Deep learning

8.1. Computer vision tasks

François Fleuret

https://fleuret.org/dlc/

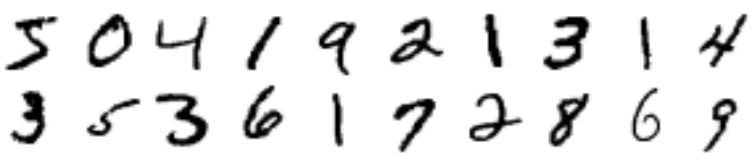
UNIVERSITÉ
DE GENÈVE

Computer vision tasks:

- classification,
- object detection,
- semantic or instance segmentation,
- other (tracking in videos, camera pose estimation, body pose estimation, 3d reconstruction, denoising, super-resolution, auto-captioning, synthesis, etc.)

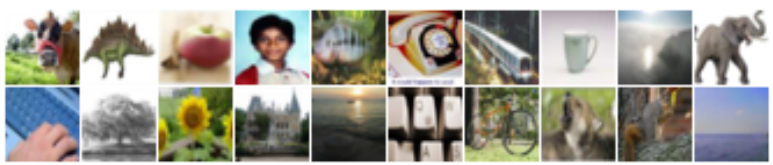
"Small scale" classification data-sets.

MNIST and Fashion-MNIST: 10 classes (digits or pieces of clothing) $50,000$ train images, $10,000$ test images, 28×28 grayscale.



(LeCun et al., 1998; Xiao et al., 2017)

CIFAR10 and CIFAR100 (10 classes and 5×20 "super classes"), $50,000$ train images, $10,000$ test images, 32×32 RGB



(Krizhevsky, 2009, chap. 3)

ImageNet

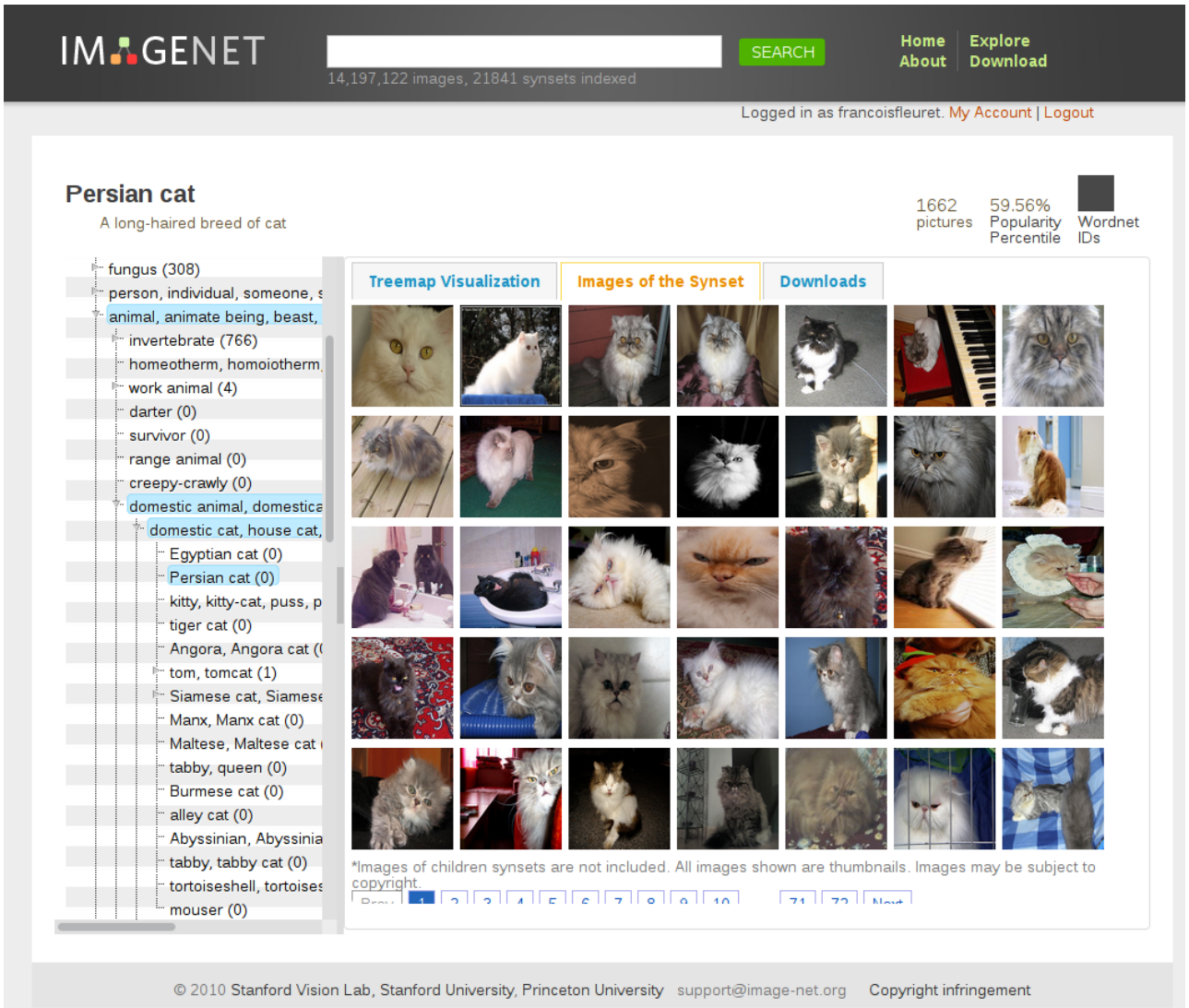
http://www.image-net.org/

This data-set is build by filling the leaves of the "Wordnet" hierarchy, called "synsets" for "sets of synonyms".

- $21,841$ non-empty synsets,
- $14,197,122$ images,
- $1,034,908$ images with bounding box annotations.

ImageNet Large Scale Visual Recognition Challenge 2012

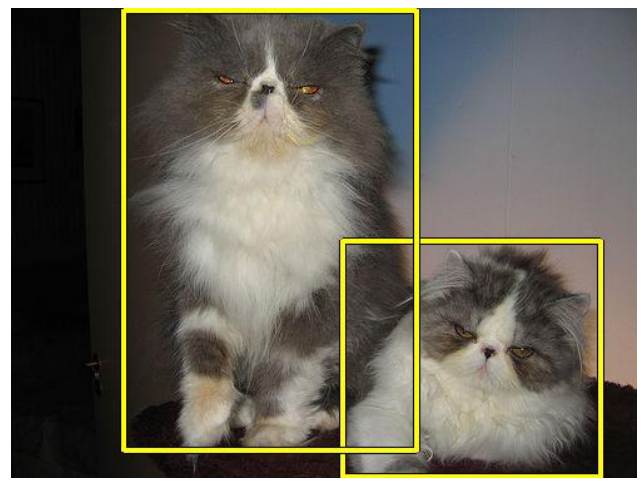
- $1,000$ classes taken among all synsets,
- $1,200,000$ training, and $50,000$ validation images.


IMAGENET
SEARCH
14,197,122 images, 21841 synsets indexed
Home
About
Explore
Download
Logged in as francoisfleuret. My Account | Logout
Persian cat
A long-haired breed of cat
1662 pictures
59.56% Popularity Percentile
Wordnet IDs
fungus (308)
person, individual, someone, s
animal, animate being, beast,
invertebrate (766)
homeotherm, homoiotherm,
work animal (4)
darter (0)
survivor (0)
range animal (0)
creepy-crawly (0)
domestic animal, domestica
domestic cat, house cat,
Egyptian cat (0)
Persian cat (0)
kitty, kitty-cat, puss, p
tiger cat (0)
Angora, Angora cat (
tom, tomcat (1)
Siamese cat, Siamese
Manx, Manx cat (0)
Maltese, Maltese cat
tabby, queen (0)
Burmese cat (0)
alley cat (0)
Abyssinian, Abyssinia
tabby, tabby cat (0)
tortoiseshell, tortoises
mouser (0)
Treemap Visualization
Images of the Synset
Downloads
*Images of children synsets are not included. All images shown are thumbnails. Images may be subject to copyright.
© 2010 Stanford Vision Lab, Stanford University, Princeton University
support@image-net.org
Copyright infringement

n02123394_2084.xml

```
<annotation>
        <folder>n02123394</folder>
        <filename>n02123394_2084</filename>
        <source>
                <database>ImageNet database</database>
        </source>
        <size>
                <width>500</width>
                <height>375</height>
                <depth>3</depth>
        </size>
        <segmented>0</segmented>
        <object>
                <name>n02123394</name>
                <pose>Unspecified</pose>
                <truncated>0</truncated>
                <difficult>0</difficult>
                <bndbox>
                        <xmin>265</xmin>
                        <ymin>185</ymin>
                        <xmax>470</xmax>
                        <ymax>374</ymax>
                </bndbox>
        </object>
        <object>
                <name>n02123394</name>
                <pose>Unspecified</pose>
                <truncated>0</truncated>
                <difficult>0</difficult>
                <bndbox>
                        <xmin>90</xmin>
                        <ymin>1</ymin>
                        <xmax>323</xmax>
                        <ymax>353</ymax>
                </bndbox>
        </object>
</annotation>
```

n02123394_2084.JPEG



Cityscapes data-set

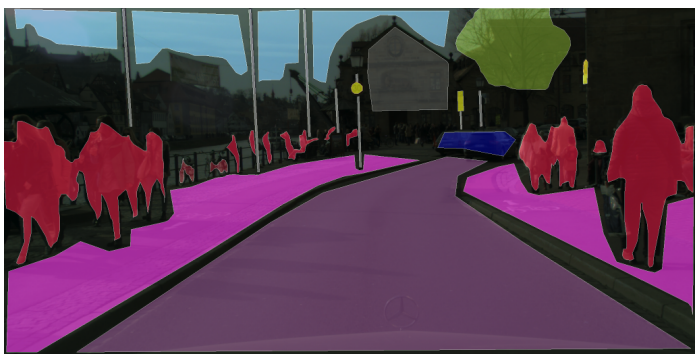
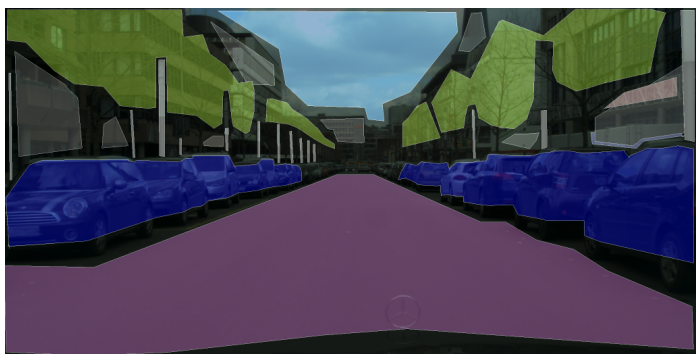
https://www.cityscapes-dataset.com/

Images from 50 cities over several months, each is the 20th image from a 30 frame video snippets (1.8s). Meta-data about vehicle position + depth.

- 30 classes
  - flat: road, sidewalk, parking, rail track
  - human: person, rider
  - vehicle: car, truck, bus, on rails, motorcycle, bicycle, caravan, trailer
  - construction: building, wall, fence, guard rail, bridge, tunnel
  - object: pole, pole group, traffic sign, traffic light
  - nature: vegetation, terrain
  - sky: sky
  - void: ground, dynamic, static
- $5,000$ images with fine annotations
- $20,000$ images with coarse annotations.

Cityscapes fine annotations ($5,000$ images)


Cityscapes coarse annotations ($20,000$ images)

Performance measures

Image classification consists of predicting the input image's class, which is often the class of the "main object" visible in it.

The standard performance measures are:

- The **error rate** $\hat{P}(f(X) \neq Y)$ or conversely the **accuracy** $\hat{P}(f(X) = Y)$,
- the **balanced error rate** (BER) $\frac{1}{C} \sum_{y=1}^{C} \hat{P}(f(X) \neq Y \mid Y = y)$.

In the two-class case, we can define the True Positive (TP) rate as $\hat{P}(f(X) = 1 \mid Y = 1)$ and the False Positive (FP) rate as $\hat{P}(f(X) = 1 \mid Y = 0)$.

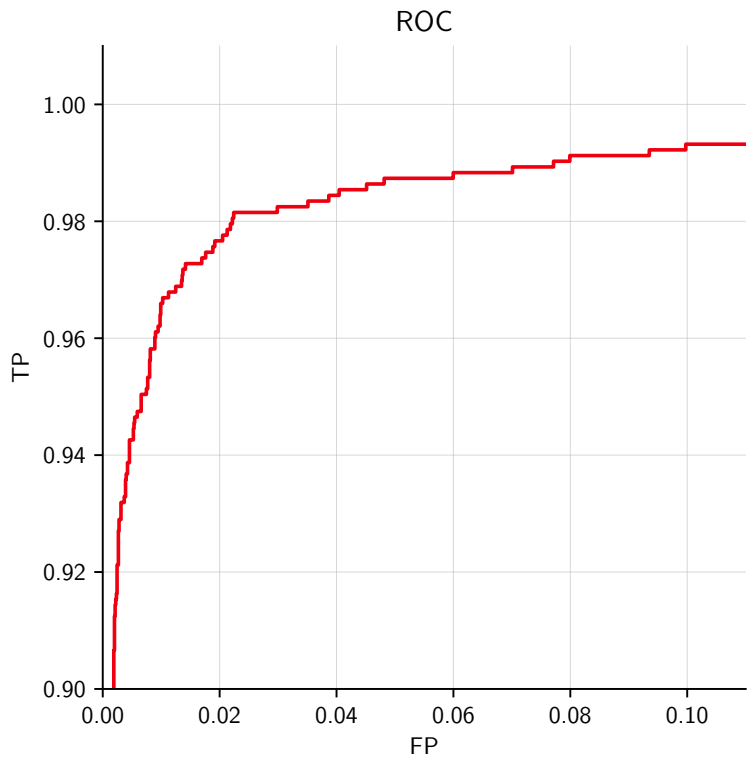
The ideal algorithm would have TP $\simeq 1$ and FP $\simeq 0$.

Most of the algorithms produce a continuous score (e.g. difference of logits), and make a hard decision with a threshold that is application-dependent. E.g.

- **Cancer detection:** Low threshold to get a high TP rate (you do not want to miss a cancer), at the cost of a high FP rate (it will be double-checked by an oncologist anyway),
- **Image retrieval:** High threshold to get a low FP rate (you do not want to bring an image that does not match the request), at the cost of a low TP rate (you have so many images that missing a lot is not an issue).

In that case, a standard performance representation is the **Receiver operating characteristic** (ROC) that shows performance at multiple thresholds.

It is the minimum increasing function above the True Positive (TP) rate $\hat{P}(f(X) = 1 \mid Y = 1)$ vs. the False Positive (FP) rate $\hat{P}(f(X) = 1 \mid Y = 0)$.



A standard measure is the **area under the curve** (AUC).

Object detection aims at predicting **classes and locations** of targets in an image. The notion of "location" is ill-defined. In the standard setup, the output of the predictor is a series of bounding boxes, each with a class label.

A standard performance assessment considers that a predicted bounding box $\hat{B}$ is correct if there is an annotated bounding box B for that class, such that the **Intersection over Union** (IoU) is large enough

$$\frac{area(B \cap \hat{B})}{area(B \cup \hat{B})} \geq \frac{1}{2}.$$

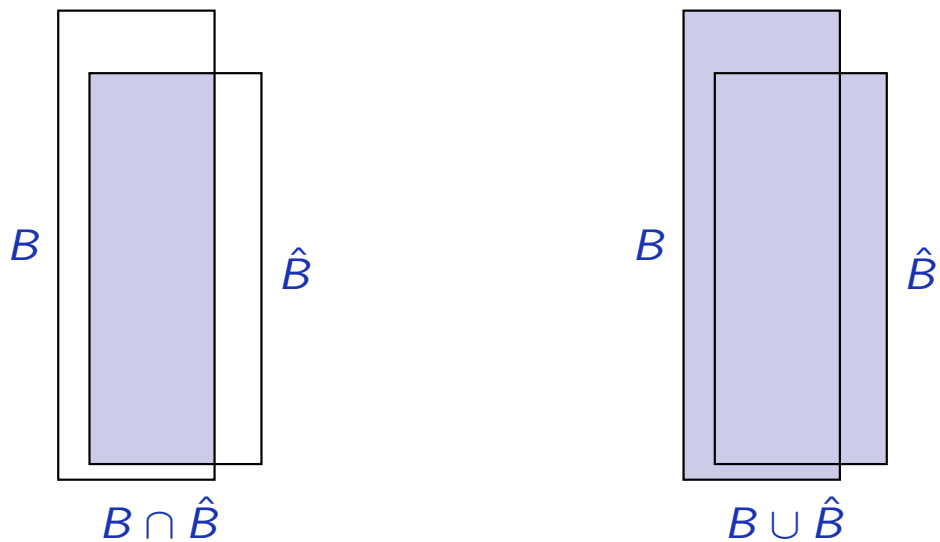


Image segmentation consists of labeling individual pixels with the class of the object it belongs to, and may also involve predicting the instance it belongs to.

The standard performance measure frames the task as a classification one. For VOC2012, the **segmentation accuracy** (SA) for a class c is defined as

$$SA = \frac{N_{Y=c,\hat{Y}=c}}{N_{Y=c,\hat{Y}=c} + N_{Y\neq c,\hat{Y}=c} + N_{Y=c,\hat{Y}\neq c}},$$

where N_α is the number of pixel with the property α, Y the real class of a pixel, and $\hat{Y}$ the predicted one.

All these performance measures are debatable, and in practice they are highly application-dependent.

In spite of their weaknesses, the ones adopted as standards by the community enable an assessment of the field's "long-term progress".

Deep learning

8.2. Networks for image classification

François Fleuret

https://fleuret.org/dlc/

UNIVERSITÉ
DE GENÈVE

Standard convnets

The standard model for image classification are the LeNet family (LeCun et al., 1989, 1998), and its modern variants such as AlexNet (Krizhevsky et al., 2012) and VGGNet (Simonyan and Zisserman, 2014).

They share a common structure of several convolutional layers seen as a feature extractor, followed by fully connected layers seen as a classifier.

The performance of AlexNet was a wake-up call for the computer vision community, as it vastly out-performed other methods in spite of its simplicity.

Recent advances rely on moving from standard convolutional layers to more complex local architectures to reduce the model size.

`torchvision.models` provides a collection of reference networks for computer vision, e.g.:

```
import torchvision
alexnet = torchvision.models.alexnet()
```

The trained models can be obtained by passing `pretrained = True` to the constructor(s). This may involve an heavy download given there size.

The networks from PyTorch listed in the coming slides may differ slightly from the reference papers which introduced them historically.

