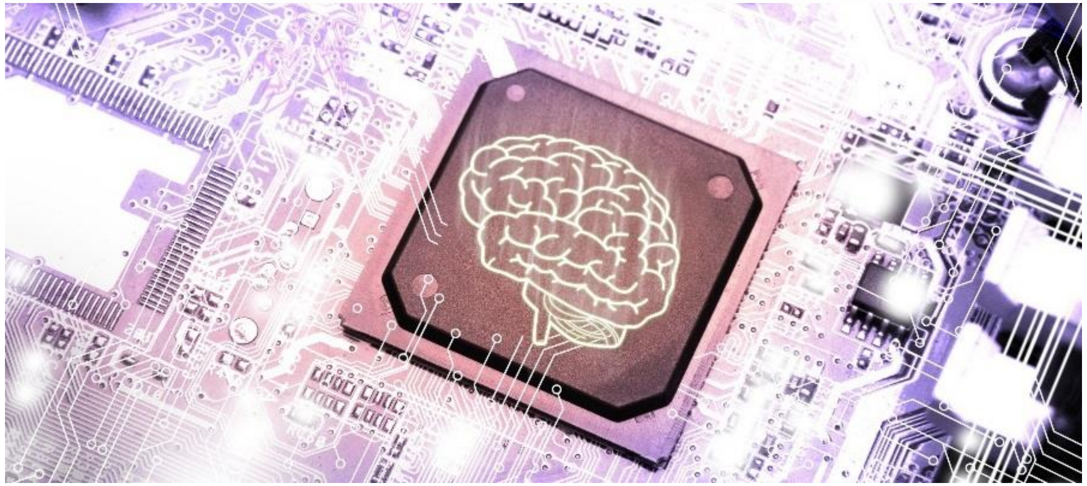


COURS - OPTIMISATION DE L'IMPLANTATION DE RÉSEAUX DE NEURONE

Olivier Bichler
CEA LIST
olivier.bichler@cea.fr

ARTIFICIAL INTELLIGENCE

Artificial Intelligence

“AI is whatever hasn't been done yet”

Very broad: understanding human speech, competing in strategic games, autonomous cars, intelligent routing, military simulations, interpreting complex data including images and videos...

Machine Learning

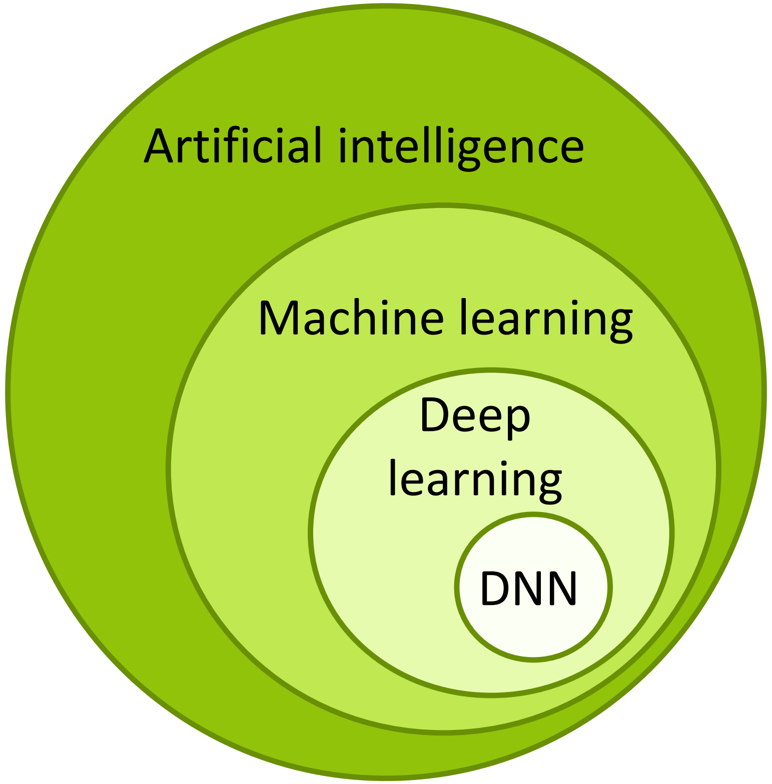
Algorithms that can learn from and make predictions on data: requires enough training data and a training algorithm

Deep Learning

Cascade of multiple nonlinear layers for feature extraction and transformation; learn multiple levels of representation

Deep Neural Networks

MLP, CNN, R-CNN, LSTM RNN…

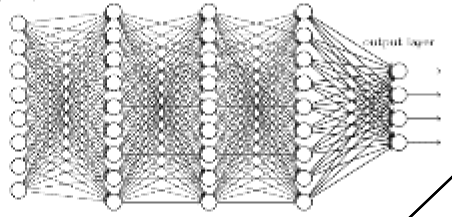


NEURAL COMPUTING & TRAINING

Days, weeks on multi-GPU server until correct accuracy
(topology, training set, parameters…)

Labeled databases → Machine learning algorithm → DNN model

Nvidia DGX-1
(8 Tesla P100)



training

prediction

New data → DNN trained model → prediction

"A car"

Low-latency inference *(TPU, FPGA, GPU, PNeuro…)*

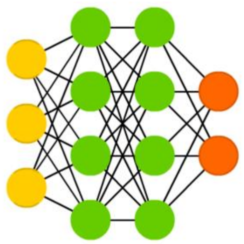
SUMMARY

1. General introduction
2. Models / topologies complexity
3. Convolution algorithms
4. Graph optimization
5. Quantization technics

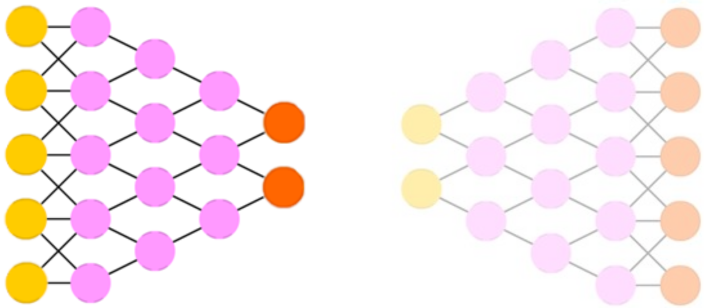
SUMMARY

NEURAL NETWORK TYPES AND MAIN PRIMITIVES

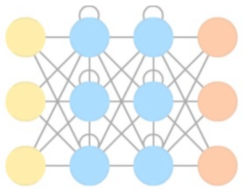
- **Fully connected (Fc)**



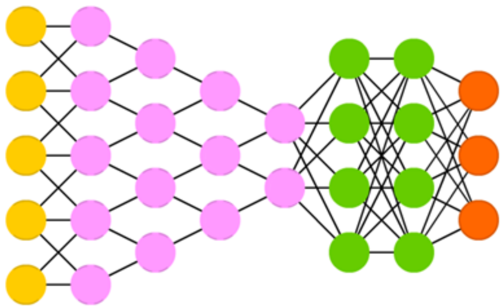
- **Fully conv. (Conv) / deconv.**

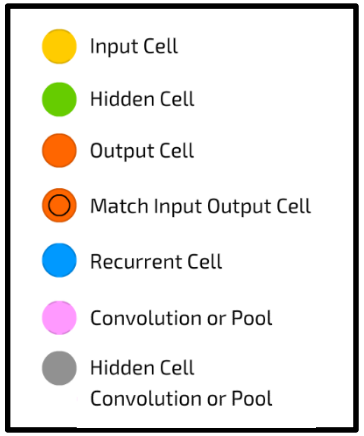


- **Recurrent NN**



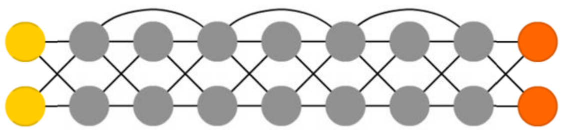
- **CNN with Conv+Fc**



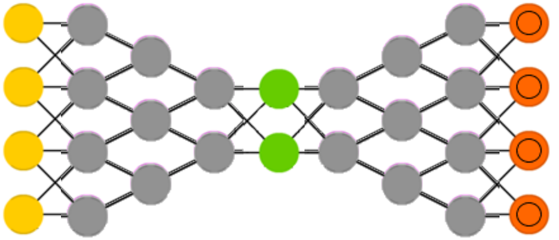


- ***More topologies:***

Residual network



Auto-encoder network



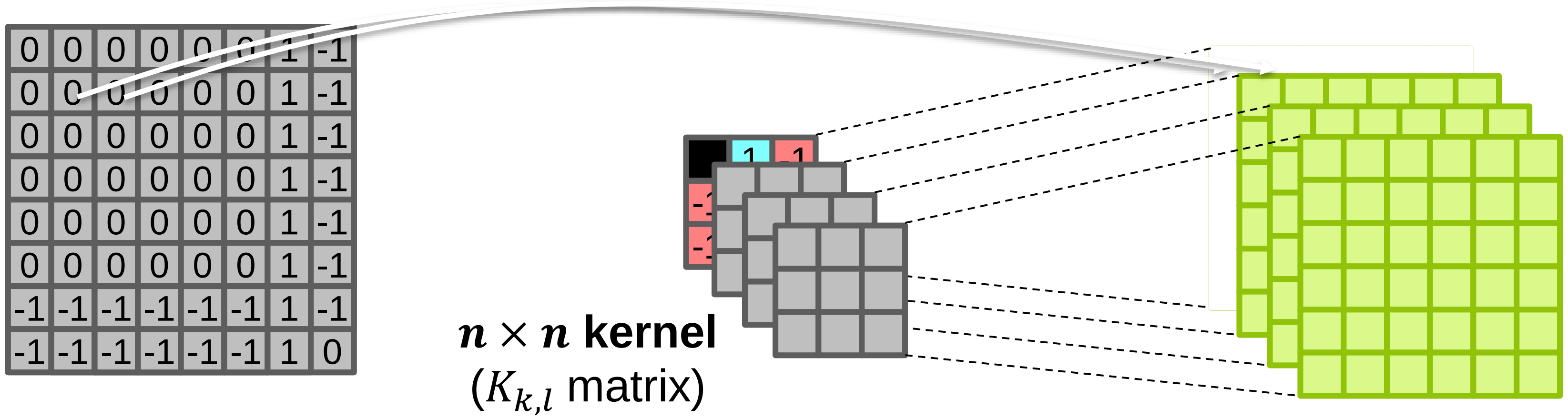
NEURAL NETWORK TYPES AND MAIN PRIMITIVES

- **CNN layer:**

Input map
($I_{i,j}$ matrix)

$$O_{i,j} = \tanh\left(\sum_{k=0}^{n-1}\sum_{l=0}^{n-1} I_{i+k,j+l}.K_{k,l}\right)$$

Output feature map
($O_{i,j}$ matrix)



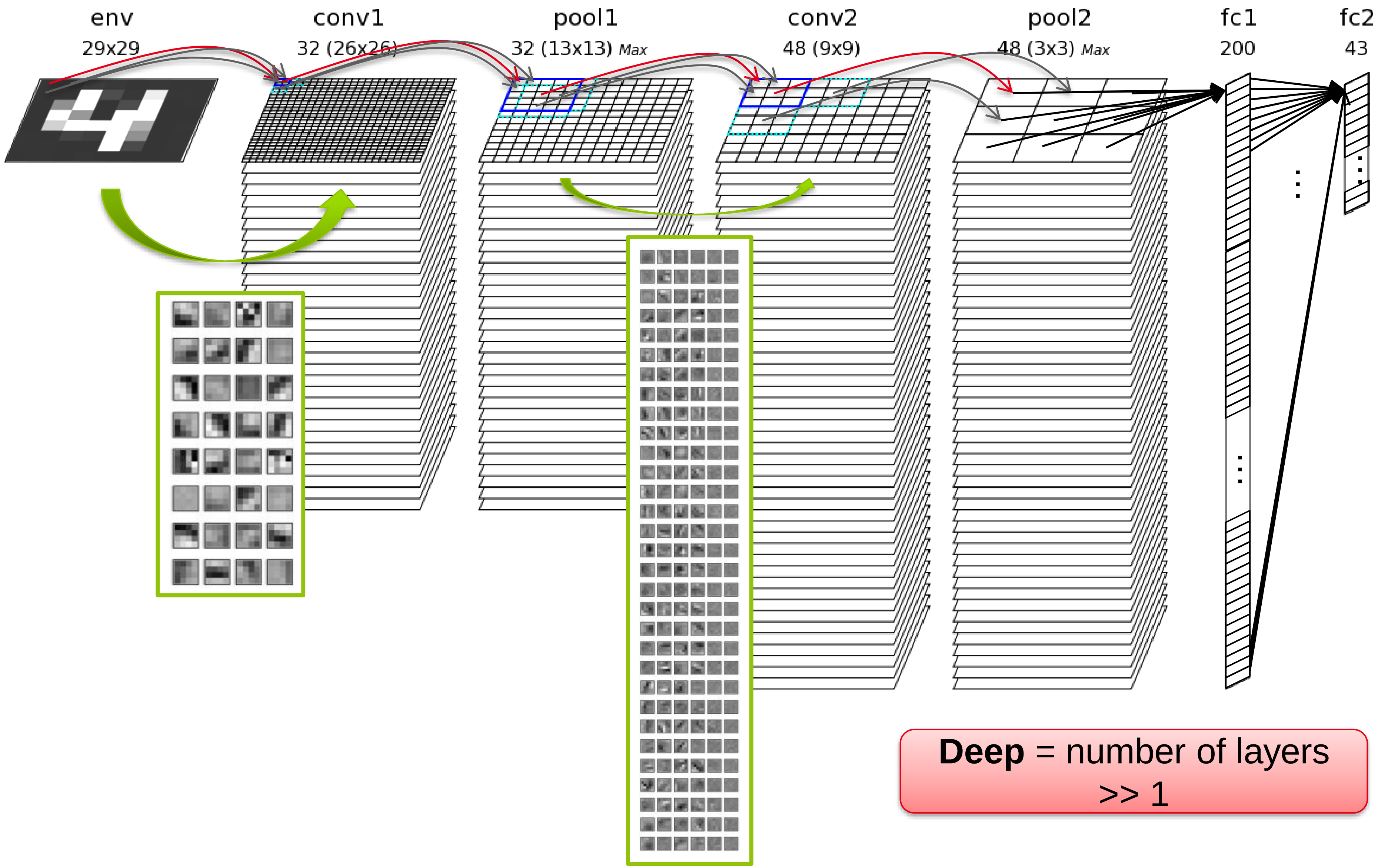
$\boldsymbol{n \times n}$ **kernel**
($K_{k,l}$ matrix)

Each kernel generates ≠ output feature maps

Convolution operation: $$O_{i,j} = \tanh\left(\sum_{k=0}^{n-1}\sum_{l=0}^{n-1} I_{i+k,j+l}.K_{k,l}\right)$$

Kernels are learned with gradient-descent algorithms
(classical back-propagation is very efficient!)

