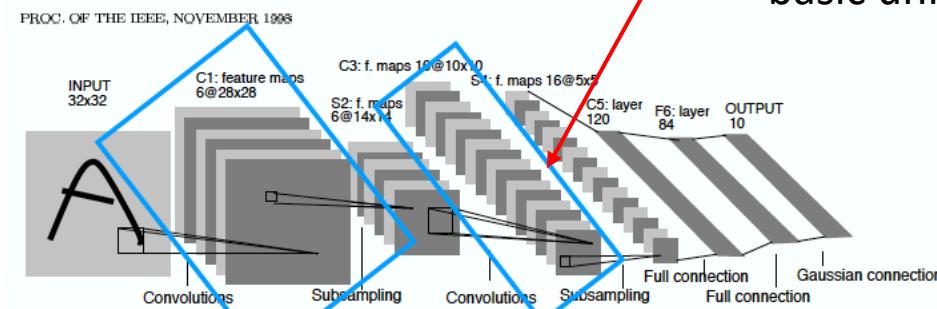
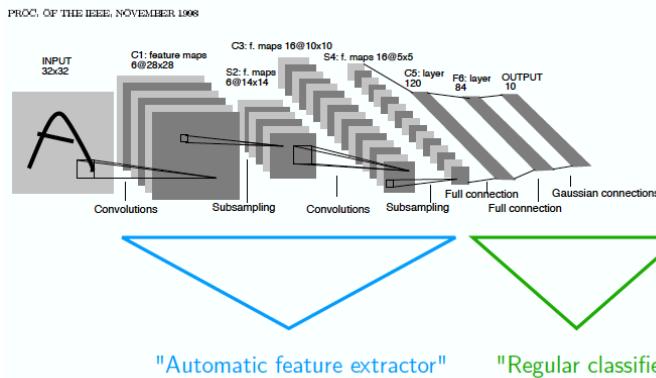
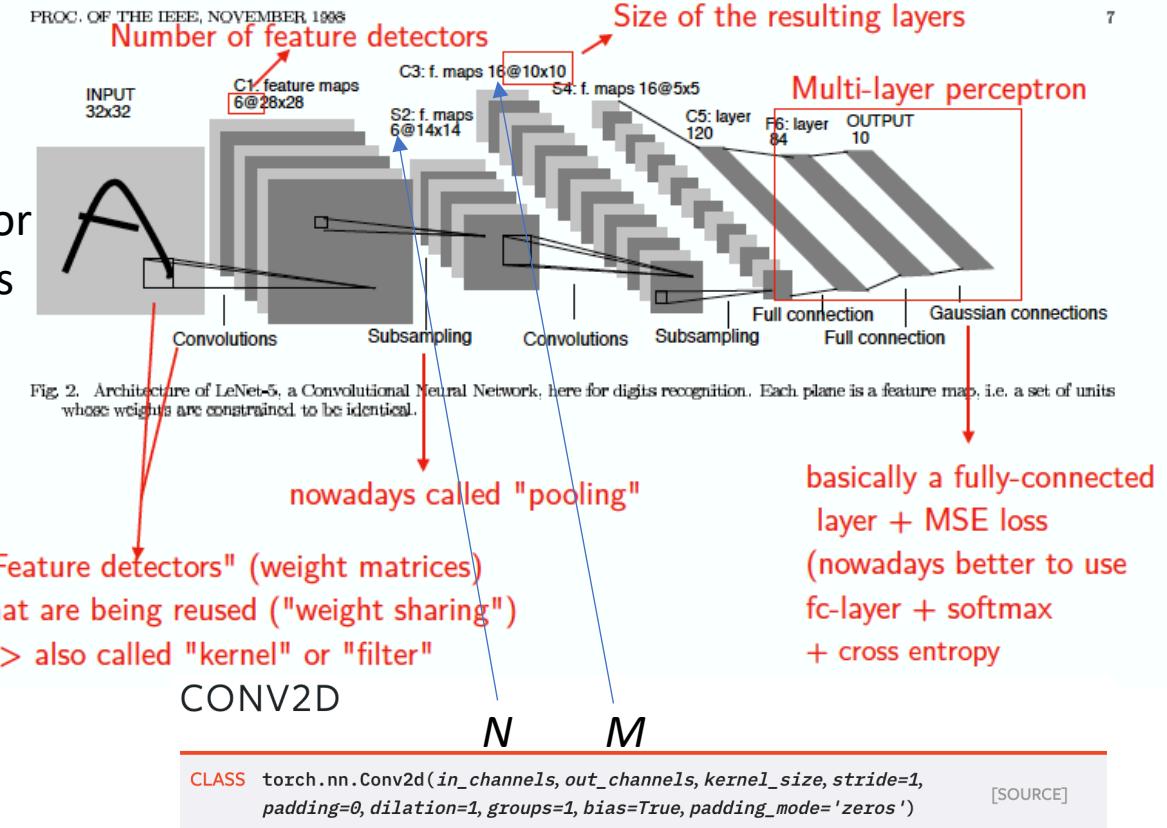


Fig. 2. Architecture of LeNet-5, a Convolutional Neural Network, here for digits recognition. Each plane is a feature map, i.e. a set of units whose weights are constrained to be identical.

Each "bunch" of feature maps represents one hidden layer in the neural network.

Convolutional Neural Networks



Applies a 2D convolution over an input signal composed of several input planes.

