EE-559 – Deep learning

8.2. Networks for image classification

François Fleuret

https://fleuret.org/ee559/

Dec 23, 2019

idiap RESEARCH INSTITUTE

EPFL

Image classification, standard convnets

The most standard networks for image classification are the LeNet family (leCun et al., 1998), and its modern extensions, among which 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 most standard networks for image classification are the LeNet family (leCun et al., 1998), and its modern extensions, among which 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.

The most standard networks for image classification are the LeNet family (leCun et al., 1998), and its modern extensions, among which 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 local complex 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()
```

`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.

`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.

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.

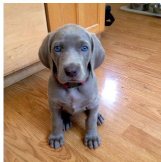
To be sure this picture did not appear in the training data, it was not taken from the web.

```
import PIL, torch, torchvision

# Imagenet class names
class_names = eval(open('imagenet1000_clsid_to_human.txt', 'r').read())

# Load and normalize the image
to_tensor = torchvision.transforms.ToTensor()
img = to_tensor(PIL.Image.open('example_images/blacklab.jpg'))
img = img.view(1, img.size(0), img.size(1), img.size(2))
img = 0.5 + 0.5 * (img - img.mean()) / img.std()
```

```
import PIL, torch, torchvision

# Imagenet class names
class_names = eval(open('imagenet1000_clsid_to_human.txt', 'r').read())

# Load and normalize the image
to_tensor = torchvision.transforms.ToTensor()
img = to_tensor(PIL.Image.open('example_images/blacklab.jpg'))
img = img.view(1, img.size(0), img.size(1), img.size(2))
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)

for k in range(15):
    print('%.02f' % scores[k].item(), 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
```

