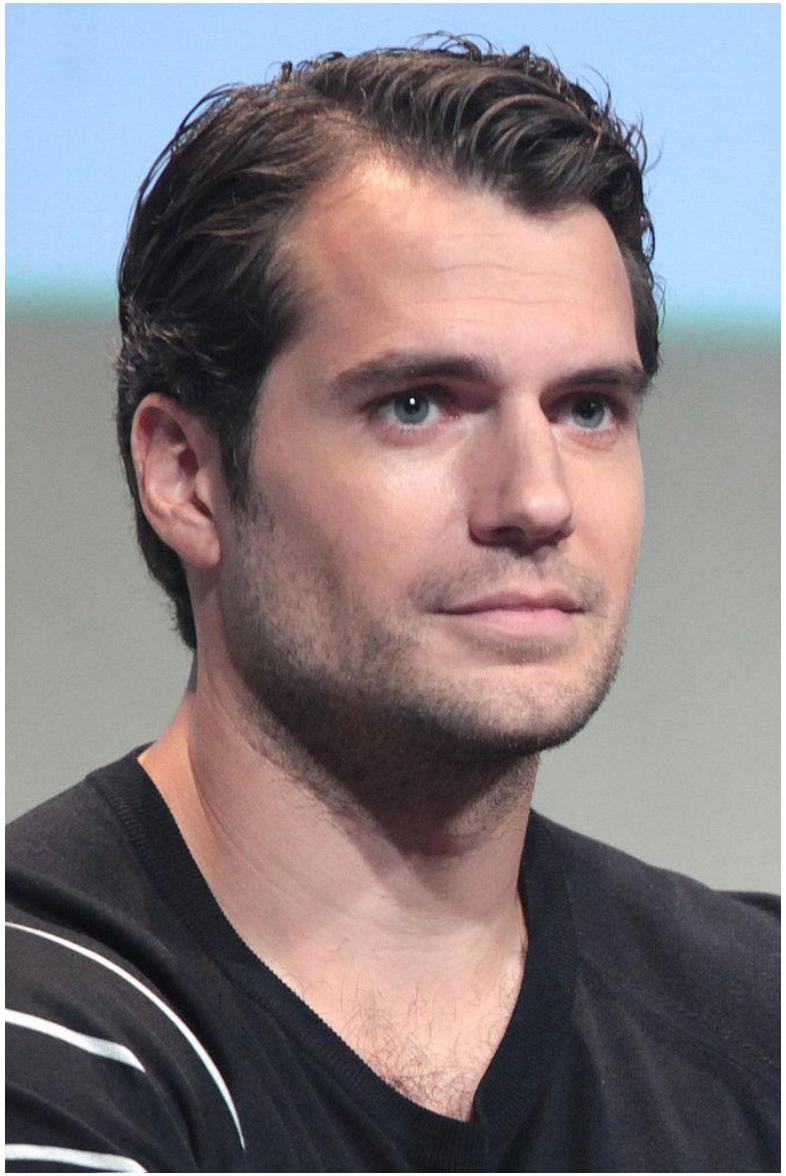
Deep Learning for images

Álvaro Barbero Jiménez
alvaro.barbero.jimenez@gmail.com



https://en.wikipedia.org/wiki/Lenna, Henry Cavill photo by Gake Skidmore - https://commons.wikimedia.org/wiki/File:Henry_Cavill_by_Gage_Skidmore.jpg

Spatial data

- Some kinds of data have spatial structure
- A distance can be defined between features: space, time, …

Images

Videos

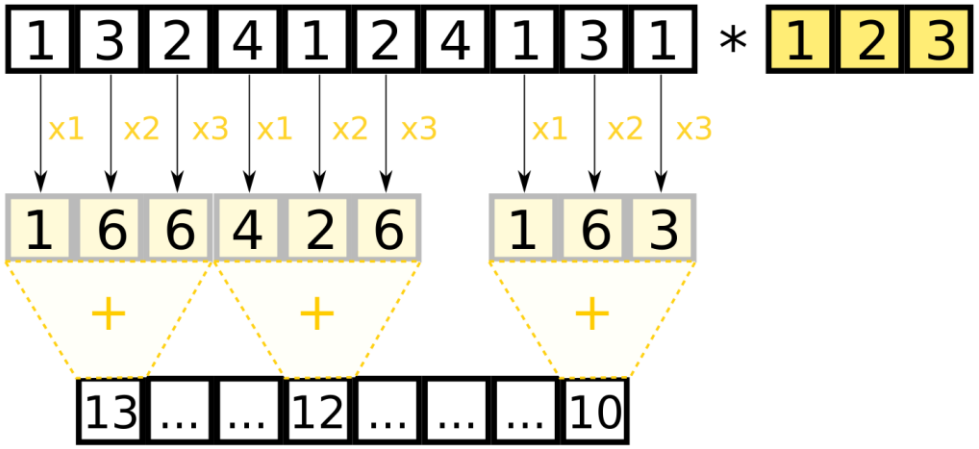
Time series

Not taking into account the distances between features means throwing away useful information

Convolution

- Operation between continuous functions $(x * z)(t) = \int_{-\infty}^{+\infty} x(a)\, z(t-a)\, \mathrm{d}a$
- For discrete signals $(x * z)_i = \sum_{j=-\infty}^{+\infty} x_j\, z_{i-j}$
- Applies a small layer of weights over each portion of the input

1D data



2D data

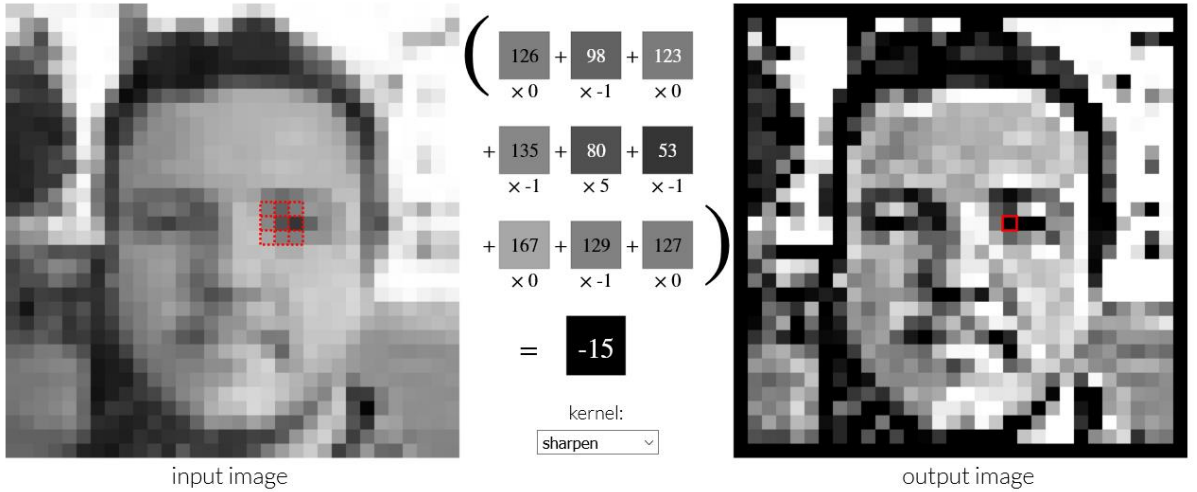


Image Kernels - http://setosa.io/ev/image-kernels/

Usefulness of convolutions

Convolving an image with a small, local kernel of weights can produce interesting image transformations

0	0	0
0	1	0
0	0	0

identity

Image Kernels - http://setosa.io/ev/image-kernels/

Usefulness of convolutions

Convolving an image with a small, local kernel of weights can produce interesting image transformations

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

bottom sobel

Image Kernels - http://setosa.io/ev/image-kernels/

Usefulness of convolutions

Convolving an image with a small, local kernel of weights can produce interesting image transformations

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

emboss

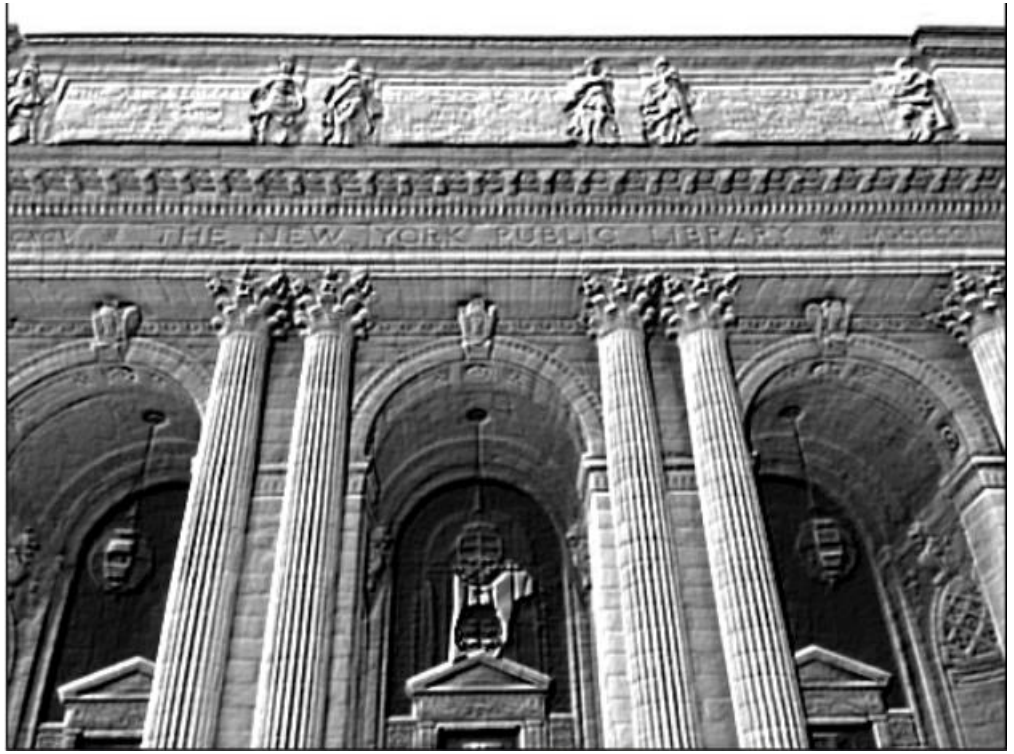
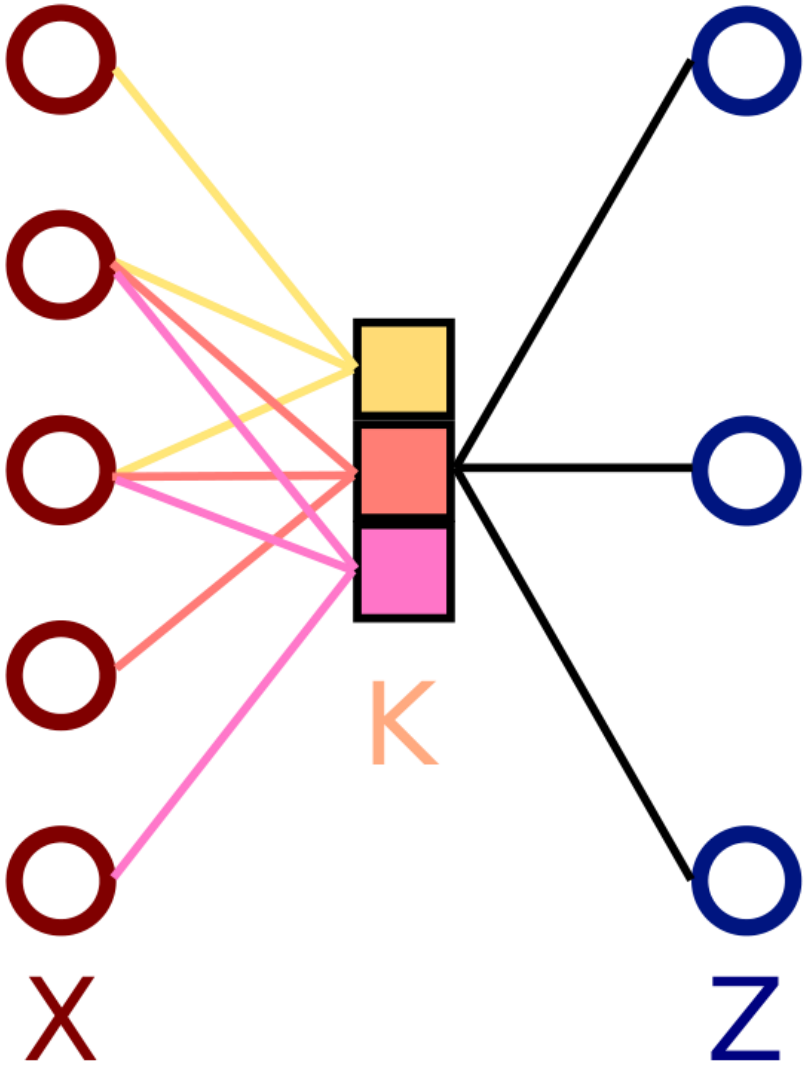