In the simplest case, the output value of the layer with input size (N, C_{in}, H, W) and output $(N, C_{out}, H_{out}, W_{out})$ can be precisely described as:

$$\text{out}(N_i, C_{out,j}) = \text{bias}(C_{out,j}) + \sum_{k=0}^{C_{in}-1} \text{weight}(C_{out,j}, k) * \text{input}(N_i, k)$$

CNN components: pooling

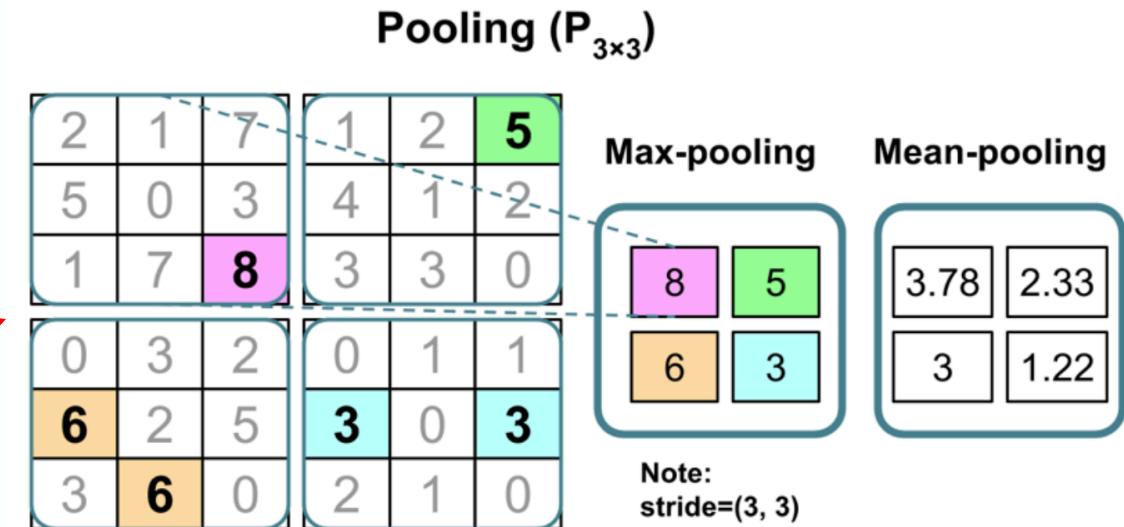
Pooling operation

- A pooling function replaces the given output activation map with map composed of **local** statistic of the nearby values
- Pooling helps to make the representation approximately **invariant** to small translations of the input.

e.g. here pool size is of 3x3 and stride between pools equals 3

The pooling layer reduces the number of parameters and calculations in the network. This improves the efficiency of the network and avoids overfitting.

Pooling Layers Can Help With Local Invariance



Note that typically pooling layers do not have any learnable parameters

CNN components: dropout and batch-normalization

Another type of regularizations used in CNNs is **dropout**:

- frequently we apply dropout to fully-connected-layers part
- but sometimes also after pooling layers

Moreover to stabilize and speed-up the training process we apply **batch-normalization** layers:

- the idea is to make activations units (in the given layer) gaussian-like by scaling their input x_i (from earlier layer)
- normalization is performed using averages for entire mini-batch: mini-batch mean and mini-batch variance (N is a batch size):

The mean and the variance are calculated:
• for given mini-batch (when training)
• for the whole training dataset (inference)



$$\mu = \frac{1}{N} \sum_{i=1}^N x_i \quad \sigma^2 = \frac{1}{N} \sum_{i=1}^N (x_i - \mu)^2$$

- and normalize inputs

$$\hat{x}_i = \frac{x_i - \mu}{\sqrt{\sigma^2 + \epsilon}}$$

each input x_i contains
• activations (MLP)
• or complete set of feature maps (conv layers)
from previous layer

Note1: SGD (training) will still work since \hat{x}_i is a differentiable function

Note2: when training CNNs we process whole mini-batch at once

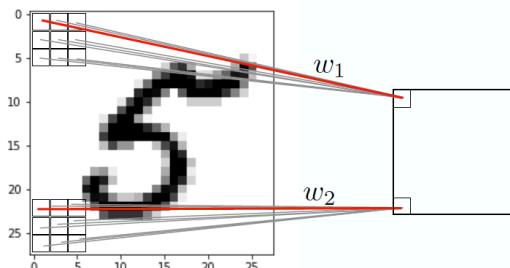
(input tensor is of size $N \times C \times W \times H$, for batch of N color images each of $W \times H$ size)

CNNs: training and activations

Backpropagation in CNNs

Same overall concept as before: Multivariable chain rule, but now with an additional weight sharing constraint

Due to weight sharing: $w_1 = w_2$



weight update:

$$w_1 := w_2 := w_1 - \eta \cdot \frac{1}{2} \left(\frac{\partial \mathcal{L}}{\partial w_1} + \frac{\partial \mathcal{L}}{\partial w_2} \right)$$

Parameter-sharing: the same weights are used for different patches of the input image.

Commonly used activation functions

□ **Rectified Linear Unit** — The rectified linear unit layer (ReLU) is an activation function g that is used on all elements of the volume. It aims at introducing non-linearities to the network. Its variants are summarized in the table below:

ReLU	Leaky ReLU	ELU
$g(z) = \max(0, z)$	$g(z) = \max(\epsilon z, z)$ with $\epsilon \ll 1$	$g(z) = \max(\alpha(e^z - 1), z)$ with $\alpha \ll 1$

• Non-linearity complexities biologically interpretable

• Addresses dying ReLU issue for negative values

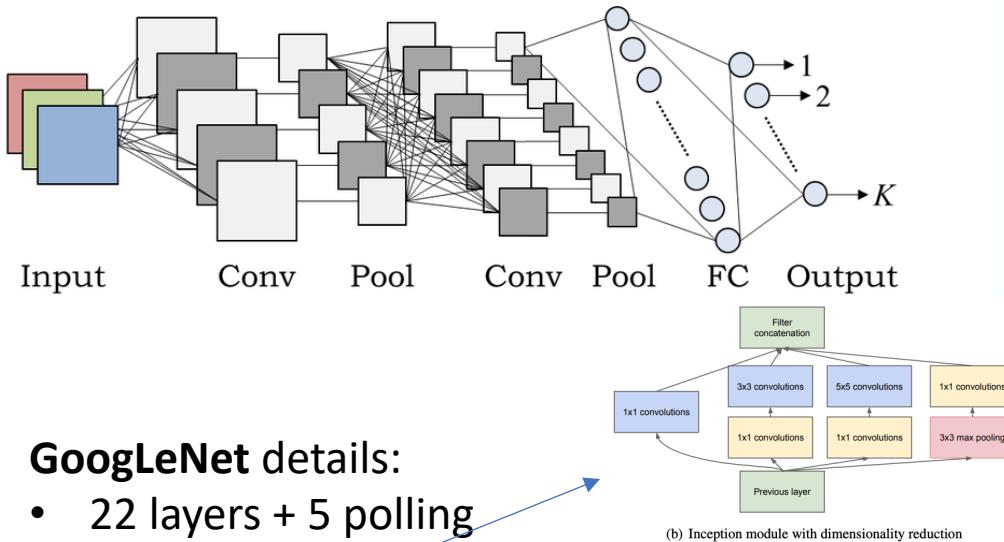
• Differentiable everywhere

Vanishing gradient problem – solutions:

