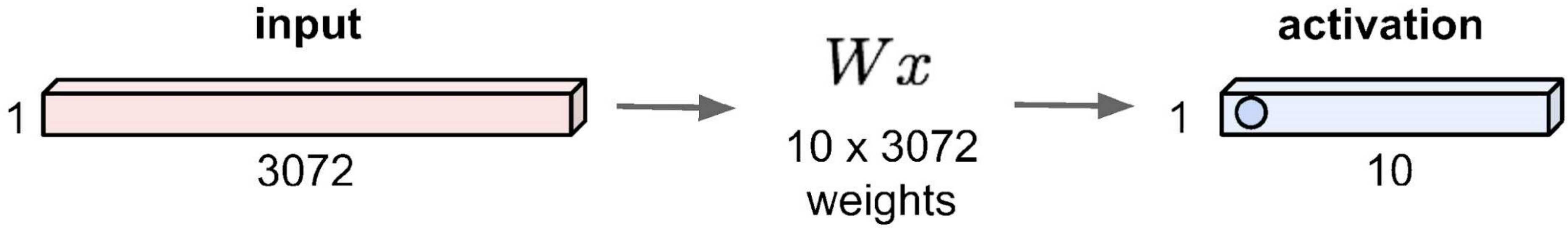
Convolutional Neural Networks

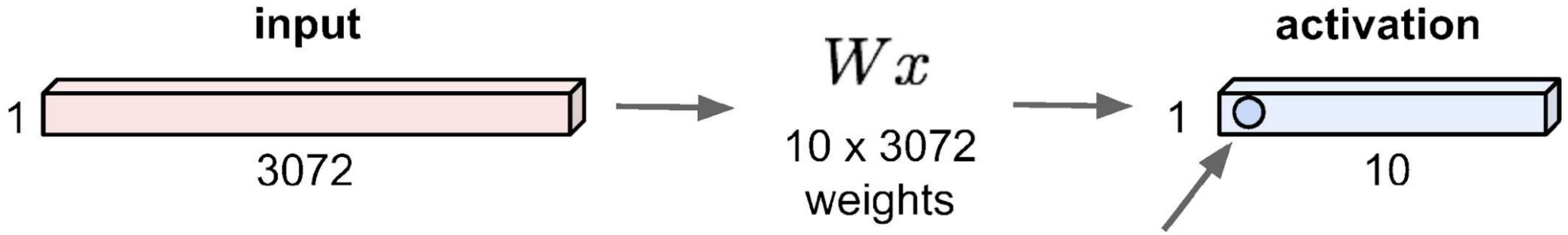
Fully Connected Layer

32x32x3 image -> stretch to 3072 x 1



Fully Connected Layer

32x32x3 image -> stretch to 3072 x 1



input

1

3072

Wx

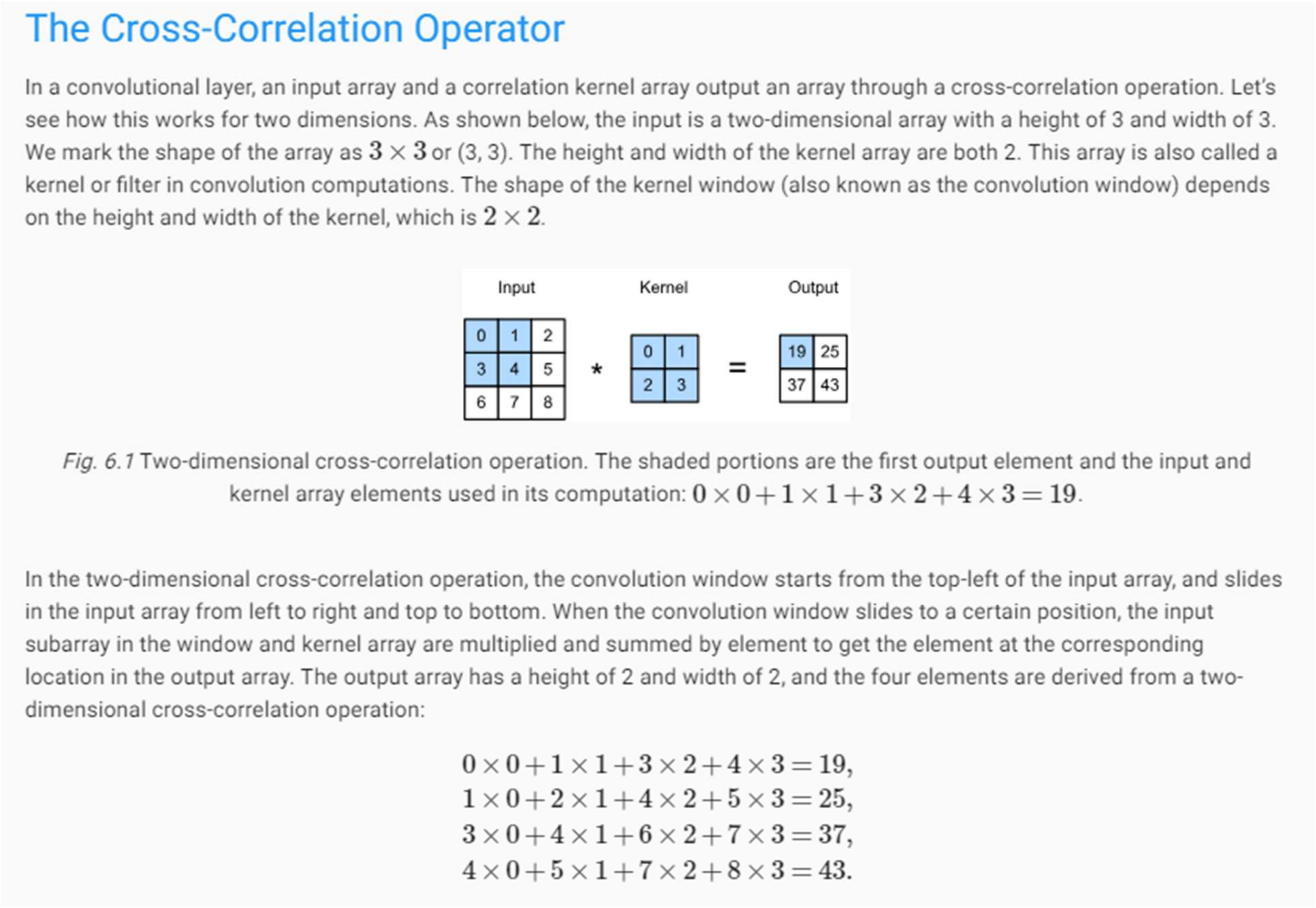
10 x 3072
weights

activation

1

10

1 number:
the result of taking a dot product
between a row of W and the input
(a 3072-dimensional dot product)



The Cross-Correlation Operator

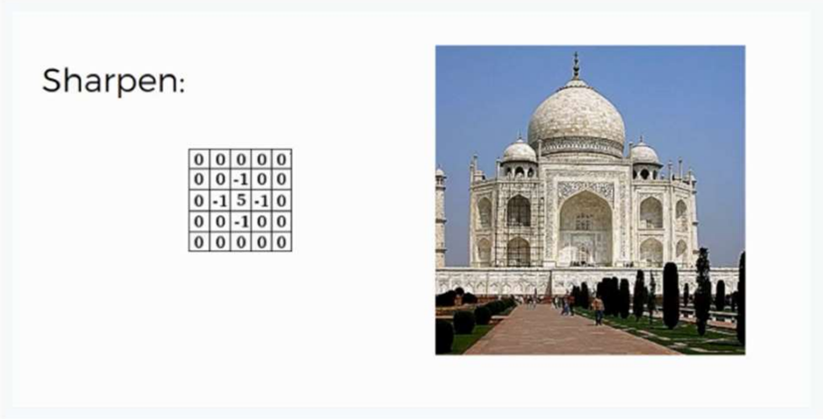
In a convolutional layer, an input array and a correlation kernel array output an array through a cross-correlation operation. Let's see how this works for two dimensions. As shown below, the input is a two-dimensional array with a height of 3 and width of 3. We mark the shape of the array as 3×3 or (3, 3). The height and width of the kernel array are both 2. This array is also called a kernel or filter in convolution computations. The shape of the kernel window (also known as the convolution window) depends on the height and width of the kernel, which is 2×2.

Fig. 6.1 Two-dimensional cross-correlation operation. The shaded portions are the first output element and the input and kernel array elements used in its computation: $0 \times 0 + 1 \times 1 + 3 \times 2 + 4 \times 3 = 19$.

In the two-dimensional cross-correlation operation, the convolution window starts from the top-left of the input array, and slides in the input array from left to right and top to bottom. When the convolution window slides to a certain position, the input subarray in the window and kernel array are multiplied and summed by element to get the element at the corresponding location in the output array. The output array has a height of 2 and width of 2, and the four elements are derived from a two-dimensional cross-correlation operation:

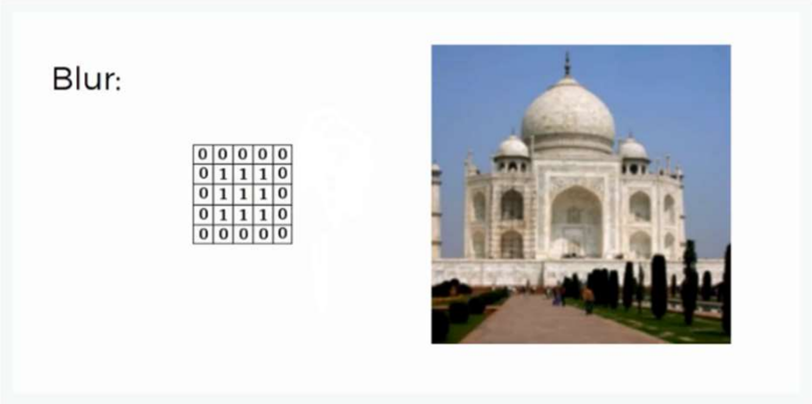
$$\begin{aligned}
0 \times 0 + 1 \times 1 + 3 \times 2 + 4 \times 3 = 19,\\
1 \times 0 + 2 \times 1 + 4 \times 2 + 5 \times 3 = 25,\\
3 \times 0 + 4 \times 1 + 6 \times 2 + 7 \times 3 = 37,\\
4 \times 0 + 5 \times 1 + 7 \times 2 + 8 \times 3 = 43.
\end{aligned}$$

Filters (kernels, convolutions)



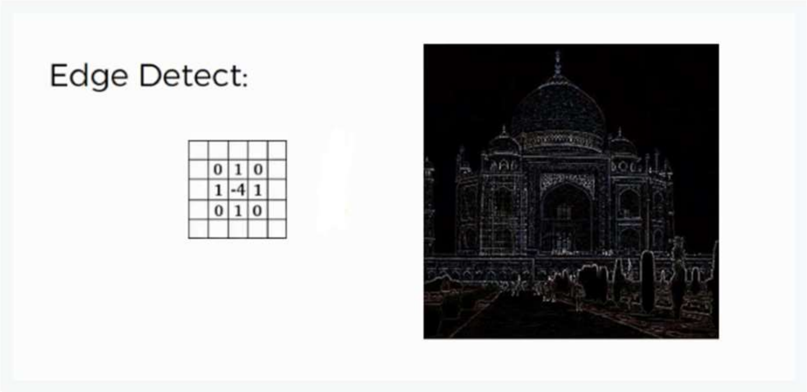
Sharpen:

0	0	0	0	0
0	0	-1	0	0
0	-1	5	-1	0
0	0	-1	0	0
0	0	0	0	0



Blur:

0	0	0	0	0
0	1	1	1	0
0	1	1	1	0
0	1	1	1	0
0	0	0	0	0



Edge Detect:

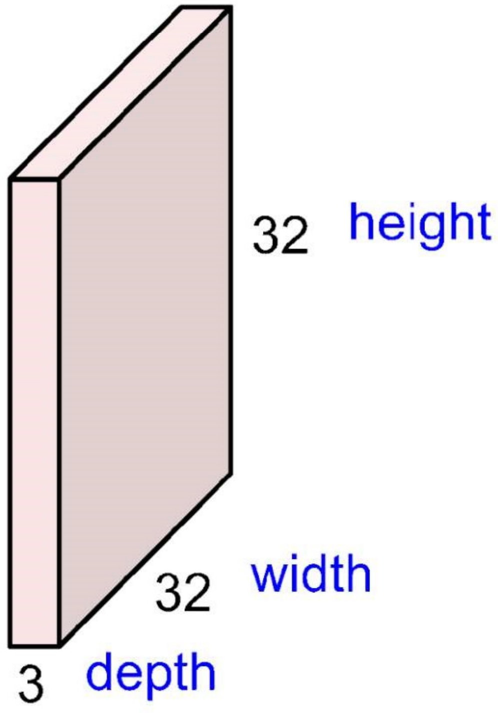
	0	1	0	
	1	-4	1	
	0	1	0	

Inverse approach:

what-if we learned the filters weights?

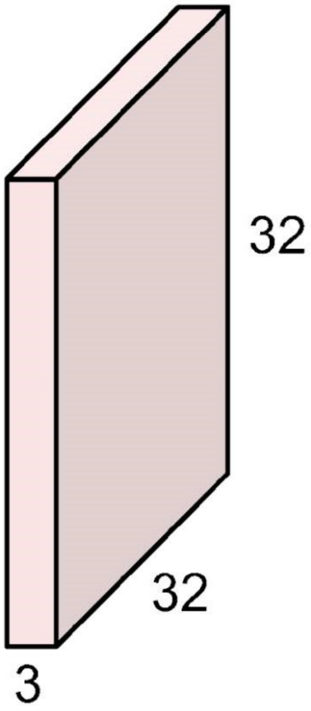
Convolution Layer

32x32x3 image -> preserve spatial structure

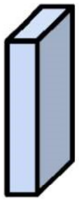


Convolution Layer

32x32x3 image



5x5x3 filter

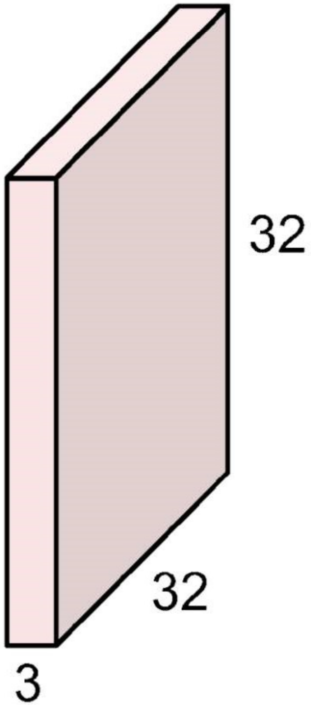


Convolve the filter with the image i.e. “slide over the image spatially, computing dot products”

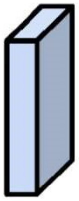
Convolution Layer

Filters always extend the full depth of the input volume

32x32x3 image

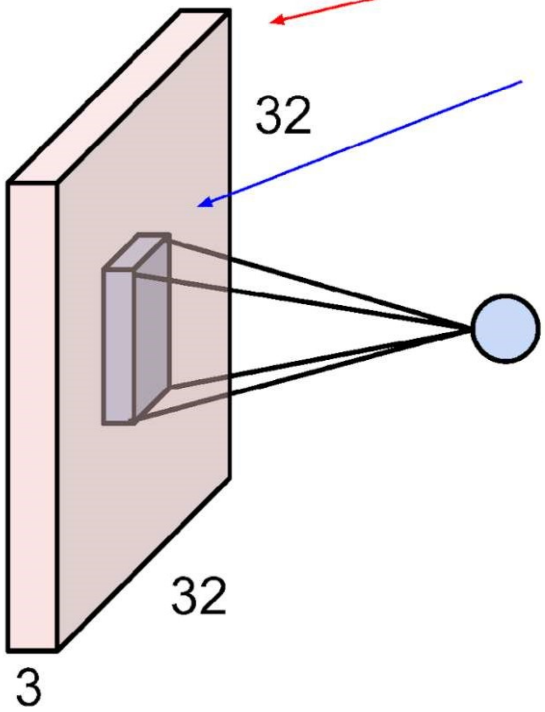


5x5x3 filter



Convolve the filter with the image i.e. “slide over the image spatially, computing dot products”

Convolution Layer



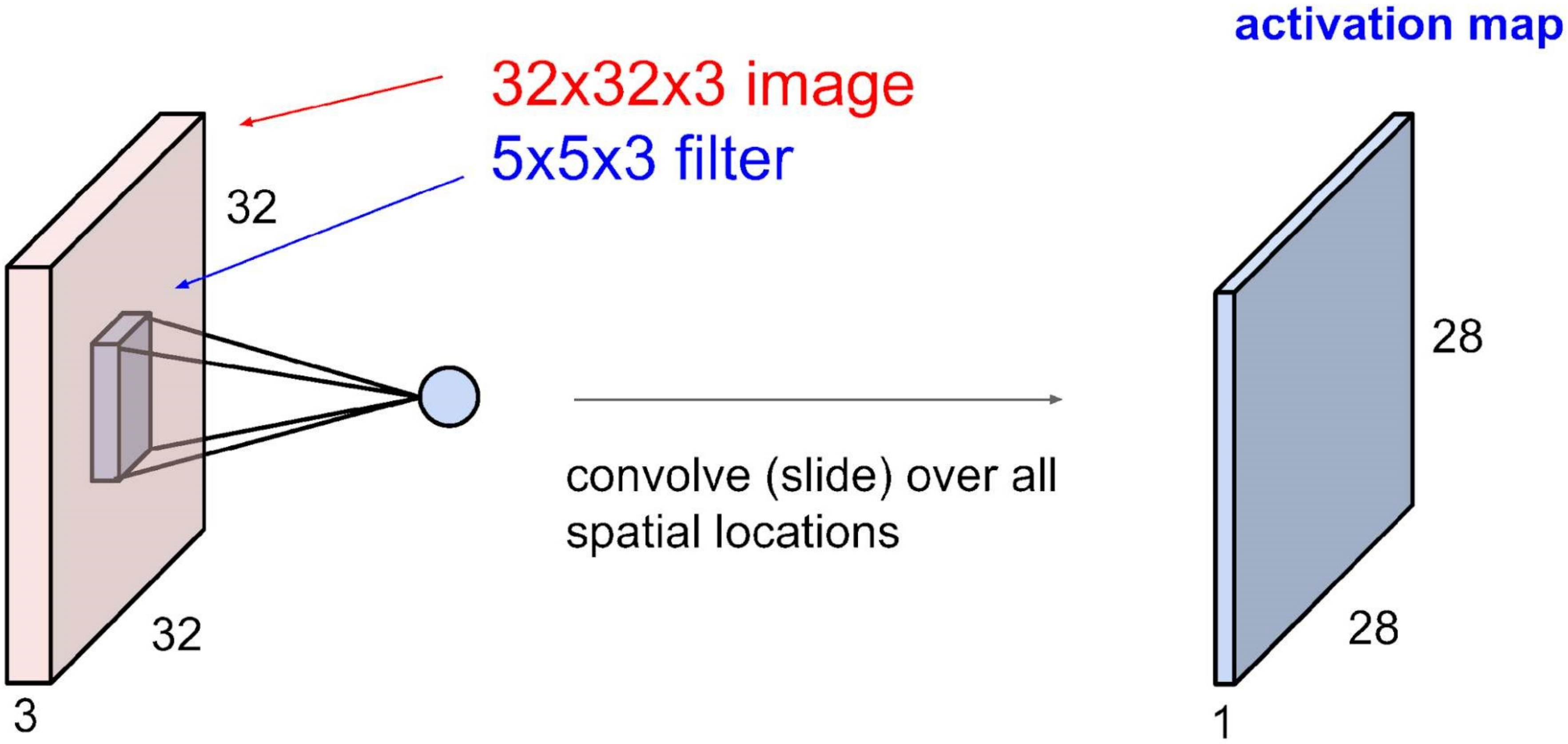
32x32x3 image

5x5x3 filter w

1 number:
the result of taking a dot product between the filter and a small 5x5x3 chunk of the image (i.e. 5*5*3 = 75-dimensional dot product + bias)

