

Convolutional Neural Networks for Semantic Segmentation

Giacomo Boracchi

giacomo.boracchi@polimi.it

Data Pre-processing and Batch Normalization

Preprocessing

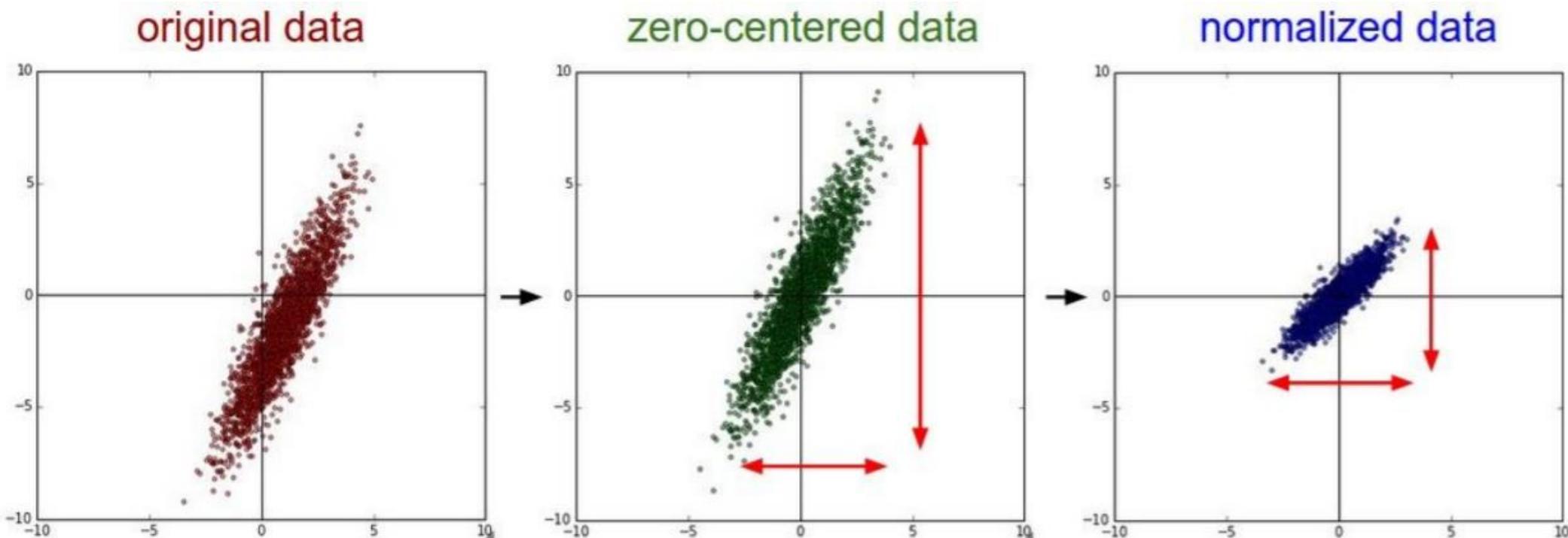
In general, normalization is useful in gradient-based optimizers.

Normalization is meant to bring training data “around the origin” and possibly further rescale the data

In practice, optimization is made easier and results are less sensitive to perturbations in the parameters

There are several options

There are different form of preprocessing



```
X -= np.mean(X, axis = 0)
```

```
X /= np.std(X, axis = 0)
```

Preprocessing for CNN

PCA/Whitening preprocessing are not commonly used with Convolutional Networks.

The most frequent option is to zero-center the data, and it is common to normalize every pixel as well.

Preprocessing and Training:

- Any preprocessing statistics (e.g. the data mean) must be computed on training data, and applied to the validation / test data.
- Do not normalize first and then split in training, validation, test
- Normalization statistics are parameters of your ML model

Preprocessing for CNNs: mean subtraction

e.g. consider CIFAR-10 example with $[32,32,3]$ images

Subtract the mean image (e.g. AlexNet)
(*mean image = $[32,32,3]$ array*)

Subtract per-channel mean (e.g. VGGNet)
(*mean along each channel = 3 numbers*)

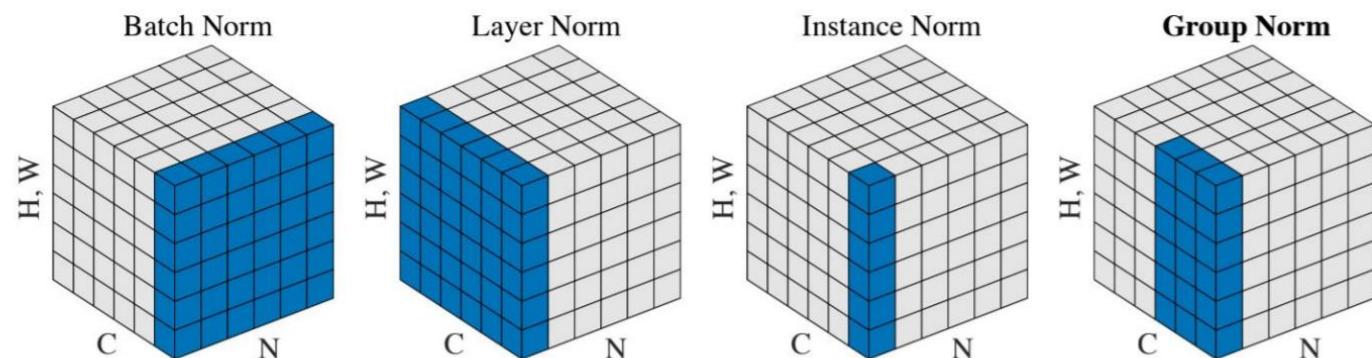
Subtract per-channel mean and Divide by per-channel std (e.g. ResNet)
(*mean along each channel = 3 numbers*)

Batch Normalization

Consider a batch of activations $\{x_i\}$, the following transformation bring these to unit variance and zero mean

$$x'_i = \frac{x_i - E[x_i]}{\sqrt{\text{var}[x_i]}}$$

Where $E[x_i]$ and $\sqrt{\text{var}[x_i]}$ are computed from each batch and separately for each channel!



Wu and He, "Group Normalization", ECCV 2018

Can we get more flexibility than zero-mean, unit variance?

Ioffe, S. and Szegedy, C., 2015, June. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International conference on machine learning* (pp. 448-456). PMLR.

Batch Normalization

Batch normalization adds after standard normalization

$$x'_i = \frac{x_i - E[x_i]}{\sqrt{\text{var}[x_i]}}$$

a further a parametric transformation

$$y_{i,j} = \gamma_j x'_i + \beta_j$$

Where parameters γ and β are learnable scale and shift parameters.

Rmk: estimates $E[x_i]$ and $\sqrt{\text{var}[x_i]}$ are computed on each minibatch, need to be fixed after training. **After training, these are replaced by (running) averages of values seen during training.**

Batch Normalization

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_1 \dots m\}$;

Parameters to be learned: γ, β

Output: $\{y_i = \text{BN}_{\gamma, \beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m} \sum_{i=1}^m x_i \quad // \text{mini-batch mean}$$

$$\sigma_{\mathcal{B}}^2 \leftarrow \frac{1}{m} \sum_{i=1}^m (x_i - \mu_{\mathcal{B}})^2 \quad // \text{mini-batch variance}$$

$$\hat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \quad // \text{normalize}$$

$$y_i \leftarrow \gamma \hat{x}_i + \beta \equiv \text{BN}_{\gamma, \beta}(x_i) \quad // \text{scale and shift}$$

Algorithm 1: Batch Normalizing Transform, applied to activation x over a mini-batch.

Batch Normalization

During testing batch normalization becomes a linear operator! Can be fused with the previous fully-connected or conv layer

In practice networks that use Batch Normalization are significantly more robust to bad initialization

Typically Batch Normalization is used in between FC layers of deep CNN, but sometimes also between Conv Layers

Batch Normalization

Pros:

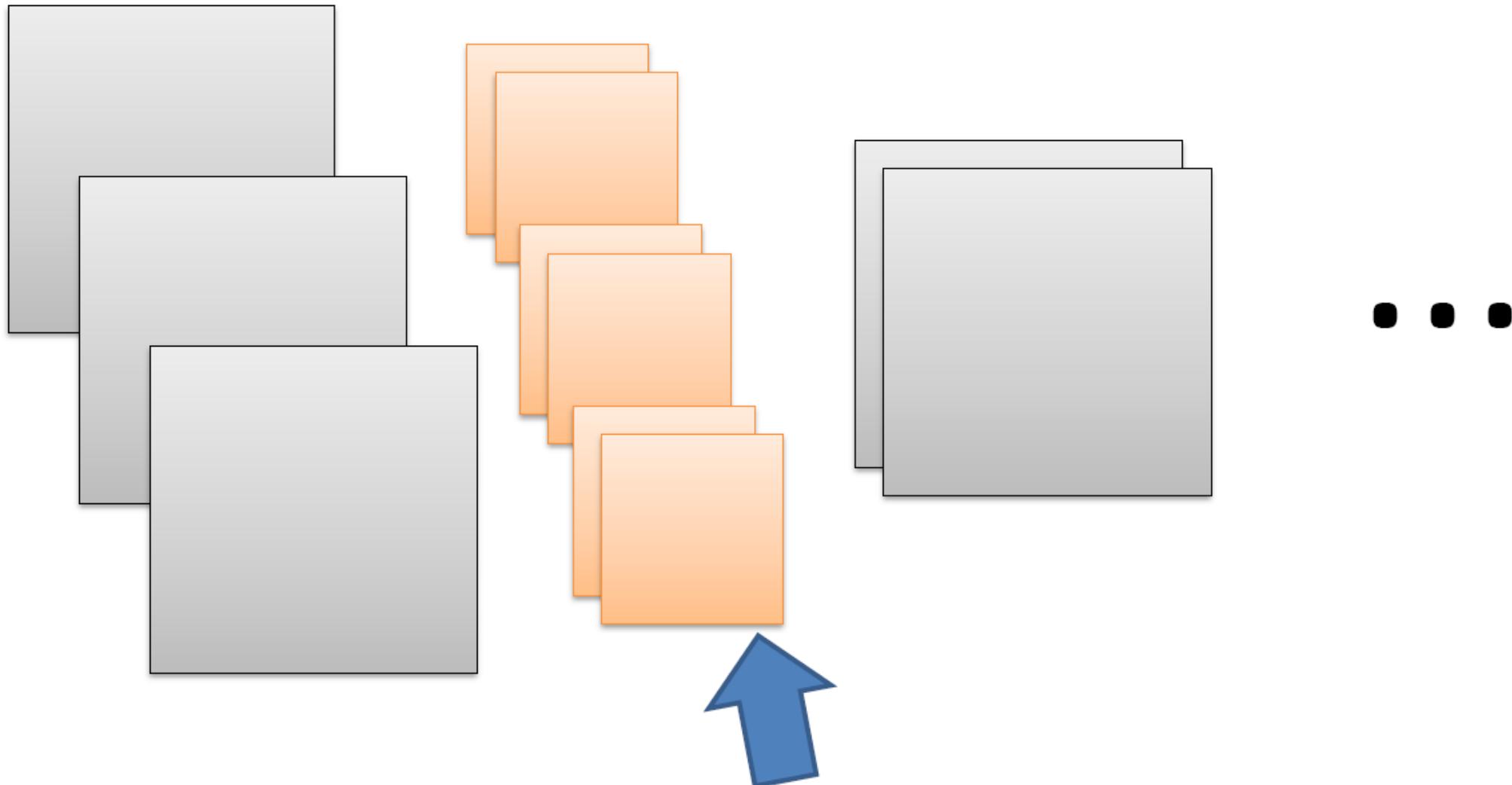
- Makes deep networks much easier to train!
- Improves gradient flow
- Allows higher learning rates, faster convergence
- Networks become more robust to initialization
- Acts as regularization during training
- Zero overhead at test-time: can be fused with conv!

Watch out:

- Behaves differently during training and testing: this is a very common source of bugs!

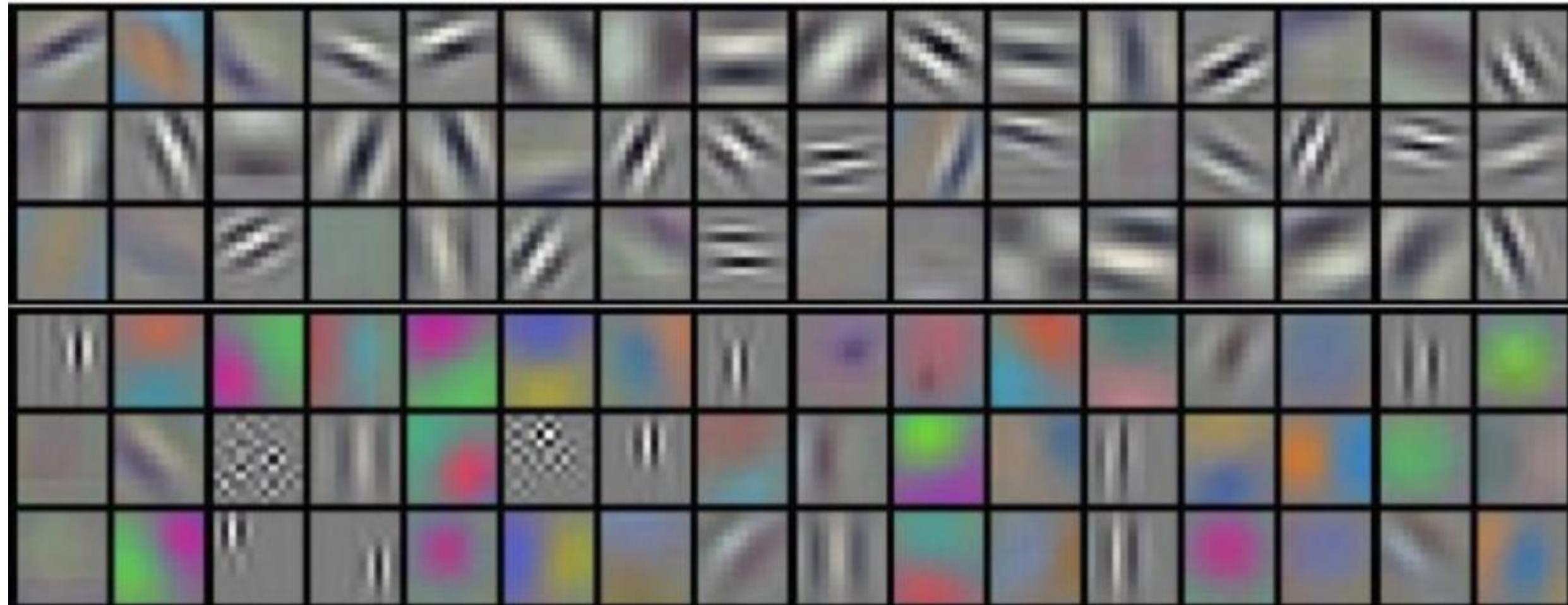
CNN Visualization

Visualizing CNN Filters



We want to see this

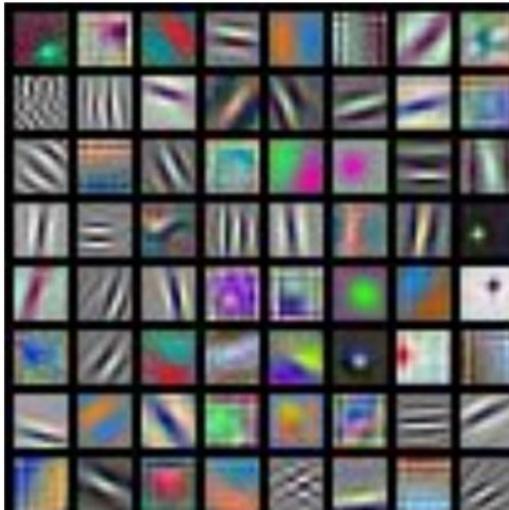
11x11x3 filters (visualized in RGB) extracted from
the first convolutional layer



11x11x3 filters (visualized in RGB) extracted from the first convolutional layer

Recall the relation between convolution and template matching:
The first layer seems to match low-level features such as edges
and simple patterns that
are discriminative to describe the data

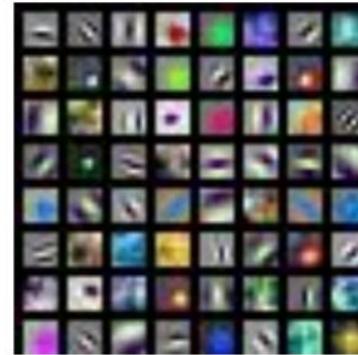
First layer's filters are often like these



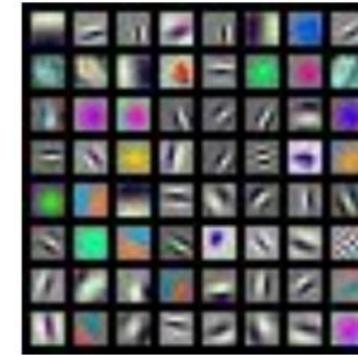
AlexNet:
 $64 \times 3 \times 11 \times 11$



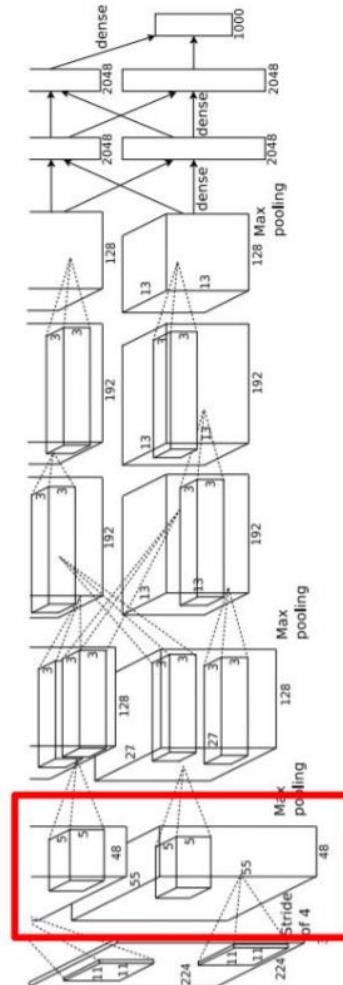
ResNet-18:
 $64 \times 3 \times 7 \times 7$



ResNet-101:
 $64 \times 3 \times 7 \times 7$



DenseNet-121:
 $64 \times 3 \times 7 \times 7$



Krizhevsky, "One weird trick for parallelizing convolutional neural networks", arXiv 2014
He et al, "Deep Residual Learning for Image Recognition", CVPR 2016
Huang et al, "Densely Connected Convolutional Networks", CVPR 2017

Difficult to interpret deeper layers

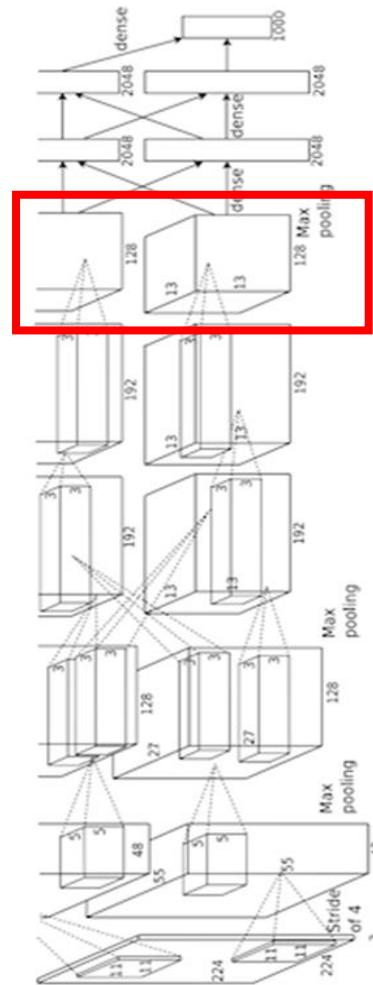
Weights:

Weights:
() () ()
() () () ()
() () () ()
() () () ()
() () ()

Weights:
() () ()
() () () ()
() () () ()
() () () ()
() () () ()

layer 1 weights

$16 \times 3 \times 7 \times 7$



layer 2 weights

$20 \times 16 \times 7 \times 7$

layer 3 weights

$20 \times 20 \times 7 \times 7$

Difficult to interpret deeper layers

Weights:

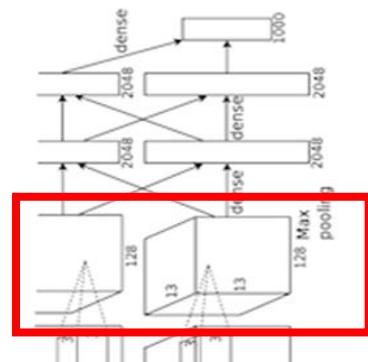


Weights:

(模糊的黑色背景)(模糊的黑色背景)(模糊的黑色背景)
(模糊的黑色背景)(模糊的黑色背景)(模糊的黑色背景)
(模糊的黑色背景)(模糊的黑色背景)(模糊的黑色背景)
(模糊的黑色背景)(模糊的黑色背景)(模糊的黑色背景)
(模糊的黑色背景)(模糊的黑色背景)(模糊的黑色背景)
(模糊的黑色背景)(模糊的黑色背景)(模糊的黑色背景)

layer 1 weights

$16 \times 3 \times 7 \times 7$



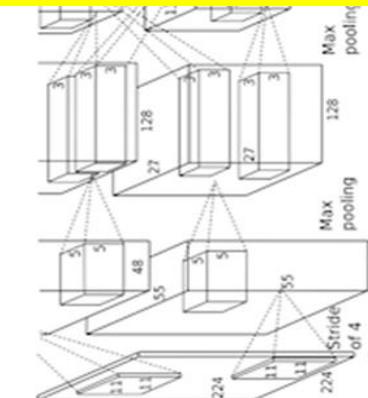
layer 2 weights

Another way to determine «what the deepest layer see» is required

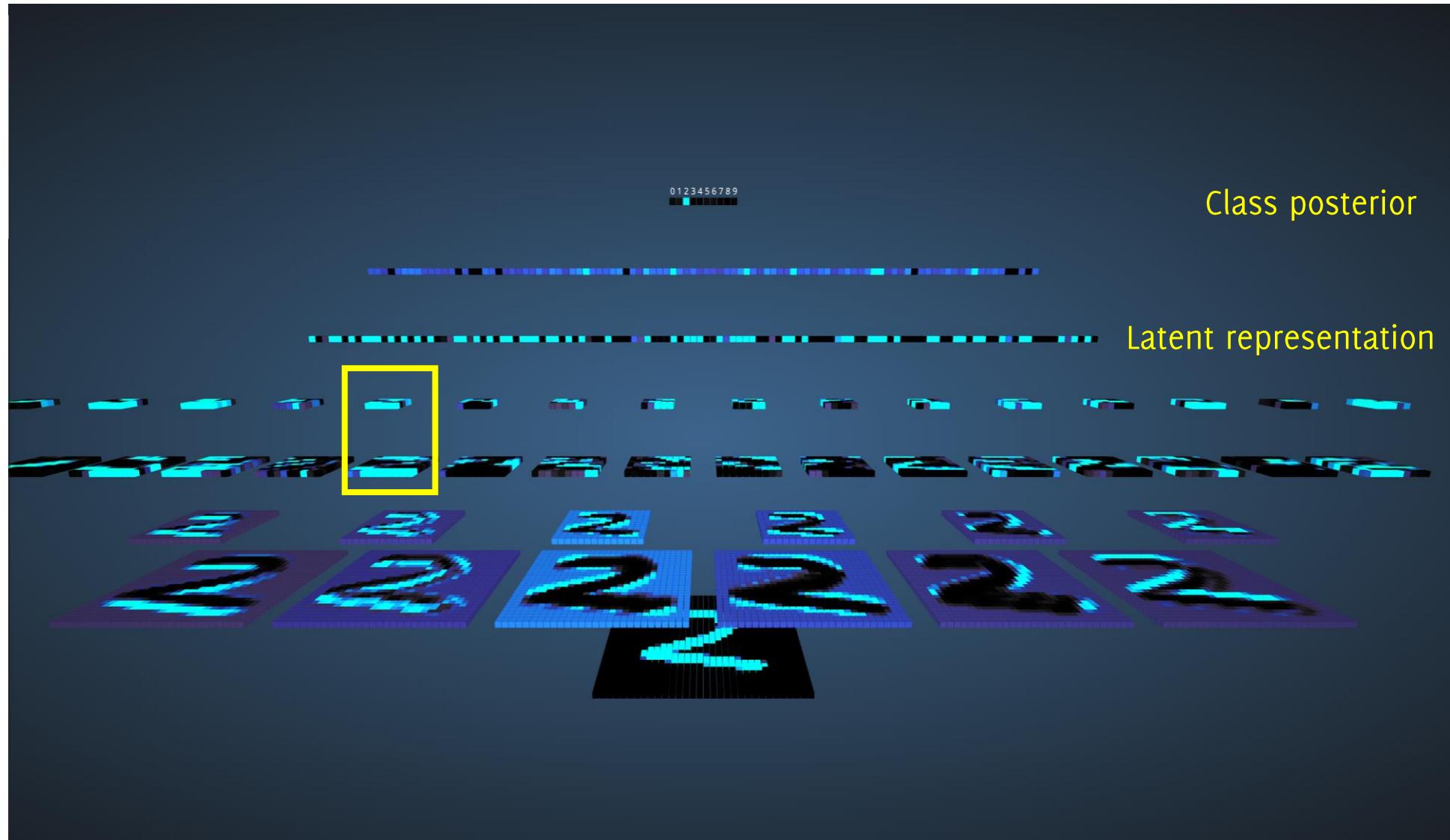
(模糊的黑色背景)(模糊的黑色背景)(模糊的黑色背景)
(模糊的黑色背景)(模糊的黑色背景)(模糊的黑色背景)
(模糊的黑色背景)(模糊的黑色背景)(模糊的黑色背景)
(模糊的黑色背景)(模糊的黑色背景)(模糊的黑色背景)
(模糊的黑色背景)(模糊的黑色背景)(模糊的黑色背景)
(模糊的黑色背景)(模糊的黑色背景)(模糊的黑色背景)

layer 3 weights

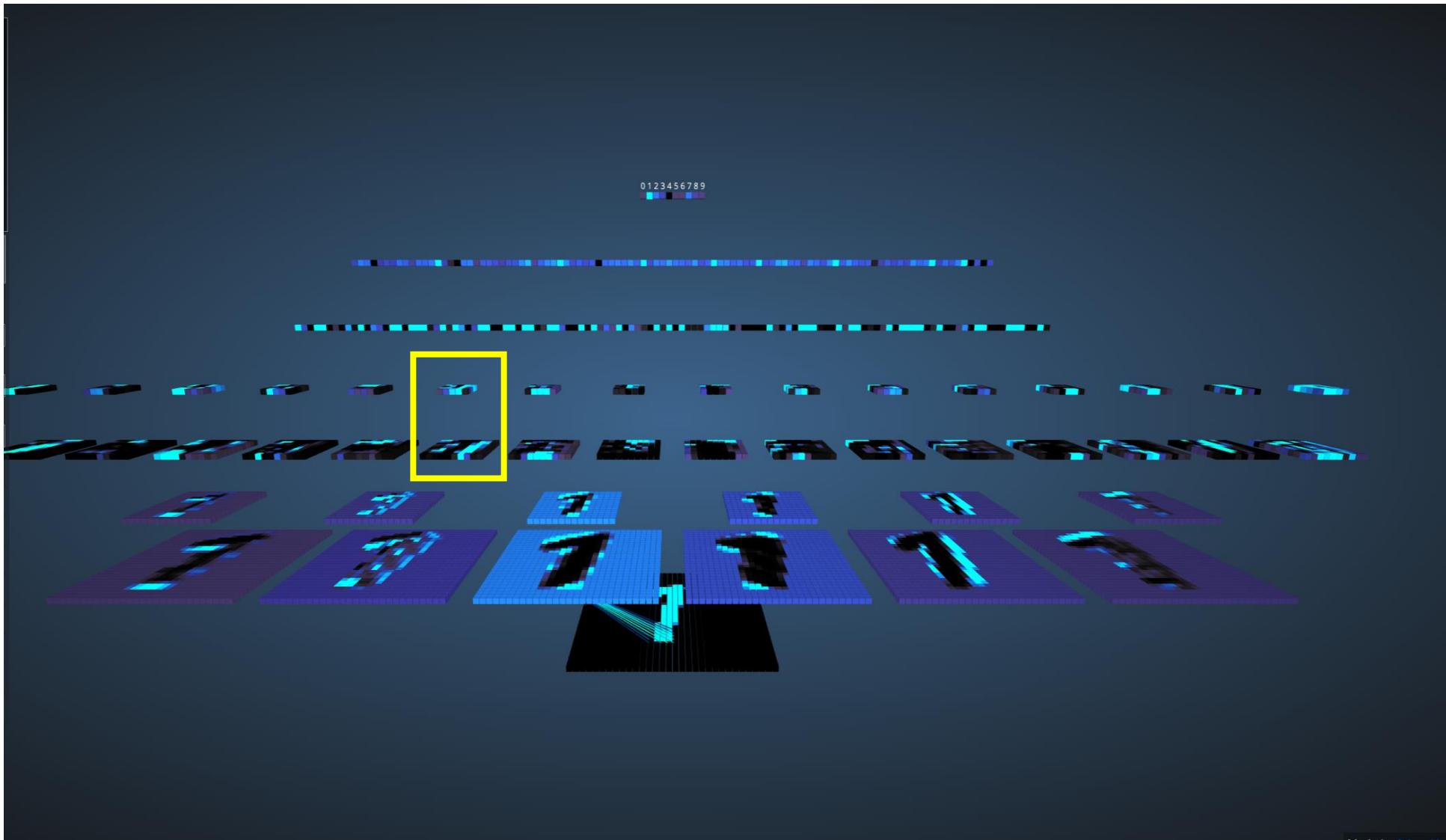
$20 \times 20 \times 7 \times 7$



What if we look at the activations?