- ReLUs activations
- batch normalization
- *residual networks*

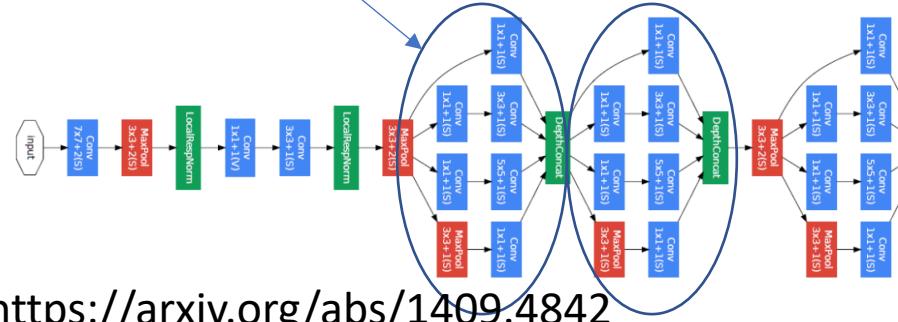
CNNs: common architectures

Typical architecture but in real tasks we may go *deeper*



GoogLeNet details:

- 22 layers + 5 polling
- 7M parameters
- *inception layers* (network in a network)



Main Breakthrough for CNNs: AlexNet & ImageNet

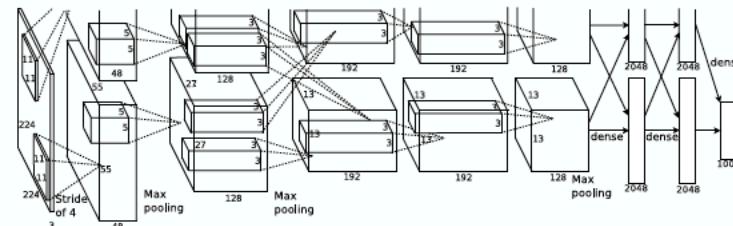
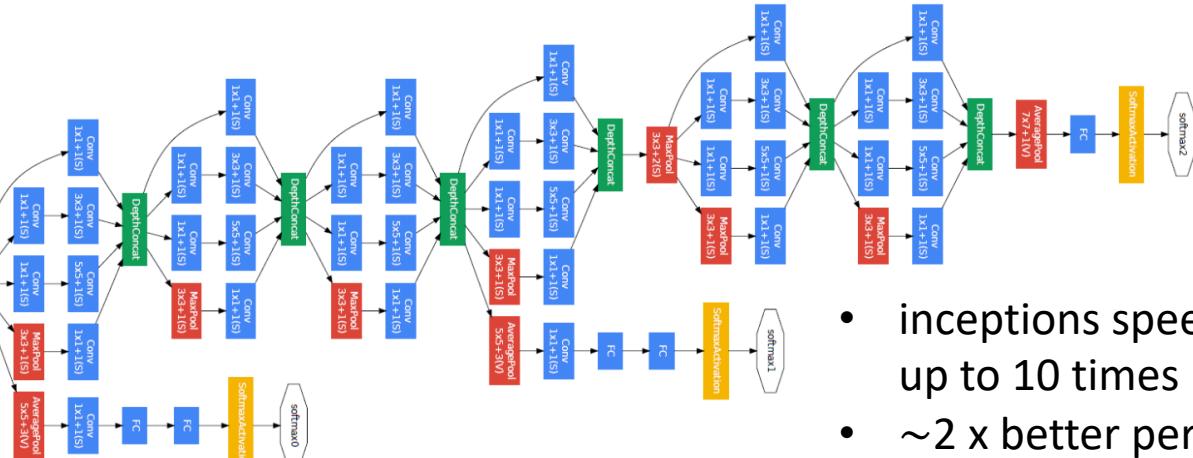


Figure 2: An illustration of the architecture of our CNN, explicitly showing the delineation of responsibilities between the two GPUs. One GPU runs the layer-parts at the top of the figure while the other runs the layer-parts at the bottom. The GPUs communicate only at certain layers. The network's input is 150,528-dimensional, and the number of neurons in the network's remaining layers is given by 253,440–186,624–64,896–64,896–43,264–4096–4096–1000.

Zehevsky, A., Sutskever, I., & Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems* (pp. 1097–1105).



AlexNet details:

parameters:

- 3.7M conv
- 58.6M fully-conn

forward computations:

- 1.08G conv
- 58.6M f-c

CNNs: common architectures

ImageNet
accuracy

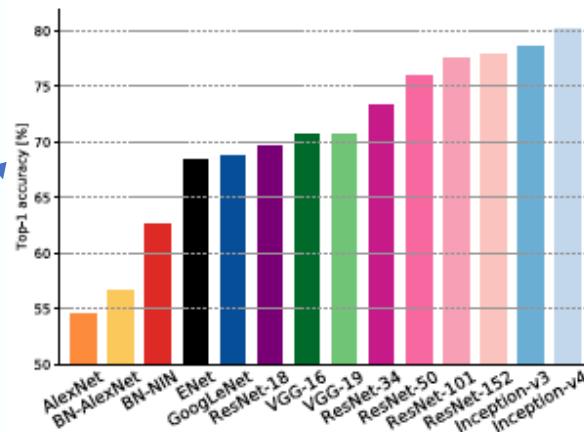


Figure 1: **Top1 vs. network.** Single-crop top-1 validation accuracies for top scoring single-model architectures. We introduce with this chart our choice of colour scheme, which will be used throughout this publication to distinguish effectively different architectures and their correspondent authors. Notice that networks of the same group share the same hue, for example ResNet are all variations of pink.