$$w^T x + b$$

Convolution Layer



32x32x3 image

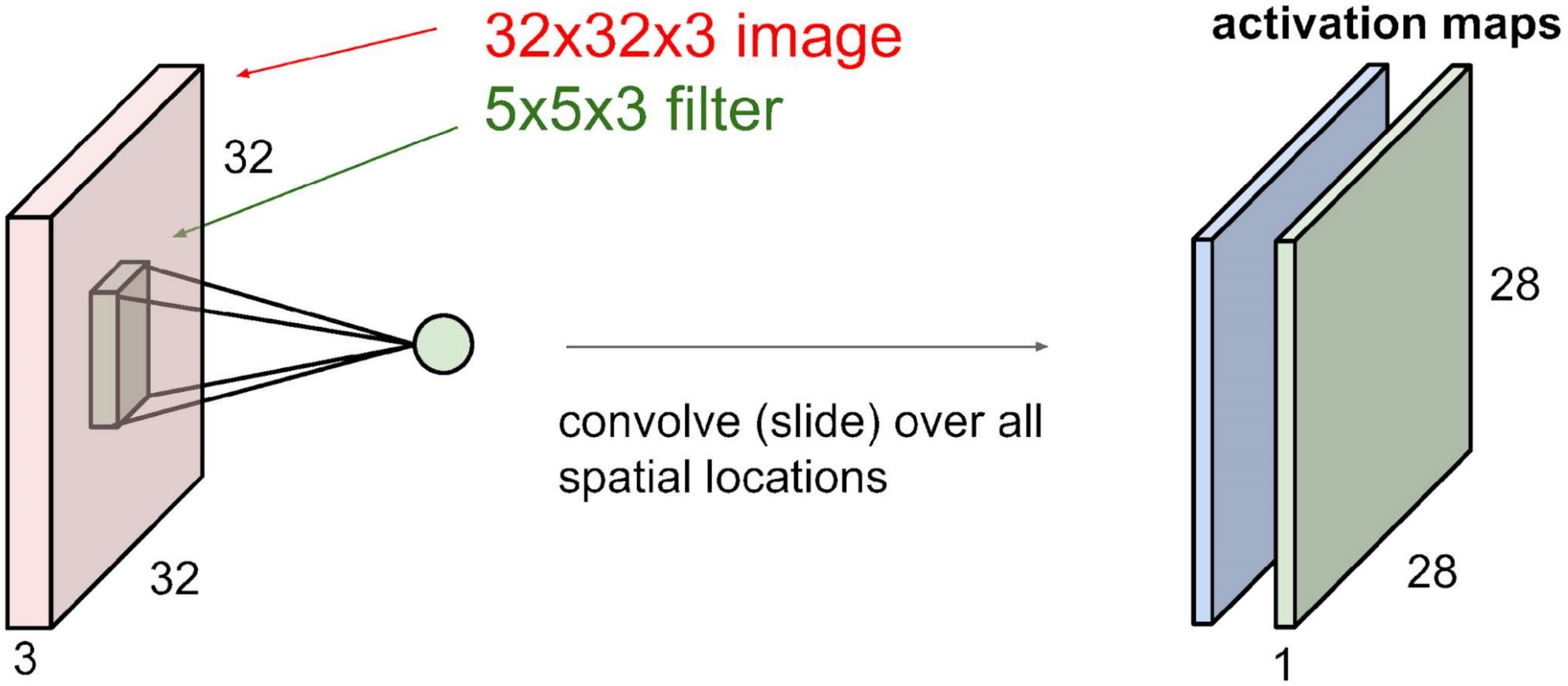
5x5x3 filter

convolve (slide) over all spatial locations

activation map

Convolution Layer

consider a second, green filter



32x32x3 image

5x5x3 filter

32

32

3

convolve (slide) over all spatial locations

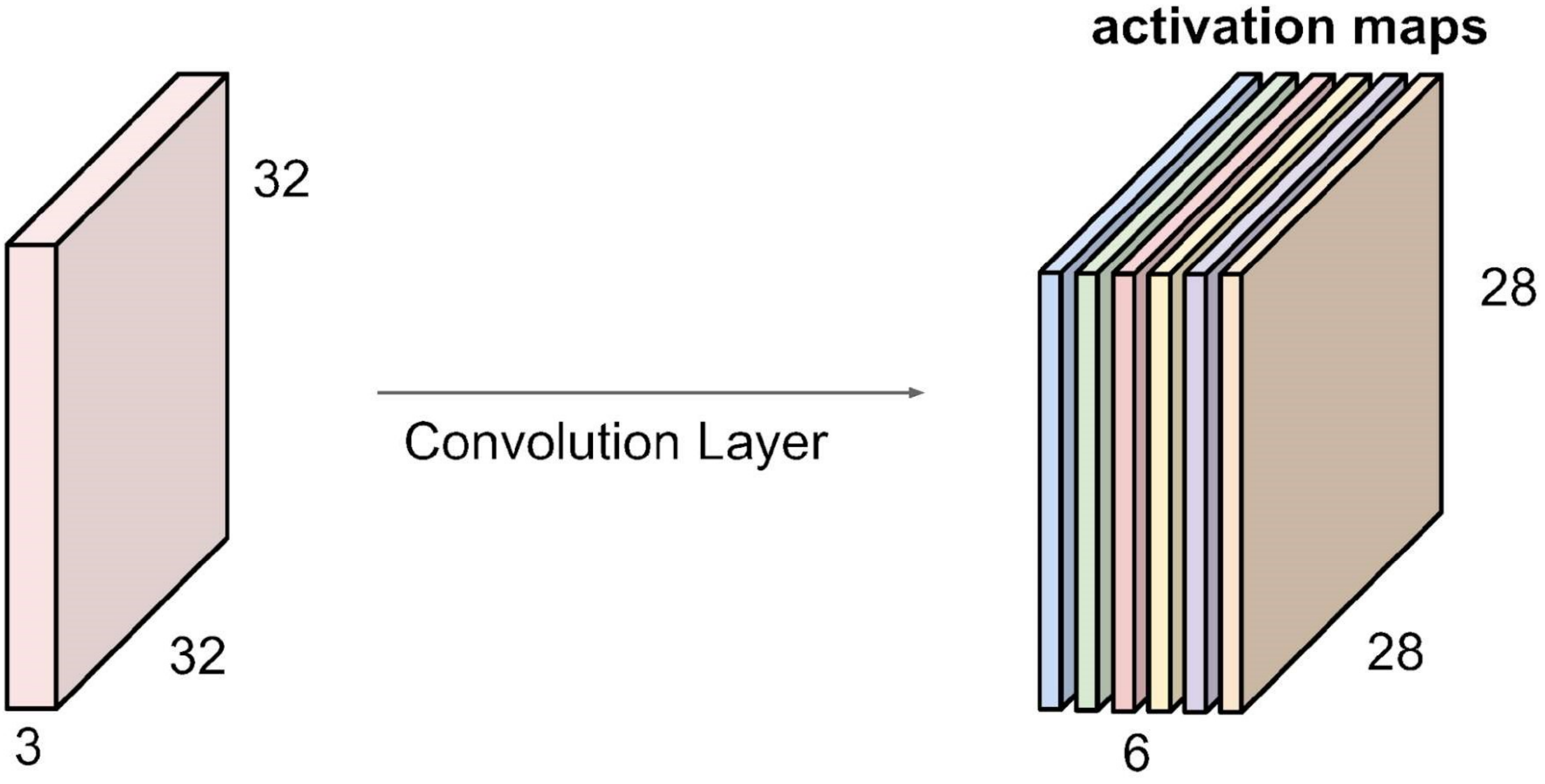
activation maps

28

28

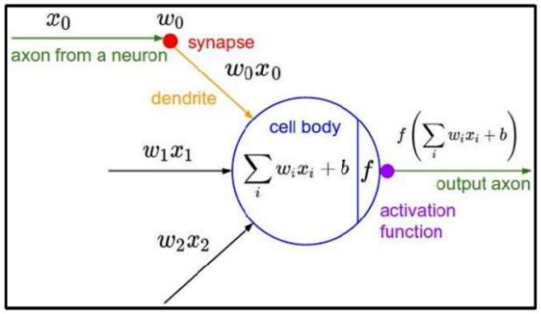
1

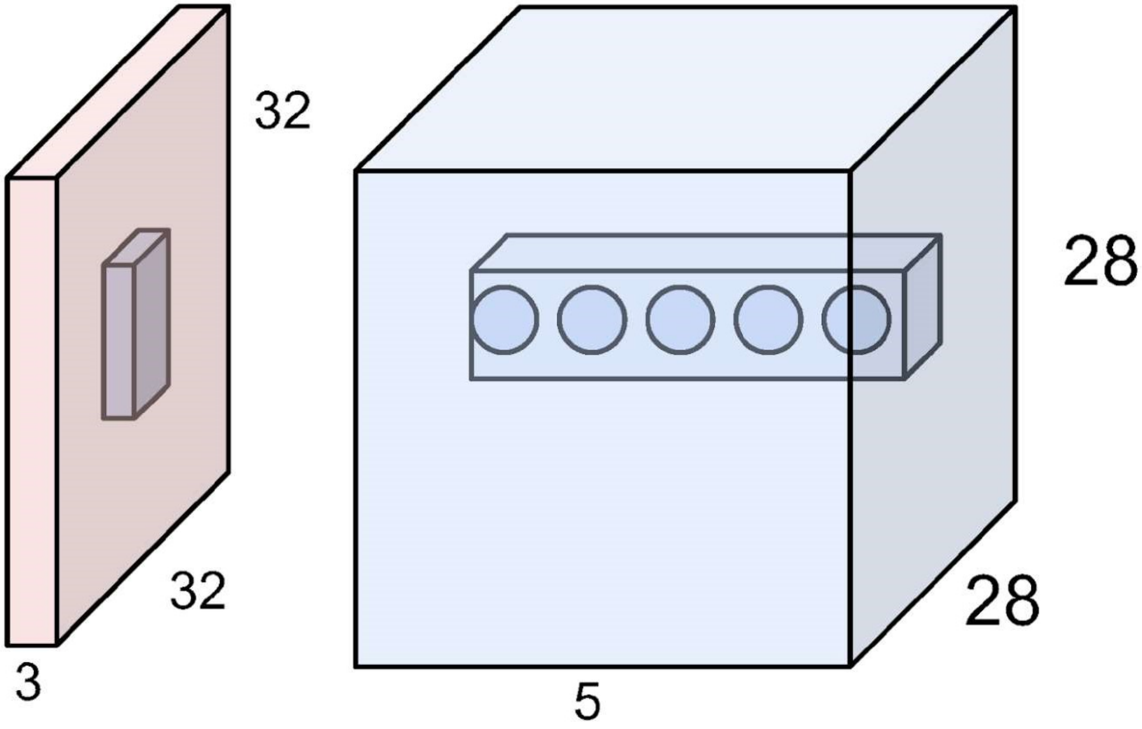
For example, if we had 6 5x5 filters, we'll get 6 separate activation maps:



We stack these up to get a "new image" of size 28x28x6!

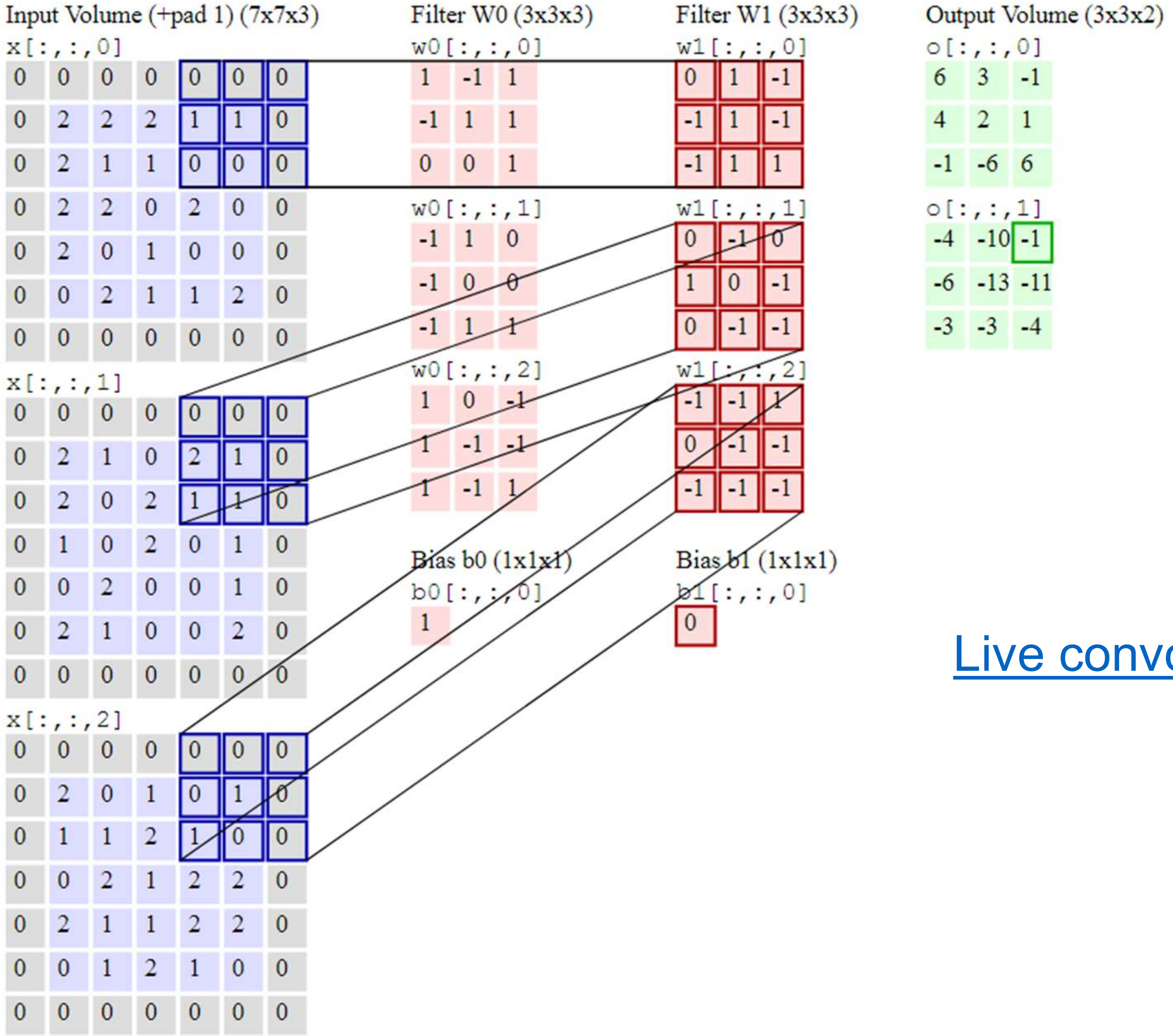
The brain/neuron view of CONV Layer





E.g. with 5 filters,
CONV layer consists of
neurons arranged in a 3D grid
(28x28x5)

There will be 5 different
neurons all looking at the same
region in the input volume



Input Volume (+pad 1) (7x7x3)

x[:,:,0]

0	0	0	0	0	0	0
0	2	2	2	1	1	0
0	2	1	1	0	0	0
0	2	2	0	2	0	0
0	2	0	1	0	0	0
0	0	2	1	1	2	0
0	0	0	0	0	0	0

x[:,:,1]

0	0	0	0	0	0	0
0	2	1	0	2	1	0
0	2	0	2	1	1	0
0	1	0	2	0	1	0
0	0	2	0	0	1	0
0	2	1	0	0	2	0
0	0	0	0	0	0	0

x[:,:,2]

0	0	0	0	0	0	0
0	2	0	1	0	1	0
0	1	1	2	1	0	0
0	0	2	1	2	2	0
0	2	1	1	2	2	0
0	0	1	2	1	0	0
0	0	0	0	0	0	0

Filter W0 (3x3x3)

w0[:,:,0]

1	-1	1
-1	1	1
0	0	1

w0[:,:,1]

-1	1	0
-1	0	0
-1	1	1

w0[:,:,2]

1	0	-1
1	-1	-1
1	-1	1

Bias b0 (1x1x1)

b0[:,:,0]

1

Filter W1 (3x3x3)

w1[:,:,0]

0	1	-1
-1	1	-1
-1	1	1

w1[:,:,1]

0	-1	0
1	0	-1
0	-1	-1

w1[:,:,2]

-1	-1	1
0	-1	-1
-1	-1	-1

Bias b1 (1x1x1)

b1[:,:,0]

0

Output Volume (3x3x2)

o[:,:,0]

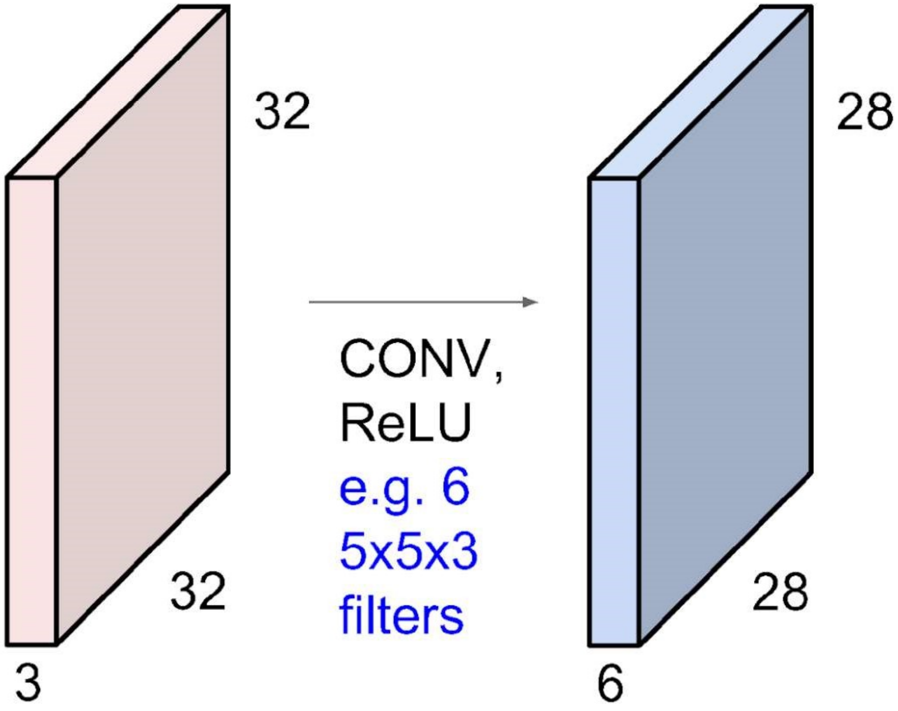
6	3	-1
4	2	1
-1	-6	6

o[:,:,1]

-4	-10	-1
-6	-13	-11
-3	-3	-4

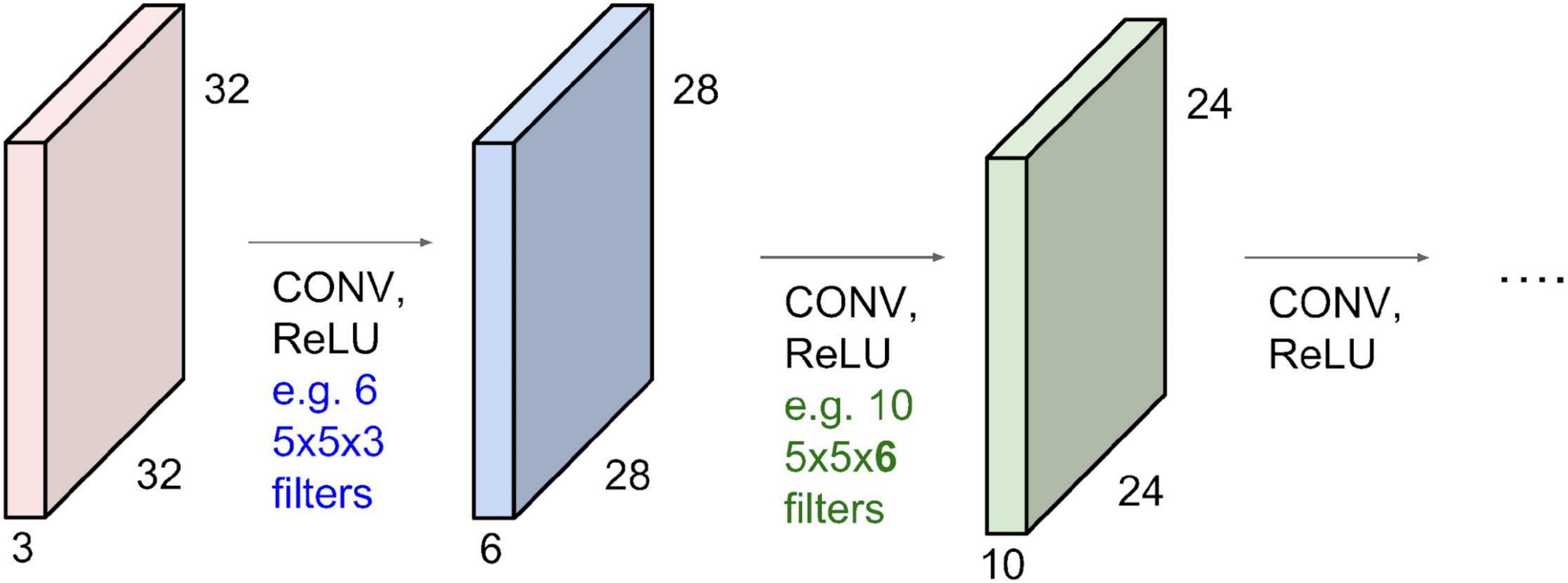
Live convolution demo

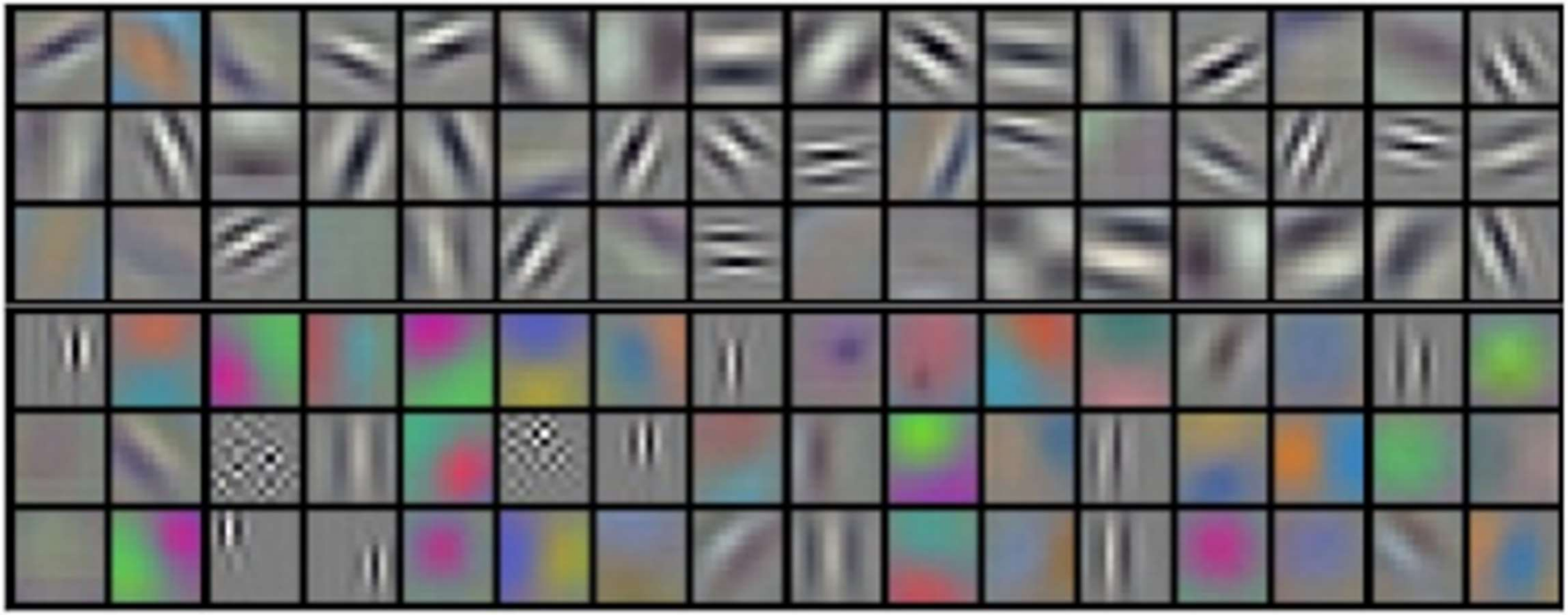
Preview: ConvNet is a sequence of Convolution Layers, interspersed with activation functions



CONV,
ReLU
e.g. 6
5x5x3
filters

Preview: ConvNet is a sequence of Convolution Layers, interspersed with activation functions

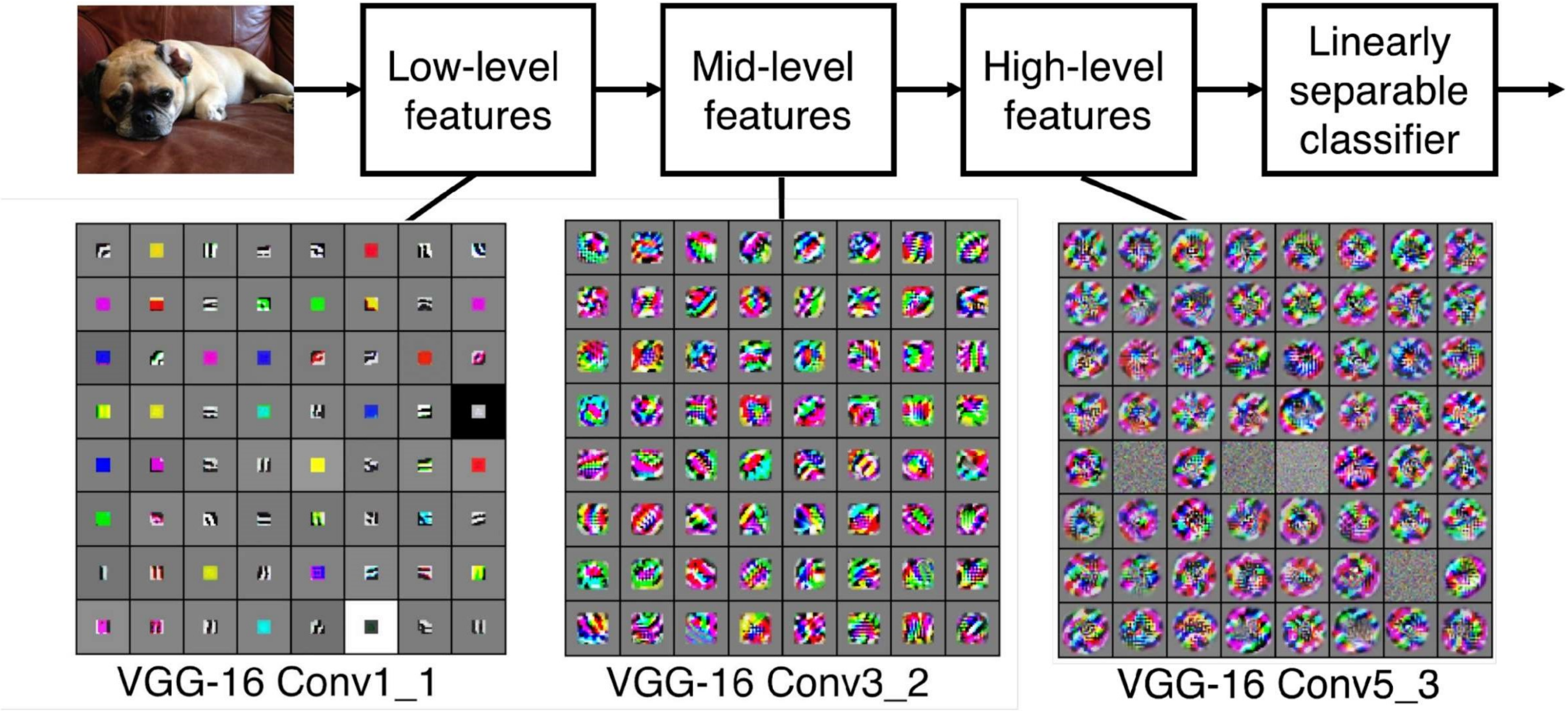




Preview

[Zeiler and Fergus 2013]

Visualization of VGG-16 by Lane McIntosh. VGG-16 architecture from [Simonyan and Zisserman 2014].