Image Kernels - http://setosa.io/ev/image-kernels/

Convolutional layers



Given a kernel of weights or filter K, output is $Z = X * K$

The weights K of the layer are learned during training

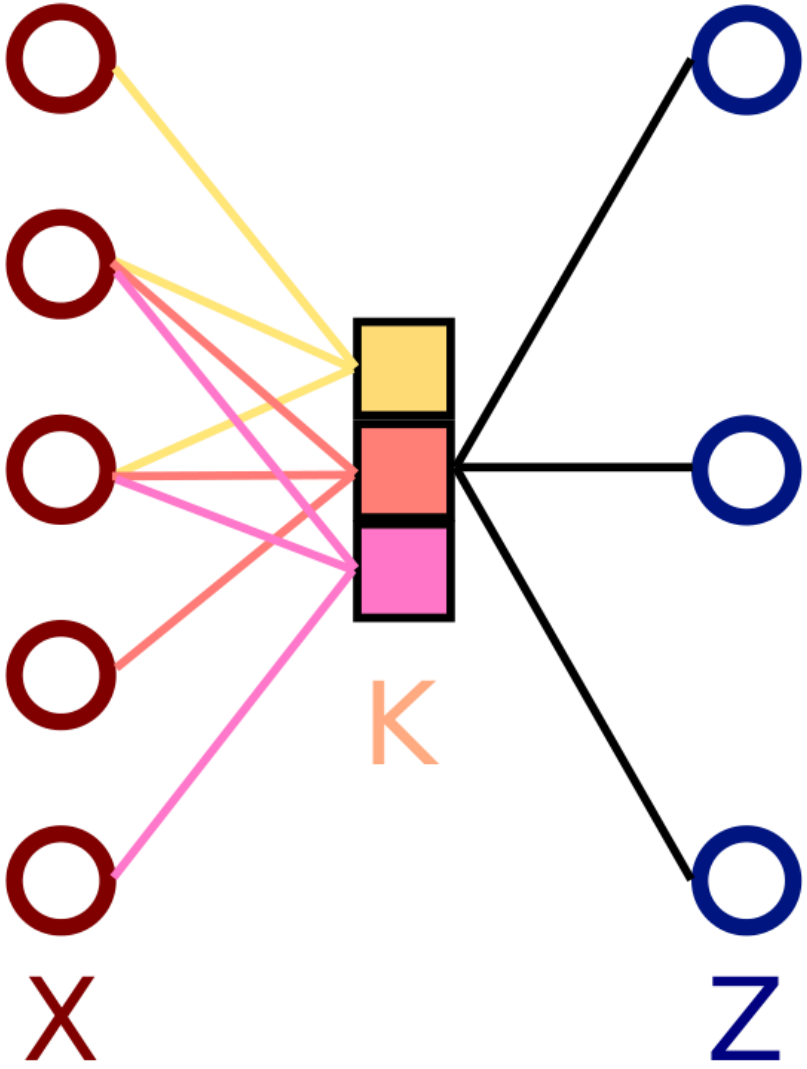
- Focus on finding local combinations of inputs that are useful for the task

In a convolutional layer usually several filter or weight matrices K_i are learned jointly

Inputs can be of arbitrary size: output varies accordingly

Boundary cases: clip outputs or fill inputs

Derivatives in convolutional layers



With w = kernel size

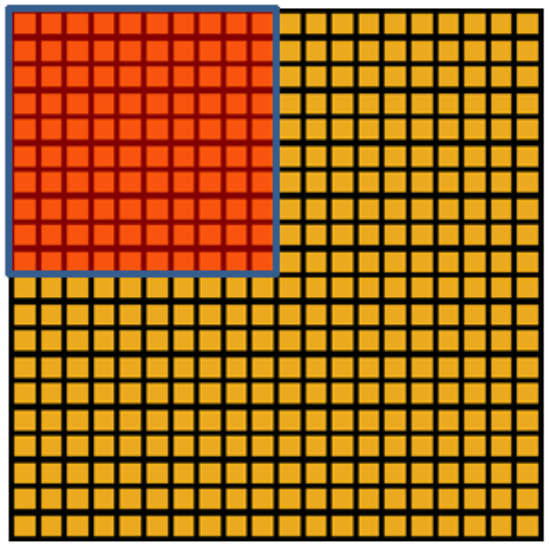
$$\frac{\partial z_i}{\partial x_j} = \frac{\partial}{\partial x_j}[k * x]_i = \frac{\partial}{\partial x_j}\sum_{m=1}^{w} k_m x_{(i-\lceil w/2 \rceil + m)}$$

$$= k_{(j-i+\lceil w/2 \rceil)} = k_{(i-\lceil w/2 \rceil + j)}{}^{flip}$$

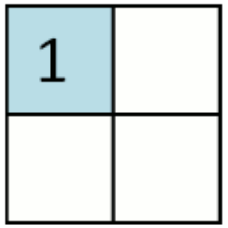
$$\frac{\partial z_i}{\partial k_j} = \frac{\partial}{\partial k_j}[k * x]_i = \frac{\partial}{\partial k_j}\sum_{m=1}^{w} K_m x_{(i-\lceil w/2 \rceil + m)}$$

$$= x_{(i-\lceil w/2 \rceil + j)}$$

Pooling



Convolved feature



Pooled feature

Computes summary statistics of groups of inputs

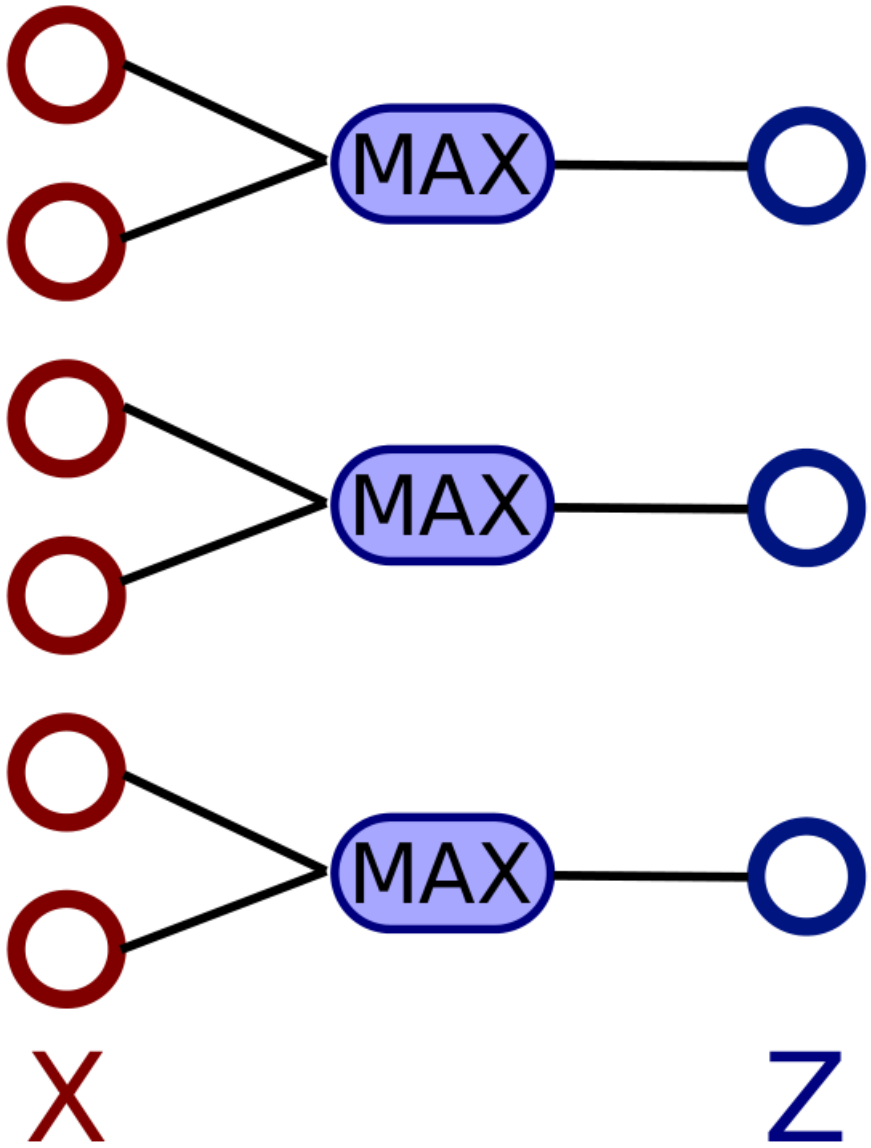
Useful for reducing a large number of features to a meaningful reduced set

Invariant to scaling and small translations

Generally max operation works best

UFLDL Tutorial - http://ufldl.stanford.edu/wiki/index.php/Pooling

Pooling layers



Apply pooling operation to sets of input neurons

$$Z = \max_i \{X_i\}$$

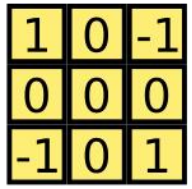
Backpropagation within a group:

$$\frac{\partial z_i}{\partial x_j} = \frac{\partial}{\partial x_j} \max_k x_k = \begin{cases} 1 & if \quad x_j = \max_k x_k \\ 0 & else \end{cases}$$

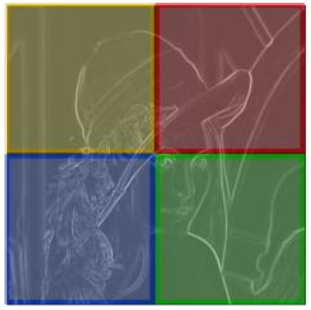
(Subgradient of max formula)