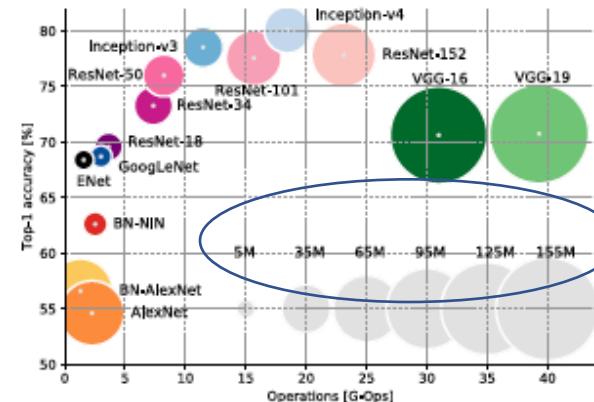
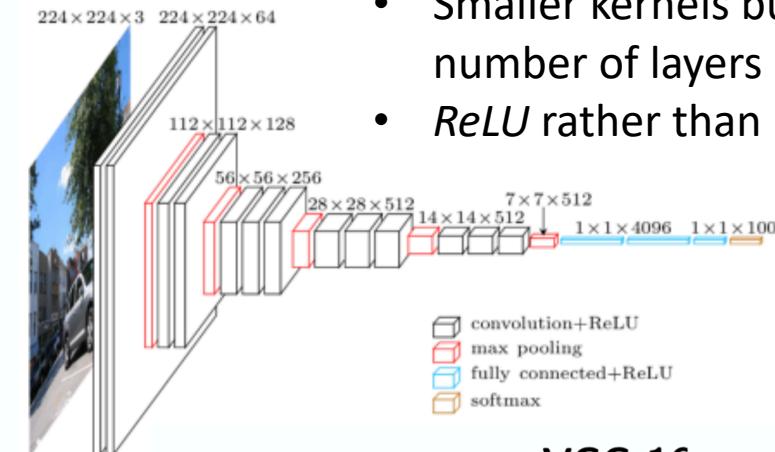


Figure 2: **Top1 vs. operations, size \propto parameters.** Top-1 one-crop accuracy versus amount of operations required for a single forward pass. The size of the blobs is proportional to the number of network parameters; a legend is reported in the bottom right corner, spanning from 5×10^6 to 155×10^6 params. Both these figures share the same y-axis, and the grey dots highlight the centre of the blobs.

VGGNet



Differences from AlexNet

- Smaller kernels but with larger number of layers
- *ReLU* rather than *tanh* activation

VGG-16

PyTorch implementation: <https://github.com/rasbt/stat453-deep-learning-ss20/blob/master/L13-cnns-part2/code/vgg16.ipynb>

ConvNet Configuration				
A	B	C	D	E
11 weight layers	11 weight layers	13 weight layers	16 weight layers	19 weight layers
conv3-64	conv3-64	conv3-64	conv3-64	conv3-64
LRN	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
	maxpool			
conv3-256	conv3-256	conv3-256	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	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	conv3-512	conv3-512
	conv3-512	conv3-512	conv3-512	conv3-512
	maxpool			
FC-4096				
FC-4096				
FC-1000				
soft-max				

Advantages:

very simple architecture,
 3×3 convs, stride=1,
"same" padding, 2×2 max pooling

Disadvantage:

very large number of parameters
and slow
(see previous slide)

Very deep CNNs: degradation problem

Is learning better networks as easy as stacking more layers?

-- very deep CNNs, problems:

- Vanishing/exploding gradients (solutions: ReLU, batch normalization)
- **degradation** problem

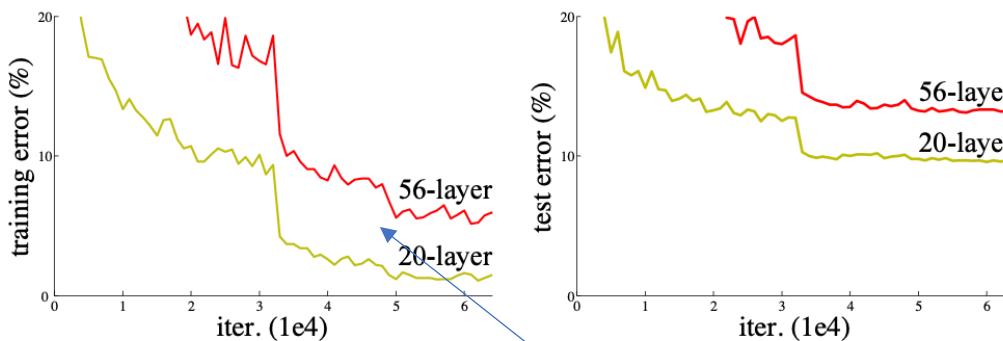


Figure 1. Training error (left) and test error (right) on CIFAR-10 with 20-layer and 56-layer “plain” networks. The deeper network has higher training error, and thus test error. Similar phenomena on ImageNet is presented in Fig. 4.

When deeper networks are able to start converging, a *degradation* problem has been exposed: with the network depth increasing, accuracy gets saturated (which might be unsurprising) and then degrades rapidly. Unexpectedly, such degradation is *not caused by overfitting*, and adding more layers to a suitably deep model leads to *higher training error*, as reported in [11, 42] and thoroughly verified by our experiments. Fig. 1 shows a typical example.