```
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
```

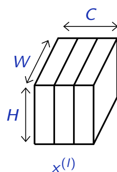


Weimaraner

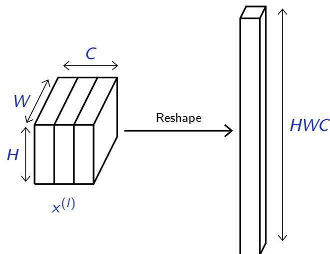
Chesapeake Bay retriever

Fully convolutional networks

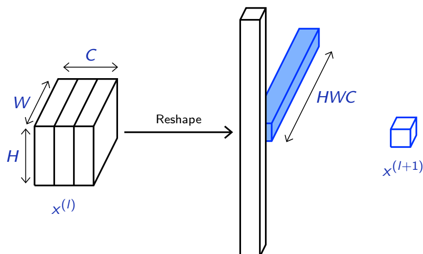
In many applications, standard convolutional networks are made **fully convolutional** by converting their fully connected layers to convolutional ones.



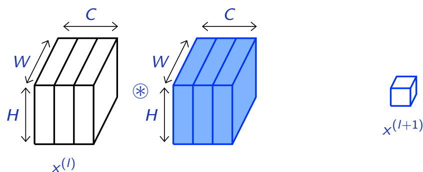
In many applications, standard convolutional networks are made **fully convolutional** by converting their fully connected layers to convolutional ones.



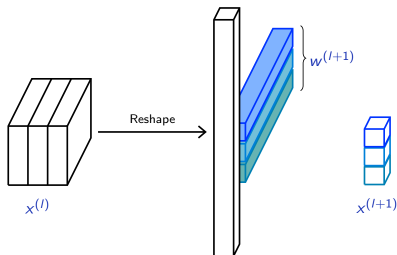
In many applications, standard convolutional networks are made **fully convolutional** by converting their fully connected layers to convolutional ones.



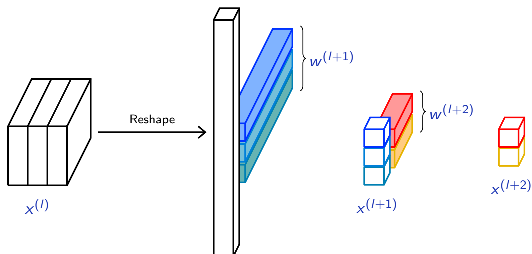
In many applications, standard convolutional networks are made **fully convolutional** by converting their fully connected layers to convolutional ones.



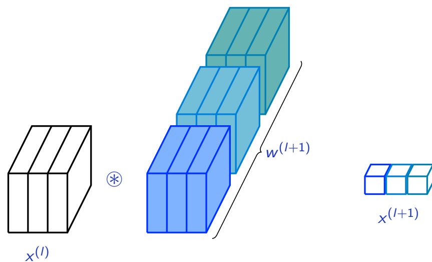
In particular multiple 1×1 convolutions can be interpreted as computing a fully-connected layer at every location of an activation map.



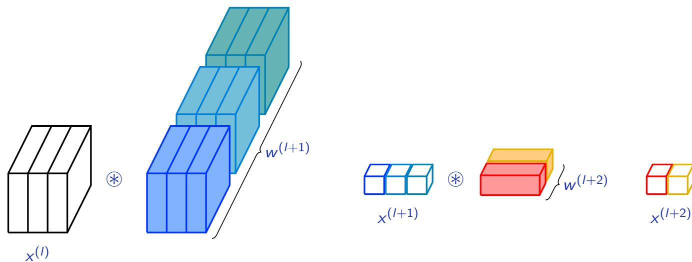
In particular multiple 1×1 convolutions can be interpreted as computing a fully-connected layer at every location of an activation map.



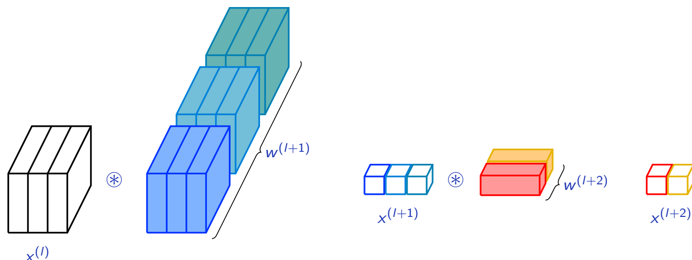
In particular multiple 1×1 convolutions can be interpreted as computing a fully-connected layer at every location of an activation map.



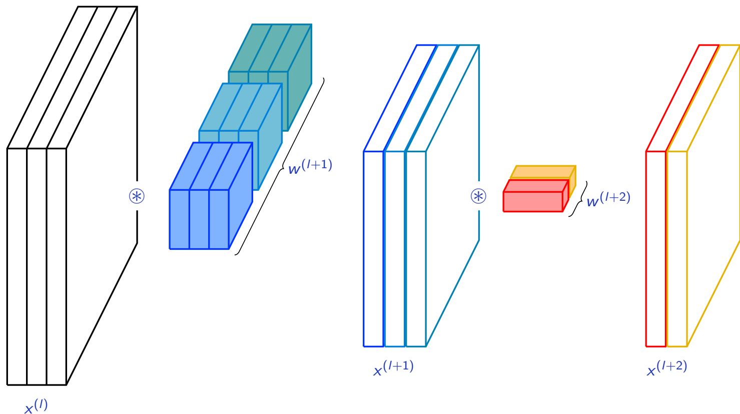
In particular multiple 1×1 convolutions can be interpreted as computing a fully-connected layer at every location of an activation map.



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.

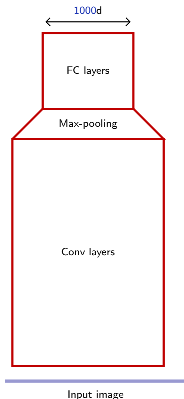
To apply this to AlexNet

```
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)
  )
)
```

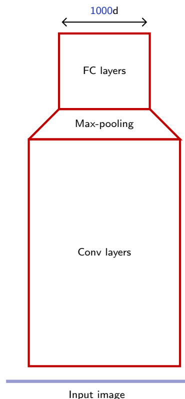
```
Sequential (
  (0): Conv2d(3, 64, kernel_size=(11, 11), stride=(4, 4), padding=(2, 2))
  (1): ReLU (inplace)
  (2): MaxPool2d (size=(3, 3), stride=(2, 2), dilation=(1, 1))
  (3): Conv2d(64, 192, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2))
  (4): ReLU (inplace)
  (5): MaxPool2d (size=(3, 3), 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.