LeNet5 (LeCun et al., 1989). 10 classes, input $1 \times 28 \times 28$.

```
(features): Sequential (
  (0): Conv2d(1, 6, kernel_size=(5, 5), stride=(1, 1))
  (1): ReLU (inplace)
  (2): MaxPool2d (size=(2, 2), stride=(2, 2), dilation=(1, 1))
  (3): Conv2d(6, 16, kernel_size=(5, 5), stride=(1, 1))
  (4): ReLU (inplace)
  (5): MaxPool2d (size=(2, 2), stride=(2, 2), dilation=(1, 1))
)

(classifier): Sequential (
  (0): Linear (256 -> 120)
  (1): ReLU (inplace)
  (2): Linear (120 -> 84)
  (3): ReLU (inplace)
  (4): Linear (84 -> 10)
)
```

Alexnet (Krizhevsky et al., 2012). $1,000$ classes, input $3 \times 224 \times 224$.

```
(features): Sequential (
  (0): Conv2d(3, 64, kernel_size=(11, 11), stride=(4, 4), padding=(2, 2))
  (1): ReLU (inplace)
  (2): MaxPool2d (size=(3, 3), stride=(2, 2), dilation=(1, 1))
  (3): Conv2d(64, 192, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2))
  (4): ReLU (inplace)
  (5): MaxPool2d (size=(3, 3), stride=(2, 2), dilation=(1, 1))
  (6): Conv2d(192, 384, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (7): ReLU (inplace)
  (8): Conv2d(384, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (9): ReLU (inplace)
  (10): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (11): ReLU (inplace)
  (12): MaxPool2d (size=(3, 3), stride=(2, 2), dilation=(1, 1))
)

(classifier): Sequential (
  (0): Dropout (p = 0.5)
  (1): Linear (9216 -> 4096)
  (2): ReLU (inplace)
  (3): Dropout (p = 0.5)
  (4): Linear (4096 -> 4096)
  (5): ReLU (inplace)
  (6): Linear (4096 -> 1000)
)
```

Krizhevsky et al. used **data augmentation** during training to reduce over-fitting.

They generated $2,048$ samples from every original training example through two classes of transformations:

- crop a 224×224 image at a random position in the original 256×256, and randomly reflect it horizontally,
- apply a color transformation using a PCA model of the color distribution.

During test the prediction is averaged over five random crops and their horizontal reflections.

VGGNet19 (Simonyan and Zisserman, 2014). $1,000$ classes, input $3 \times 224 \times 224$. 16 convolutional layers $+$ 3 fully connected layers.

```
  (features): Sequential (
    (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): ReLU (inplace)
    (2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (3): ReLU (inplace)
    (4): MaxPool2d (size=(2, 2), stride=(2, 2), dilation=(1, 1))
    (5): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (6): ReLU (inplace)
    (7): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (8): ReLU (inplace)
    (9): MaxPool2d (size=(2, 2), stride=(2, 2), dilation=(1, 1))
    (10): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (11): ReLU (inplace)
    (12): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (13): ReLU (inplace)
    (14): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (15): ReLU (inplace)
    (16): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (17): ReLU (inplace)
    (18): MaxPool2d (size=(2, 2), stride=(2, 2), dilation=(1, 1))
    (19): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (20): ReLU (inplace)
    (21): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (22): ReLU (inplace)
    (23): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (24): ReLU (inplace)
    (25): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (26): ReLU (inplace)
    (27): MaxPool2d (size=(2, 2), stride=(2, 2), dilation=(1, 1))
/.../
```

VGGNet19 (cont.)

```
  (classifier): Sequential (
    (0): Linear (25088 -> 4096)
    (1): ReLU (inplace)
    (2): Dropout (p = 0.5)
    (3): Linear (4096 -> 4096)
    (4): ReLU (inplace)
    (5): Dropout (p = 0.5)
    (6): Linear (4096 -> 1000)
  )
```

We can illustrate the convenience of these pre-trained models on a simple image-classification problem.

To be sure this picture did not appear in the training data, it was not taken from the web.

```
import PIL, torch, torchvision

# Load and normalize the image
to_tensor = torchvision.transforms.ToTensor()
img = to_tensor(PIL.Image.open('../example_images/blacklab.jpg'))
img = img.unsqueeze(0)
img = 0.5 + 0.5 * (img - img.mean()) / img.std()

# Load and evaluate the network
alexnet = torchvision.models.alexnet(pretrained = True)
alexnet.eval()

output = alexnet(img)

# Prints the classes
scores, indexes = output.view(-1).sort(descending = True)

class_names = eval(open('imagenet1000_clsid_to_human.txt', 'r').read())

for k in range(12):
    print(f'#{k+1} {scores[k].item():.02f} {class_names[indexes[k].item()]}')
```

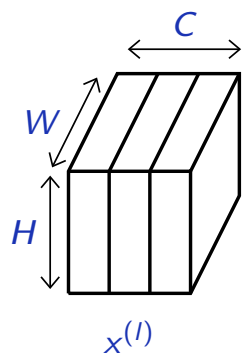
```
12.26 Weimaraner
10.95 Chesapeake Bay retriever
10.87 Labrador retriever
10.10 Staffordshire bullterrier, Staffordshire bull terrier
9.55 flat-coated retriever
9.40 Italian greyhound
9.31 American Staffordshire terrier, Staffordshire terrier, American pit bull terrier, pit bull terrier
9.12 Great Dane
8.94 German short-haired pointer
8.53 Doberman, Doberman pinscher
8.35 Rottweiler
8.25 kelpie
8.24 barrow, garden cart, lawn cart, wheelbarrow
8.12 bucket, pail
8.07 soccer ball
```

Weimaraner

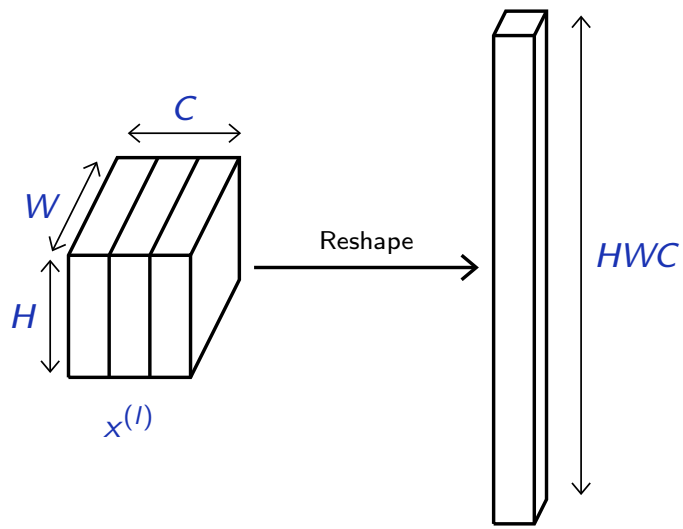
Chesapeake Bay retriever

Fully convolutional networks

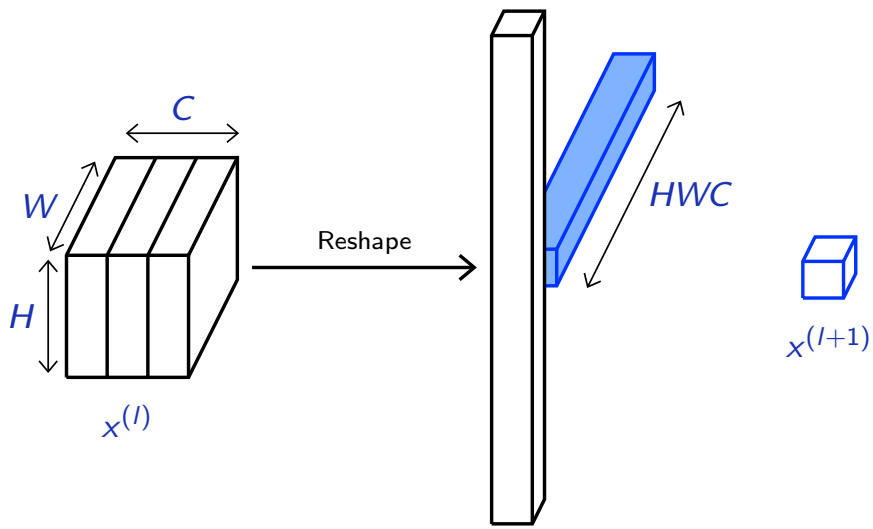
In many applications, standard convolutional networks are made **fully convolutional** by converting their fully connected layers to convolutional ones.



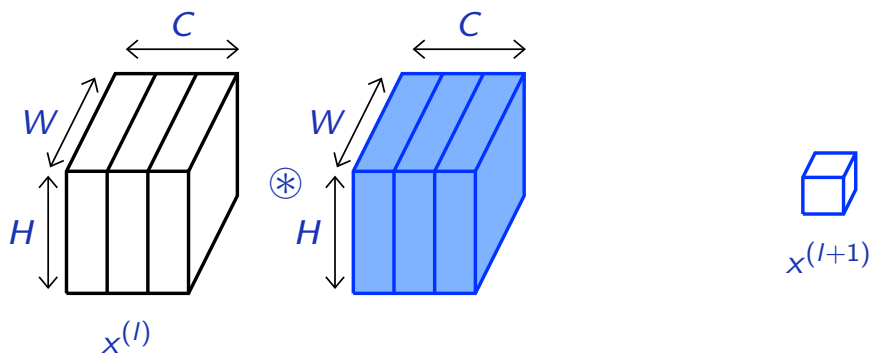
In many applications, standard convolutional networks are made **fully convolutional** by converting their fully connected layers to convolutional ones.



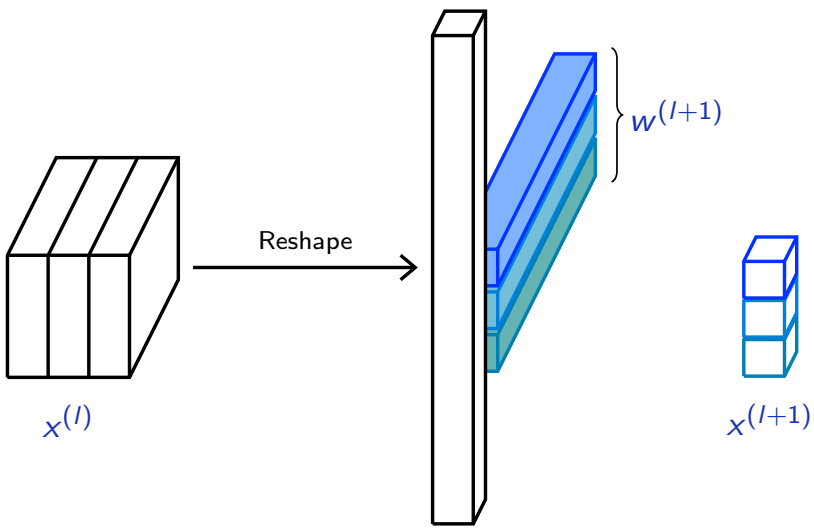
In many applications, standard convolutional networks are made **fully convolutional** by converting their fully connected layers to convolutional ones.



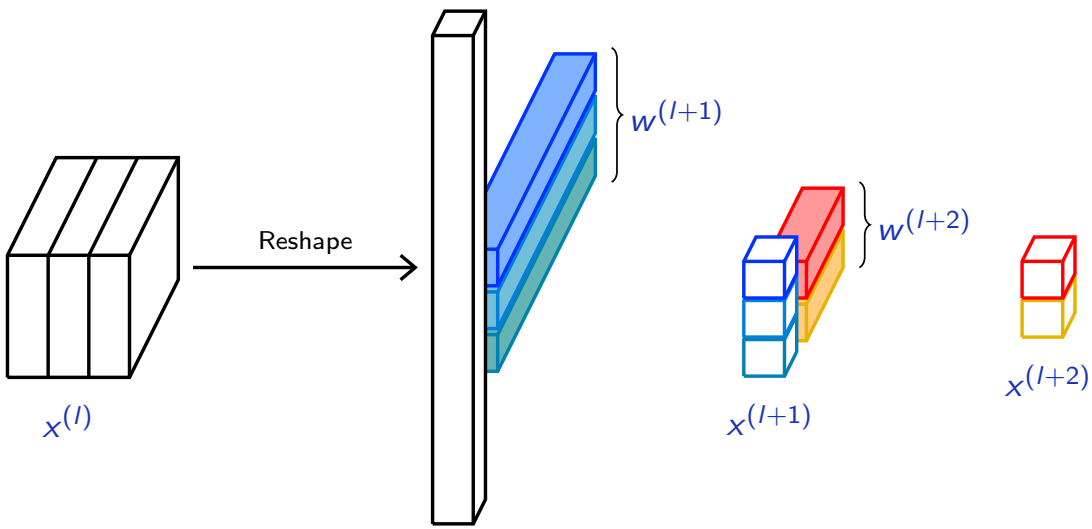
In many applications, standard convolutional networks are made **fully convolutional** by converting their fully connected layers to convolutional ones.



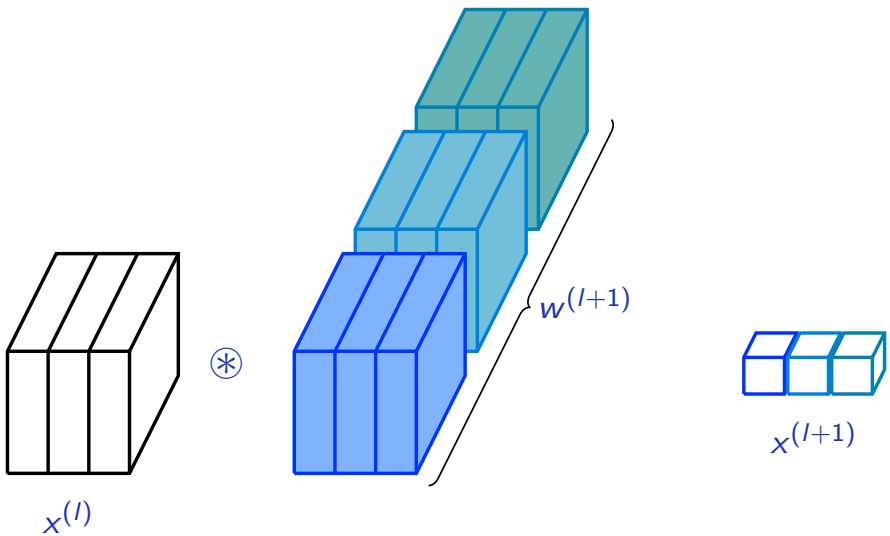
We can re-interpret a series of fully connected layers as a series of 1×1 convolutions over $D \times 1 \times 1$ tensors.



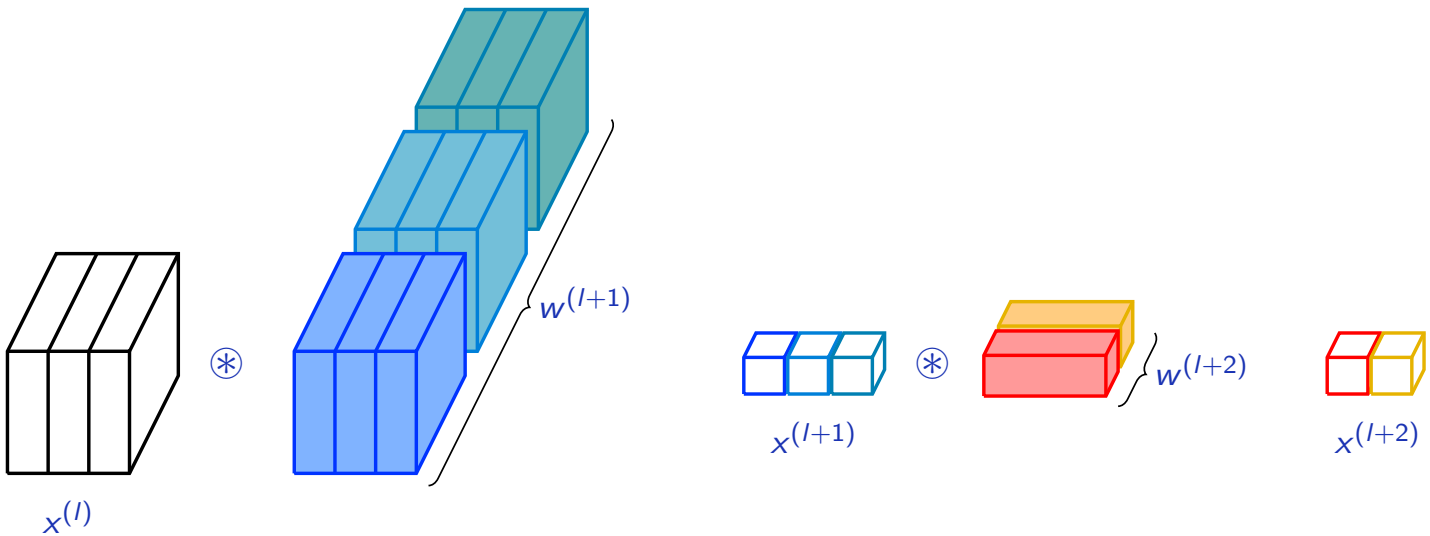
We can re-interpret a series of fully connected layers as a series of 1×1 convolutions over $D \times 1 \times 1$ tensors.



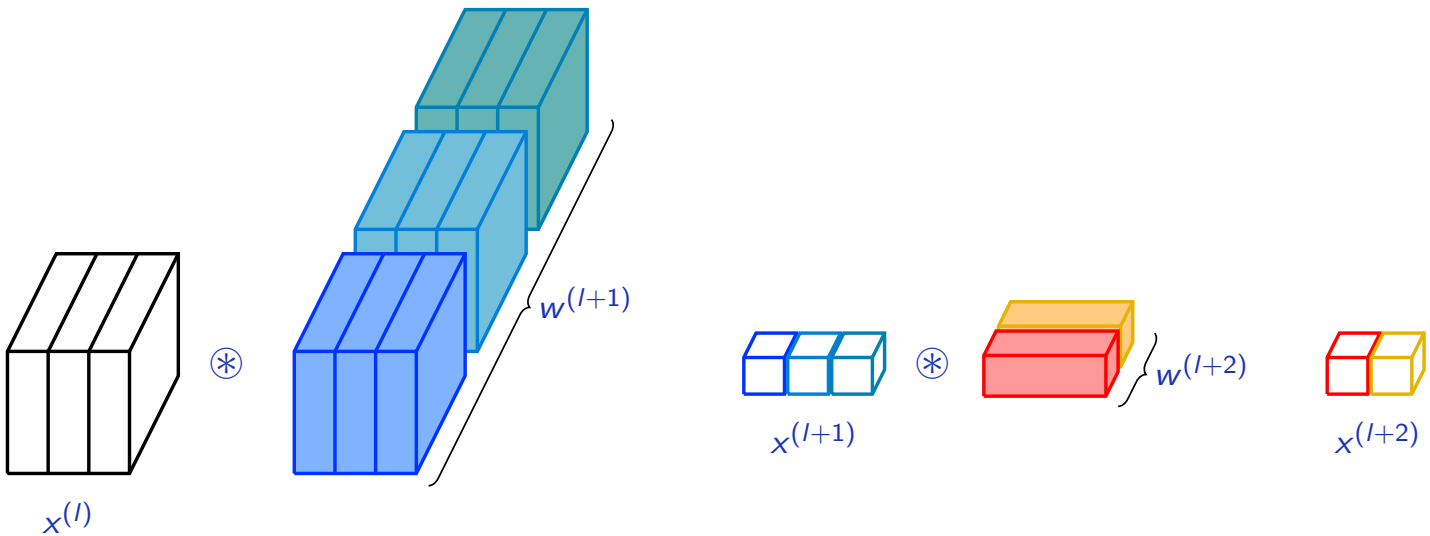
We can re-interpret a series of fully connected layers as a series of 1×1 convolutions over $D \times 1 \times 1$ tensors.