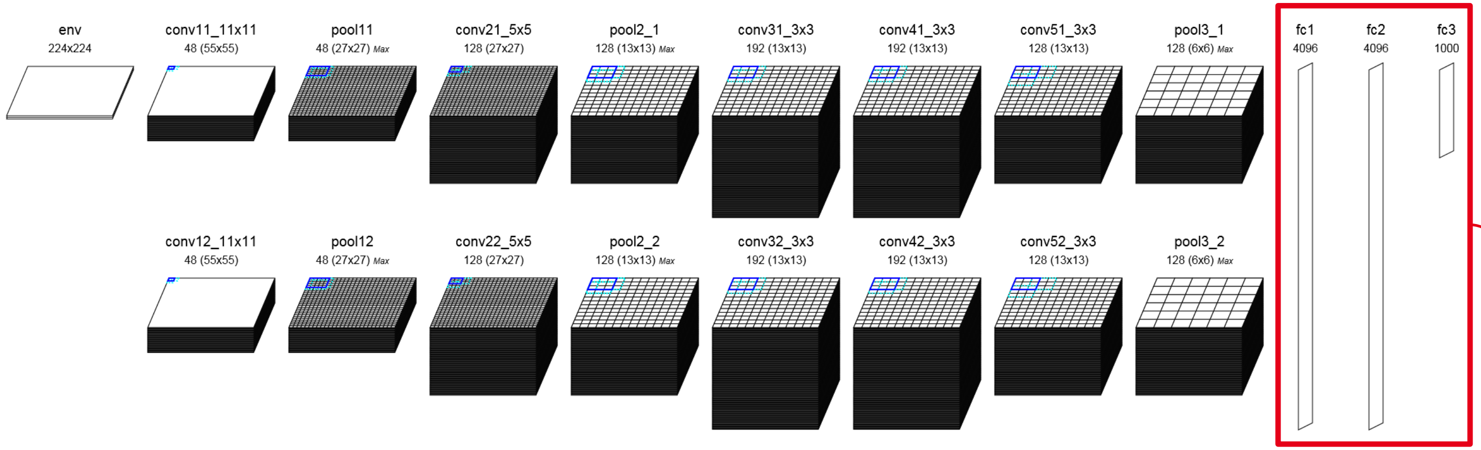
CONVOLUTIONAL NEURAL NETWORKS OVERVIEW



Deep = number of layers >> 1

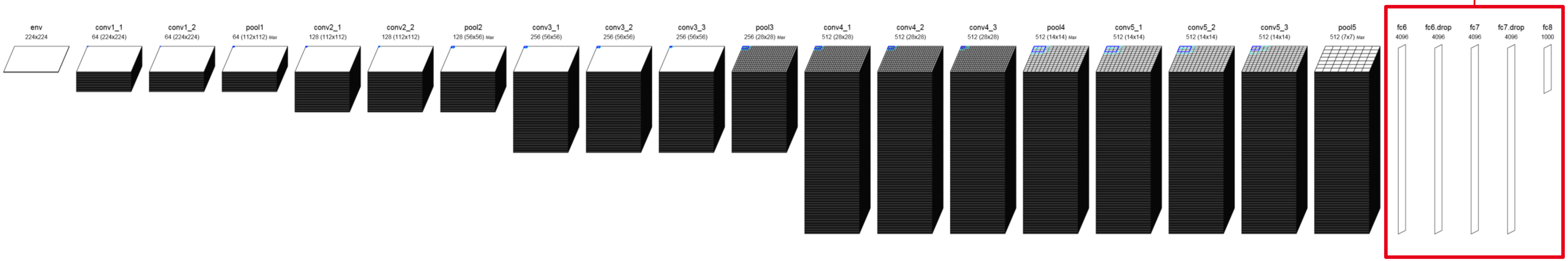
CONVOLUTIONAL NEURAL NETWORKS OVERVIEW

- **AlexNet** *(2012)*



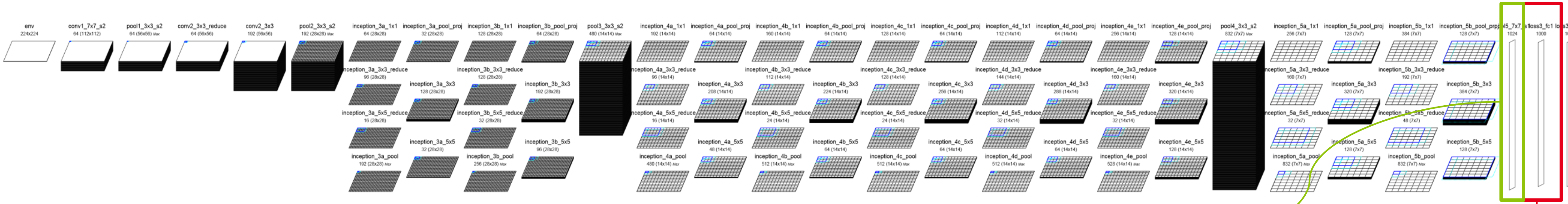
3 fully connected layers
fc1: 4,096 neurons
fc2: 4,096 neurons
fc3: 1,000 neurons

- **VGG-16** *(2014)*



CONVOLUTIONAL NEURAL NETWORKS OVERVIEW

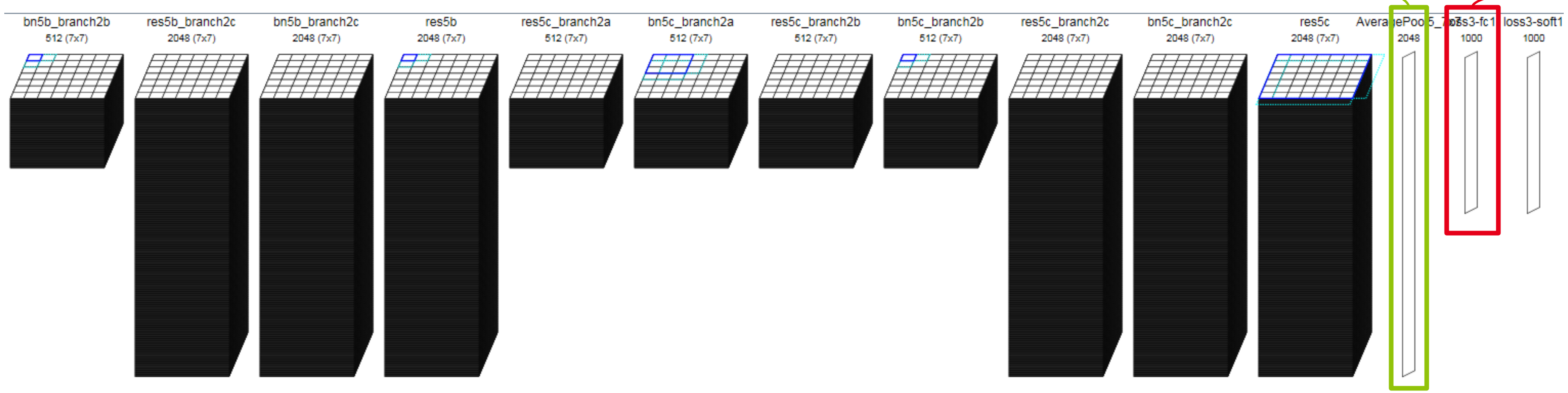
- **GoogleNet** *(2014)*



Average pooling used to reduce classifier dimension

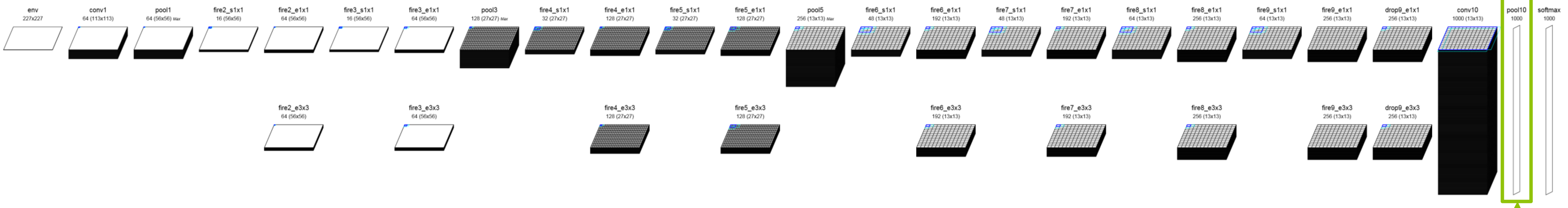
1 fully connected layer
fc1: 1,000 neurons

- **ResNet-50** *(2015)*



CONVOLUTIONAL NEURAL NETWORKS OVERVIEW

- **SqueezeNet** *(2016)*



Average pooling only

GoogleNet
24 layers
3,228k neurons
1 fully connected (1000)

AlexNet
8 layers
660k neurons
3 fully connected (4096-1000-1000)

2012

VGG-16
16 layers
13,565k neurons
3 fully connected (4096-1000-1000)

2014

ResNet-50
50 layers
10,589k neurons
1 fully connected (1000)

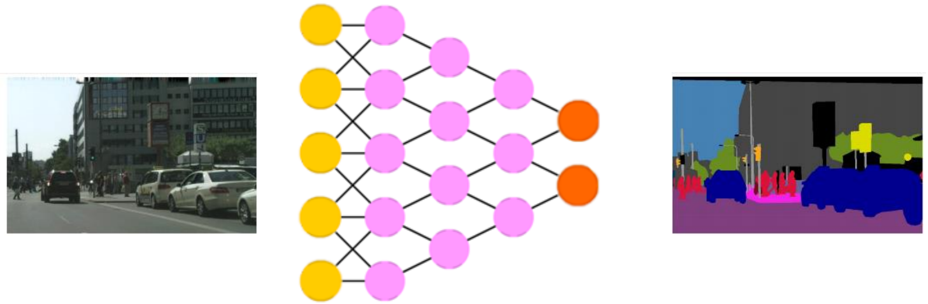
2015

SqueezeNet
20 layers
2,737k neurons
No fully connected layer

2016

COMPLEX NEURAL NETWORK SYSTEMS (1/3)

- **Combining and/or cascading multiple neural networks**

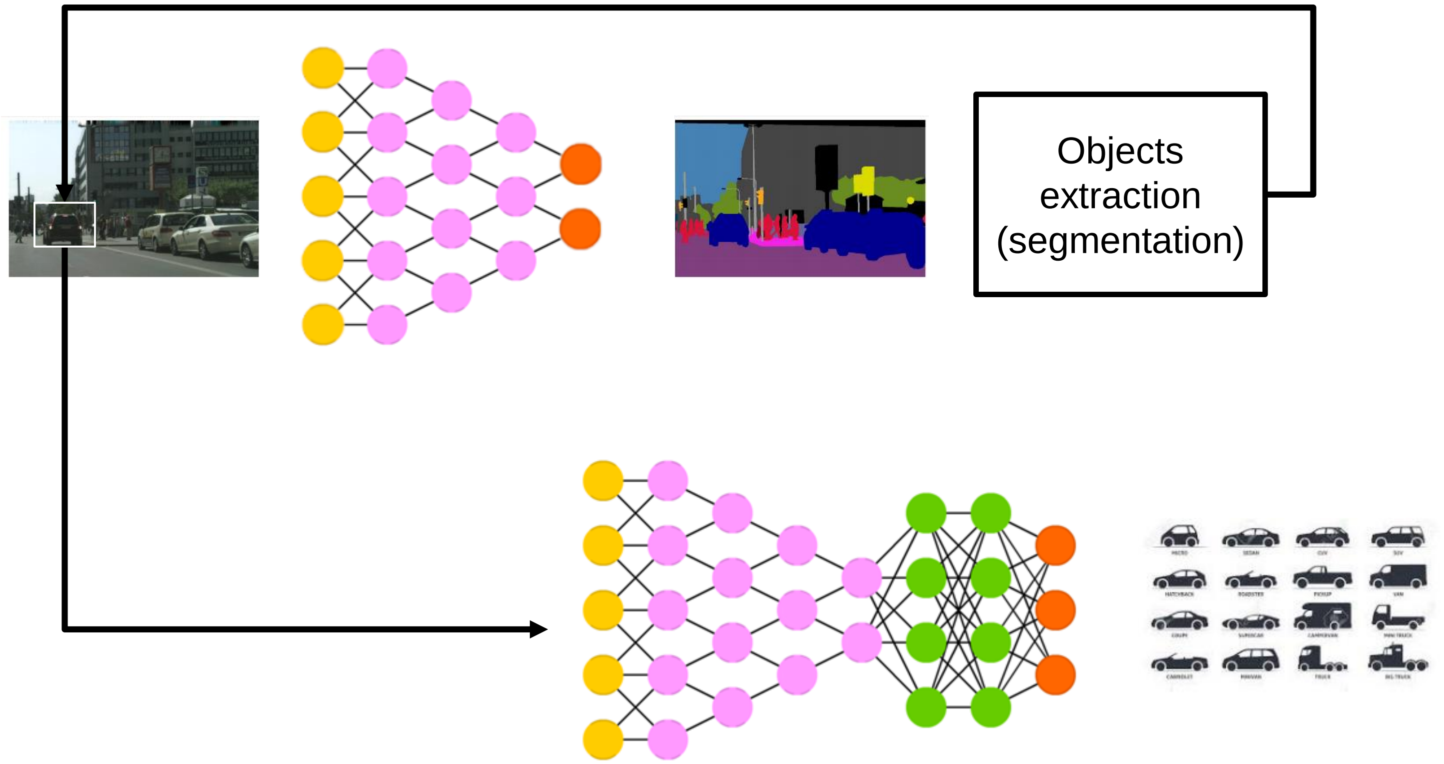


Objects
extraction
(segmentation)

1) Objects extraction:
using typically fully convolutional neural network

COMPLEX NEURAL NETWORK SYSTEMS (1/3)