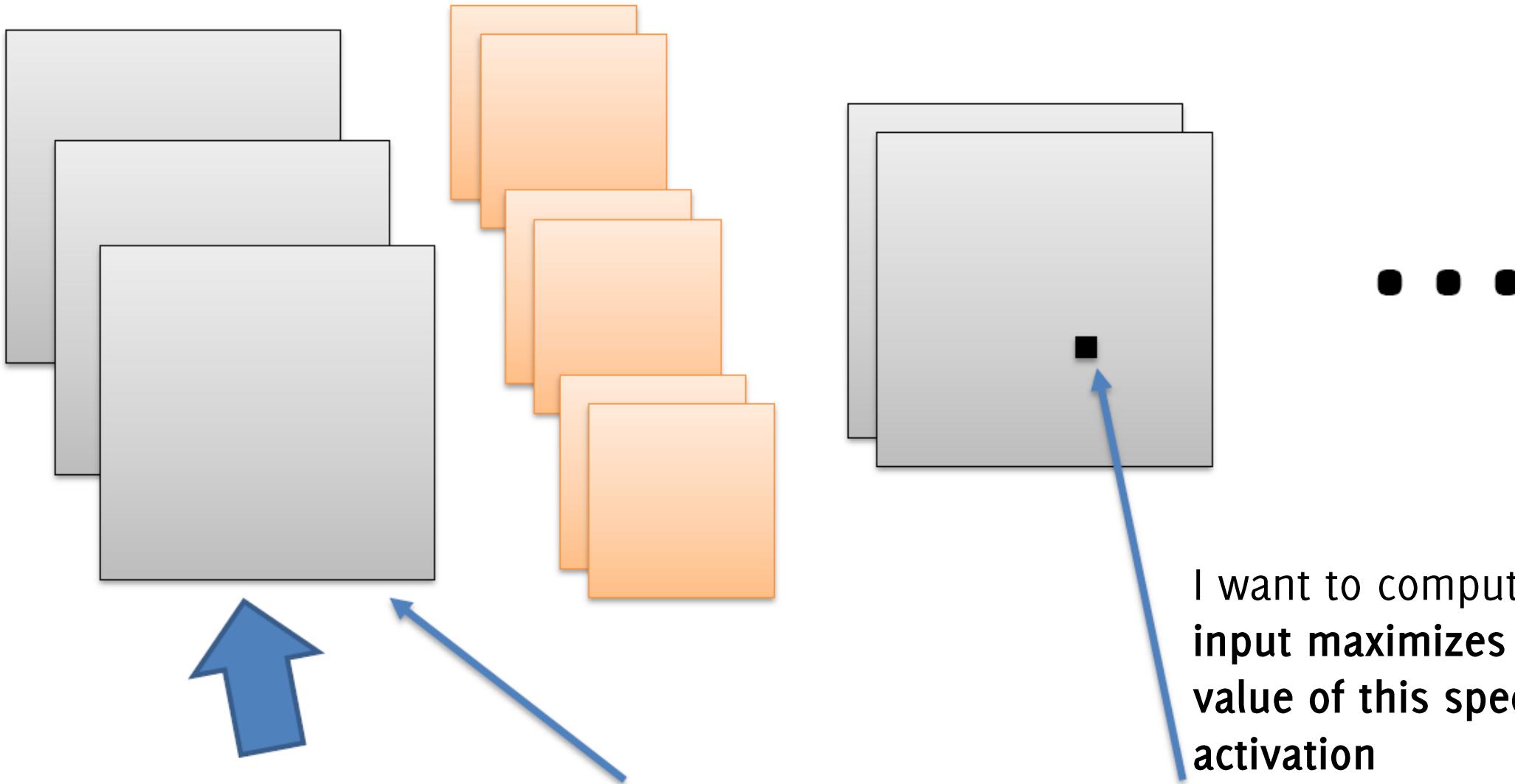


What if we look at the activations?



Made by Adam Harley

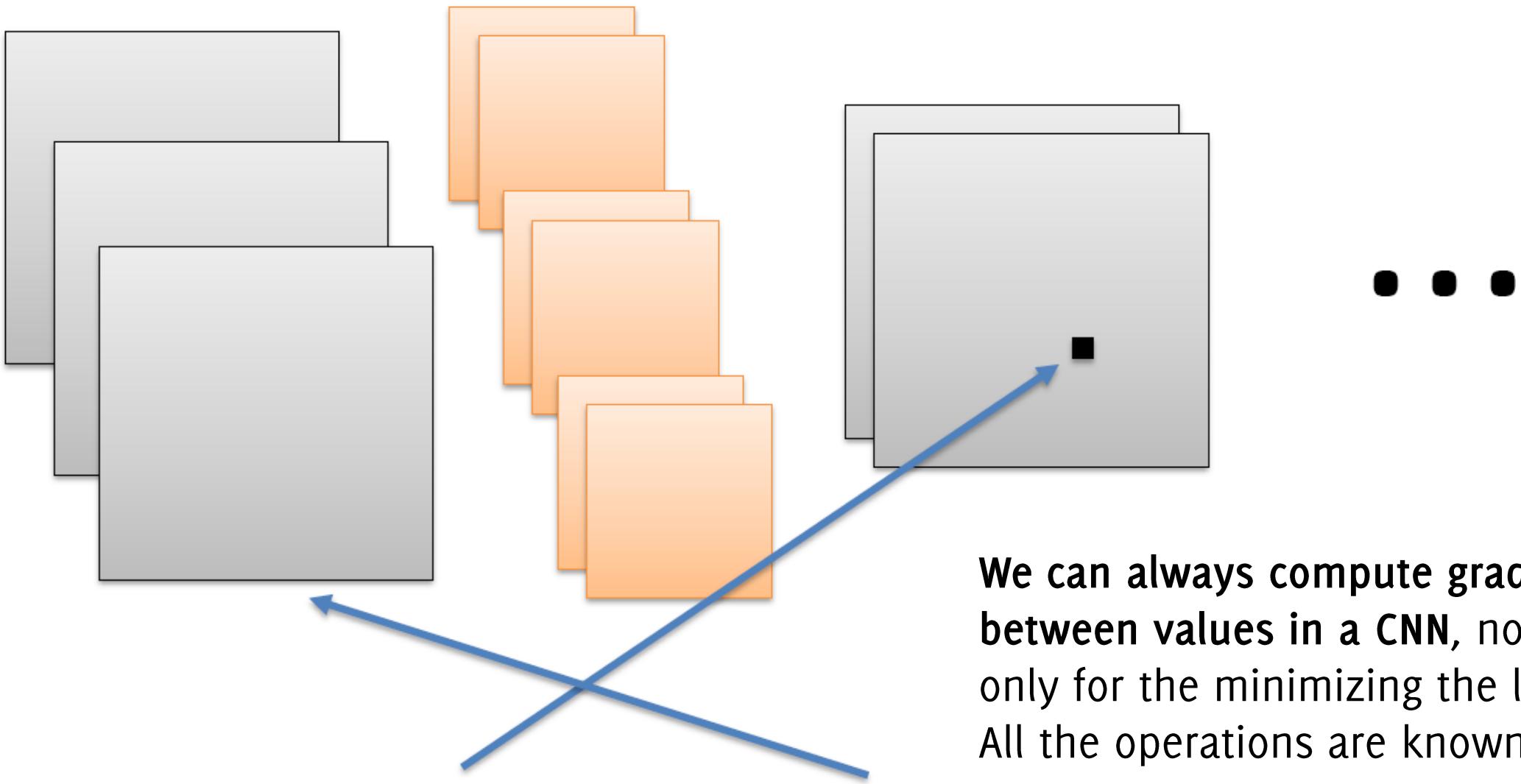
Visualizing the Input that maximally activates some neurons



We want to compute (and see) the input that maximally activates this guy

I want to compute **which input maximizes the value of this specific activation**

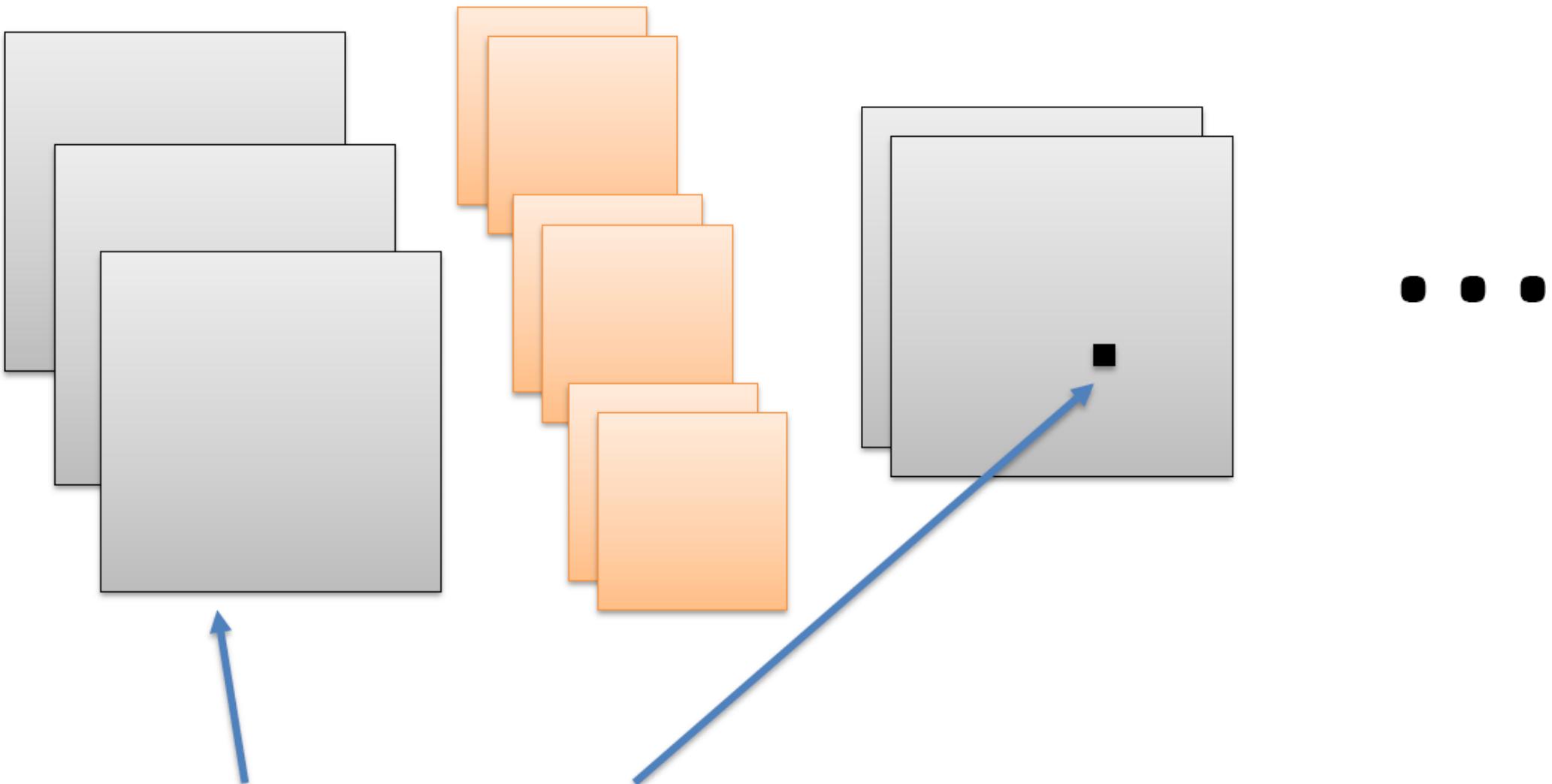
Step 1



Compute the gradient of this with respect to the input

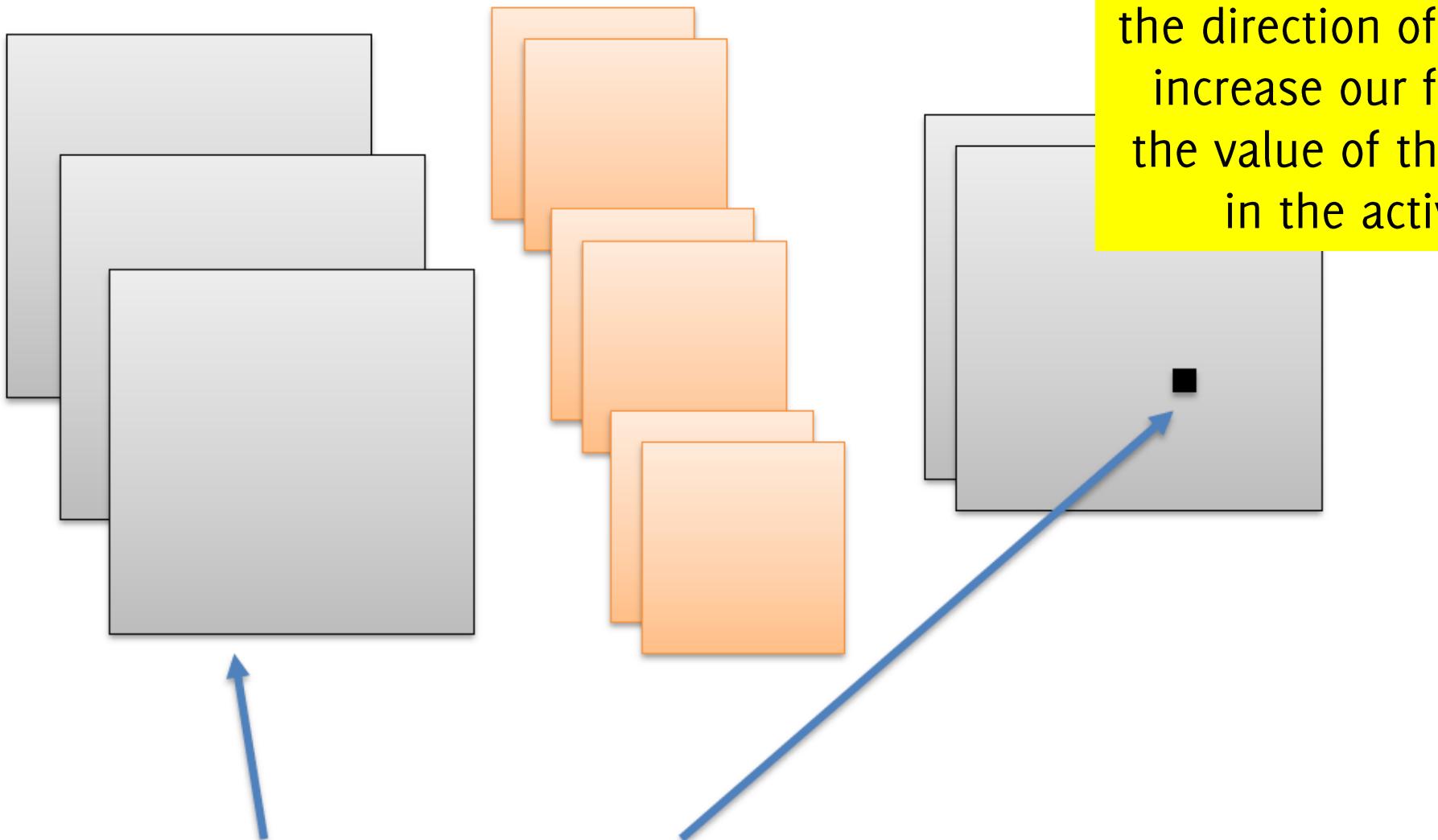
We can always compute gradient between values in a CNN, not only for the minimizing the loss. All the operations are known

Step 2



Nudge the input accordingly: our guy will increase its activation

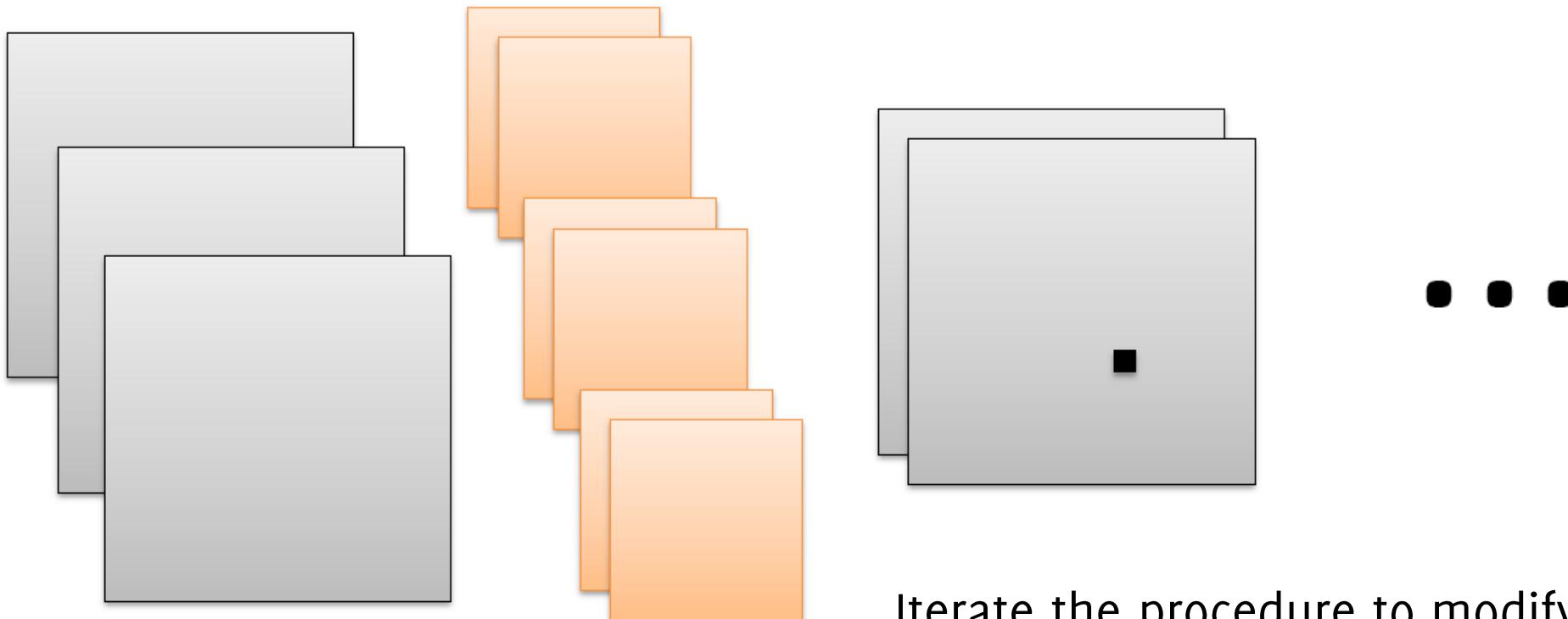
Step 2



Nudge the input accordingly: our guy will increase its activation

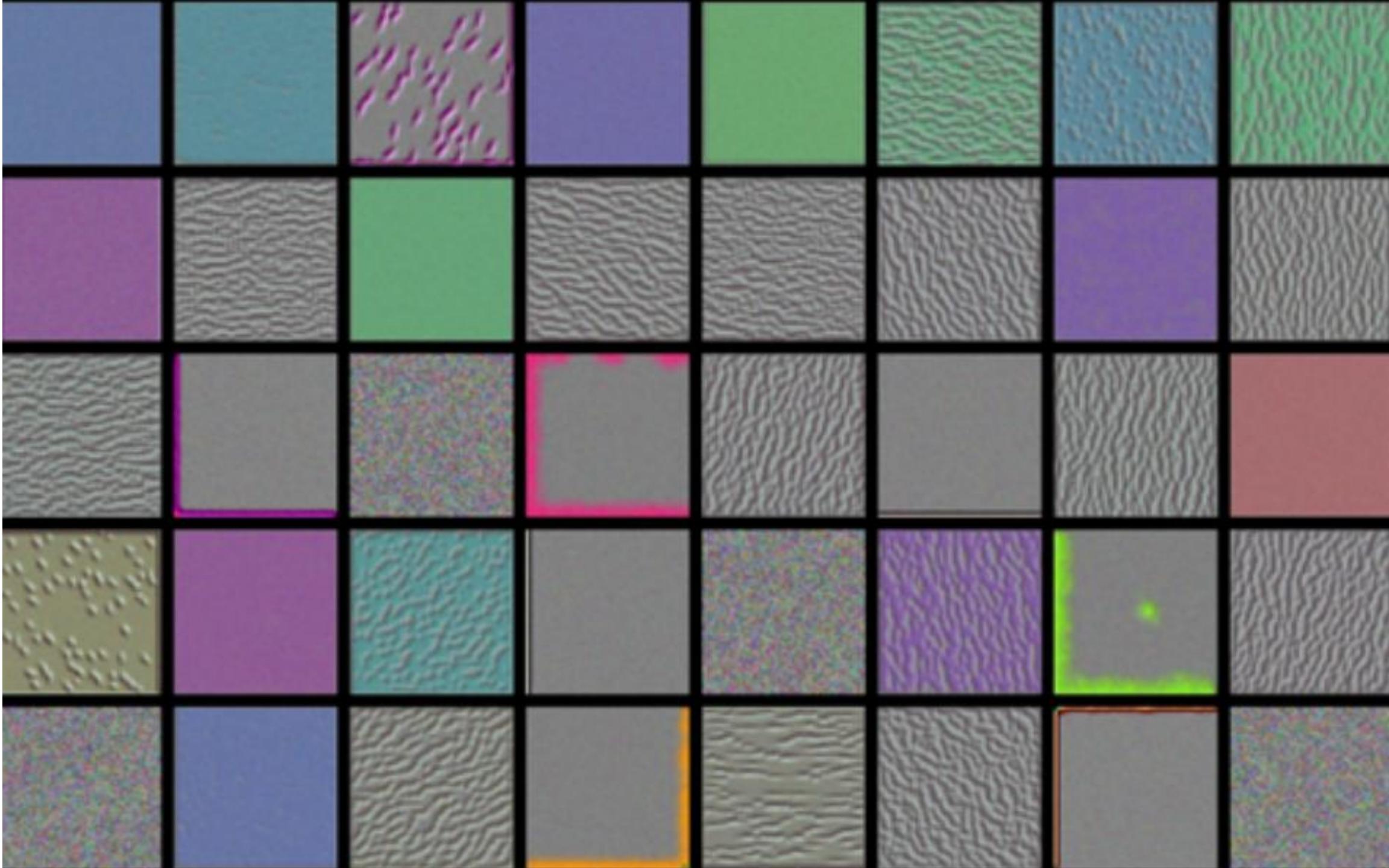
Now we perform gradient ascent, we modify the input in the direction of the gradient, to increase our function that is the value of the selected pixel in the activation map

Back to step 1 and iterate

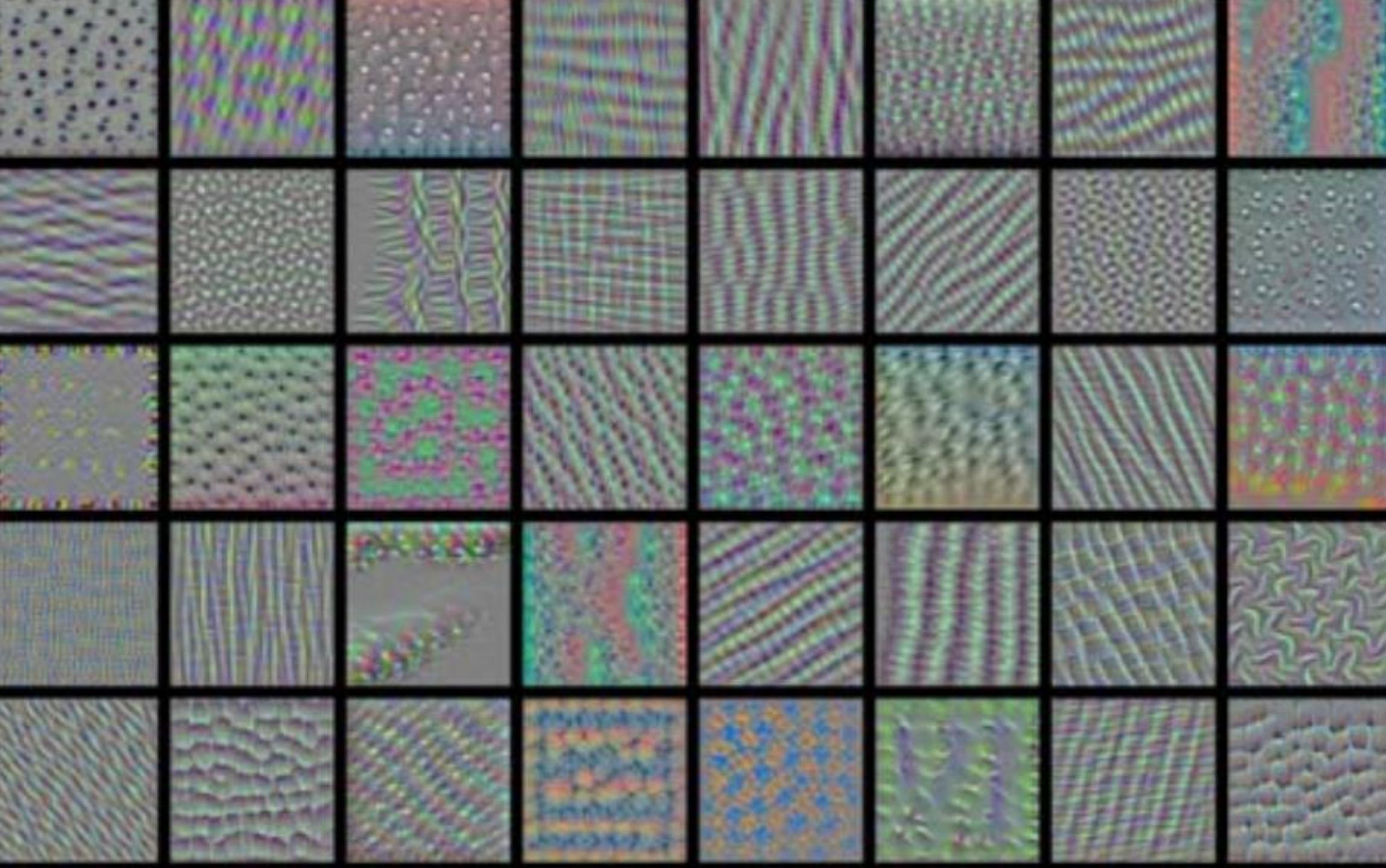


Iterate the procedure to modify the input.
Some form of regularization can be added
to the selected pixel value to steer the
input to look more like a natural image