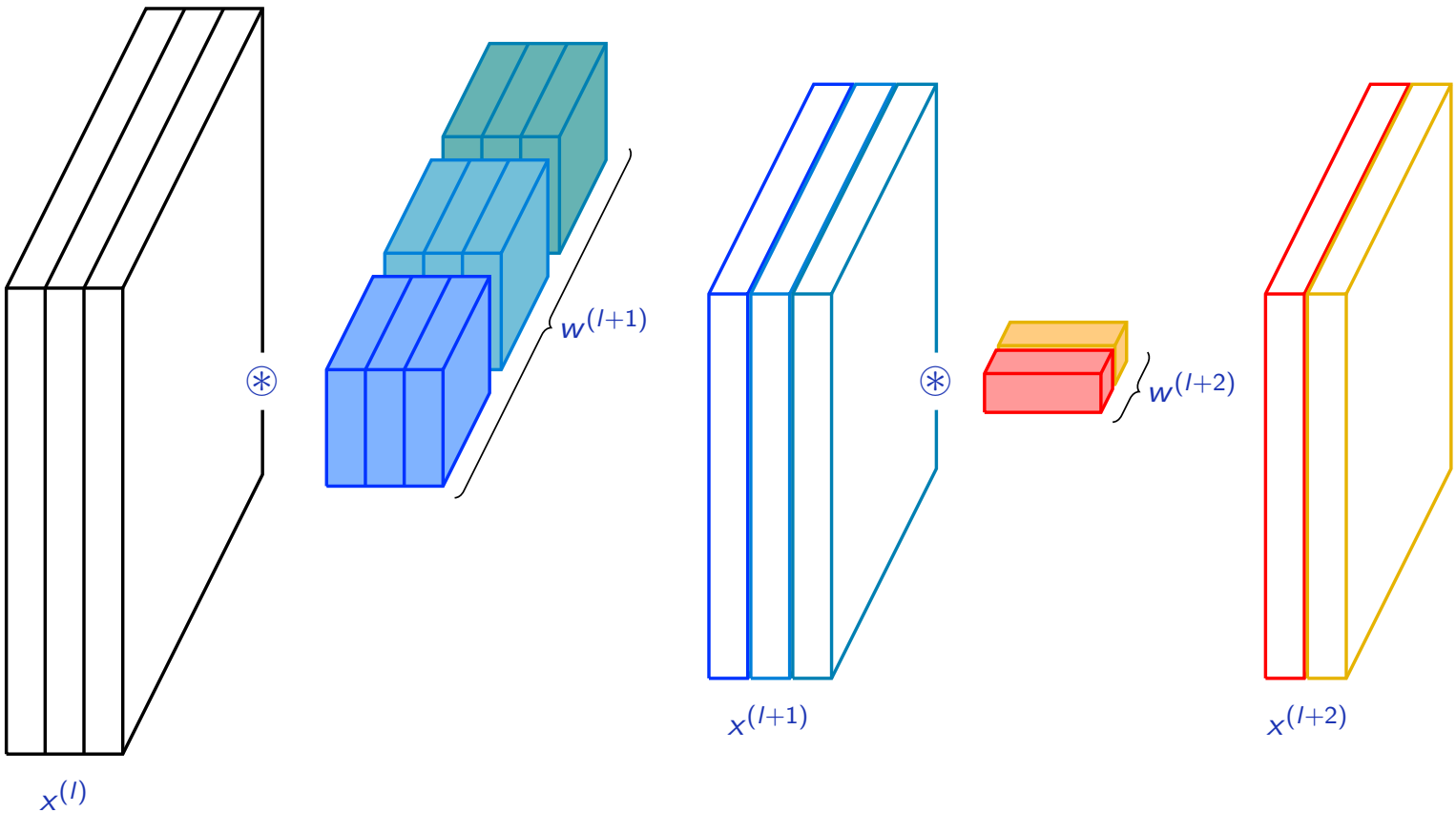
We can re-interpret a series of fully connected layers as a series of 1×1 convolutions over $D \times 1 \times 1$ tensors.



This "convolutionization" does not change anything if the input size is such that the output has a single spatial cell, but **it fully re-uses computation to get a prediction at multiple locations** when the input is larger.



This "convolutionization" does not change anything if the input size is such that the output has a single spatial cell, but **it fully re-uses computation to get a prediction at multiple locations** when the input is larger.



We can write a routine that transforms a series of layers from a standard convnets to make it fully convolutional:

```
def convolutionize(layers, input_size):
    result_layers = []
    x = torch.zeros((1, ) + input_size)

    for m in layers:
        if isinstance(m, torch.nn.Linear):
            n = torch.nn.Conv2d(in_channels = x.size(1),
                                out_channels = m.weight.size(0),
                                kernel_size = (x.size(2), x.size(3)))
            with torch.no_grad():
                n.weight.view(-1).copy_(m.weight.view(-1))
                n.bias.view(-1).copy_(m.bias.view(-1))
            m = n

        result_layers.append(m)
        x = m(x)

    return result_layers
```

This function makes the [strong and disputable] assumption that only `nn.Linear` has to be converted.

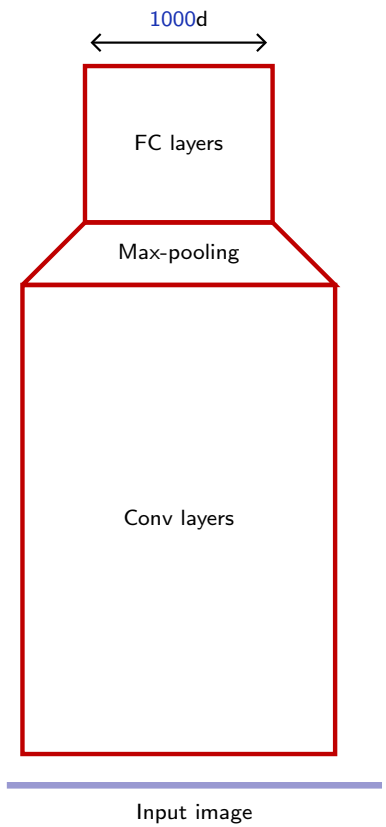
To apply this to AlexNet

```
model = torchvision.models.alexnet(pretrained = True)
print(model)

layers = list(model.features) + list(model.classifier)

model = nn.Sequential(*convolutionize(layers, (3, 224, 224)))
print(model)
```

```
AlexNet (
  (features): Sequential (
    (0): Conv2d(3, 64, kernel_size=(11, 11), stride=(4, 4), padding=(2, 2))
    (1): ReLU (inplace)
    (2): MaxPool2d (size=(3, 3), stride=(2, 2), dilation=(1, 1))
    (3): Conv2d(64, 192, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2))
    (4): ReLU (inplace)
    (5): MaxPool2d (size=(3, 3), stride=(2, 2), dilation=(1, 1))
    (6): Conv2d(192, 384, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (7): ReLU (inplace)
    (8): Conv2d(384, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (9): ReLU (inplace)
    (10): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (11): ReLU (inplace)
    (12): MaxPool2d (size=(3, 3), stride=(2, 2), dilation=(1, 1))
  )
  (classifier): Sequential (
    (0): Dropout (p = 0.5)
    (1): Linear (9216 -> 4096)
    (2): ReLU (inplace)
    (3): Dropout (p = 0.5)
    (4): Linear (4096 -> 4096)
    (5): ReLU (inplace)
    (6): Linear (4096 -> 1000)
  )
)
```

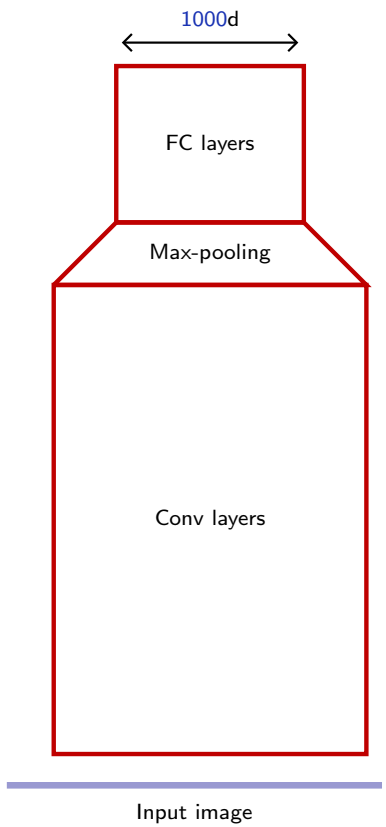
```
Sequential (
  (0): Conv2d(3, 64, kernel_size=(11, 11), stride=(4, 4), padding=(2, 2))
  (1): ReLU (inplace)
  (2): MaxPool2d (size=(3, 3), stride=(2, 2), dilation=(1, 1))
  (3): Conv2d(64, 192, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2))
  (4): ReLU (inplace)
  (5): MaxPool2d (size=(3, 3), stride=(2, 2), dilation=(1, 1))
  (6): Conv2d(192, 384, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (7): ReLU (inplace)
  (8): Conv2d(384, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (9): ReLU (inplace)
  (10): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (11): ReLU (inplace)
  (12): MaxPool2d (size=(3, 3), stride=(2, 2), dilation=(1, 1))
  (13): Dropout (p = 0.5)
  (14): Conv2d(256, 4096, kernel_size=(6, 6), stride=(1, 1))
  (15): ReLU (inplace)
  (16): Dropout (p = 0.5)
  (17): Conv2d(4096, 4096, kernel_size=(1, 1), stride=(1, 1))
  (18): ReLU (inplace)
  (19): Conv2d(4096, 1000, kernel_size=(1, 1), stride=(1, 1))
)
```

In their "overfeat" approach, Sermanet et al. (2013) combined this with a stride 1 final max-pooling to get multiple predictions.



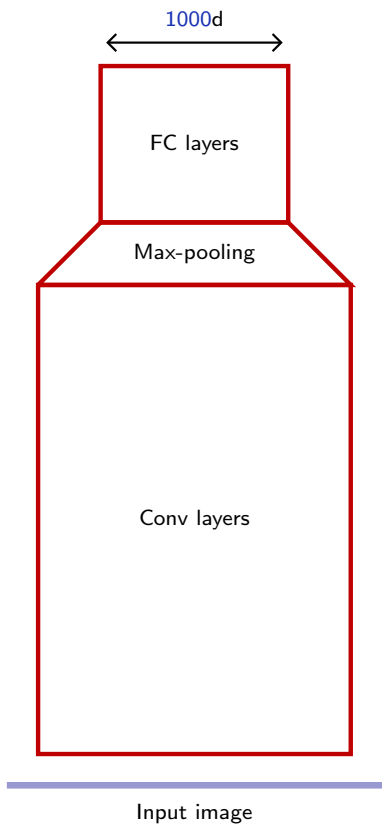
AlexNet random cropping

In their "overfeat" approach, Sermanet et al. (2013) combined this with a stride 1 final max-pooling to get multiple predictions.



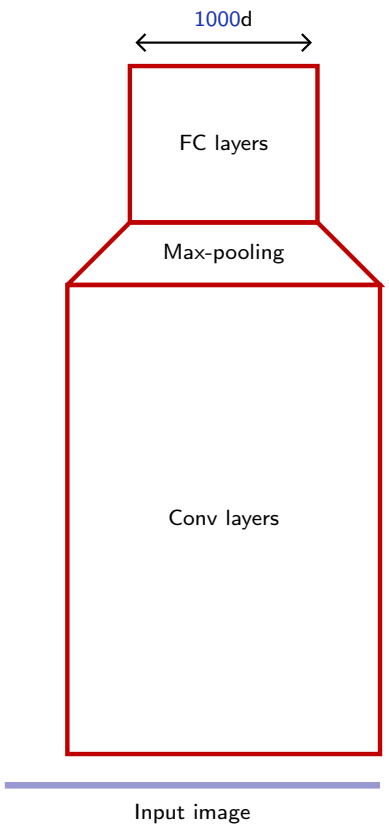
AlexNet random cropping

In their "overfeat" approach, Sermanet et al. (2013) combined this with a stride 1 final max-pooling to get multiple predictions.



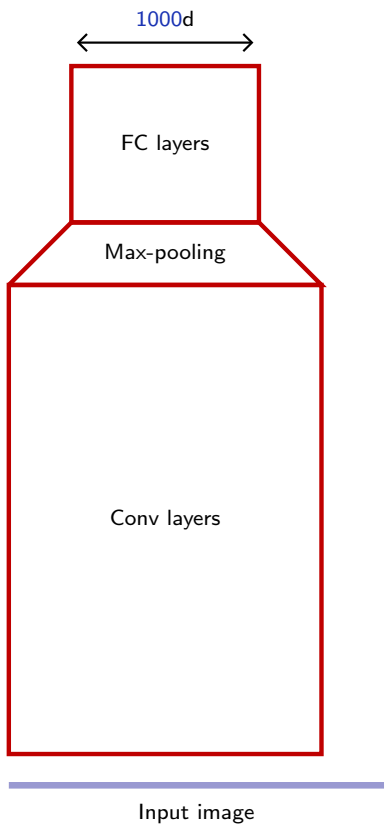
AlexNet random cropping

In their "overfeat" approach, Sermanet et al. (2013) combined this with a stride 1 final max-pooling to get multiple predictions.



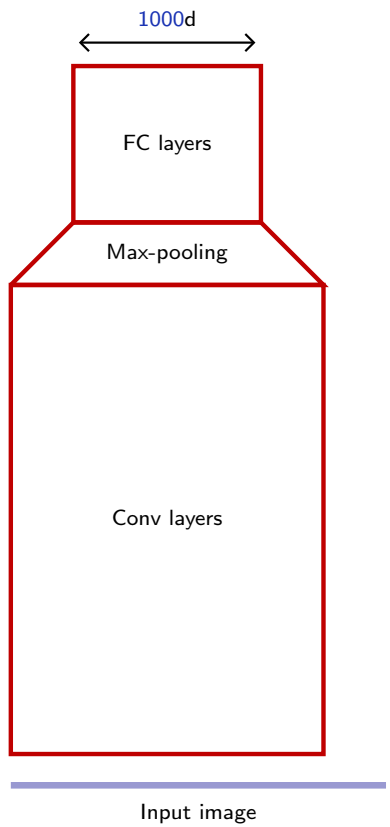
AlexNet random cropping

In their "overfeat" approach, Sermanet et al. (2013) combined this with a stride 1 final max-pooling to get multiple predictions.

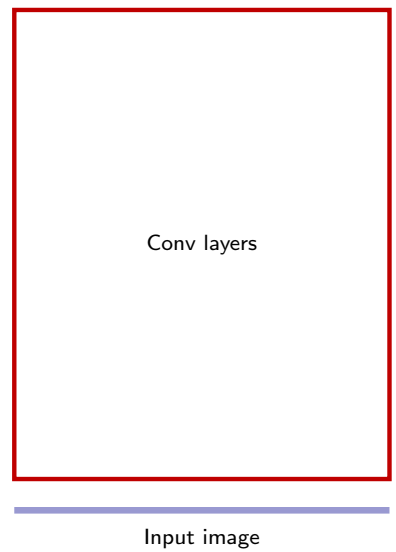


AlexNet random cropping

In their "overfeat" approach, Sermanet et al. (2013) combined this with a stride 1 final max-pooling to get multiple predictions.

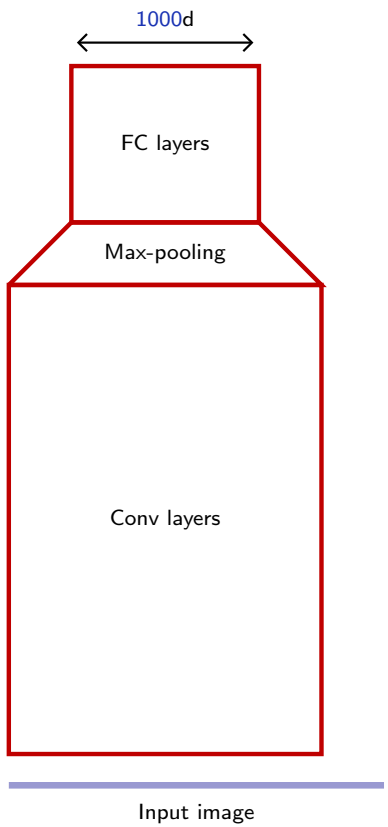


AlexNet random cropping

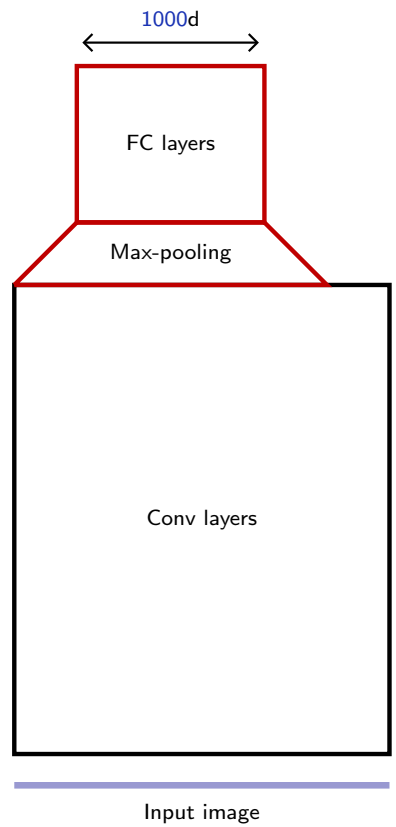


Overfeat dense max-pooling

In their "overfeat" approach, Sermanet et al. (2013) combined this with a stride 1 final max-pooling to get multiple predictions.