- **Combining and/or cascading multiple neural networks**



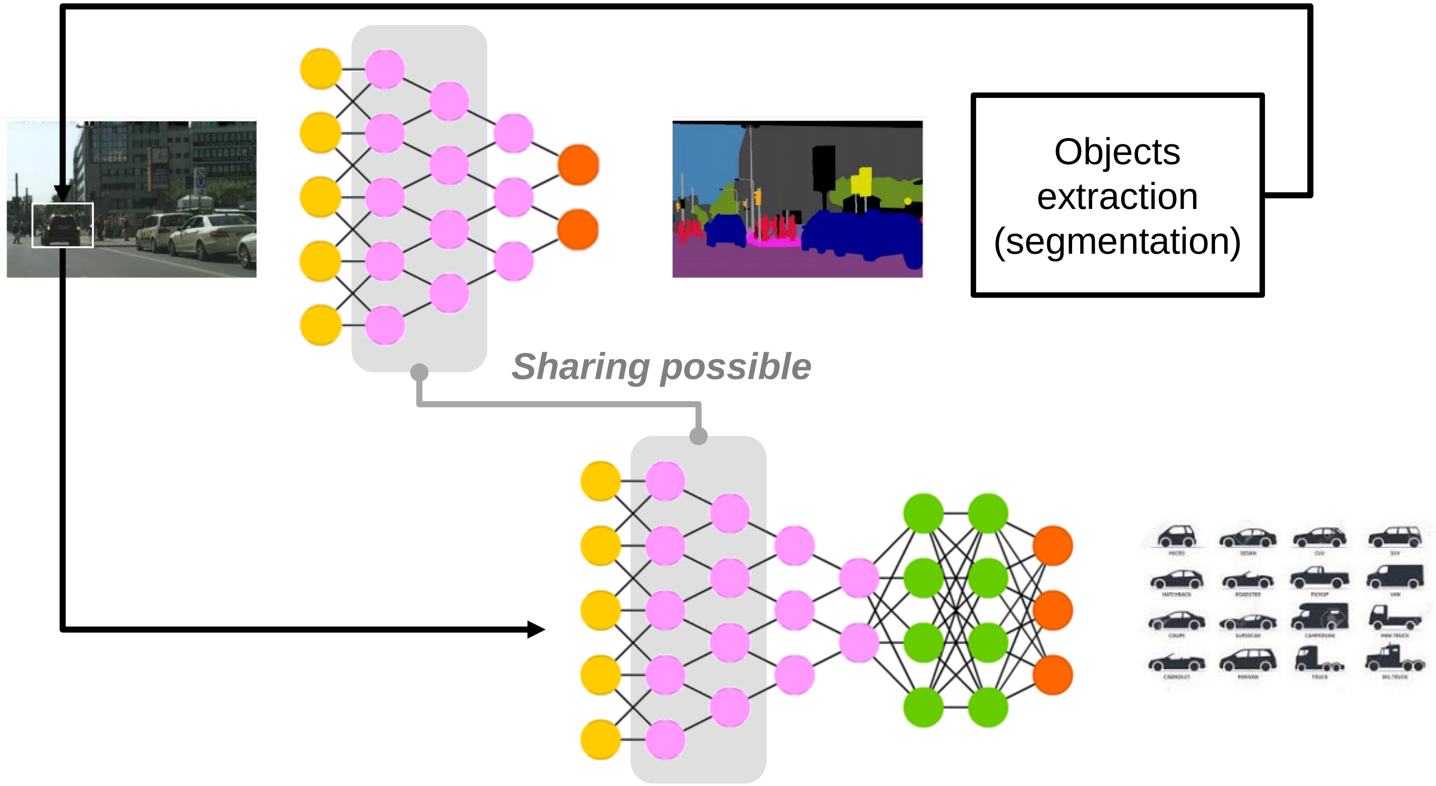
1) Objects extraction:
using typically fully convolutional neural network

+

2) Objects classification:
using typically CNN with softmax output

COMPLEX NEURAL NETWORK SYSTEMS (1/3)

- **Combining and/or cascading multiple neural networks**



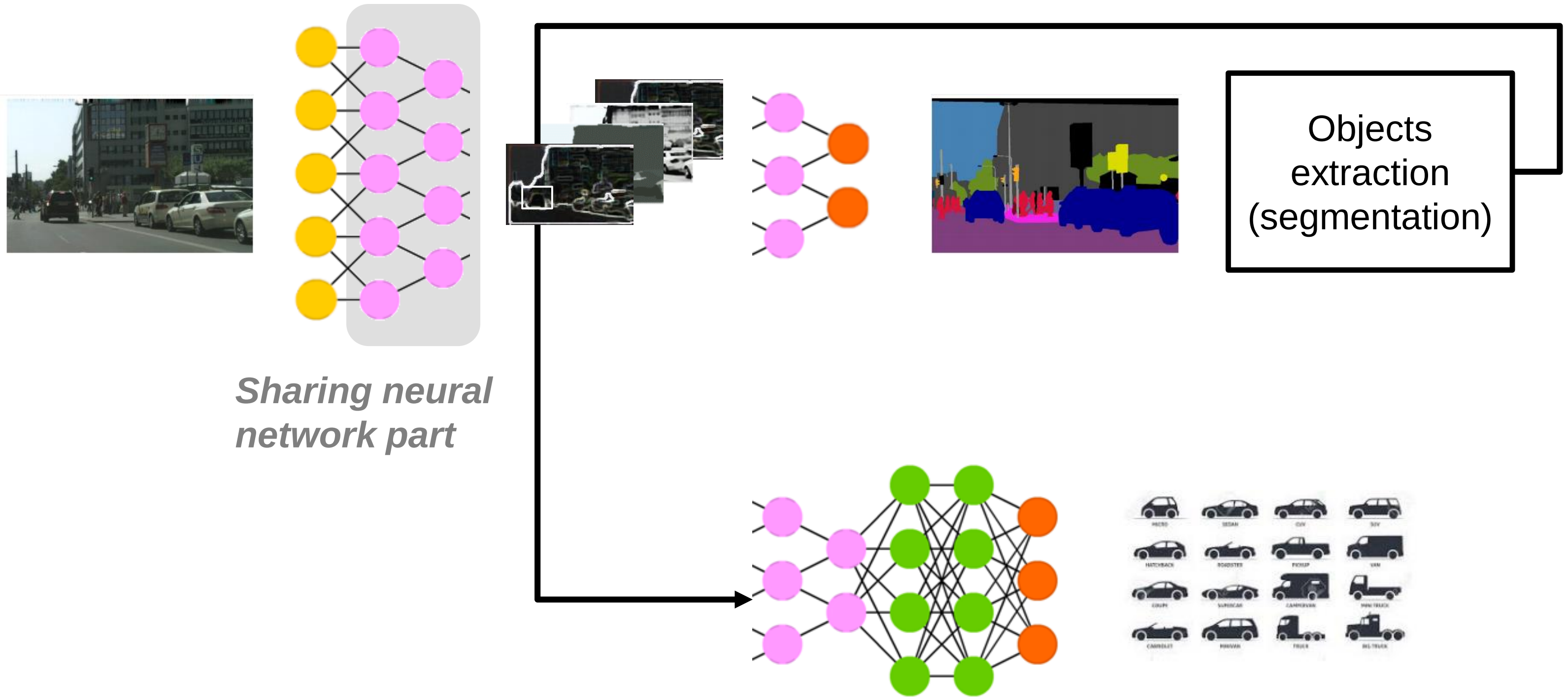
1) Objects extraction:
using typically fully convolutional neural network

+

2) Objects classification:
using typically CNN with softmax output

COMPLEX NEURAL NETWORK SYSTEMS (1/3)

- **Combining and/or cascading multiple neural networks**

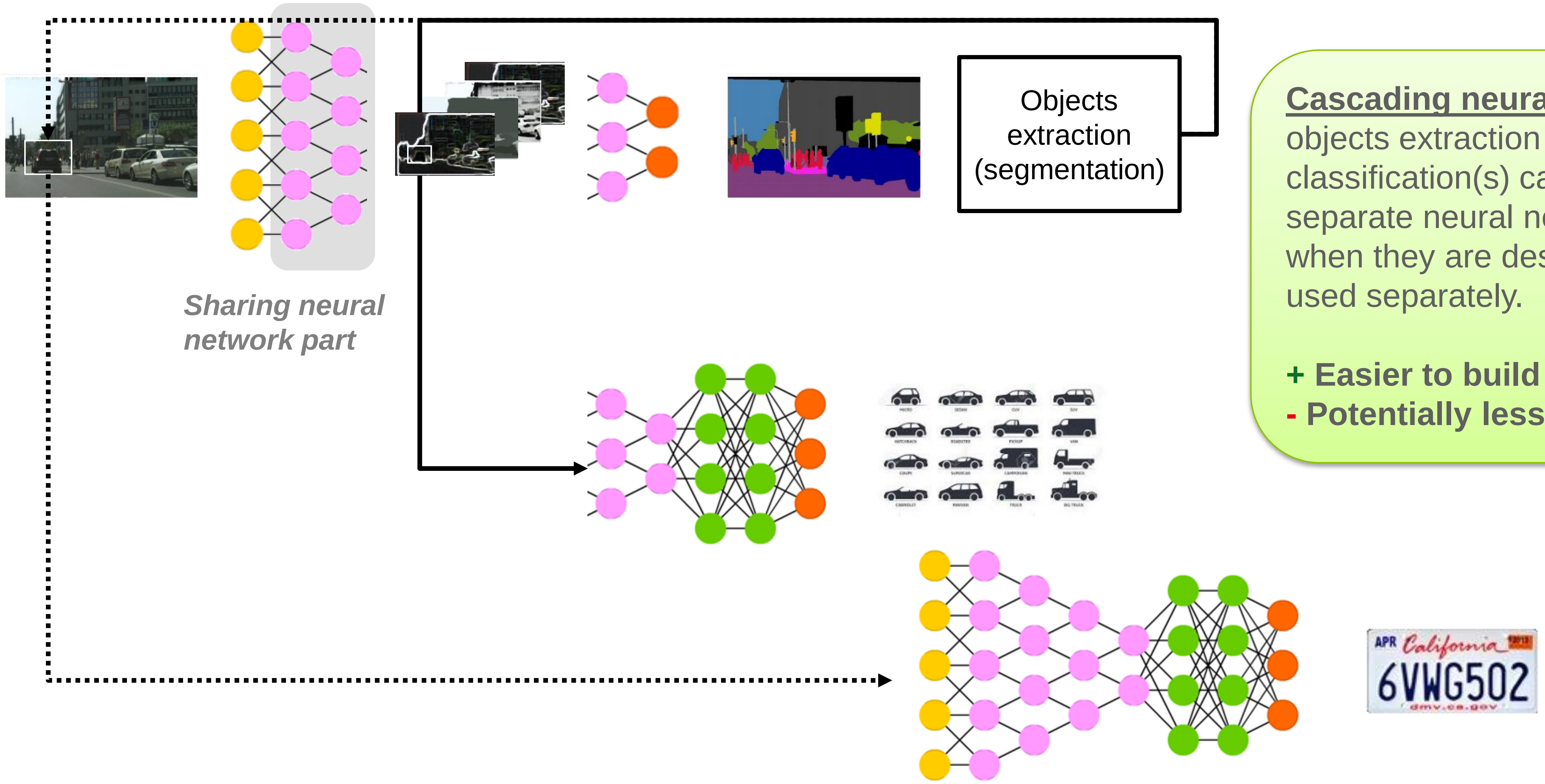


Combining neural networks: when several classification tasks essentially require the same features, they can be shared in a single network with different output branches to reduce computing complexity and memory footprint

Principle used in object detectors (YOLO, SSD, Faster-RCNN…)

COMPLEX NEURAL NETWORK SYSTEMS (1/3)

- **Combining and/or cascading multiple neural networks**

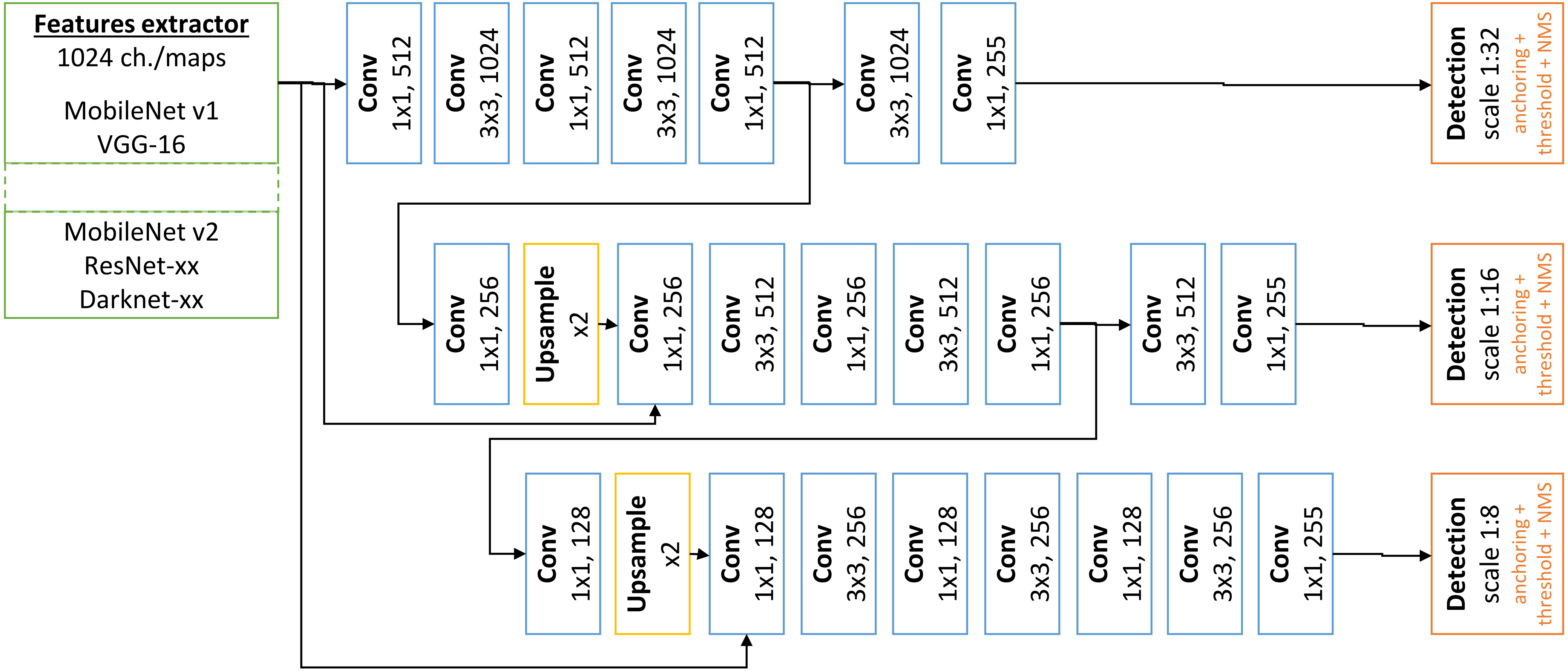


Cascading neural networks: objects extraction and classification(s) can be done in separate neural networks, when they are designed and/or used separately.