Low-level features

Mid-level features

High-level features

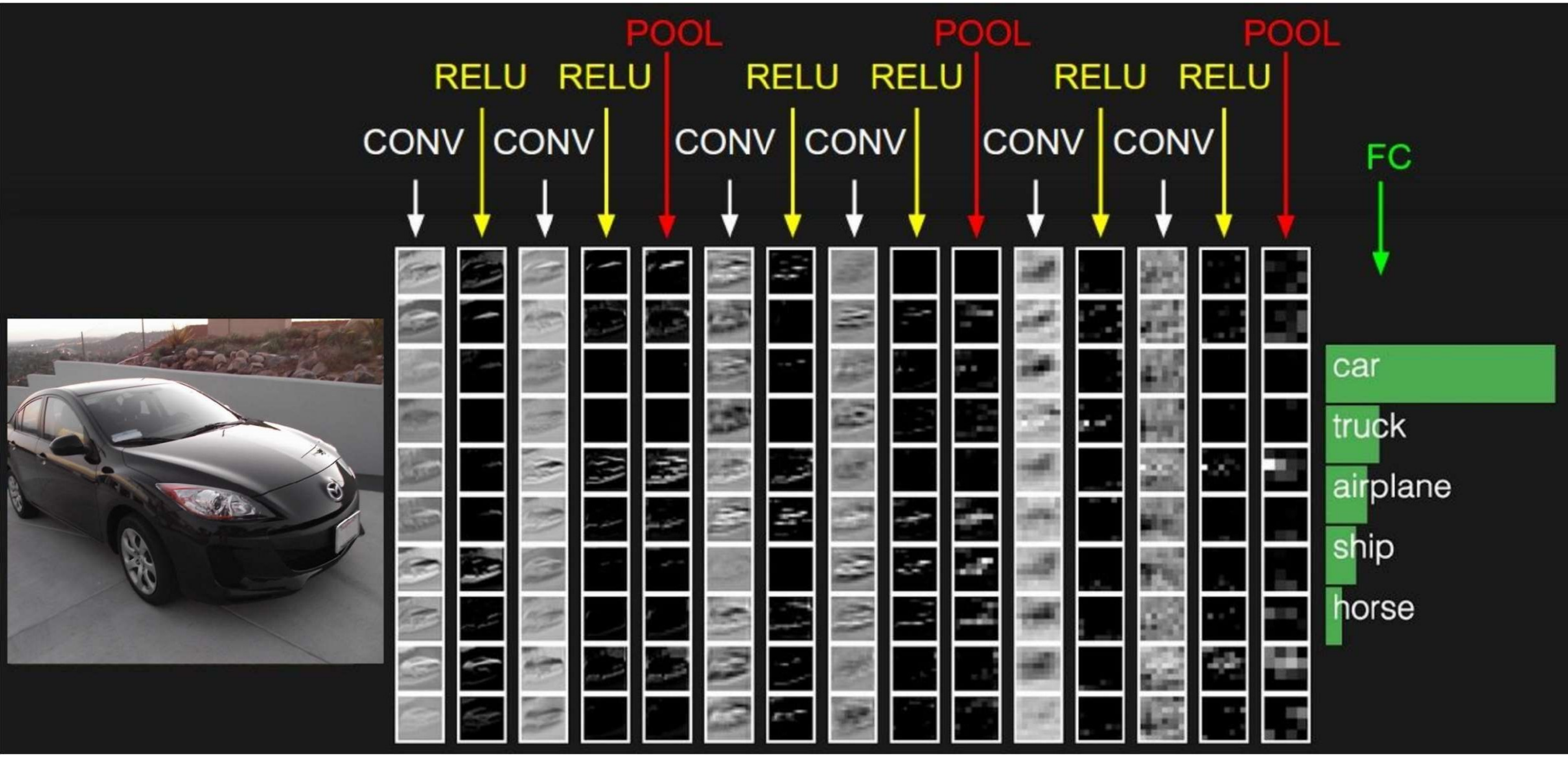
Linearly separable classifier

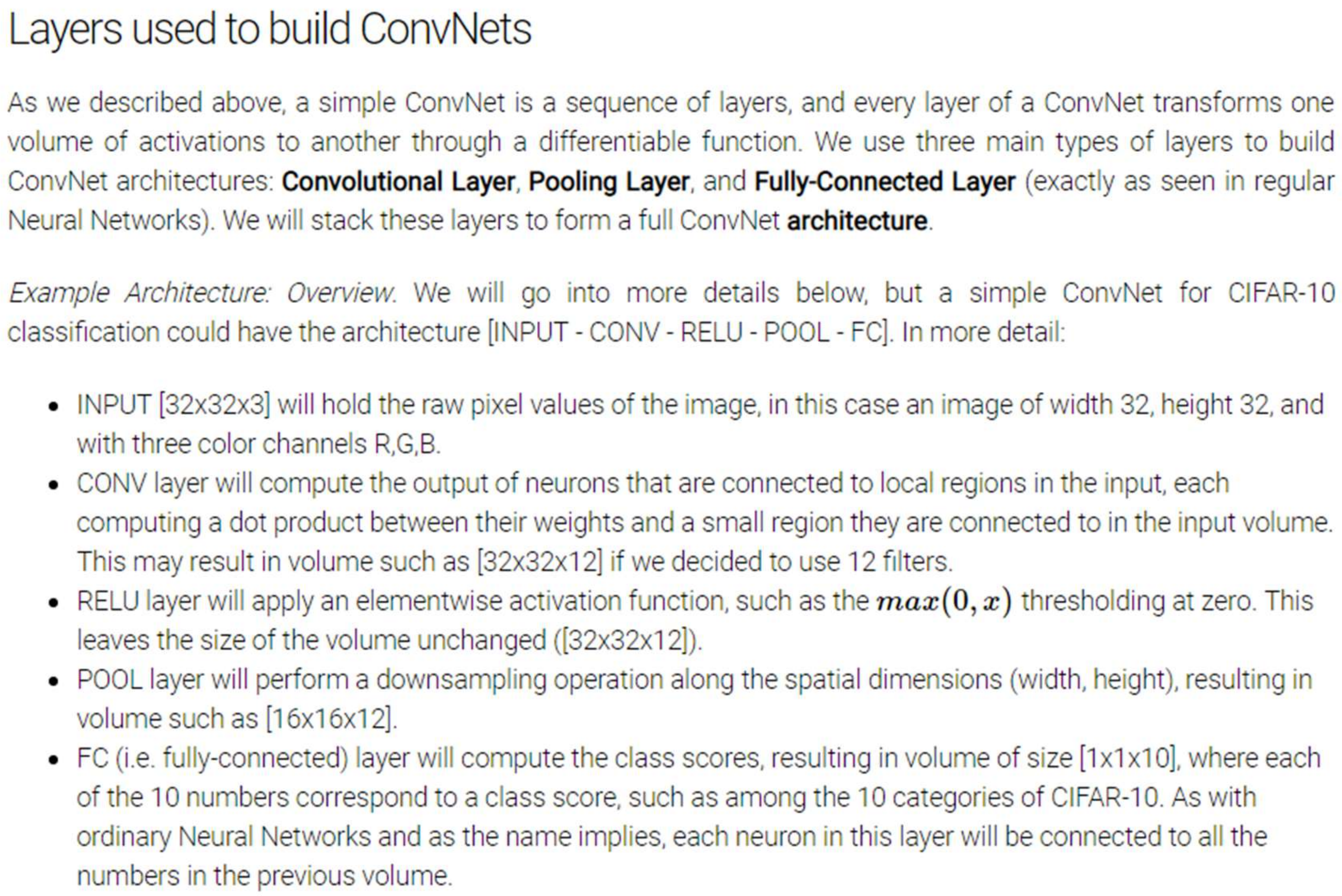
VGG-16 Conv1_1

VGG-16 Conv3_2

VGG-16 Conv5_3

preview:


POOL
POOL
POOL
RELU
RELU
RELU
RELU
RELU
RELU
CONV
CONV
CONV
CONV
CONV
CONV
FC
car
truck
airplane
ship
horse



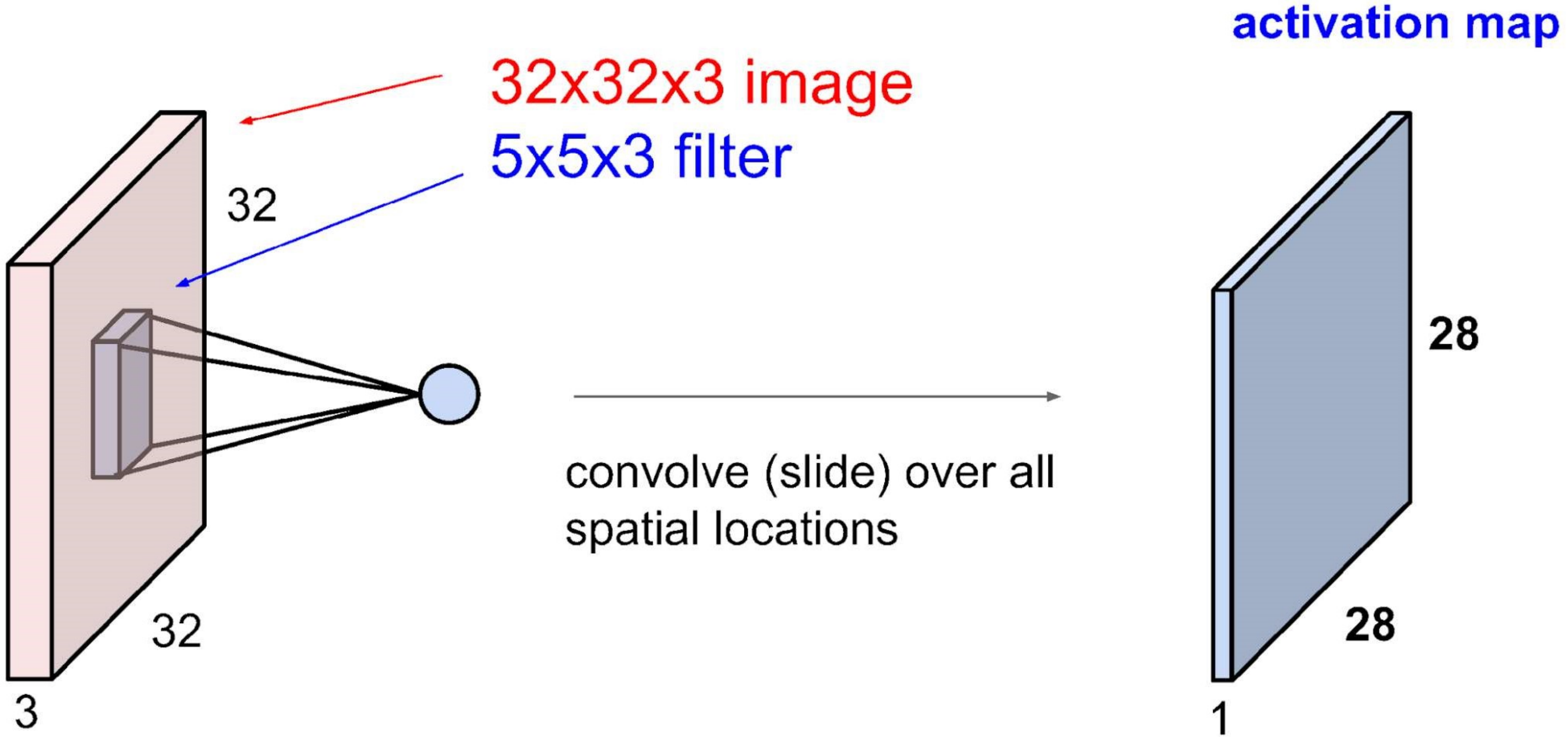
Layers used to build ConvNets

As we described above, a simple ConvNet is a sequence of layers, and every layer of a ConvNet transforms one volume of activations to another through a differentiable function. We use three main types of layers to build ConvNet architectures: **Convolutional Layer**, **Pooling Layer**, and **Fully-Connected Layer** (exactly as seen in regular Neural Networks). We will stack these layers to form a full ConvNet **architecture**.

Example Architecture: Overview. We will go into more details below, but a simple ConvNet for CIFAR-10 classification could have the architecture [INPUT - CONV - RELU - POOL - FC]. In more detail:

- INPUT [32x32x3] will hold the raw pixel values of the image, in this case an image of width 32, height 32, and with three color channels R,G,B.
- CONV layer will compute the output of neurons that are connected to local regions in the input, each computing a dot product between their weights and a small region they are connected to in the input volume. This may result in volume such as [32x32x12] if we decided to use 12 filters.
- RELU layer will apply an elementwise activation function, such as the $max(0, x)$ thresholding at zero. This leaves the size of the volume unchanged ([32x32x12]).
- POOL layer will perform a downsampling operation along the spatial dimensions (width, height), resulting in volume such as [16x16x12].
- FC (i.e. fully-connected) layer will compute the class scores, resulting in volume of size [1x1x10], where each of the 10 numbers correspond to a class score, such as among the 10 categories of CIFAR-10. As with ordinary Neural Networks and as the name implies, each neuron in this layer will be connected to all the numbers in the previous volume.

A closer look at spatial dimensions:



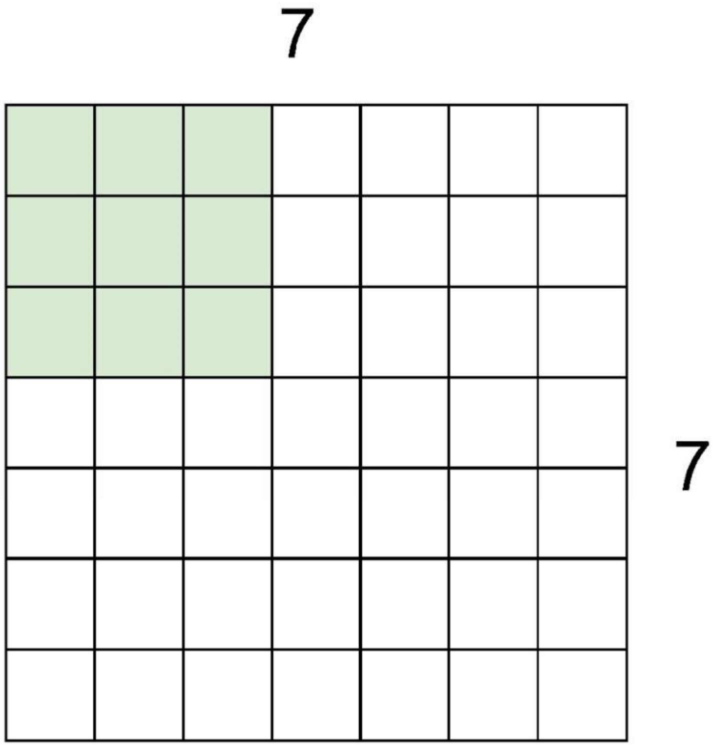
32x32x3 image

5x5x3 filter

convolve (slide) over all spatial locations

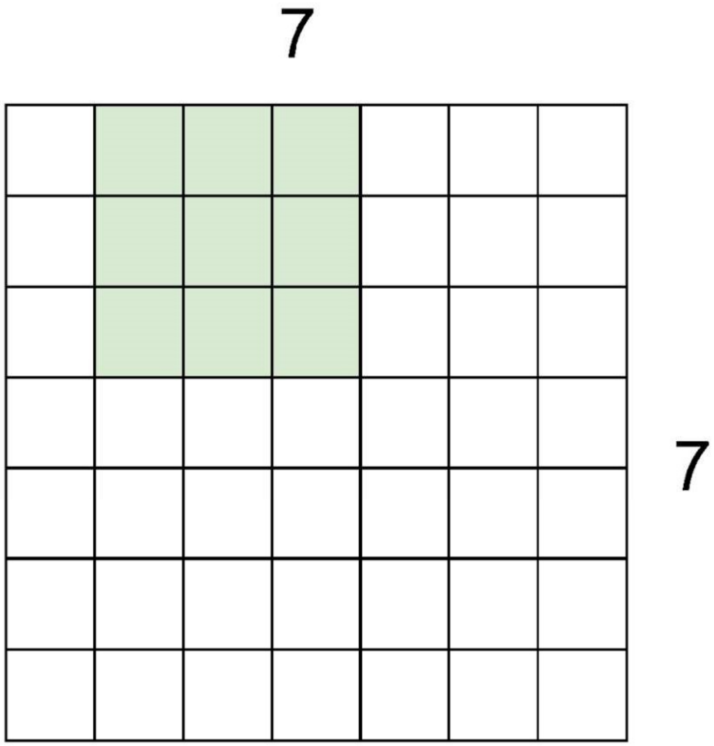
activation map

A closer look at spatial dimensions:



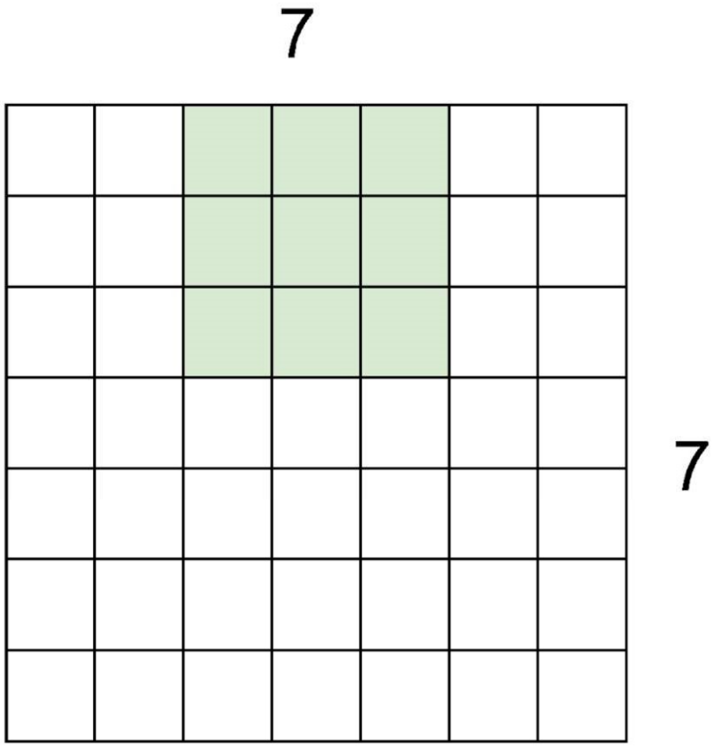
7x7 input (spatially)
assume 3x3 filter

A closer look at spatial dimensions:



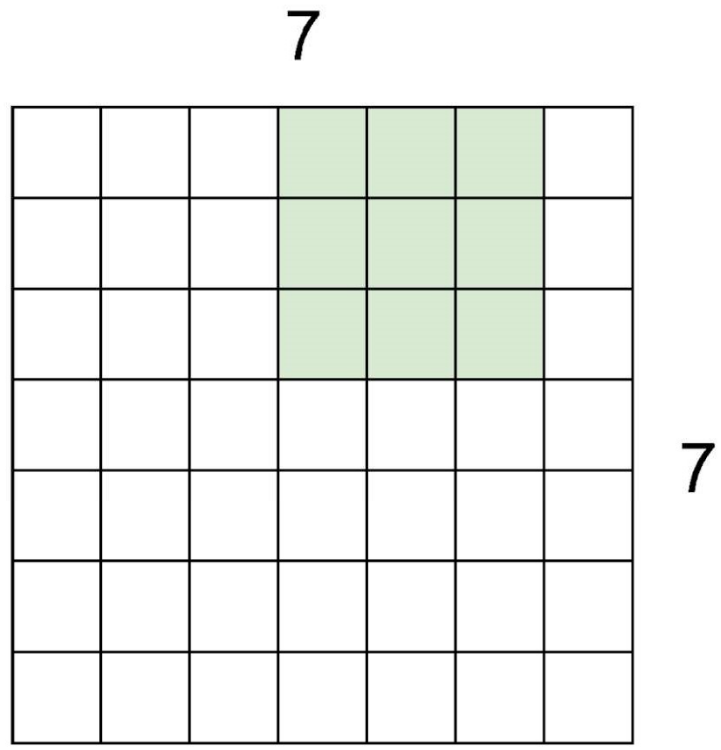
7x7 input (spatially)
assume 3x3 filter

A closer look at spatial dimensions:



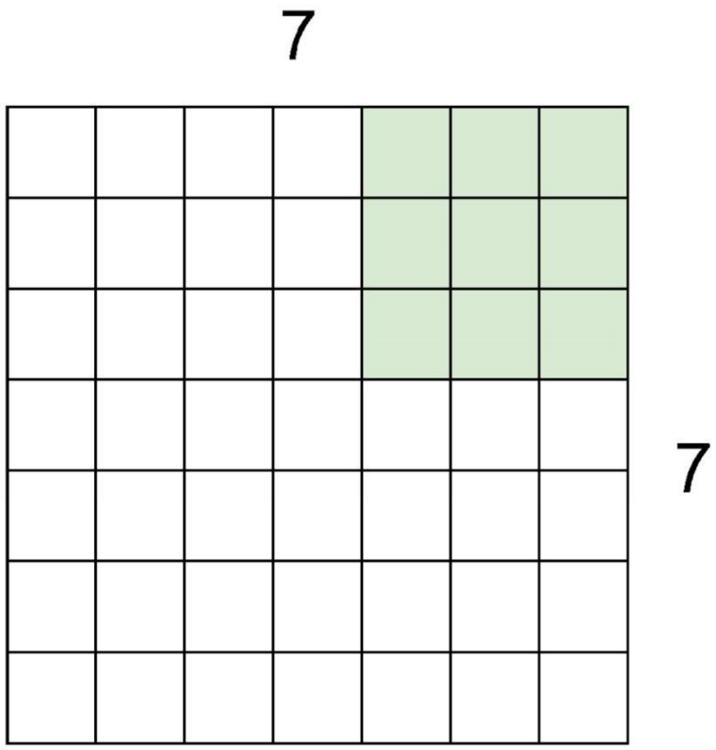
7x7 input (spatially)
assume 3x3 filter

A closer look at spatial dimensions:



7x7 input (spatially)
assume 3x3 filter

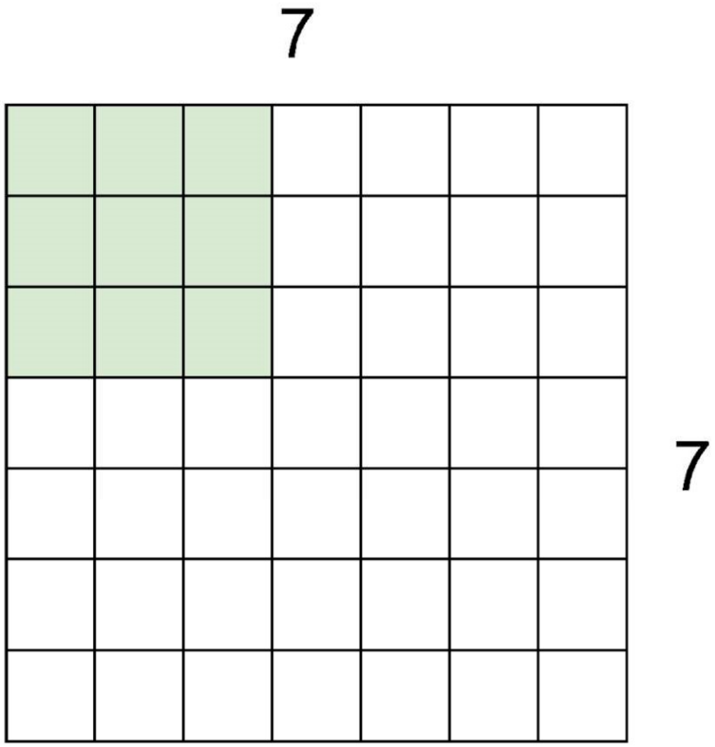
A closer look at spatial dimensions:



7x7 input (spatially)
assume 3x3 filter

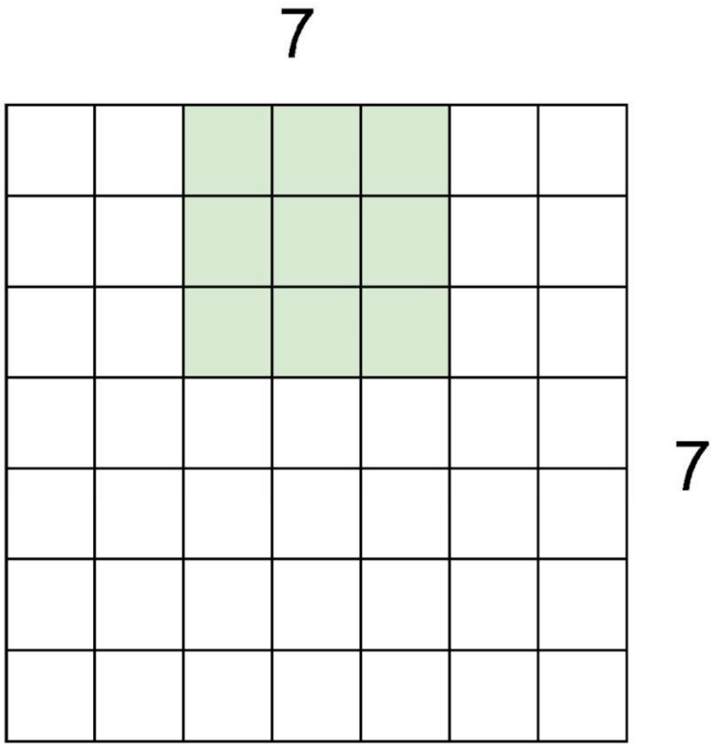
=> 5x5 output

A closer look at spatial dimensions:



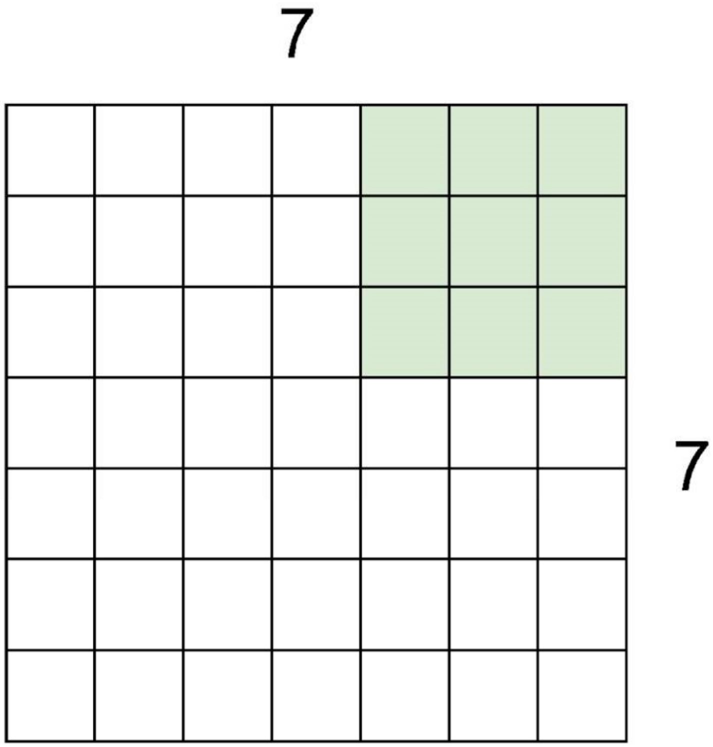
7x7 input (spatially)
assume 3x3 filter
applied **with stride 2**

A closer look at spatial dimensions:



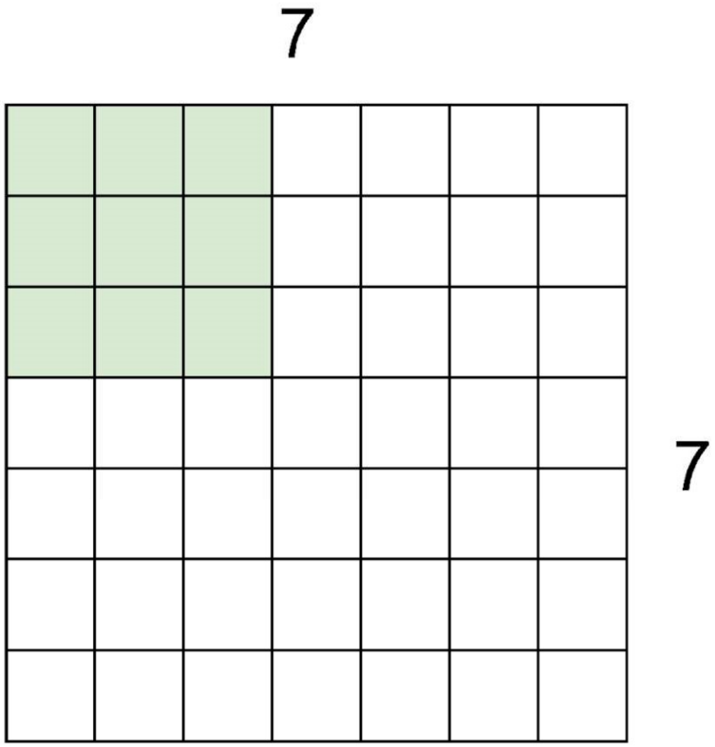
7x7 input (spatially)
assume 3x3 filter
applied **with stride 2**

A closer look at spatial dimensions:



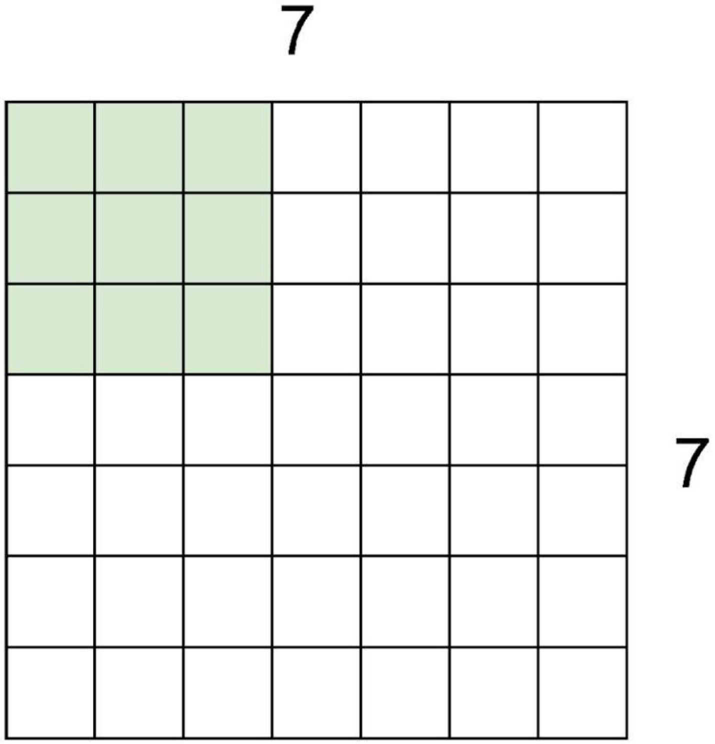
7x7 input (spatially)
assume 3x3 filter
applied **with stride 2**
=> 3x3 output!