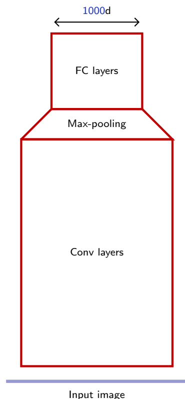
AlexNet random cropping

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

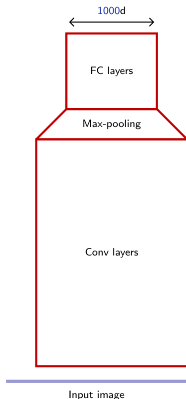


AlexNet random cropping

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

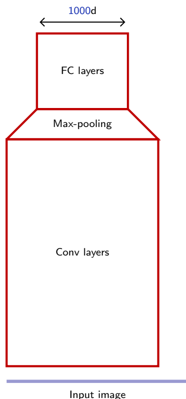


In their “overfeat” approach, Sermanet et al. (2013) combined this with a stride 1 final max-pooling to get multiple predictions.



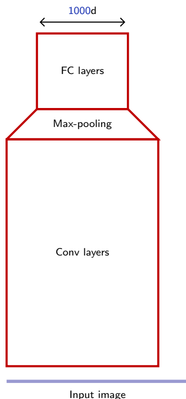
AlexNet random cropping

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

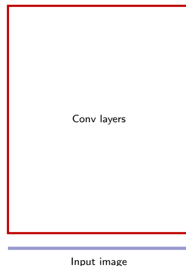


AlexNet random cropping

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

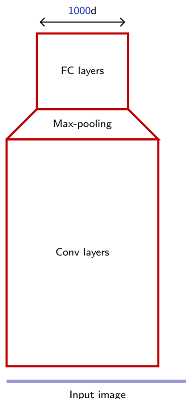


AlexNet random cropping

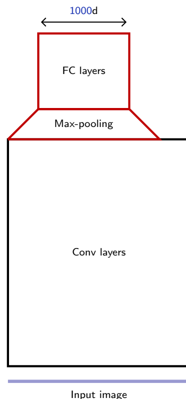


Overfeat dense max-pooling

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

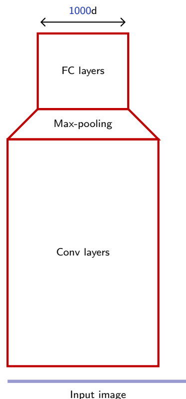


AlexNet random cropping

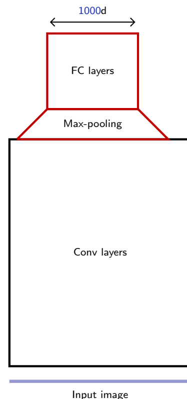


Overfeat dense max-pooling

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

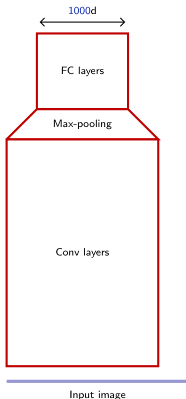


AlexNet random cropping

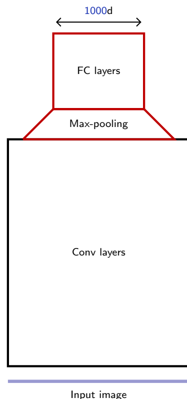


Overfeat dense max-pooling

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

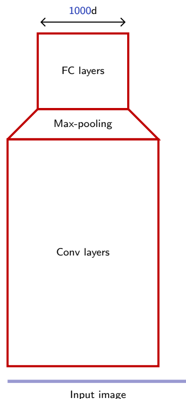


AlexNet random cropping

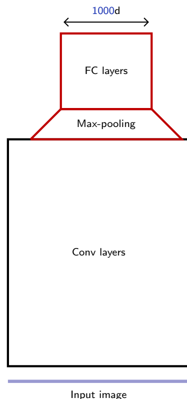


Overfeat dense max-pooling

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

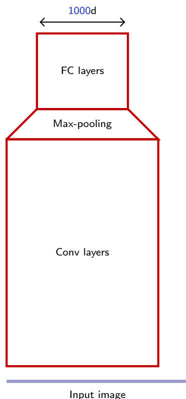


AlexNet random cropping

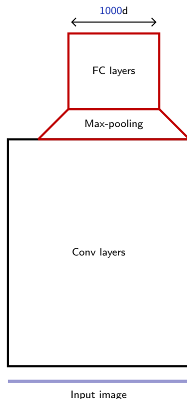