+ Easier to build and train
- Potentially less efficient

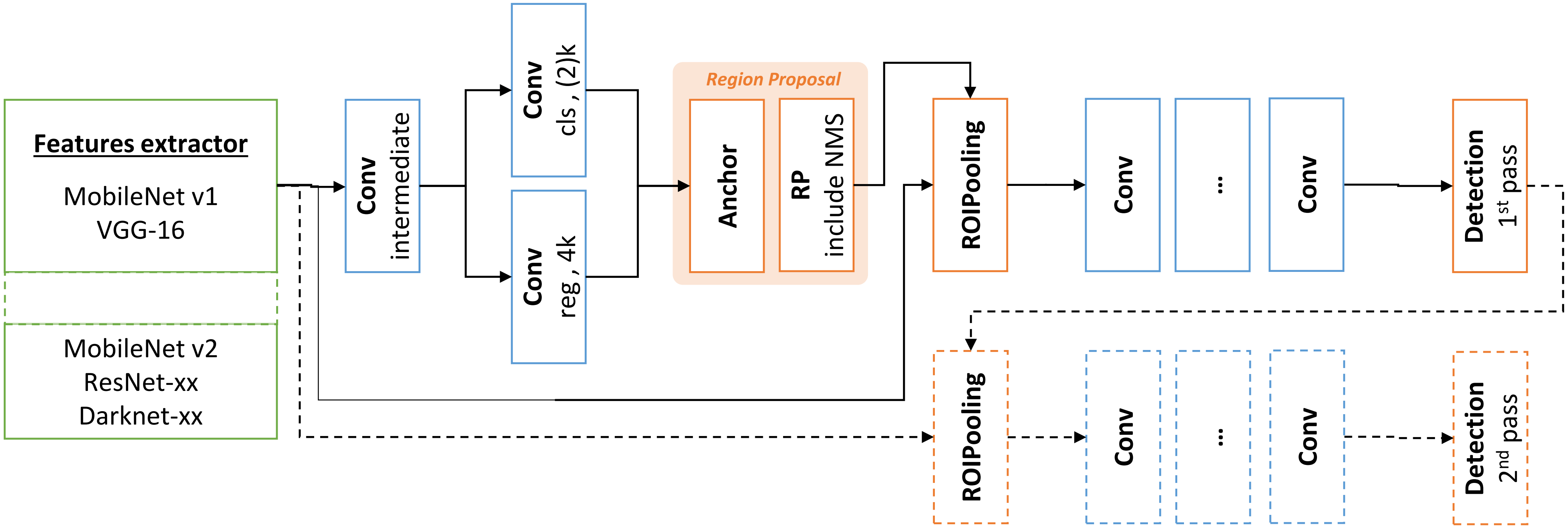
COMPLEX NEURAL NETWORK SYSTEMS (2/3)

- **YOLO v3 object detector network architecture**



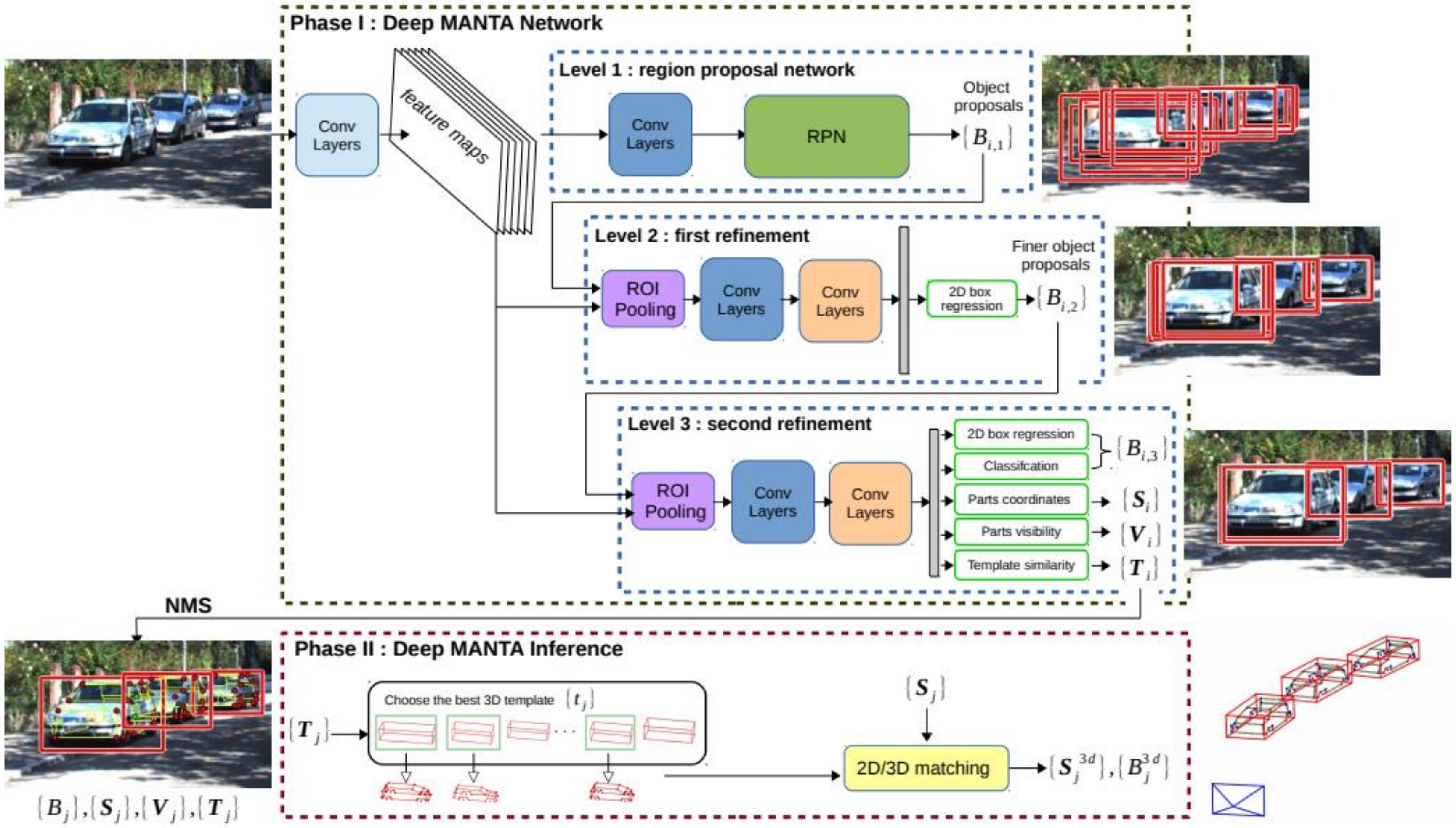
COMPLEX NEURAL NETWORK SYSTEMS (3/3)

- **Faster-RCNN object detector network architecture**



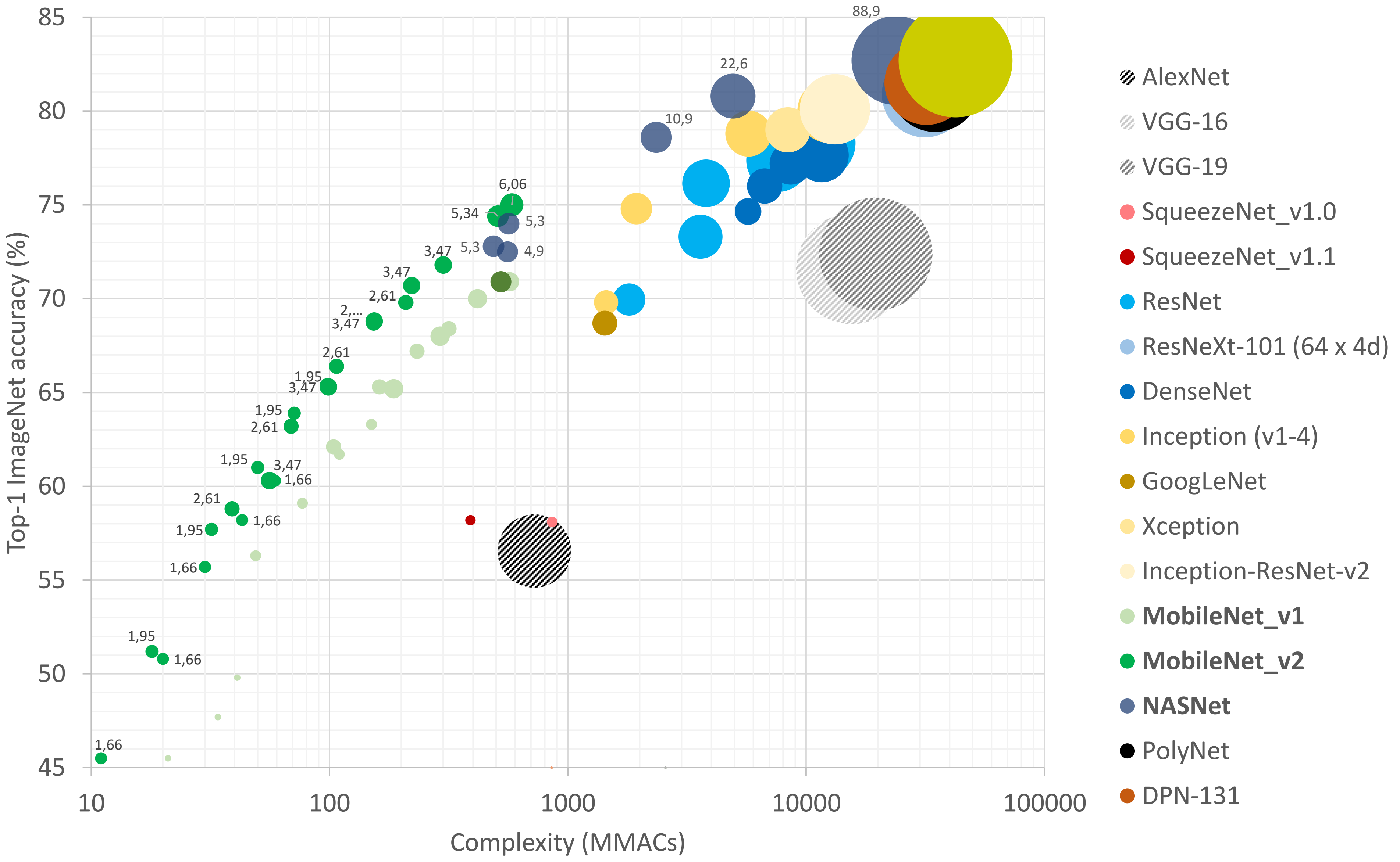
COMPLEX NEURAL NETWORK SYSTEMS (3/3)

- **Faster-RCNN applied to ADAS:**



MODELS / TOPOLOGIES COMPLEXITY

CONVOLUTIONAL NEURAL NETWORKS OVERVIEW



RESNET

- **Main innovation: use residual learning, thanks to identity « shortcuts »**

"We hypothesize that it is easier to optimize the residual mapping than to optimize the original, unreferenced mapping. To the extreme, if an identity mapping were optimal, it would be easier to push the residual to zero than to fit an identity mapping by a stack of nonlinear layers."

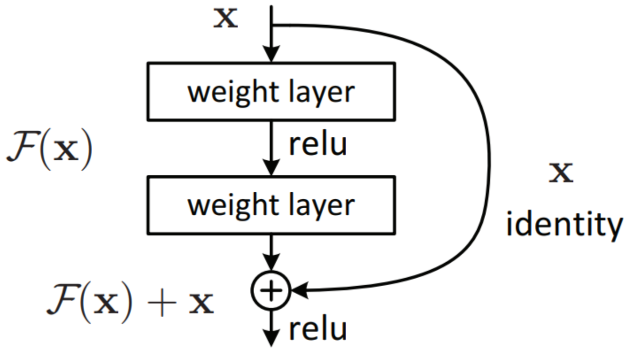


Figure 2. Residual learning: a building block.

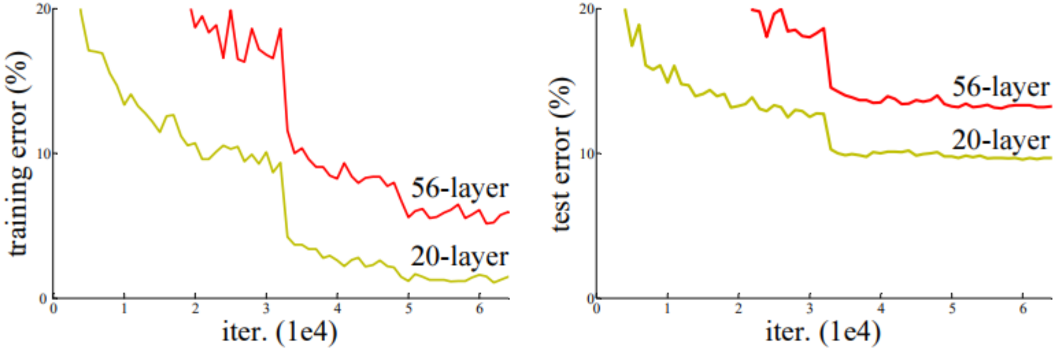
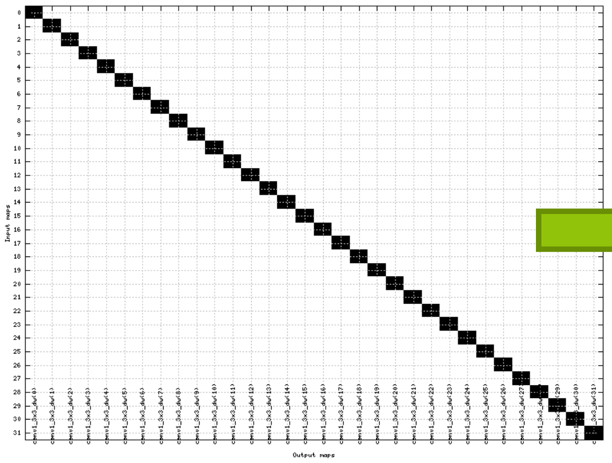


Figure 1. Training error (left) and test error (right) on CIFAR-10 with 20-layer and 56-layer "plain" networks. The deeper network has higher training error, and thus test error. Similar phenomena on ImageNet is presented in Fig. 4.

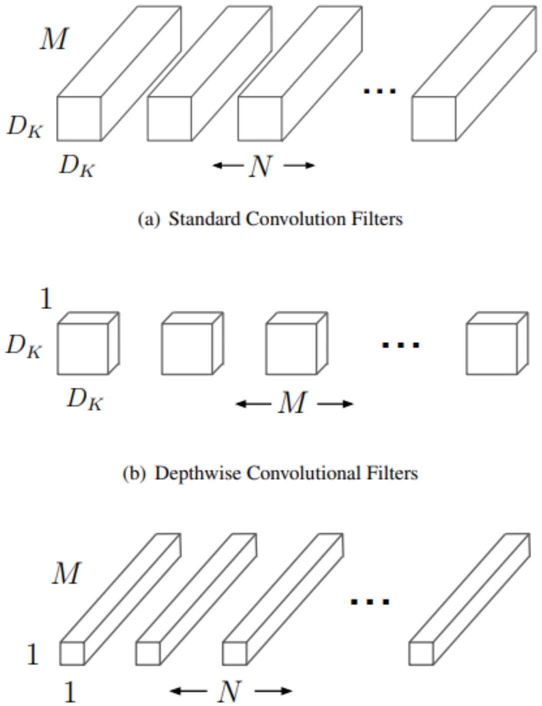
MOBILENET V1

- **Main innovation: use Deepwise Separable Convolution**
- **Result: same accuracy on ImageNet than AlexNet with x15 less parameters x1.3 less computation**

M = INPUT CHANNELS



N = OUTPUT CHANNELS



(a) Standard Convolution Filters

(b) Depthwise Convolutional Filters

(c) 1 × 1 Convolutional Filters called Pointwise Convolution in the co[ntext] of Depthwise Separable Convolution

Figure 2. The standard convolutional filters in (a) are replaced [by] two layers: depthwise convolution in (b) and pointwise convo[lu]tion in (c) to build a depthwise separable filter.