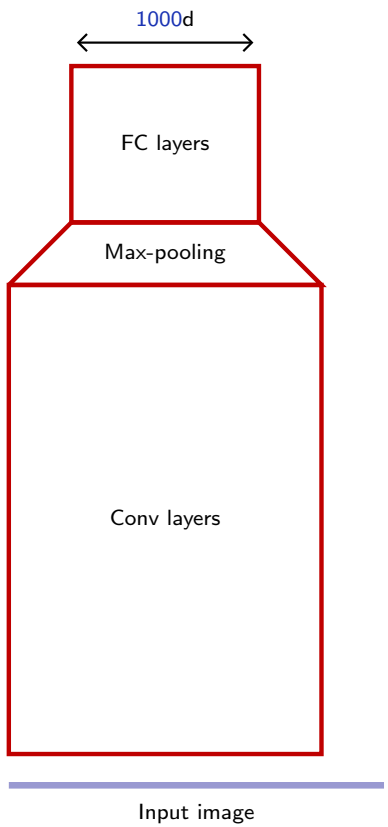


AlexNet random cropping

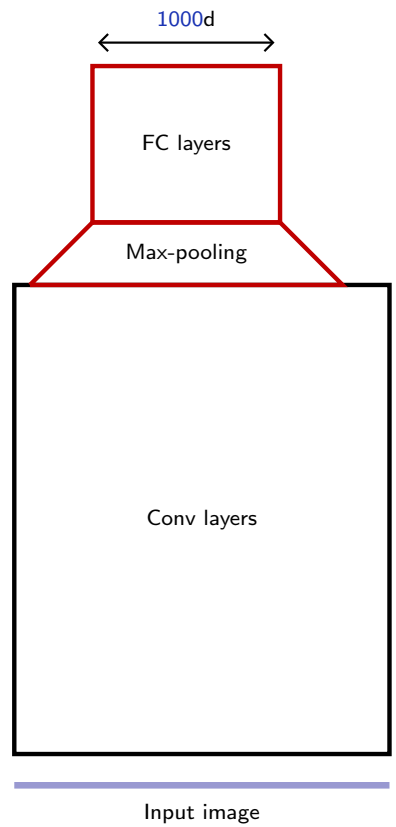


Overfeat dense max-pooling

In their "overfeat" approach, Sermanet et al. (2013) combined this with a stride 1 final max-pooling to get multiple predictions.

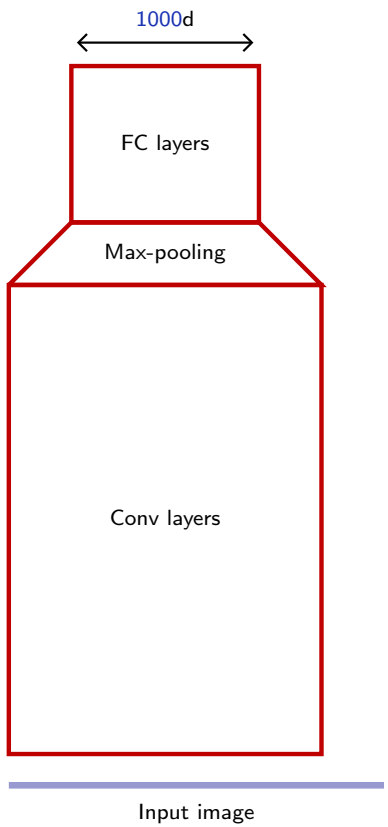


AlexNet random cropping

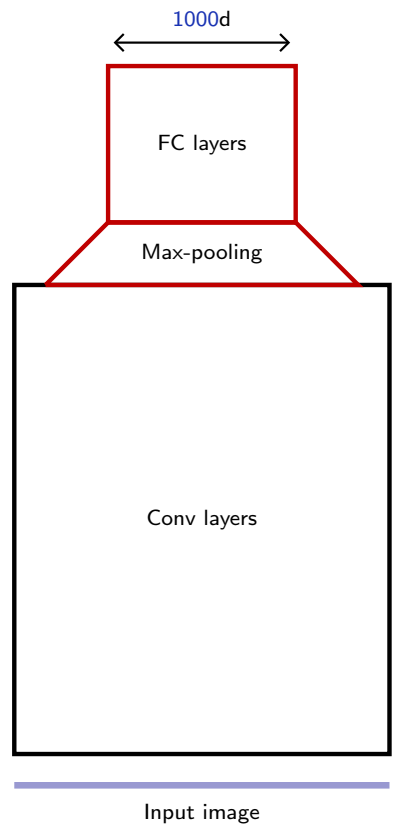


Overfeat dense max-pooling

In their "overfeat" approach, Sermanet et al. (2013) combined this with a stride 1 final max-pooling to get multiple predictions.

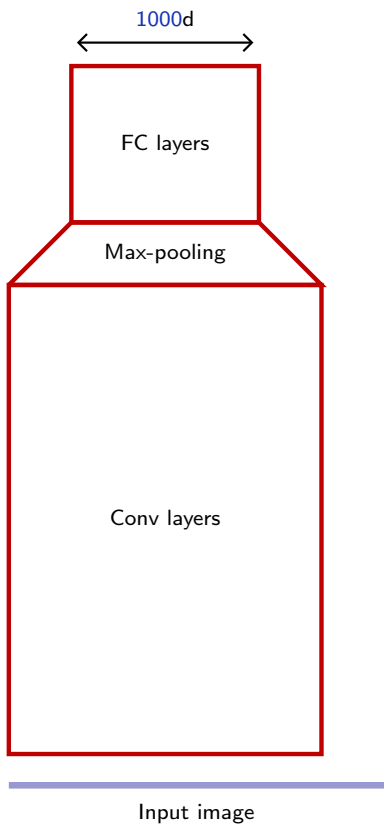


AlexNet random cropping

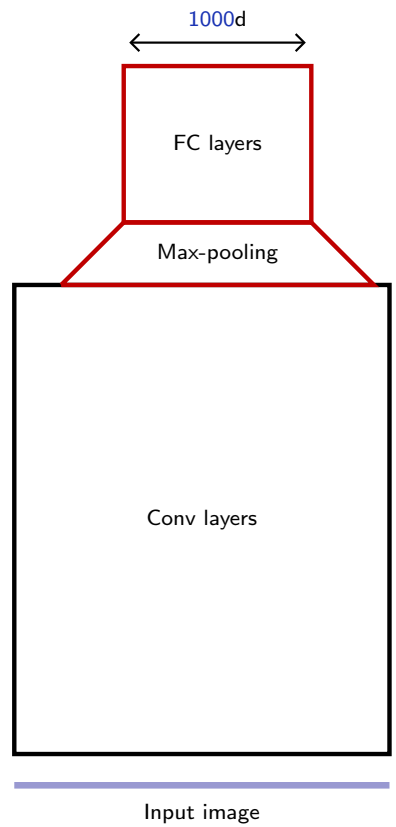


Overfeat dense max-pooling

In their "overfeat" approach, Sermanet et al. (2013) combined this with a stride 1 final max-pooling to get multiple predictions.

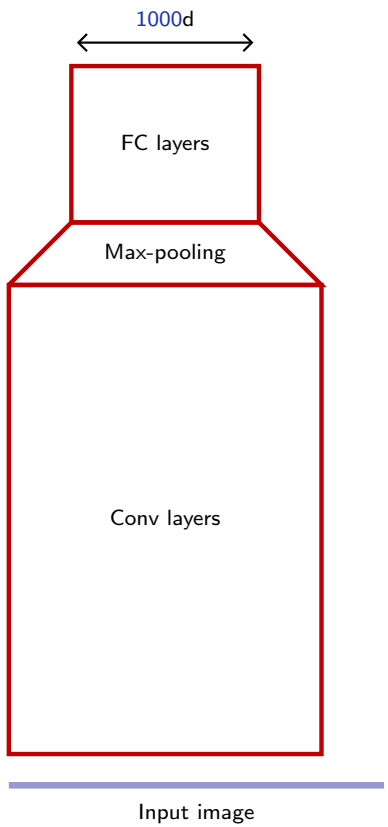


AlexNet random cropping

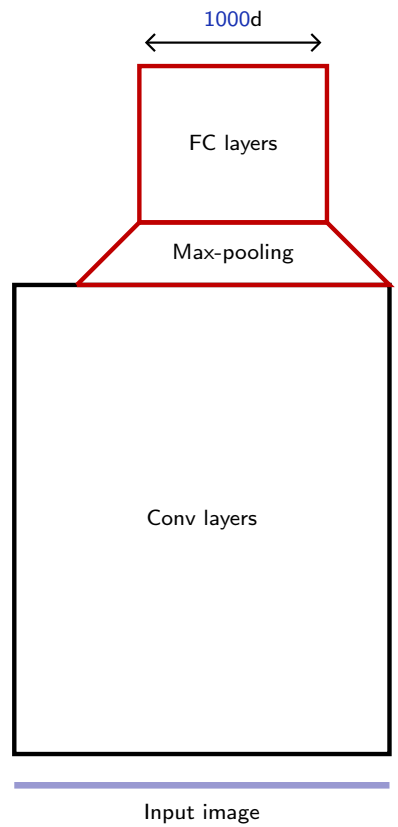


Overfeat dense max-pooling

In their "overfeat" approach, Sermanet et al. (2013) combined this with a stride 1 final max-pooling to get multiple predictions.

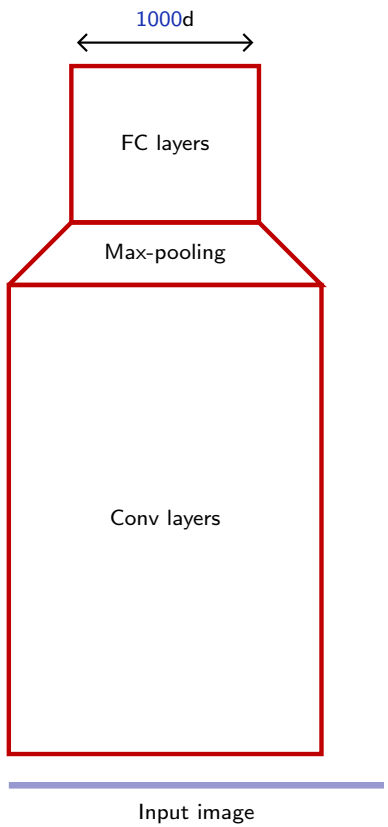


AlexNet random cropping

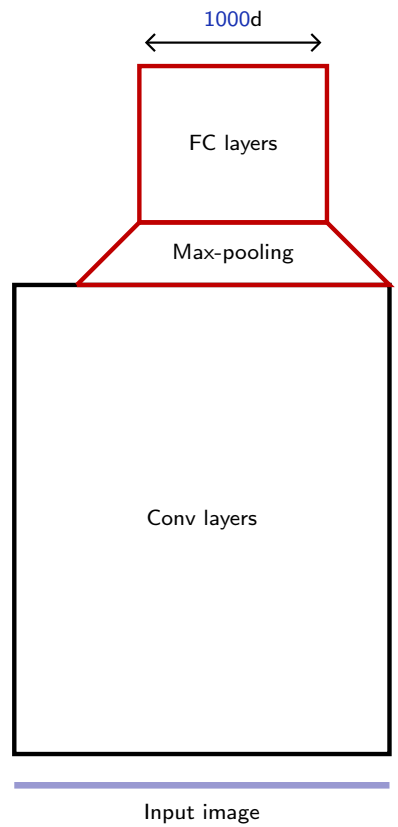


Overfeat dense max-pooling

In their "overfeat" approach, Sermanet et al. (2013) combined this with a stride 1 final max-pooling to get multiple predictions.



AlexNet random cropping



Overfeat dense max-pooling

Doing so, they could afford parsing the scene at 6 scales to improve invariance.

This "convolutionization" has a practical consequence, as we can now re-use classification networks for **dense prediction** without re-training.

Also, and maybe more importantly, it blurs the conceptual boundary between "features" and "classifier" and leads to an intuitive understanding of convnet activations as gradually transitioning from appearance to semantic.

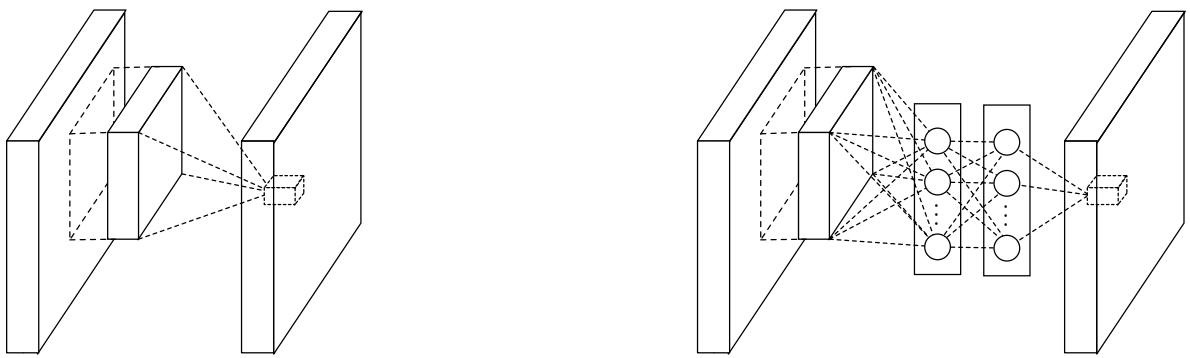
We will come back to this in lecture 9.2. "Looking at activations".

In the case of a large output prediction map, a final prediction can be obtained by averaging the final output map channel-wise.

If the last layer is linear, the averaging can be done first, as in the residual networks (He et al., 2015).

Network in network

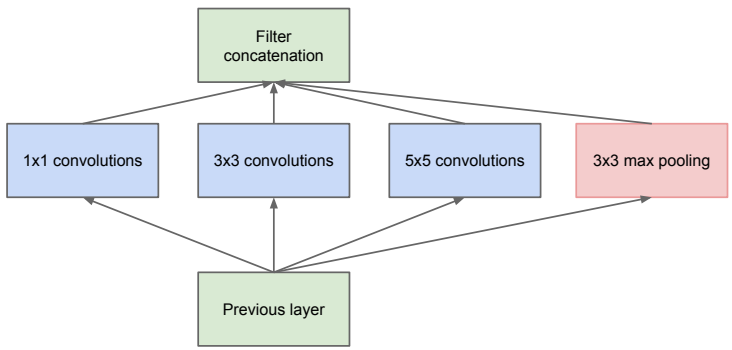
Lin et al. (2013) re-interpreted a convolution filter as a one-layer perceptron, and extended it with an "MLP convolution" (aka "network in network") to improve the capacity vs. parameter ratio.



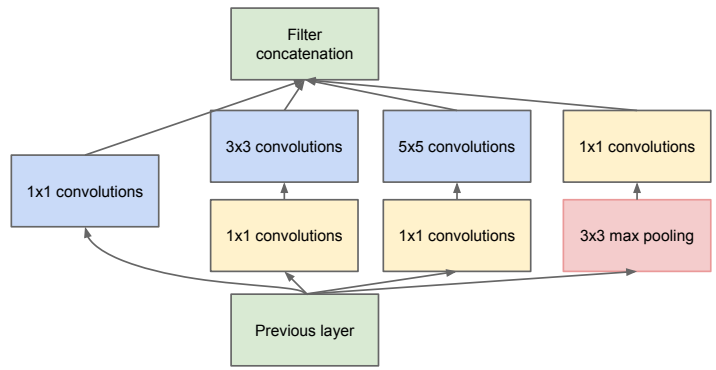
(Lin et al., 2013)

As for the fully convolutional networks, such local MLPs can be implemented with 1×1 convolutions.

The same notion was generalized by Szegedy et al. (2015) for their GoogLeNet, through the use of module combining convolutions at multiple scales to let the optimal ones be picked during training.



(a) Inception module, naïve version



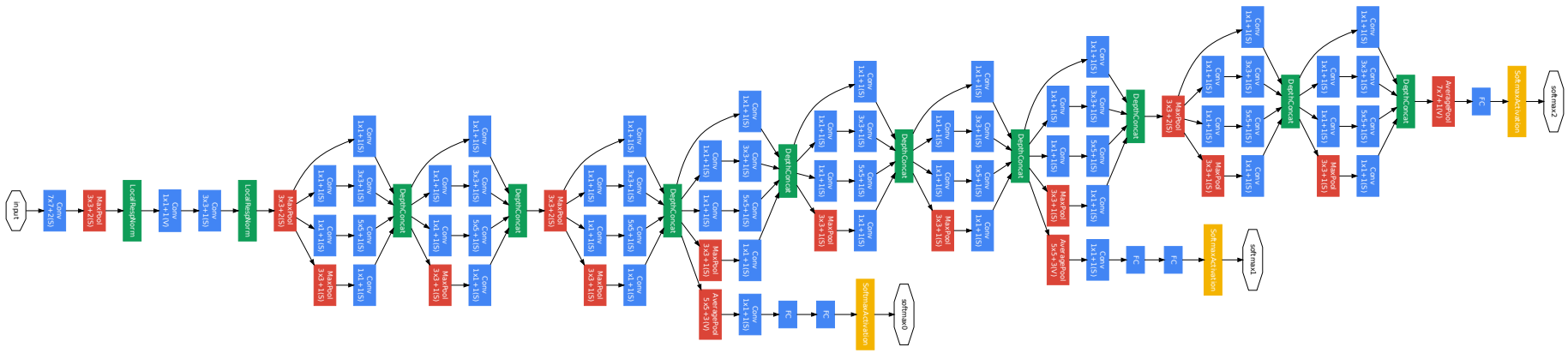
(b) Inception module with dimension reductions

(Szegedy et al., 2015)

Szegedy et al. (2015) also introduce the idea of **auxiliary classifiers** to help the propagation of the gradient in the early layers.

This is motivated by the reasonable performance of shallow networks that indicates early layers already encode informative and invariant features.

The resulting GoogLeNet has 12 times less parameters than AlexNet and is more accurate on ILSVRC14 (Szegedy et al., 2015).



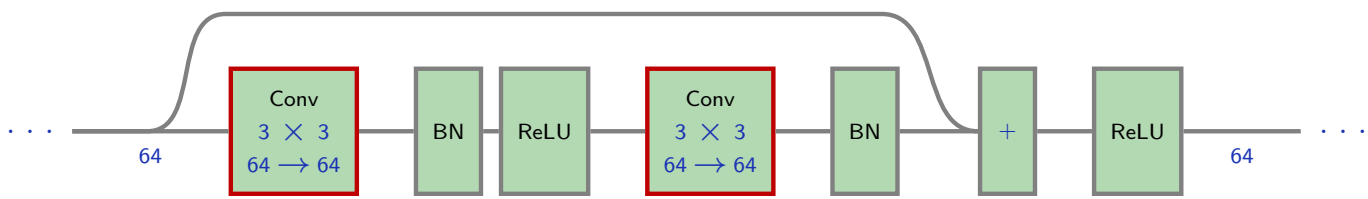
(Szegedy et al., 2015)

It was later extended with techniques we are going to see in the next slides: batch-normalization (Ioffe and Szegedy, 2015) and pass-through à la resnet (Szegedy et al., 2016).