<https://arxiv.org/pdf/1512.03385.pdf>

deeper network has higher **training** error
(this is not just due to overfitting)

Residual networks

Solution: **residual networks** – add identity shortcut to enable skipping, while existing layers are able to fit resting (*residual*) mapping

With their simple trick of allowing skip connections (the possibility to learn identity functions and skip layers that are not useful), ResNets allow us to implement very, very deep architectures

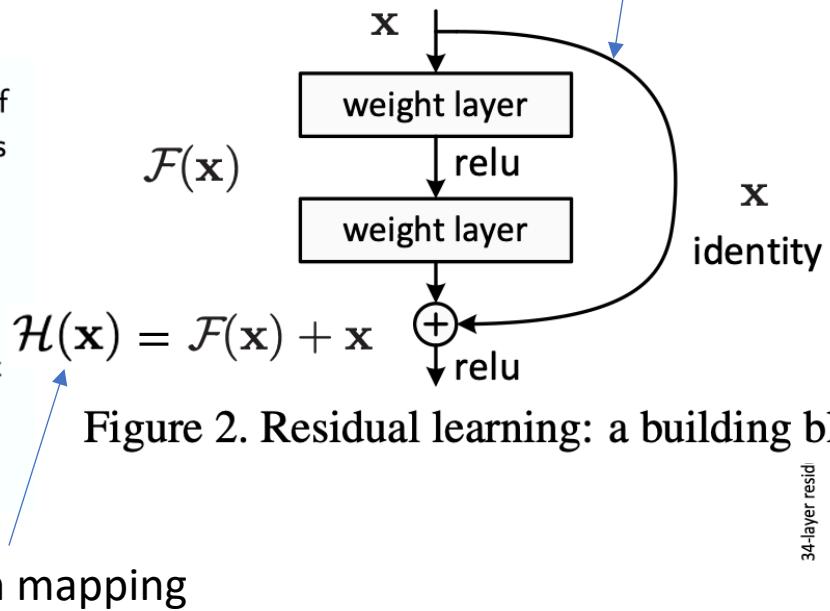
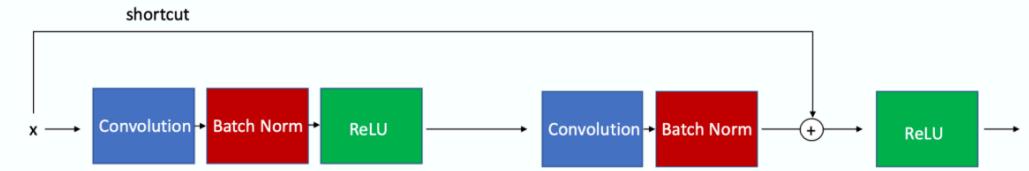
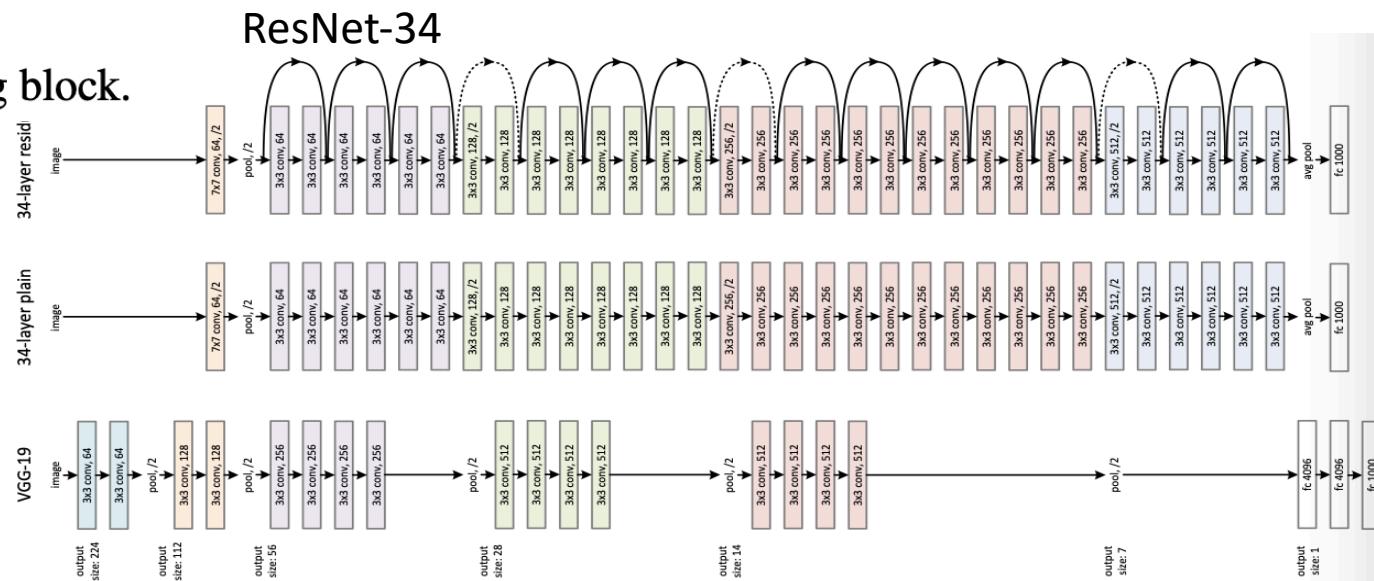


Figure 2. Residual learning: a building block.