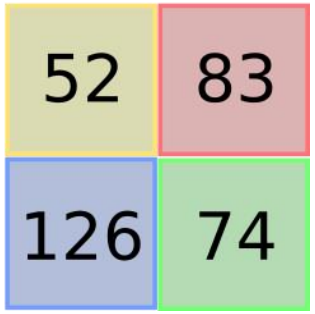
Convolution + pooling

Convolution
Detect useful local features

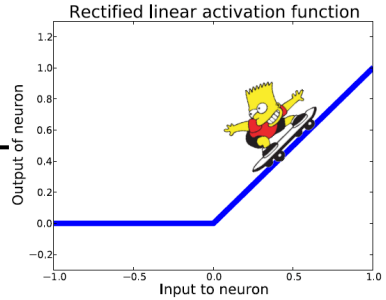


Max pooling
Focus on most prominent features

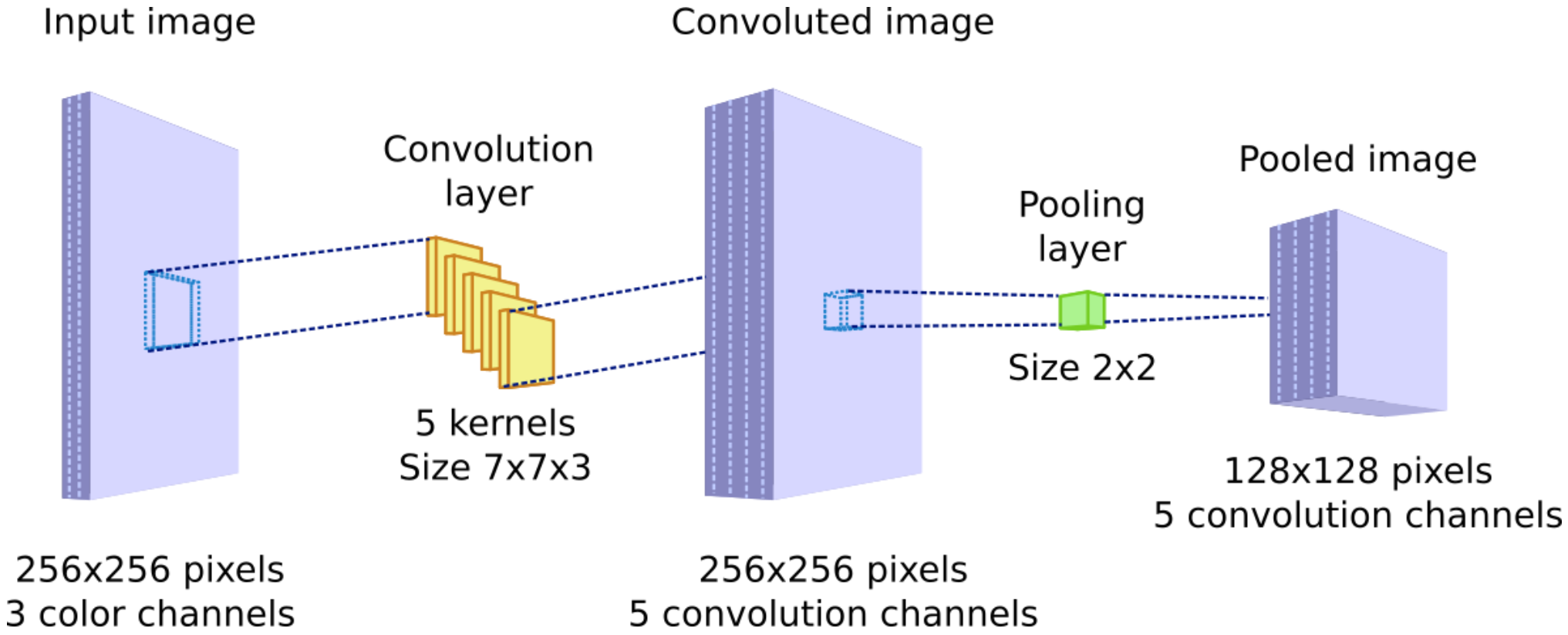




ReLU (optional)
Ignore negative features

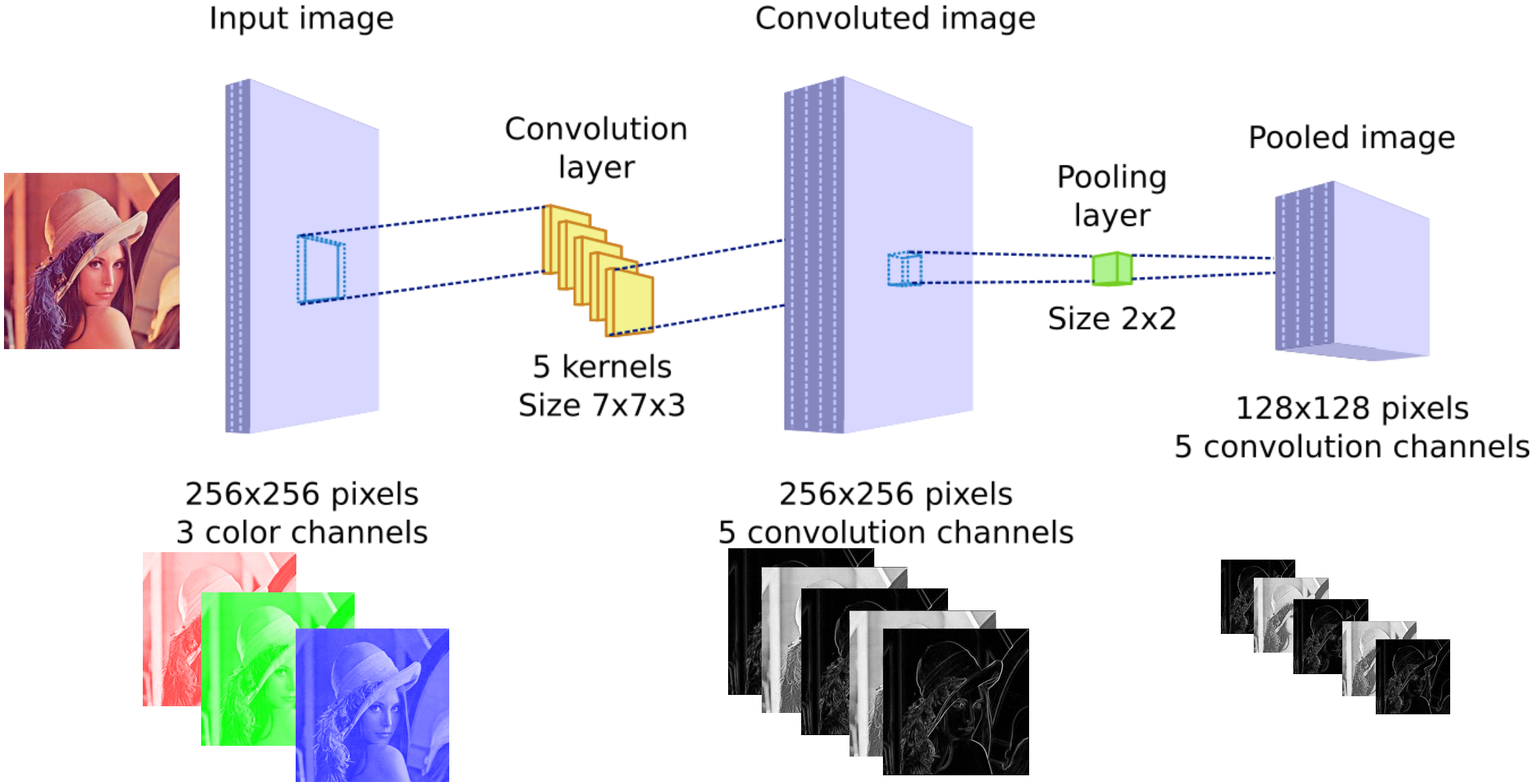


Convolution + pooling for images

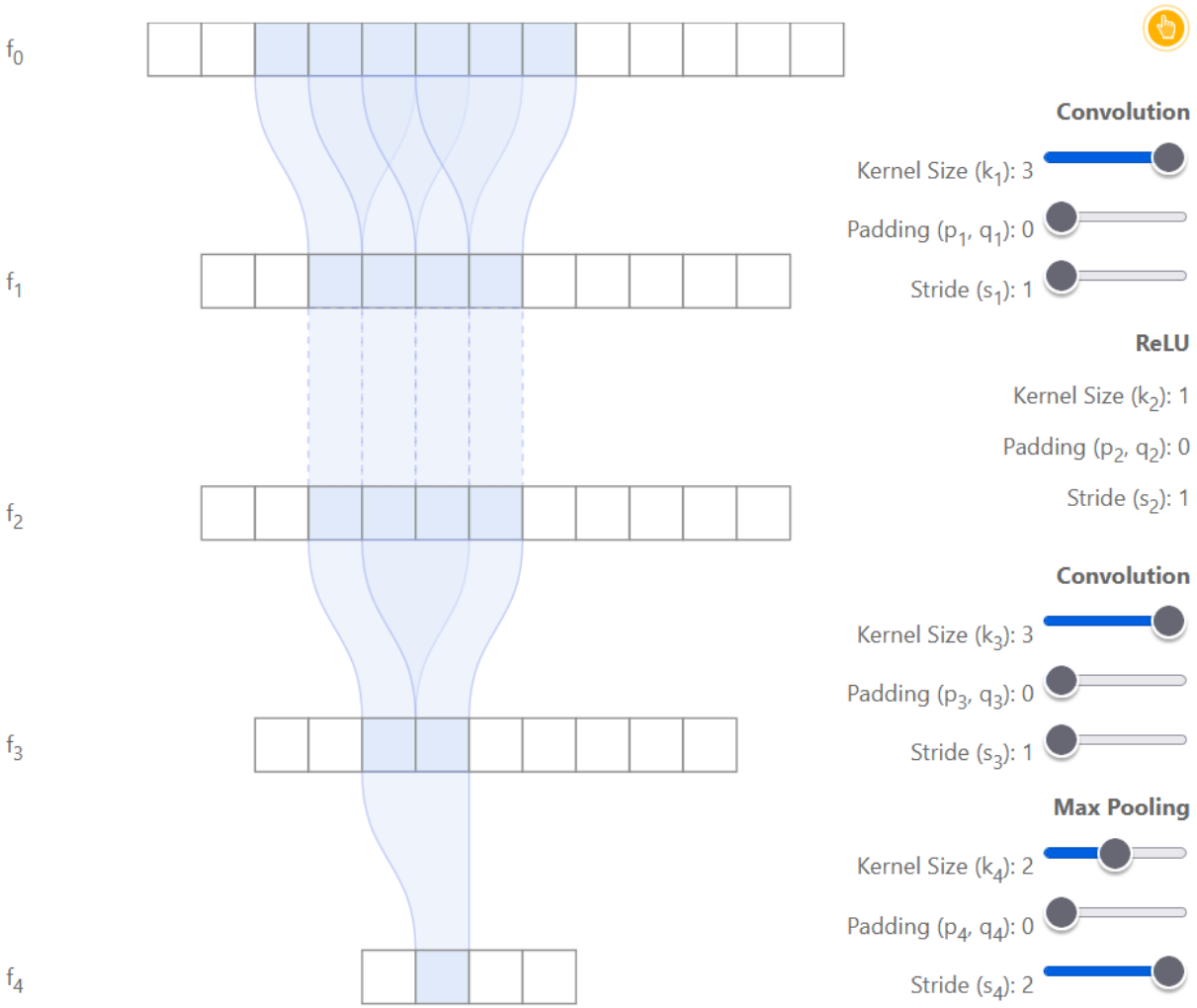


- All operations are made in terms of 3D tensors (width x height x channels)
- Convolutions replace original (color) channels by filter transformations
- Kernels operate over all channels

Convolution + pooling for images: example



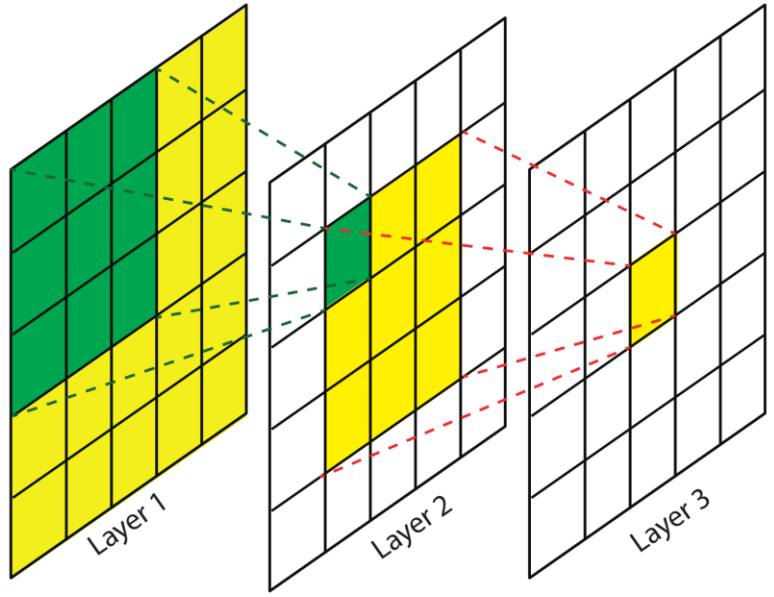
Receptive field



The receptive field of a layer is the set of pixel inputs that are used to compute the features in such layer.

The size of the field grows linearly with the number of layers in the network.

In this way, deeper convolutions compute features based on a larger number of input pixels → larger objects can be detected.



Adaloglou - Understanding the receptive field of deep convolutional networks

Araujo et al - Computing Receptive Fields of Convolutional Neural Networks

Lin et al - Maritime Semantic Labeling of Optical Remote Sensing Images with Multi-Scale Fully Convolutional Network

Usefulness of large receptive fields

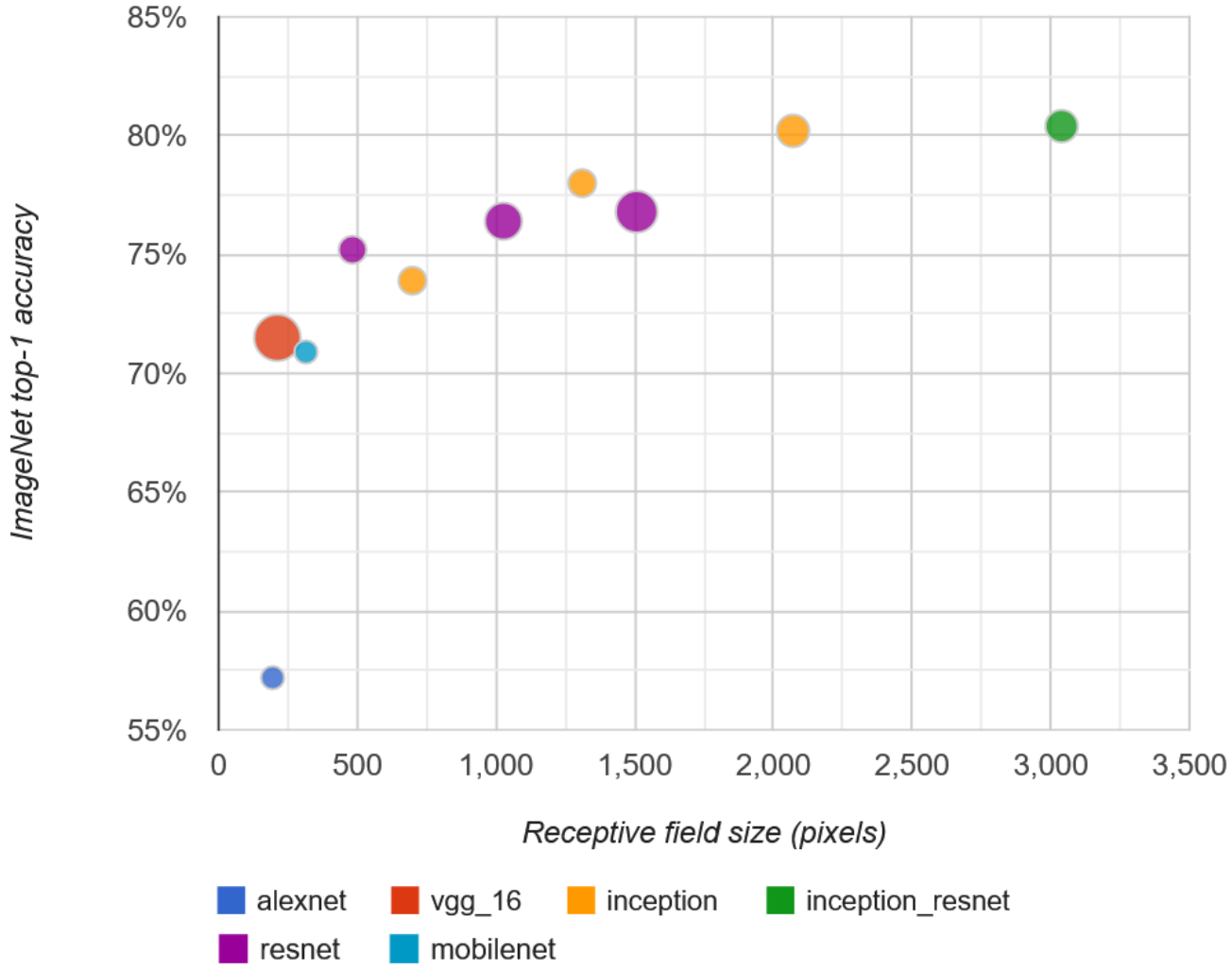
The size of the receptive field of a network with L convolutional layers can be computed as

$$\sum_{l=1}^{L} k_l + 1$$

with k_l the kernel size of the l-th layer.

When pooling is used, each pooling layer multiplies the size of the receptive field by the pooling size. For example: pool size = 2 multiplies the receptive field by 2.

Networks with larger receptive fields produce better results.