Residual networks

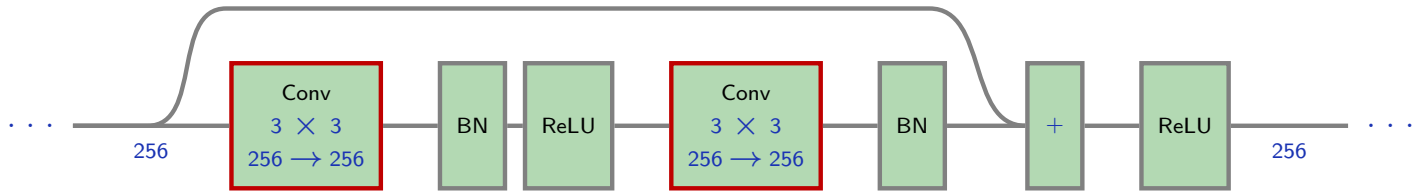
We already saw the structure of the residual networks and how well they perform on CIFAR10 (He et al., 2015).

The default residual block proposed by He et al. is of the form



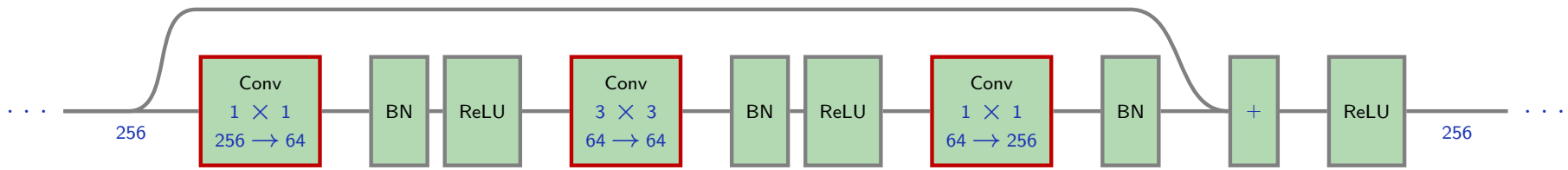
and as such requires $2 \times (3 \times 3 \times 64 + 1) \times 64 \simeq 73k$ parameters.

To apply the same architecture to ImageNet, more channels are required, e.g.



However, such a block requires $2 \times (3 \times 3 \times 256 + 1) \times 256 \simeq 1.2m$ parameters.

They mitigated that requirement with what they call a **bottleneck** block:



$256 \times 64 + (3 \times 3 \times 64 + 1) \times 64 + 64 \times 256 \simeq 70k$ parameters.

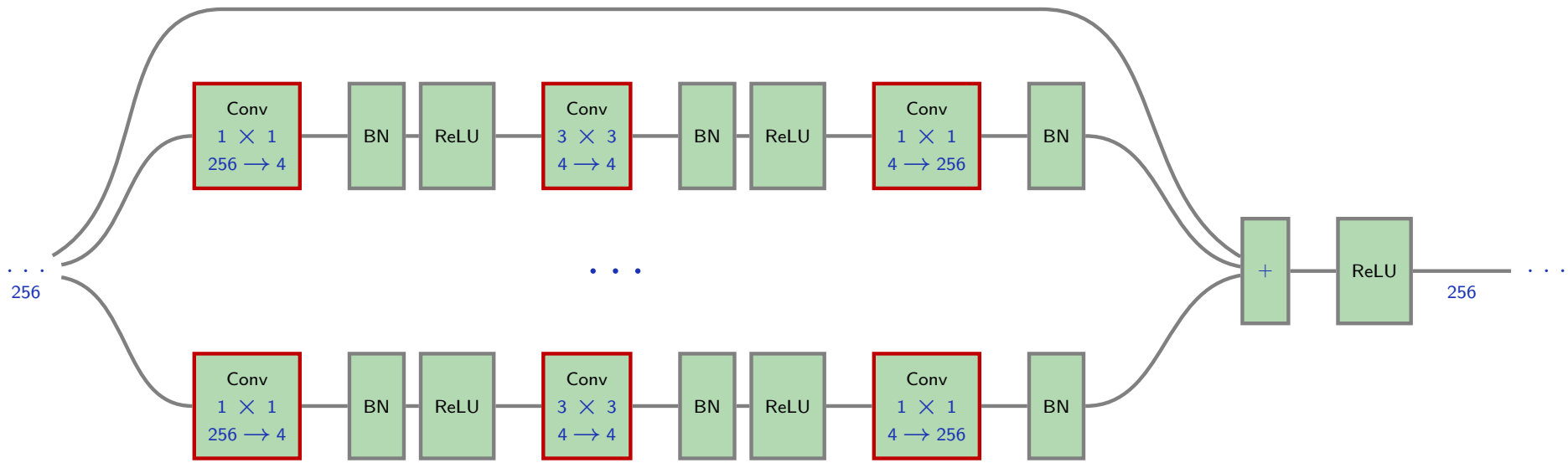
The encoding pushed between blocks is high-dimensional, but the "contextual reasoning" in convolutional layers is done on a simpler feature representation.

method	top-5 err. (**test**)
VGG [41] (ILSVRC'14)	7.32
GoogLeNet [44] (ILSVRC'14)	6.66
VGG [41] (v5)	6.8
PReLU-net [13]	4.94
BN-inception [16]	4.82
ResNet (ILSVRC'15)	**3.57**

Table 5. Error rates (%) of **ensembles**. The top-5 error is on the test set of ImageNet and reported by the test server.

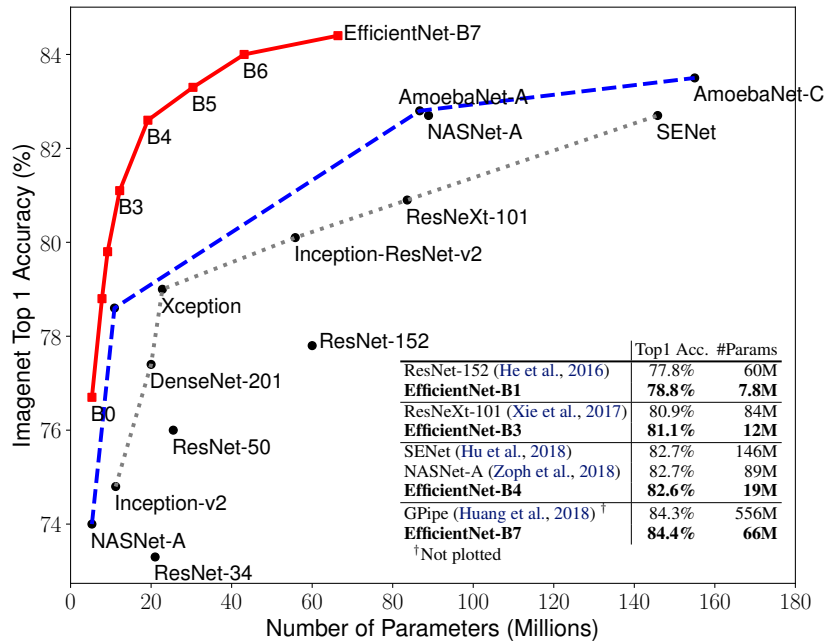
(He et al., 2015)

This was extended to the ResNeXt architecture by Xie et al. (2016), with blocks with similar number of parameters, but split into 32 "aggregated" pathways.



When equalizing the number of parameters, this architecture performs better than a standard resnet.

Tan and Le (2019) proposed to scale depth, width, and resolutions uniformly when increasing the size of a network.



	Top1 Acc.	#Params
ResNet-152 (He et al., 2016)	77.8%	60M
EfficientNet-B1	**78.8%**	**7.8M**
ResNeXt-101 (Xie et al., 2017)	80.9%	84M
EfficientNet-B3	**81.1%**	**12M**
SENet (Hu et al., 2018)	82.7%	146M
NASNet-A (Zoph et al., 2018)	82.7%	89M
EfficientNet-B4	**82.6%**	**19M**
GPipe (Huang et al., 2018) †	84.3%	556M
EfficientNet-B7	**84.4%**	**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 84.4% 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.

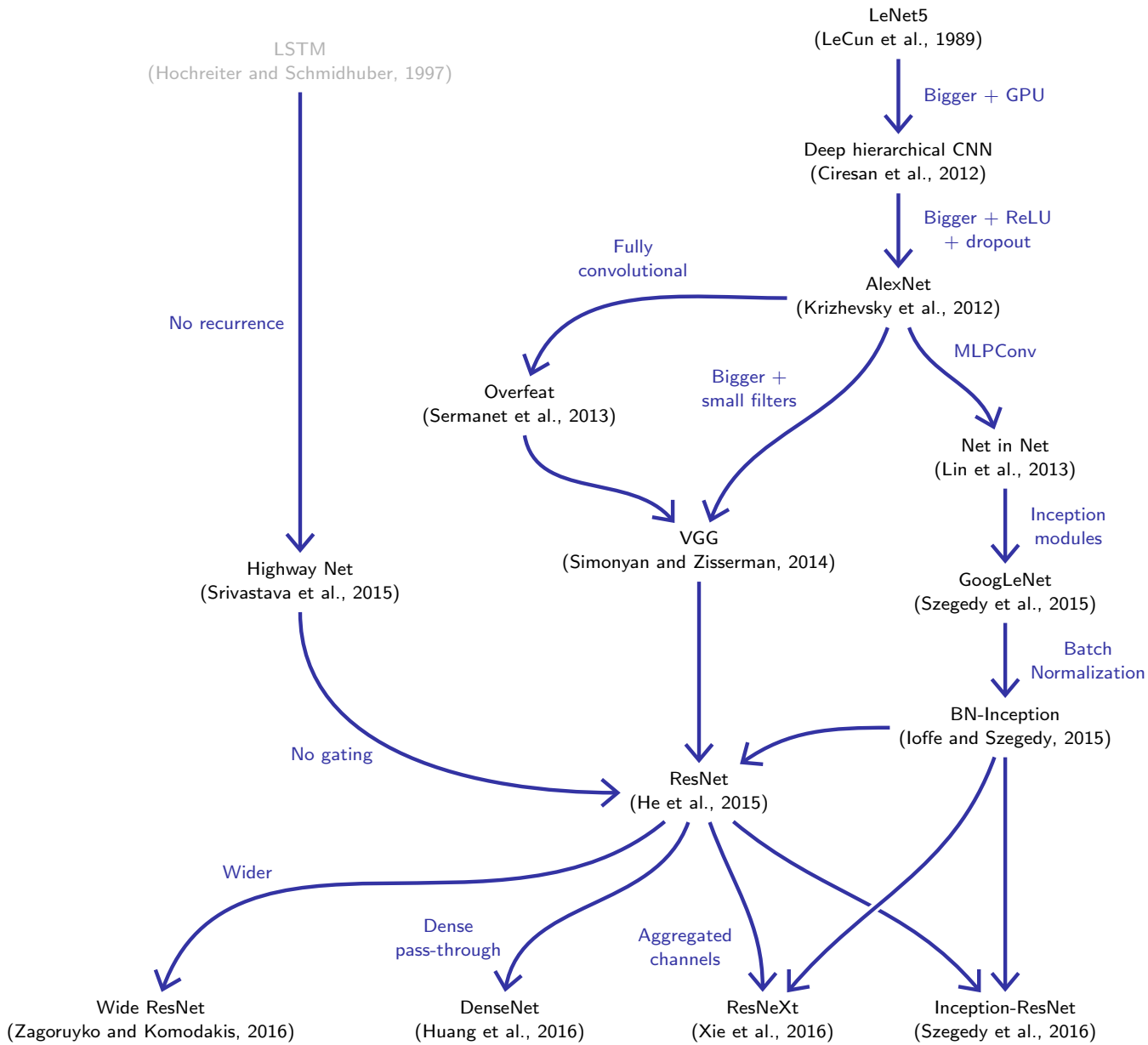
(Tan and Le, 2019)

Summary

To summarize roughly the evolution of convnets for image classification:

- standard ones are extensions of LeNet5,
- everybody loves ReLU,
- state-of-the-art networks have 100s of channels and 10s of layers,
- they can (should?) be fully convolutional,
- pass-through connections allow deeper "residual" nets,
- bottleneck local structures reduce the number of parameters,
- aggregated pathways reduce the number of parameters.

Image classification networks



Deep learning

8.3. Networks for object detection

François Fleuret

https://fleuret.org/dlc/

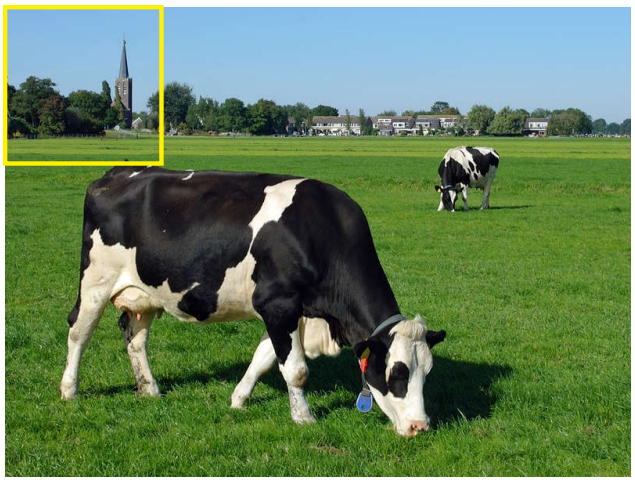
UNIVERSITÉ
DE GENÈVE

The simplest strategy for object detection is to classify local regions, at multiple scales and locations.

Parsing at fixed scale

Final list of detections

The simplest strategy for object detection is to classify local regions, at multiple scales and locations.



Parsing at fixed scale

Final list of detections

The simplest strategy for object detection is to classify local regions, at multiple scales and locations.

Parsing at fixed scale

Final list of detections

The simplest strategy for object detection is to classify local regions, at multiple scales and locations.

Parsing at fixed scale

Final list of detections

The simplest strategy for object detection is to classify local regions, at multiple scales and locations.

Parsing at fixed scale

Final list of detections

The simplest strategy for object detection is to classify local regions, at multiple scales and locations.

Parsing at fixed scale

Final list of detections

The simplest strategy for object detection is to classify local regions, at multiple scales and locations.

Parsing at fixed scale

Final list of detections

The simplest strategy for object detection is to classify local regions, at multiple scales and locations.

Parsing at fixed scale

Final list of detections

The simplest strategy for object detection is to classify local regions, at multiple scales and locations.

Parsing at fixed scale

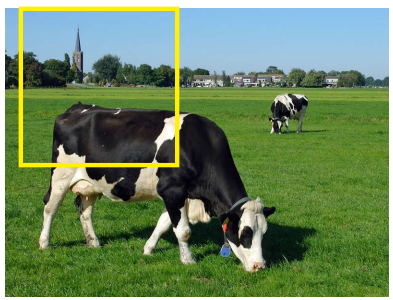
Final list of detections

The simplest strategy for object detection is to classify local regions, at multiple scales and locations.

Parsing at fixed scale

Final list of detections

The simplest strategy for object detection is to classify local regions, at multiple scales and locations.



Parsing at fixed scale

Final list of detections

The simplest strategy for object detection is to classify local regions, at multiple scales and locations.

Parsing at fixed scale

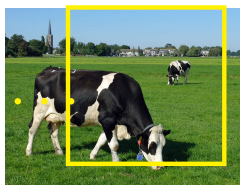
Final list of detections

The simplest strategy for object detection is to classify local regions, at multiple scales and locations.

Parsing at fixed scale

Final list of detections

The simplest strategy for object detection is to classify local regions, at multiple scales and locations.



Parsing at fixed scale

Final list of detections

The simplest strategy for object detection is to classify local regions, at multiple scales and locations.

Parsing at fixed scale

Final list of detections

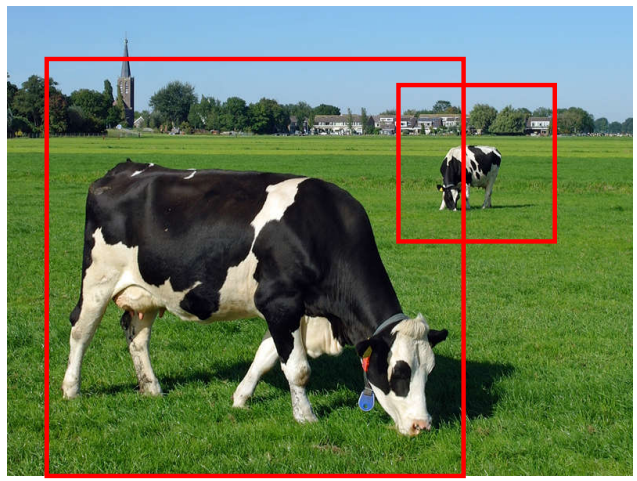
The simplest strategy for object detection is to classify local regions, at multiple scales and locations.

Parsing at fixed scale

Final list of detections

The simplest strategy for object detection is to classify local regions, at multiple scales and locations.

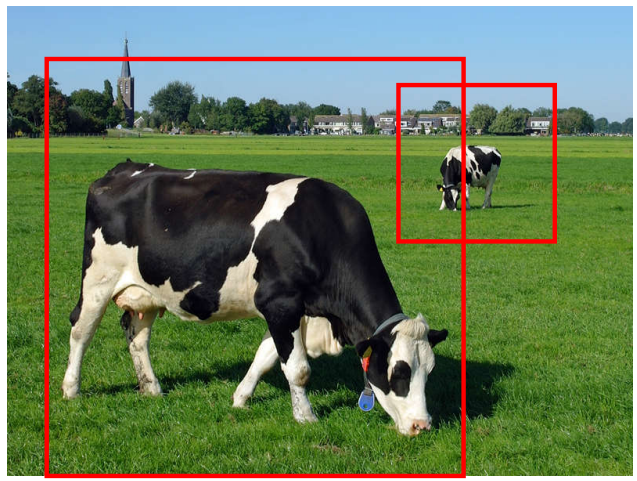
Parsing at fixed scale



Final list of detections

The simplest strategy for object detection is to classify local regions, at multiple scales and locations.

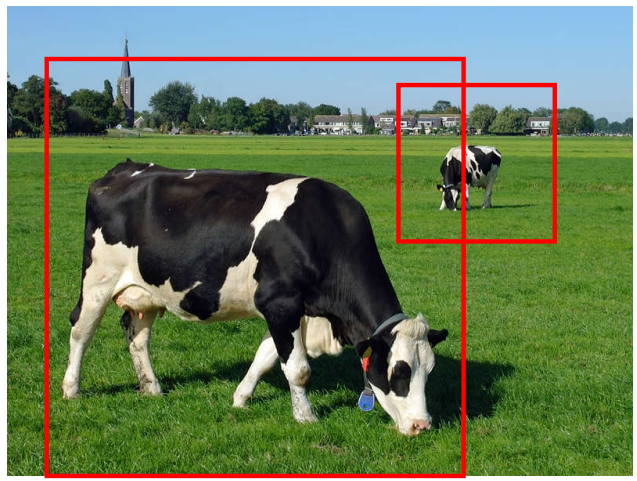
Parsing at fixed scale



Final list of detections

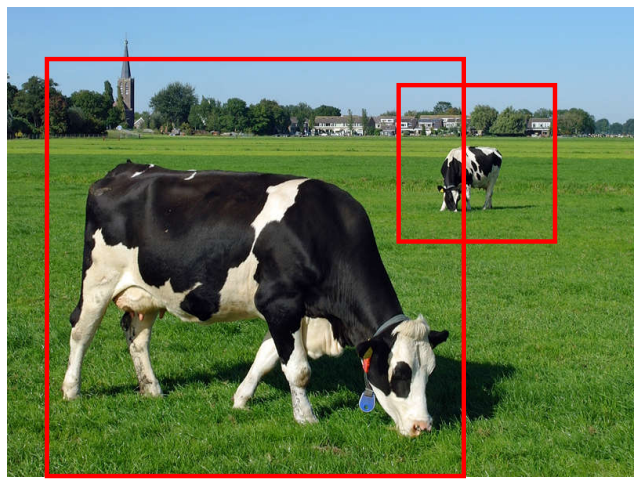
The simplest strategy for object detection is to classify local regions, at multiple scales and locations.