A closer look at spatial dimensions:



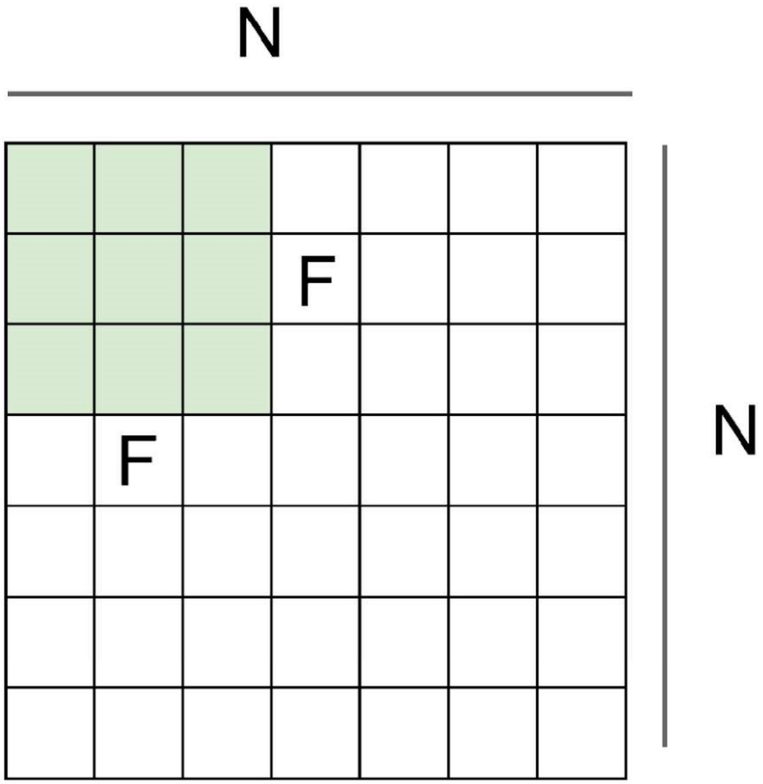
7x7 input (spatially)
assume 3x3 filter
applied **with stride 3?**

A closer look at spatial dimensions:



7x7 input (spatially)
assume 3x3 filter
applied **with stride 3?**

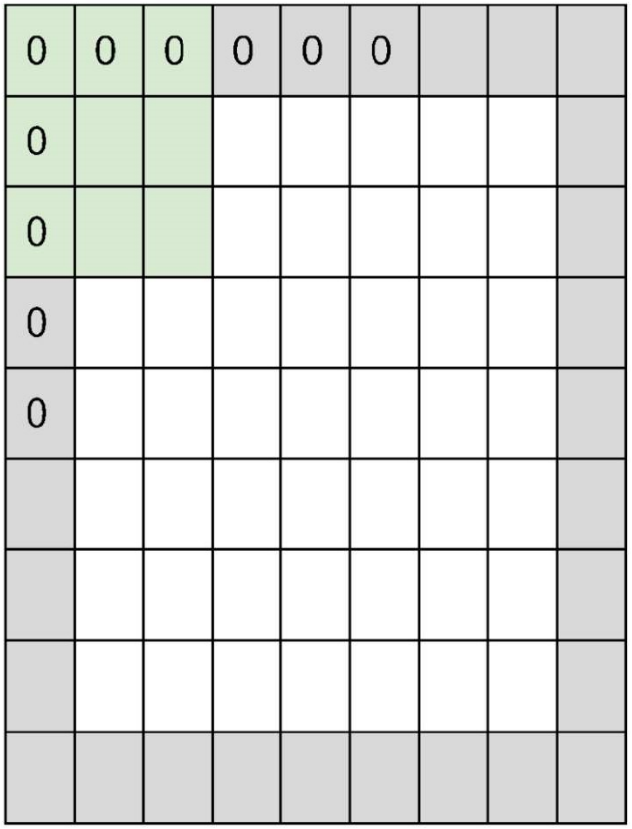
doesn’t fit!
cannot apply 3x3 filter on
7x7 input with stride 3.



Output size:
(N - F) / stride + 1

e.g. N = 7, F = 3:
stride 1 => (7 - 3)/1 + 1 = 5
stride 2 => (7 - 3)/2 + 1 = 3
stride 3 => (7 - 3)/3 + 1 = 2.33 :\

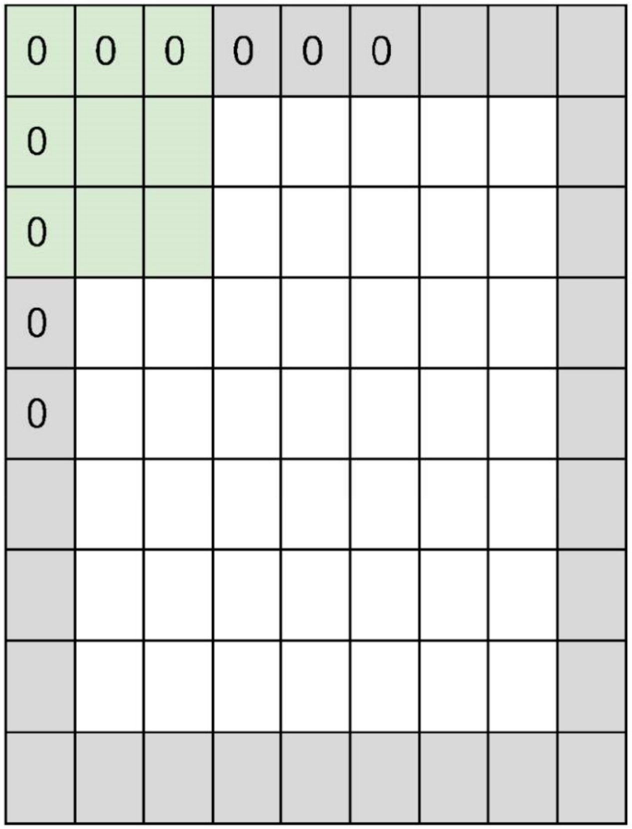
In practice: Common to zero pad the border



e.g. input 7x7
3x3 filter, applied with **stride 1**
pad with 1 pixel border => what is the output?

(recall:)
(N - F) / stride + 1

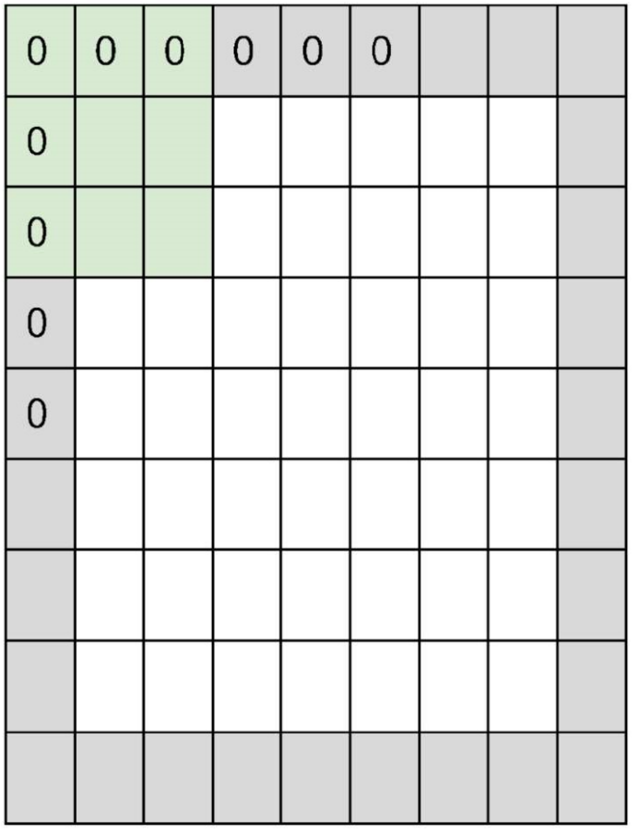
In practice: Common to zero pad the border



e.g. input 7x7
3x3 filter, applied with **stride 1**
pad with 1 pixel border => what is the output?

7x7 output!

In practice: Common to zero pad the border

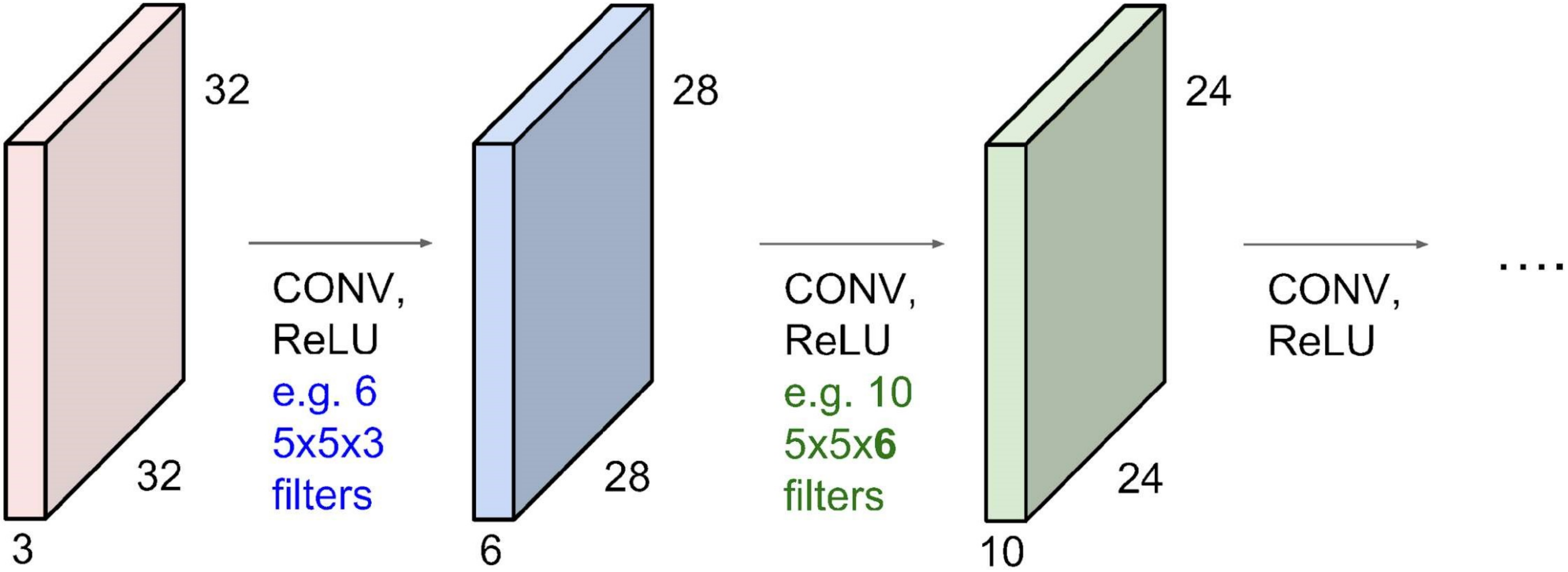


e.g. input 7x7
3x3 filter, applied with **stride 1**
pad with 1 pixel border => what is the output?

7x7 output!
in general, common to see CONV layers with stride 1, filters of size FxF, and zero-padding with (F-1)/2. (will preserve size spatially)
e.g. F = 3 => zero pad with 1
F = 5 => zero pad with 2
F = 7 => zero pad with 3

Remember back to…

E.g. 32x32 input convolved repeatedly with 5x5 filters shrinks volumes spatially! (32 -> 28 -> 24 ...). Shrinking too fast is not good, doesn’t work well.



CONV, ReLU e.g. 6 5x5x3 filters

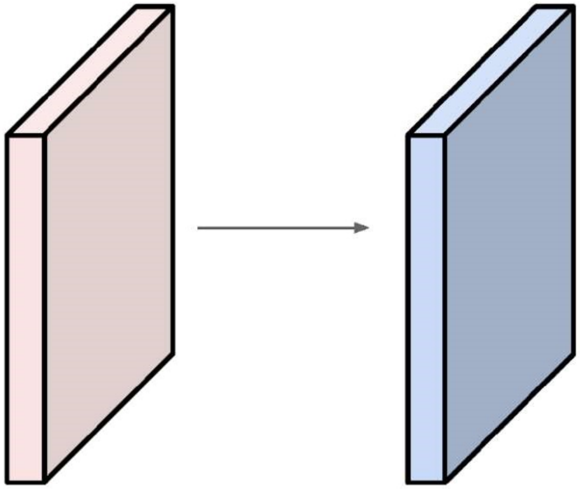
CONV, ReLU e.g. 10 5x5x**6** filters

CONV, ReLU

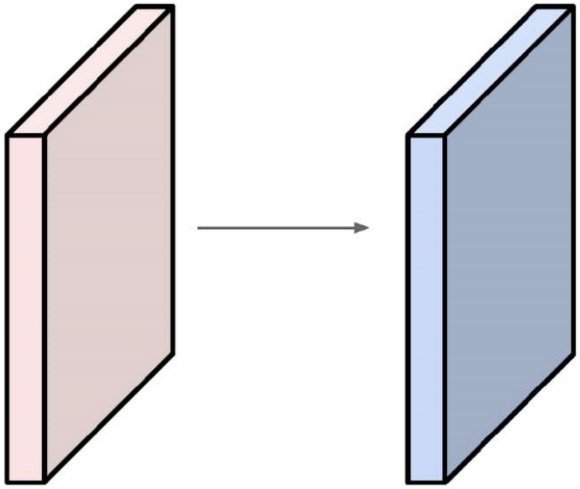
Examples time:

Input volume: **32x32x3**
10 5x5 filters with stride 1, pad 2

Output volume size: ?



Examples time:



Input volume: **32x32x3**

10 5x5 filters with stride 1, pad 2

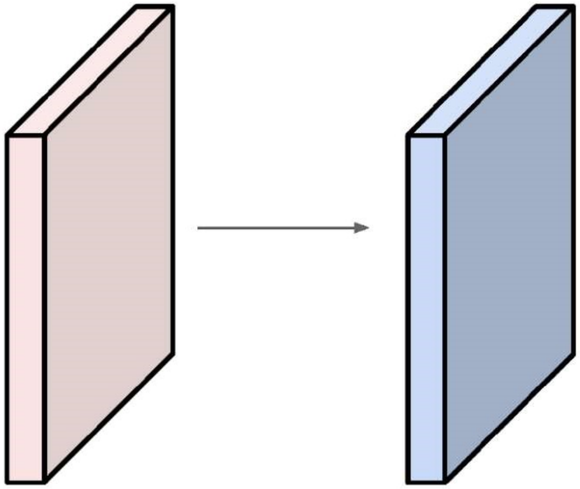
Output volume size:

(32+2*2-5)/1+1 = 32 spatially, so

32x32x10

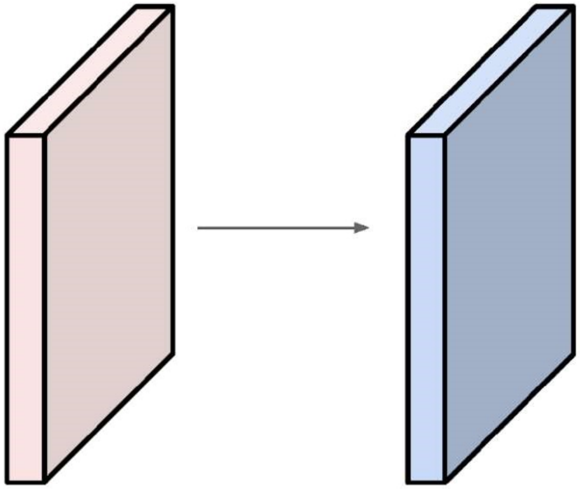
Examples time:

Input volume: **32x32x3**
10 5x5 filters with stride 1, pad 2



Number of parameters in this layer?

Examples time:



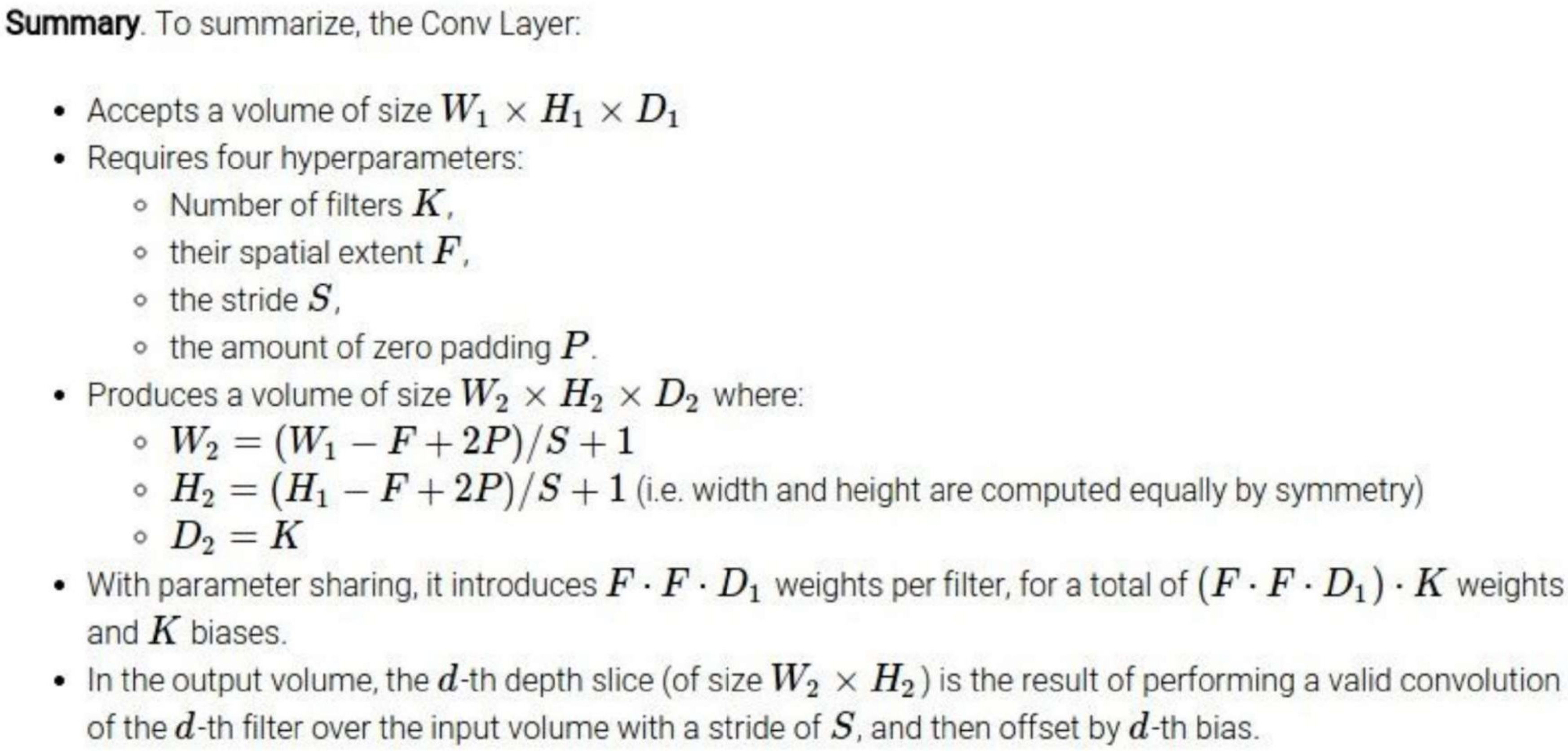
Input volume: **32x32x3**

10 5x5 filters with stride 1, pad 2

Number of parameters in this layer?