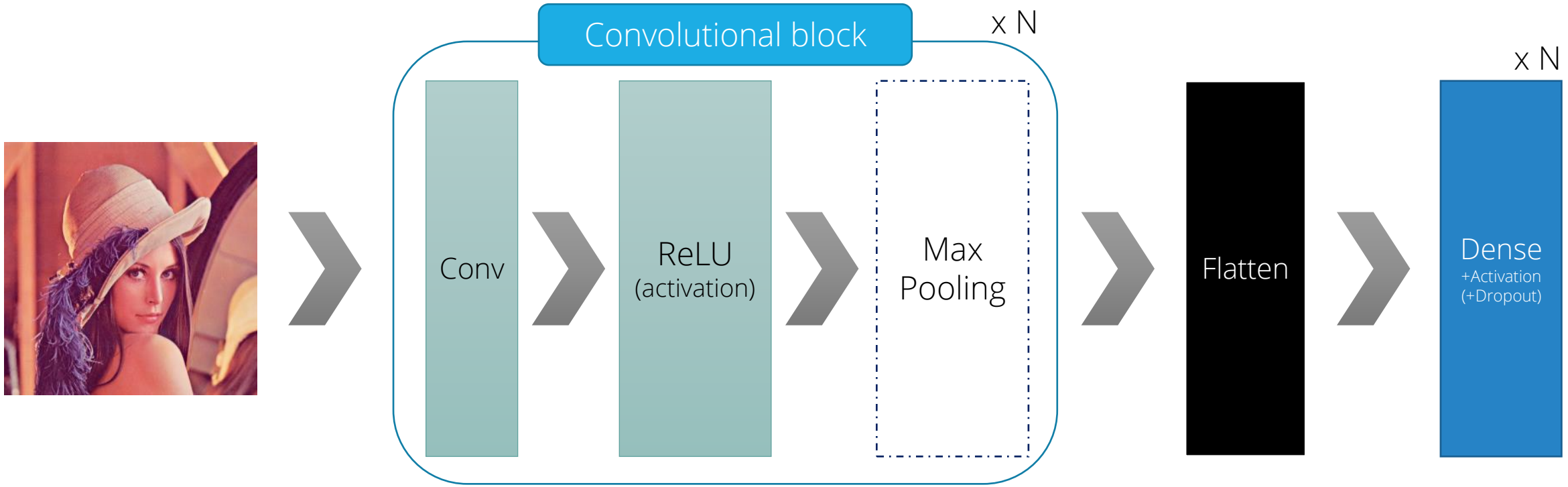
Adaloglou - Understanding the receptive field of deep convolutional networks

Araujo et al - Computing Receptive Fields of Convolutional Neural Networks

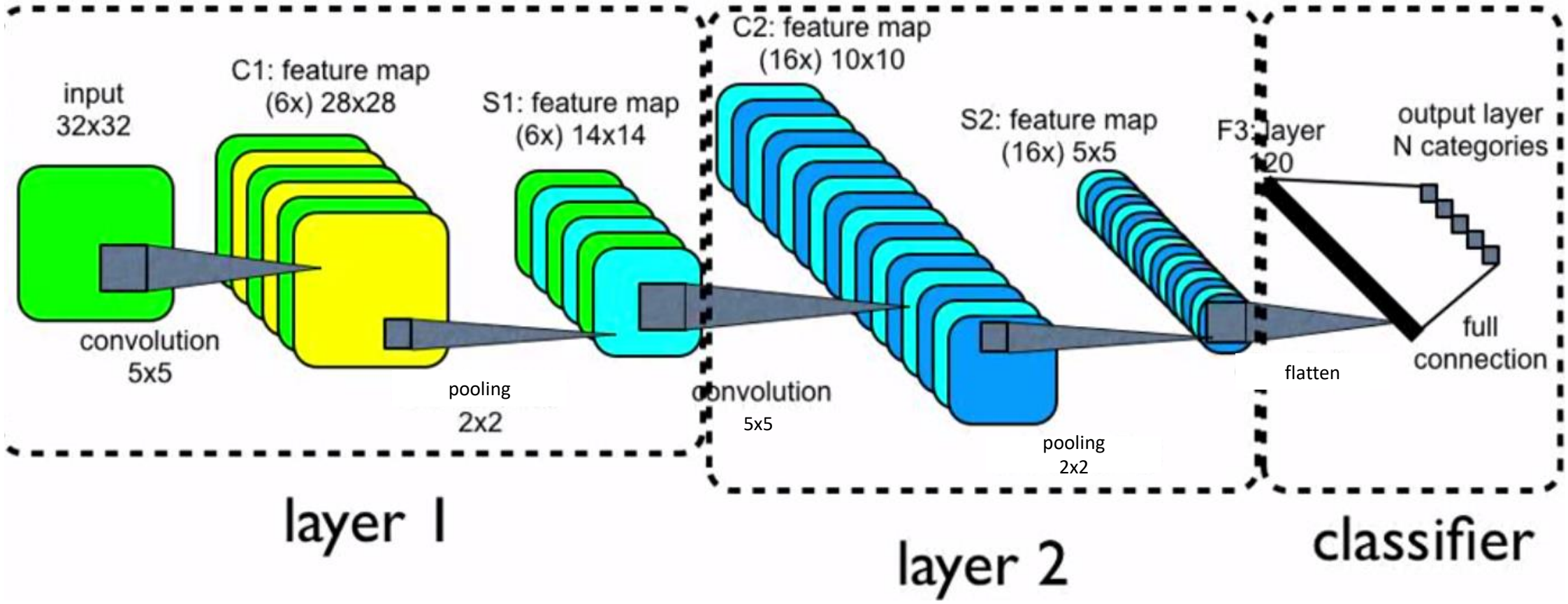
Convolutional neural networks (CNNs)

A Convolutional Neural Network (CNN) is built by stacking groups of convolutional + pooling layers, then flattening the outputs of the last layer into a feature vector. Such vector is then used as input to standard (Dense) neural network layers.

In this way, a CNN is basically a network that can engineer its own features.



CNN example: basic network with 2 convs + pools



CNN example: ImageNet 2012 winner

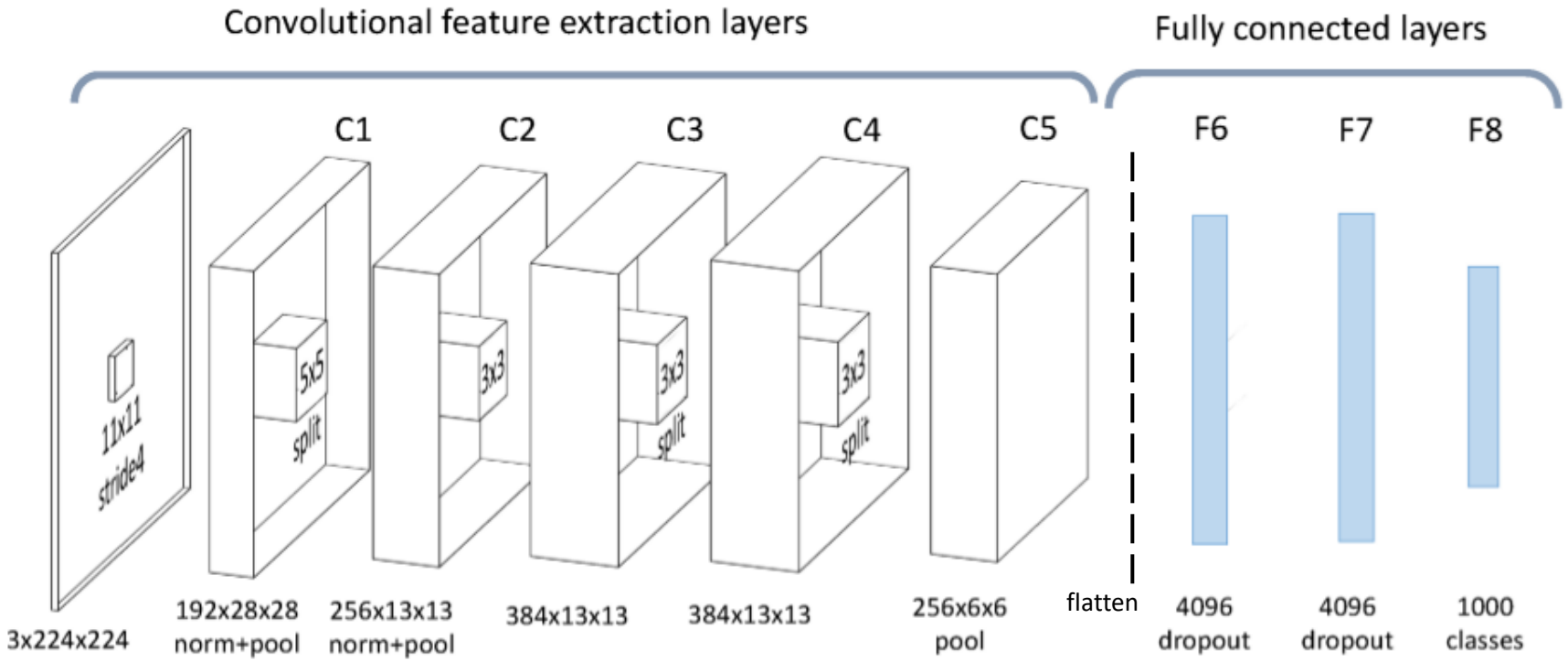
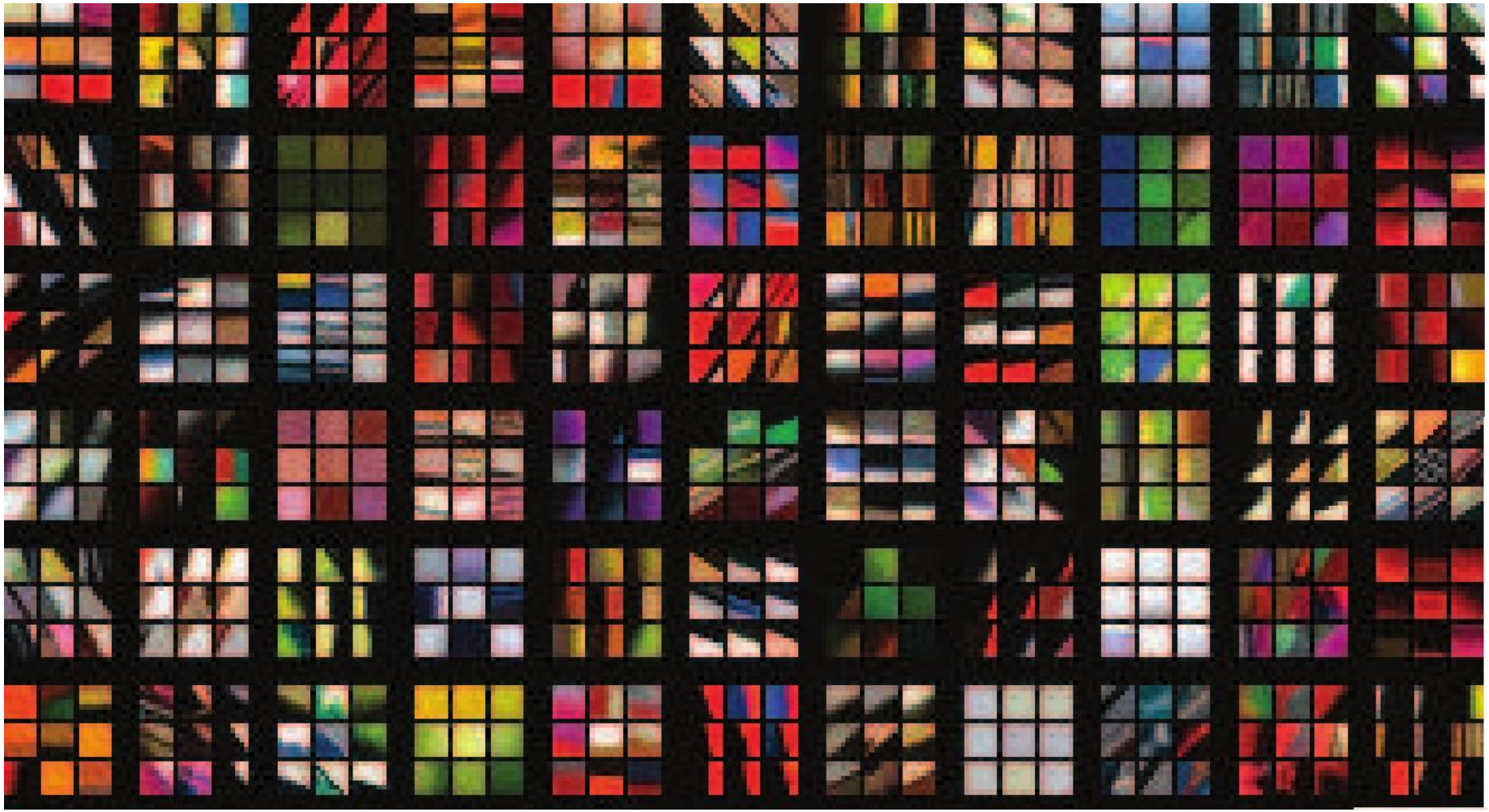