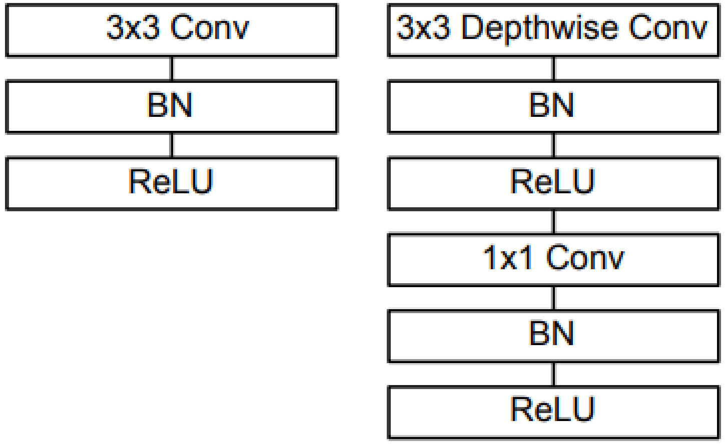
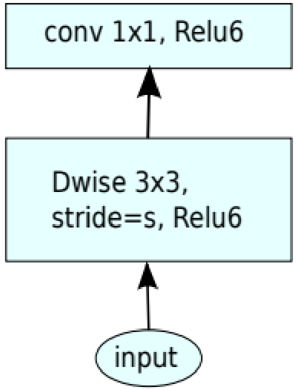


Figure 3. Left: Standard convolutional layer with batchnorm and ReLU. Right: Depthwise Separable convolutions with Depthwise and Pointwise layers followed by batchnorm and ReLU.

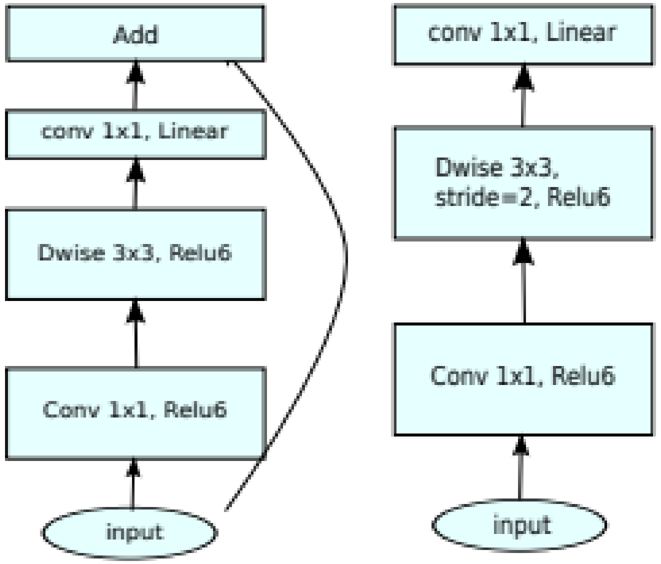
MOBILENET V2

- **Main innovation: combine Deepwise Separable Convolution with inverted residual blocks (« bootleneck »)**

- **Inverted residual: same principle than residual, but a little more memory efficient**



(b) MobileNet[27]



Stride=1 block

Stride=2 block

(d) Mobilenet V2

Input	Operator	t	c	n	s
$224^2 \times 3$	conv2d	-	32	1	2
$112^2 \times 32$	bottleneck	1	16	1	1
$112^2 \times 16$	bottleneck	6	24	2	2
$56^2 \times 24$	bottleneck	6	32	3	2
$28^2 \times 32$	bottleneck	6	64	4	2
$14^2 \times 64$	bottleneck	6	96	3	1
$14^2 \times 96$	bottleneck	6	160	3	2
$7^2 \times 160$	bottleneck	6	320	1	1
$7^2 \times 320$	conv2d 1x1	-	1280	1	1
$7^2 \times 1280$	avgpool 7x7	-	-	1	-
$1 \times 1 \times 1280$	conv2d 1x1	-	k	-	

Table 2: MobileNetV2 : Each line describes a sequence of 1 or more identical (modulo stride) layers, repeated n times. All layers in the same sequence have the same number c of output channels. The first layer of each sequence has a stride s and all others use stride 1. All spatial convolutions use 3×3 kernels. The expansion factor t is always applied to the input size as described in Table 1.

MODELS / TOPOLOGIES COMPLEXITY

EFFICIENTNET

- **Main innovation: compound scaling method for neural architecture search**

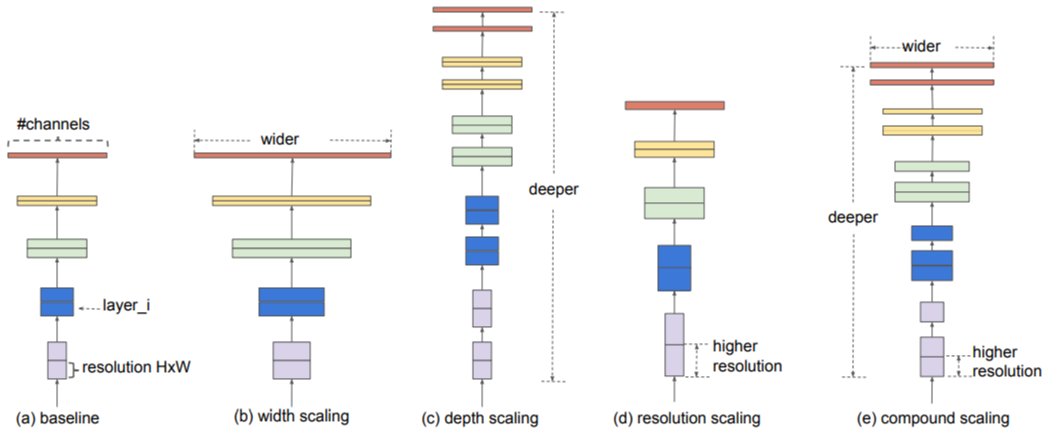


Figure 2. **Model Scaling.** (a) is a baseline network example; (b)-(d) are conventional scaling that only increases one dimension of network width, depth, or resolution. (e) is our proposed compound scaling method that uniformly scales all three dimensions with a fixed ratio.

- **Use "Swish" activation function:**

$$f(x) = x.sigmoid(\beta.x)$$

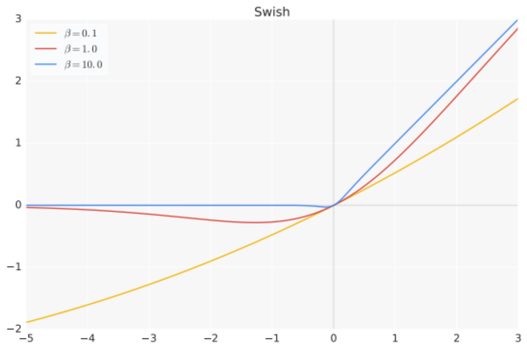
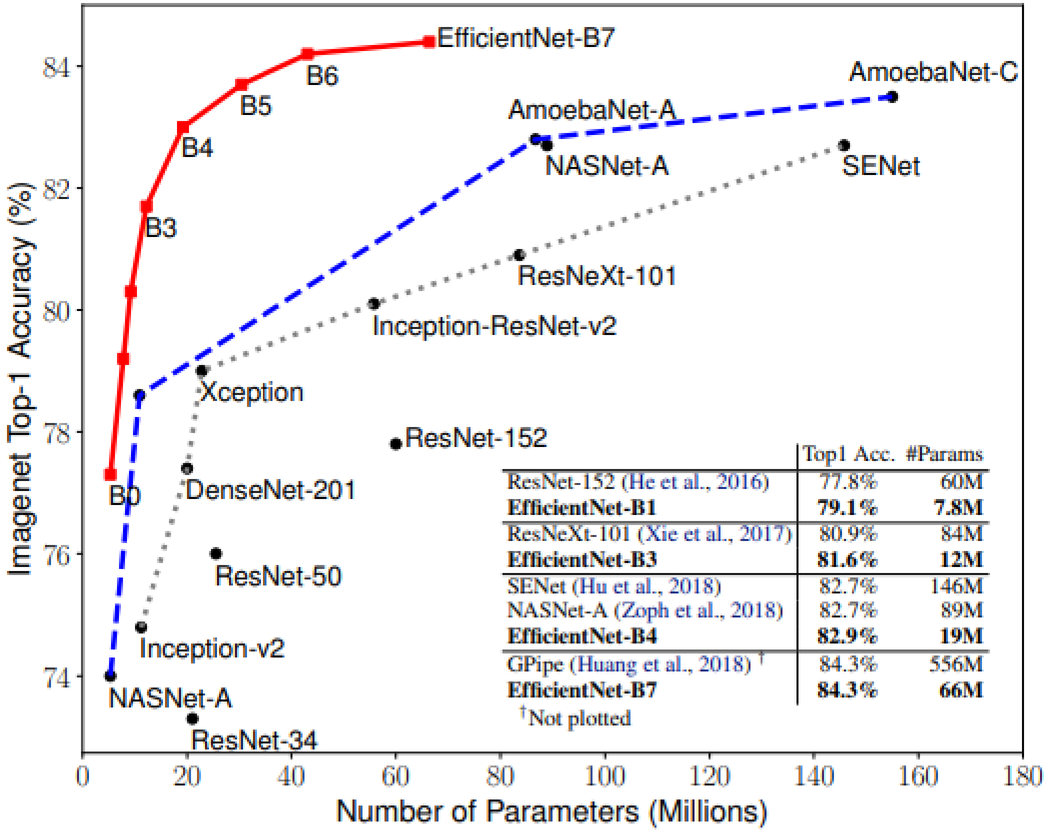


Figure 4: The Swish activation function.



	Top1 Acc.	#Params
ResNet-152 (He et al., 2016)	77.8%	60M
EfficientNet-B1	**79.1%**	**7.8M**
ResNeXt-101 (Xie et al., 2017)	80.9%	84M
EfficientNet-B3	**81.6%**	**12M**
SENet (Hu et al., 2018)	82.7%	146M
NASNet-A (Zoph et al., 2018)	82.7%	89M
EfficientNet-B4	**82.9%**	**19M**
GPipe (Huang et al., 2018) †	84.3%	556M
EfficientNet-B7	**84.3%**	**66M**

† Not plotted

Figure 1. **Model Size vs. ImageNet Accuracy.** All numbers are for single-crop, single-model. Our EfficientNets significantly outperform other ConvNets. In particular, EfficientNet-B7 achieves new state-of-the-art 84.3% top-1 accuracy but being 8.4x smaller and 6.1x faster than GPipe. EfficientNet-B1 is 7.6x smaller and 5.7x faster than ResNet-152. Details are in Table 2 and 4.

TAKE-AWAY MESSAGE

Before starting to optimize the implementation, choose the right topology!

AlexNet and VGG are NO GO!

CONVOLUTION ALGORITHMS

1. General introduction
2. Models / topologies complexity
3. **Convolution algorithms**
   - Direct convolution
   - Matrix multiplication (GEMM)
   - Winograd for 3x3 convolution
   - Other algorithms
4. Graph optimization
5. Quantization technics

CONVOLUTION ALGORITHMS

DIRECT CONVOLUTION

```
for (int oy = 0; oy < OUTPUTS_HEIGHT; ++oy) {
    const int syMin = max(PADDING_Y - (oy * STRIDE_Y), 0);
    const int syMax = clamp(CHANNELS_HEIGHT + PADDING_Y - (oy * STRIDE_Y), 0, KERNEL_HEIGHT);
    const int iy = (oy * STRIDE_Y) - PADDING_Y;

    for (int ox = 0; ox < OUTPUTS_WIDTH; ++ox) {
        const int sxMin = max(PADDING_X - (ox * STRIDE_X), 0);
        const int sxMax = clamp(CHANNELS_WIDTH + PADDING_X - (ox * STRIDE_X), 0, KERNEL_WIDTH);
        const int ix = (ox * STRIDE_X) - PADDING_X;

N       for (int output = 0; output < NB_OUTPUTS; ++output) {
            const int oPos = (ox + OUTPUTS_WIDTH * oy);
            const int oOffset = NB_OUTPUTS * oPos;

            SUM_T weightedSum = biasses[output];

H           for (int sy = syMin; sy < syMax; ++sy) {
                if (sy >= syMax - syMin)
                    break;

                const int iPos = ((sxMin + ix) + CHANNELS_WIDTH * (iy + syMin + sy));
                const int iOffset = NB_CHANNELS * iPos;
                const int wOffset = NB_CHANNELS * (sxMin + KERNEL_WIDTH * (syMin + sy + KERNEL_HEIGHT * output));

W               for (int sx = 0; sx < KERNEL_WIDTH; ++sx) {
                    if (sx >= sxMax - sxMin)
                        break;

C                   for (int ch = 0; ch < NB_CHANNELS; ++ch)
                        weightedSum += weights[wOffset + sx * NB_CHANNELS] * inputs[iOffset + sx * NB_CHANNELS];
                }
            }

            outputs[oOffset + output] = ACTIVATION(weightedSum);
        }
    }
}
```

These loops can be merged in any order
➔ Parallelization possible

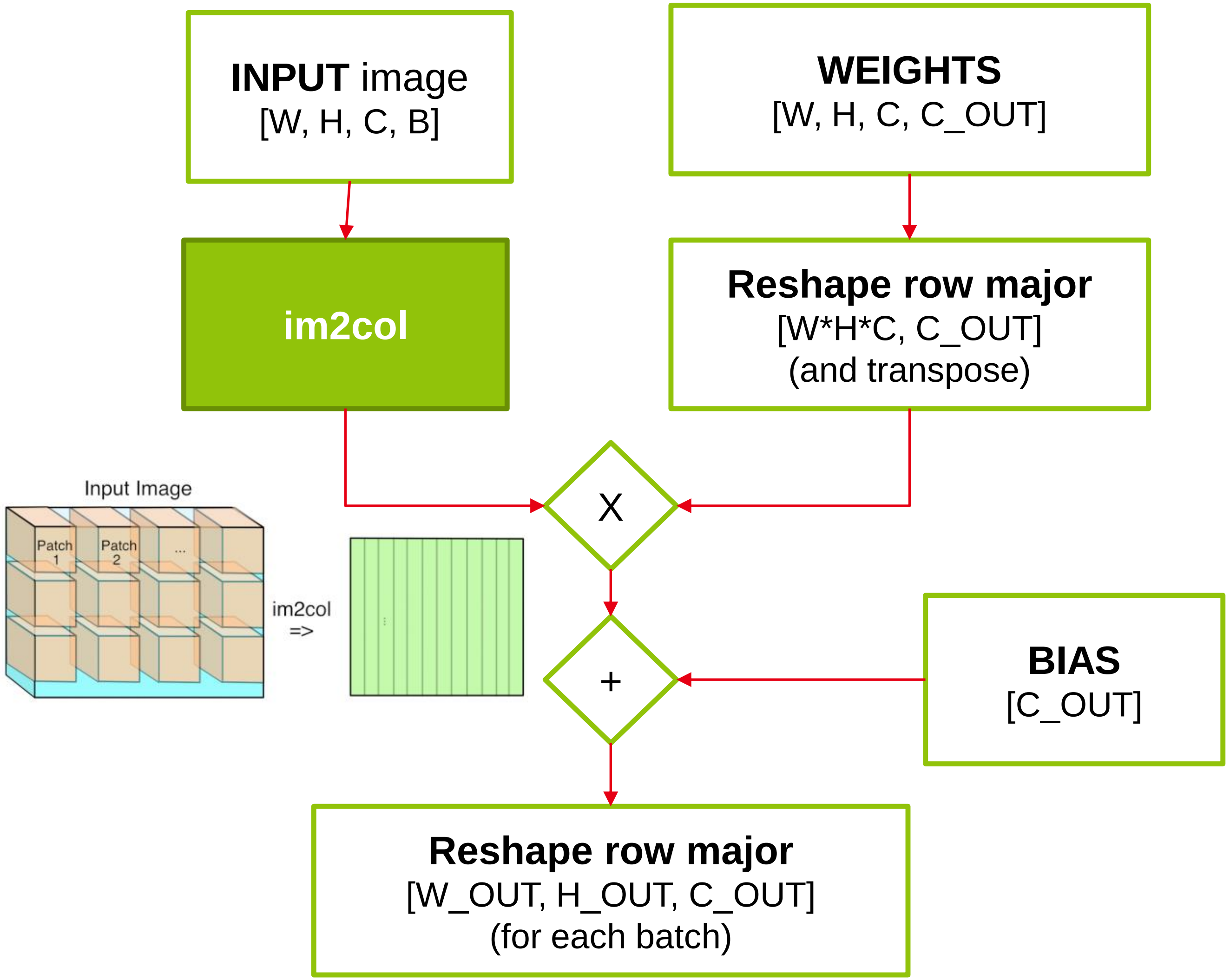
Contiguous data in memory
➔ SIMD instructions possible