$$\text{In general: } a^{(l+2)} = \sigma(z^{(l+2)} + a^{(l)})$$



Residual networks

Results for various types (depths) of ResNets

- degradation problem disappears
- much faster training

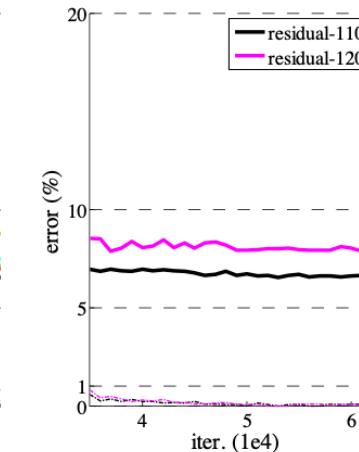
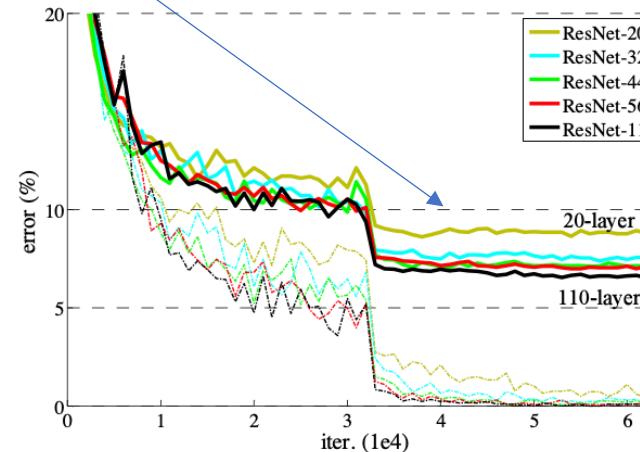
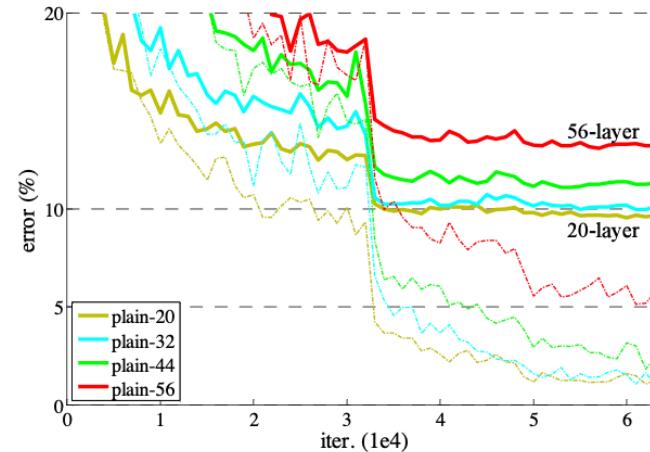
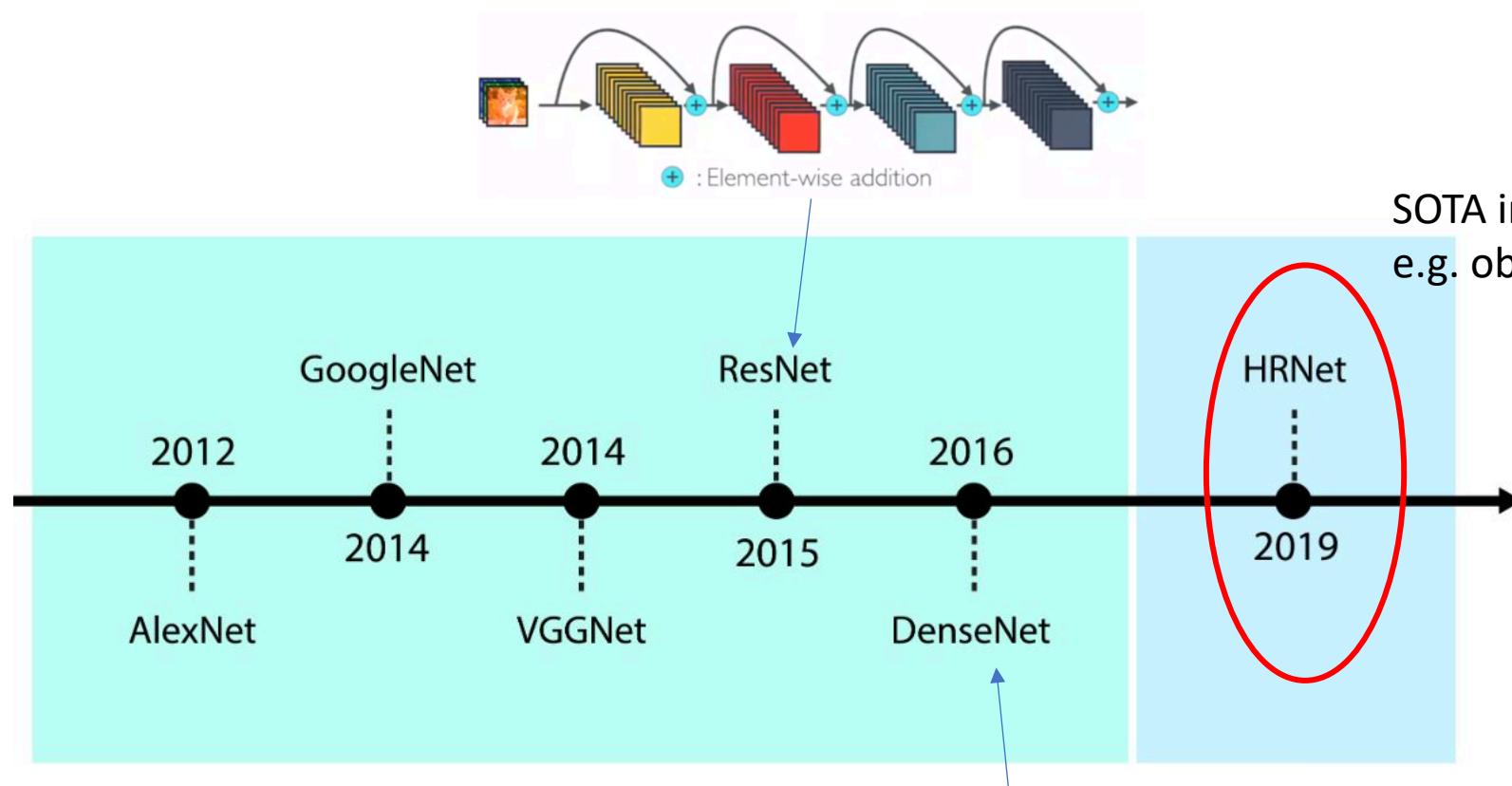


Figure 6. Training on **CIFAR-10**. Dashed lines denote training error, and bold lines denote testing error. **Left:** plain networks. The error of plain-110 is higher than 60% and not displayed. **Middle:** ResNets. **Right:** ResNets with 110 and 1202 layers.

CNNs: milestone architectures



SOTA in computer vision,
e.g. objects recognition

2019

<https://www.microsoft.com/en-us/research/blog/high-resolution-network-a-universal-neural-architecture-for-visual-recognition/>

High-resolution representations

High resolution is essential for tasks beyond classification

- object *detection* on images
- *Segmentation* of detected objects require **spatially fine** representations



Figure 2. Mask R-CNN results on the COCO test set. These results are based on ResNet-101 [15], achieving a mAP of 33.9% running at 5 fps. Masks are shown in color, and bounding box, category, and confidences are also shown.