Parsing at fixed scale



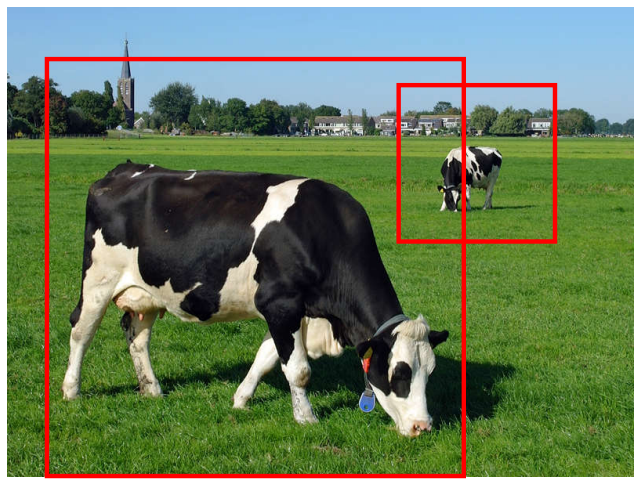
Final list of detections

The simplest strategy for object detection is to classify local regions, at multiple scales and locations.



Final list of detections

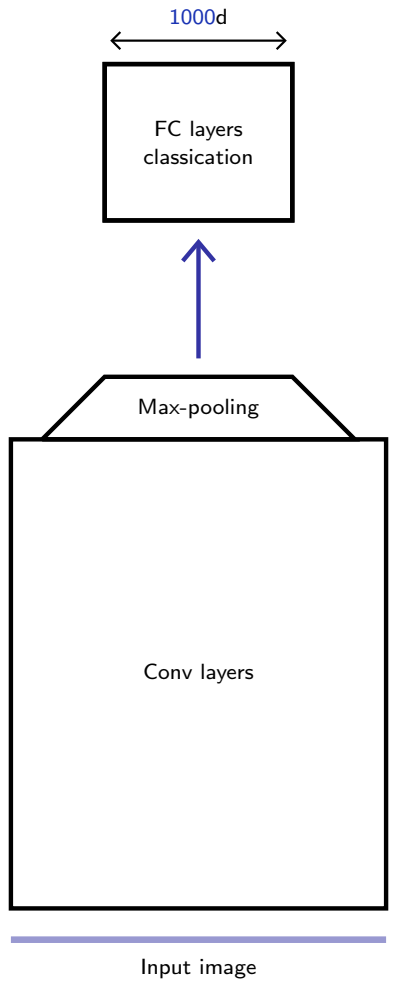
The simplest strategy for object detection is to classify local regions, at multiple scales and locations.



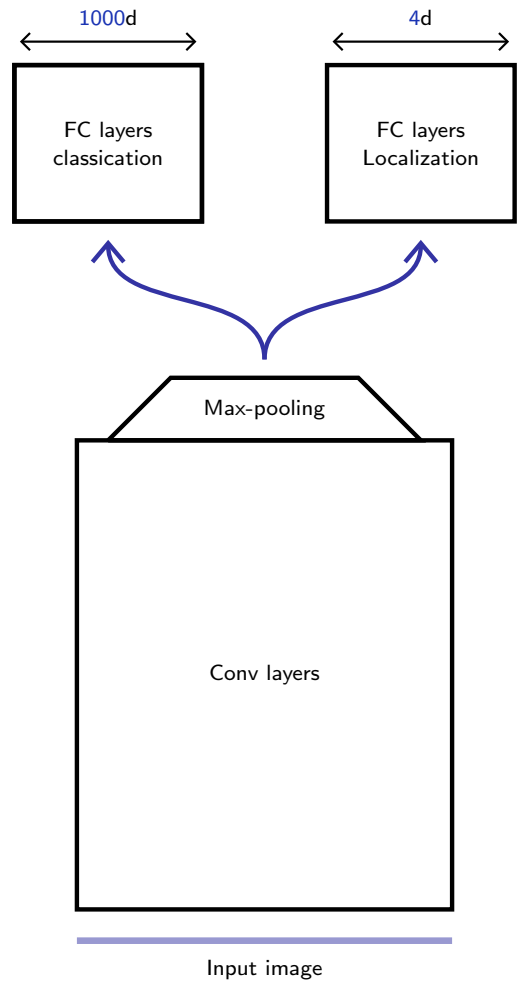
Final list of detections

This "sliding window" approach evaluates a classifier multiple times, and its computational cost increases with the prediction accuracy.

This was mitigated in overfeat (Sermanet et al., 2013) by adding a regression part to predict the object's bounding box.



This was mitigated in overfeat (Sermanet et al., 2013) by adding a regression part to predict the object's bounding box.



In the single-object case, the convolutional layers are frozen, and the localization layers are trained with a L_2 loss.

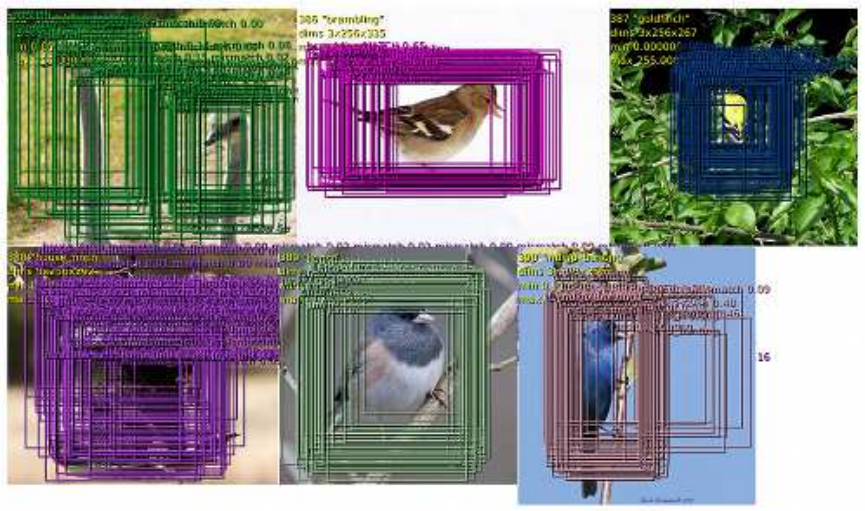


Figure 7: **Examples of bounding boxes produced by the regression network**, before being combined into final predictions. The examples shown here are at a single scale. Predictions may be more optimal at other scales depending on the objects. Here, most of the bounding boxes which are initially organized as a grid, converge to a single location and scale. This indicates that the network is very confident in the location of the object, as opposed to being spread out randomly. The top left image shows that it can also correctly identify multiple location if several objects are present. The various aspect ratios of the predicted bounding boxes shows that the network is able to cope with various object poses.

(Sermanet et al., 2013)

Combining the multiple boxes is done with an *ad hoc* greedy algorithm.

This architecture can be applied directly to detection by adding a class "Background" to the object classes.

Negative samples are taken in each scene either at random or by selecting the ones with the worst miss-classification.

Surprisingly, using class-specific localization layers did not provide better results than having a single one shared across classes (Sermanet et al., 2013).

Other approaches evolved from AlexNet, relying on **region proposals**:

- Generate thousands of proposal bounding boxes with a non-CNN "objectness" approach such as Selective search (Uijlings et al., 2013),
- feed to an AlexNet-like network sub-images cropped and warped from the input image ("R-CNN", Girshick et al., 2013), or from the convolutional feature maps to share computation ("Fast R-CNN", Girshick, 2015).

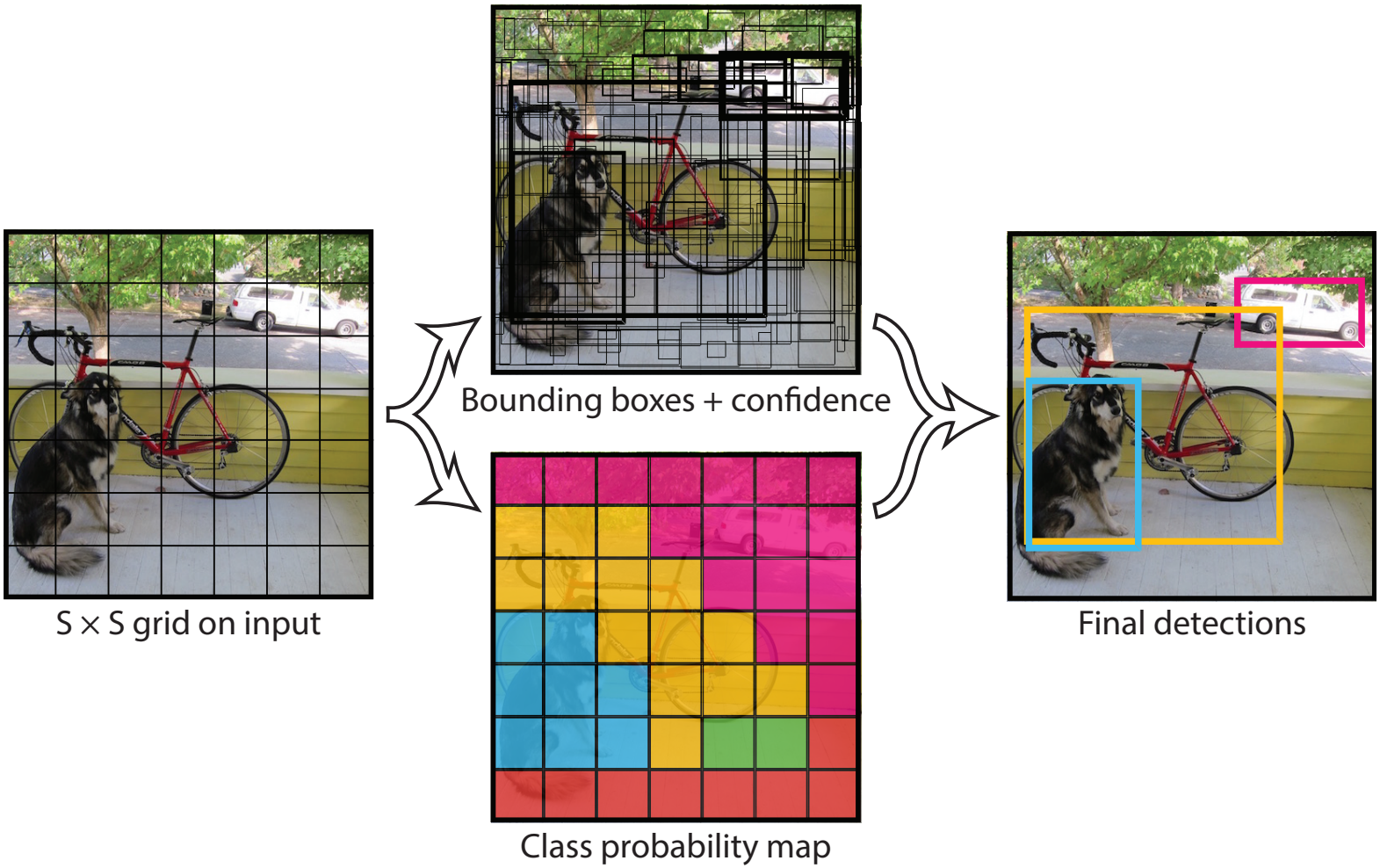
These methods suffer from the cost of the region proposal computation, which is non-convolutional and not implementable on GPU.

They were improved by Ren et al. (2015) in "Faster R-CNN" by replacing the region proposal algorithm with a convolutional processing similar to Overfeat.

The most famous algorithm from this lineage is "You Only Look Once" (YOLO, Redmon et al. 2015).

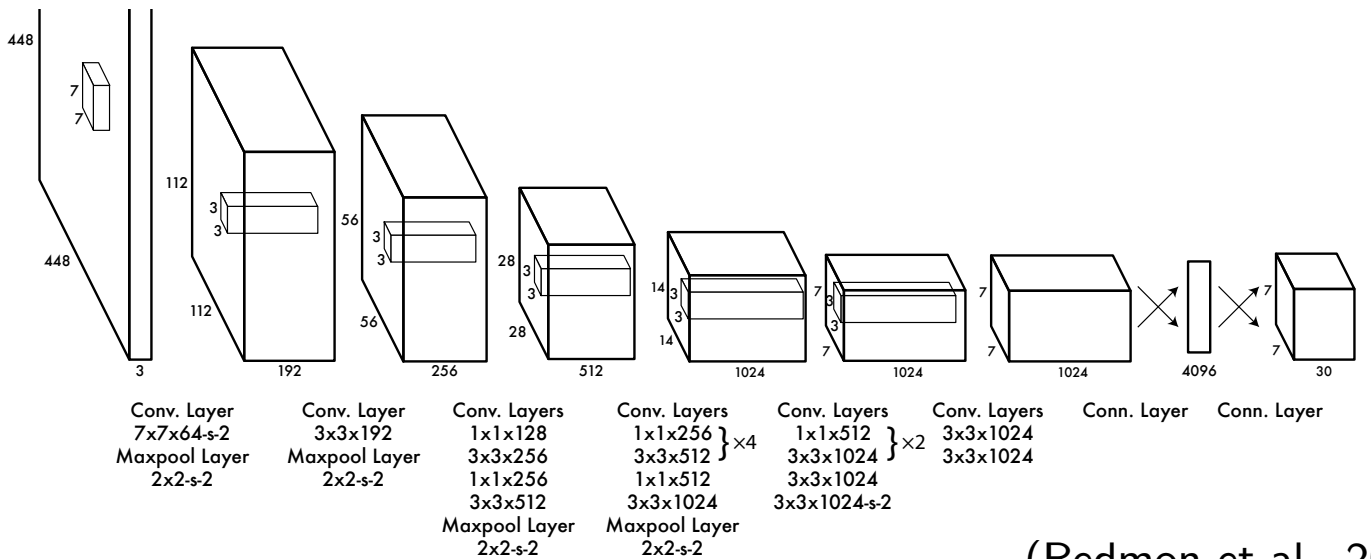
It comes back to a classical architecture with a series of convolutional layers followed by a few fully connected layers. It is sometime described as "one shot" since a single information pathway suffices.

YOLO's network is not a pre-existing one. It uses leaky ReLU, and its convolutional layers make use of the 1×1 bottleneck filters (Lin et al., 2013) to control the memory footprint and computational cost.



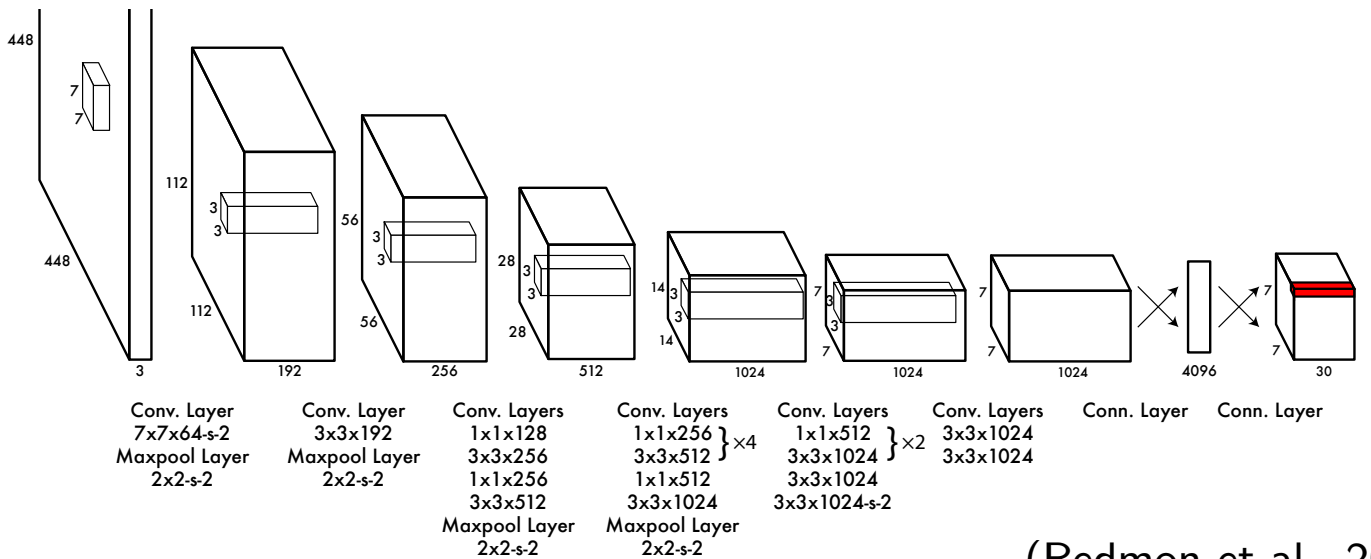
(Redmon et al., 2015)

The output corresponds to splitting the image into a regular $S \times S$ grid, with $S = 7$



(Redmon et al., 2015)

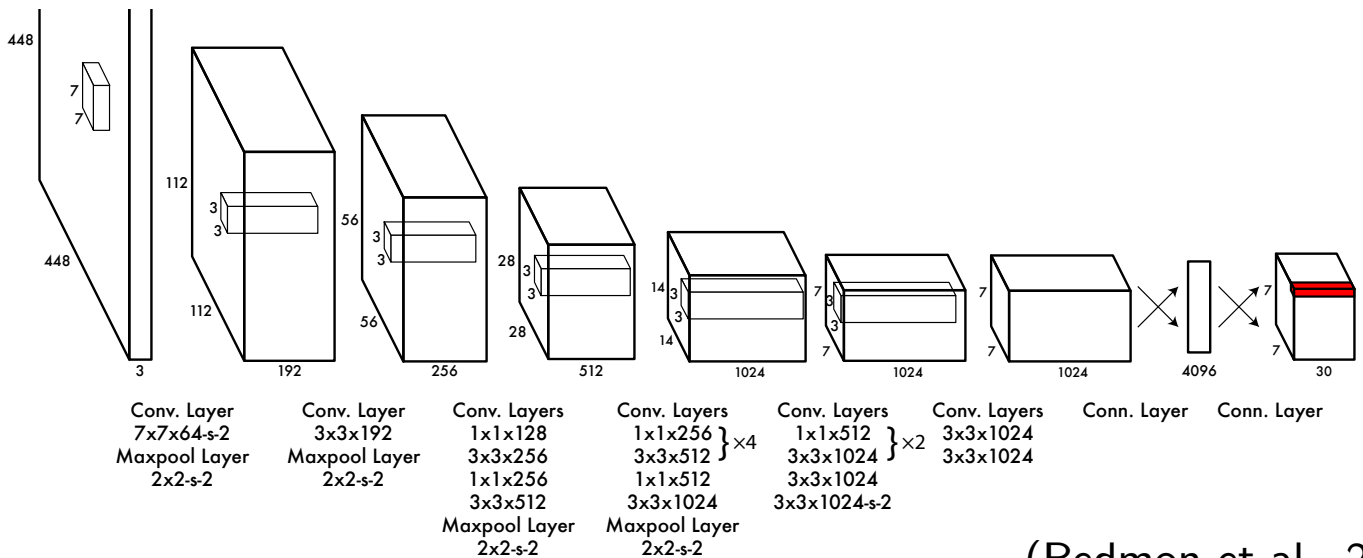
The output corresponds to splitting the image into a regular $S \times S$ grid, with $S = 7$, and for each cell, to predict a 30d vector



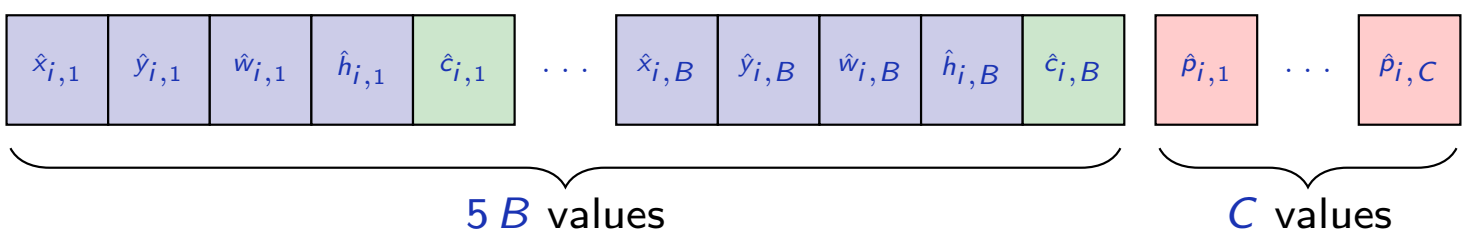
(Redmon et al., 2015)

The output corresponds to splitting the image into a regular $S \times S$ grid, with $S = 7$, and for each cell, to predict a 30d vector:

- $B = 2$ bounding boxes coordinates and confidence,
- $C = 20$ class probabilities, corresponding to the classes of Pascal VOC.