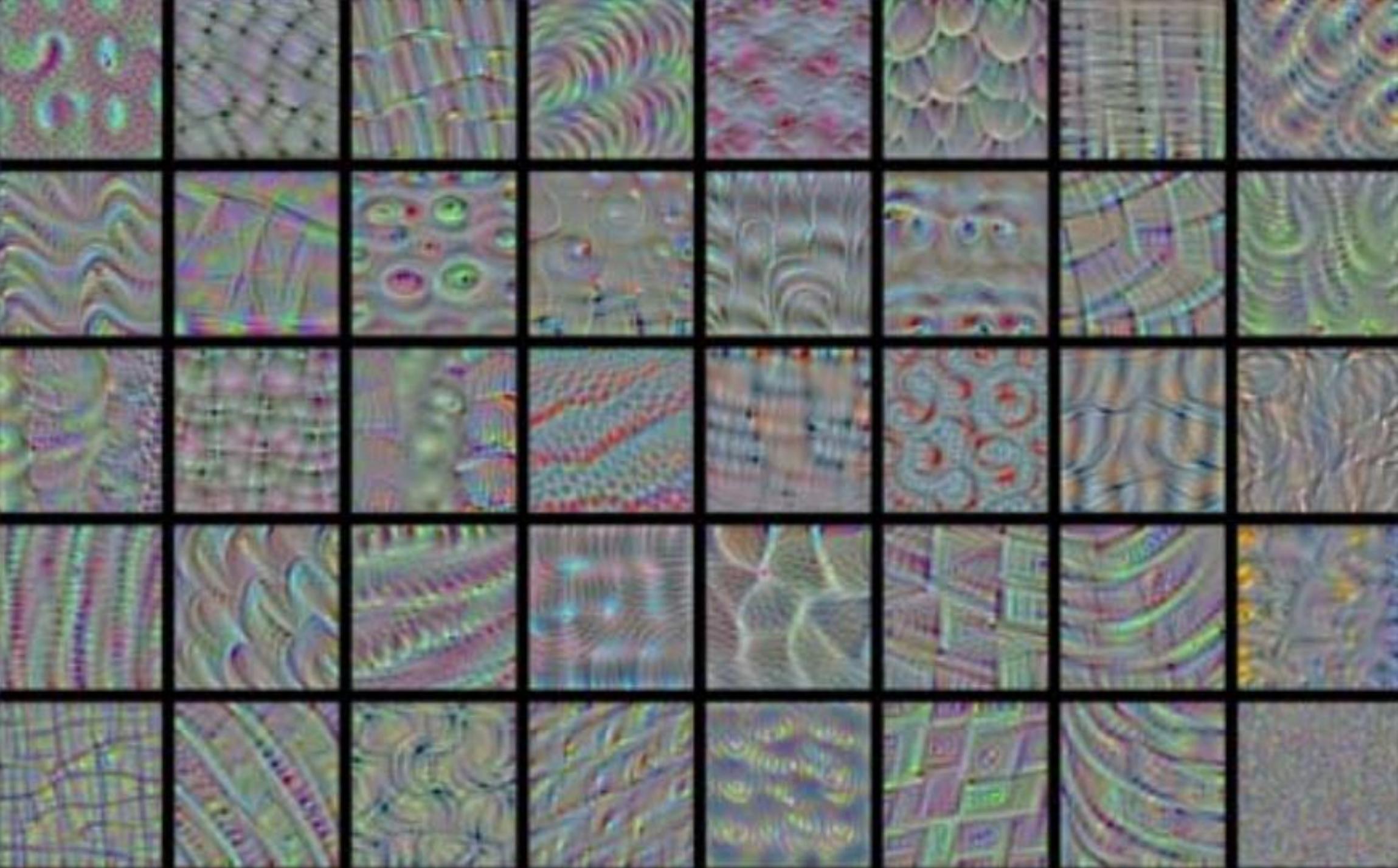
Shallow layers respond to fine, low-level patterns



Intermediate layers ...

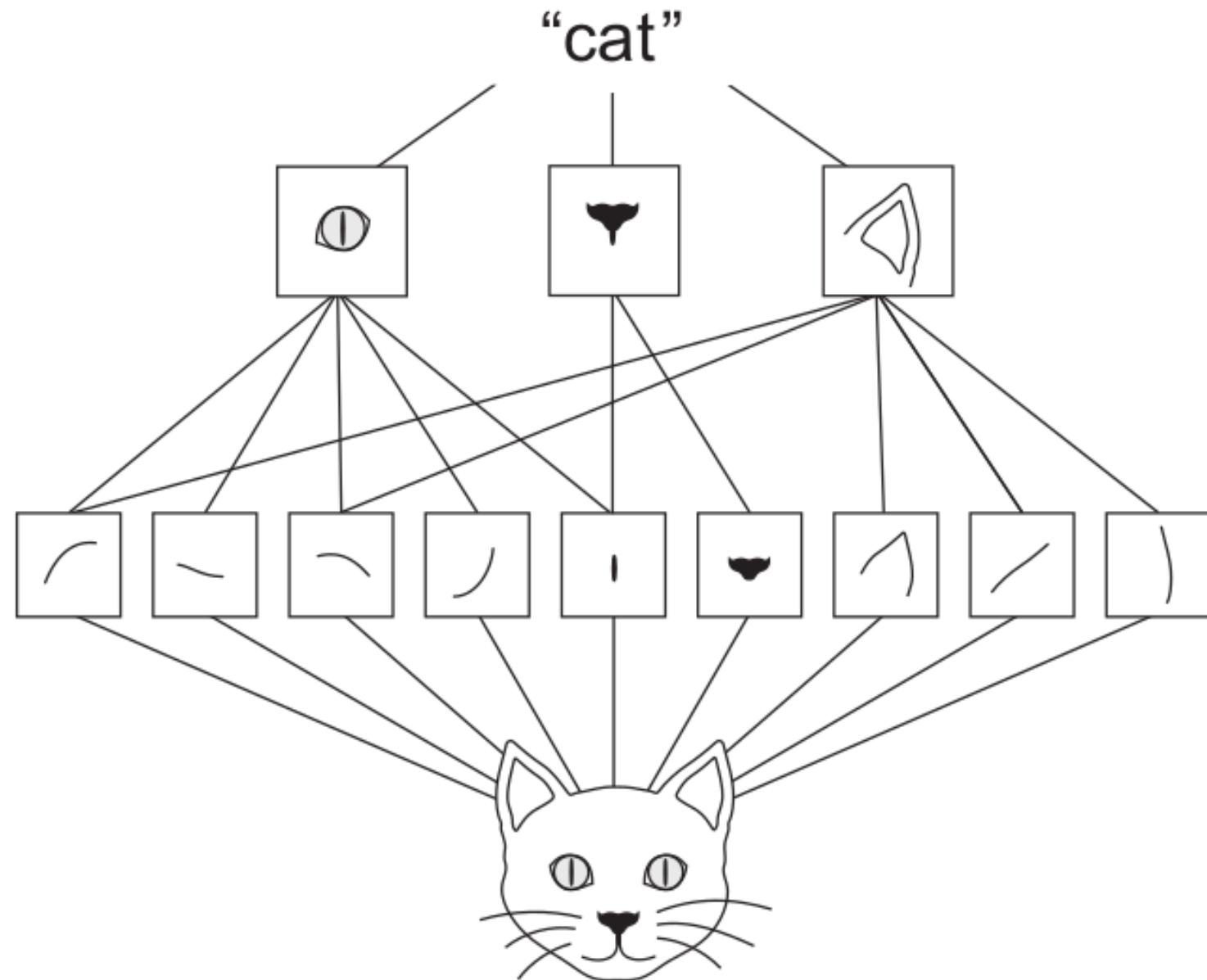


Deep layers respond to complex,
high-level patterns



Why CNNs work

Convnets learn a **hierarchy** of **translation-invariant** spatial pattern detectors

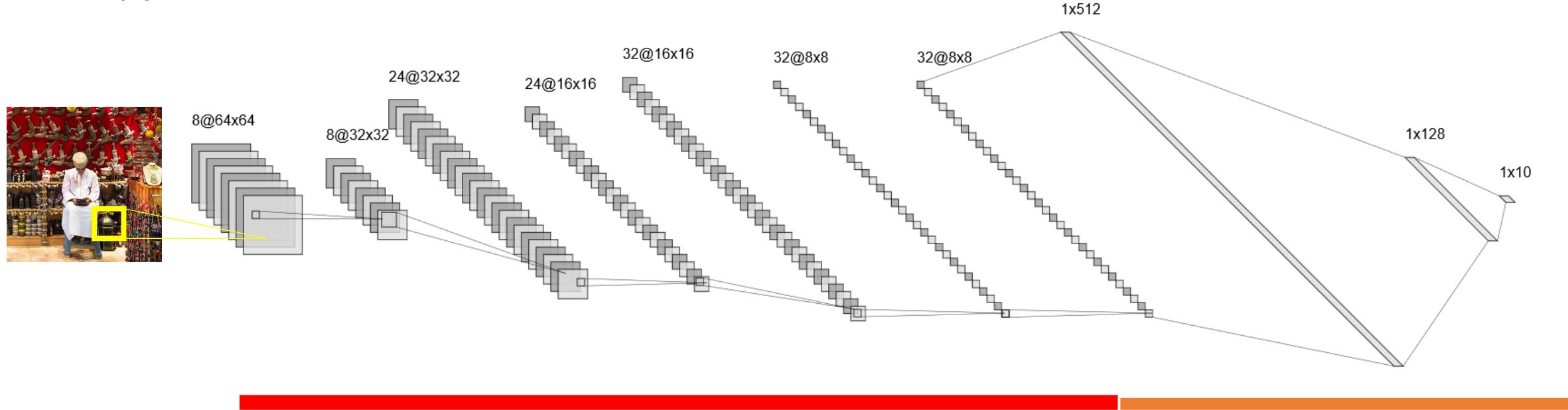


Fully Convolutional Networks

What happens when feeding the network with
an image having different size?

Convolutional Neural Networks (CNN)

The typical architecture of a convolutional neural network



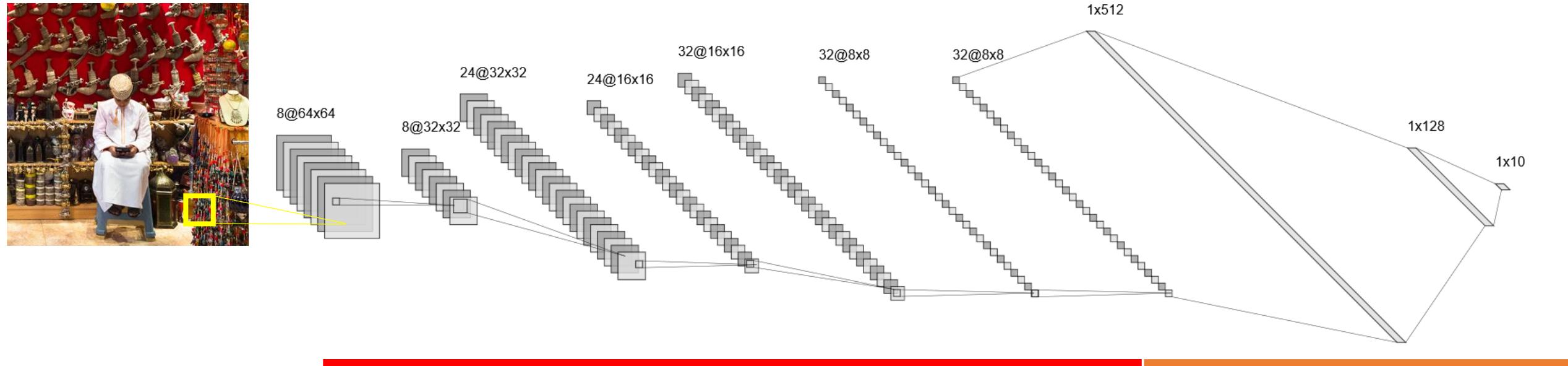
CNNs are meant to process input of a fixed size (e.g. 200 x 200).

The **convolutional and subsampling layers** operate in a sliding manner over image having arbitrary size

The **fully connected** layer constrains the input to a fixed size.

Convolutional Neural Networks (CNN)

The typical architecture of a convolutional neural network



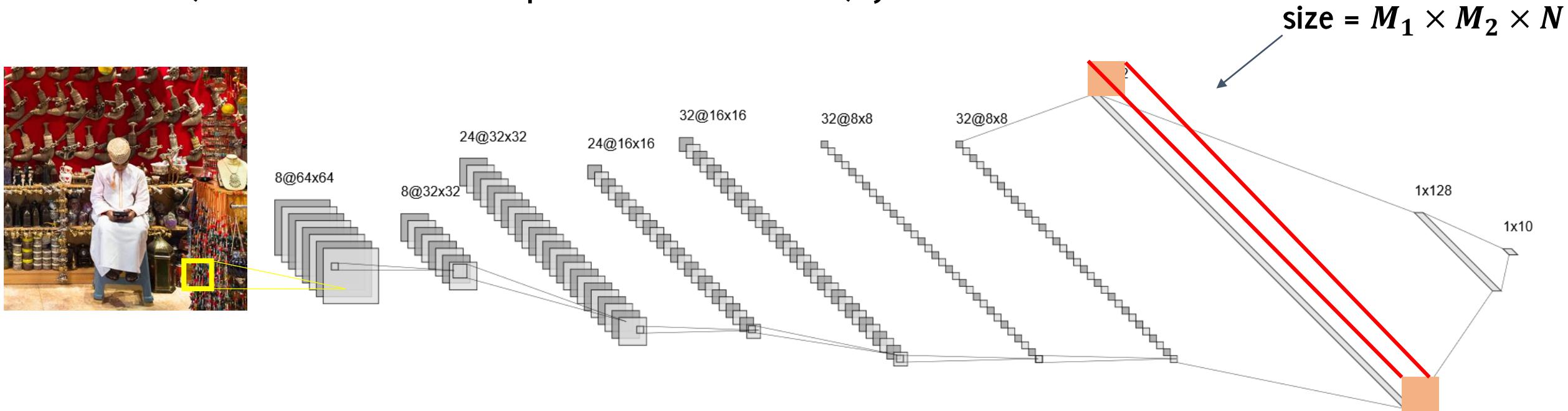
What happens when we feed a larger image to the network?

Convolutional Neural Networks (CNN)

Convolutional filters can be applied to volumes of any size, yielding larger volumes in the network until the FC layer.

The FC network however does require a fixed input size

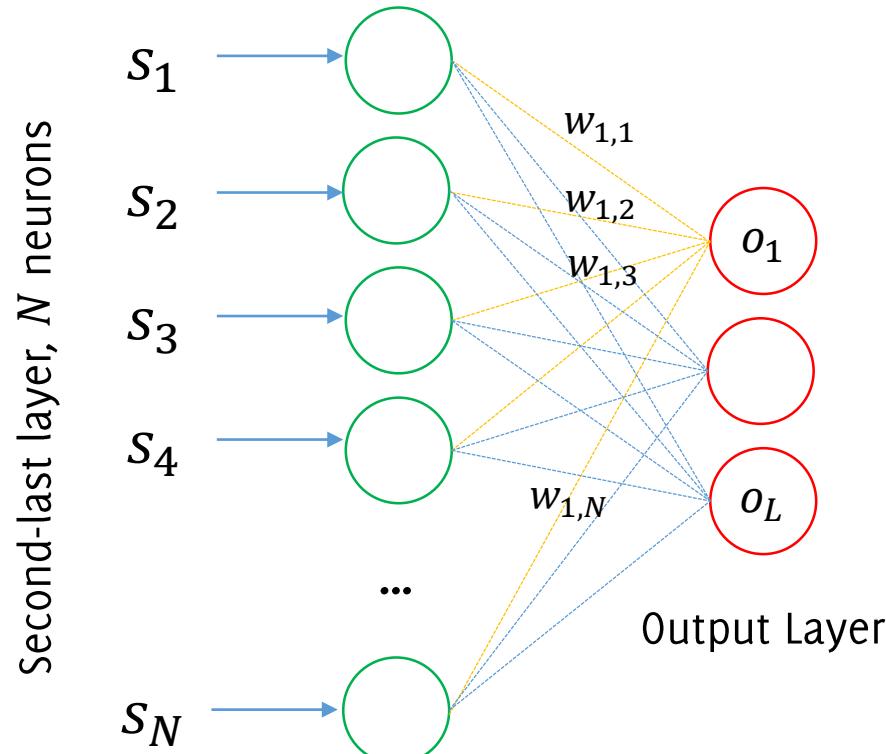
Thus, CNN cannot compute class scores, yet can extract features!



Fully Convolutional Neural Networks (FC-CNN)

However, since the FC is linear, it can be represented as convolution!

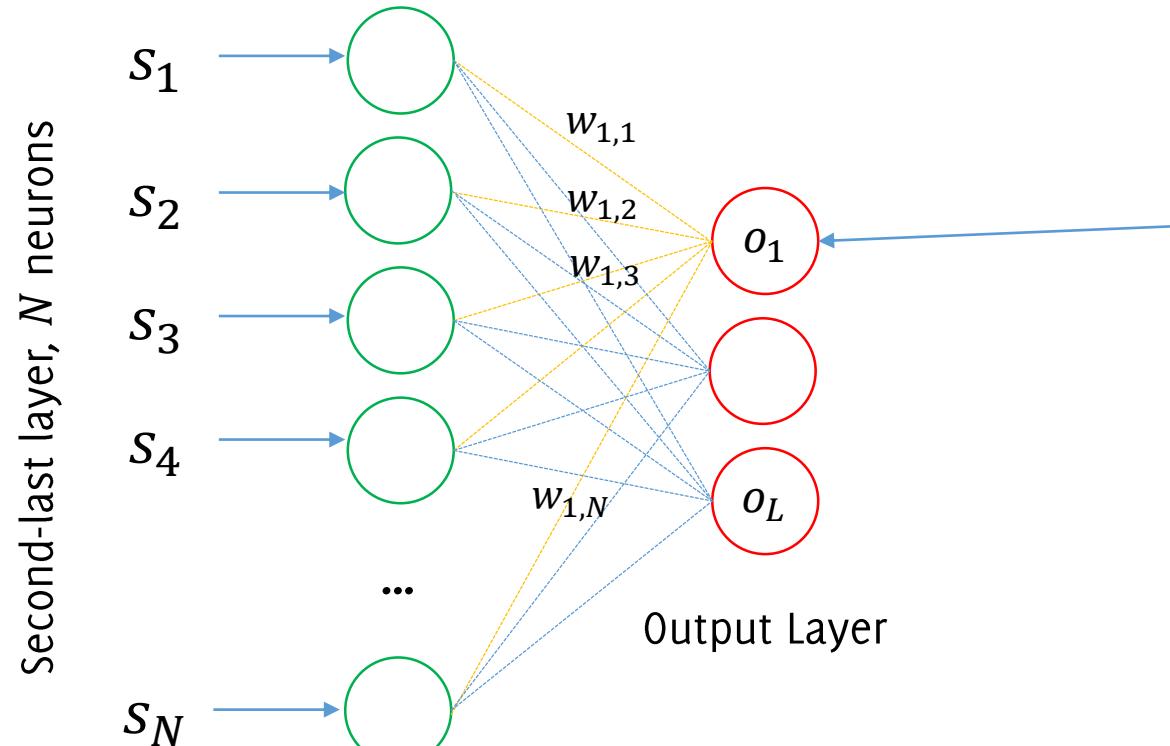
Weights associated to output neuron i : $\mathbf{w}_i = [w_{i,j}]_{j=1:N}$



Fully Convolutional Neural Networks (FC-CNN)

However, since the FC is linear, it can be represented as convolution!

Weights associated to output neuron i : $\mathbf{w}_i = [w_{i,j}]_{j=1:N}$.



$$o_i = \sum_{j=1:N} w_{i,j} s_j + b_i$$

$$o_i = (\mathbf{w}_i)' \mathbf{s} + b_i$$

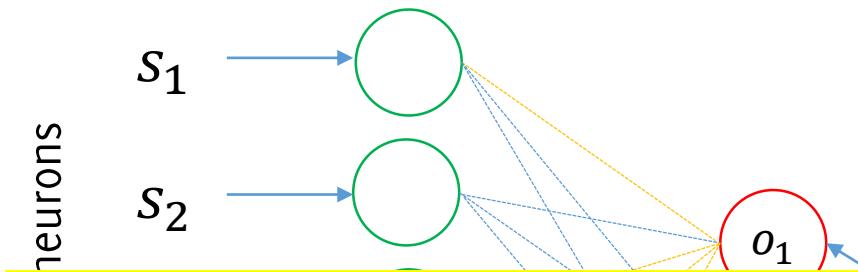
$$o_i = (\mathbf{w}_i \circledast \mathbf{s})(0,0) + b_i$$

A FC layer of L outputs is a 2DConv Layer against L filters having size $1 \times 1 \times N$

Fully Convolutional Neural Networks (FC-CNN)

However, since the FC is linear, it can be represented as convolution!

Weights associated to output neuron i : $\mathbf{w}_i = [w_{i,j}]_{j=1:N}$



$$o_i = \sum w_{i,j} s_j + b_i$$

This transformation can be applied to each hidden layer of a FC network placed at the CNN top

This transformation consists in reading weights from the neuron in the dense layer and recycling together with bias in the new convolutional layer.

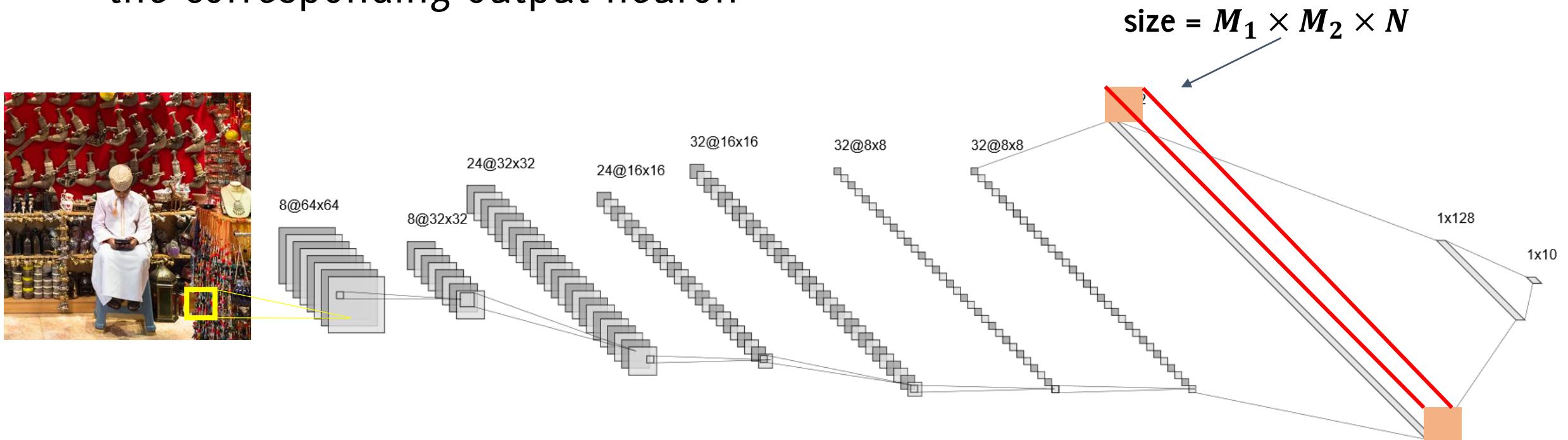
A FC layer
 $1 \times N$

$L \times$

Fully Convolutional Neural Networks (FC-CNN)

However, since the FC is linear, it can be represented as convolution against L filters of size $1 \times 1 \times N$

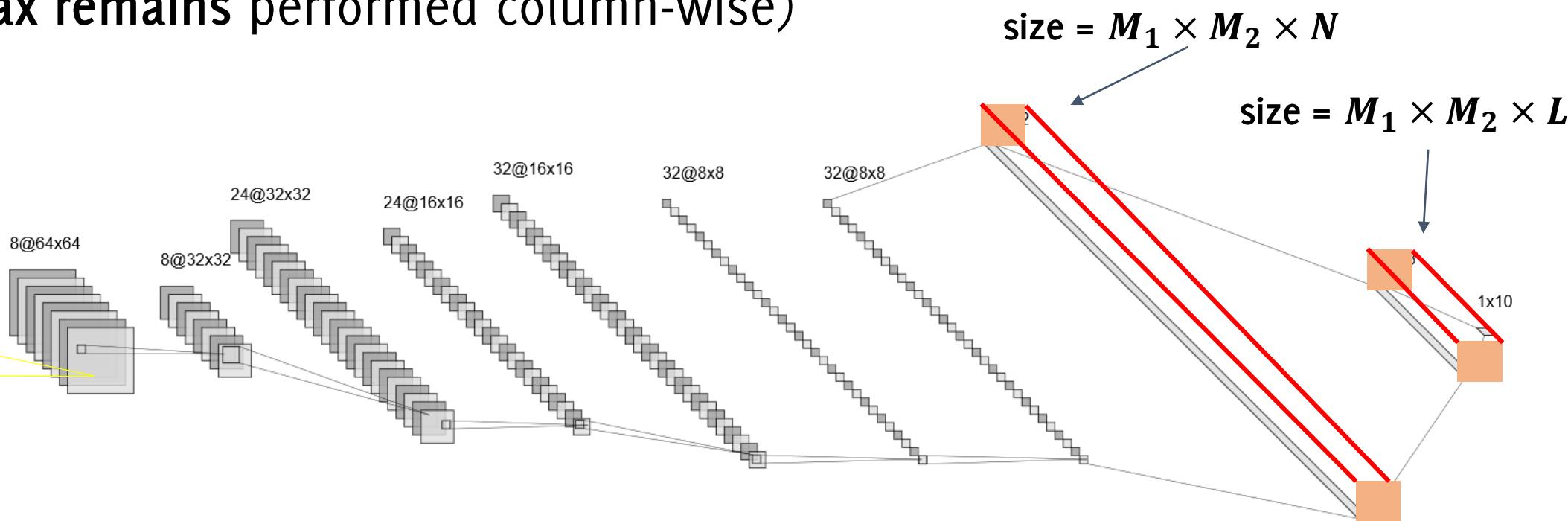
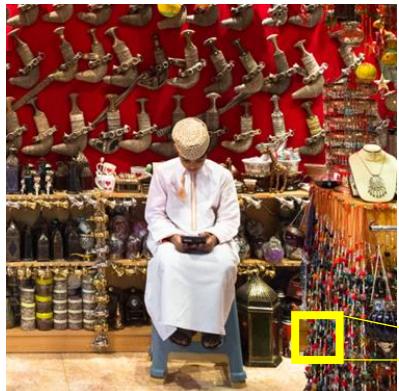
Each of these convolutional filters contains the weights of the FC for the corresponding output neuron