Redmon, J., Divvala, S., Girshick, R., & Farhadi, A. (2016). You only look once: Unified, real-time object detection. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (pp. 779-788).

He, Kaiming, Georgia Gkioxari, Piotr Dollar, and Ross Girshick. "Mask R-CNN." In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 2961-2969. 2017.

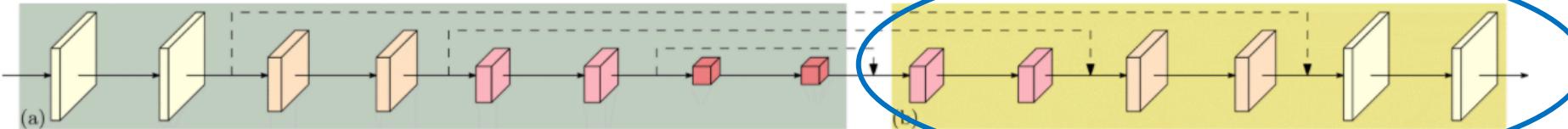
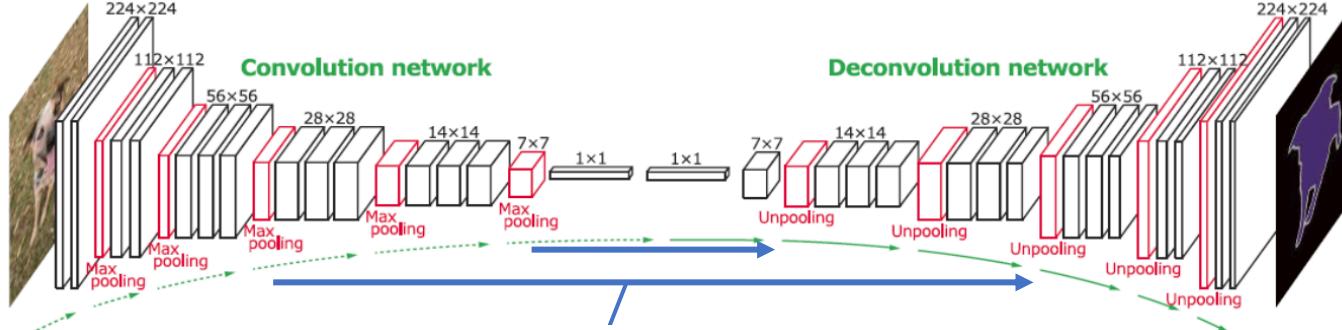


Figure 2: The structure of recovering high resolution from low resolution. (a) A low-resolution representation learning subnetwork (such as AlexNet, GoogleNet, VGGNet, ResNet, DenseNet), which is formed by connecting high-to-low convolutions in series. (b) A high-resolution representation recovering subnetwork, which is formed by connecting low-to-high convolutions in series. Representative examples include SegNet, DeconvNet, U-Net and Hourglass, encoder-decoder, and SimpleBaseline.

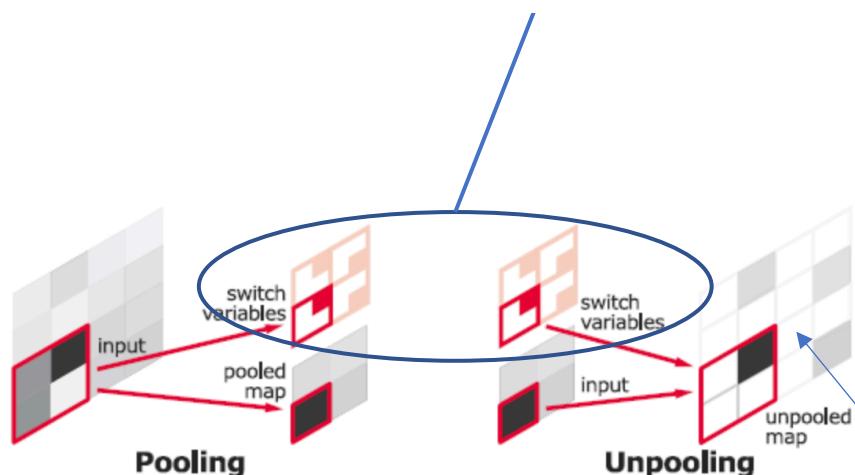
Hourglass structure:
added subnetwork
recovering high-resolution representation

High-resolution representations: deconvolution layers

<https://arxiv.org/abs/1505.04366>



the locations of maximum activations
selected during pooling operation



Remember positions when Pooling (Left), Reuse the position information during Unpooling (right)

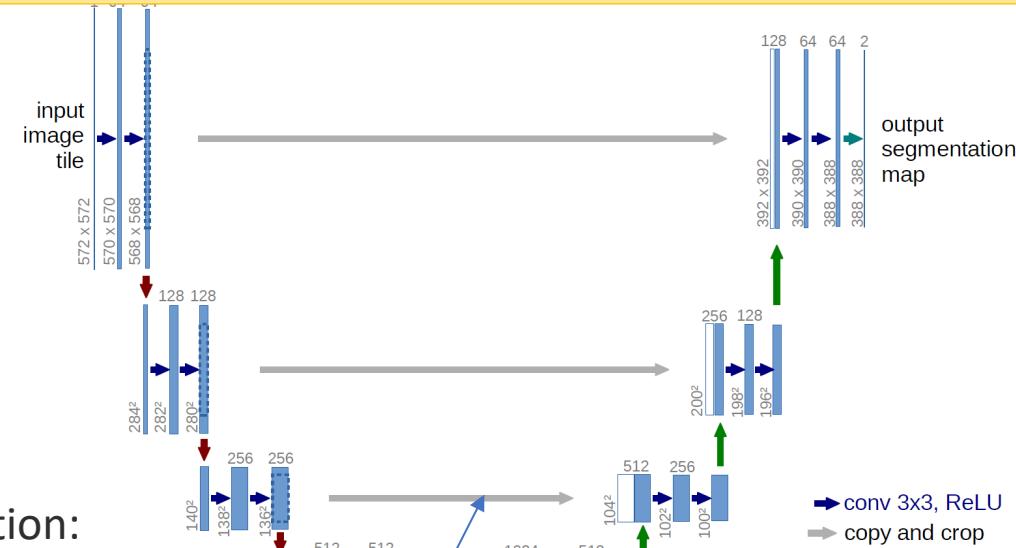
sparse map, subsequent
convolutions densify it