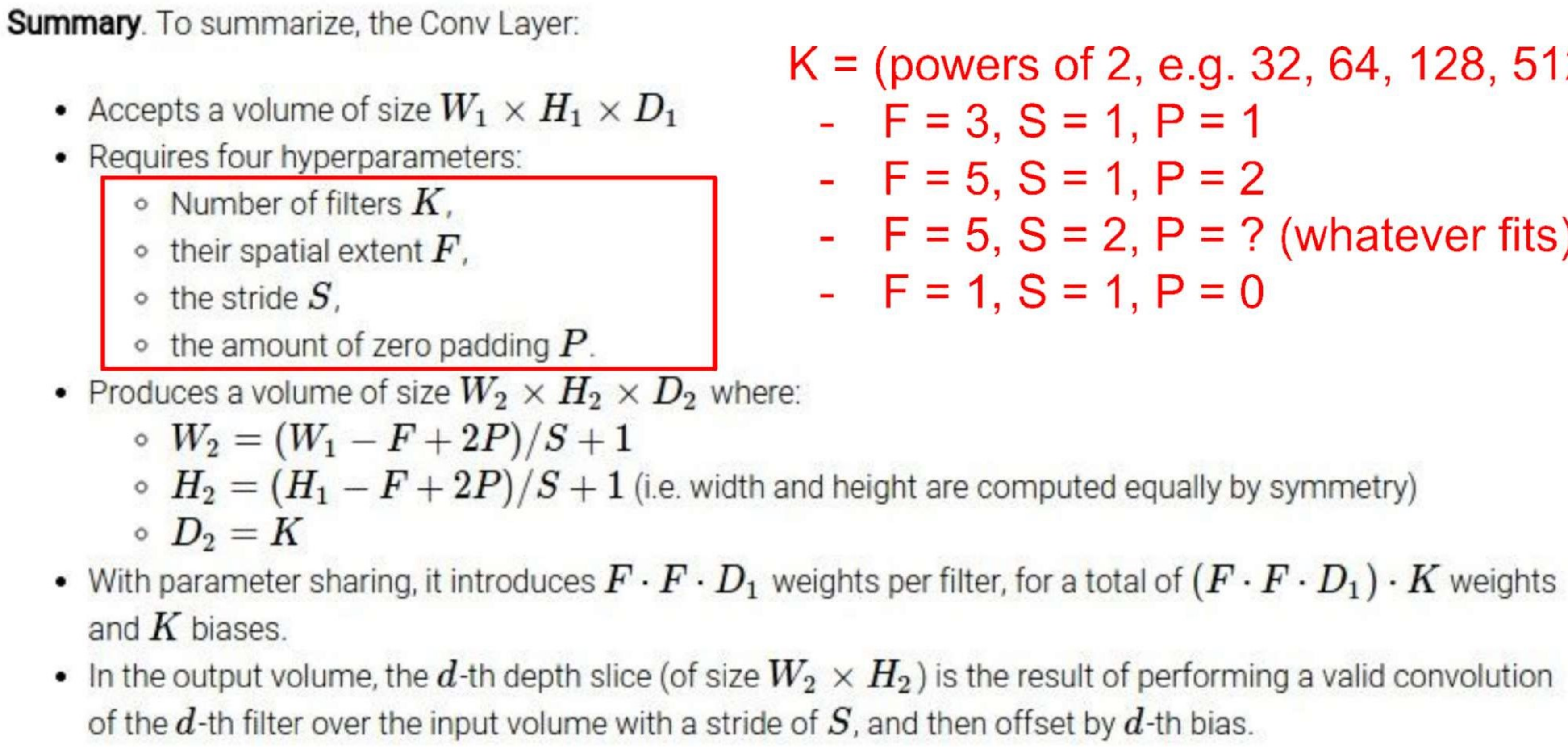
each filter has 5*5*3 + 1 = 76 params (+1 for bias)

=> 76*10 = **760**



Summary. To summarize, the Conv Layer:

- Accepts a volume of size $W_1 \times H_1 \times D_1$
- Requires four hyperparameters:
  - Number of filters K,
  - their spatial extent F,
  - the stride S,
  - the amount of zero padding P.
- Produces a volume of size $W_2 \times H_2 \times D_2$ where:
  - $W_2 = (W_1 - F + 2P)/S + 1$
  - $H_2 = (H_1 - F + 2P)/S + 1$ (i.e. width and height are computed equally by symmetry)
  - $D_2 = K$
- With parameter sharing, it introduces $F \cdot F \cdot D_1$ weights per filter, for a total of $(F \cdot F \cdot D_1) \cdot K$ weights and K biases.
- In the output volume, the d-th depth slice (of size $W_2 \times H_2$) is the result of performing a valid convolution of the d-th filter over the input volume with a stride of S, and then offset by d-th bias.



Summary. To summarize, the Conv Layer:

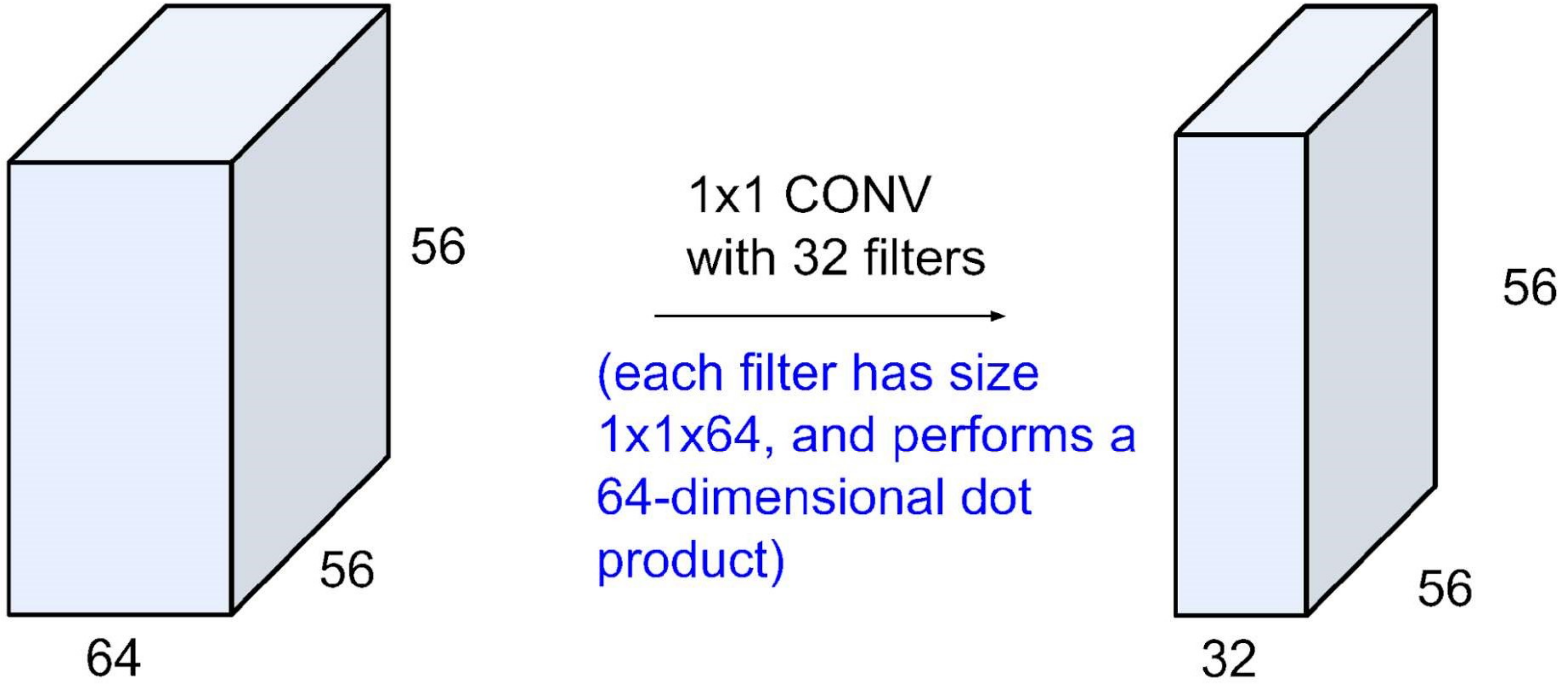
- Accepts a volume of size $W_1 \times H_1 \times D_1$
- Requires four hyperparameters:
  - Number of filters K,
  - their spatial extent F,
  - the stride S,
  - the amount of zero padding P.
- Produces a volume of size $W_2 \times H_2 \times D_2$ where:
  - $W_2 = (W_1 - F + 2P)/S + 1$
  - $H_2 = (H_1 - F + 2P)/S + 1$ (i.e. width and height are computed equally by symmetry)
  - $D_2 = K$
- With parameter sharing, it introduces $F \cdot F \cdot D_1$ weights per filter, for a total of $(F \cdot F \cdot D_1) \cdot K$ weights and K biases.
- In the output volume, the d-th depth slice (of size $W_2 \times H_2$) is the result of performing a valid convolution of the d-th filter over the input volume with a stride of S, and then offset by d-th bias.

Common settings:

K = (powers of 2, e.g. 32, 64, 128, 512)

- F = 3, S = 1, P = 1
- F = 5, S = 1, P = 2
- F = 5, S = 2, P = ? (whatever fits)
- F = 1, S = 1, P = 0

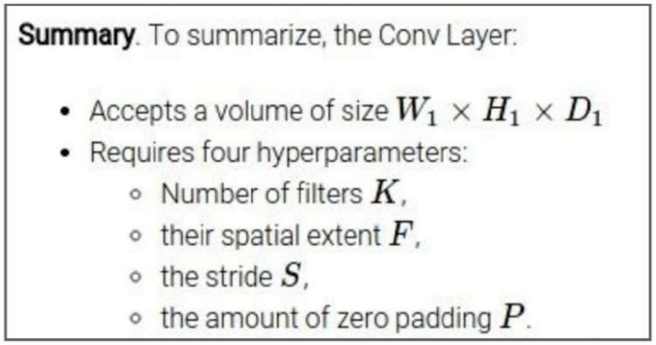
(btw, 1x1 convolution layers make perfect sense)



1x1 CONV
with 32 filters

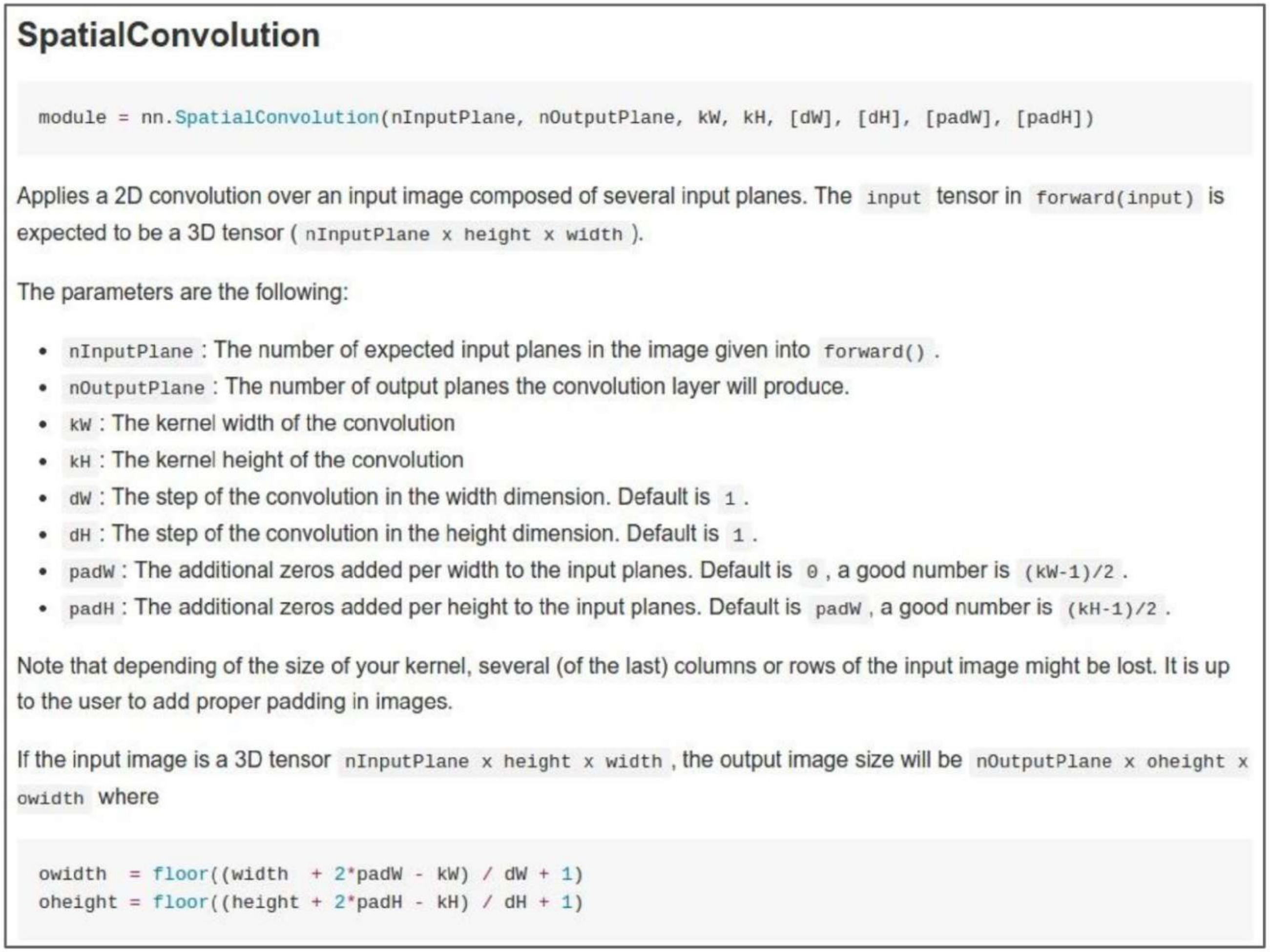
(each filter has size 1x1x64, and performs a 64-dimensional dot product)

Example: CONV layer in Torch



Summary. To summarize, the Conv Layer:

- Accepts a volume of size $W_1 \times H_1 \times D_1$
- Requires four hyperparameters:
  - Number of filters K,
  - their spatial extent F,
  - the stride S,
  - the amount of zero padding P.



SpatialConvolution

```
module = nn.SpatialConvolution(nInputPlane, nOutputPlane, kW, kH, [dW], [dH], [padW], [padH])
```

Applies a 2D convolution over an input image composed of several input planes. The `input` tensor in `forward(input)` is expected to be a 3D tensor (`nInputPlane x height x width`).

The parameters are the following:

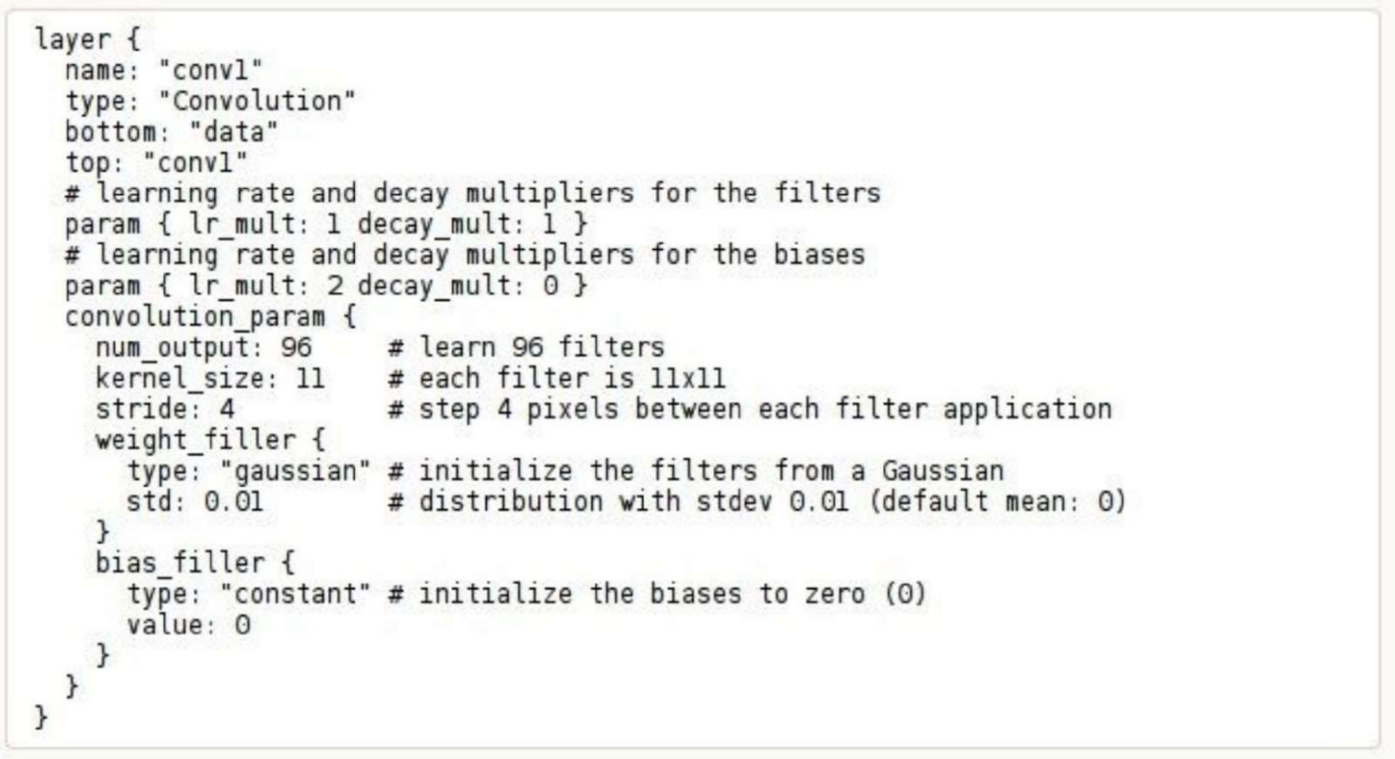
- `nInputPlane` : The number of expected input planes in the image given into `forward()` .
- `nOutputPlane` : The number of output planes the convolution layer will produce.
- `kW` : The kernel width of the convolution
- `kH` : The kernel height of the convolution
- `dW` : The step of the convolution in the width dimension. Default is `1` .
- `dH` : The step of the convolution in the height dimension. Default is `1` .
- `padW` : The additional zeros added per width to the input planes. Default is `0` , a good number is `(kW-1)/2` .
- `padH` : The additional zeros added per height to the input planes. Default is `padW` , a good number is `(kH-1)/2` .

Note that depending of the size of your kernel, several (of the last) columns or rows of the input image might be lost. It is up to the user to add proper padding in images.

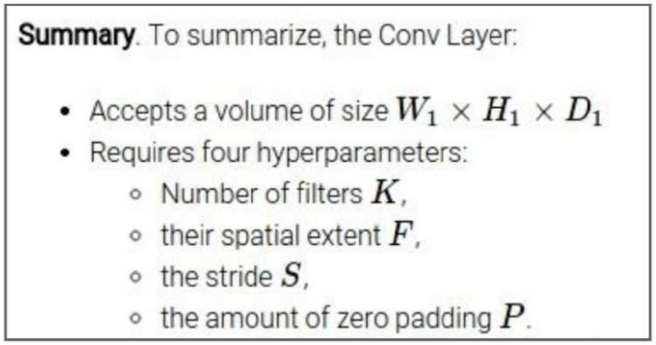
If the input image is a 3D tensor `nInputPlane x height x width` , the output image size will be `nOutputPlane x oheight x owidth` where

```
owidth  = floor((width  + 2*padW - kW) / dW + 1)
oheight = floor((height + 2*padH - kH) / dH + 1)
```

Example: CONV layer in Caffe



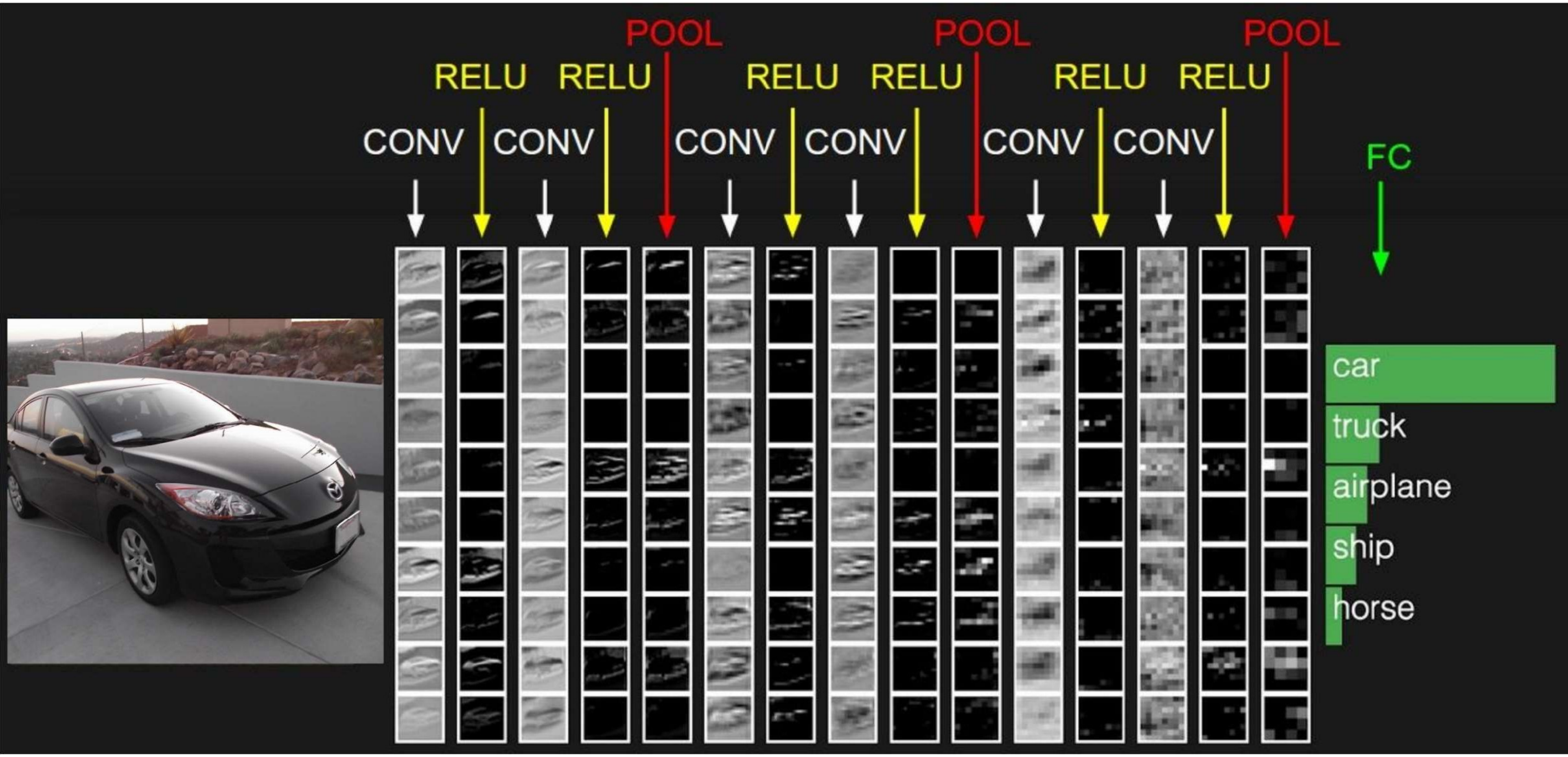
```
layer {
  name: "conv1"
  type: "Convolution"
  bottom: "data"
  top: "conv1"
  # learning rate and decay multipliers for the filters
  param { lr_mult: 1 decay_mult: 1 }
  # learning rate and decay multipliers for the biases
  param { lr_mult: 2 decay_mult: 0 }
  convolution_param {
    num_output: 96     # learn 96 filters
    kernel_size: 11    # each filter is 11x11
    stride: 4          # step 4 pixels between each filter application
    weight_filler {
      type: "gaussian" # initialize the filters from a Gaussian
      std: 0.01        # distribution with stdev 0.01 (default mean: 0)
    }
    bias_filler {
      type: "constant" # initialize the biases to zero (0)
      value: 0
    }
  }
}
```



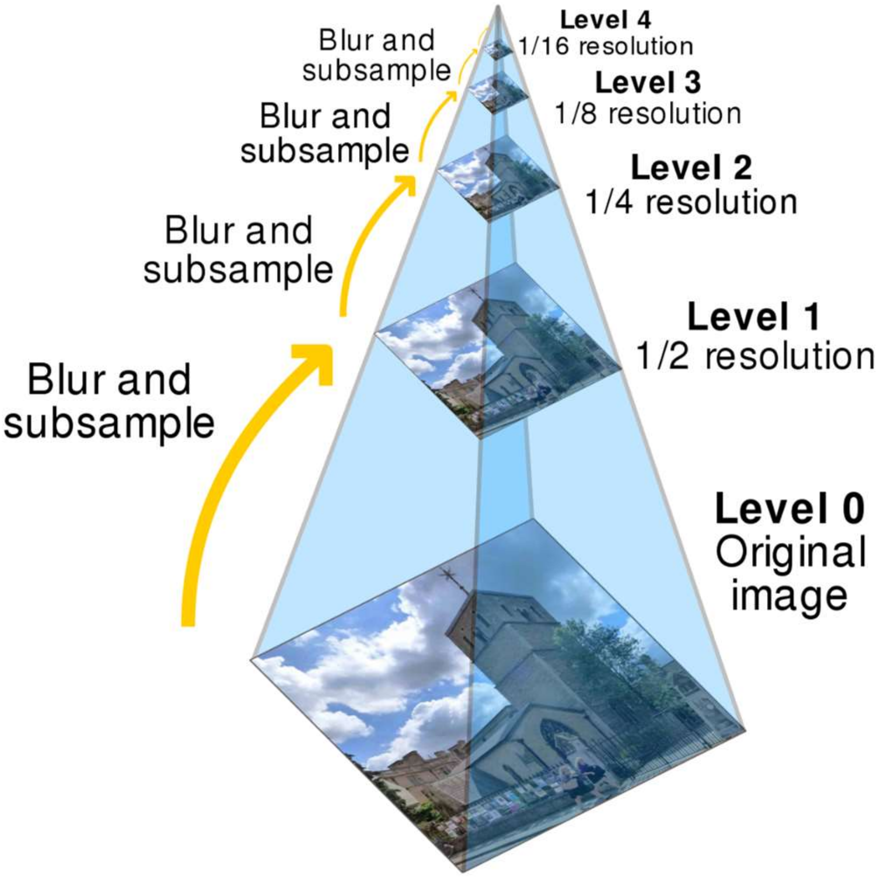
Summary. To summarize, the Conv Layer:

- Accepts a volume of size $W_1 \times H_1 \times D_1$
- Requires four hyperparameters:
  - Number of filters K,
  - their spatial extent F,
  - the stride S,
  - the amount of zero padding P.