MATRIX MULTIPLICATION (GEMM)

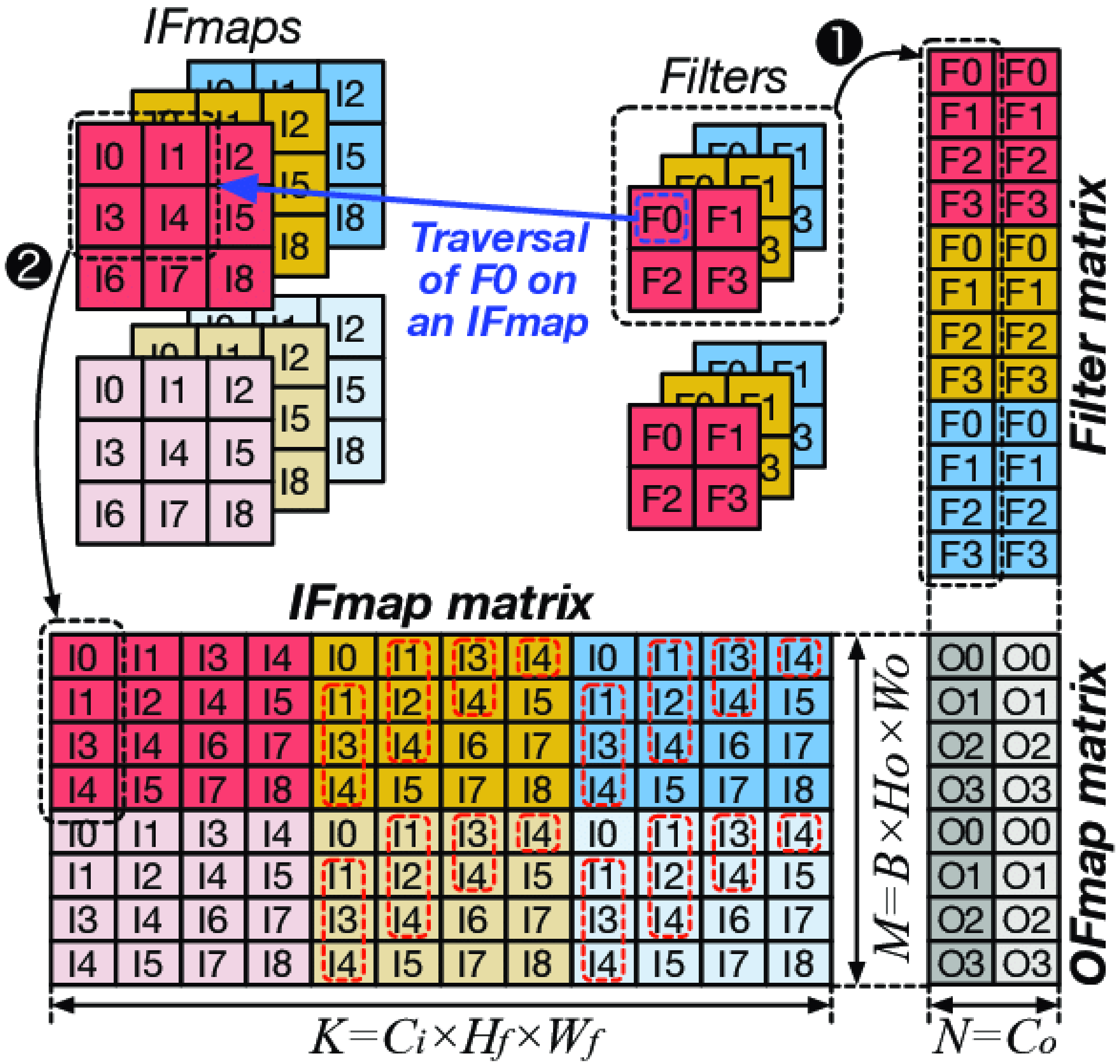
- **Direct convolution:**
  - No memory overhead
  - **But**, poor usage of vectorization instructions (MMX, SSE, …)

- **Why use GEMM for convolution?**
  - Benefit from highly optimized libraries and processor instructions that have been developed for decades for this purpose
  - Up to x100 speed-up!
  - Higher gain with large number of filters and/or large batches

- **Need to rearrange the input: im2col**
- **Memory overhead: values are duplicated in the resulting matrix (depends on stride)**



MATRIX MULTIPLICATION (GEMM) – IM2COL PRINCIPLE



WINOGRAD FOR 3X3 CONVOLUTION

- **Simple example with a (1D) image of size [4,1] and (1D) convolution kernel [3,1]:**

$$I = [1\ 2\ 3\ 4],\ F = [-1\ \ -2\ \ -3]$$

- **Using im2col gives:**

$$I = \begin{bmatrix} 1 & 2 & 3 \\ 2 & 3 & 4 \end{bmatrix}$$

- **Wingrad idea: express the result of the matrix multiplication as:**

$$\boldsymbol{O} = \begin{bmatrix} 1 & 2 & 3 \\ 2 & 3 & 4 \end{bmatrix} . \begin{bmatrix} -1 \\ -2 \\ -3 \end{bmatrix} = \begin{bmatrix} m_1 + m_2 + m_3 \\ m_2 - m_3 - m_4 \end{bmatrix}$$

- **Let's generalize and try to solve this equation:**

$$\boldsymbol{O} = \begin{bmatrix} d_0 & d_1 & d_2 \\ d_1 & d_2 & d_3 \end{bmatrix} . \begin{bmatrix} w_0 \\ w_1 \\ w_2 \end{bmatrix} = \begin{bmatrix} m_1 + m_2 + m_3 \\ m_2 - m_3 - m_4 \end{bmatrix}$$

$$m_1 = (d_0 - d_2).w_0 \qquad m_2 = (d_1 + d_2).\boxed{\frac{w_0+w_1+w_2}{2}}$$

$$m_4 = (d_1 - d_3).w_2 \qquad m_3 = (d_2 - d_1).\boxed{\frac{w_0-w_1-w_2}{2}}$$

Only kernel parameters, can be pre-computed!

➔ 4 MULT instead of 6 MULT. Can be generalized for any input size and 2D 3x3 kernels.

OTHER ALGORITHMS

- **Extract of NVidia's CuDNN library:**

`CUDNN_CONVOLUTION_FWD_ALGO_IMPLICIT_GEMM`
: This algorithm expresses the convolution as a matrix product without actually explicitly forming the matrix that holds the input tensor data.

`CUDNN_CONVOLUTION_FWD_ALGO_IMPLICIT_PRECOMP_GEMM`
: This algorithm expresses convolution as a matrix product without actually explicitly forming the matrix that holds the input tensor data, but still needs some memory workspace to precompute some indices in order to facilitate the implicit construction of the matrix that holds the input tensor data.

`CUDNN_CONVOLUTION_FWD_ALGO_GEMM`
: This algorithm expresses the convolution as an explicit matrix product. A significant memory workspace is needed to store the matrix that holds the input tensor data.

`CUDNN_CONVOLUTION_FWD_ALGO_DIRECT`
: This algorithm expresses the convolution as a direct convolution (for example, without implicitly or explicitly doing a matrix multiplication).

`CUDNN_CONVOLUTION_FWD_ALGO_FFT`
: This algorithm uses the Fast-Fourier Transform approach to compute the convolution. A significant memory workspace is needed to store intermediate results.

`CUDNN_CONVOLUTION_FWD_ALGO_FFT_TILING`
: This algorithm uses the Fast-Fourier Transform approach but splits the inputs into tiles. A significant memory workspace is needed to store intermediate results but less than `CUDNN_CONVOLUTION_FWD_ALGO_FFT` for large size images.

`CUDNN_CONVOLUTION_FWD_ALGO_WINOGRAD`
: This algorithm uses the Winograd Transform approach to compute the convolution. A reasonably sized workspace is needed to store intermediate results.

`CUDNN_CONVOLUTION_FWD_ALGO_WINOGRAD_NONFUSED`
: This algorithm uses the Winograd Transform approach to compute the convolution. A significant workspace may be needed to store intermediate results.

TAKE-AWAY MESSAGE

Things to consider for the choice of a convolution algorithms:

- Memory constrains (only “direct” has no overhead!)
- Availability of a GEMM library (BLAS for instance)
- Availability of SIMD instructions

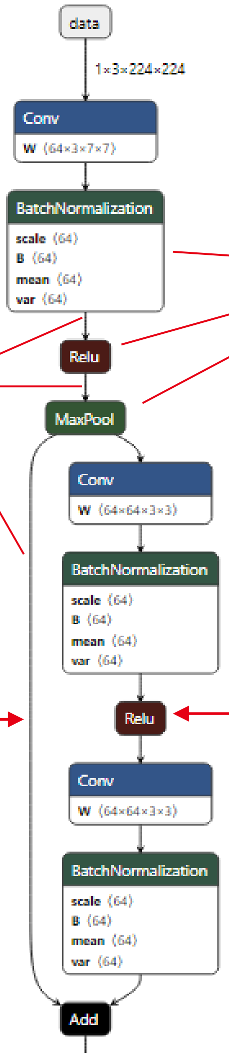
For simple “embedded” architectures (RISC V, ARM…), “direct” is generally the preferred choice, as other algorithms don’t provide any benefit (memory overhead…)

Winograd is also worth considering for 3x3 convolutions

GRAPH OPTIMIZATION

GRAPH OPTIMIZATION

GRAPH REPRESENTATION OF A NETWORK



Operators

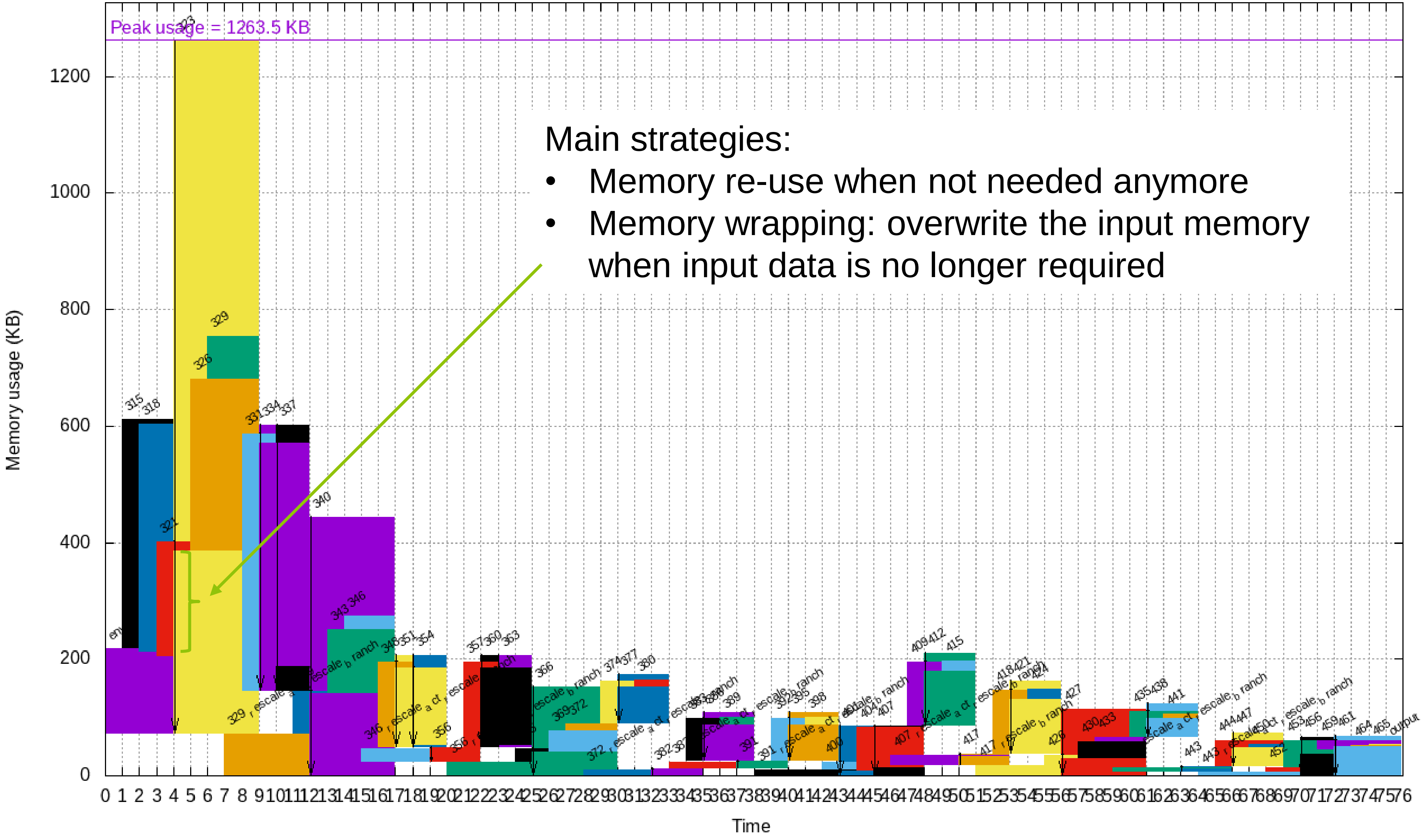
Tensors

Intermediate memory:
Data tensor must be kept in memory during computation of the branch

In-place operation:
output can be directly written in the input tensor (no data dependency)