Overfeat dense max-pooling

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

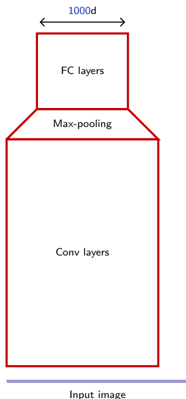


AlexNet random cropping

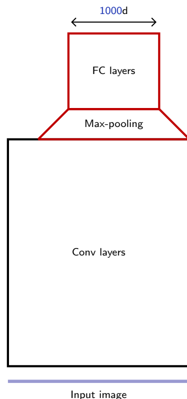


Overfeat dense max-pooling

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.

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

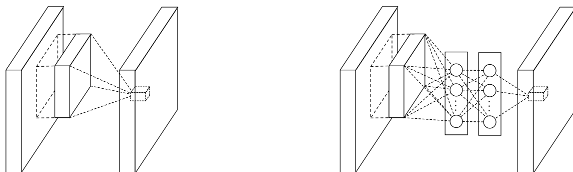
Also, and maybe more importantly, it blurs the conceptual boundary between "features" and "classifier" and leads to an intuitive understanding of convnet activations as gradually transitioning from appearance to semantic.

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).

Image classification, network in network

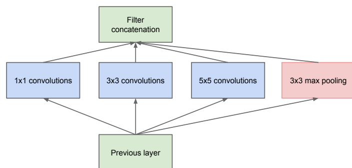
Lin et al. (2013) re-interpreted a convolution filter as a one-layer perceptron, and extended it with an "MLP convolution" (aka "network in network") to improve the capacity vs. parameter ratio.



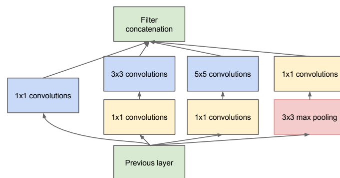
(Lin et al., 2013)

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



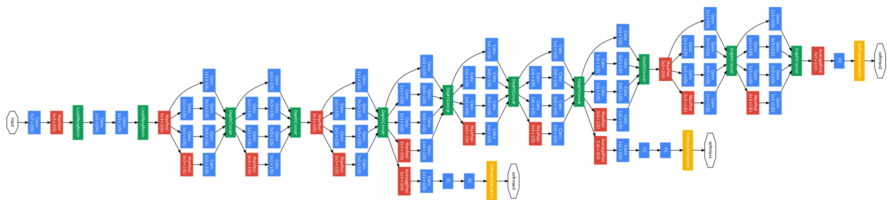
(b) Inception module with dimension reductions

(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).



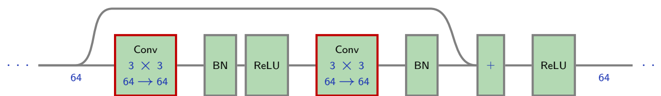
(Szegedy et al., 2015)

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).

Image classification, residual networks

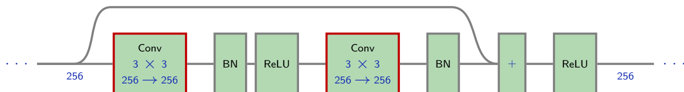
We already saw the structure of the residual networks and how well they perform on CIFAR10 (He et al., 2015).