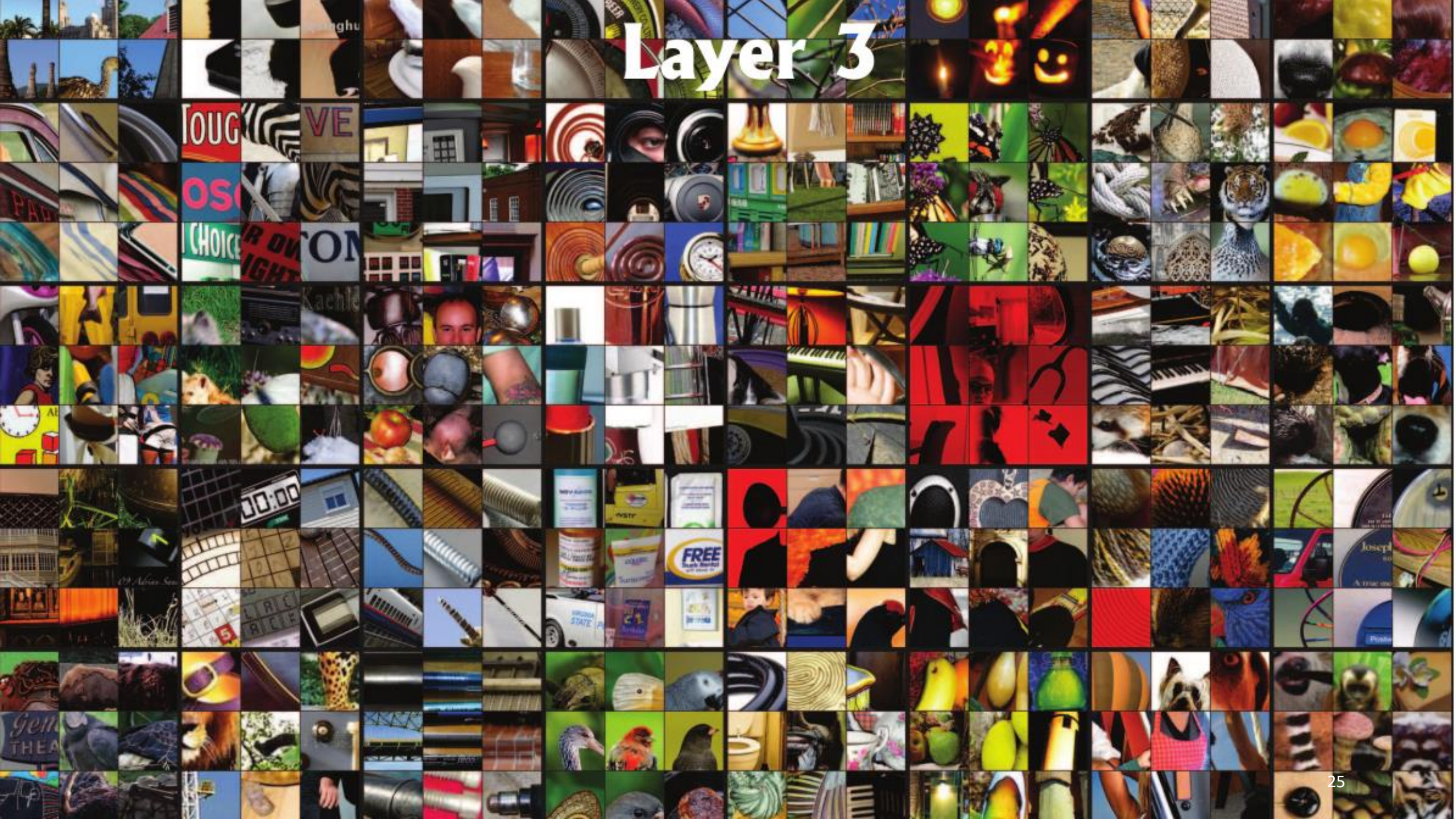
Fig. 2.3: Architecture for image recognition. The 2012 ILSVRC winner consists of eight layers [82]. Each layer performs a linear transformation (specifically, convolutions in layers C1–C5 and matrix multiplication in layers F6–F8) followed by nonlinear transformations (rectification in all layers, contrast normalization in C1–C2, and pooling in C1–C2 and C5). Regularization with dropout noise is used in layers F6–F7.

Bottou et al - Optimization Methods for Large-Scale Machine Learning

Patches that give largest activations – Layer 1

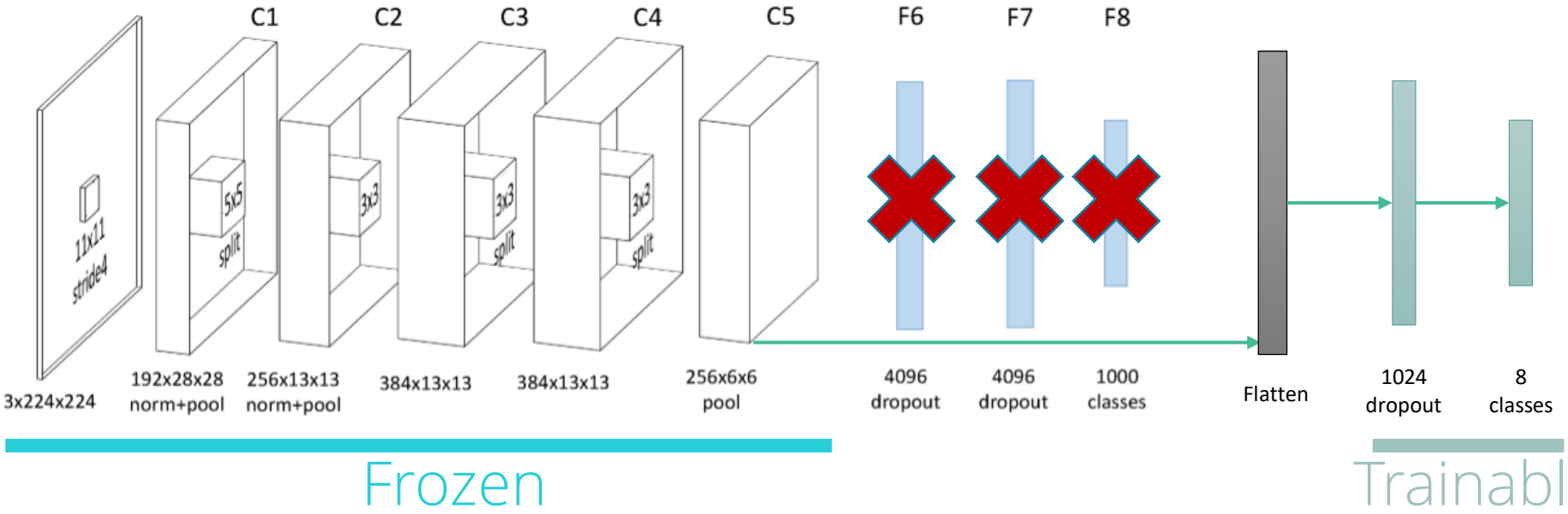



Layer 2


Layer 3


Layer 5

Transfer learning: bottleneck features



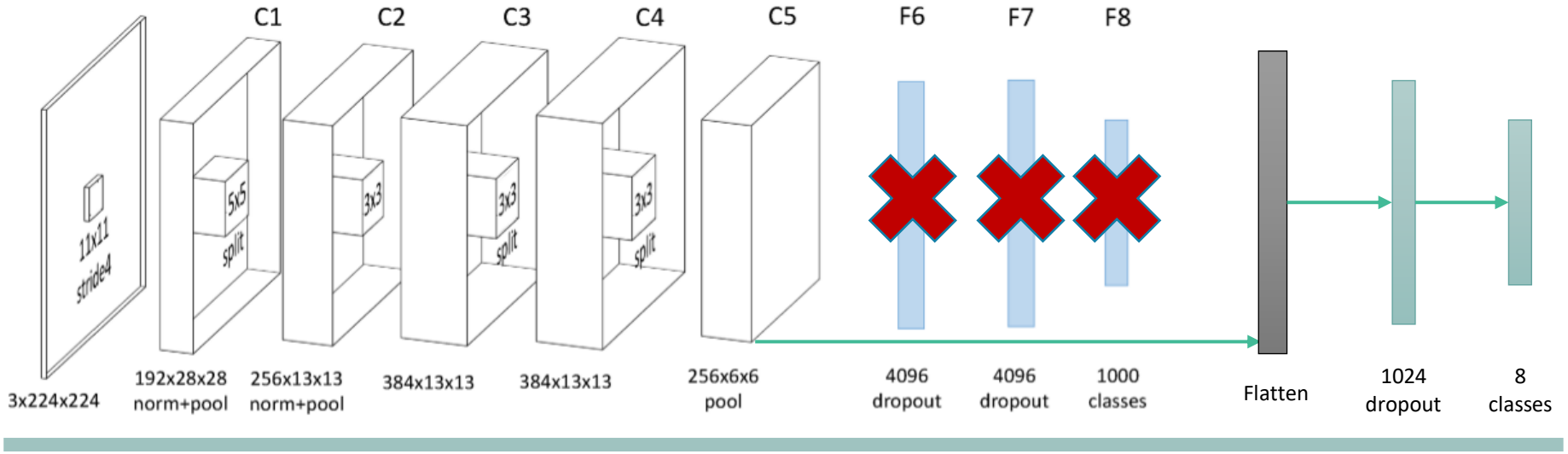
Use a large network pre-trained on a different, large dataset as a feature generator.

- VGG16, VGG19, ResNet50, InceptionV3, DenseNet, MobileNet, …

Extract the value of activations in the last convolutional layer, add flattening and small trainable layers after that.

Warning: make sure the input images are normalized with the same procedure that was used in the pre-trained network. Especially if the network includes Batch Normalization layers, as the running statistics will be inadequate.

Transfer learning: fine-tuning

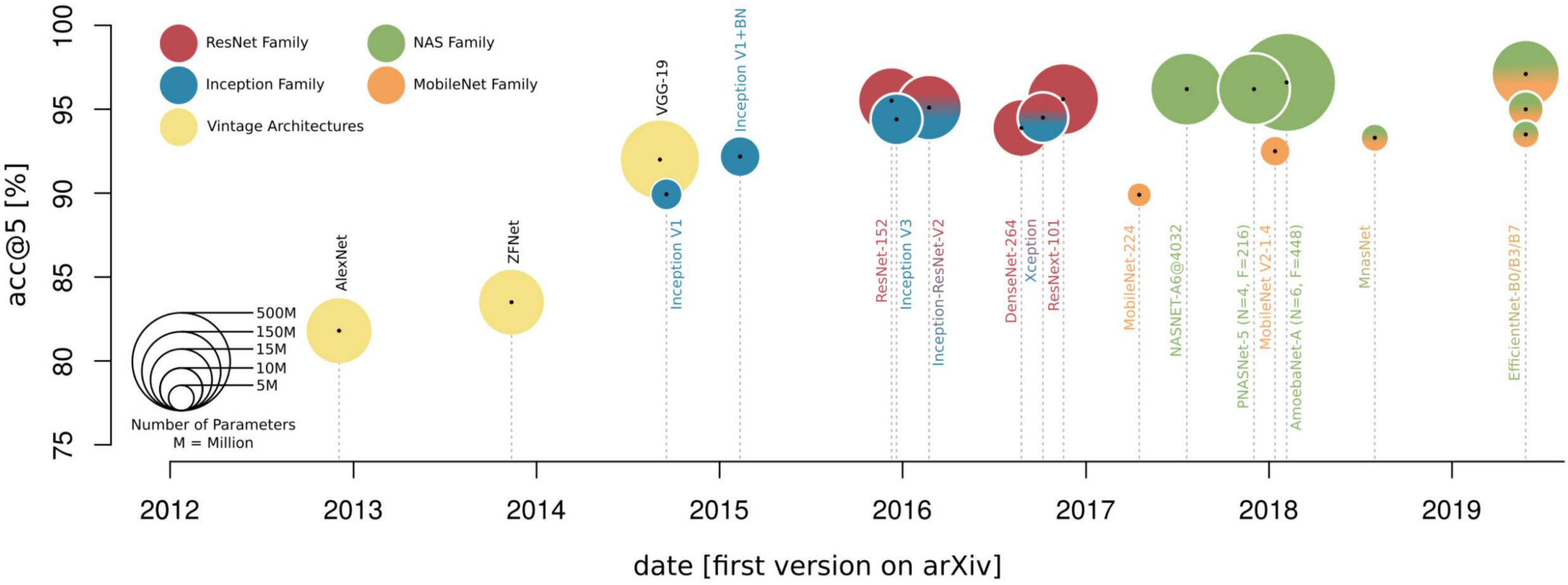


Alternatively, remove again the head and add some custom layers, but train the whole network.

To avoid catastrophic forgetting, use a small learning rate ($\mathbf{0.0001}$) in the optimizer, and preserve the original image normalization procedure.