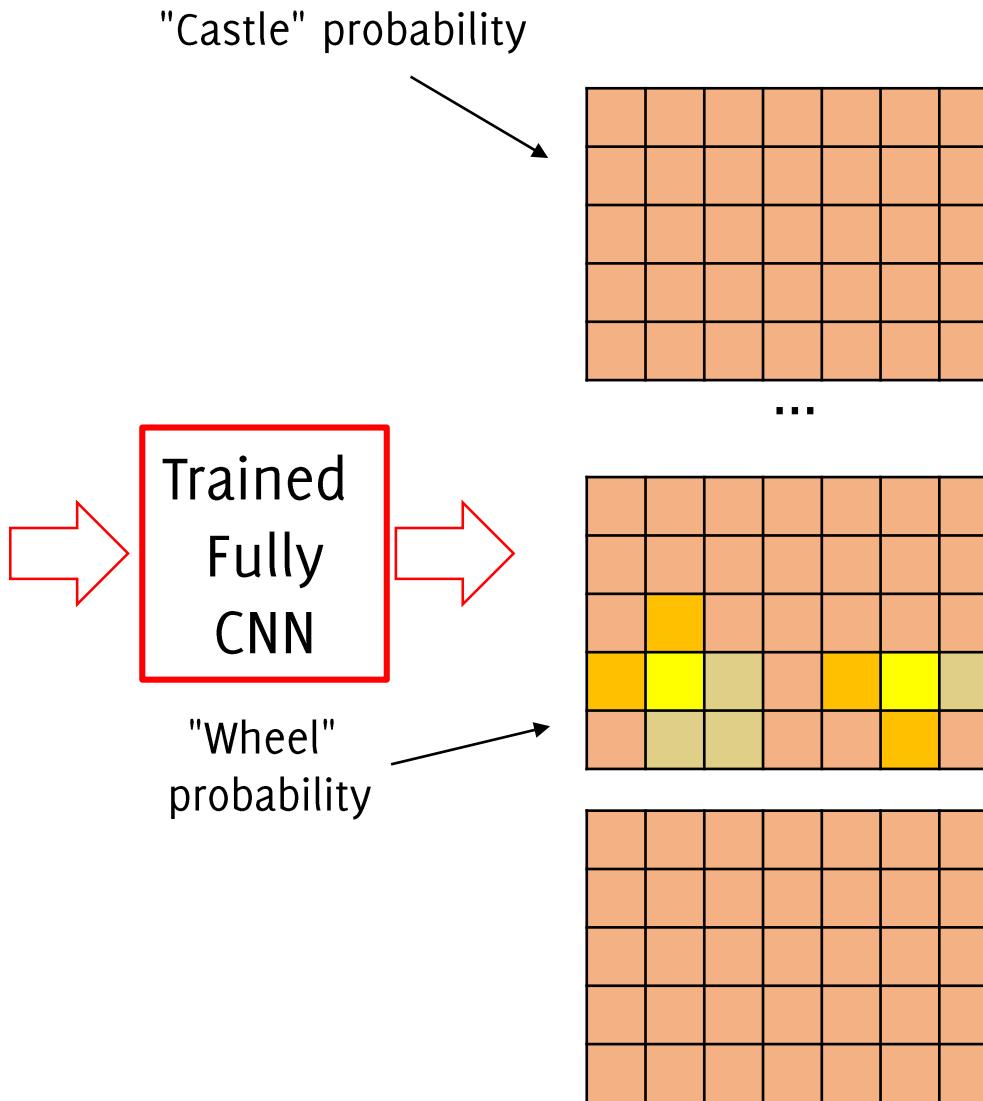
Fully Convolutional Neural Networks (FC-CNN)

For each output class we obtain an image, having:

- Lower resolution than the input image
- Class probabilities for the receptive field of each pixel (assuming softmax remains performed column-wise)

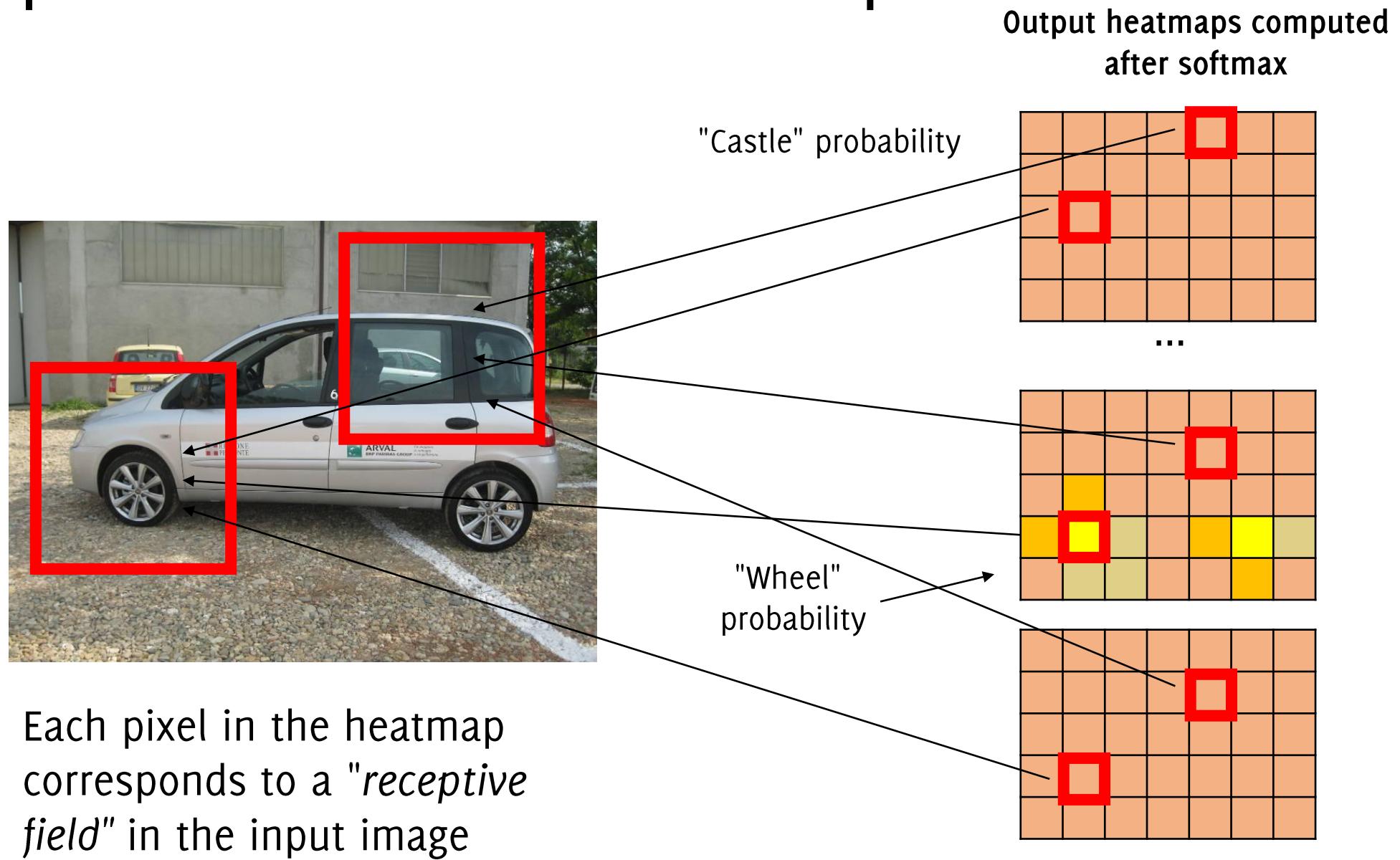


Output of a FC-CNN as heatmaps

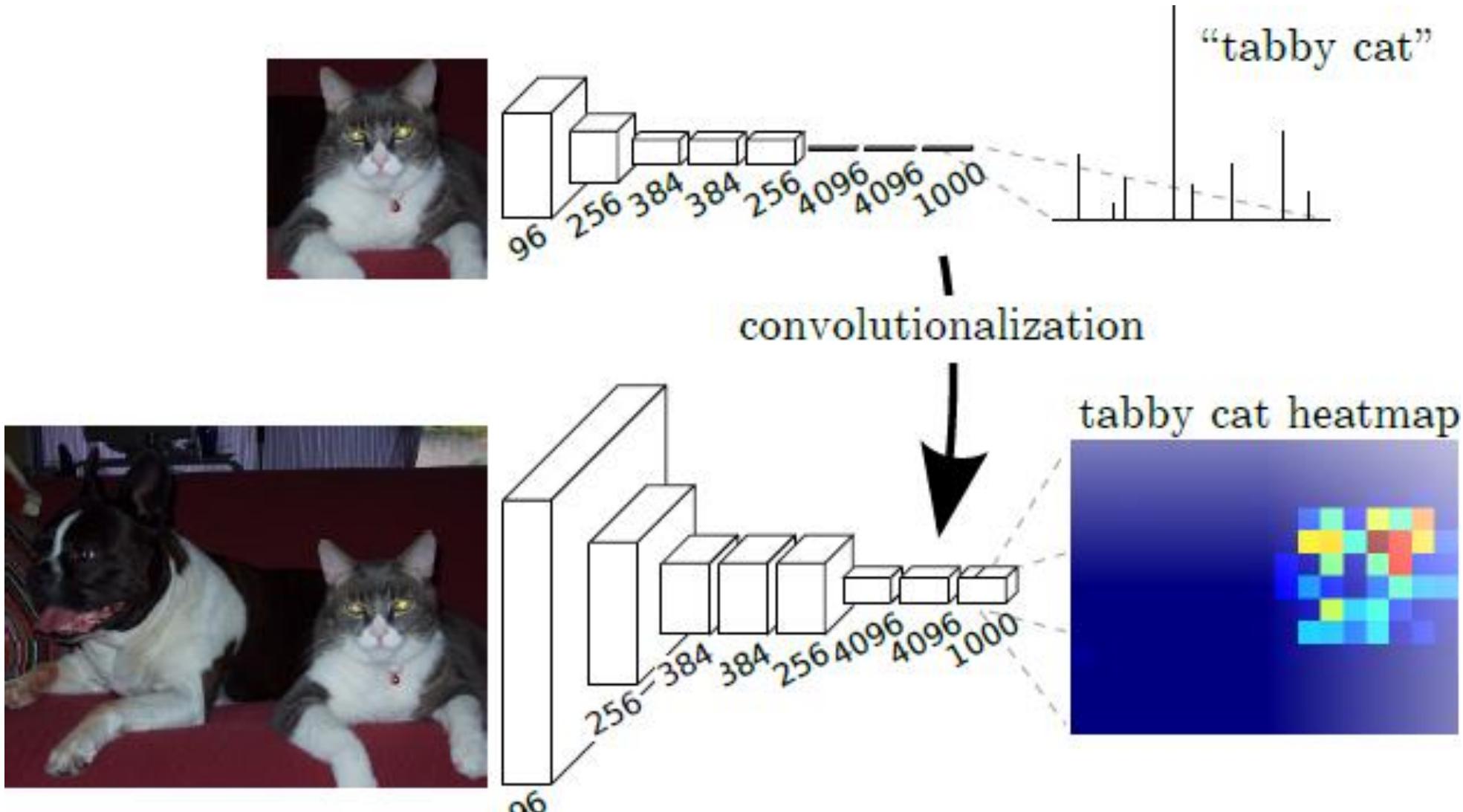


A larger image than
those used for training
the network

Output of a FCNN as heatmaps



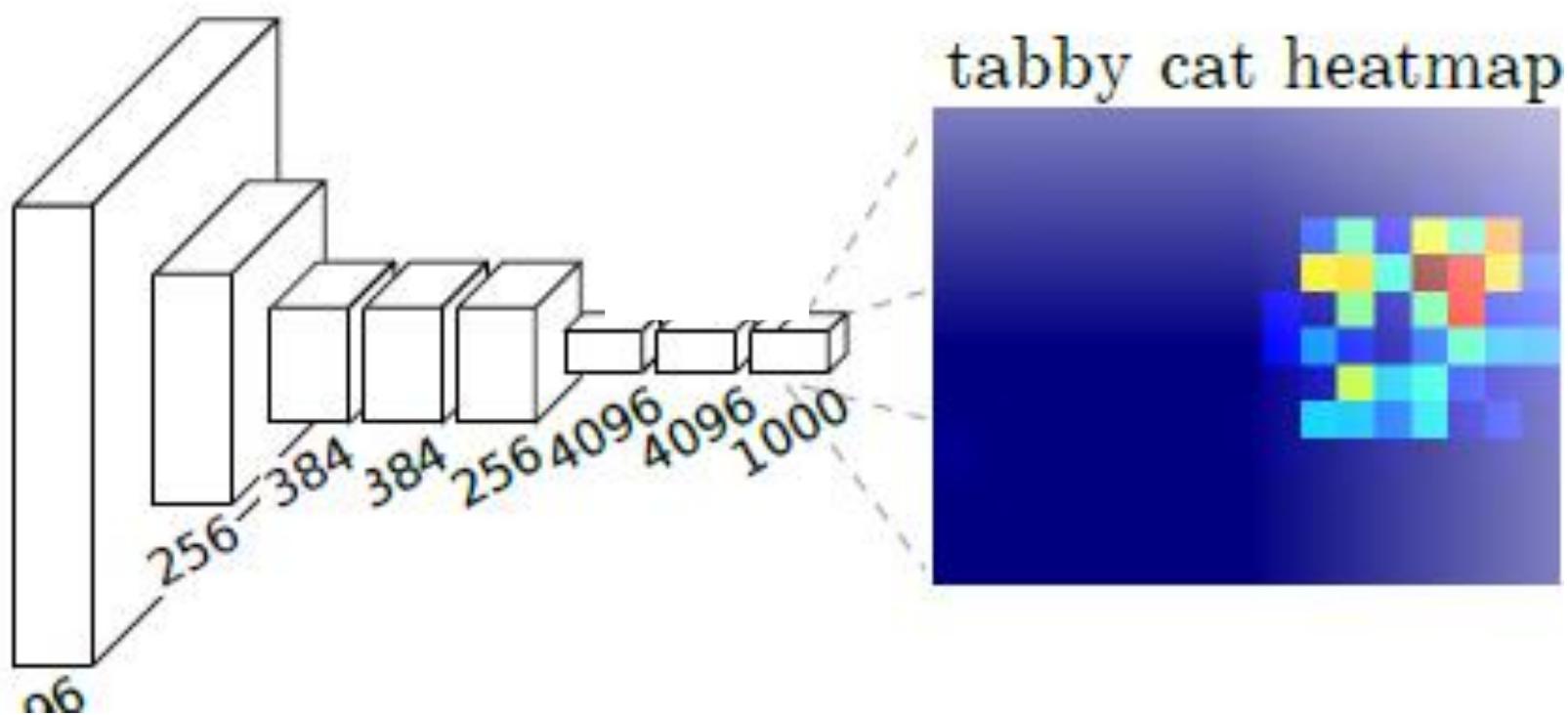
Migration to FCNN of a pretrained model



Migration to FCNN of a pretrained model

This stack of convolutions operates on the whole image as a filter.

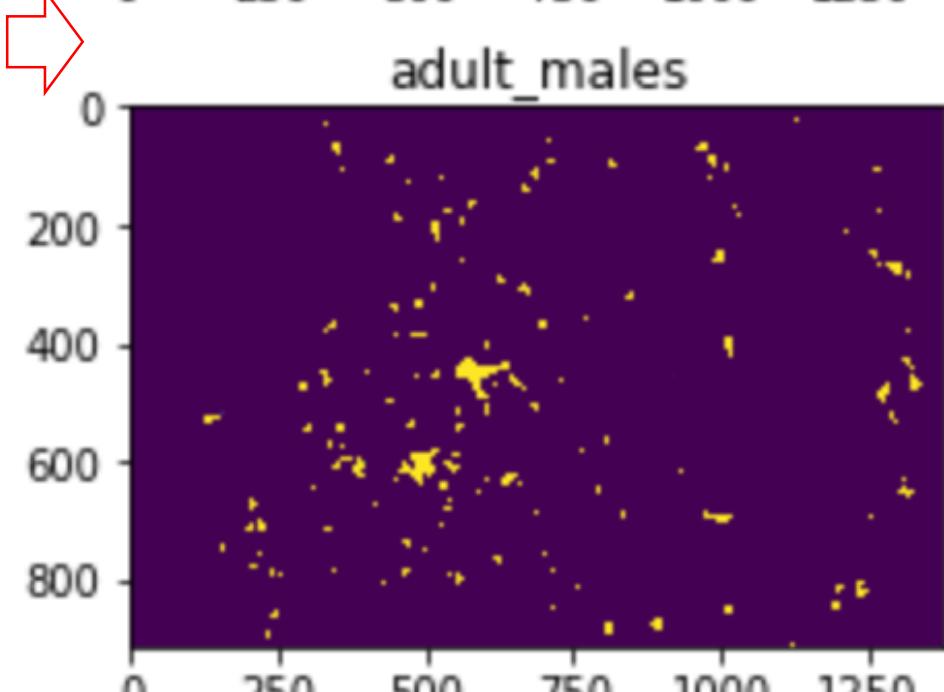
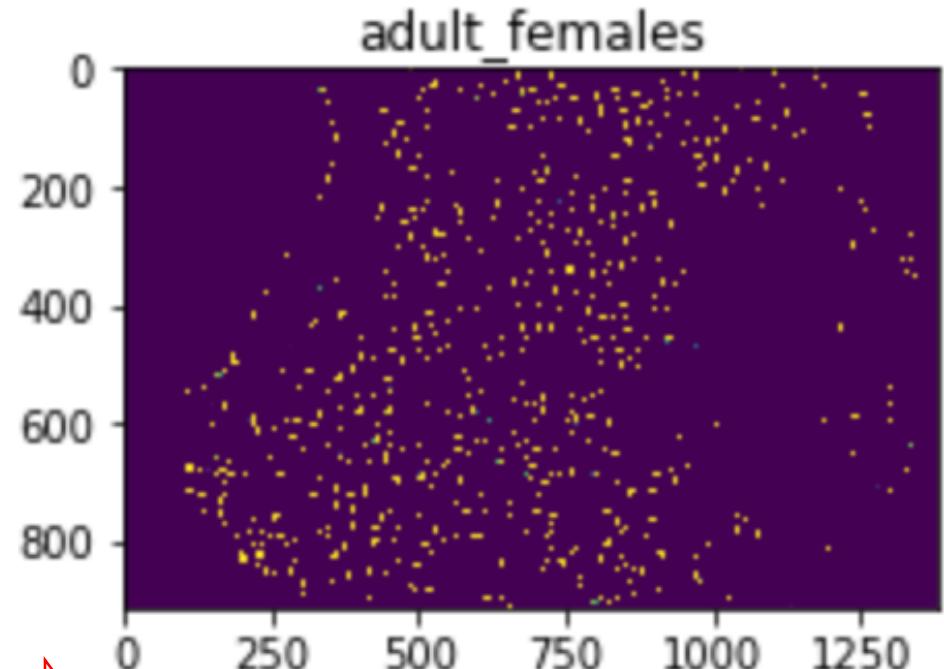
Significantly more efficient than patch extraction and classification: this avoids multiple repeated computations within overlapping patches



Sealion HeatMaps



FCNN



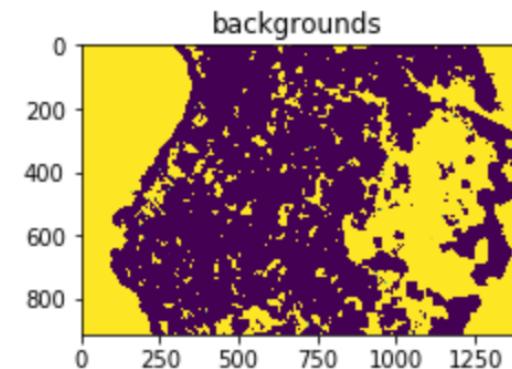
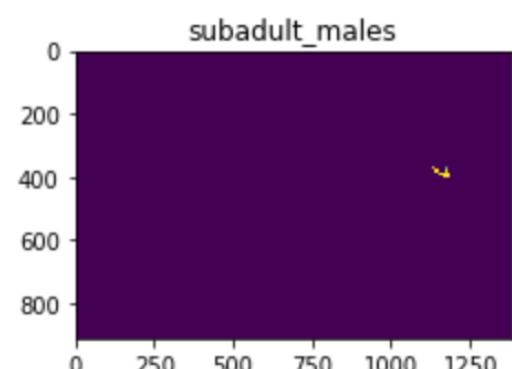
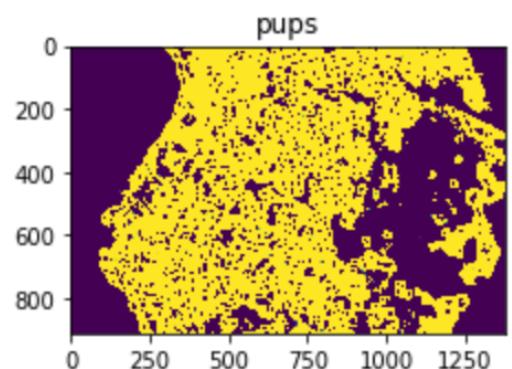
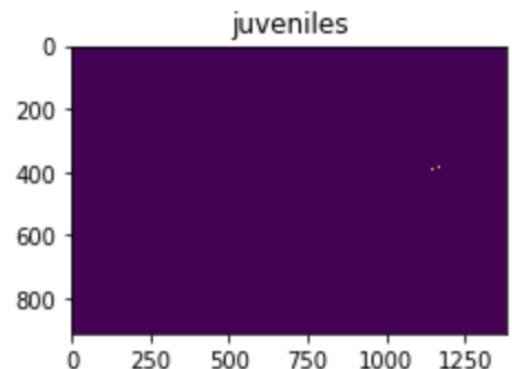
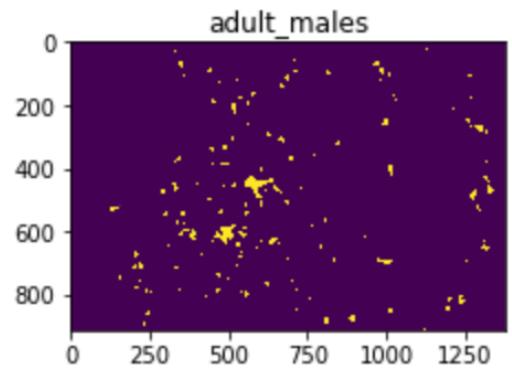
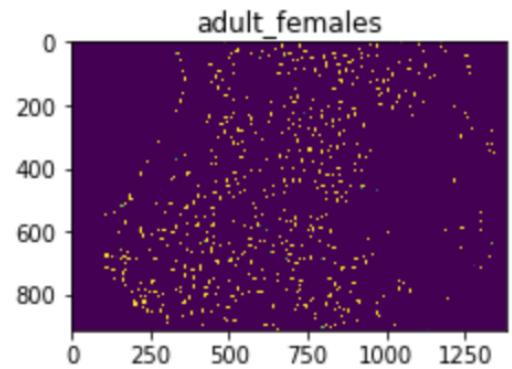
Credits Yinan Zhou

<https://github.com/marioZYN/FC-CNN-Demo>

Sealion HeatMaps



→ FCNN →



Credits Yinan Zhou

<https://github.com/marioZYN/FC-CNN-Demo>

Fully convolutional networks in keras

It is necessary to get and set the weights of networks by means of the methods `get_weights` and `set_weights`

- get the weights of the trained CNN

```
w7, b7 = model.layers[7].get_weights()
```

- reshape these weights to become ten 1x1 convolutional filters having depth equal to 256 (these sizes are dictated by the sizes of the dense layer in the “traditional” CNN)

```
w7.reshape(1, 1, 256, 10)
```

- assign these weights to the FCNN architecture

```
model2.layers[7].set_weights(w7, b7)
```

Visual Recognition Problems

Setting up the stage..

Image Classification



I

$$\Lambda = \{"wheel", "cars", \dots, "castle", "baboon"\}$$

→ “wheel”



I

→ “castle”

Image Classification



I

$$\Lambda = \{"wheel", "cars", \dots, "castle", "baboon"\}$$

→ “wheel” 65%, “tyre” 30%..



I

→ “castle” 55%, “tower” 43%..