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



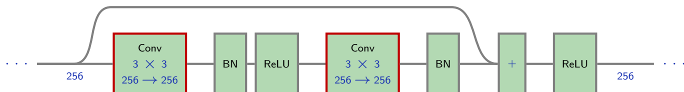
and as such requires $2 \times (3 \times 3 \times 64 + 1) \times 64 \simeq 73k$ parameters.

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



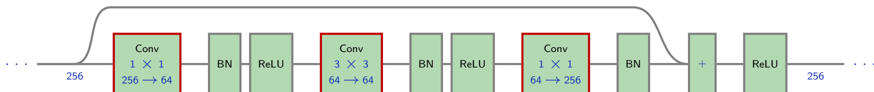
However, such a block requires $2 \times (3 \times 3 \times 256 + 1) \times 256 \simeq 1.2m$ parameters.

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



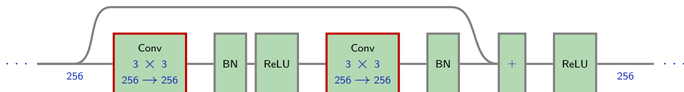
However, such a block requires $2 \times (3 \times 3 \times 256 + 1) \times 256 \simeq 1.2m$ parameters.

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



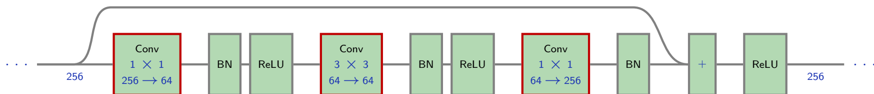
$$256 \times 64 + (3 \times 3 \times 64 + 1) \times 64 + 64 \times 256 \simeq 70k \text{ parameters.}$$

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.

layer name	output size	18-layer	34-layer	50-layer	101-layer	152-layer
conv1	112×112	7×7, 64, stride 2				
conv2_x	56×56	3×3 max pool, stride 2				
		$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$
conv3_x	28×28	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times8$
conv4_x	14×14	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times23$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times36$
conv5_x	7×7	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$
	1×1	average pool, 1000-d fc, softmax				
FLOPs		1.8×10^9	3.6×10^9	3.8×10^9	7.6×10^9	11.3×10^9

Table 1. Architectures for ImageNet. Building blocks are shown in brackets (see also Fig. 5), with the numbers of blocks stacked. Downsampling is performed by conv3_1, conv4_1, and conv5_1 with a stride of 2.

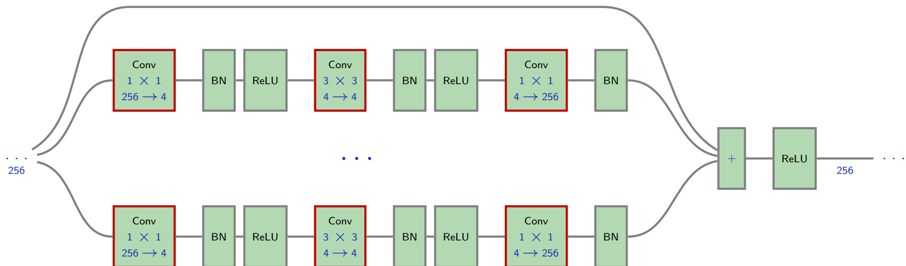
(He et al., 2015)

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**

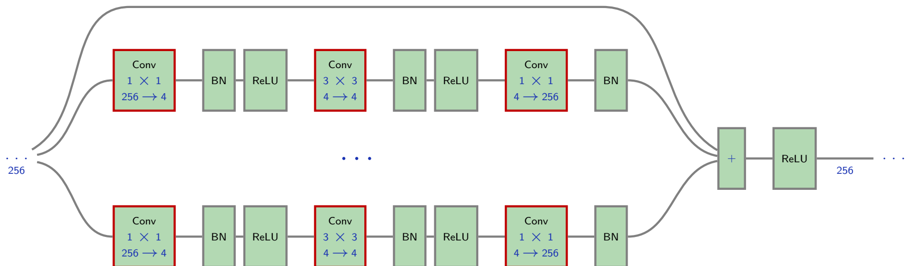
Table 5. Error rates (%) of **ensembles**. The top-5 error is on the test set of ImageNet and reported by the test server.