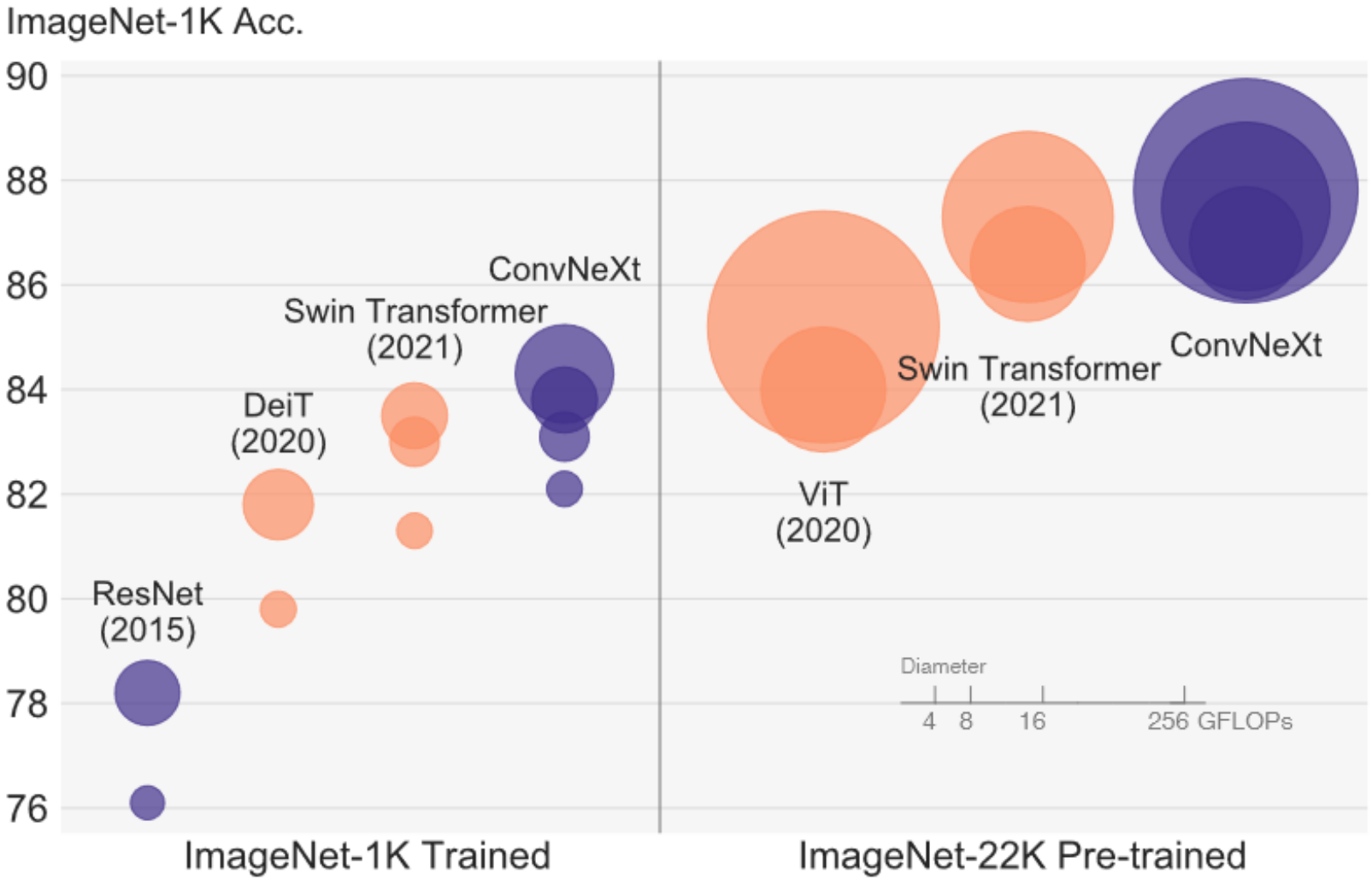
Sometimes it produces better results to run first some iterations of bottleneck features, then do fine-tuning over the whole network.

Comparison of pre-trained networks in Imagenet



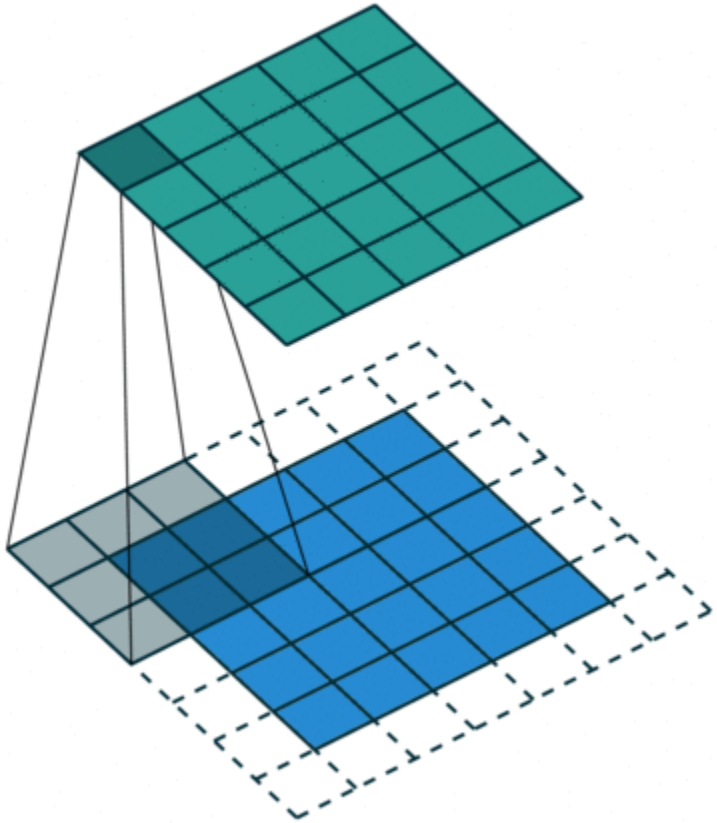
Hoeser and Kuenzer - Object Detection and Image Segmentation with Deep Learning on Earth Observation Data: A Review-Part I: Evolution and Recent Trends

Comparison of pre-trained networks in Imagenet

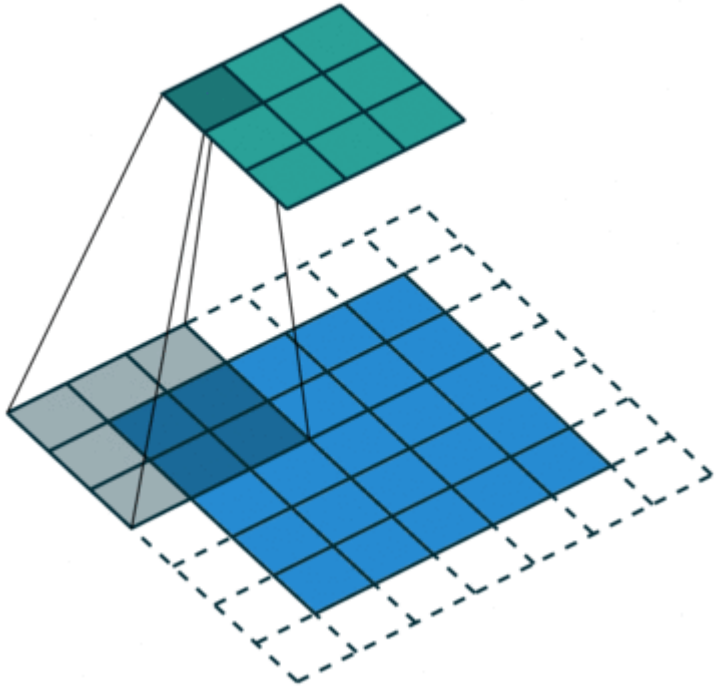


A ConvNet for the 2020s - https://github.com/facebookresearch/ConvNeXt

Strided convolution ≃ convolution + learnable pooling



No stride

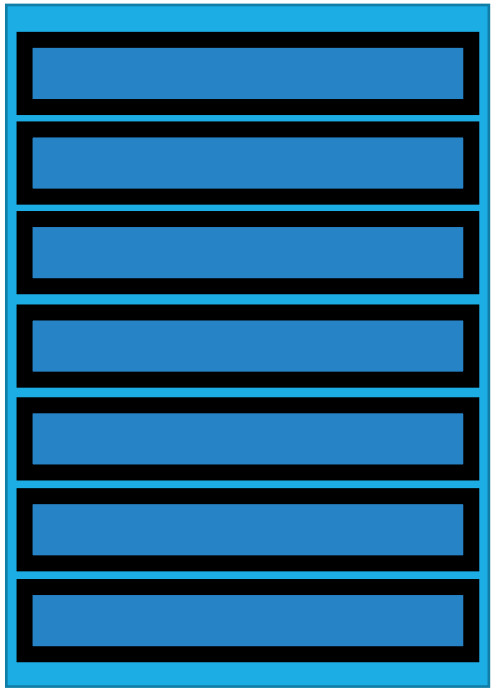


stride = 2

A technical report on convolution arithmetic in the context of deep learning - https://github.com/vdumoulin/conv_arithmetic

Normalization layers

Data batch x



Normalize

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

Rescale

$$y_i = \gamma \hat{x}_i + \beta$$

Output

y

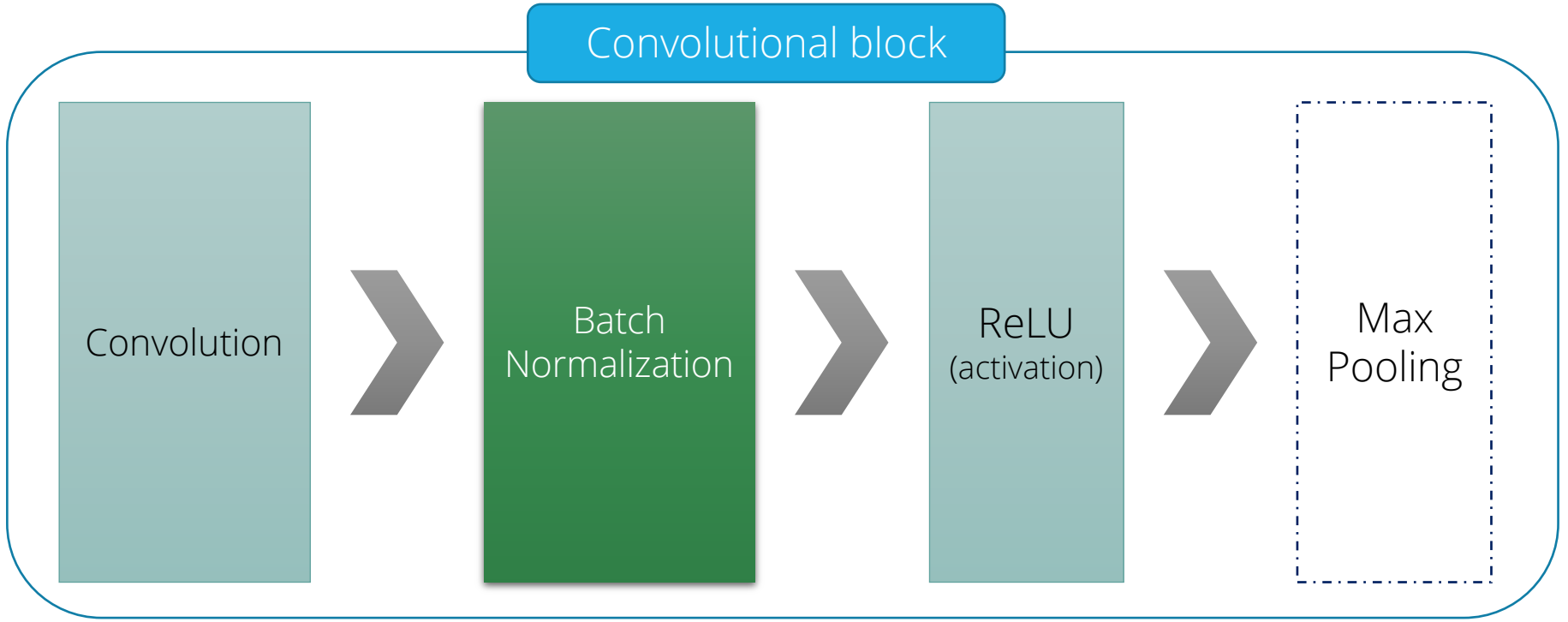
μ_B, σ_B^2

Compute batch statistics

- Batch statistics (mean μ_B, variance σ_B^2) are computed for every training step
- Data is normalized with batch statistics, to obtain $\hat{x}$ versions with zero mean and unit variance
  - Small ϵ added for numerical stability
- Learnable rescale parameters γ, β are available to recover extreme values in case the network really needs them
- This layer is usually added before the inputs of other processing layers (except the first one)
- At prediction time or when the layer is frozen: μ_b, σ_B^2 are set to a moving average of the mean and variance of the batches seen during training.

Ioffe and Szegedy. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariance Shift

Normalization layers: usage



The recommended place to add Batch Normalization is between convolution and activation layers (like ReLU). This way we make sure the inputs to the activation are close to the point of non-linearity (0).

Combining BatchNormalization with Dropout can result in too much noise: Dropout random deactivations + normalization statistics computed for just a single batch. Therefore, BatchNormalization is usually applied to Convolutional blocks, while Dropout is applied to Dense blocks.