Image Classification, the problem

Assign to an input image $I \in \mathbb{R}^{R \times C \times 3}$:

- a label l from a fixed set of categories
 $\Lambda = \{"wheel", "cars", ..., "castle", "baboon"\}$

$$I \rightarrow l \in \Lambda$$

Image Classification Example

Sat, Apr 6



Thu, Apr 4



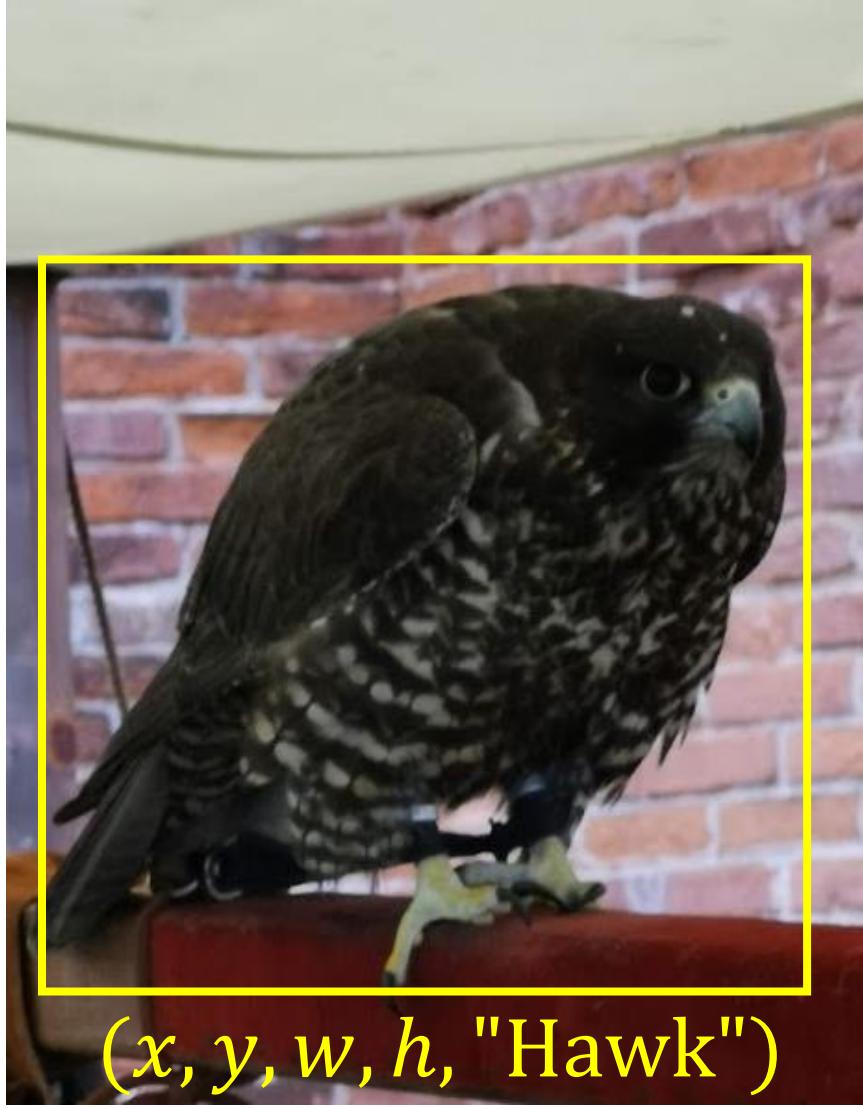
Sat, Apr 9, 2016



Inbox (39) - giacomo79@gmail.c x rabbit - Google Photos x +
← → C photos.google.com/search/rabbit

Search bar: rabbit

Localization



$(x, y, w, h, \text{"Hawk"})$

Localization, the problem

Assign to an input image $I \in \mathbb{R}^R \times C \times 3$:

- a label l from a fixed set of categories
 $\Lambda = \{"wheel", "cars", ..., "castle", "baboon"\}$
- the coordinates (x, y, h, w) of the bounding box enclosing that object

$$I \rightarrow (x, y, h, w, l)$$

Object Detection



Object Detection, the problem

Assign to an input image $I \in \mathbb{R}^{R \times C \times 3}$:

- **multiple** labels $\{l_i\}$ from a fixed set of categories $\Lambda = \{"wheel", "cars", \dots, "castle", "baboon"\}$, each corresponding to an instance of that object
- the coordinates $\{(x, y, h, w)_i\}$ of the bounding box enclosing each object

$$I \rightarrow \{(x, y, h, w, l)_1, \dots, (x, y, h, w, l)_N\}$$

Semantic Segmentation

Objects appearing in the image:

Boat

Dining table

Person



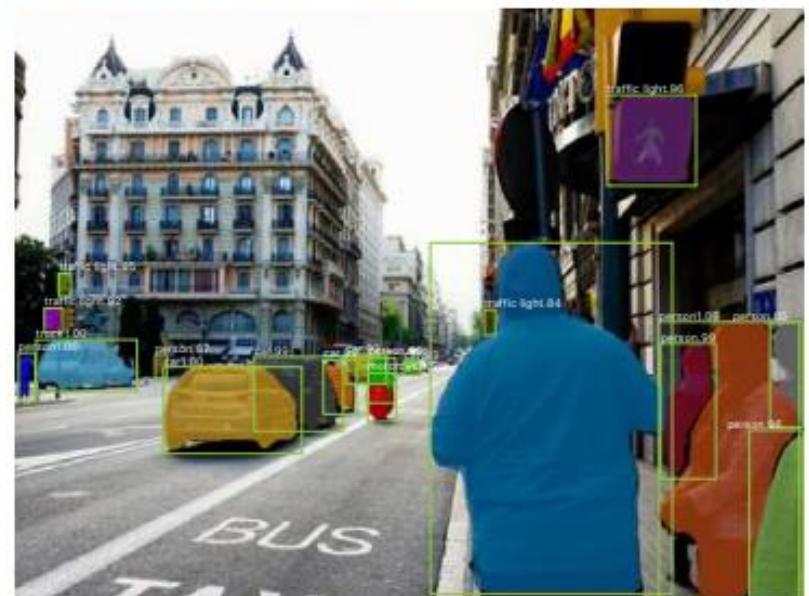
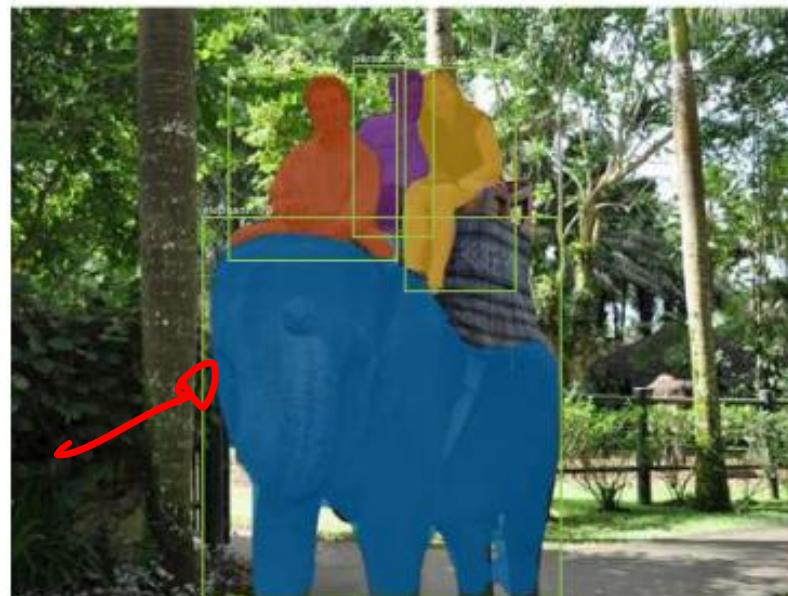
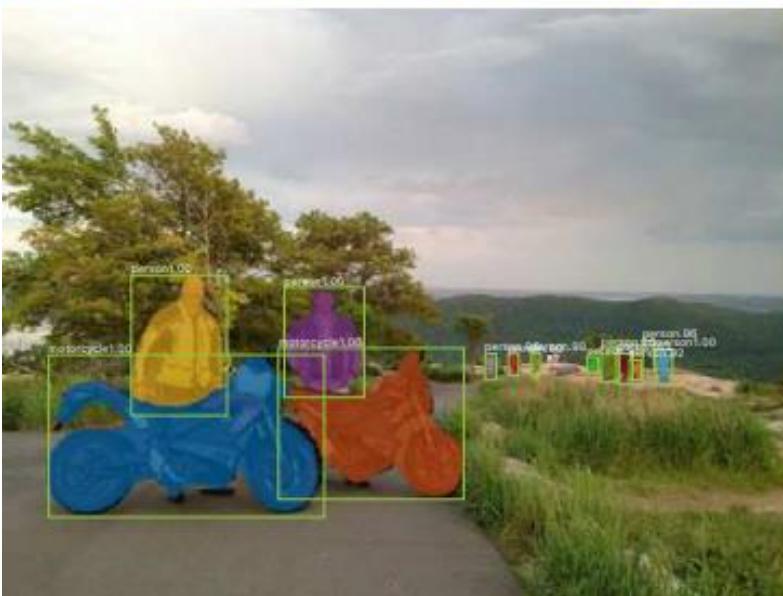
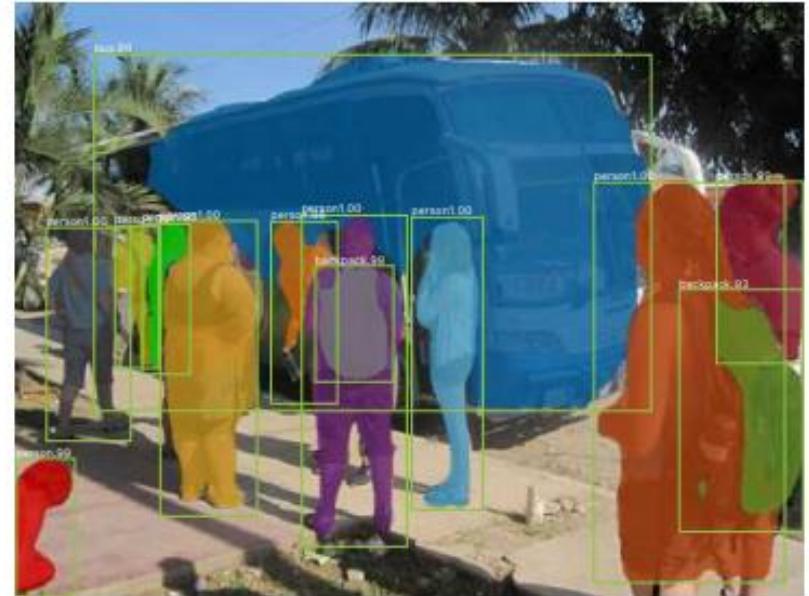
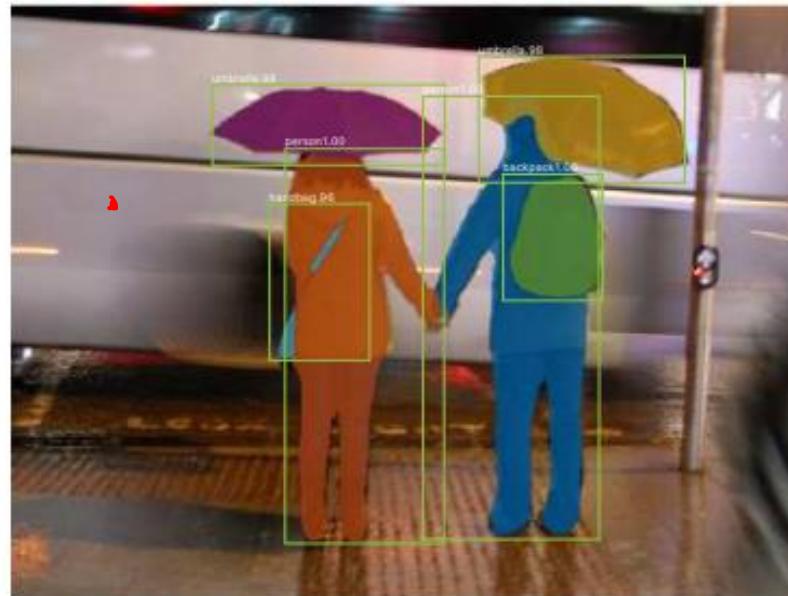
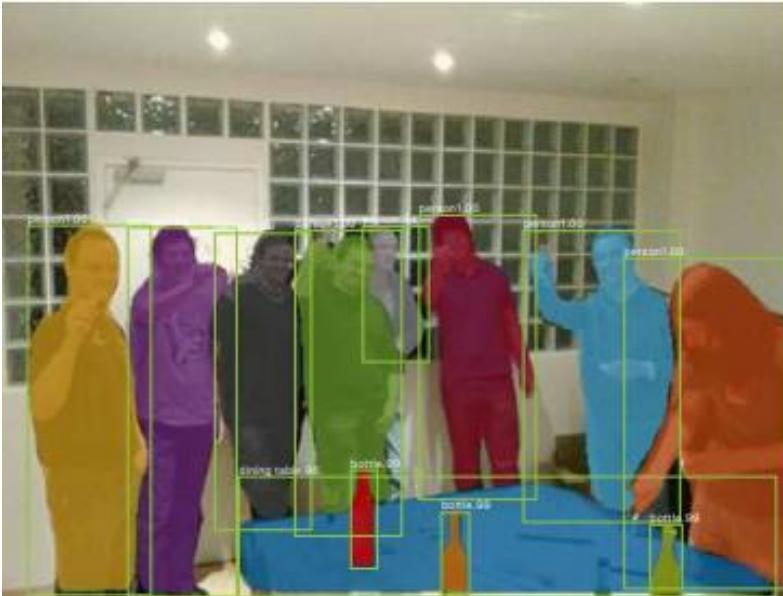
Semantic Segmentation, the problem

Assign to each pixel of an image $I \in \mathbb{R}^{R \times C \times 3}$:

- a label $\{l_i\}$ from a fixed set of categories
 $\Lambda = \{"wheel", "cars", \dots, "castle", "baboon"\},$
 $I \rightarrow S \in \Lambda^{R \times C}$

where $S(x, y) \in \Lambda$ denotes the class associated to the pixel (x, y)

Instance Segmentation: Mask R-CNN



Instance Segmentation, the problem

Assign to an input image I :

- **multiple** labels $\{l_i\}$ from a fixed set of categories $\Lambda = \{"wheel", "cars", \dots, "castle", "baboon"\}$, each corresponding to **an instance of that object**
- the coordinates $\{(x, y, h, w)_i\}$ of the **bounding box** enclosing each **object**
- the **set of pixels** S in each bounding box corresponding to that label

$$I \rightarrow \{(x, y, h, w, l, S)_1, \dots, (x, y, h, w, l, S)_N\}$$

Instance Segmentation



Semantic Segmentation by Fully Convolutional Neural Networks

Predicting dense outputs for arbitrary-sized inputs

-

Semantic Segmentation Task

The goal of semantic segmentation is:

Given an image I , associate to each pixel (r, c) a label from Λ .

The result of segmentation is a map of labels containing in each pixel the estimated class.

Remark: In this image there is no distinction among persons.

Segmentation does not separate different instances belonging to the same class.

That would be instance segmentation.



Training Set

The training set is made of pairs (I, GT) , where the GT is a pixel-wise annotated image over the categories in Λ



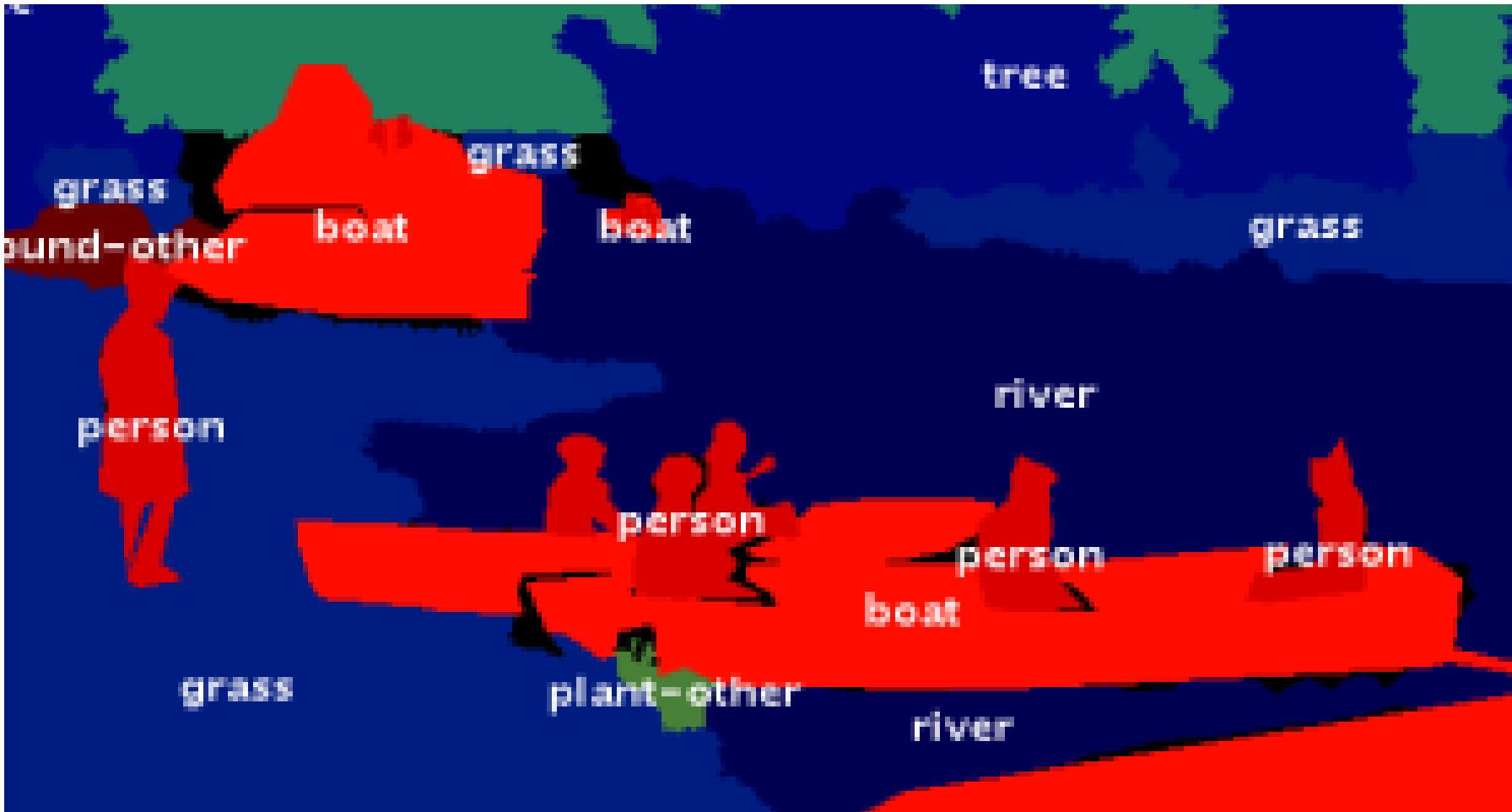
I



GT

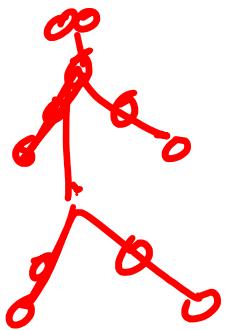
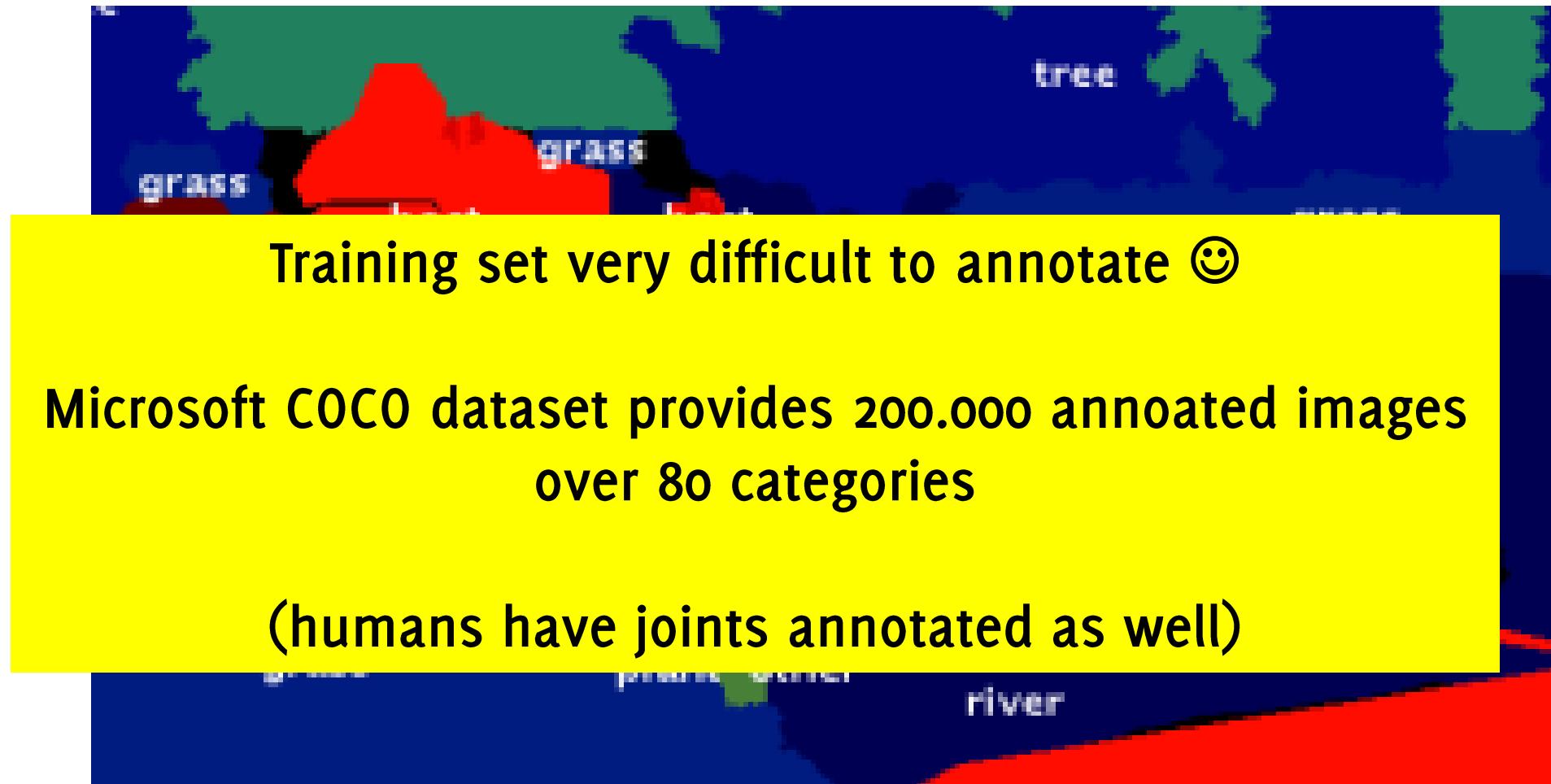
Training Set

The training set is made of pairs (I, GT) , where the GT is a pixel-wise annotated image over the categories in Λ



Training Set

The training set is made of pairs (I, GT) , where the GT is a pixel-wise annotated image over the categories in Λ





This CVPR2015 paper is the Open Access version, provided by the Computer Vision Foundation.
The authoritative version of this paper is available in IEEE Xplore.

Fully Convolutional Networks for Semantic Segmentation

Jonathan Long*

Evan Shelhamer*

Trevor Darrell

UC Berkeley

{jonlong, shelhamer, trevor}@cs.berkeley.edu

From Classification to Segmentation

Setup:

- You are given a pre-trained CNN for classification
- You possibly performed transfer learning to the problem at hand
- You have «convolutionalized» it to extract heatmaps

Goal:

- Obtain a Semantic Segmentation network for images of arbitrary size

Simple Solution (1): Direct Heatmap Predictions

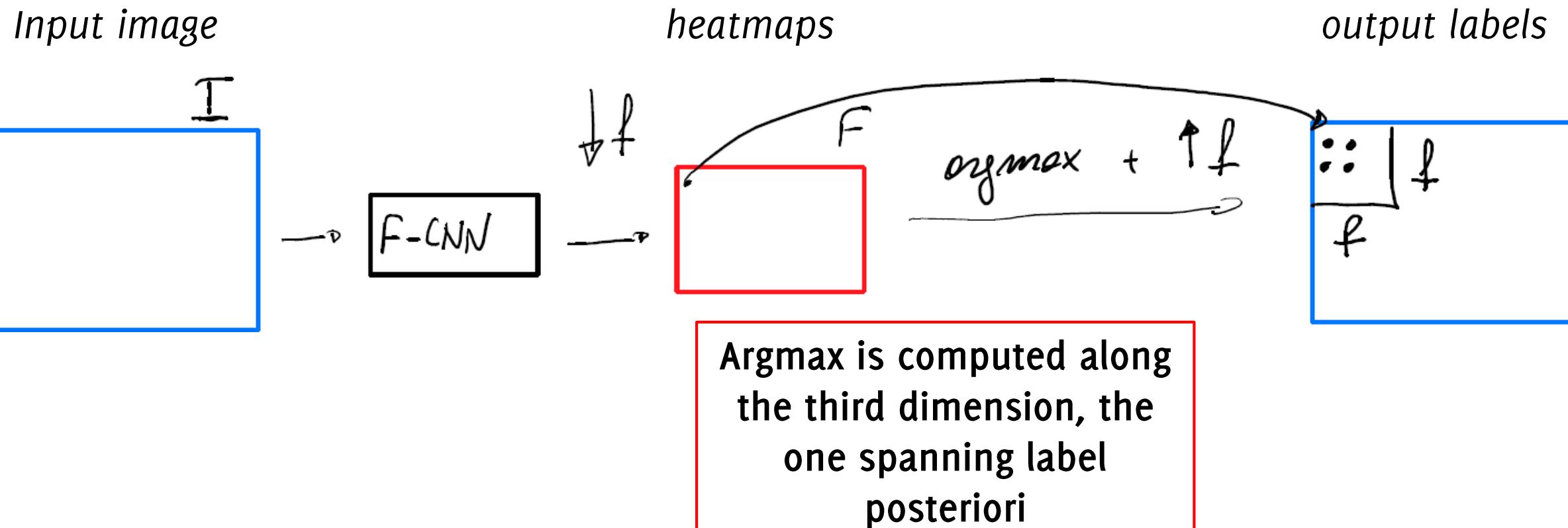
We can assign the predicted label in the heatmap to the whole receptive field, however that would be a **very coarse estimate**



We need to upsample the estimated map!

Simple Solution (1): Direct Heatmap Predictions

Very coarse estimates



Simple Solution (2): The Shift and Stich

Shift and Stich: Assume there is a downsampling ratio f between the size of input and of the output heatmap

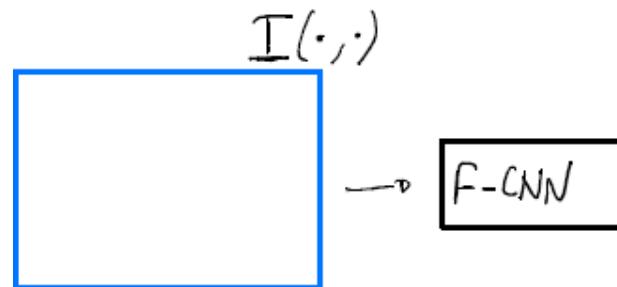
- Compute heatmaps for all f^2 possible shifts of the input ($0 \leq r, c < f$)
- Map predictions from the f^2 heatmaps to the image: each pixel in the heatmap provides prediction of the central pixel of the receptive field
- Interleave the heatmaps to form an image as large as the input

This exploits the whole depth of the network

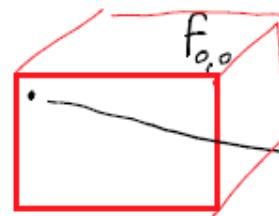
Efficient implementation through the à trous algorithm in wavelet

However, the upsampling method is very rigid

Input image

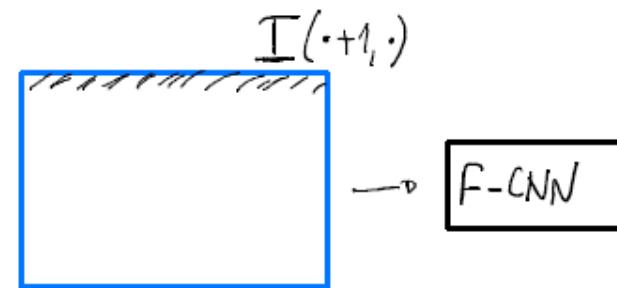


heatmaps

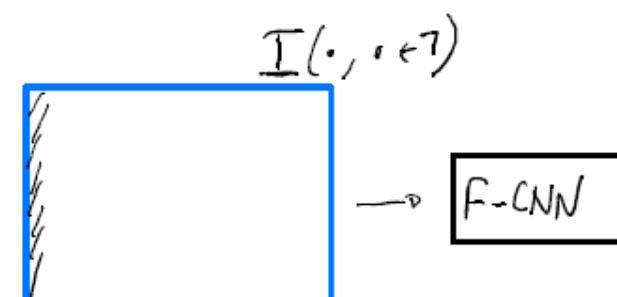


output labels

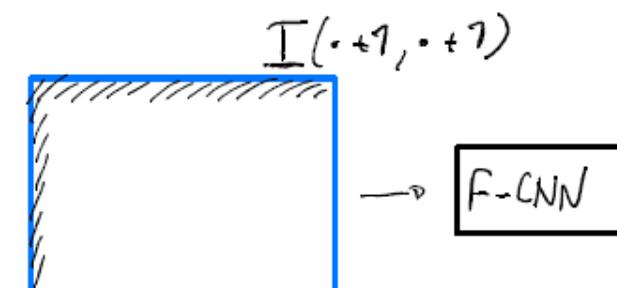
$$\text{softmax}(F_{0,0}, 3)$$



$$\text{softmax}(F_{1,0}, 3)$$



$$\text{softmax}(F_{0,1}, 3)$$



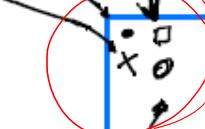
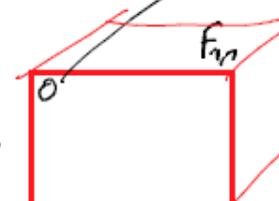
That's a very rigid form of upsampling!