(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.



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.

Image classification, 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,

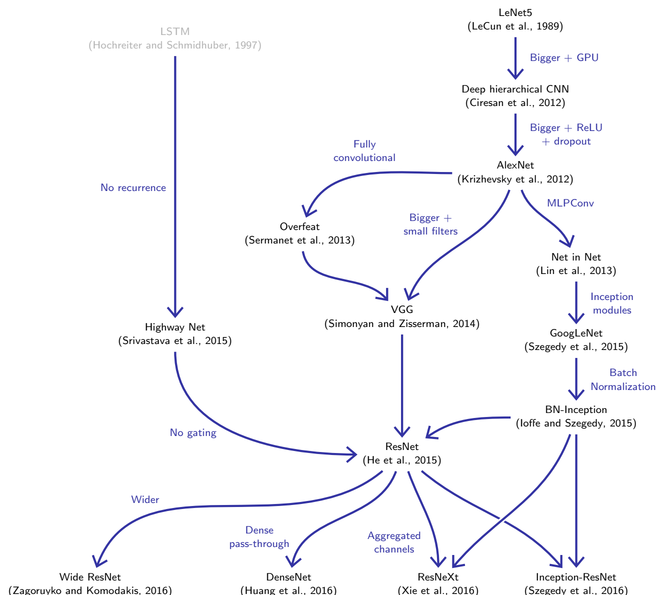
To summarize roughly the evolution of convnets for image classification:

- 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,

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



The end

References

D. Ciresan, U. Meier, and J. Schmidhuber. **Multi-column deep neural networks for image classification**. CoRR, abs/1202.2745, 2012.

K. He, X. Zhang, S. Ren, and J. Sun. **Deep residual learning for image recognition**. CoRR, abs/1512.03385, 2015.

S. Hochreiter and J. Schmidhuber. **Long short-term memory**. Neural Computation, 9(8): 1735–1780, 1997.

G. Huang, Z. Liu, K. Weinberger, and L. van der Maaten. **Densely connected convolutional networks**. CoRR, abs/1608.06993, 2016.

S. Ioffe and C. Szegedy. **Batch normalization: Accelerating deep network training by reducing internal covariate shift**. In International Conference on Machine Learning (ICML), 2015.

A. Krizhevsky, I. Sutskever, and G. Hinton. **Imagenet classification with deep convolutional neural networks**. In Neural Information Processing Systems (NIPS), 2012.

Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel. **Backpropagation applied to handwritten zip code recognition**. Neural Computation, 1(4):541–551, 1989.

Y. leCun, L. Bottou, Y. Bengio, and P. Haffner. **Gradient-based learning applied to document recognition**. Proceedings of the IEEE, 86(11):2278–2324, 1998.

M. Lin, Q. Chen, and S. Yan. **Network in network**. CoRR, abs/1312.4400, 2013.

P. Sermanet, D. Eigen, X. Zhang, M. Mathieu, R. Fergus, and Y. LeCun. **Overfeat: Integrated recognition, localization and detection using convolutional networks**. CoRR, abs/1312.6229, 2013.

K. Simonyan and A. Zisserman. **Very deep convolutional networks for large-scale image recognition**. CoRR, abs/1409.1556, 2014.

R. Srivastava, K. Greff, and J. Schmidhuber. **Highway networks**. CoRR, abs/1505.00387, 2015.

C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich. **Going deeper with convolutions**. In Conference on Computer Vision and Pattern Recognition (CVPR), 2015.

C. Szegedy, S. Ioffe, and V. Vanhoucke. **Inception-v4, inception-resnet and the impact of residual connections on learning**. CoRR, abs/1602.07261, 2016.

S. Xie, R. Girshick, P. Dollár, Z. Tu, and K. He. **Aggregated residual transformations for deep neural networks**. CoRR, abs/1611.05431.pdf, 2016.

S. Zagoruyko and N. Komodakis. **Wide residual networks**. CoRR, abs/1605.07146, 2016.