GRAPH OPTIMIZATION

OPTIMIZED MEMORY MAPPING

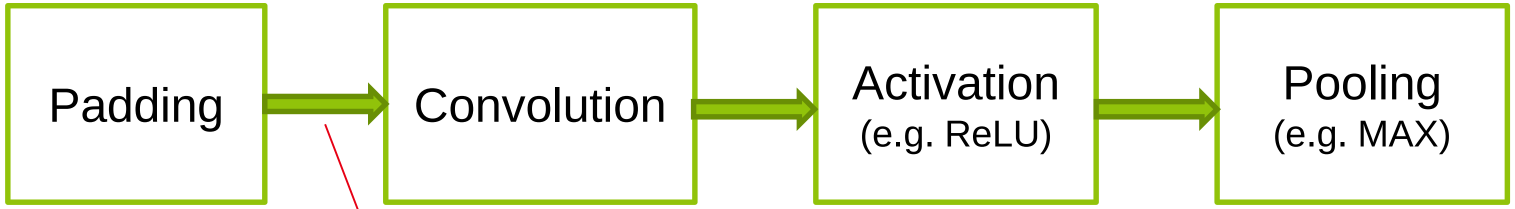


Main strategies:

- Memory re-use when not needed anymore
- Memory wrapping: overwrite the input memory when input data is no longer required

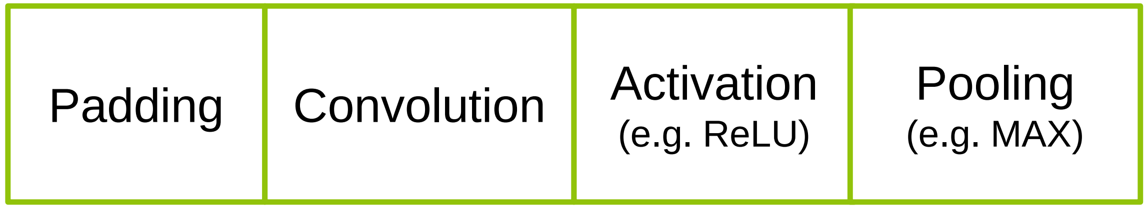
OPERATORS MERGING PRINCIPLE

- **From separate computation kernels:**



Intermediate buffers
Extra memory read/write
Successive “for” loops: control overhead, poor vectorization

- **To single monolithic kernel:**



GRAPH OPTIMIZATION
FUSE BATCHNORM WITH CONVOLUTION

- **Batch normalization recall:**
  - $x_{c,i,j}$: input *ij* of channel *c*
  - $\hat{x}_{c,i,j}$: batch normalized output *ij* of channel *c*

$$\hat{x}_{c,i,j} = \gamma.\frac{x_{c,i,j} - \mu_c}{\sqrt{\sigma_c^2 + \epsilon}} + \beta_c \quad \text{with} \quad \begin{cases} \mu_c = \frac{1}{N}\sum x_{c,i,j}\text{: mean over the batch} \\ \sigma_c^2 = \frac{1}{N}\sum\left(x_{c,i,j} - \mu_c\right)^2\text{: variance over the batch} \end{cases}$$

- **Batch normalization as 1x1 convolution:**

$$\begin{pmatrix} \hat{F}_{1,i,j} \\ \hat{F}_{2,i,j} \\ \vdots \\ \hat{F}_{C-1,i,j} \\ \hat{F}_{C,i,j} \end{pmatrix} = \begin{pmatrix} \frac{\gamma_1}{\sqrt{\hat{\sigma}_1^2+\epsilon}} & 0 & \cdots & & 0 \\ 0 & \frac{\gamma_2}{\sqrt{\hat{\sigma}_2^2+\epsilon}} & & & \\ \vdots & & \ddots & & \vdots \\ & & & \frac{\gamma_{C-1}}{\sqrt{\hat{\sigma}_{C-1}^2+\epsilon}} & 0 \\ 0 & & \cdots & 0 & \frac{\gamma_C}{\sqrt{\hat{\sigma}_C^2+\epsilon}} \end{pmatrix} \cdot \begin{pmatrix} F_{1,i,j} \\ F_{2,i,j} \\ \vdots \\ F_{C-1,i,j} \\ F_{C,i,j} \end{pmatrix} + \begin{pmatrix} \beta_1 - \gamma_1 \frac{\hat{\mu}_1}{\sqrt{\hat{\sigma}_1^2+\epsilon}} \\ \beta_2 - \gamma_2 \frac{\hat{\mu}_2}{\sqrt{\hat{\sigma}_2^2+\epsilon}} \\ \vdots \\ \beta_{C-1} - \gamma_{C-1} \frac{\hat{\mu}_{C-1}}{\sqrt{\hat{\sigma}_{C-1}^2+\epsilon}} \\ \beta_C - \gamma_C \frac{\hat{\mu}_C}{\sqrt{\hat{\sigma}_C^2+\epsilon}} \end{pmatrix}$$

"Freezed" batch norm. equivalent to 1x1 convolution layer

- <u>**Preceding convolution with linear activation:**</u>

$$y = \boldsymbol{W_{BN}}.(\boldsymbol{W_{conv}}.x + \boldsymbol{B_{conv}}) + \boldsymbol{B_{BN}} = \mathbf{W}.\mathrm{x} + \mathbf{B} \quad \text{with} \quad \begin{cases} \mathbf{W} = \boldsymbol{W_{BN}}.\boldsymbol{W_{conv}} \\ \mathbf{B} = \boldsymbol{W_{BN}}.\boldsymbol{B_{conv}} + \boldsymbol{B_{BN}} \end{cases}$$

➔ Single convolution layer!

TAKE-AWAY MESSAGE

Do not neglect graph optimization technics:

- **Merge simple operators with no data dependency**
- **Careful about memory over-usage**
- **Change the topology to allow batch norm. merging with conv.**
  Non-mergeable batch norm. can generally be moved or removed without penalty on final accuracy

QUANTIZATION TECHNICS

1. General introduction
2. Models / topologies complexity
3. Convolution algorithms
4. Graph optimization
5. **Quantization technics**
   - Post-training quantization
   - Training-aware quantization
   - Non-uniform quantization

POST-TRAINING QUANTIZATION BASIC ALGORITHM

- **Post-training quantization algorithm in 3 steps**
  - Weights normalization
    - All weights are rescaled in the range [-1.0, 1.0]
    - Per layer normalization
    - Per layer and per output channel normalization
      ➔ finer grain, better usage of the quantized range for some output channels

  - Activations normalization
    - Activations at each layer are rescaled in the range [-1.0, 1.0] for signed outputs
      and [0.0, 1.0] for unsigned outputs
    - Find **optimal quantization threshold value** of the activation output of each layer
      ➔ using the validation dataset
    - Iterative process: need to take into account previous layers normalizing factors

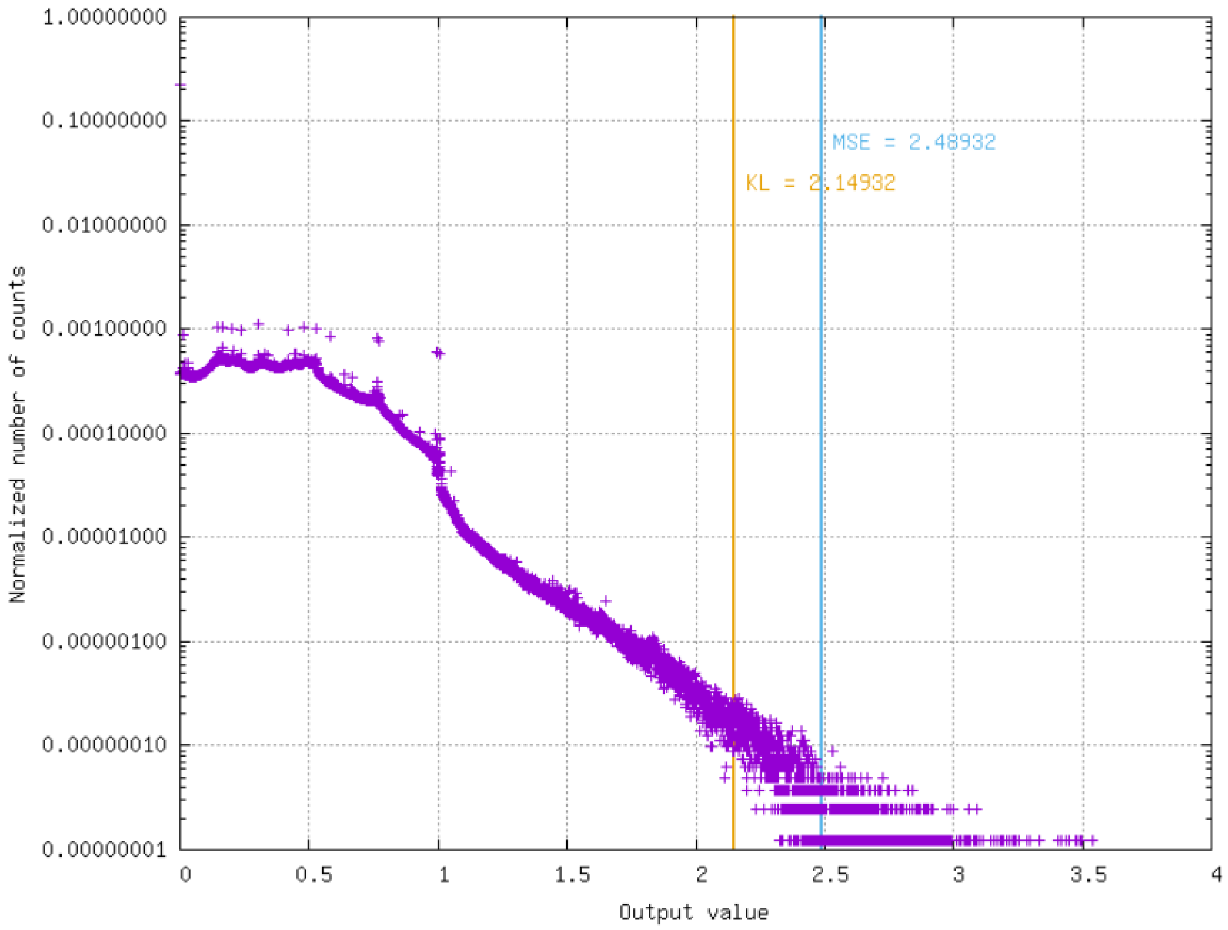
  - Quantization
    - Inputs, weights, biases and activations are quantized to the desired *nbbits* precision
    - Convert ranges from [-1.0, 1.0] and [0.0, 1.0] to $[-2^{nbbits-1}-1, 2^{nbbits-1}-1]$ and $[0, 2^{nbbits}-1]$ taking into account all dependencies

POST-TRAINING QUANTIZATION BASIC ALGORITHM

- **Find** **optimal quantization threshold value** **of the activation output of each layer**
  - Compute histogram of activation values
  - Find threshold that minimizes distance between original distribution and clipped quantized distribution

    ➔ two distance algorithms can be used:
    - Mean Squared Error (MSE)
    - Kullback–Leibler divergence metric (KL-divergence)

Threshold value = activation scaling factor to be taken into account during quantization



POST-TRAINING QUANTIZATION BASIC ALGORITHM

- **Additional optimization strategies**

  - Weights clipping *(optional)*
    same as activations: find optimal quantization threshold value

  - Activation scaling factor approximation
    - Fixed-point
      α ➔ $x2^{-p}$
    - Single-shift
      α ➔ 2^{x}
    - Double-shift
      α ➔ $2^{n} + 2^{m}$

POST-TRAINING: DATA FREE QUANTIZATION (DFQ)

- **Goal: avoid the need to use data for calibration**

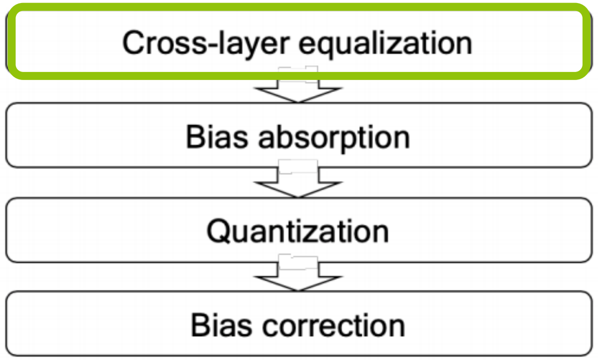


Figure 4. Flow diagram of the proposed DFQ algorithm.

$f(s.x) = s.f(x)$ for ReLU

➔ scaling invariance

$$r_i^{(1)} = max_j \left|W^{ij^{(1)}}\right|$$

$$s_i = \frac{1}{r_i^{(2)}}\sqrt{r_i^{(1)}.r_i^{(2)}}$$

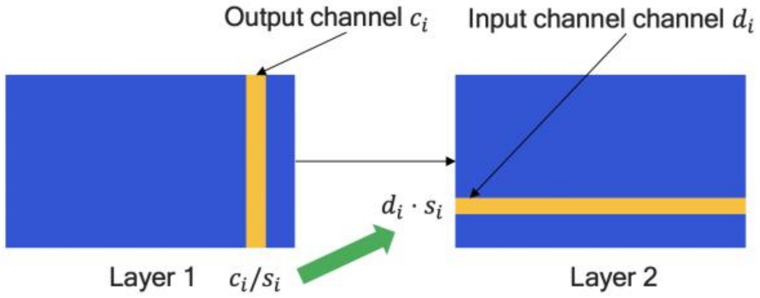


Figure 5. Illustration of the rescaling for a single channel. If scaling factor s_i scales c_i in layer 1; we can instead factor it out and multiply d_i in layer 2.

	∼D	∼BP	∼AC	MobileNetV2 FP32	MobileNetV2 INT8	MobileNetV1 FP32	MobileNetV1 INT8	ResNet18 FP32	ResNet18 INT8	ResNet18 INT6
DFQ (ours)	✓	✓	✓	71.7%	**71.2%**	70.8%	**70.5%**	69.7%	**69.7%**	66.3%
Per-layer [18]	✓	✓	✓	71.9%	0.1%	70.9%	0.1%	69.7%	69.2%*	63.8%*
Per-channel [18]	✓	✓	✓	71.9%	69.7%	70.9%	70.3%	69.7%	69.6%*	**67.5%***
QT [16] ^	✗	✗	✓	71.9%	70.9%	70.9%	70.0%	-	**70.3%**†	67.3%†
SR+DR†	✗	✗	✓	-	-	-	**71.3%**	-	68.2%	59.3%
QMN [31]	✗	✗	✗	-	-	70.8%	68.0%	-	-	-
RQ [21]	✗	✗	✗	-	-	-	70.4%	-	69.9%	**68.6%**

Table 5. Top1 ImageNet validation results for different models and quantization approaches. The top half compares level 1 approaches (∼D: data free, ∼BP: backpropagation-free, ∼AC: Architecture change free) whereas in the second half we also compare to higher level approaches in literature. Results with * indicates our own implementation since results are not provided, ^ results provided by [18] and † results from table 2 in [21].