two more layers to go: POOL/FC

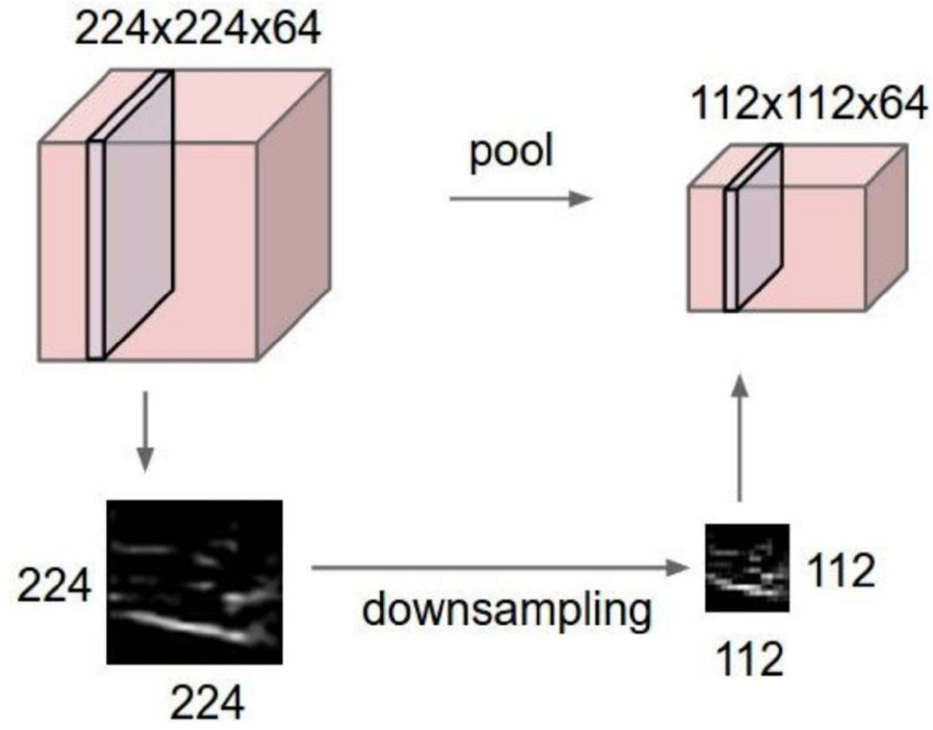


Pyramid (Image processing) - Subsampling

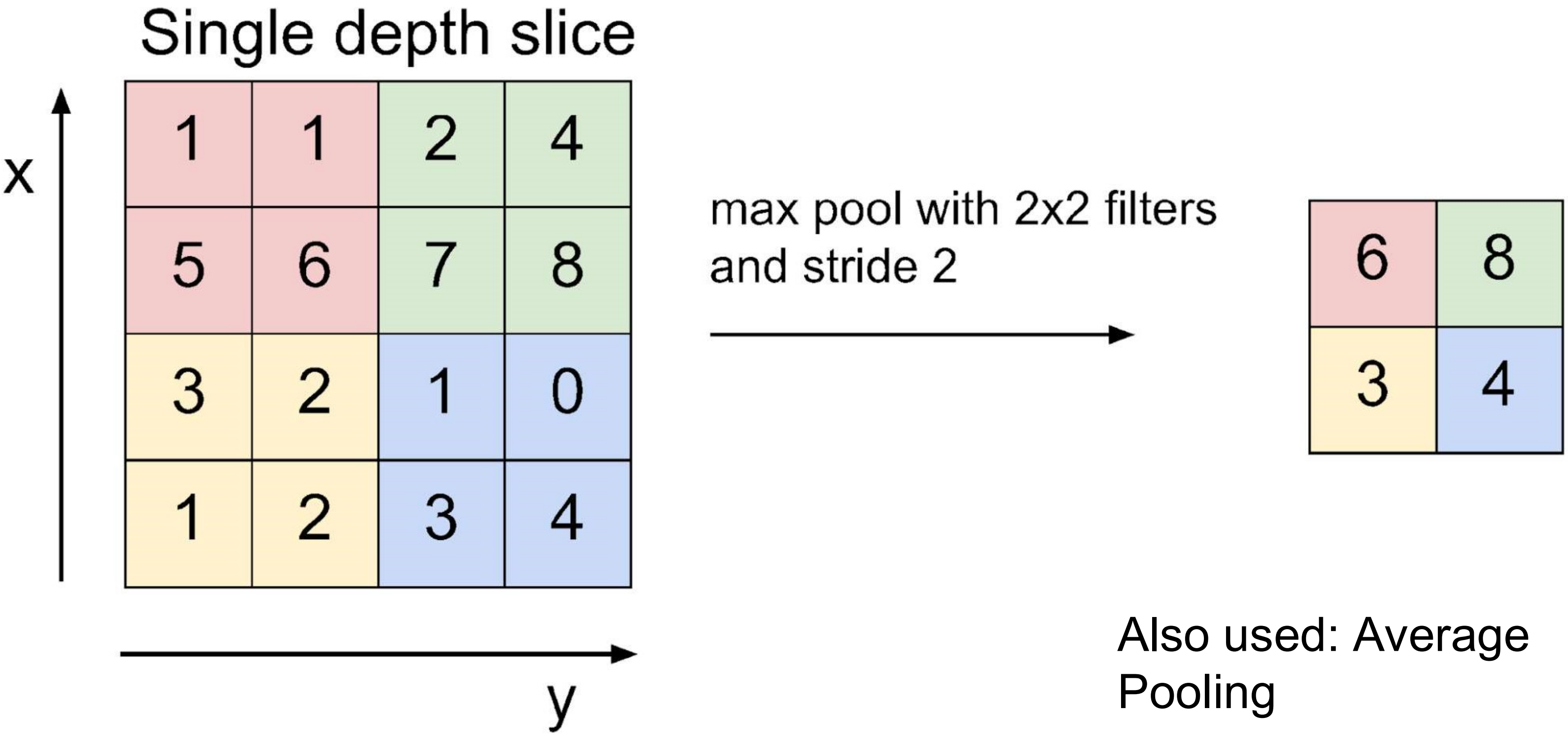


Pooling layer

- makes the representations smaller and more manageable
- operates over each activation map independently:



MAX POOLING



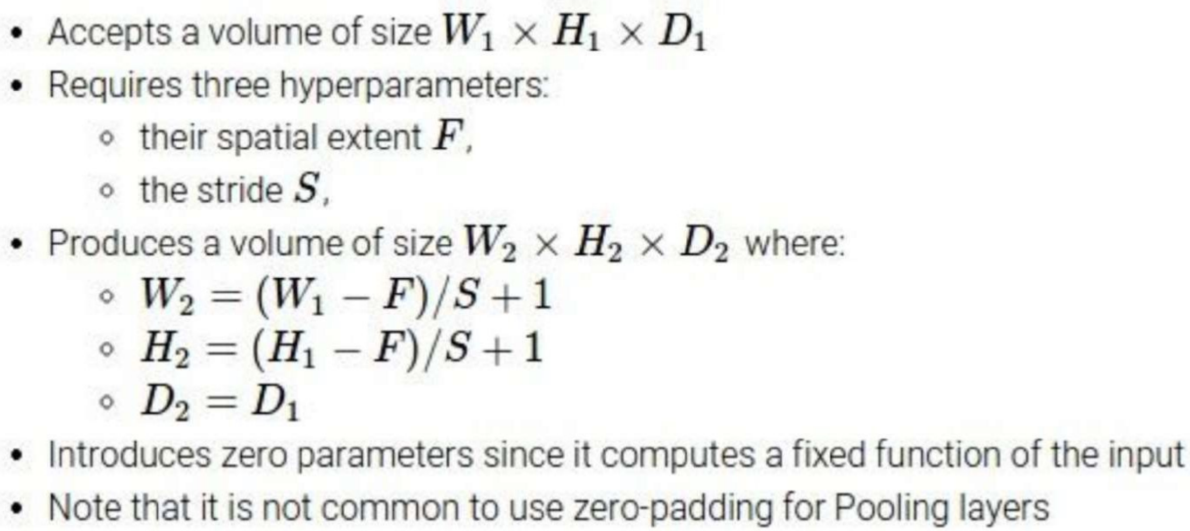
Single depth slice

1	1	2	4
5	6	7	8
3	2	1	0
1	2	3	4

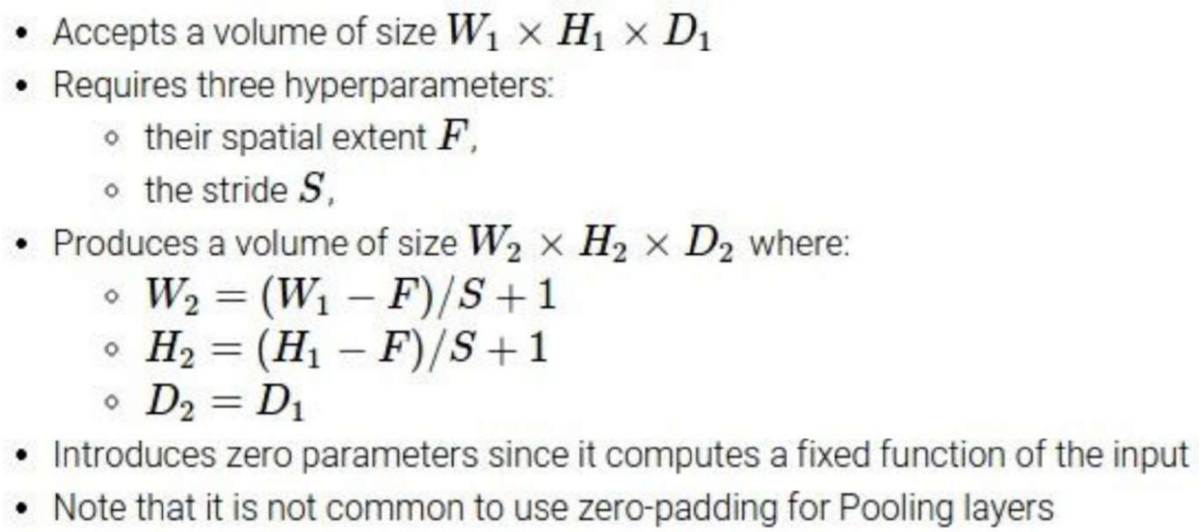
max pool with 2x2 filters and stride 2

6	8
3	4

Also used: Average Pooling



- Accepts a volume of size $W_1 \times H_1 \times D_1$
- Requires three hyperparameters:
  - their spatial extent F,
  - the stride S,
- Produces a volume of size $W_2 \times H_2 \times D_2$ where:
  - $W_2 = (W_1 - F)/S + 1$
  - $H_2 = (H_1 - F)/S + 1$
  - $D_2 = D_1$
- Introduces zero parameters since it computes a fixed function of the input
- Note that it is not common to use zero-padding for Pooling layers



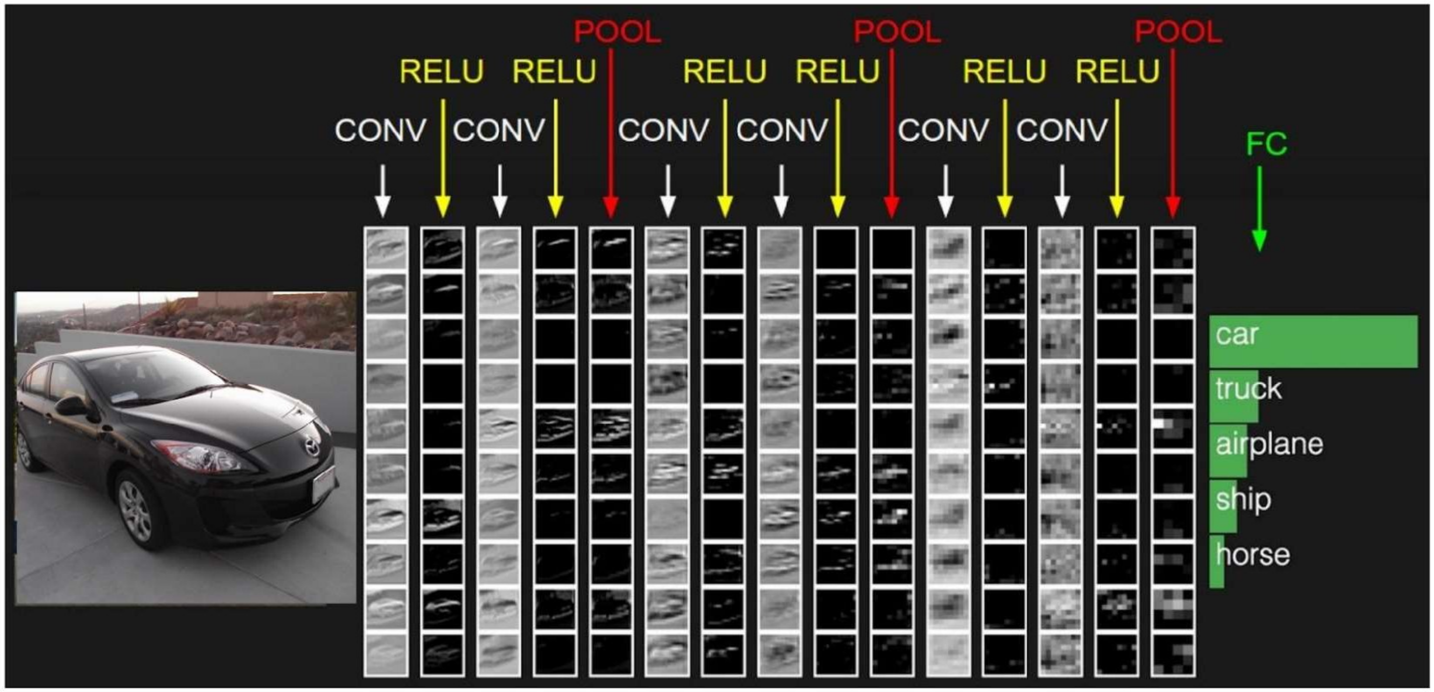
- Accepts a volume of size $W_1 \times H_1 \times D_1$
- Requires three hyperparameters:
  - their spatial extent F,
  - the stride S,
- Produces a volume of size $W_2 \times H_2 \times D_2$ where:
  - $W_2 = (W_1 - F)/S + 1$
  - $H_2 = (H_1 - F)/S + 1$
  - $D_2 = D_1$
- Introduces zero parameters since it computes a fixed function of the input
- Note that it is not common to use zero-padding for Pooling layers

Common settings:

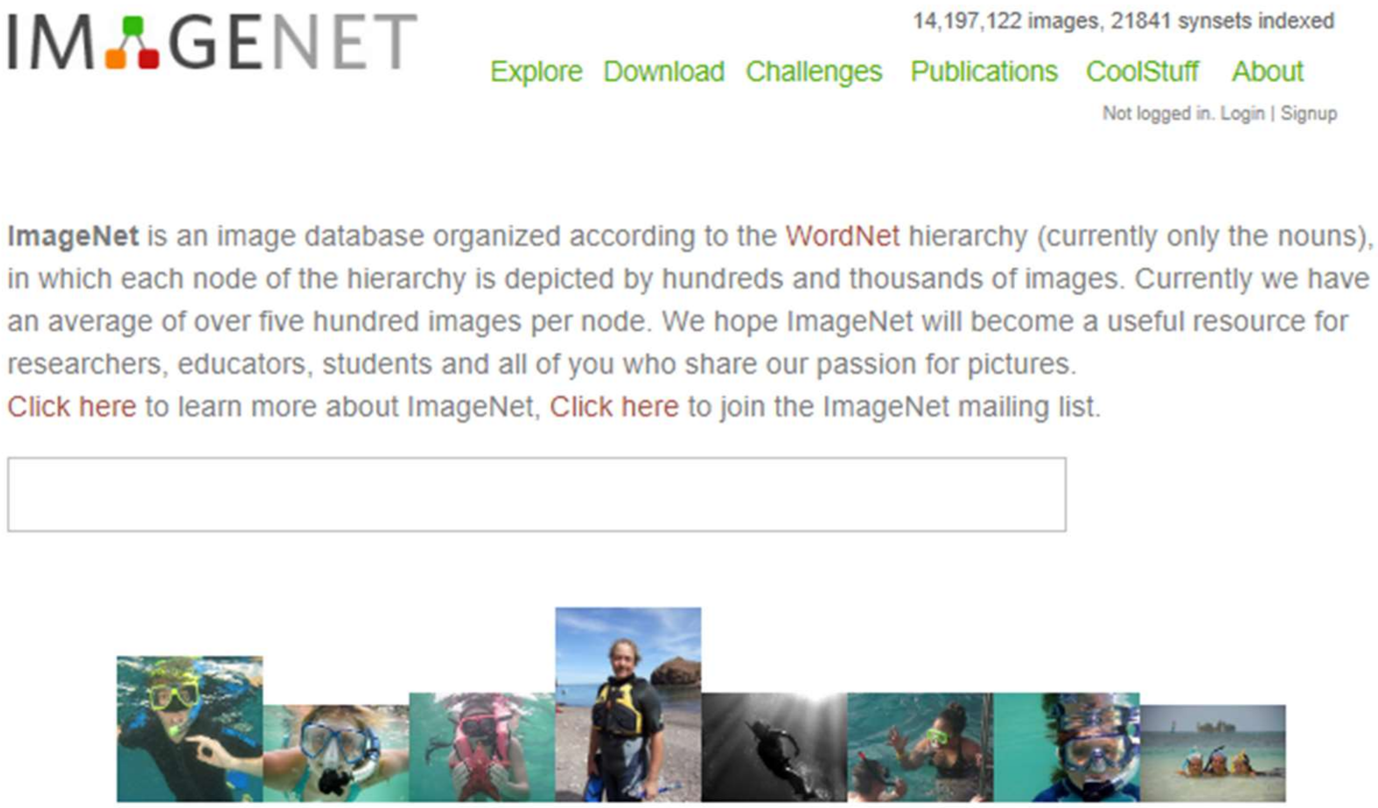
F = 2, S = 2
F = 3, S = 2

Fully Connected Layer (FC layer)

- Contains neurons that connect to the entire input volume, as in ordinary Neural Networks



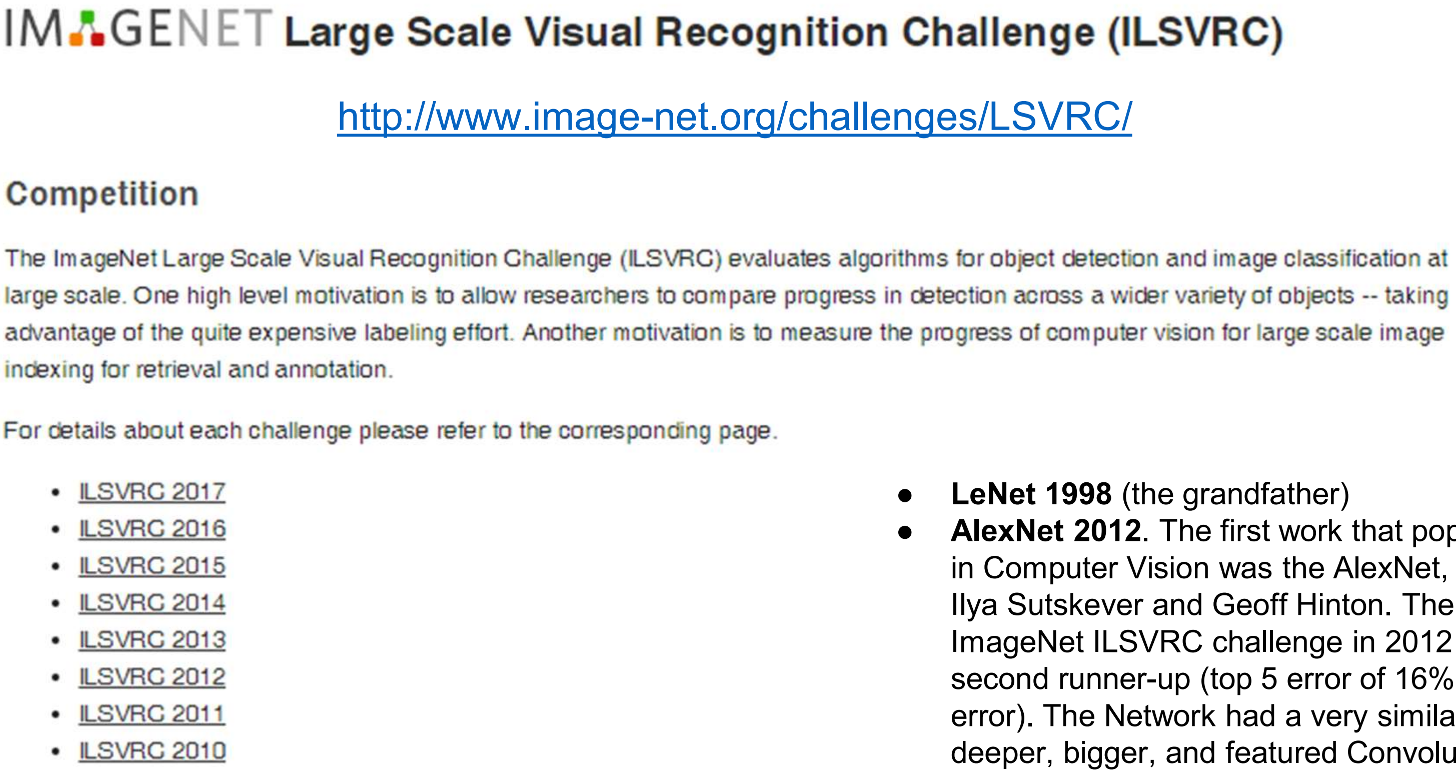
CNN Architectures
(ImageNet)



14,197,122 images, 21841 synsets indexed

ImageNet is an image database organized according to the WordNet hierarchy (currently only the nouns), in which each node of the hierarchy is depicted by hundreds and thousands of images. Currently we have an average of over five hundred images per node. We hope ImageNet will become a useful resource for researchers, educators, students and all of you who share our passion for pictures.
Click here to learn more about ImageNet, Click here to join the ImageNet mailing list.

http://www.image-net.org/
A very large labeled images dataset
Currently: 14.197.122 images in 21.841 categories



IMAGENET Large Scale Visual Recognition Challenge (ILSVRC)

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

Competition

The ImageNet Large Scale Visual Recognition Challenge (ILSVRC) evaluates algorithms for object detection and image classification at large scale. One high level motivation is to allow researchers to compare progress in detection across a wider variety of objects -- taking advantage of the quite expensive labeling effort. Another motivation is to measure the progress of computer vision for large scale image indexing for retrieval and annotation.

For details about each challenge please refer to the corresponding page.

- ILSVRC 2017
- ILSVRC 2016
- ILSVRC 2015
- ILSVRC 2014
- ILSVRC 2013
- ILSVRC 2012
- ILSVRC 2011
- ILSVRC 2010

- **LeNet 1998** (the grandfather)
- **AlexNet 2012**. The first work that popularized Convolutional Networks in Computer Vision was the AlexNet, developed by Alex Krizhevsky, Ilya Sutskever and Geoff Hinton. The AlexNet was submitted to the ImageNet ILSVRC challenge in 2012 and significantly outperformed the second runner-up (top 5 error of 16% compared to runner-up with 26% error). The Network had a very similar architecture to LeNet, but was deeper, bigger, and featured Convolutional Layers stacked on top of each other (previously it was common to only have a single CONV layer always immediately followed by a POOL layer).
- **ZF Net**. ILSVRC 2013 winner
- **GoogLeNet**. ILSVRC 2014 winner
- **VGGNet**. The runner-up in ILSVRC 2014.
- **ResNet**. winner of ILSVRC 2015.