Ioffe and Szegedy. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariance Shift

Normalization layers: results

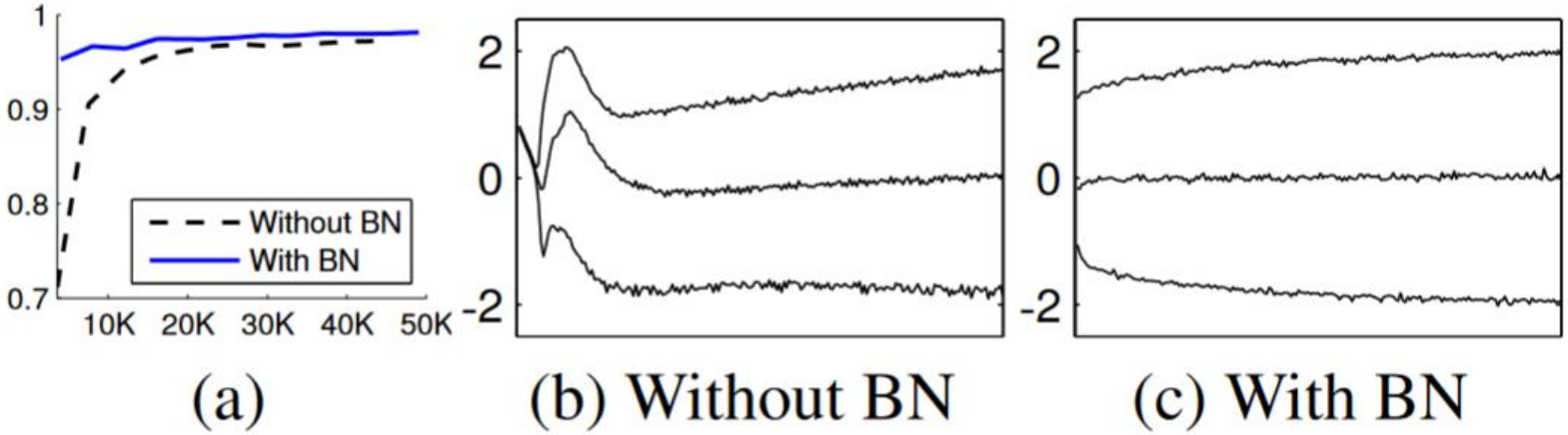


Figure 1. (a) *The test accuracy of the MNIST network trained with and without Batch Normalization, vs. the number of training steps. Batch Normalization helps the network train faster and achieve higher accuracy.* (b, c) *The evolution of input distributions to a typical sigmoid, over the course of training, shown as* $\{15, 50, 85\}$*th percentiles. Batch Normalization makes the distribution more stable and reduces the internal covariate shift.*

Ioffe and Szegedy. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariance Shift

Problems in training very deep networks

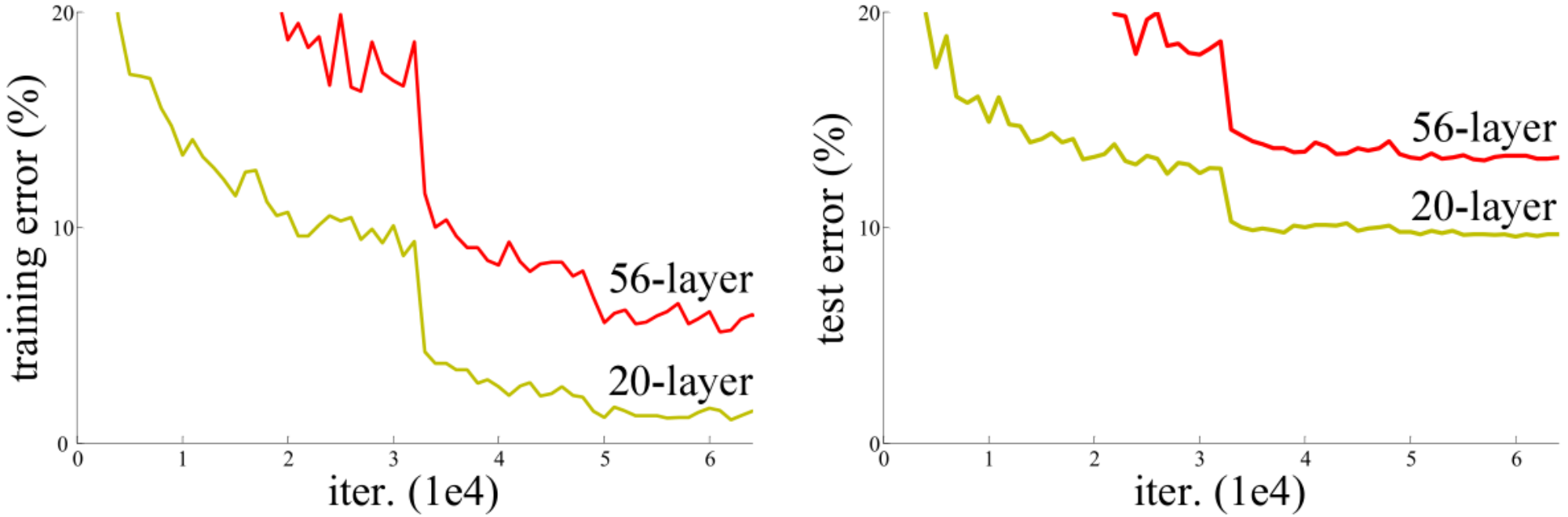
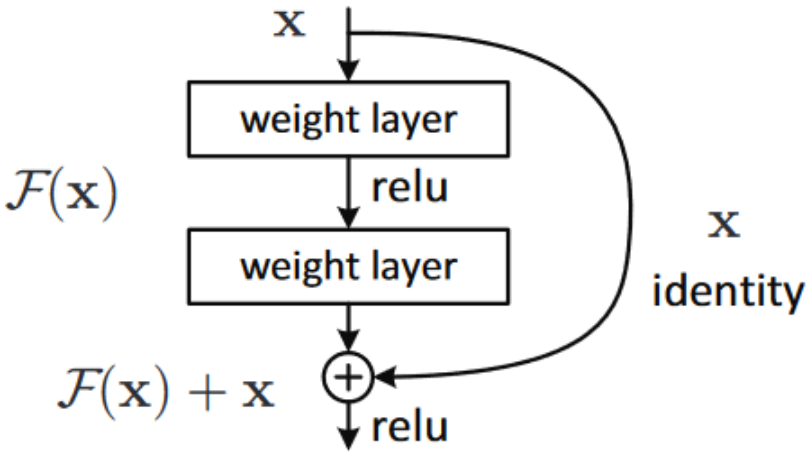


Figure 1. Training error (left) and test error (right) on CIFAR-10 with 20-layer and 56-layer "plain" networks. The deeper network has higher training error, and thus test error. Similar phenomena on ImageNet is presented in Fig. 4.

He et al. Deep Residual Learning for Image Recognition

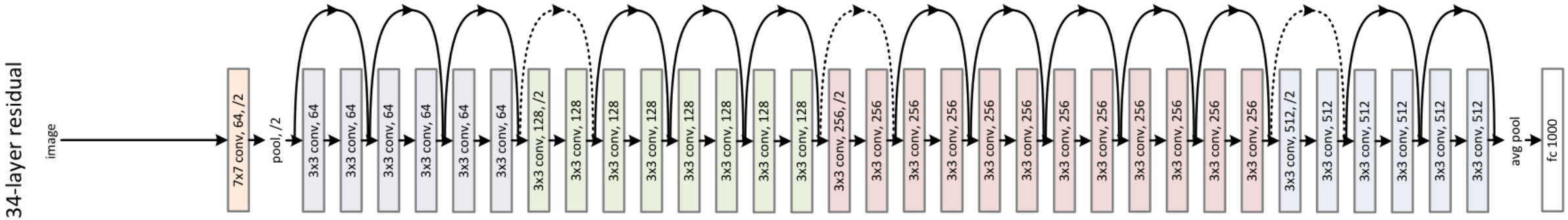
Residual connections



In very deep networks (>100 layers) it might be impractical to make a real use for of the deepest layers, even resorting to all the training tricks before.

A way to accelerate training in deepest layers is to allow residual connections: connections that allow the data to skip layers if necessary.

This allows training very deep networks, as well as accelerating medium-sized networks.



He et al. Deep Residual Learning for Image Recognition

Residual connections: performance

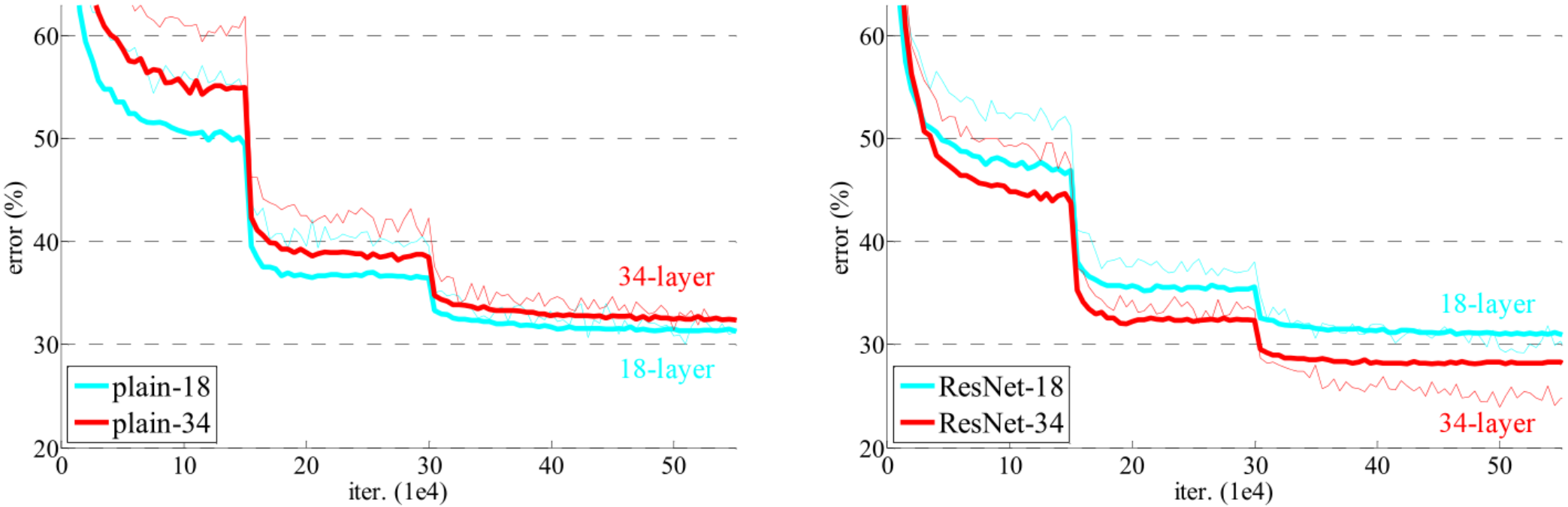
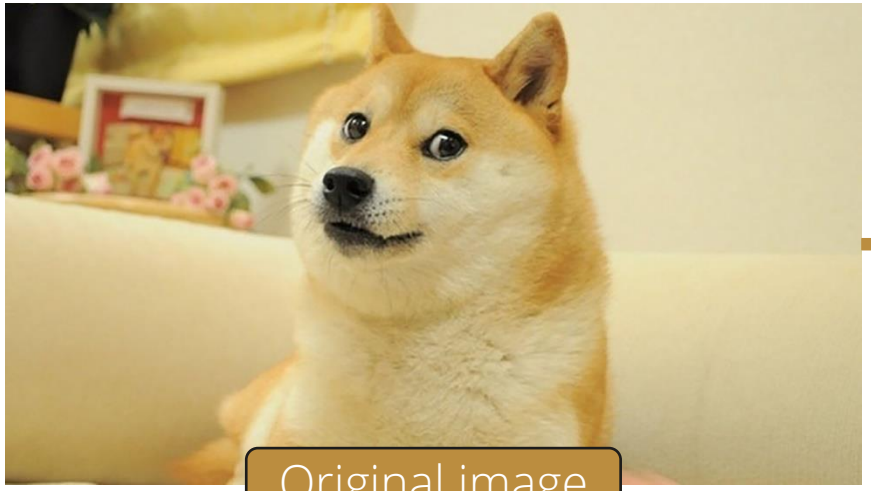


Figure 4. Training on **ImageNet**. Thin curves denote training error, and bold curves denote validation error of the center crops. Left: plain networks of 18 and 34 layers. Right: ResNets of 18 and 34 layers. In this plot, the residual networks have no extra parameter compared to their plain counterparts.

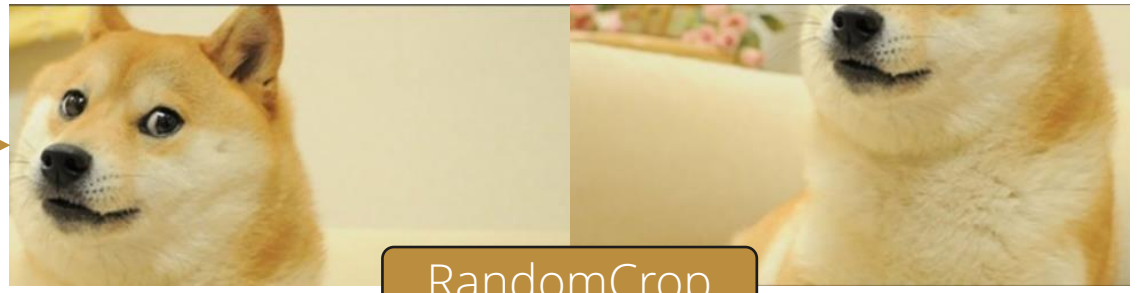
He et al. Deep Residual Learning for Image Recognition

Image augmentation

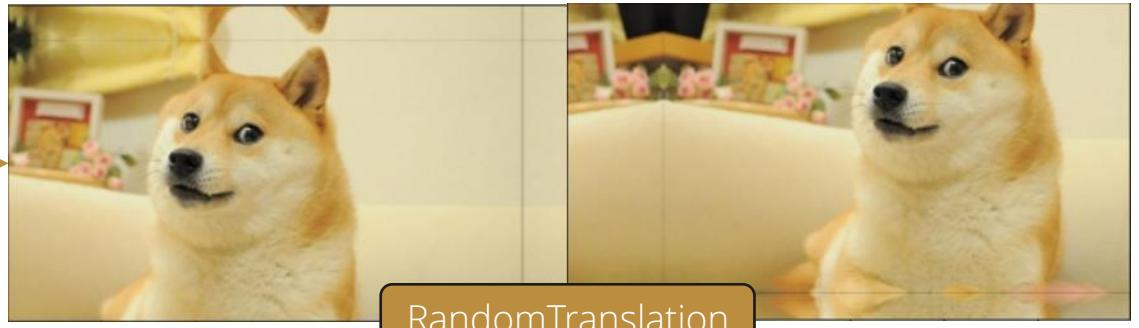


Original image

RandomFlip



RandomCrop



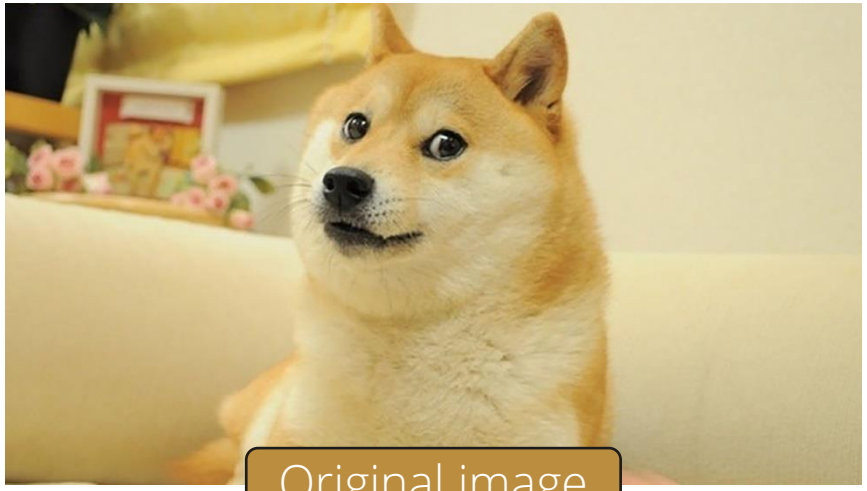
RandomTranslation

We can augment the training dataset by creating new synthetic images, derived from transformations of the original images. The strength of the transformations must be controlled to avoid creating very distorted images.