(Redmon et al., 2015)



So the network predicts class scores and bounding-box regressions, and **although the output comes from fully connected layers, it has a 2D structure**.

It allows in particular YOLO to leverage the absolute location in the image to improve performance (e.g. vehicles tend to be at the bottom, umbrella at the top), which may or may not be desirable.

During training, YOLO makes the assumption that any of the S^2 cells contains at most [the center of] a single object. We define for every image, cell index $i = 1, \ldots, S^2$, predicted box index $j = 1, \ldots, B$ and class index $c = 1, \ldots, C$

- $\mathbf{1}_i^{obj}$ is 1 if there is an object in cell i and 0 otherwise,
- $\mathbf{1}_{i,j}^{obj}$ is 1 if there is an object in cell i and predicted box j is the most fitting one, 0 otherwise.
- $p_{i,c}$ is 1 if there is an object of class c in cell i, and 0 otherwise,
- x_i, y_i, w_i, h_i the annotated object bounding box (defined only if $\mathbf{1}_i^{obj} = 1$, and relative in location and scale to the cell),
- $c_{i,j}$ IOU between the predicted box and the ground truth target.

The training procedure first computes on each image the value of the $\mathbf{1}_{i,j}^{obj}$s and $c_{i,j}$, and then does one step to minimize

$$\lambda_{coord} \sum_{i=1}^{S^2} \sum_{j=1}^{B} \mathbf{1}_{i,j}^{obj} \left((x_i - \hat{x}_{i,j})^2 + (y_i - \hat{y}_{i,j})^2 + \left(\sqrt{w_i} - \sqrt{\hat{w}_{i,j}} \right)^2 + \left(\sqrt{h_i} - \sqrt{\hat{h}_{i,j}} \right)^2 \right)$$

$$+ \lambda_{obj} \sum_{i=1}^{S^2} \sum_{j=1}^{B} \mathbf{1}_{i,j}^{obj} (c_{i,j} - \hat{c}_{i,j})^2 + \lambda_{noobj} \sum_{i=1}^{S^2} \sum_{j=1}^{B} \left(1 - \mathbf{1}_{i,j}^{obj} \right) \hat{c}_{i,j}^2$$

$$+ \lambda_{classes} \sum_{i=1}^{S^2} \mathbf{1}_{i}^{obj} \sum_{c=1}^{C} \left(p_{i,c} - \hat{p}_{i,c} \right)^2 .$$

where $\hat{p}_{i,c}, \hat{x}_{i,j}, \hat{y}_{i,j}, \hat{w}_{i,j}, \hat{h}_{i,j}, \hat{c}_{i,j}$ are the network's outputs.

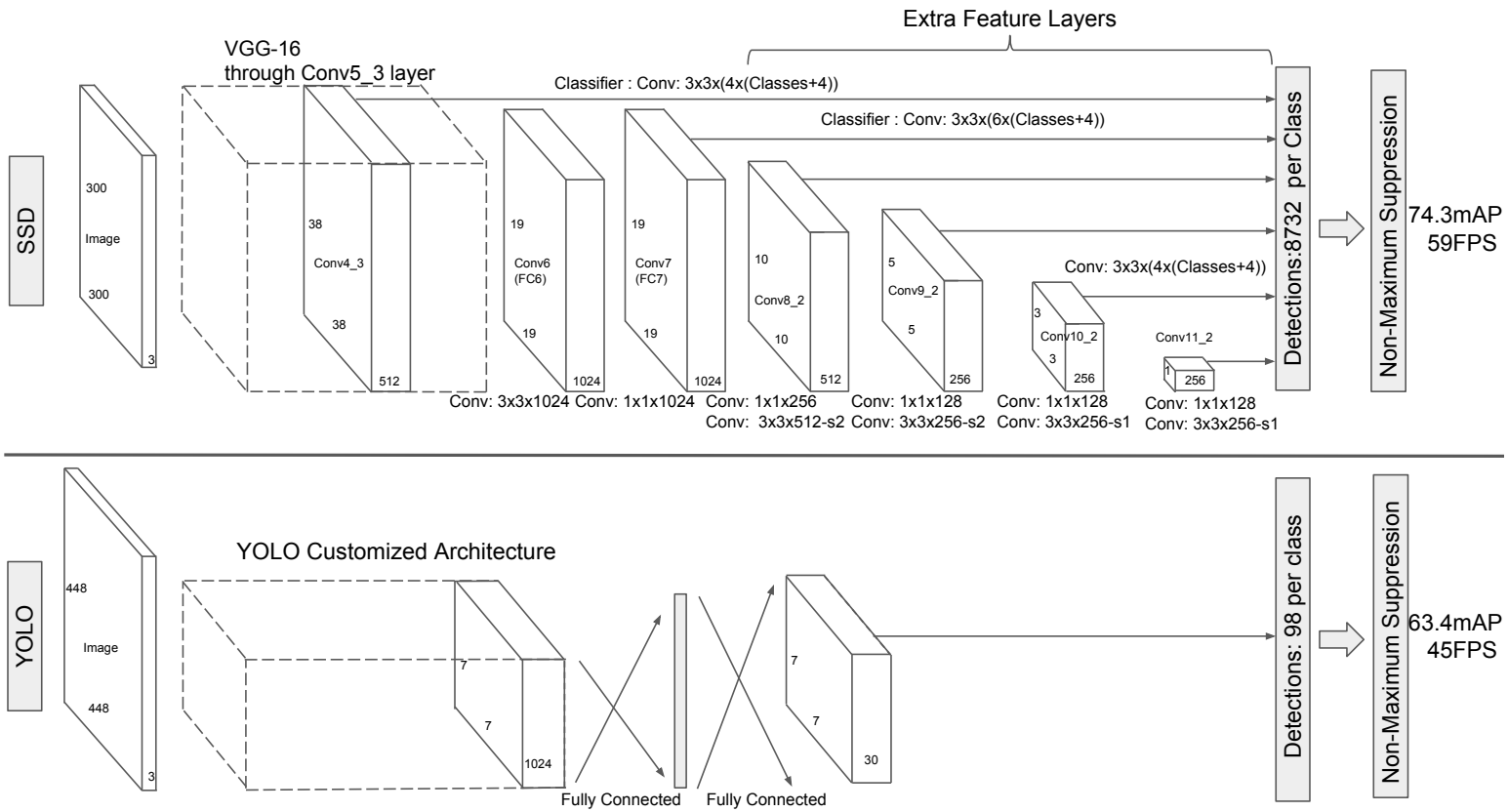
(slightly re-written from Redmon et al. 2015)

Training YOLO relies on many engineering choices that illustrate well how involved is deep-learning "in practice":

- Pre-train the 20 first convolutional layers on ImageNet classification,
- use 448×448 input for detection, instead of 224×224,
- use Leaky ReLU for all layers,
- dropout after the first fully connected layer,
- normalize bounding boxes parameters in $[0, 1]$,
- use a quadratic loss not only for the bounding box coordinates, but also for the confidence and the class scores,
- reduce the weight of large bounding boxes by using the square roots of the size in the loss,
- reduce the importance of empty cells by weighting less the confidence-related loss on them,
- use momentum 0.9, decay $5e-4$,
- data augmentation with scaling, translation, and HSV transformation.

A critical technical point is the design of the loss function that articulates both a classification and a regression objectives.

The Single Shot Multi-box Detector (SSD, Liu et al., 2015) improves upon YOLO with a fully-convolutional architectures and multi-scale maps.

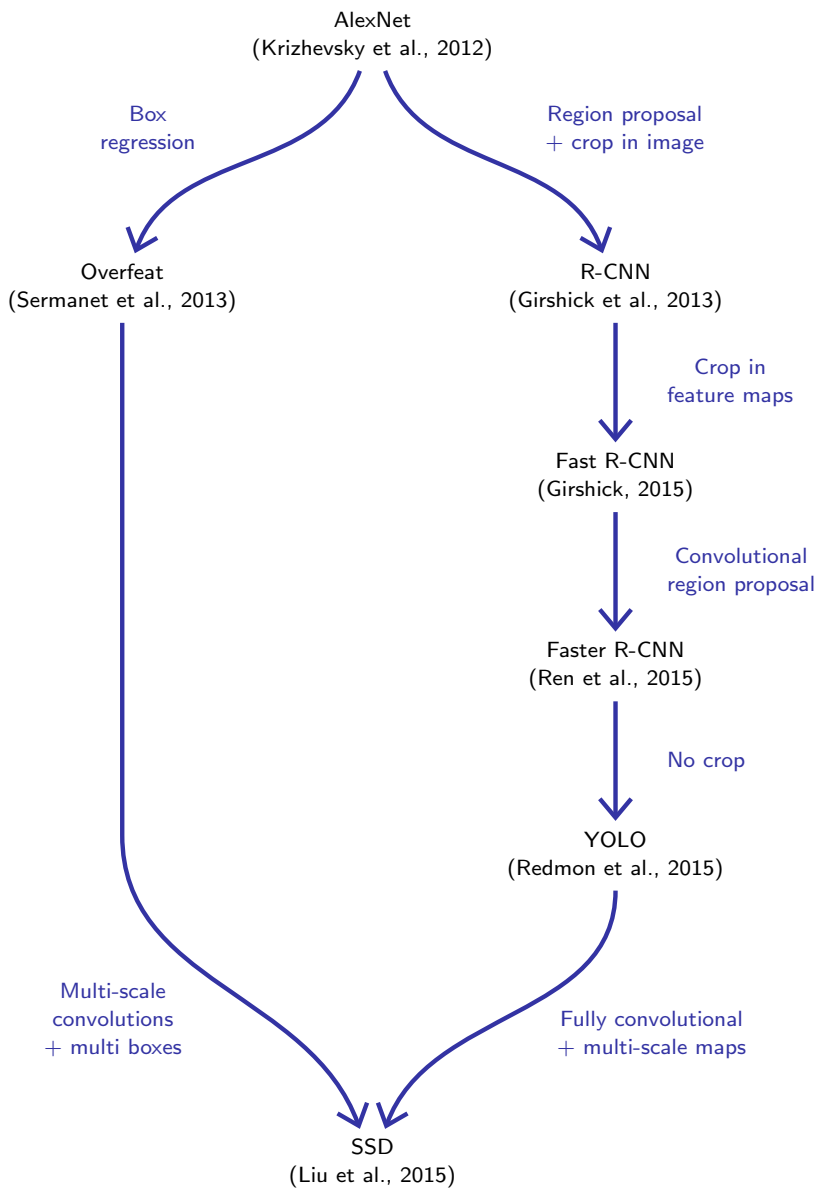


(Liu et al., 2015)

To summarize roughly how "one shot" deep detection can be achieved:

- networks trained on image classification capture localization information,
- regression layers can be attached to classification-trained networks,
- object localization does not have to be class-specific,
- multiple detection are estimated at each location to account for different aspect ratios and scales.

Object detection networks



Deep learning

8.4. Networks for semantic segmentation

François Fleuret

https://fleuret.org/dlc/

UNIVERSITÉ
DE GENÈVE

The historical approach to image segmentation was to define a measure of similarity between pixels, and to cluster groups of similar pixels. Such approaches account poorly for semantic content.

The deep-learning approach re-casts semantic segmentation as pixel classification, and re-uses networks trained for image classification by making them fully convolutional.

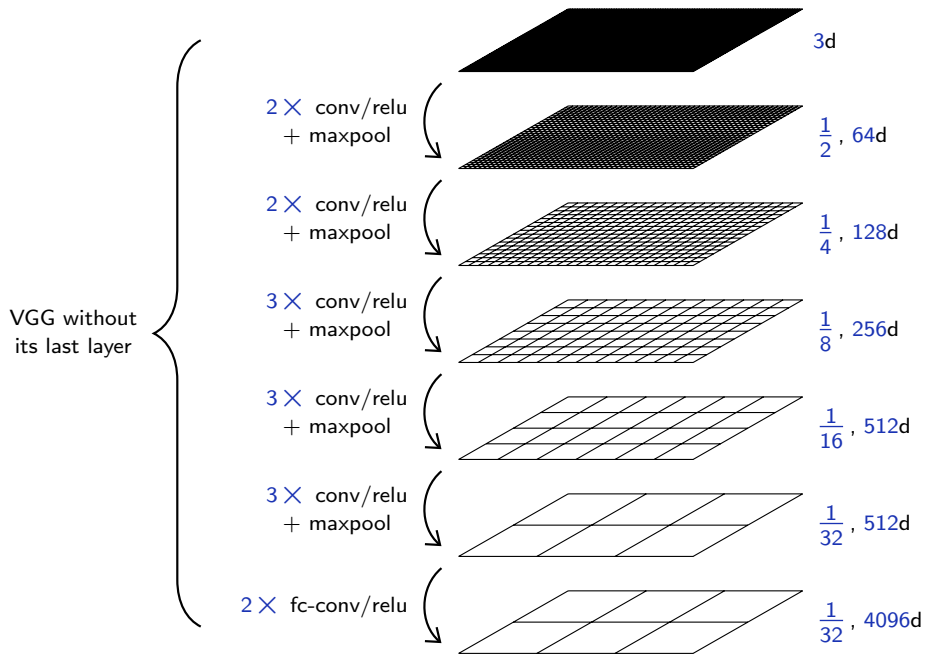
Shelhamer et al. (2016) proposed the FCN ("Fully Convolutional Network") that uses a pre-trained classification network (e.g. VGG 16 layers).

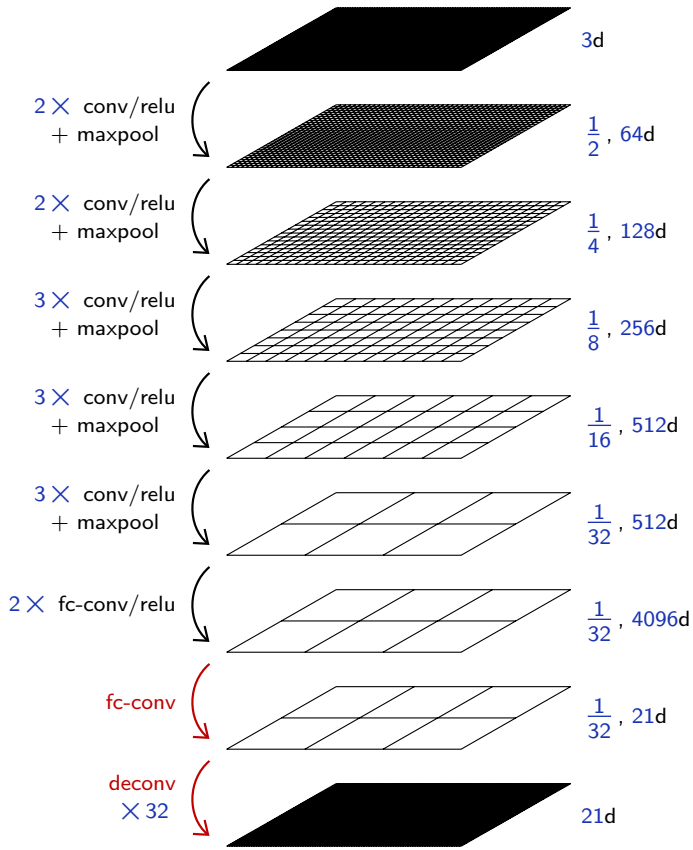
The fully connected layers are converted to 1×1 convolutional filters, and the final one retrained for 21 output channels (VOC 20 classes + "background").

Since VGG16 has 5 max-pooling with 2×2 kernels, with proper padding, the output is $1/2^5 = 1/32$ the size of the input.

This map is then up-scaled with a transposed convolution layer with kernel 64×64 and stride 32×32 to get a final map of same size as the input image.