$\downarrow 2$

$\downarrow 2$

$\downarrow 2$

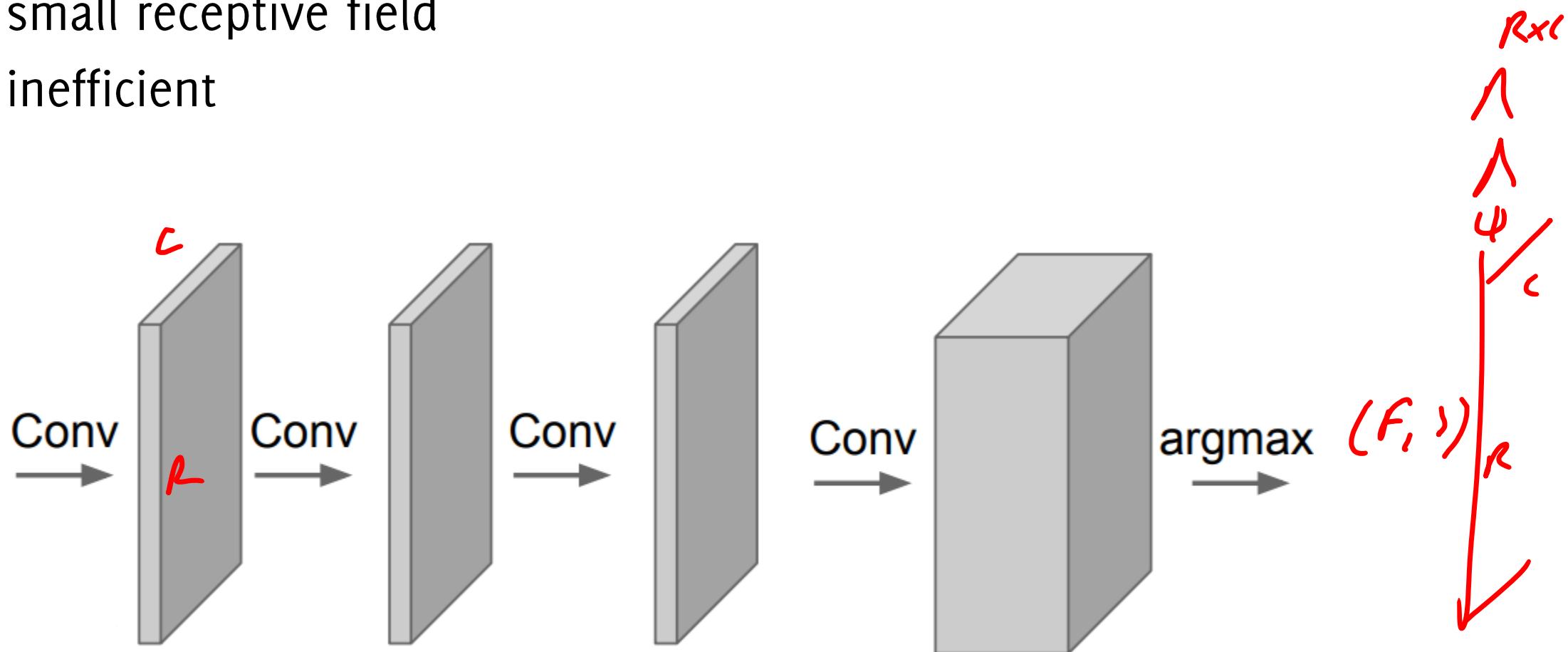
$\downarrow 2$



Simple Solution (3): Only Convolutions

What if we avoid any pooling (just conv2d and activation layers)?

- Very small receptive field
- Very inefficient



Drawbacks of convolutions only

On the one hand **we need to “go deep” to extract high level information on the image**

On the other hand **we want to stay local not to loose spatial resolution in the predictions**

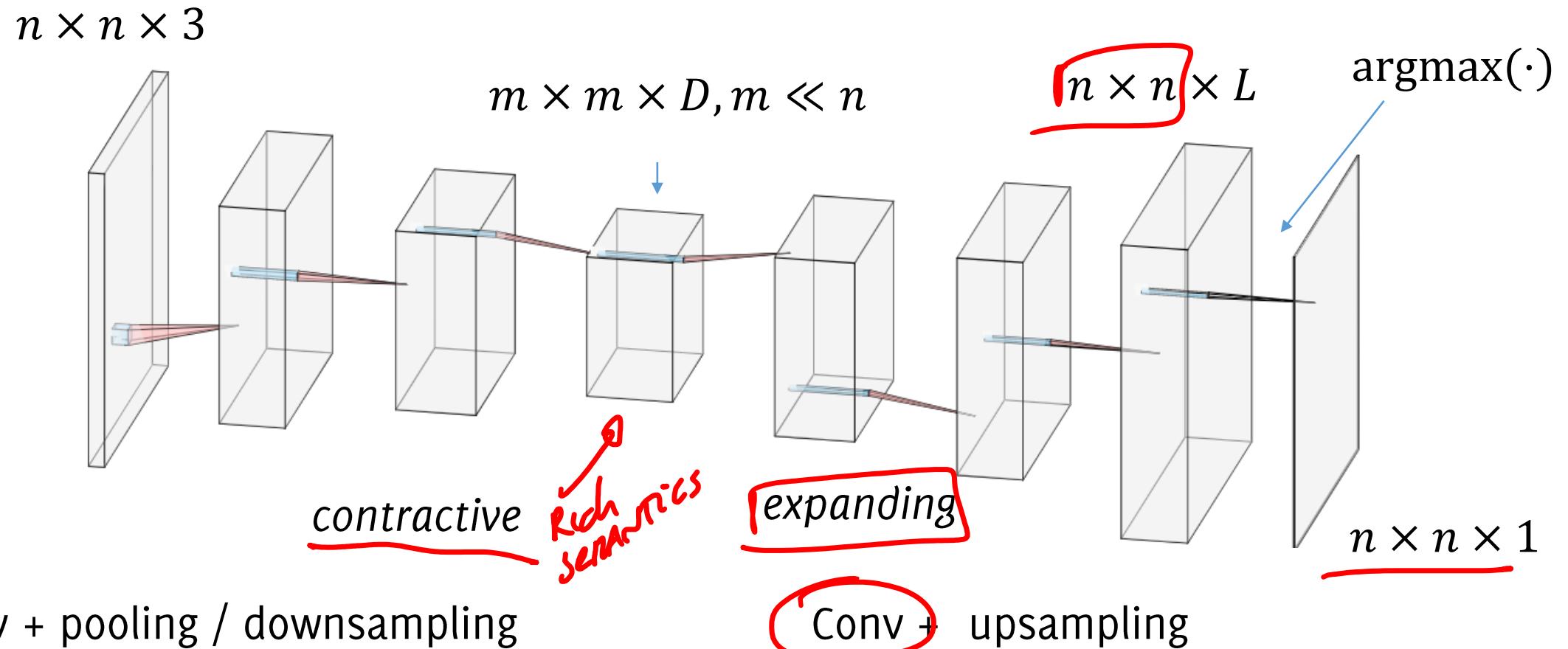
Semantic segmentation faces an inherent tension between semantics and location:

- *global information resolves what, while*
- *local information resolves where*

Combining fine layers and coarse layers lets the model make local predictions that respect global structure.

Reduce latent representation dimension

An architecture like the following would probably be more suitable for semantic segmentation

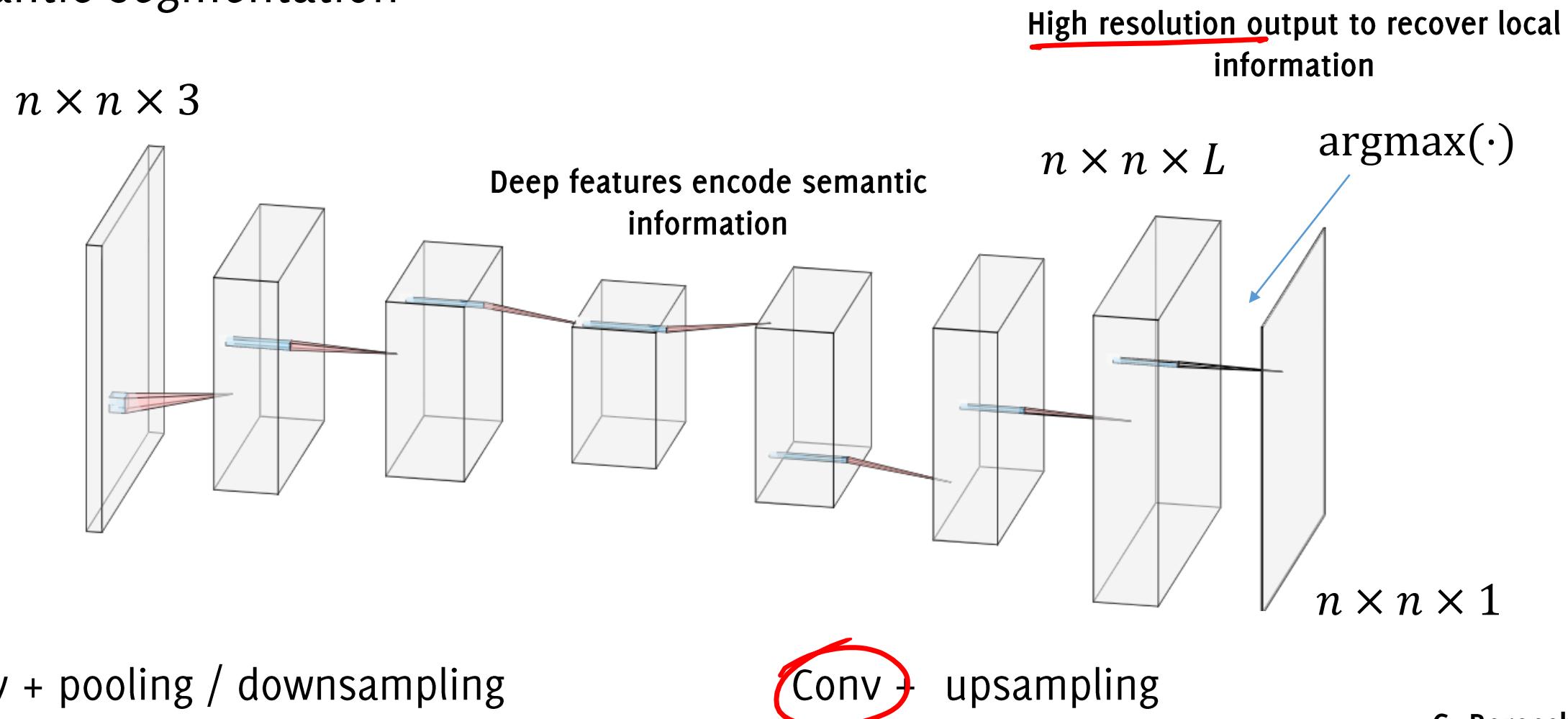


Conv + pooling / downsampling

Conv + upsampling

Reduce latent representation dimension

An architecture like the following would probably be more suitable for semantic segmentation



Conv + pooling / downsampling

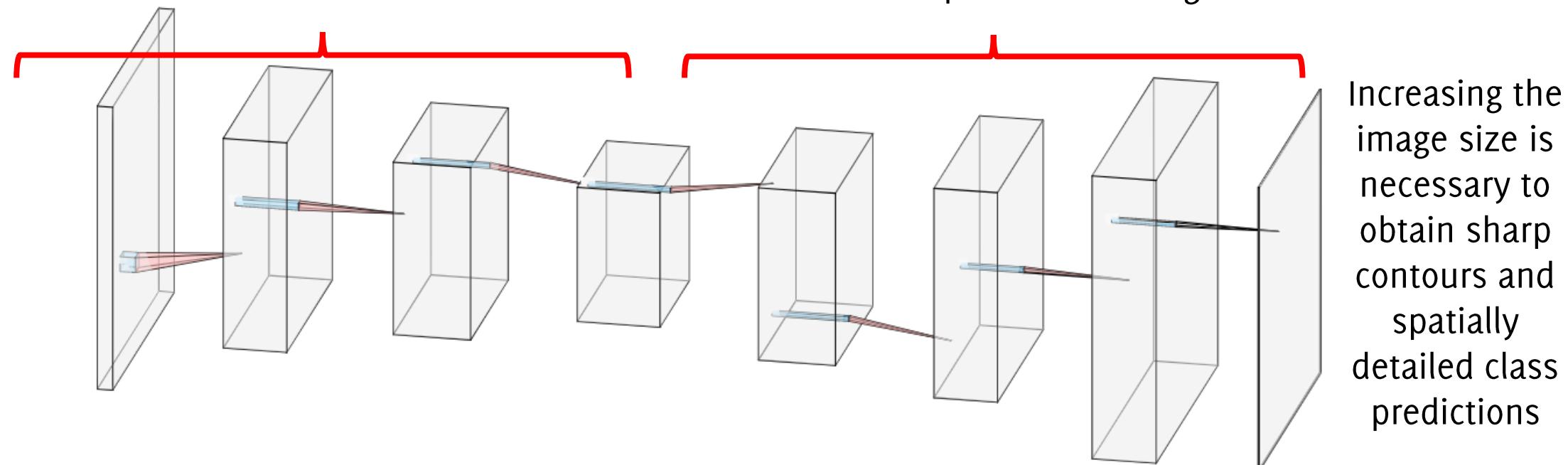
Conv - upsampling

Reduce latent representation dimension

An architecture like the following would probably be more suitable for semantic segmentation

The first half is the same of a classification network

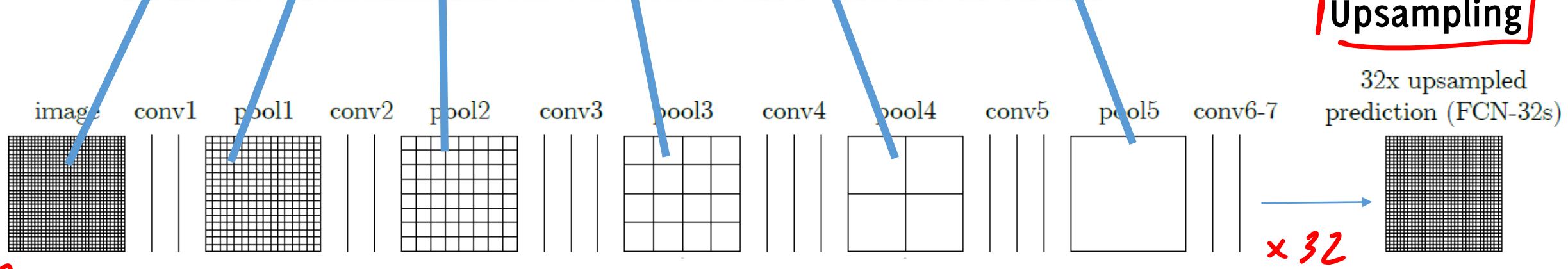
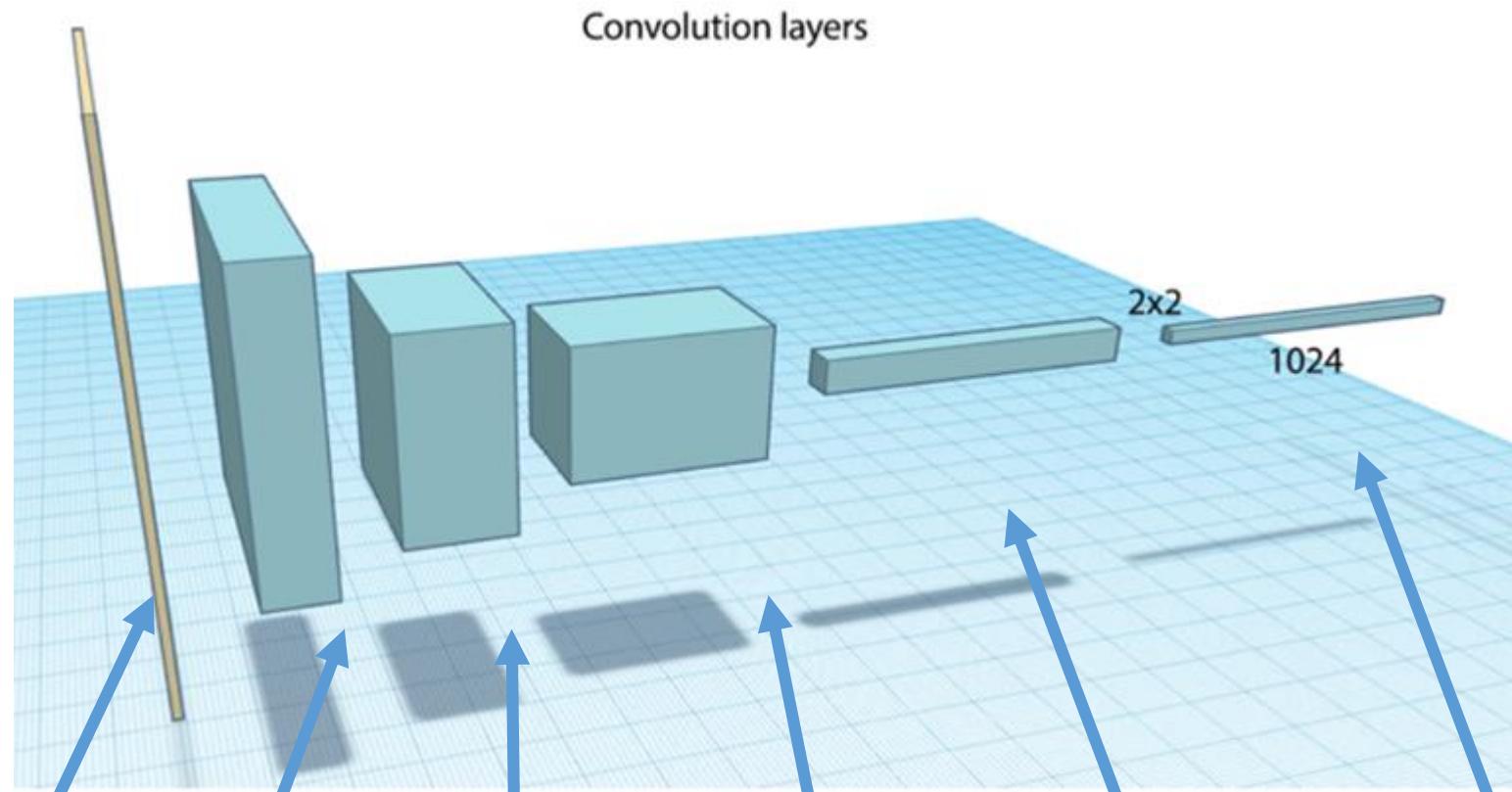
The second half is meant to upsample the predictions to cover each pixel in the image



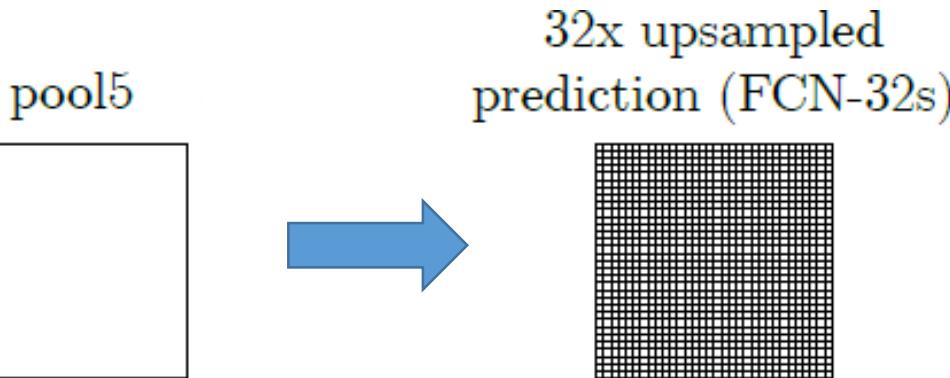
Conv + pooling / downsampling

Conv + upsampling

Increasing the image size is necessary to obtain sharp contours and spatially detailed class predictions



How to perform upsampling?



Nearest Neighbor

1	2
3	4



1	1	2	2
1	1	2	2
3	3	4	4
3	3	4	4

Input: 2 x 2

Output: 4 x 4

“Bed of Nails”

1	2
3	4



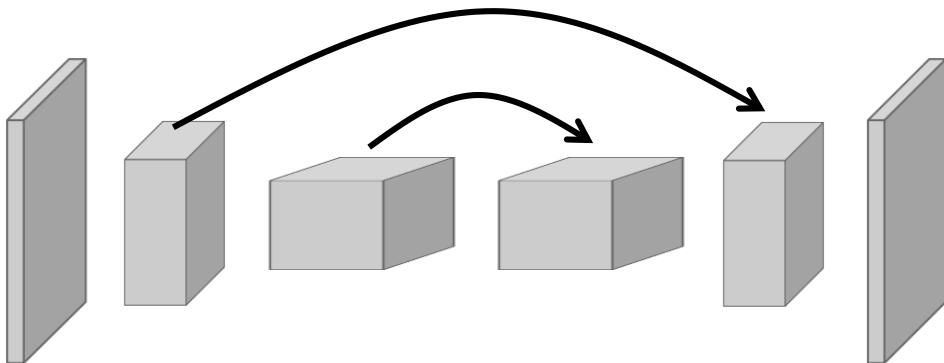
1	0	2	0
0	0	0	0
3	0	4	0
0	0	0	0

Input: 2 x 2

Output: 4 x 4

Max Unpooling

You have to keep track of the locations of the max during maxpooling



Max Pooling

Remember which element was max!

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

Input: 4 x 4

Output: 2 x 2

Max Unpooling

Use positions from pooling layer

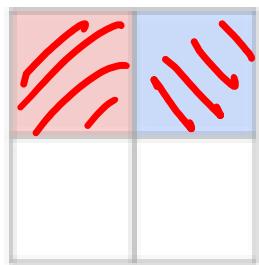
0	0	2	0
1	2	0	0
3	4	0	0
3	0	0	4

Input: 2 x 2

Output: 4 x 4

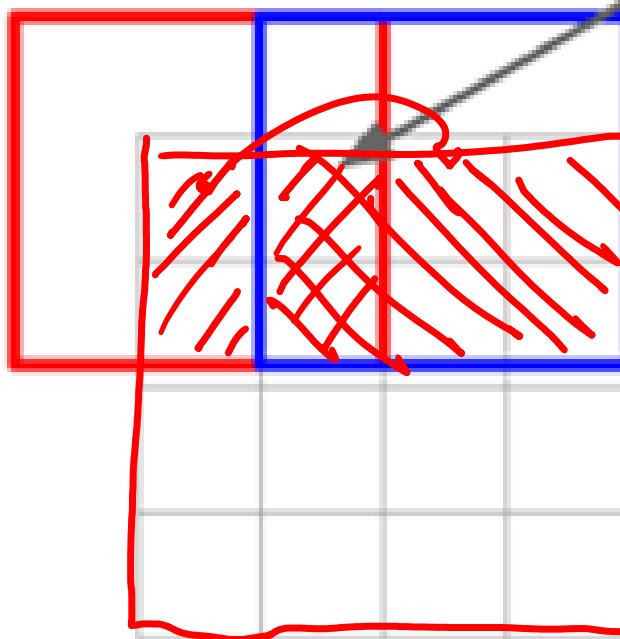
Transpose Convolution

3 x 3 transpose convolution, stride 2 pad 1



Input gives weight for a 3x3 filter

Input: 2 x 2



Output: 4 x 4

Sum where output overlaps

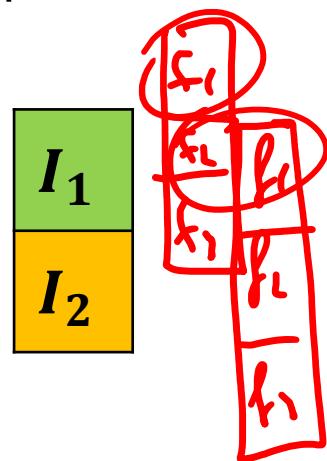
Filter moves 2 pixels in the output for every one pixel in the input

Stride gives ratio between movement in output and input

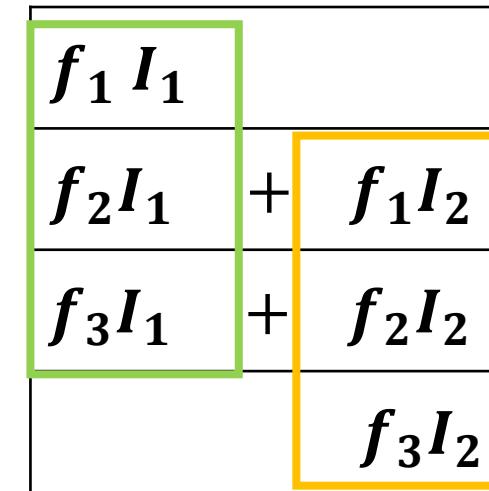
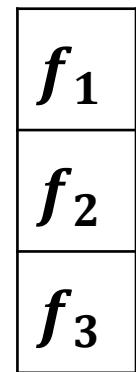
Transpose Convolution

Transpose convolution with stride 1

Input 2×1



Filter 3×1

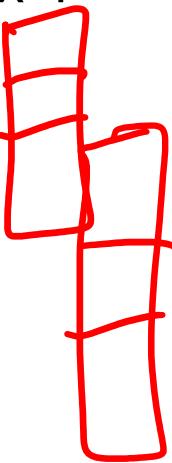


Transpose Convolution

Transpose convolution with stride 2

Input 2×1

$$\begin{matrix} I_1 \\ I_2 \end{matrix}$$



Filter 3×1

$$\begin{matrix} f_1 \\ f_2 \\ f_3 \end{matrix}$$

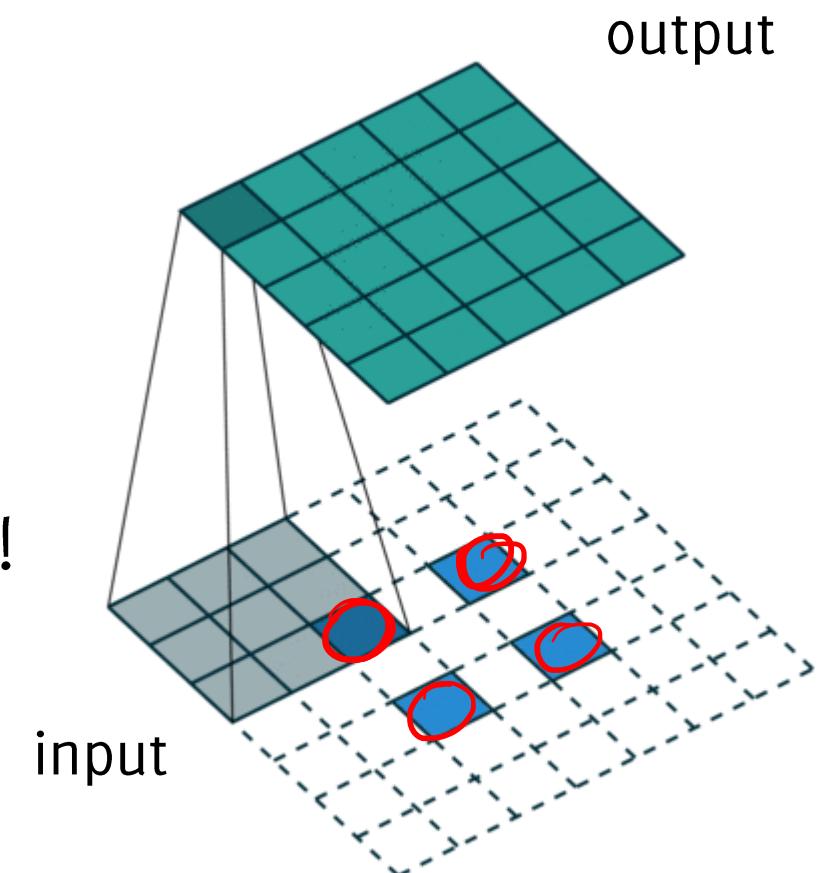
$$\begin{array}{c|c} f_1 I_1 & \\ \hline f_2 I_1 & \\ \hline f_3 I_1 & + f_1 I_2 \\ \hline f_2 I_2 & \\ \hline f_3 I_2 & \end{array}$$

How to perform upsampling?