We restore high resolution:

- *unpooling layers (DeconvNet)*
- *upsampling layers (U-Net)*

Upsampling:

1. *transposed convolutions*
2. *concatenation with respective feature maps from “down” part*



https://github.com/vdumoulin/conv_arithmetic

Transposed convolution

4.1 Convolution as a matrix operation

Take for example the convolution represented in Figure 2.1. If the input and output were to be unrolled into vectors from left to right, top to bottom, the convolution could be represented as a sparse matrix \mathbf{C} where the non-zero elements are the elements $w_{i,j}$ of the kernel (with i and j being the row and column of the kernel respectively):

$$\begin{pmatrix} w_{0,0} & w_{0,1} & w_{0,2} & 0 & w_{1,0} & w_{1,1} & w_{1,2} & 0 & w_{2,0} & w_{2,1} & w_{2,2} & 0 & 0 & 0 & 0 & 0 \\ 0 & w_{0,0} & w_{0,1} & w_{0,2} & 0 & w_{1,0} & w_{1,1} & w_{1,2} & 0 & w_{2,0} & w_{2,1} & w_{2,2} & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & w_{0,0} & w_{0,1} & w_{0,2} & 0 & w_{1,0} & w_{1,1} & w_{1,2} & 0 & w_{2,0} & w_{2,1} & w_{2,2} & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & w_{0,0} & w_{0,1} & w_{0,2} & 0 & w_{1,0} & w_{1,1} & w_{1,2} & 0 & w_{2,0} & w_{2,1} & w_{2,2} & 0 \end{pmatrix}^T = \mathbf{C}^T$$

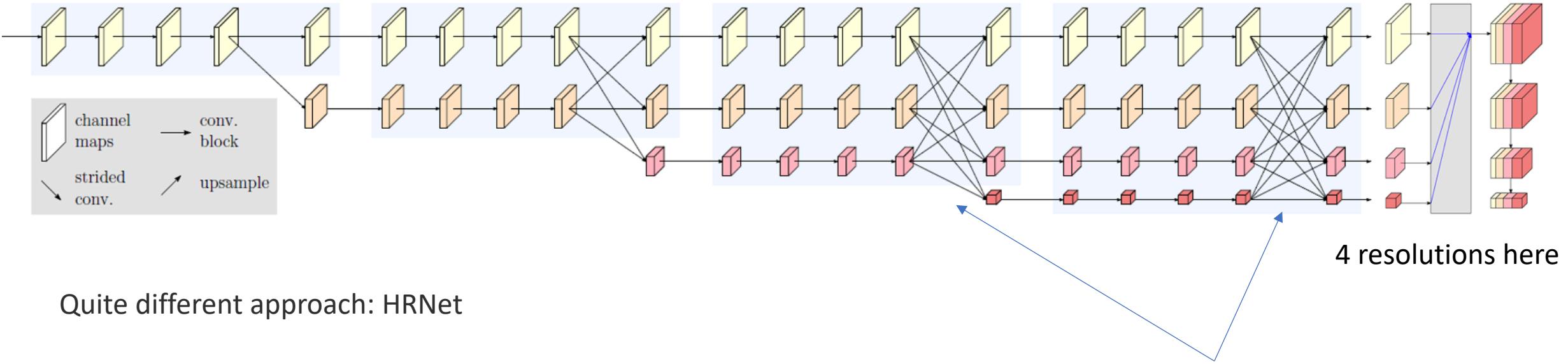
This linear operation takes the input matrix flattened as a 16-dimensional vector and produces a 4-dimensional vector that is later reshaped as the 2×2 output matrix.

Using this representation, the backward pass is easily obtained by transposing \mathbf{C} ; in other words, the error is backpropagated by multiplying the loss with \mathbf{C}^T . This operation takes a 4-dimensional vector as input and produces a 16-dimensional vector as output, and its connectivity pattern is compatible with \mathbf{C} by construction.

Notably, the kernel \mathbf{w} defines both the matrices \mathbf{C} and \mathbf{C}^T used for the forward and backward passes.

4.2 Transposed convolution

HRNet: a high-resolution network



Quite different approach: HRNet

- connecting multi resolution convolution *streams* **in parallel**
- maintains (not just recovers!) high-resolution representations through the whole process

Fusion module (just summation)
with appropriate (down-) up-samplings

<https://github.com/HRNet>

<https://arxiv.org/pdf/1908.07919.pdf>

Convolutional neural networks

Hope you now understand this meme ☺

Basic Math

$$\begin{pmatrix} A_{11} & A_{12} & \cdots & A_{1m} \\ A_{21} & A_{22} & \cdots & A_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ A_{n1} & A_{n2} & \cdots & A_{nm} \end{pmatrix}$$

Dangerous Artificial Intelligence

$$\begin{pmatrix} A_{11} & A_{12} & \cdots & A_{1m} \\ A_{21} & A_{22} & \cdots & A_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ A_{n1} & A_{n2} & \cdots & A_{nm} \end{pmatrix} * \begin{pmatrix} A_{11} & A_{12} & \cdots & A_{1m} \\ A_{21} & A_{22} & \cdots & A_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ A_{n1} & A_{n2} & \cdots & A_{nm} \end{pmatrix} * \begin{pmatrix} A_{11} & A_{12} & \cdots & A_{1m} \\ A_{21} & A_{22} & \cdots & A_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ A_{n1} & A_{n2} & \cdots & A_{nm} \end{pmatrix} * \begin{pmatrix} A_{11} & A_{12} & \cdots & A_{1m} \\ A_{21} & A_{22} & \cdots & A_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ A_{n1} & A_{n2} & \cdots & A_{nm} \end{pmatrix}$$