POST-TRAINING QUANTIZATION BASIC ALGORITHM

- **Performances (post-training quantization)**

Network model Input: 224x224 Output: 1000 or 1001	**FP accuracy on ImageNet in N2D2 [vs reported]**	**Export quantized accuracy (default)** (without eq. from [3])
MobileNet_V1 ONNX model from TF	71.04% [70.9%]	60.39% (~52%)
MobileNet_V1 ONNX model from MXNet	70.17% [71.05%]	68.84% (~62%)
MobileNet_V2 ONNX model from TF	[71.8%]	Not working (scaling issue)
MobileNet_V2 ONNX model from PyTorch	69.20% [71.8%]	65.52% (-)
MobileNet_V2 ONNX model from MXNet	[70.94%]	65.43% (~17%)

Post-training quantization performances may be sensible on how the network was trained!

Challenges in MobileNet quantization:

[1] Tao Sheng et al., "A Quantization-Friendly Separable Convolution for MobileNets, 2019
[2] Alexander Finkelstein et al., Fighting Quantization Bias With Bias, 2019
[3] Nagel, Markus, et al. "Data-free quantization through weight equalization and bias correction.“, 2019

POST-TRAINING QUANTIZATION BASIC ALGORITHM

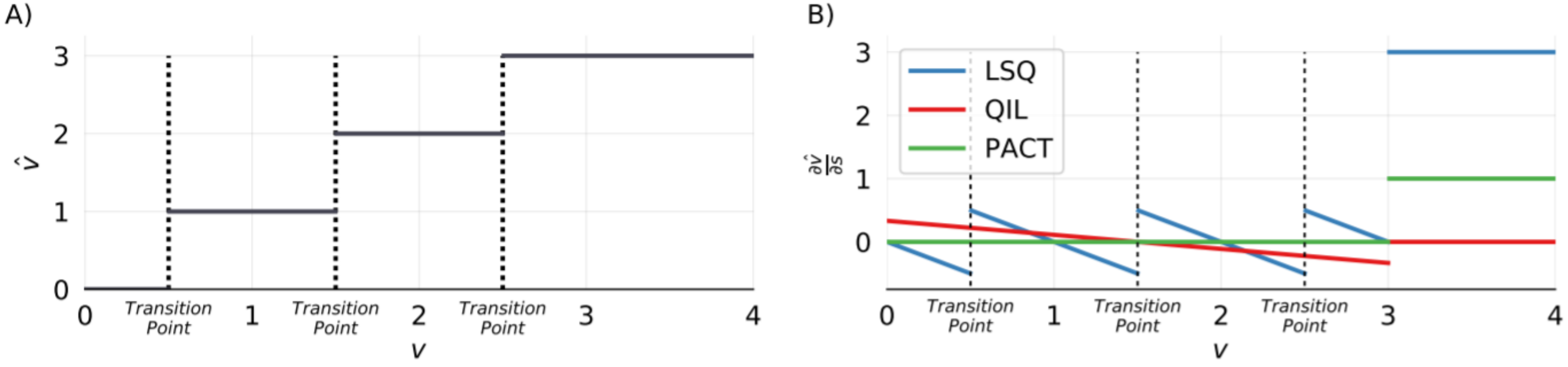
- **Performances (post-training quantization)**
  - Accuracy loss analysis: example with MobileNet_V2 ONNX model from PyTorch

Baseline (FP)	Quantization friendly restructuring	Integer conversion	(Single-shift rescaling)
	- Dropout removal - BatchNorm fuse with Conv (Zero-variance issue correction [1]) - Padding fuse with Conv/Pool - Cross layer equalization [3] (with clipping removal) - Weights & biases rescaling - Activations rescaling	- Rounding	- Single-shift rescaling
69.67%	**68.39% (-1.28)**	**66.74% (-2.93)**	**65.44% (-4.23)**

TRAINING AWARE QUANTIZATION

- **LSQ (Esser 2019) and LSQ+ (Bhalgat 2020)**
  - Learned Step Size Quantization

Quantizer function



Gradient used for training

  - 3-bit models able to reach the full precision baseline accuracy
  - First and last layer: always use 8-bit (standard SofA practice)
  - Initialized from a trained full precision model

Table 4: Accuracy for low precision networks trained with LSQ and knowledge distillation, which is improved over using LSQ alone, with 3-bit networks reaching the accuracy of full precision (32-bit) baselines (shown for comparison).

	Top-1 Accuracy @ Precision					Top-5 Accuracy @ Precision				
Network	2	3	4	8	*32*	2	3	4	8	*32*
ResNet-18	67.9	70.6	71.2	71.1	*70.5*	88.1	89.7	90.1	90.1	*89.6*
ResNet-34	72.4	74.3	74.8	74.1	*74.1*	90.8	91.8	92.1	91.7	*91.8*
ResNet-50	74.6	76.9	77.6	76.8	*76.9*	92.1	93.4	93.7	93.3	*93.4*

TRAINING AWARE QUANTIZATION

- **Scale-Adjusted Training (SAT)**
  - Combine previous SotA technics:
    - DoReFa scheme (Zhou et al. (2016)) for weight quantization
    - PACT (Choi et al., 2018) for activation quantization

  - Efficient Training Rule I: prevent logits from entering saturation region of the cross entropy loss
  - Efficient Training Rule II:
    - BN layers should be used after linear layers such as convolution and fully-connected layers
    - Variance of effective weights should be on the order of the reciprocal nb. neurons of the linear layer

  - Weight quantization: SAT restores the variance of effective weights
    - Clamping: constant rescaling

$$W^*_{ij} = \frac{1}{\sqrt{\widehat{n}\mathbb{VAR}[\widehat{W}_{rs}]}}\widehat{W}_{ij}$$

    - Quantization

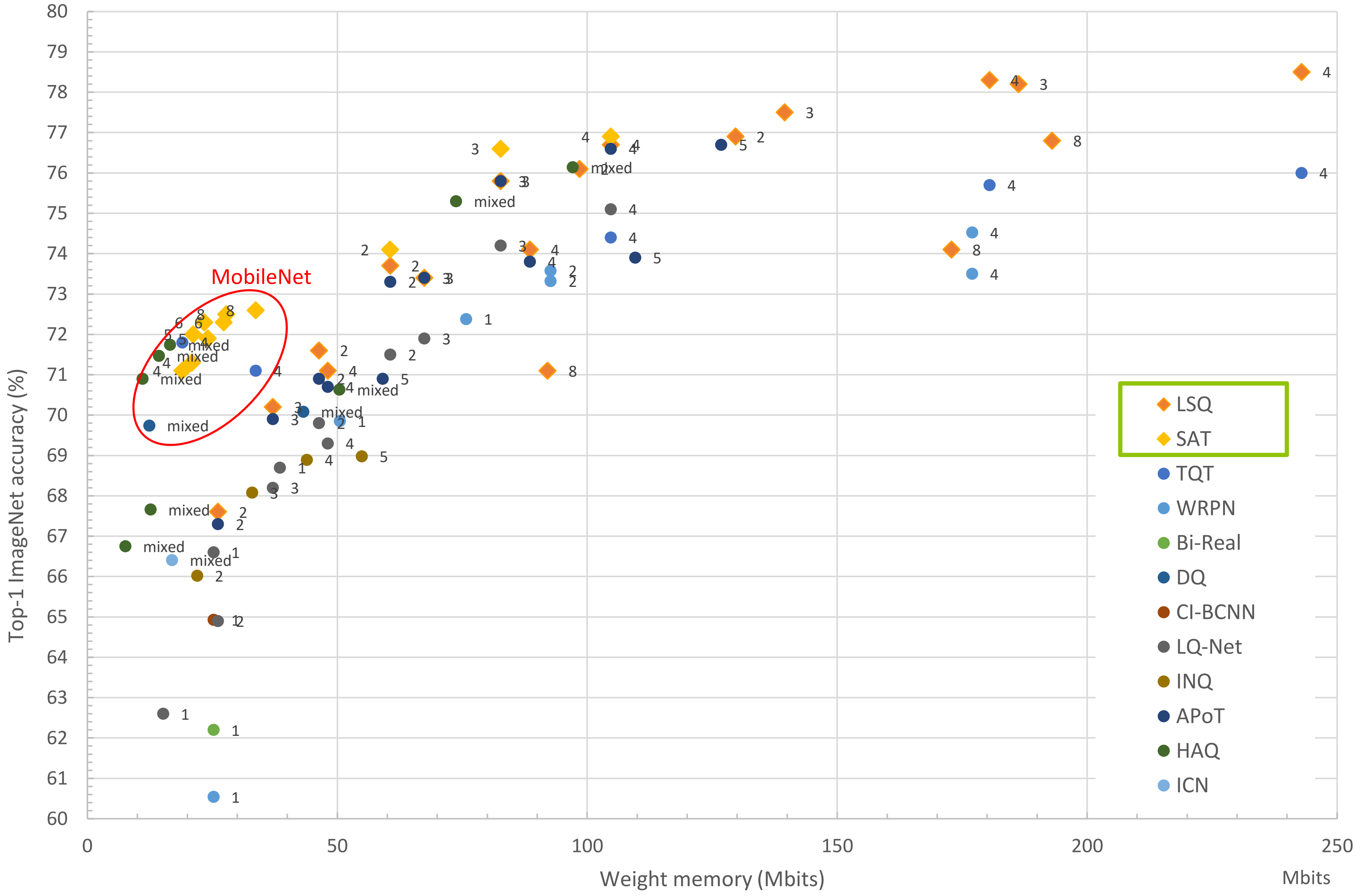
$$Q^*_{ij} = \frac{1}{\sqrt{n_{\text{out}}\mathbb{VAR}[Q_{rs}]}}Q_{ij}$$

  - Activation quantization

$$\frac{\partial q}{\partial \alpha} = \begin{cases} q_k\left(\frac{\widetilde{x}}{\alpha}\right) - \frac{\widetilde{x}}{\alpha} & x < \alpha \\ 1 & x > \alpha \end{cases}$$

QUANTIZATION TECHNICS

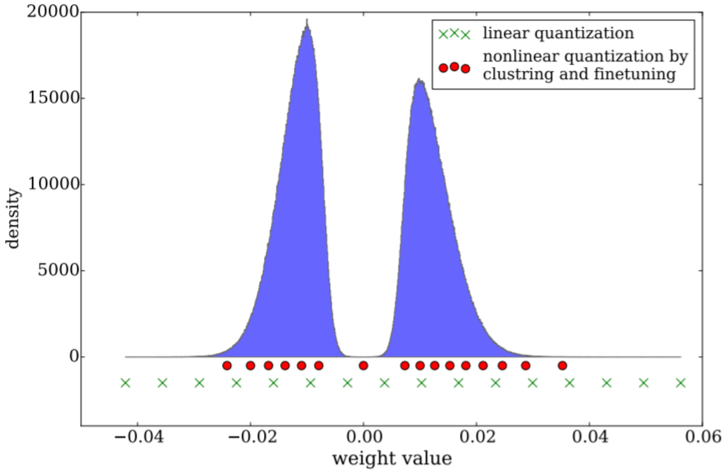
TRAINING AWARE QUANTIZATION



NON-UNIFORM QUANTIZATION

- **Goal: Allow non uniform quantizers to provide better « understanding » of the distribution**

Deep Compression



Use of k-means to determine the position of the quantization points. Compression using pruning, weight sharing and Huffman coding. No accuracy loss, and big compression factors on old architectures.

Drawback : Has only been tested on old architectures (2016 : before RESNET)

UNIQ

Uniform noise injection for non-uniform quantization

Concept: Non-uniform k-quantile quantizer

Idea : Keep the quantization operation differentiable by injecting distributior $F_W^{-1}(F_W(w)+e)$ Differentiable quantization

F_W : cumulative distribution function of the gaussian that fits best to the shape of the layer's weights.

e : uniform random noise in [$-\frac{1}{2k}$, $\frac{1}{2k}$], where k in the number of quantization points (= quantiles).

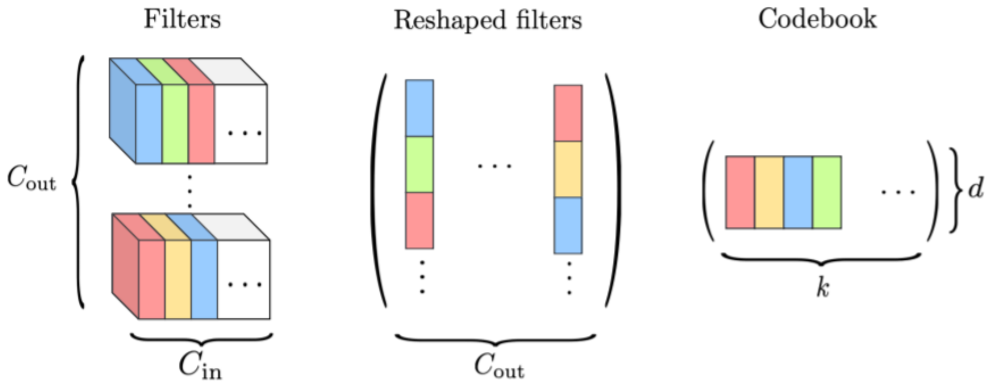
Results : Works well for small/medium-sized networks. The scheme needs to be applied gradually to train deeper networks (because of the accumulation of noise).

NON-UNIFORM QUANTIZATION

Vector quantization

Concept: learnable codebook, updated through k-means and fine-tuned using SGD

Idea: As in Deep Compression, we want to learn a custom distribution for the quantization points. But unlike Deep Compression, the quantization points are vectors. Each column of a tensor to quantize is split into subvectors of the same dimension d. These subvectors are then quantized thanks to a learnable codebook, updated through a k-means strategy and then fine-tuned using SGD.



Example of the quantization of a convolutive layer. Same color = Same codeword assigned.

E-step : Each subvector **v** of the a column of a tensor to quantize is assigned to a codeword (exhaustive search).

$$\mathbf{c}_j = \operatorname*{argmin}_{\mathbf{c} \in \mathcal{C}} \|\widetilde{\mathbf{x}}(\mathbf{c} - \mathbf{v})\|_2^2$$

M-step : weighted k-means update of each codeword, considering the set of subvectors assigned to it.

$$\mathbf{c}^\star = \operatorname*{argmin}_{\mathbf{c} \in \mathbf{R}^d} \sum_{p \in I_\mathbf{c}} \|\widetilde{\mathbf{x}}(\mathbf{c} - \mathbf{v}_p)\|_2^2$$

Finetuning : Codewords updated by SGD using the set of subvectors assigned to them.

$$\mathbf{c} \leftarrow \mathbf{c} - \eta \frac{1}{|I_\mathbf{c}|} \sum_{p \in I_\mathbf{c}} \frac{\partial \mathcal{L}}{\partial \mathbf{b}_p}$$

TAKE-AWAY MESSAGE

Post-training quantization:

- Good enough down to 8-bit
- Simple: no retraining required,
  but may still need data for calibration

Quantization-aware training:

- Similar accuracy in 4-bit vs floating point
- Down to 3- or 2-bit with lower accuracy
- Complex: require data and specific quantized training

Other (non uniform: k-mean, codebook…):

benefits vs latest (uniform) quantization-aware not clear at the moment

Questions?

Centre de Saclay
Nano-Innov PC 172 - 91191 Gif sur Yvette Cedex