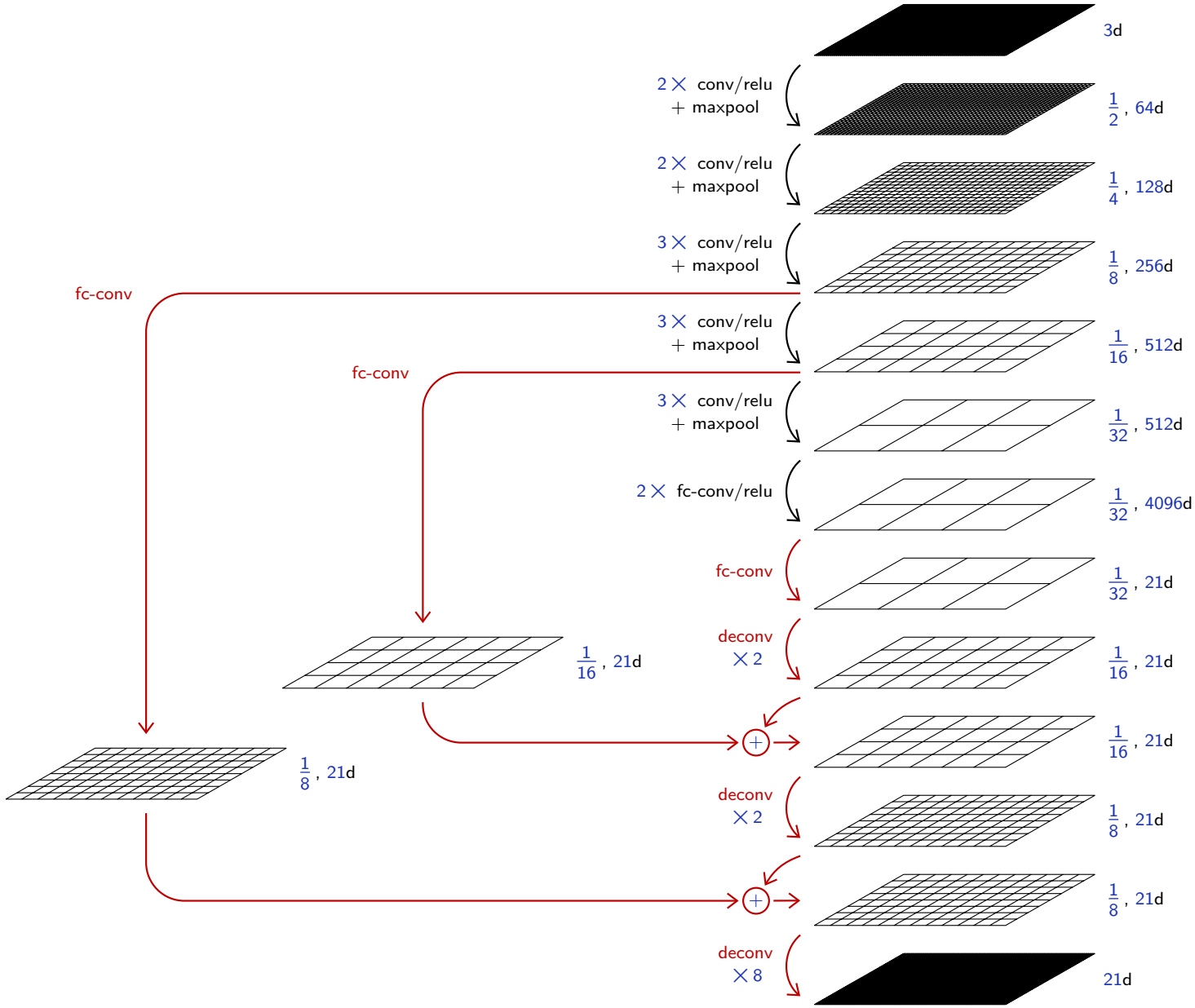
Training is achieved with full images and pixel-wise cross-entropy, starting with a pre-trained VGG16. All layers are fine-tuned, although fixing the up-scaling transposed convolution to bilinear does as well.

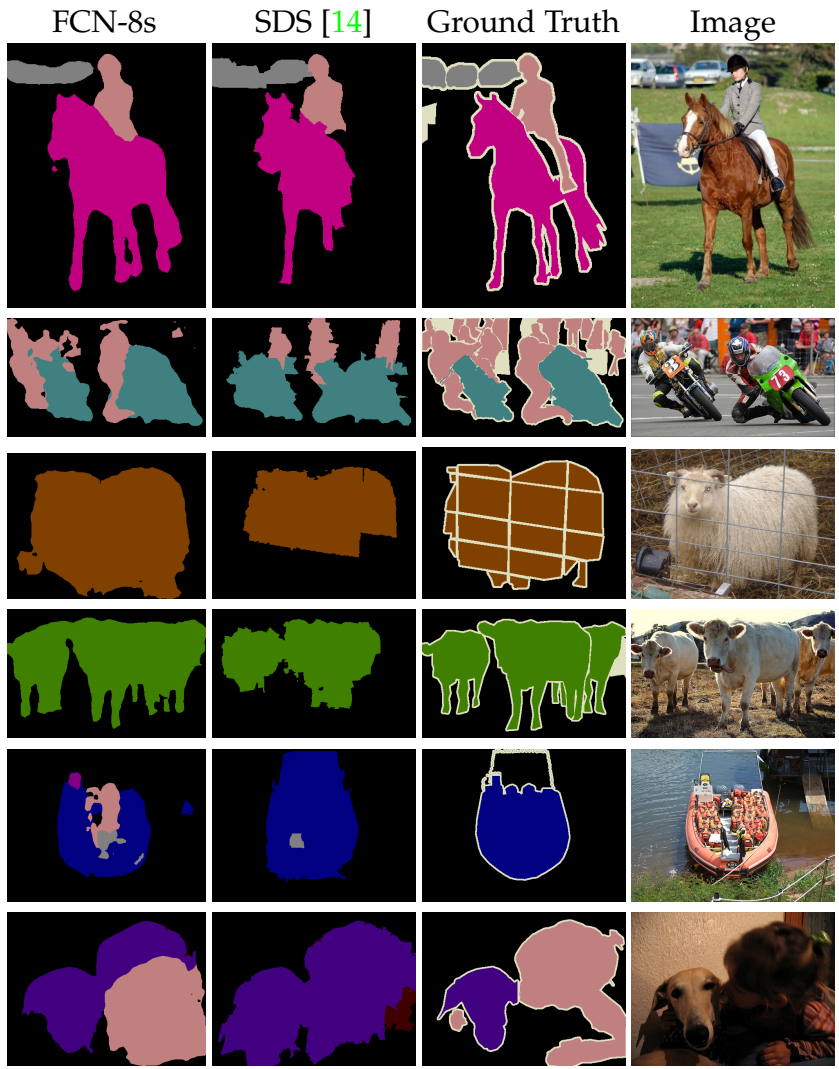

VGG without
its last layer
2 × conv/relu
+ maxpool
2 × conv/relu
+ maxpool
3 × conv/relu
+ maxpool
3 × conv/relu
+ maxpool
3 × conv/relu
+ maxpool
2 × fc-conv/relu
3d
1/2, 64d
1/4, 128d
1/8, 256d
1/16, 512d
1/32, 512d
1/32, 4096d


3d
2 × conv/relu
+ maxpool
1/2, 64d
2 × conv/relu
+ maxpool
1/4, 128d
3 × conv/relu
+ maxpool
1/8, 256d
3 × conv/relu
+ maxpool
1/16, 512d
3 × conv/relu
+ maxpool
1/32, 512d
2 × fc-conv/relu
1/32, 4096d
fc-conv
1/32, 21d
deconv
× 32
21d

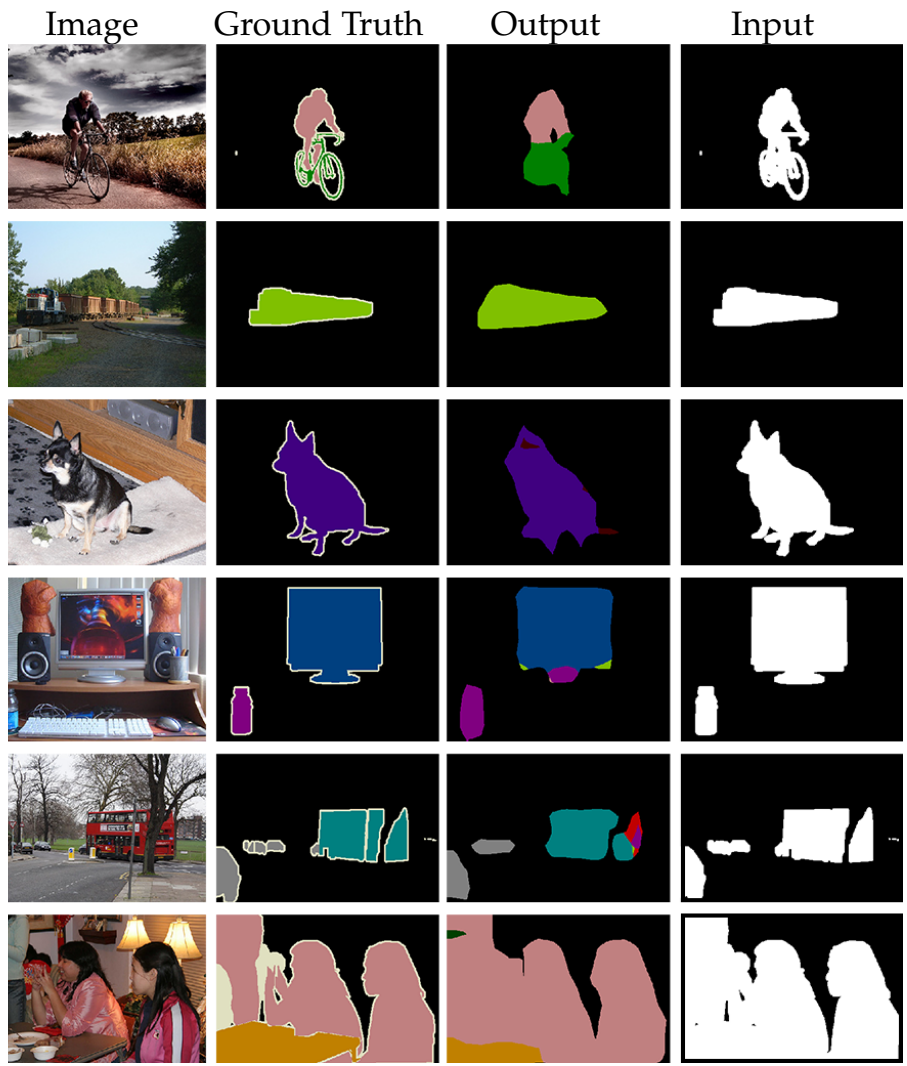
Although the FCN achieved almost state-of-the-art results when published, its main weakness is the coarseness of the signal from which the final output is produced ($1/32$ of the original resolution).

Shelhamer et al. proposed an additional element, that consists of using the same prediction/up-scaling from intermediate layers of the VGG network.


3d
2 × conv/relu + maxpool
1/2, 64d
2 × conv/relu + maxpool
1/4, 128d
3 × conv/relu + maxpool
1/8, 256d
fc-conv
3 × conv/relu + maxpool
1/16, 512d
fc-conv
3 × conv/relu + maxpool
1/32, 512d
2 × fc-conv/relu
1/32, 4096d
fc-conv
1/32, 21d
deconv × 2
1/16, 21d
1/16, 21d
1/16, 21d
1/8, 21d
deconv × 2
1/8, 21d
1/8, 21d
deconv × 8
21d



Left column is the best network from Shelhamer et al. (2016).



Results with a network trained from mask only (Shelhamer et al., 2016).

The most sophisticated object detection methods achieve **instance segmentation** and estimate a segmentation mask per detected object.

Mask R-CNN (He et al., 2017) adds a branch to the Faster R-CNN model to estimate a mask for each detected region of interest.

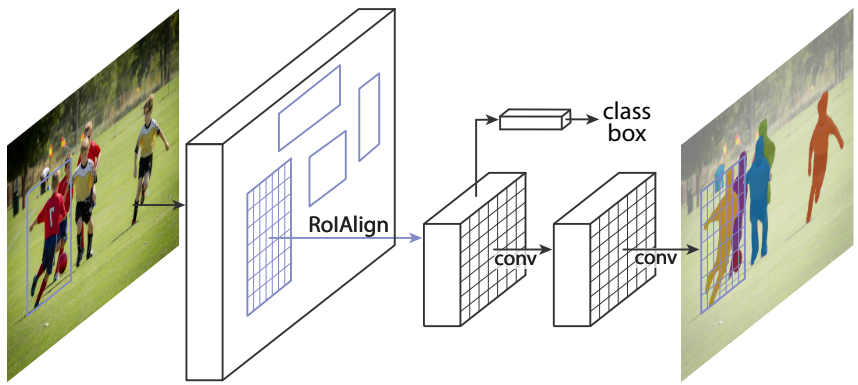


Figure 1. The **Mask R-CNN** framework for instance segmentation.

(He et al., 2017)

Figure 5. More results of **Mask R-CNN** on COCO test images, using ResNet-101-FPN and running at 5 fps, with 35.7 mask AP (Table 1).

(He et al., 2017)

It is noteworthy that for detection and semantic segmentation, there is an heavy re-use of large networks trained for classification.

The models themselves, as much as the source code of the algorithm that produced them, or the training data, are generic and re-usable assets.

Deep learning

8.5. DataLoader and neuro-surgery

François Fleuret

https://fleuret.org/dlc/

UNIVERSITÉ
DE GENÈVE

torch.utils.data.DataLoader

Until now, we have dealt with image sets that could fit in memory, and we manipulated them as regular tensors, e.g.

```
train_set = torchvision.datasets.MNIST(root = data_dir,
                                       train = True, download = True)
train_input = train_set.data.view(-1, 1, 28, 28).float()
train_targets = train_set.targets
```

However, large sets do not fit in memory, and samples have to be constantly loaded during training.

ImageNet LSVRC 2012	Images	151Gb
LSUN (all classes)	Images	1.7Tb
OSCAR	Text	6Tb

This requires a [sophisticated] machinery to parallelize the loading itself, but also the normalization, and data-augmentation operations.

PyTorch offers the `torch.utils.data.DataLoader` object which combines a **data-set** and a **sampling policy** to create an iterator over mini-batches.

Standard data-sets are available in `torchvision.datasets`, and they allow to apply transformations over the images or the labels transparently.

If needed, `torchvision.datasets.ImageFolder` creates a data-set from files located in a folder, and `torch.utils.data.TensorDataset` from a tensor. The latter is useful for synthetic toy examples or small data-sets.

```
from torch.utils.data import DataLoader
from torchvision import datasets, transforms

data_dir = os.environ.get('PYTORCH_DATA_DIR') or './data/mnist/'

train_transforms = transforms.Compose(
    [
        transforms.ToTensor(),
        transforms.Normalize(mean = (0.1302,), std = (0.3069, ))
    ]
)

train_loader = DataLoader(
    datasets.MNIST(root = data_dir, train = True, download = True,
                   transform = train_transforms),
    batch_size = 100,
    num_workers = 4,
    shuffle = True,
    pin_memory = torch.cuda.is_available()
)
```

Given this `train_loader`, we can now re-write our training procedure with a loop over the mini-batches

```
for e in range(nb_epochs):
    for input, targets in iter(train_loader):

        input, targets = input.to(device), targets.to(device)

        output = model(input)
        loss = criterion(output, targets)

        model.zero_grad()
        loss.backward()
        optimizer.step()
```

Example of neuro-surgery and fine-tuning in PyTorch

As an example of re-using a network and fine-tuning it, we will construct a network for CIFAR10 composed of:

- the first layer of an [already trained] AlexNet,
- several resnet blocks,
- a final channel-wise averaging, using `nn.AvgPool2d`, and
- a final fully connected linear layer `nn.Linear`.

During training, we will keep the AlexNet features frozen for a few epochs. This is done by setting `requires_grad` of the related `Parameter`s to `False`.

```
data_dir = os.environ.get('PYTORCH_DATA_DIR') or './data/cifar10/'

num_workers = 4
batch_size = 64

transform = torchvision.transforms.ToTensor()

train_set = datasets.CIFAR10(root = data_dir, train = True,
                             download = True, transform = transform)

train_loader = utils.data.DataLoader(train_set, batch_size = batch_size,
                                     shuffle = True, num_workers = num_workers)

test_set = datasets.CIFAR10(root = data_dir, train = False,
                            download = True, transform = transform)

test_loader = utils.data.DataLoader(test_set, batch_size = batch_size,
                                    shuffle = False, num_workers = num_workers)
```

```
class ResBlock(nn.Module):
    def __init__(self, nb_channels, kernel_size):
        super().__init__()

        self.conv1 = nn.Conv2d(nb_channels, nb_channels, kernel_size,
                               padding = (kernel_size-1)//2)
        self.bn1 = nn.BatchNorm2d(nb_channels)

        self.conv2 = nn.Conv2d(nb_channels, nb_channels, kernel_size,
                               padding = (kernel_size-1)//2)
        self.bn2 = nn.BatchNorm2d(nb_channels)

    def forward(self, x):
        y = self.bn1(self.conv1(x))
        y = F.relu(y)
        y = self.bn2(self.conv2(y))
        y += x
        y = F.relu(y)
        return y
```

```
class Monster(nn.Module):
    def __init__(self, nb_blocks, nb_channels):
        super().__init__()

        alexnet = torchvision.models.alexnet(pretrained = True)

        self.features = nn.Sequential(alexnet.features[0], nn.ReLU(inplace = True))

        dummy = self.features(torch.zeros(1, 3, 32, 32)).size()
        alexnet_nb_channels = dummy[1]
        alexnet_map_size = tuple(dummy[2:4])

        self.conv = nn.Conv2d(alexnet_nb_channels, nb_channels, kernel_size = 1)

        self.resblocks = nn.Sequential(
            *(ResBlock(nb_channels, kernel_size = 3) for _ in range(nb_blocks))
        )

        self.avg = nn.AvgPool2d(kernel_size = alexnet_map_size)
        self.fc = nn.Linear(nb_channels, 10)
```

```
def forward(self, x):
    x = self.features(x)
    x = F.relu(self.conv(x))
    x = self.resblocks(x)
    x = F.relu(self.avg(x))
    x = x.view(x.size(0), -1)
    x = self.fc(x)
    return x
```

```
nb_epochs = 50
nb_blocks, nb_channels = 8, 64

model, criterion = Monster(nb_blocks, nb_channels), nn.CrossEntropyLoss()

model.to(device)
criterion.to(device)

optimizer = torch.optim.Adam(model.parameters(), lr = 1e-2)

for e in range(nb_epochs):
    # Freeze the features during half of the epochs
    for p in model.features.parameters():
        p.requires_grad = e >= nb_epochs // 2

    acc_loss = 0.0

    for input, targets in iter(train_loader):
        input, targets = input.to(device), targets.to(device)

        output = model(input)
        loss = criterion(output, targets)
        acc_loss += loss.item()

        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

    print(e, acc_loss)
```

```
nb_test_errors, nb_test_samples = 0, 0

model.eval()

for input, targets in iter(test_loader):
    input, targets = input.to(device), targets.to(device)

    output = model(input)
    wta = torch.argmax(output.data, 1).view(-1)

    for i in range(targets.size(0)):
        nb_test_samples += 1
        if wta[i] != targets[i]: nb_test_errors += 1

test_error = 100 * nb_test_errors / nb_test_samples
print(f'test_error {test_error:.02f}% ({nb_test_errors}/{nb_test_samples})')
```

The end