This is only done for training data! We want to use original images for validation and test, to correctly measure the performance of our model.

Keras image preprocessing and augmentation layers - https://keras.io/api/layers/preprocessing_layers/image_preprocessing/

Image augmentation



Original image

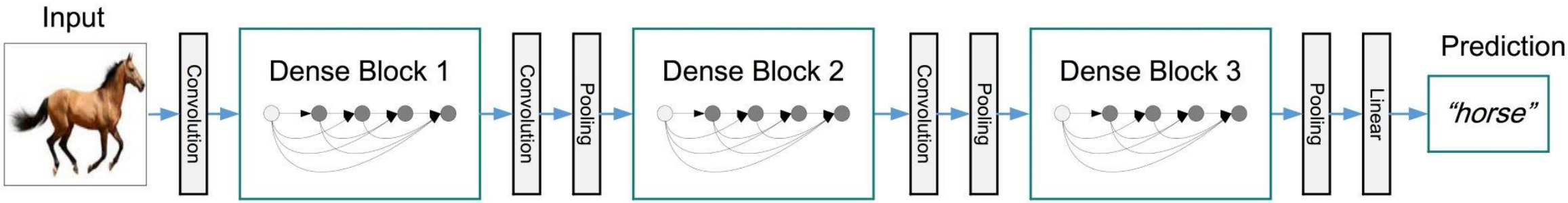
```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import RandomFlip, RandomCrop. RandomTranslation
from tensorflow.keras.layers import RandomRotation, RandomZoom

model = Sequential()
model.add(RandomFlip(mode="horizontal"))
model.add(RandomZoom(height_factor=(-0.2, 0.2), width_factor=(-0.2, 0.2)))
model.add(RandomRotation(factor=(-0.1, 0.1)))
model.add(RandomTranslation(height_factor=(-0.2, 0.2), width_factor=(-0.2, 0.2)))
model.add(RandomCrop(height=400, width=800))
# ... and then add the rest of the network layers
```

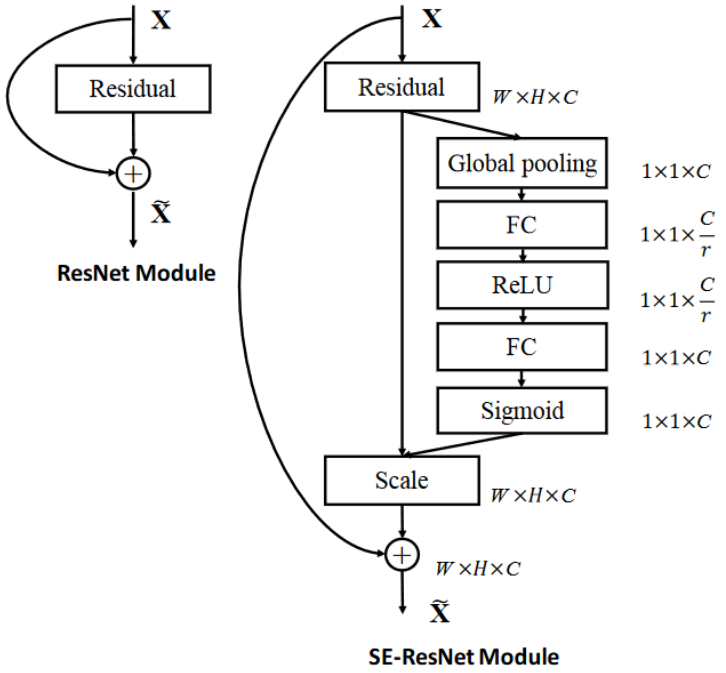
Image augmentation can be done easily in keras by adding image preprocessing layers at the beginning of the network. Several augmentation functions can be used jointly. Keras automatically deactivates these layers when not training.

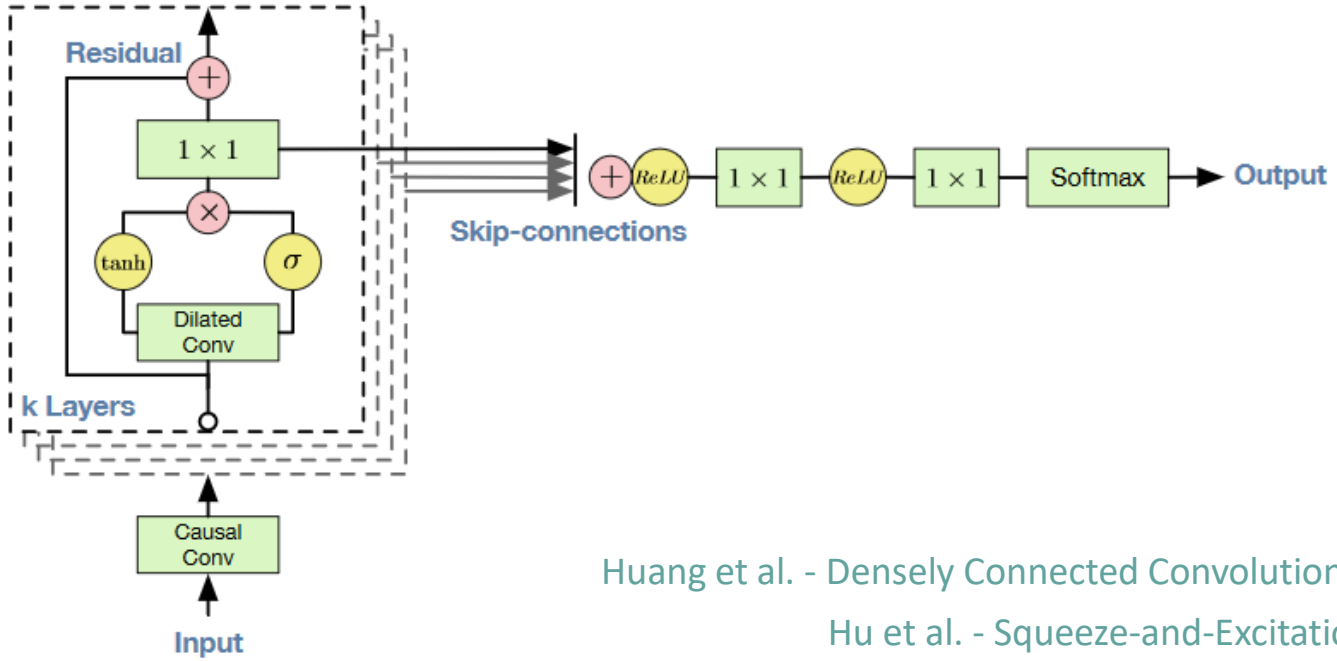
The random transformations will change for each batch.

Keras image preprocessing and augmentation layers - https://keras.io/api/layers/preprocessing_layers/image_preprocessing/

Block-based modelling





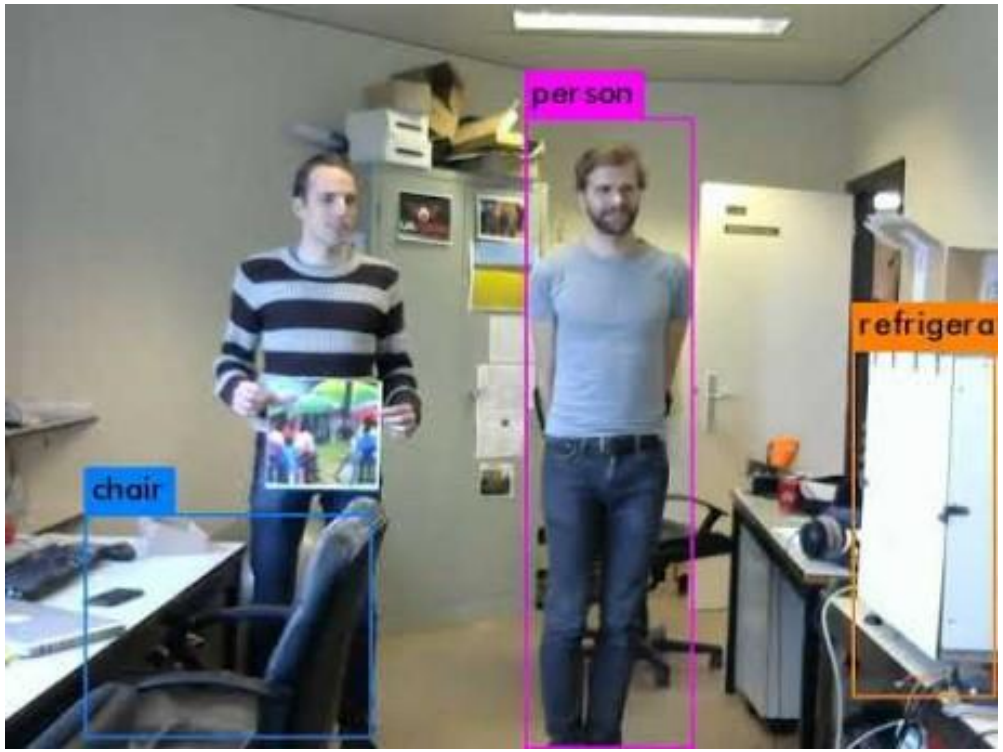


Huang et al. - Densely Connected Convolutional Networks

Hu et al. - Squeeze-and-Excitation Networks

Oorde et al – WaveNet: A Generative Model for Raw Audio

When networks fail



"When today's intelligent systems fail, they often fail spectacularly disgracefully without warning or explanation, leaving a user staring at an incoherent output, wondering why the system did what it did."

Selvaraju et al - Grad-CAM: Visual Explanations from Deep Networks via Gradient-based Localization

Visual explanations from the network

In order to understand what the model is doing, we can obtain visual explanations of the model decisions through the Grad-CAM method.

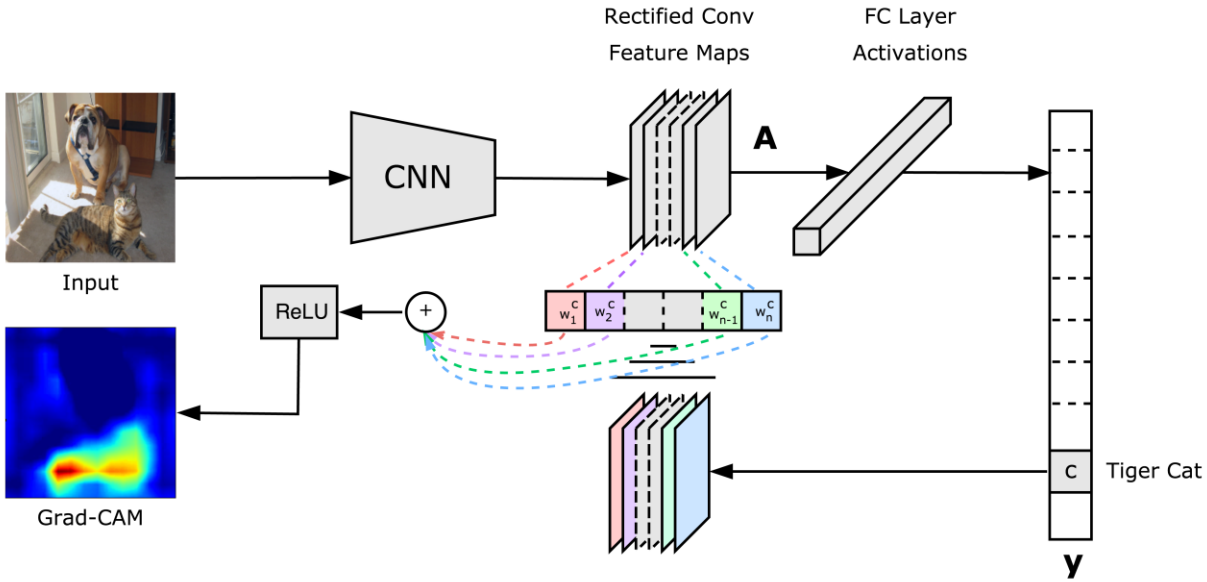
1. Compute score for class $\mathbf{c}$, before softmax: y^c
2. Define A_{ij}^k as an output value from the last convolutional layer: row i, column j, kernel k
3. Compute how sensitive is the score w.r.t the outputs of the last convolutional layer:

$$\frac{\partial y^c}{\partial A_{ij}^k}$$

4. Compute global relevance of each kernel k in the output layer:

$w_k^c = \frac{1}{Z}\sum_{i,j}\frac{\partial y^c}{\partial A_{ij}^k}$, with Z = number of pixels in A

5. The Grad-CAM score for each output of the convolution is $L_{Grad-CAM}^c = ReLU\left(\sum_k w_k^c A^k\right)$



(c) Grad-CAM 'Cat'

(i) Grad-CAM 'Dog'

Grad-CAM demo

launch binder

Afi Escuela