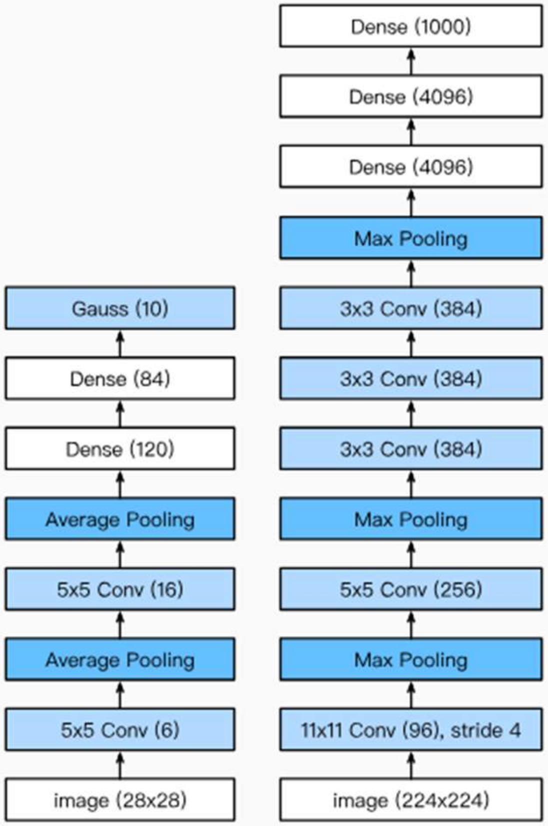
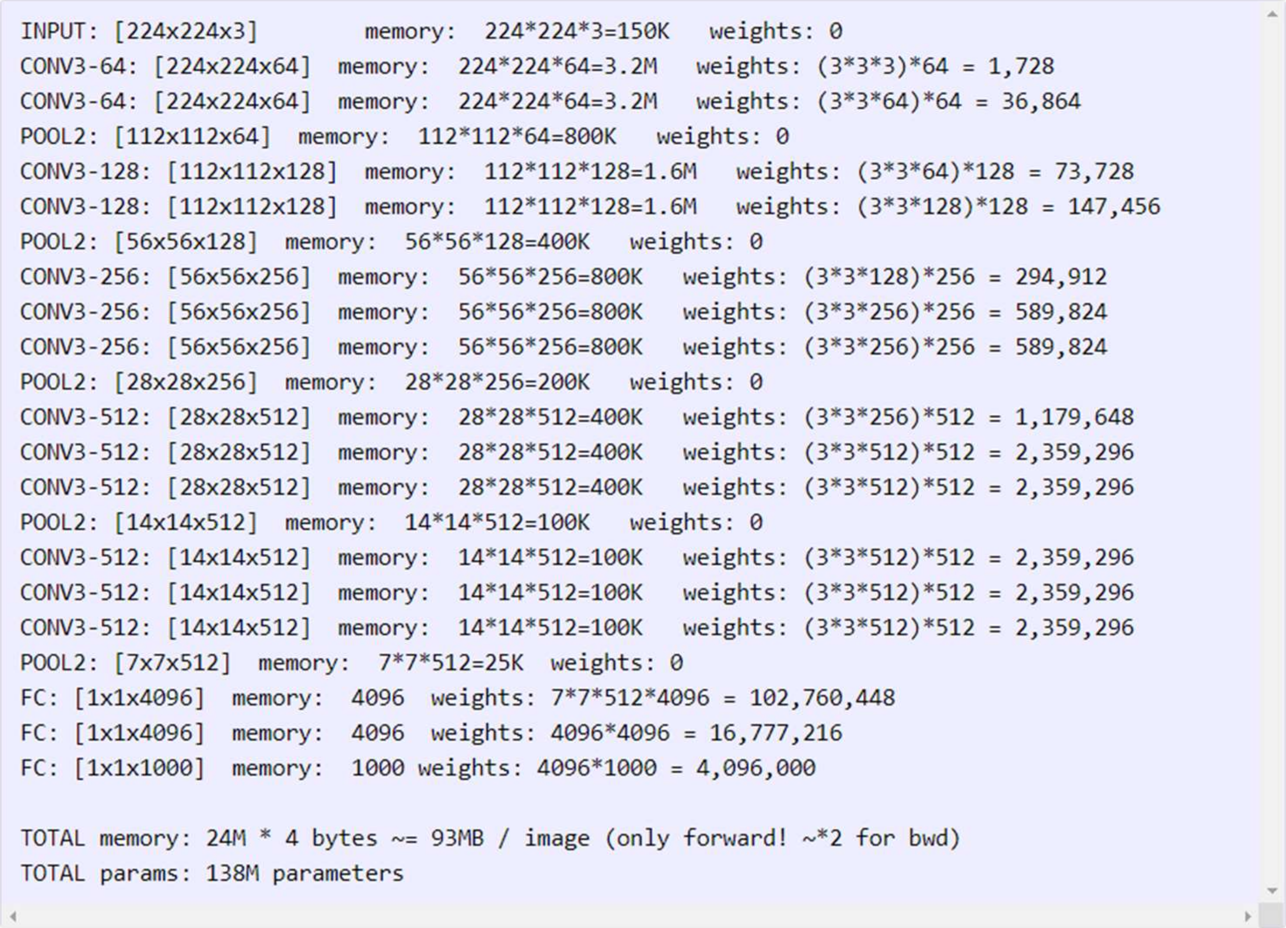


Fig. 6.10 LeNet (left) and AlexNet (right)

VGGNet in detail. Lets break down the VGGNet in more detail as a case study. The whole VGGNet is composed of CONV layers that perform 3x3 convolutions with stride 1 and pad 1, and of POOL layers that perform 2x2 max pooling with stride 2 (and no padding). We can write out the size of the representation at each step of the processing and keep track of both the representation size and the total number of weights:



```
INPUT: [224x224x3]        memory:  224*224*3=150K   weights: 0
CONV3-64: [224x224x64]  memory:  224*224*64=3.2M   weights: (3*3*3)*64 = 1,728
CONV3-64: [224x224x64]  memory:  224*224*64=3.2M   weights: (3*3*64)*64 = 36,864
POOL2: [112x112x64]  memory:  112*112*64=800K   weights: 0
CONV3-128: [112x112x128]  memory:  112*112*128=1.6M   weights: (3*3*64)*128 = 73,728
CONV3-128: [112x112x128]  memory:  112*112*128=1.6M   weights: (3*3*128)*128 = 147,456
POOL2: [56x56x128]  memory:  56*56*128=400K   weights: 0
CONV3-256: [56x56x256]  memory:  56*56*256=800K   weights: (3*3*128)*256 = 294,912
CONV3-256: [56x56x256]  memory:  56*56*256=800K   weights: (3*3*256)*256 = 589,824
CONV3-256: [56x56x256]  memory:  56*56*256=800K   weights: (3*3*256)*256 = 589,824
POOL2: [28x28x256]  memory:  28*28*256=200K   weights: 0
CONV3-512: [28x28x512]  memory:  28*28*512=400K   weights: (3*3*256)*512 = 1,179,648
CONV3-512: [28x28x512]  memory:  28*28*512=400K   weights: (3*3*512)*512 = 2,359,296
CONV3-512: [28x28x512]  memory:  28*28*512=400K   weights: (3*3*512)*512 = 2,359,296
POOL2: [14x14x512]  memory:  14*14*512=100K   weights: 0
CONV3-512: [14x14x512]  memory:  14*14*512=100K   weights: (3*3*512)*512 = 2,359,296
CONV3-512: [14x14x512]  memory:  14*14*512=100K   weights: (3*3*512)*512 = 2,359,296
CONV3-512: [14x14x512]  memory:  14*14*512=100K   weights: (3*3*512)*512 = 2,359,296
POOL2: [7x7x512]  memory:  7*7*512=25K  weights: 0
FC: [1x1x4096]  memory:  4096  weights: 7*7*512*4096 = 102,760,448
FC: [1x1x4096]  memory:  4096  weights: 4096*4096 = 16,777,216
FC: [1x1x1000]  memory:  1000 weights: 4096*1000 = 4,096,000

TOTAL memory: 24M * 4 bytes ~= 93MB / image (only forward! ~*2 for bwd)
TOTAL params: 138M parameters
```

Summary

- ConvNets stack CONV,POOL,FC layers
- Trend towards smaller filters and deeper architectures
- Trend towards getting rid of POOL/FC layers (just CONV)
- Typical architectures look like

  [(CONV-RELU)*N-POOL?]*M-(FC-RELU)*K,SOFTMAX

  where N is usually up to ~5, M is large, 0 <= K <= 2.

  - but recent advances such as ResNet/GoogLeNet challenge this paradigm

Transfer Learning

"You need a lot of a data if you want to train/use CNNs"

Transfer Learning

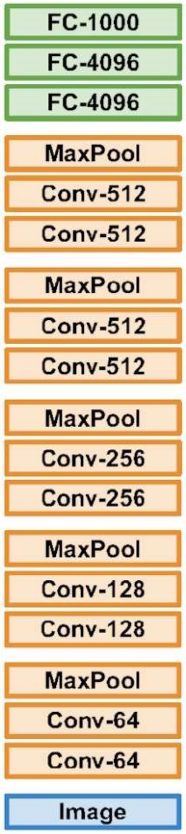
“You need a lot of a data if you want to train/use CNNs”

BUSTED

Transfer Learning with CNNs

Donahue et al, "DeCAF: A Deep Convolutional Activation Feature for Generic Visual Recognition", ICML 2014
Razavian et al, "CNN Features Off-the-Shelf: An Astounding Baseline for Recognition", CVPR Workshops 2014

1. Train on Imagenet

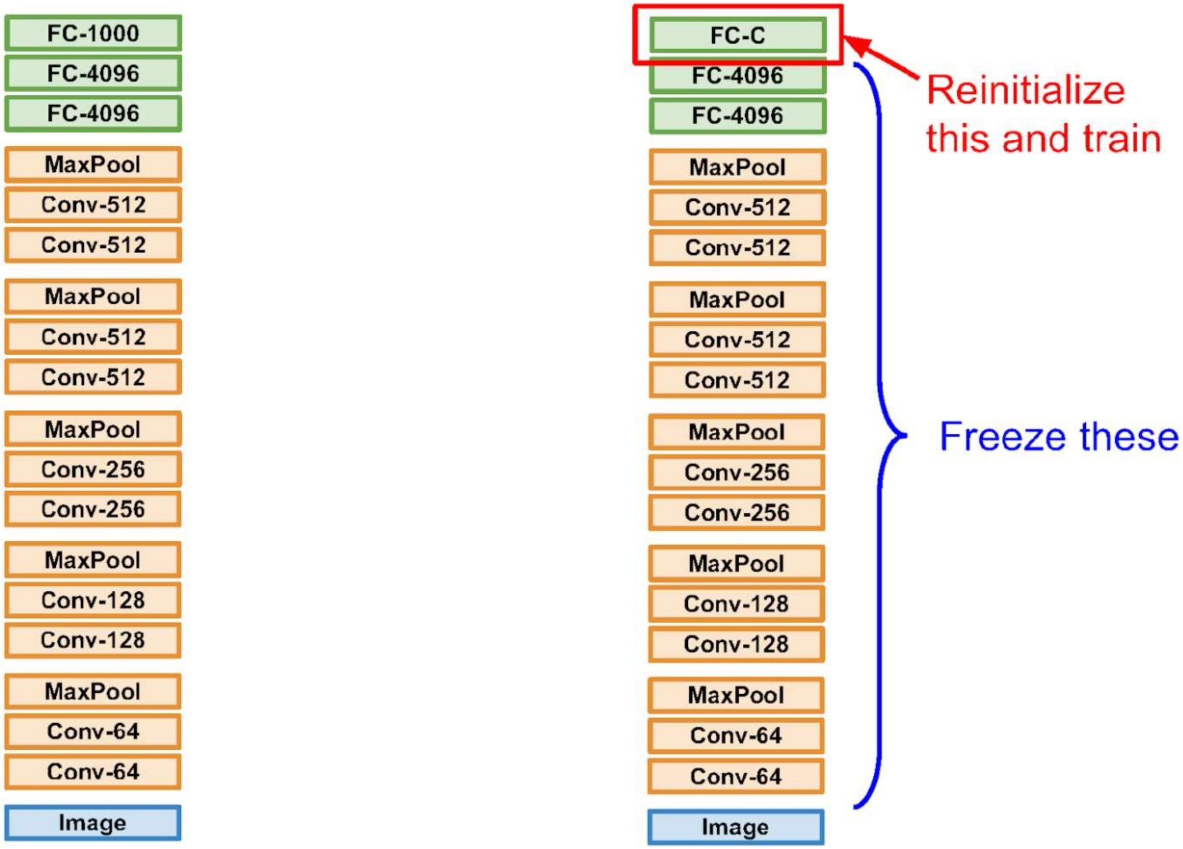


Transfer Learning with CNNs

Donahue et al, "DeCAF: A Deep Convolutional Activation Feature for Generic Visual Recognition", ICML 2014
Razavian et al, "CNN Features Off-the-Shelf: An Astounding Baseline for Recognition", CVPR Workshops 2014

1. Train on Imagenet

2. Small Dataset (C classes)



FC-1000
FC-4096
FC-4096
MaxPool
Conv-512
Conv-512
MaxPool
Conv-512
Conv-512
MaxPool
Conv-256
Conv-256
MaxPool
Conv-128
Conv-128
MaxPool
Conv-64
Conv-64
Image

FC-C
FC-4096
FC-4096
MaxPool
Conv-512
Conv-512
MaxPool
Conv-512
Conv-512
MaxPool
Conv-256
Conv-256
MaxPool
Conv-128
Conv-128
MaxPool
Conv-64
Conv-64
Image

Reinitialize this and train

Freeze these

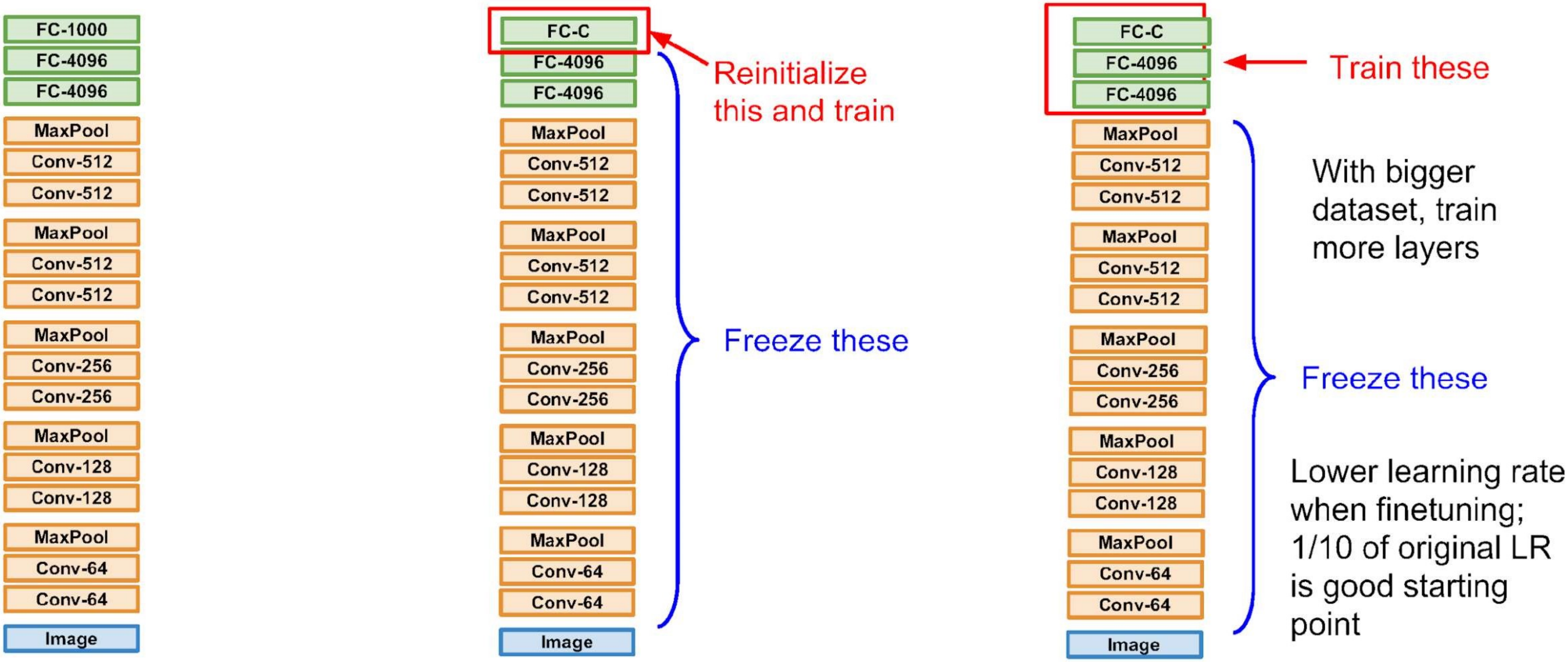
Transfer Learning with CNNs

Donahue et al, "DeCAF: A Deep Convolutional Activation Feature for Generic Visual Recognition", ICML 2014
Razavian et al, "CNN Features Off-the-Shelf: An Astounding Baseline for Recognition", CVPR Workshops 2014

1. Train on Imagenet

2. Small Dataset (C classes)

3. Bigger dataset



Reinitialize this and train

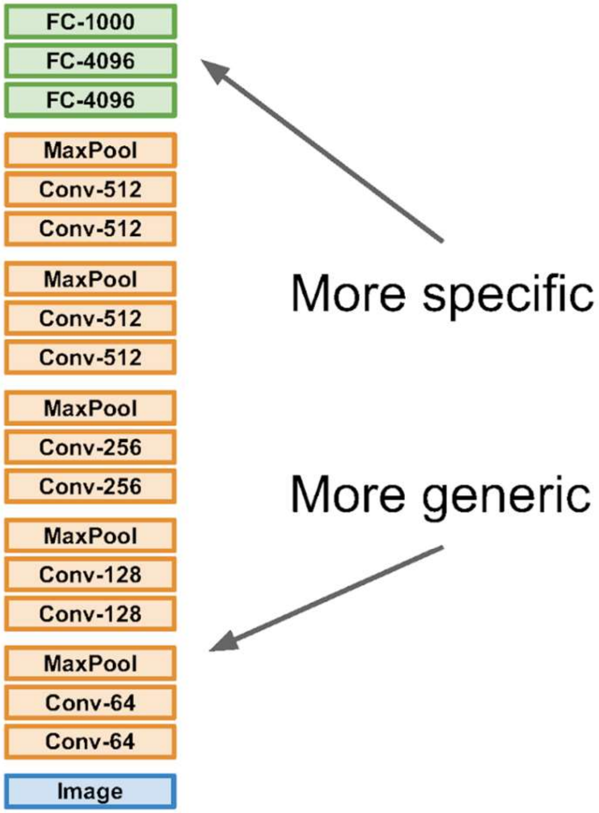
Freeze these

Train these

With bigger dataset, train more layers

Freeze these

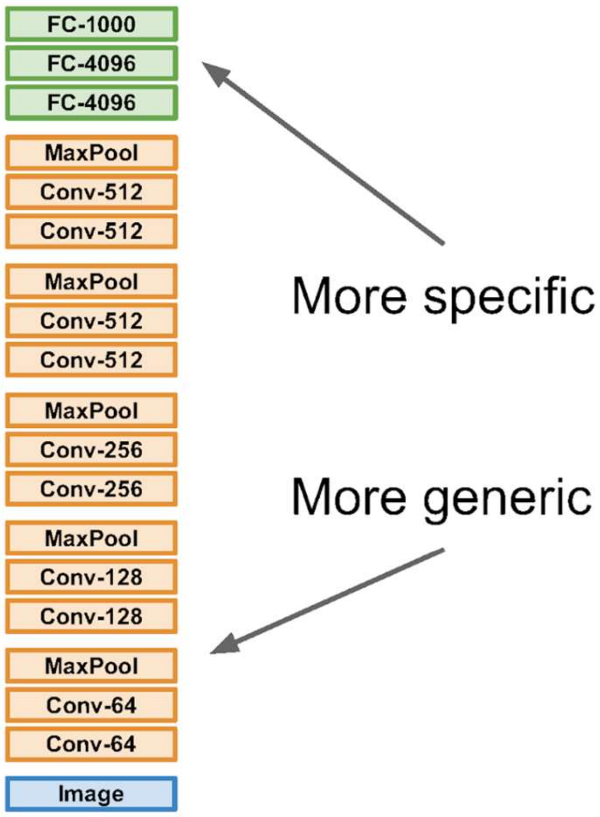
Lower learning rate when finetuning; 1/10 of original LR is good starting point



More specific

More generic

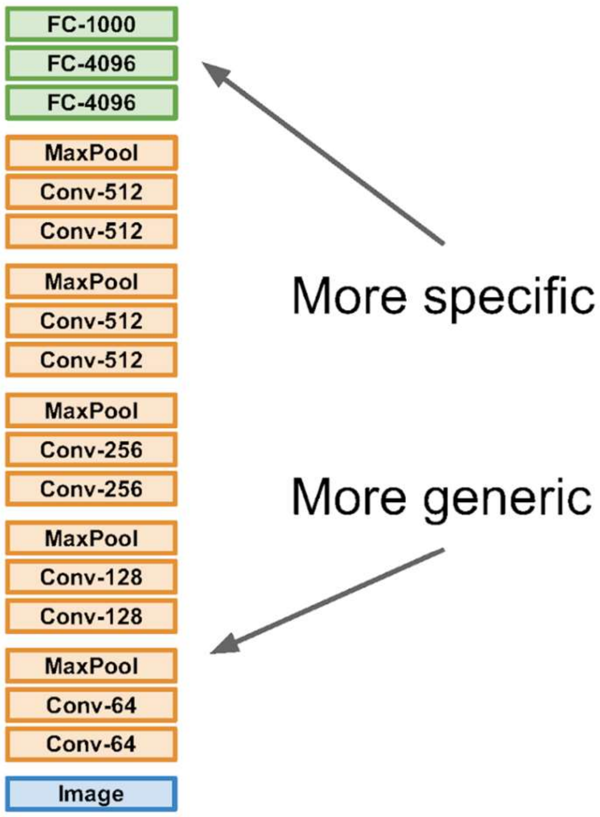
	very similar dataset	**very different dataset**
very little data	?	?
quite a lot of data	?	?



More specific

More generic

	very similar dataset	**very different dataset**
very little data	Use Linear Classifier on top layer	?
quite a lot of data	Finetune a few layers	?



FC-1000
FC-4096
FC-4096
MaxPool
Conv-512
Conv-512
MaxPool
Conv-512
Conv-512
MaxPool
Conv-256
Conv-256
MaxPool
Conv-128
Conv-128
MaxPool
Conv-64
Conv-64
Image

More specific

More generic

	very similar dataset	**very different dataset**
very little data	Use Linear Classifier on top layer	You're in trouble… Try linear classifier from different stages
quite a lot of data	Finetune a few layers	Finetune a larger number of layers

Transfer learning with CNNs is pervasive…

(it's the norm, not an exception)

Object Detection
(Fast R-CNN)

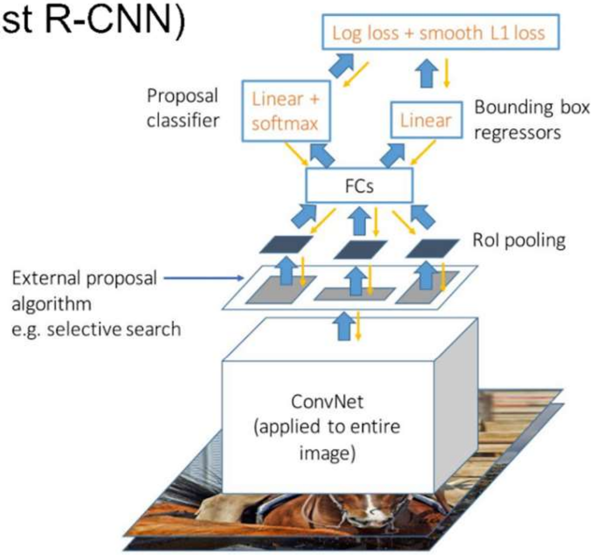
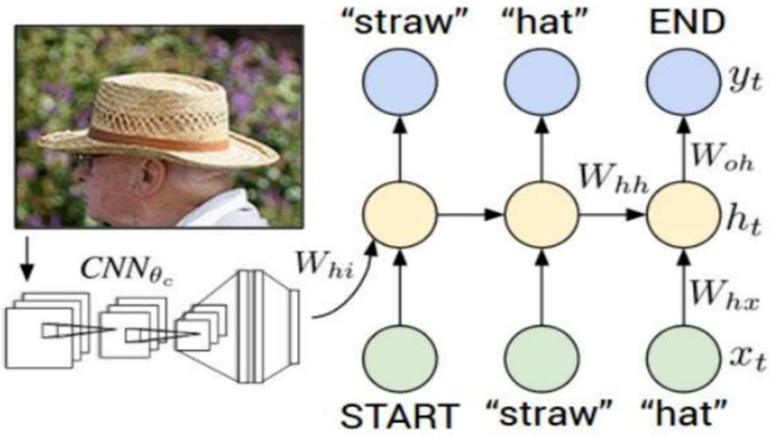

Log loss + smooth L1 loss
Proposal classifier
Linear + softmax
Linear
Bounding box regressors
FCs
RoI pooling
External proposal algorithm e.g. selective search
ConvNet (applied to entire image)

Image Captioning: CNN + RNN



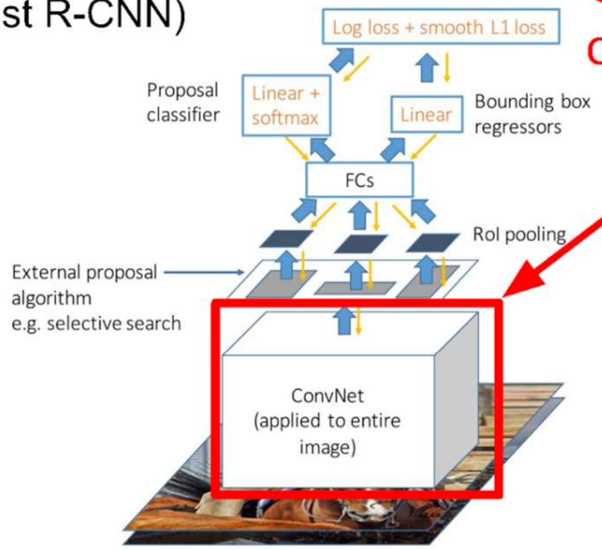
Girshick, "Fast R-CNN", ICCV 2015
Figure copyright Ross Girshick, 2015. Reproduced with permission.

Karpathy and Fei-Fei, "Deep Visual-Semantic Alignments for Generating Image Descriptions", CVPR 2015
Figure copyright IEEE, 2015. Reproduced for educational purposes.