Transpose Convolution can be seen as a traditional convolution after having upsampled the input image

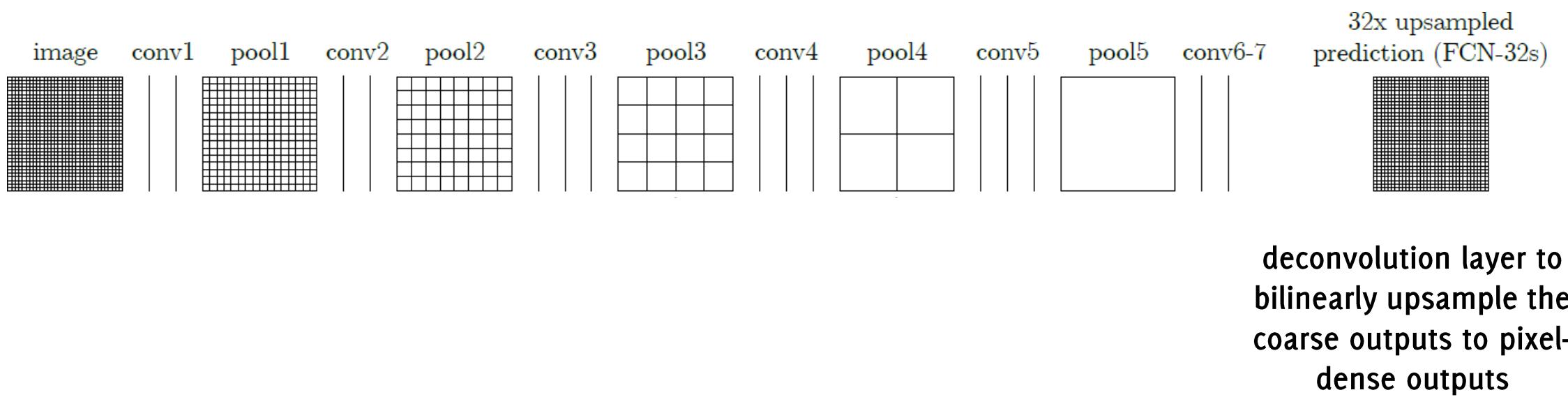
Many names for transpose convolution: fractional strided convolution, backward strided convolution, ~~deconvolution~~ (very misleading!!!)

Upsampling based on convolution gives more degrees of freedom, since the **filters can be learned!**



https://github.com/vdumoulin/conv_arithmetic

FC-CNN Upsampling



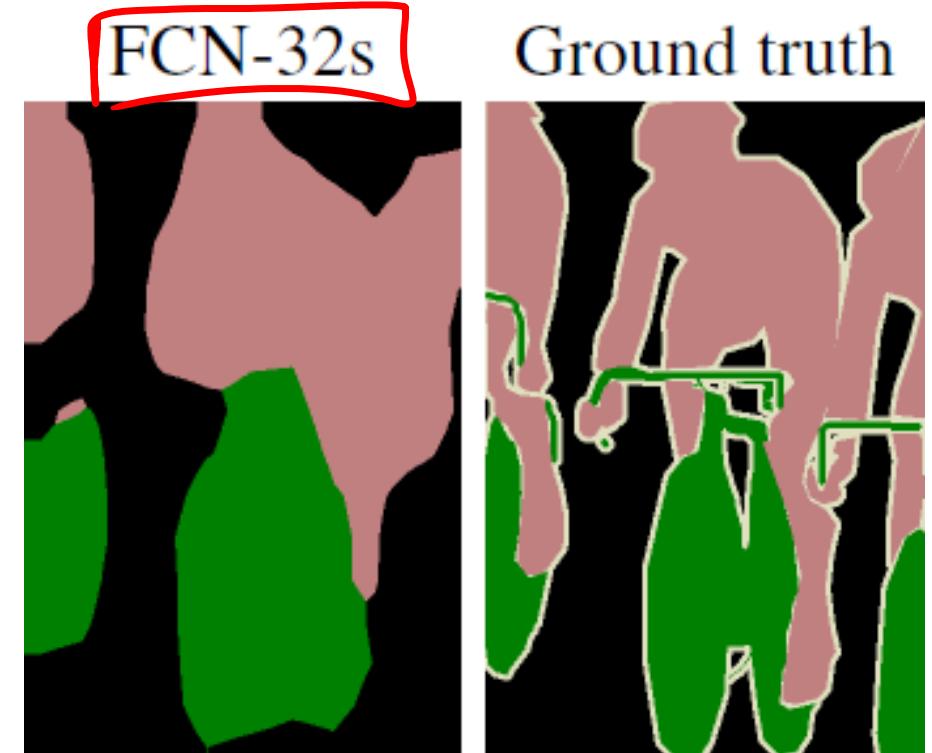
Prediction Upsampling

Linear upsampling of a factor f can be implemented as a convolution against a filter with a fractional stride $1/f$.

Upsampling filters can thus be learned during network training.

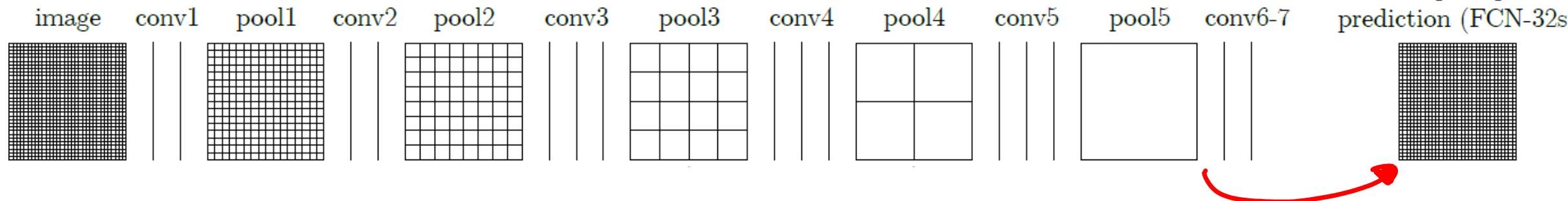
These predictions however are **very coarse**

Upsampling filters are learned with initialization equal to the bilinear interpolation



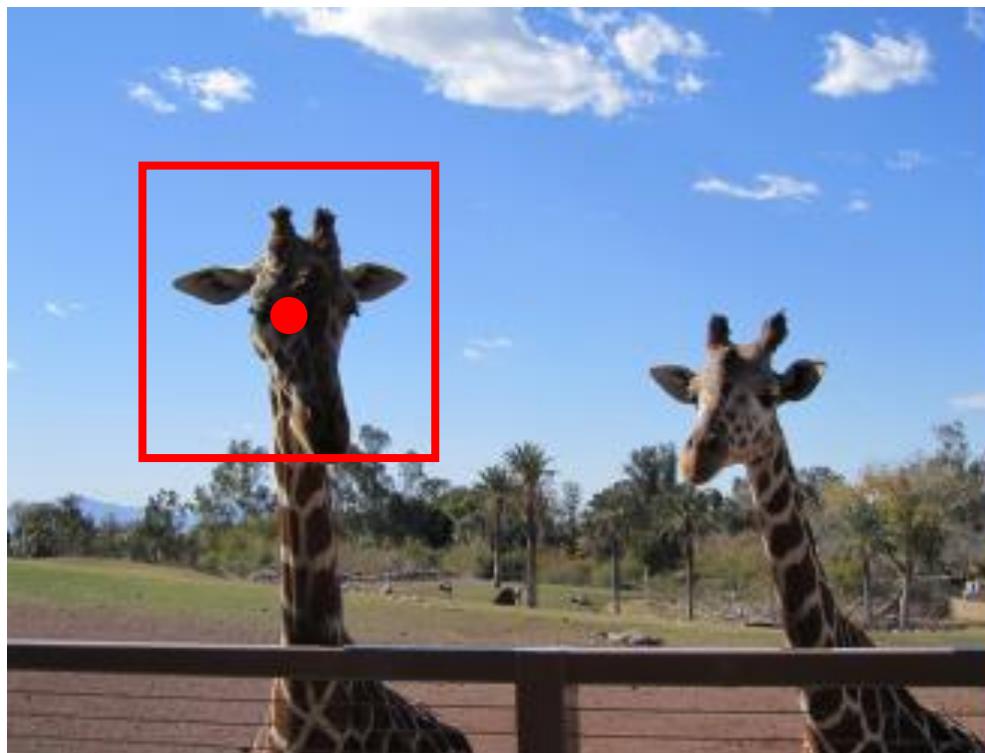
Upsampling filters

32x upsampled
prediction (FCN-32s)



FCNN Training Options: the «patch-based» way

- Prepare a training set for a classification network
- Crop as many patches x_i from annotated images and assign to each patch, the label corresponding to the patch center



FCNN Training Options

The «patch-based» way:

- Prepare a training set for a classification network
- Crop as many patches x_i from annotated images and assign to each patch, **the label corresponding to the patch center**
- **Train a CNN for classification** from scratches, or fine tune a pre-trained model over the segmentation classes
- Convolutionalization: once trained the network, **move the FC layers to 1x1 convolutions**
- **Design the upsampling side of the network** and train these filters

FCNN Training Options

The «patch-based» way:

- The classification network is trained to minimize the classification loss ℓ over a mini-batch

$$\hat{\theta} = \operatorname{argmin}_{\theta} \sum_{x_j} \ell(x_j, \theta)$$

where x_j belongs to a mini-batch

- Batches of patches are **randomly assembled during training**
- **It is possible to resample patches for solving class imbalance**
- **It is very inefficient**, since convolutions on overlapping patches are repeated multiple times

FCNN Training Options: The «full-image» way

Since the network provide dense predictions, it is possible to directly train a FCNN that includes upsampling layers as well

Learning becomes:

$$\hat{\theta} = \operatorname{argmin}_{x_j \in I} \sum_{x_j} \ell(x_j, \theta)$$

Where x_j are all the pixels in a region of the input image and the loss is evaluated over the corresponding labels in the annotation. Therefore, each patch provides already a mini-batch estimate for computing gradient.

Whole image fully convolutional training is identical to patchwise training where each batch consists of all the receptive fields of the units below the loss for an image.

FCNN Training Options

The «full-image» way:

- **FCNN are trained in an end-to-end manner to predict the segmented output $S(\cdot, \cdot)$**
- This loss is the sum of losses over different pixels. Derivatives can be easily computed through the whole network, and this can be trained through backpropagation
- No need to pass through a classification network first
- Takes **advantage of FCNN efficiency**, does not have to re-compute convolutional features in overlapping regions

FCNN Training Options

Limitations of full-image training and solutions:

- Minibatches in patch-wise training are assembled randomly. Image regions in full-image training are not. To make the estimated loss a bit stochastic, adopt random mask

$$\text{minimize} \sum_{x_j} M(x_j) \ell(x_j, \theta)$$

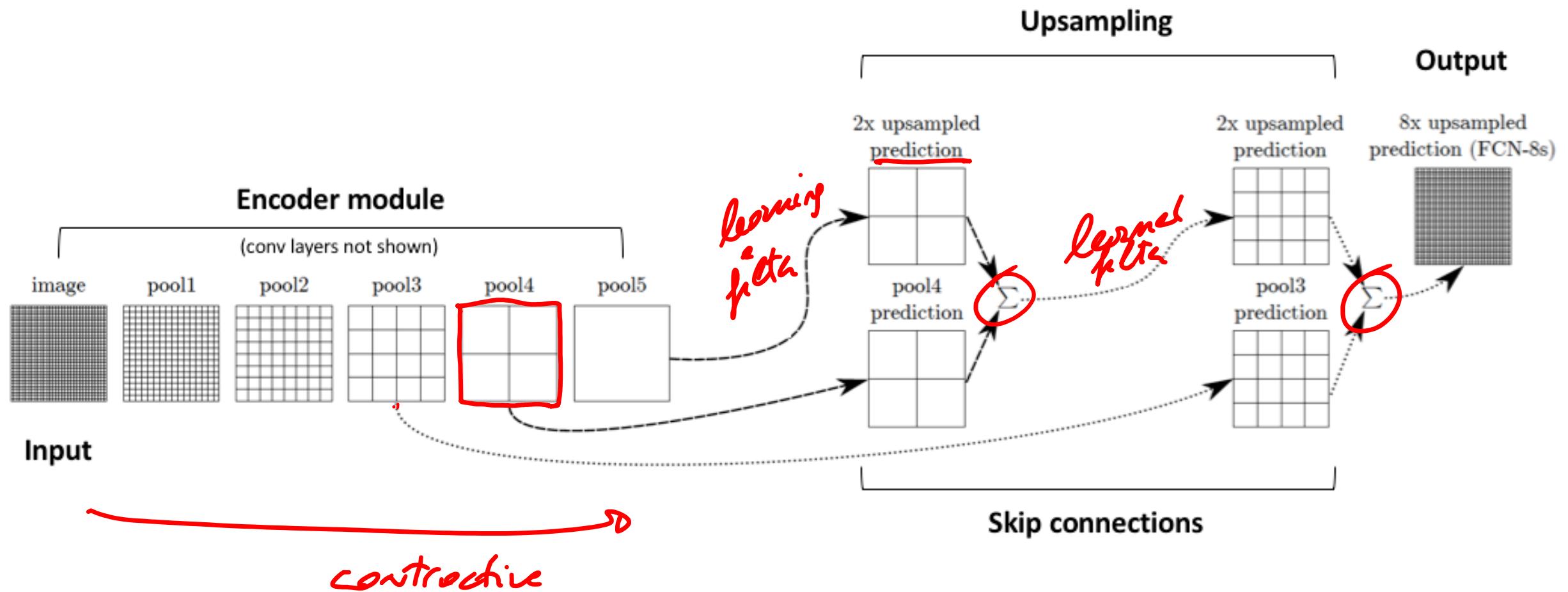
being $M(x_j)$ a binary random variable

- It is not possible to perform patch resampling to compensate for class imbalance. One should go for weighting the loss over different labels

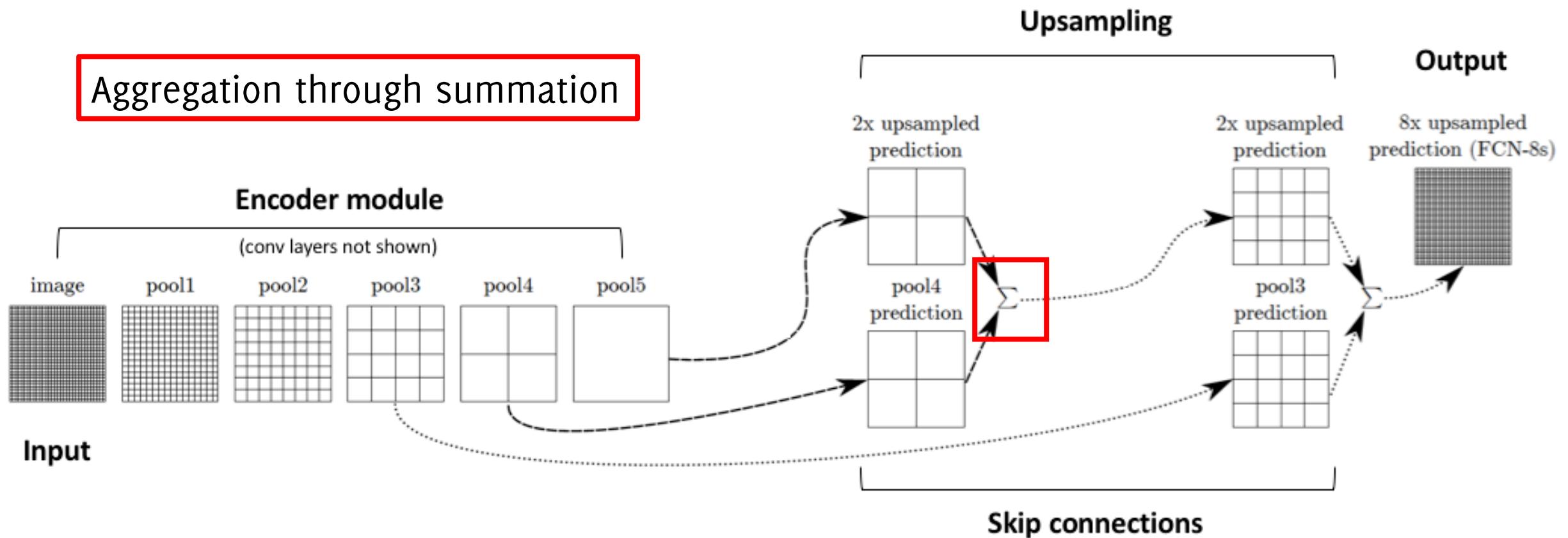
$$\text{minimize} \sum_{x_j} w(x_j) \ell(x_j, \theta)$$

being $w(x_j)$ a weight that takes into account the true label of x_j

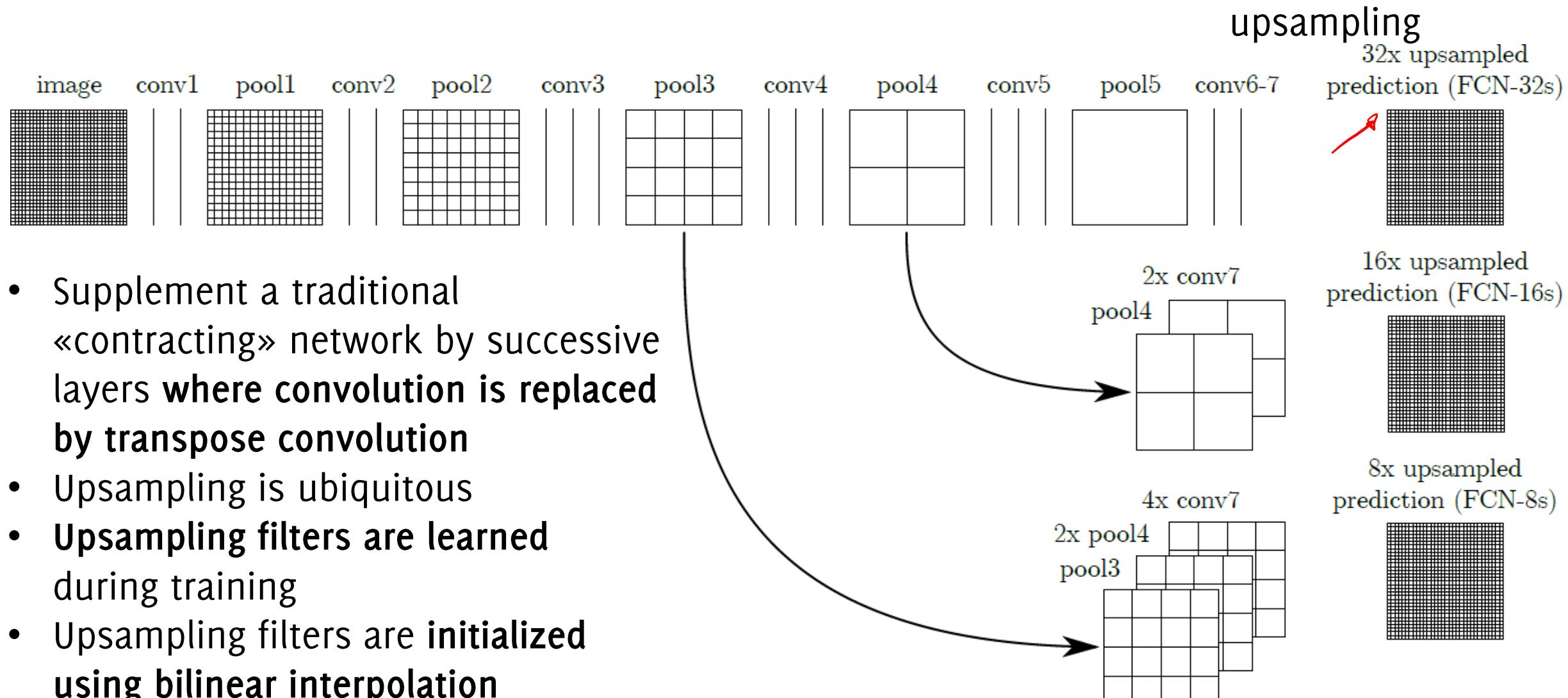
Improve segmentation: Skip Connections!



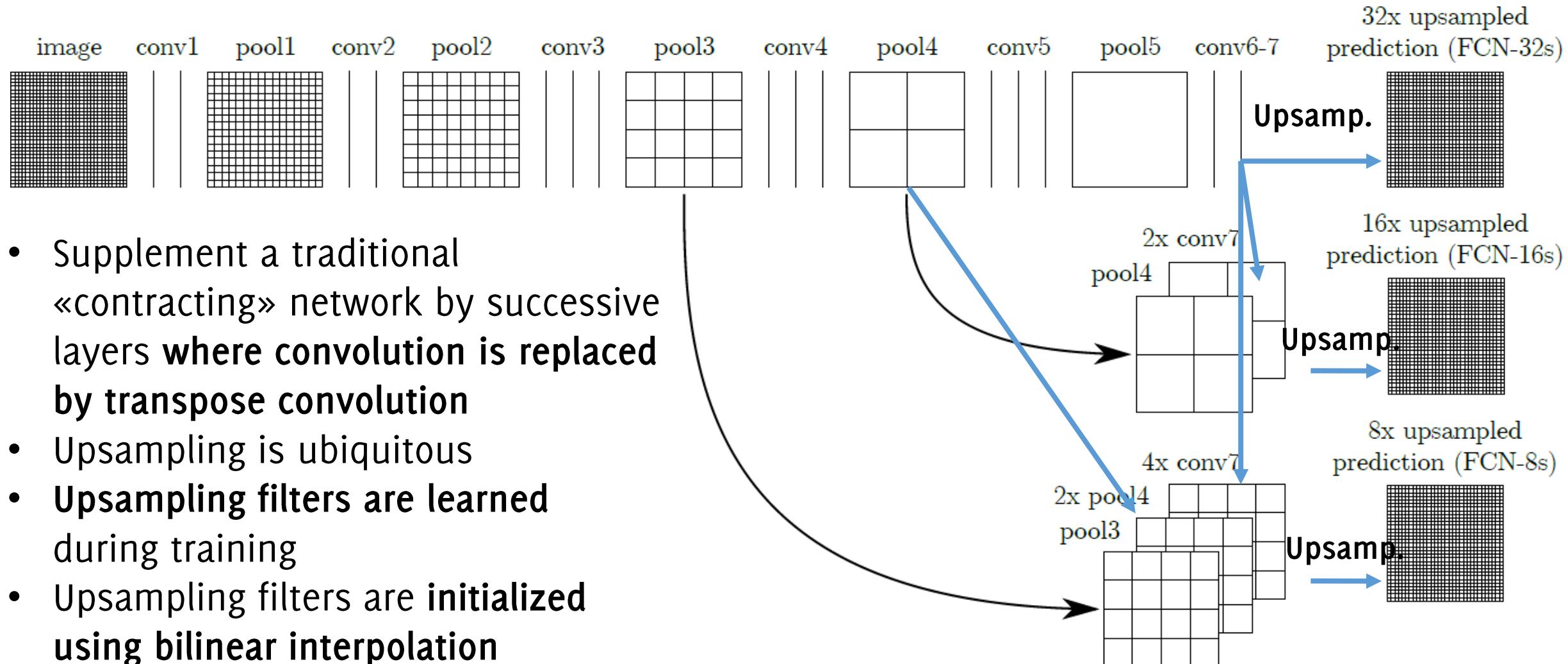
Solution: Skip Connections!



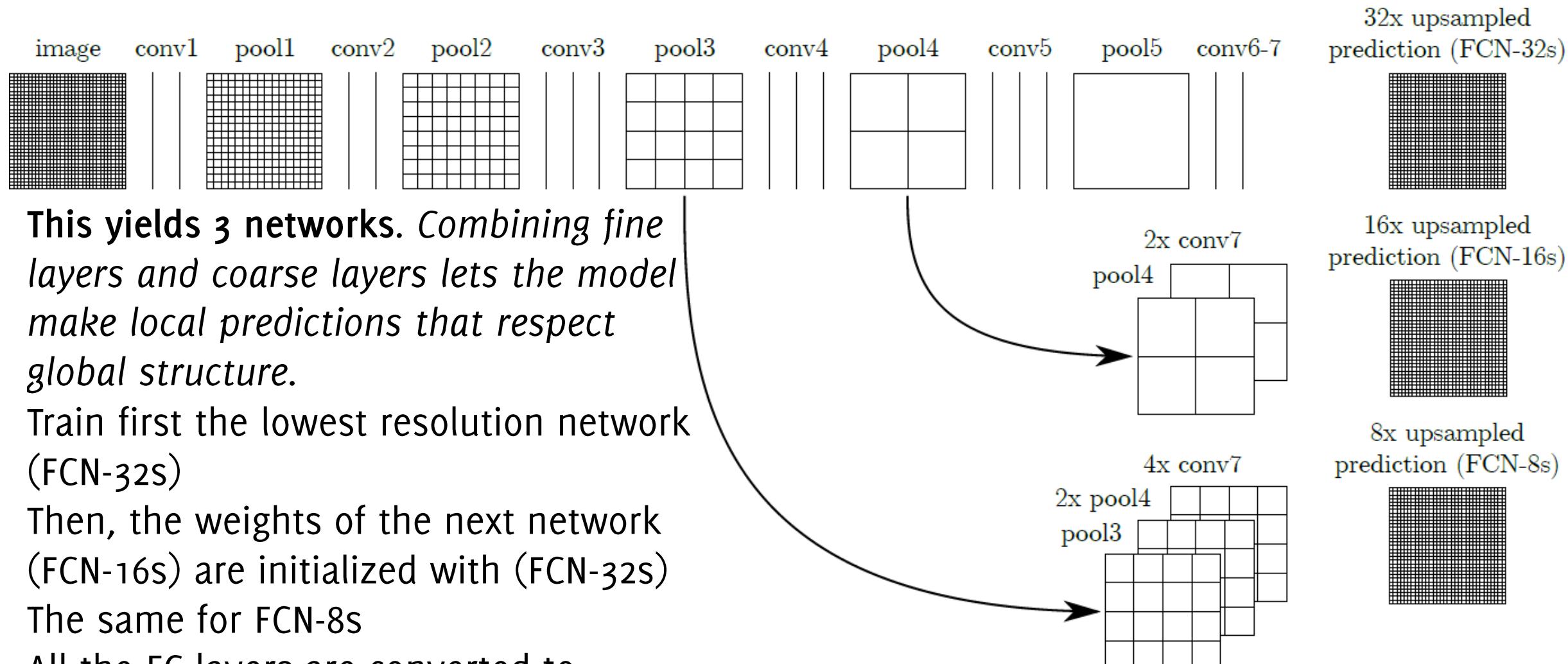
Solution: Skip Connections!



Solution: Skip Connections!

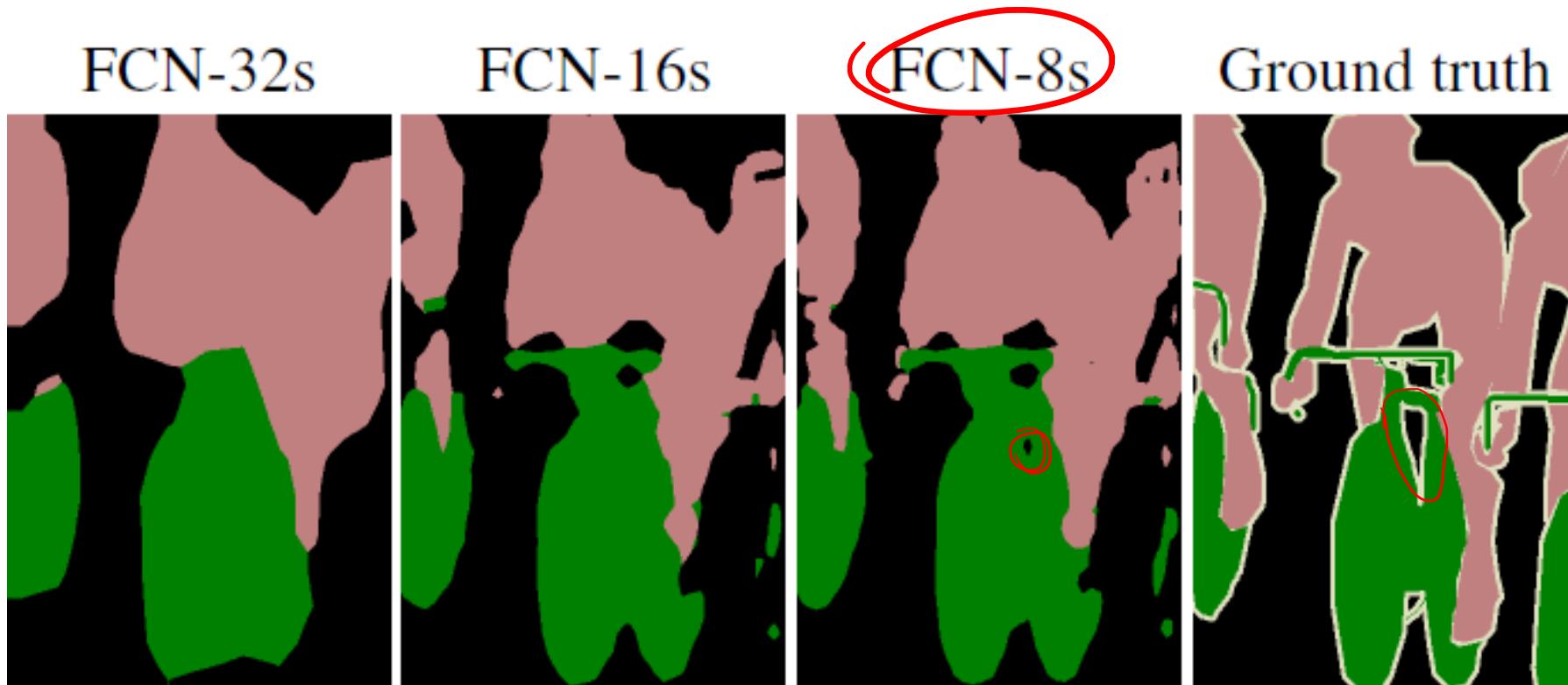


Solution: Skip Connections!

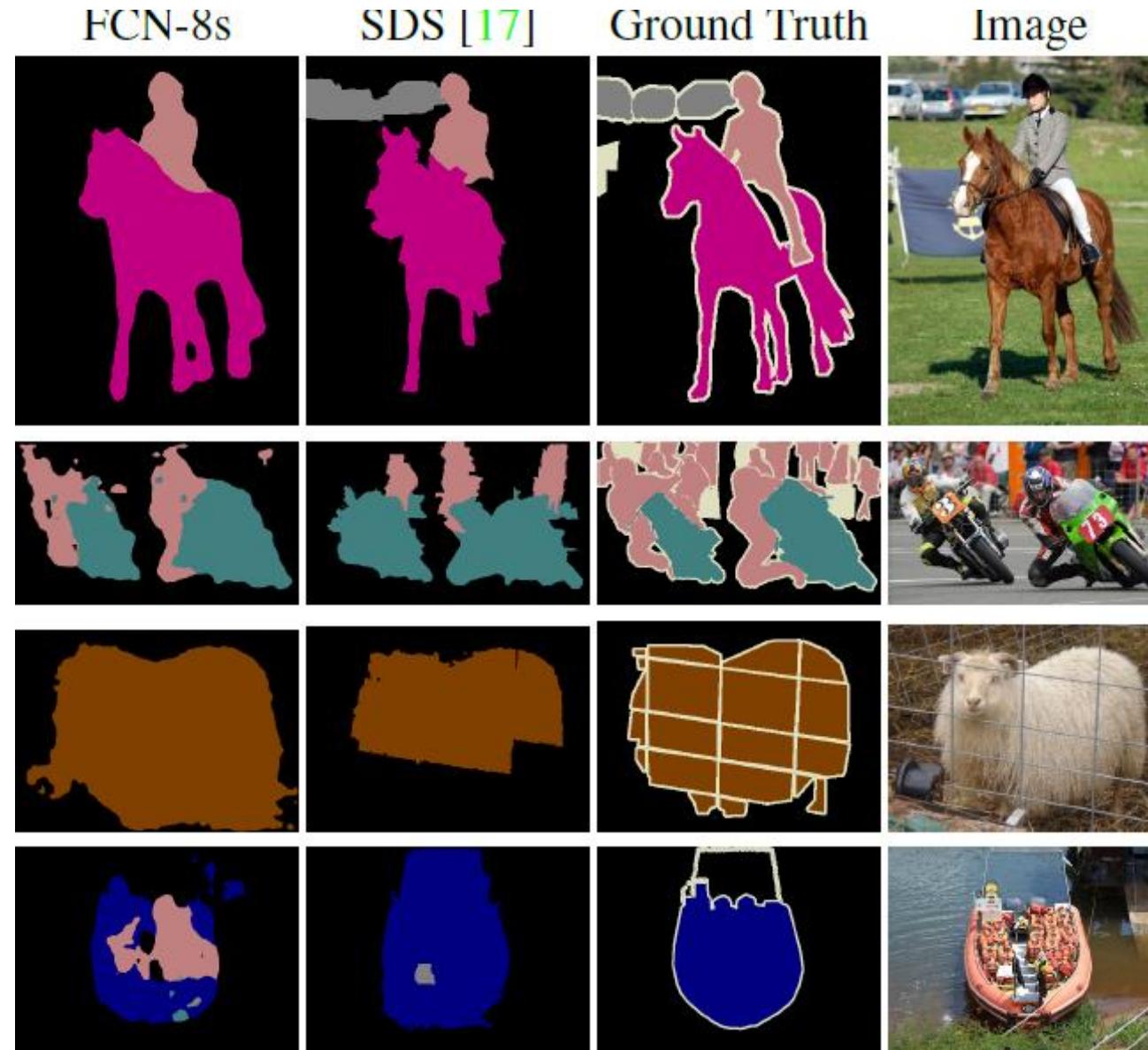


Semantic Segmentation

Retaining intermediate information is beneficial, the deeper layers contribute to provide a better refined estimate of segments



Semantic Segmentation Results



Comments

- Both learning and inference can be performed on the whole-image-at-a-time
- Both in full-image or batch-training it is possible to **perform transfer learning/fine tuning of pre-trained classification models** (segmentation typically requires fewer labels than classification)
- **Accurate pixel-wise prediction** is achieved by upsampling layers
- **End-to-end training is more efficient** than patch-wise training
- Outperforms state-of the art in 2015
- Being fully convolutional, this network **handles arbitrarily sized input**

U-Net: Convolutional Networks for Biomedical Image Segmentation

Olaf Ronneberger, Philipp Fischer, and Thomas Brox

Computer Science Department and BIOSS Centre for Biological Signalling Studies,
University of Freiburg, Germany

ronneber@informatik.uni-freiburg.de,

WWW home page: <http://lmb.informatik.uni-freiburg.de/>

U-Net

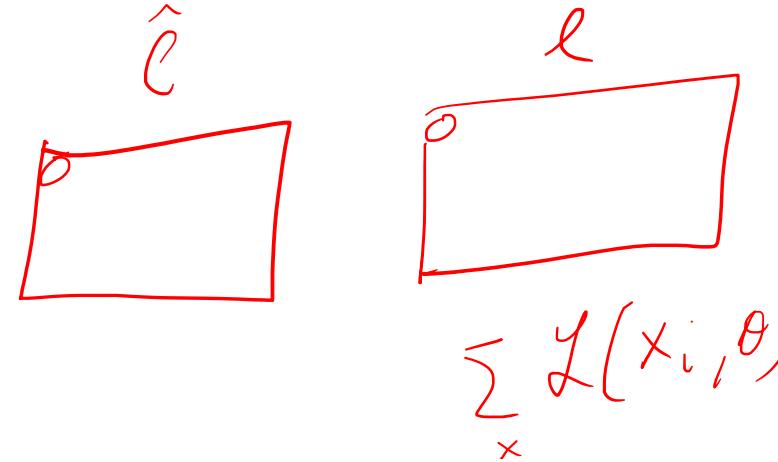
Network formed by:

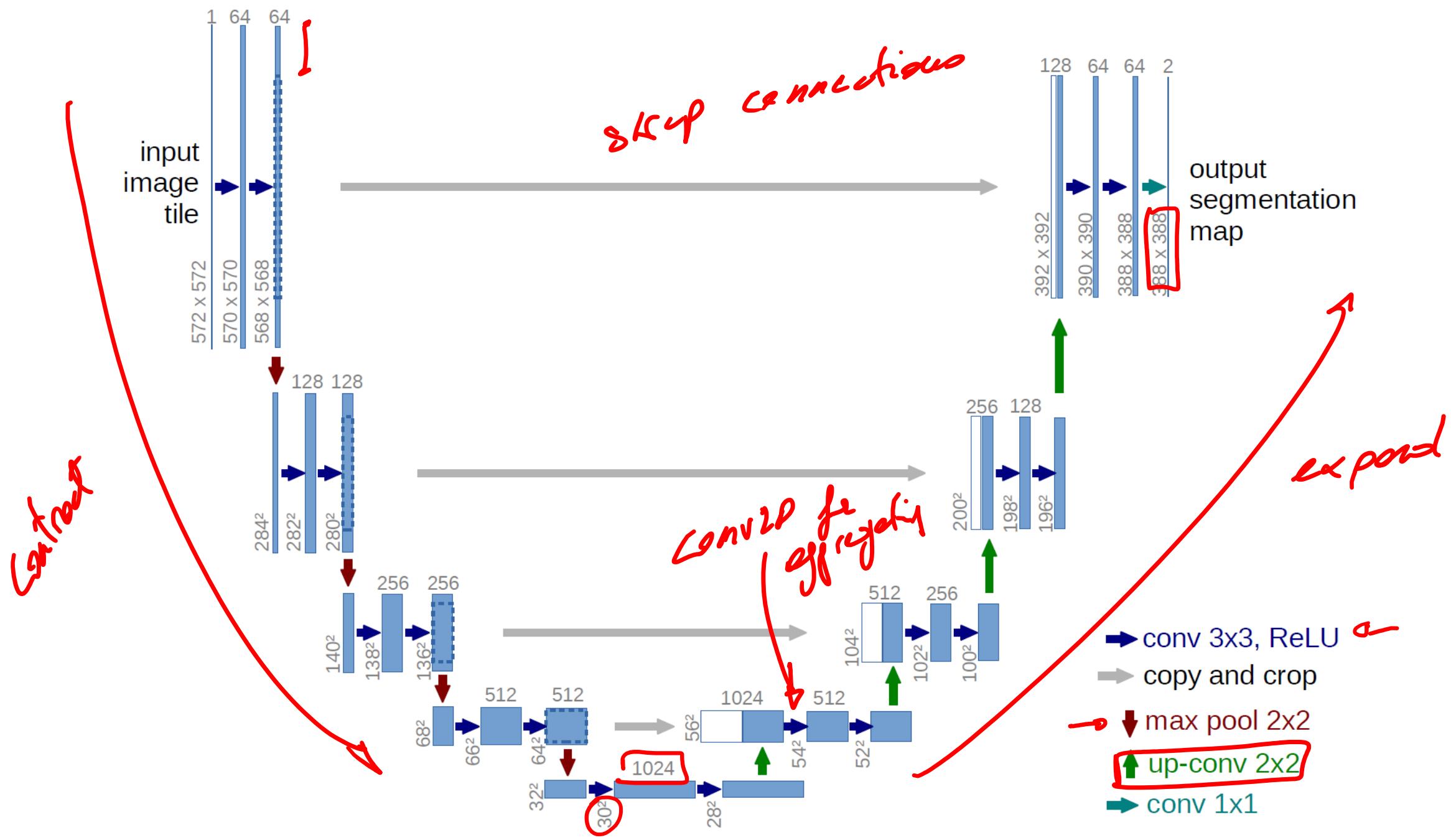
- A contracting path
- An expansive path

No fully connected layers

Major differences w.r.t. (long et al. 2015):

- use a **large number of feature channels** in the upsampling part, while in (long et al. 2015) there were a few upsampling. The network become symmetric
- **Use excessive data-augmentation** by applying elastic deformations to the training images





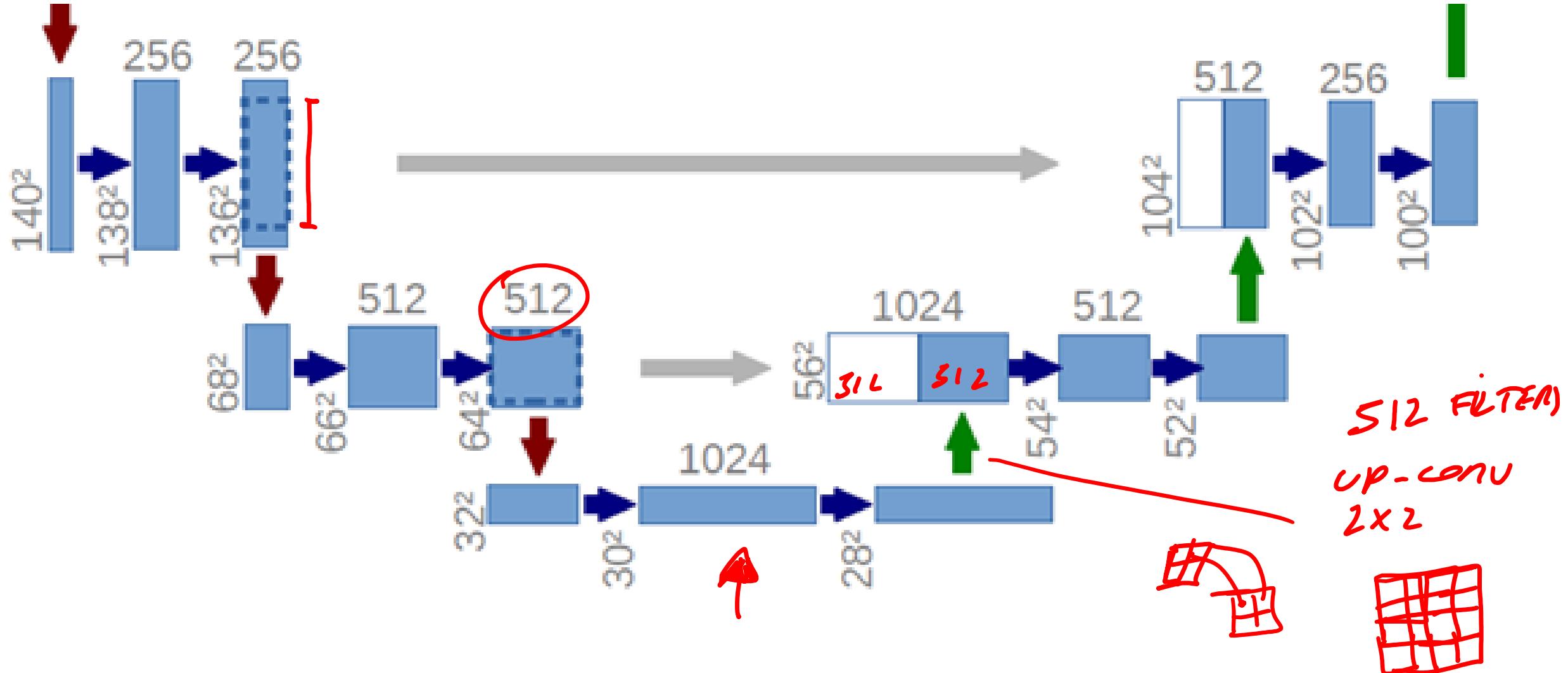
U-Net: Contracting path

Repeats blocks of:

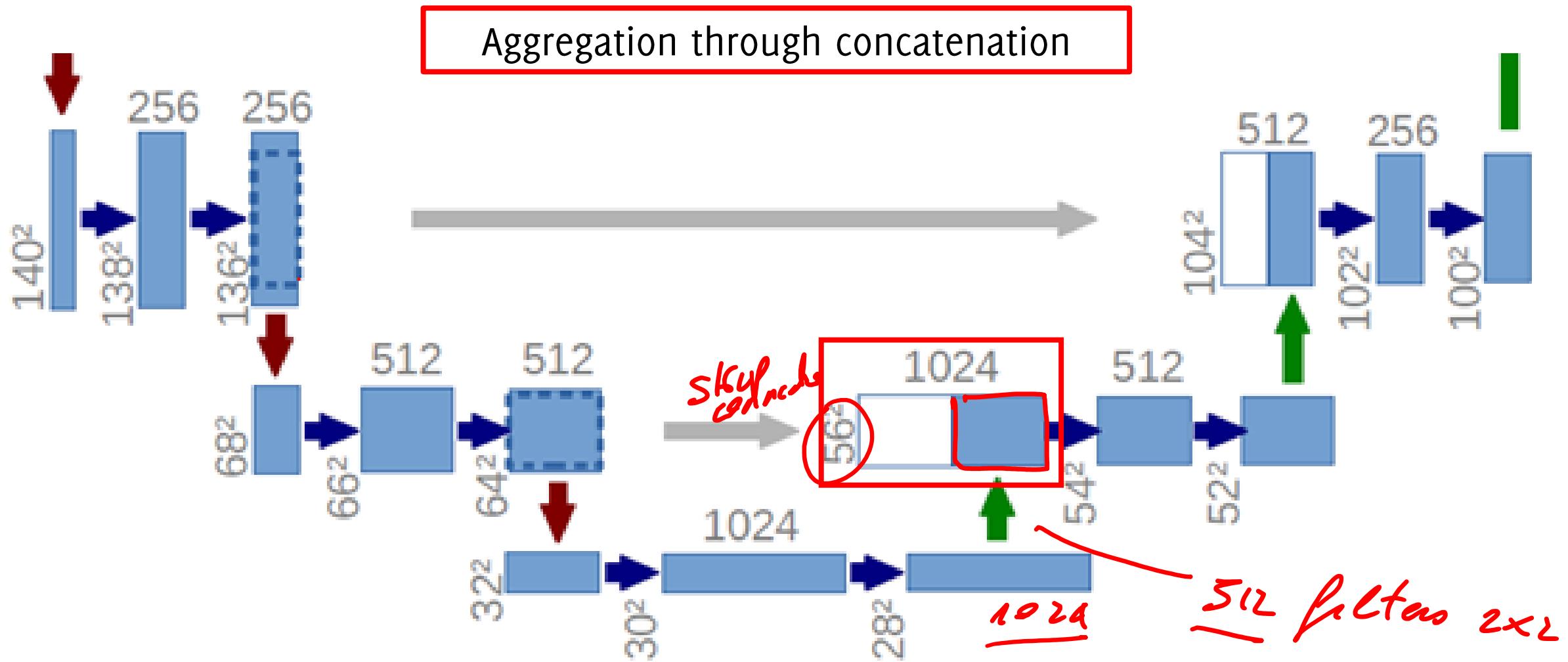
- 3×3 convolution + ReLU ('valid' option, no padding)
- 3×3 convolution + ReLU ('valid' option, no padding)
- Maxpooling 2×2

At each downsampling the number of feature maps is doubled

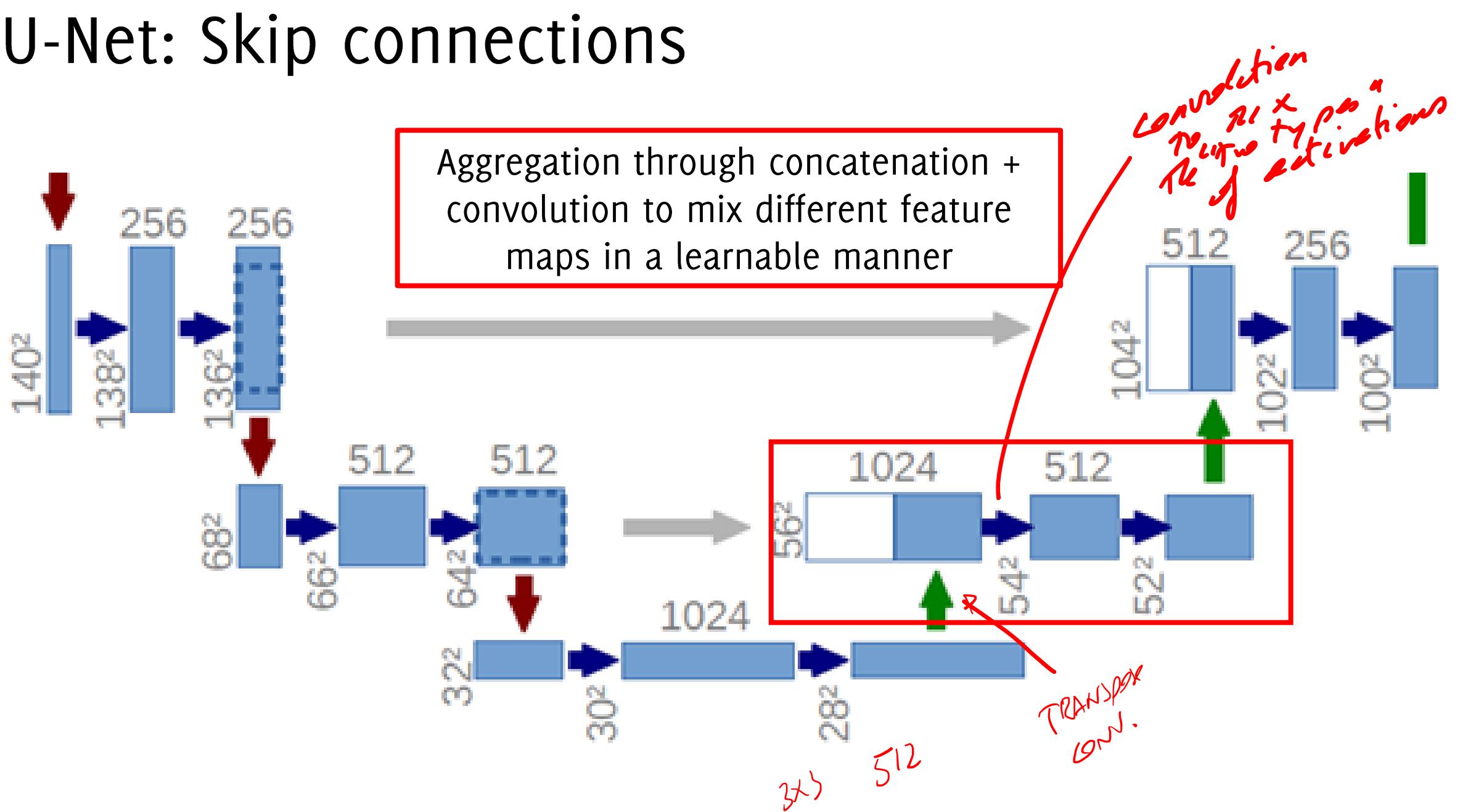
U-Net: Skip connections



U-Net: Skip connections



U-Net: Skip connections



U-Net: Expanding path

Repeats blocks of:

- 2×2 transpose convolution, halving the number of feature maps (but doubling the spatial resolution)
 - Concatenation of corresponding cropped features
 - 3×3 convolution + ReLU
 - 3×3 convolution + ReLU
- 
- Aggregation during
upsampling

U-Net: Network Top

No fully connected layers: there are L convolutions against filters $1 \times 1 \times N$, to yield predictions out of the convolutional feature maps
Output image is smaller than the input image by a constant border

U-Net: Training

Full-image training by a weighted loss function

$$\hat{\theta} = \min_{\theta} \sum_{x_j} w(x_j) \ell(x_j, \theta)$$

where the weight

$$w(x) = w_c(x) + w_0 e^{-\frac{(d_1(x)+d_2(x))^2}{2\sigma^2}}$$

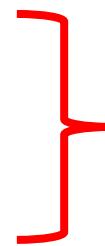
- w_c is used to balance class proportions (remember no patch resampling in full-image training)
- d_1 is the distance to the border of the closest cell
- d_2 is the distance to the border of the second closest cell

U-net: Training

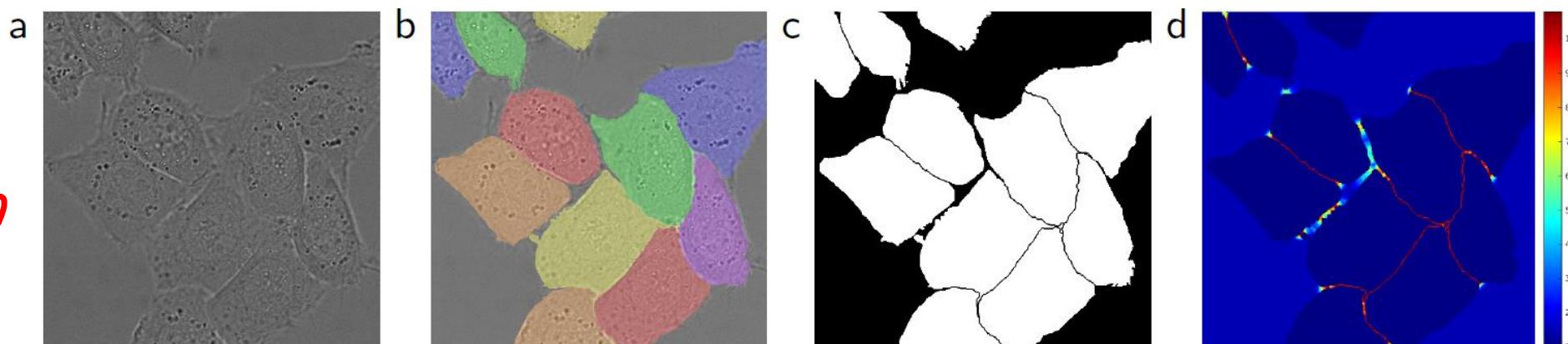
Full-image training by a weighted loss function

$$w(x) = w_c(x) + w_0 e^{-\frac{(d_1(x)+d_2(x))^2}{2\sigma^2}}$$

- w_c is used to balance class proportions (remember no patch resampling in full-image training)
- d_1 is the distance to the border of the closest cell
- d_2 is the distance to the border of the second closest cell



Weights are large when the distance to the first two closest cells is small



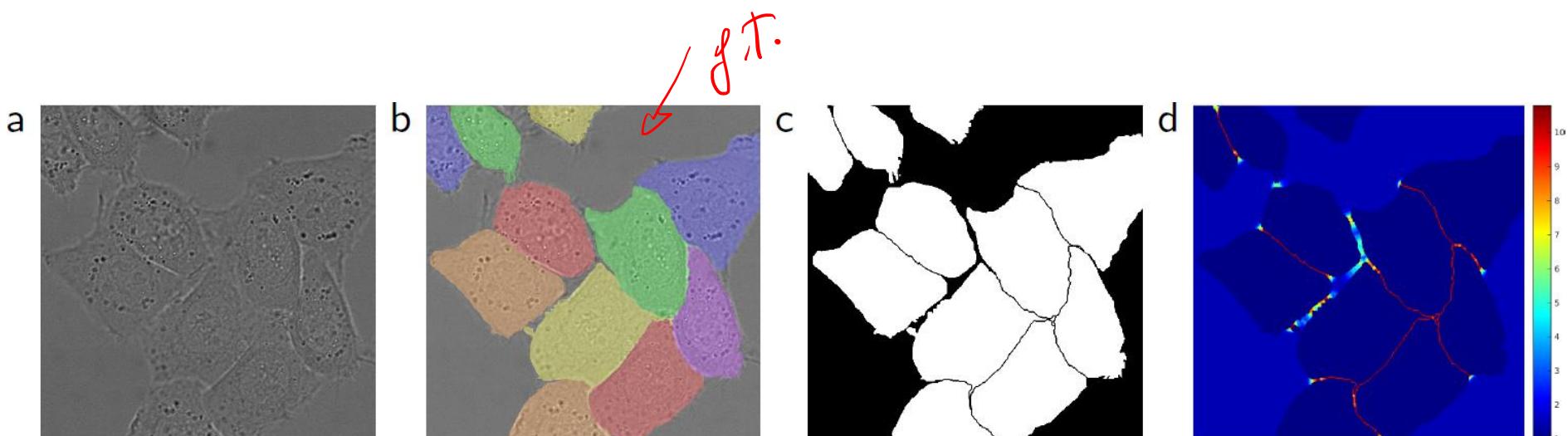
U-net: Training

Full-image training by a weighted loss function

$$w(x) = w_c(x) + w_0 e^{-\frac{(d_1(x)+d_2(x))^2}{2\sigma^2}}$$

Takes into account class unbalance in the training set

Enhances classification performance at borders of different objects



U-net: Training

This term is large at pixels close to borders delimiting objects of different cells

Full-image training by a weighted loss function

$$w(x) = w_c(x) + w_0 e^{-\frac{(d_1(x)+d_2(x))^2}{2\sigma^2}}$$