Transfer learning with CNNs is pervasive…

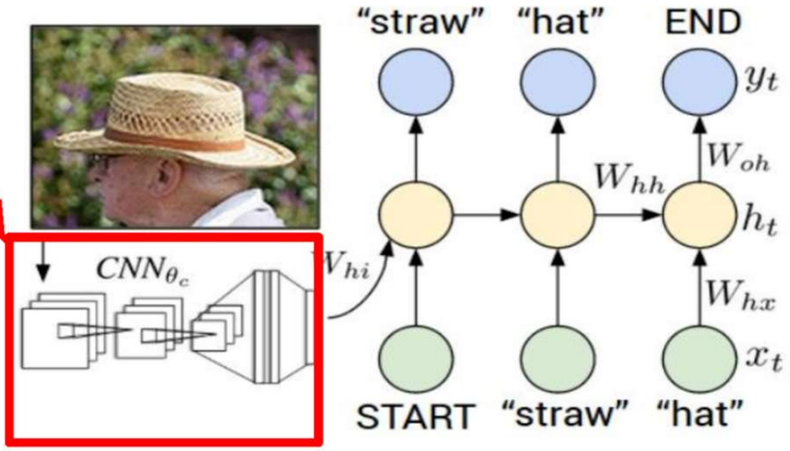
(it’s the norm, not an exception)

Object Detection
(Fast R-CNN)


Log loss + smooth L1 loss
Proposal classifier
Linear + softmax
Linear
Bounding box regressors
FCs
RoI pooling
External proposal algorithm e.g. selective search
ConvNet (applied to entire image)

CNN pretrained on ImageNet

Image Captioning: CNN + RNN

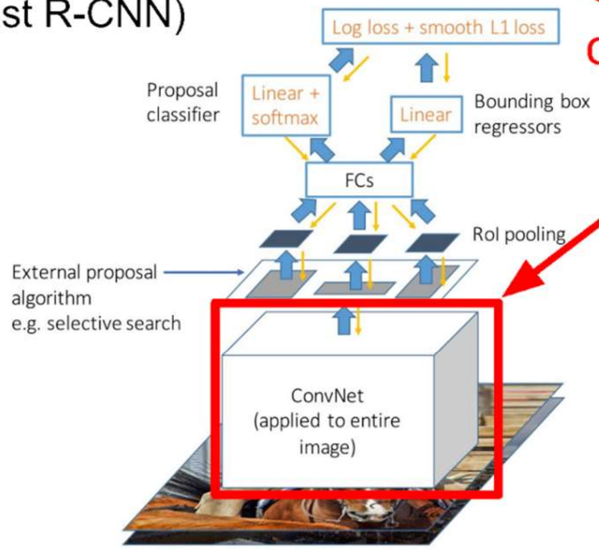


Girshick, “Fast R-CNN”, ICCV 2015
Figure copyright Ross Girshick, 2015. Reproduced with permission.

Karpathy and Fei-Fei, “Deep Visual-Semantic Alignments for Generating Image Descriptions”, CVPR 2015
Figure copyright IEEE, 2015. Reproduced for educational purposes.

Transfer learning with CNNs is pervasive…
(it’s the norm, not an exception)

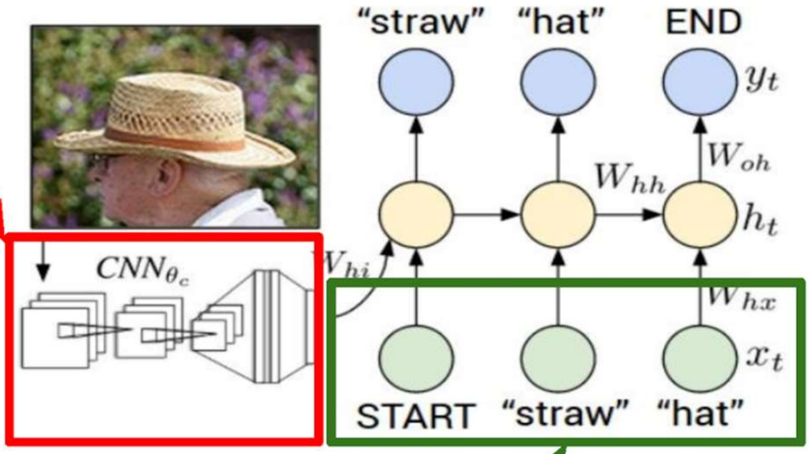
Object Detection
(Fast R-CNN)



Log loss + smooth L1 loss
Proposal classifier
Linear + softmax
Linear
Bounding box regressors
FCs
RoI pooling
External proposal algorithm e.g. selective search
ConvNet (applied to entire image)

CNN pretrained on ImageNet

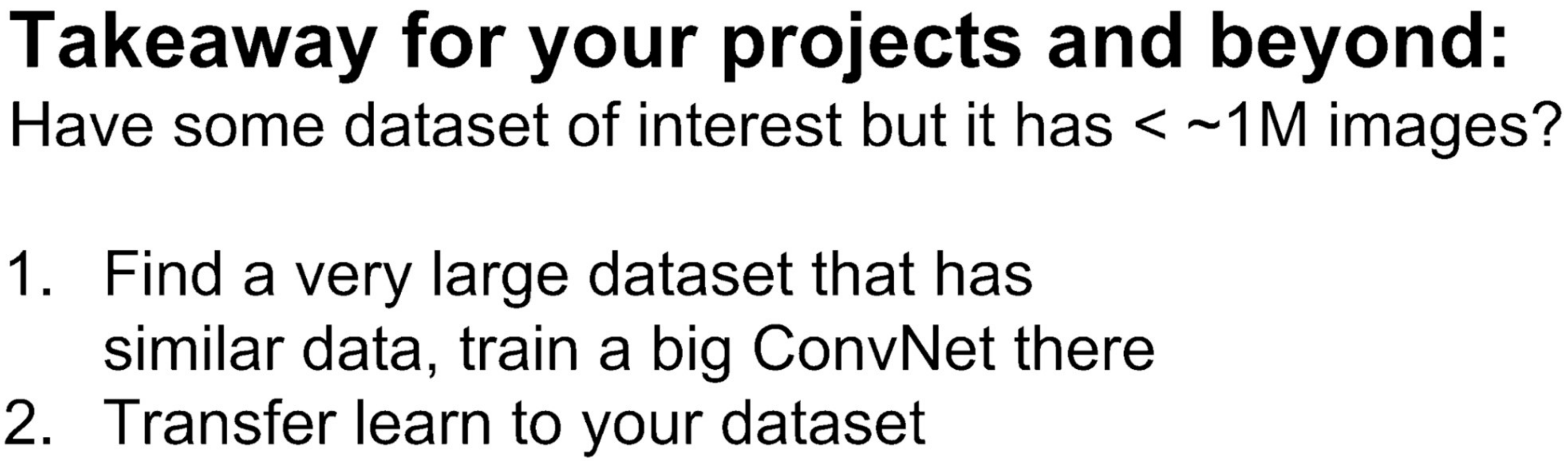
Image Captioning: CNN + RNN



Word vectors pretrained with word2vec

Girshick, “Fast R-CNN”, ICCV 2015
Figure copyright Ross Girshick, 2015. Reproduced with permission.

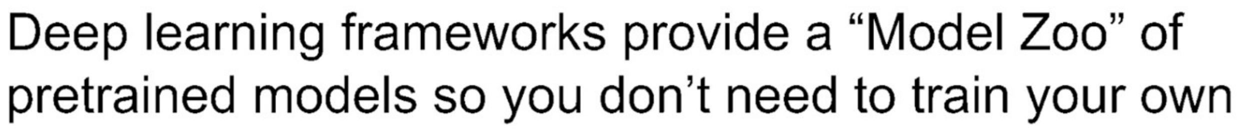
Karpathy and Fei-Fei, “Deep Visual-Semantic Alignments for Generating Image Descriptions”, CVPR 2015
Figure copyright IEEE, 2015. Reproduced for educational purposes.



Takeaway for your projects and beyond:

Have some dataset of interest but it has < ~1M images?

1. Find a very large dataset that has similar data, train a big ConvNet there
2. Transfer learn to your dataset



Deep learning frameworks provide a "Model Zoo" of pretrained models so you don't need to train your own

Keras https://keras.io/applications/
PyTorch https://pytorch.org/docs/stable/torchvision/models.html
MXNet (Gluon) https://mxnet.apache.org/api/python/gluon/model_zoo.html
TensorFlow https://github.com/tensorflow/models
Caffe https://github.com/BVLC/caffe/wiki/Model-Zoo

Επιλεγμένες πηγές και πρακτικά παραδείγματα

Convolutional Neural Networks

Stanford Intro to CNNs
Dive into Deep Learning
Περιέχουν εισαγωγή στα CNNs και παρουσίαση των διαφόρων αρχιτεκτονικών ConvNets του Imagenet
Ένα απλό παράδειγμα CNN στο MNIST με Keras. Η είσοδος (εικόνα) έχει ένα μόνο επίπεδο καθώς είναι grayscale. Keras CNN στο CIFAR-10 (πατήστε “Next” για να δείτε το ίδιο πρόβλημα με data augmentation, με το ResNet καθώς και μια οπτικοποίηση των συνελικτικών φίλτρων.

Transfer Learning

Εισαγωγή του Stanford
Tutorial σε Pytorch που δείχνει δύο διαφορετικές στρατηγικές training μετά το transfer learning. Εισάγουμε το ResNet18 και κάνουμε train πρώτα σε ολόκληρο το δίκτυο και μετά μόνο στο τελικό fully connected επίπεδο

Βασική ιστορική βιβλιογραφία

- LeCun, Y., Bottou, L., Bengio, Y., & Haffner, P. (1998). Gradient-based learning applied to document recognition. Proceedings of the IEEE, 86(11), 2278-2324.
- Hinton, G. E., & Salakhutdinov, R. R. (2006). Reducing the dimensionality of data with neural networks. science, 313(5786), 504-507.
- Hinton, G. E., Osindero, S., & Teh, Y. W. (2006). A fast learning algorithm for deep belief nets. Neural computation, 18(7), 1527-1554.
- Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. In Advances in neural information processing systems (pp. 1097-1105).

στον φάκελο bibliography. Μπορείτε να βρείτε τις βιβλιογραφικές αναφορές και τα papers για τις διάφορες αρχιτεκτονικές του ImageNet στις εισαγωγές του Stanford και του Dive (προηγούμενο slide)

DEEP LEARNING

- ImageNet Challenge & Breakthrough Deep Neural Networks
  1. ImageNet (Dataset & Challenge)
  2. AlexNet (first DNN winner of ImageNet)
  3. VGGNet (Deeper DNN, runner up in ImageNet)

- Going Deeper: Is this the solution?

- ResNet: the solution

- References

ImageNet Dataset

- large annotated photographs’ dataset for computer vision research
- goal: resource for promoting research and development of improved methods for computer vision [1]

ImageNet Large Scale Visual Recognition Challenge (ILSVRC)

- annual competition held between 2010 and 2017 [2]

- challenge tasks use subsets (approximately 1.2 million images) of the ImageNet dataset for:

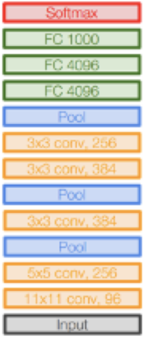
i) "*image classification*": assigning a class label to each image based on the main object in the photograph (among 1,000 object classes)

ii) "*object detection*": localizing the objects within each photograph

AlexNet: First Deep Neural Network Winner of ILSVRC 2012

- In 2012, AlexNet [3] significantly outperformed all prior competitors (error 15.3%; prior competitors’ error was 25.7% and 28.2%)

- The runner up was not a deep learning method (error 26.2%)

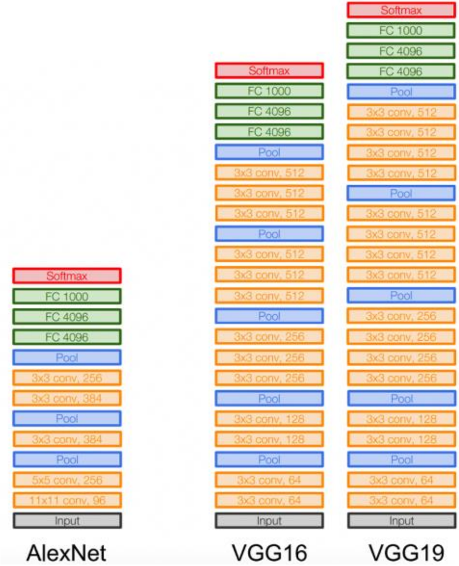


AlexNet

VGG: Runner-Up of ILSVRC 2014, Deeper than AlexNet

- was the runner-up at the ILSVRC 2014
- achieved error 7.3% (vs 15.3% of AlexNet)
- has 16 or 19 layers and is deeper than AlexNet (8 layers)
- however, VGG [4] consists of 138 million parameters (AlexNet consists of 61 million parameters)

VGG & AlexNet Architectures



Softmax
FC 1000
FC 4096
FC 4096
Pool
3x3 conv, 256
3x3 conv, 384
Pool
3x3 conv, 384
Pool
5x5 conv, 256
11x11 conv, 96
Input

AlexNet

Softmax
FC 1000
FC 4096
FC 4096
Pool
3x3 conv, 512
3x3 conv, 512
3x3 conv, 512
Pool
3x3 conv, 512
3x3 conv, 512
3x3 conv, 512
Pool
3x3 conv, 256
3x3 conv, 256
3x3 conv, 256
Pool
3x3 conv, 128
3x3 conv, 128
Pool
3x3 conv, 64
3x3 conv, 64
Input

VGG16

Softmax
FC 1000
FC 4096
FC 4096
Pool
3x3 conv, 512
3x3 conv, 512
3x3 conv, 512
3x3 conv, 512
Pool
3x3 conv, 512
3x3 conv, 512
3x3 conv, 512
3x3 conv, 512
Pool
3x3 conv, 256
3x3 conv, 256
3x3 conv, 256
3x3 conv, 256
Pool
3x3 conv, 128
3x3 conv, 128
Pool
3x3 conv, 64
3x3 conv, 64
Input

VGG19

VGG/AlexNet: Conclusions

Should we make a Neural Network (NN) deeper and why?

more layers → more high-level features → better understanding of data and better prediction

Neural Networks → make them deeper → problem solved?

If yes, how deep?

- ImageNet Challenge & Breakthrough Deep Neural Networks
- Going Deeper: Is this the solution?
  1. Issues to consider
  2. Specific Problems:
     - Vanishing Gradients
       (definition, cause, significance, comparison shallow & deep NN, solutions)
     - Degradation
       (definition, analogy, not overfitting)
- ResNet: the solution
- References

Depends on:

- The complexity of the task at hand
- Available computational capacity during training
- Available computational capacity during inference

If the task needs a lot of parameters:

- Can we train very deep networks efficiently using current optimisation solvers?
- Is training a better model as simple as adding more and more layers?

1. Vanishing Gradients

1) Vanishing Gradients: the problem

- During each iteration of standard neural network training, all weights receive an update proportional to the partial derivative (gradient) of the cost function with respect to their current value
- If the gradient is very small then the weights will not change effectively
- As a consequence, this may completely stop the neural network from further training
- This is called the vanishing gradient problem.

Vanishing Gradients: the causes

The Vanishing Gradient Problem is met in Neural Networks:

- with certain activation functions
- trained with gradient based methods (e.g Back Propagation [5])

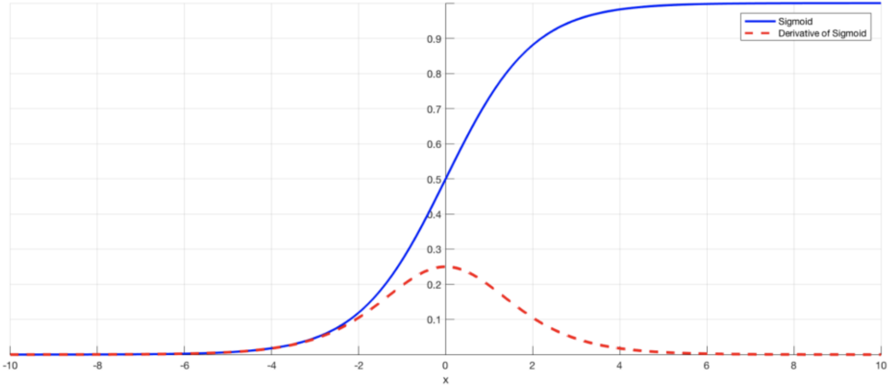
It gets worse as the number of layers in the neural network increases.

Vanishing Gradients: caused by activation function (1)

Vanishing gradient problem depends on the choice of the activation function:

- many common activation functions (e.g., sigmoid [6], tanh [7]) 'squash' their input into a very small output range in a very non-linear fashion
- for example, sigmoid maps the real number line onto a "small" range of [0, 1] → large regions of the input space are mapped to an extremely small range
- in these regions of the input space, even a large change in the input will produce a small change in the output - the gradient is small.

Vanishing Gradients: caused by activation function (2)



Vanishing Gradients: caused by gradient descent training

Gradients of neural networks are usually computed using backpropagation:

- backpropagation finds the derivatives of the network by moving layer by layer from the final to the initial one
- using the chain rule, the derivatives of each layer are multiplied down the network (from the final layer to the initial) to compute the derivatives of the initial layers
- when *n* hidden layers use an activation like the sigmoid function, *n* small derivatives are multiplied together

Vanishing Gradients: shallow vs deep networks

- thus, the gradient decreases exponentially as we propagate down to the initial layers
- a small gradient means that weights & biases of initial layers will not be updated effectively during training
- since initial layers are often crucial to recognise the core elements of input data, this can lead to overall network inaccuracy.

For shallow networks, with only a few layers that use these activations, this isn't a big problem. However, when more layers are used, this can cause the gradient to be too small for training to work effectively.

Vanishing Gradients: Solution 1: Use ReLU

Use activation functions which don't 'squash' the input space into a small region.

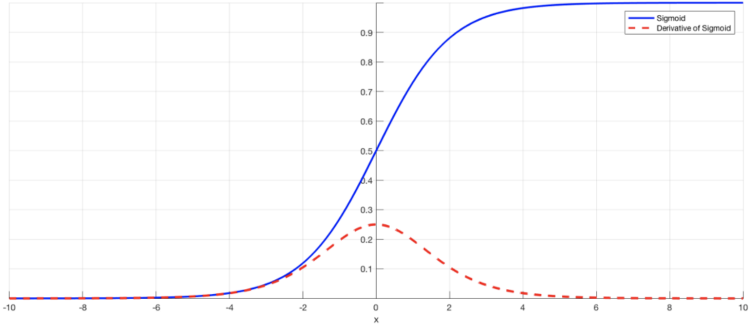
A popular choice is Rectified Linear Unit (ReLU) [8] which maps x to $max(0,x)$.

Vanishing Gradients: Solution 2: Use Batch Normalisation (1)

Problem: when a large input space is mapped to a small one, causing the derivatives to disappear.

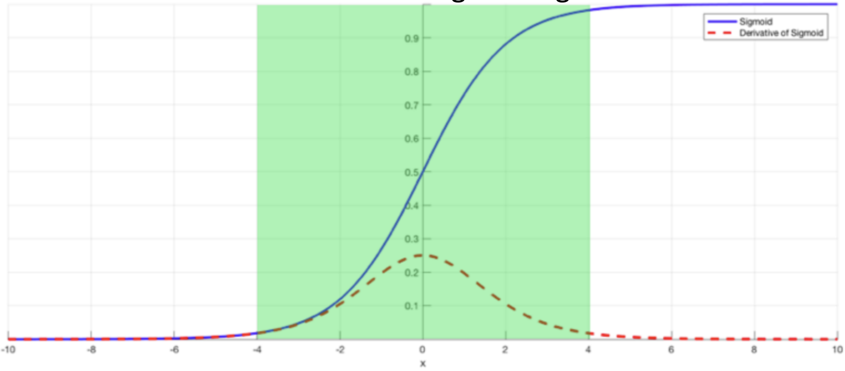
sigmoid activation function; $x = wu + b$ for a neuron anywhere in the hidden layers of a NN; u: layer's input; w: weights matrix; b : bias vector

x = very big/small → gradient = 0



Vanishing Gradients: Solution 2: Use Batch Normalisation (2)

Batch Normalisation: Step 1: normalise the input by subtracting its mean and dividing by its standard deviation (ensures zero mean and unit variance)



x doesn’t reach outer edges of sigmoid

Vanishing Gradients: Solution 2: Use Batch Normalisation (4)

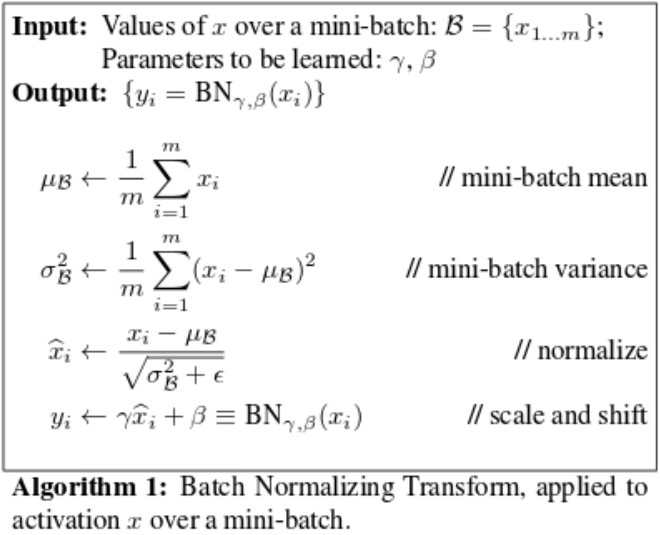
Batch Normalisation [9]: Step 2: the normalized output of Step 1 is multiplied by a “standard deviation” parameter (gamma; γ) and a “mean” parameter (beta; β) is added to the product

- these two parameters are optimised during network training

- Batch Normalisation increases stability of a Neural Network & speeds up training

Vanishing Gradients: Solution 2: Use Batch Normalisation (5)

Algorithm:



```
Input: Values of $x$ over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
       Parameters to be learned: $\gamma$, $\beta$
Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

    $\mu_\mathcal{B} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i$                          // mini-batch mean
    $\sigma_\mathcal{B}^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_\mathcal{B})^2$          // mini-batch variance
    $\widehat{x}_i \leftarrow \frac{x_i - \mu_\mathcal{B}}{\sqrt{\sigma_\mathcal{B}^2 + \epsilon}}$                  // normalize
    $y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i)$          // scale and shift
```

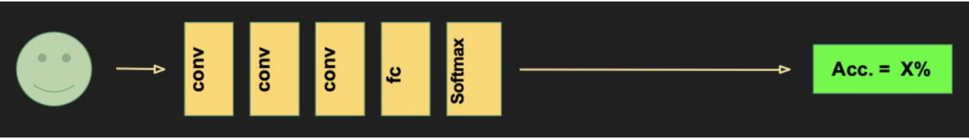
Algorithm 1: Batch Normalizing Transform, applied to activation x over a mini-batch.

1. Vanishing gradients
2. Degradation problem [10]

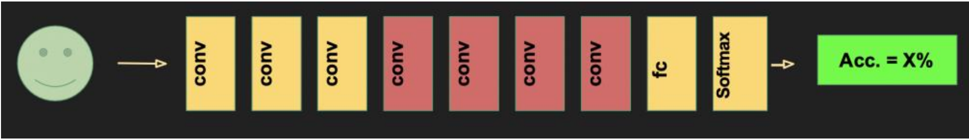
Degradation problem: Definition (1)

Image Classification Problem

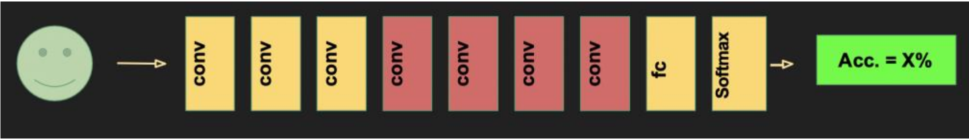
Consider a network having n layers. This network produces some error/accuracy.



Now consider a deeper network with m layers ($m>n$).

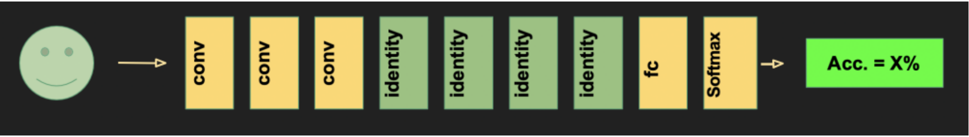


Degradation problem: Definition (2)



When we train this network, we expect it to perform *__at least as well as__* the shallower network. Why?

Replace the first n layers of the deep network with the trained n layers of the shallower network. Now replace the remaining n−m layers in the deeper network with an identity mapping (these layers simply output what is fed into them).

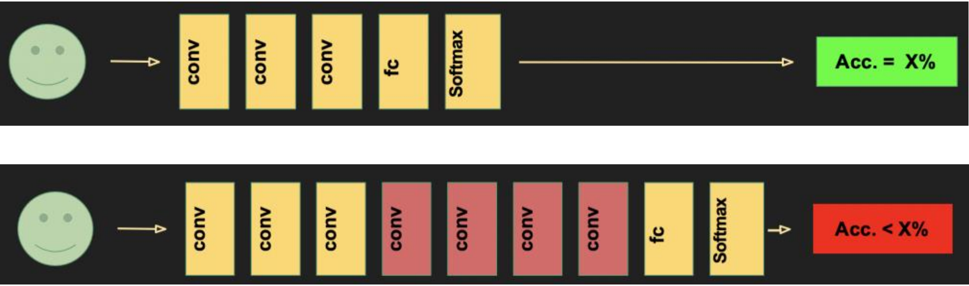


Degradation problem: Definition (3)

Thus, our deeper model can easily learn the shallower model’s representation.

If there exists a more complex representation of data, we expect the deep model to learn this.

Degradation problem: Definition – In Practice



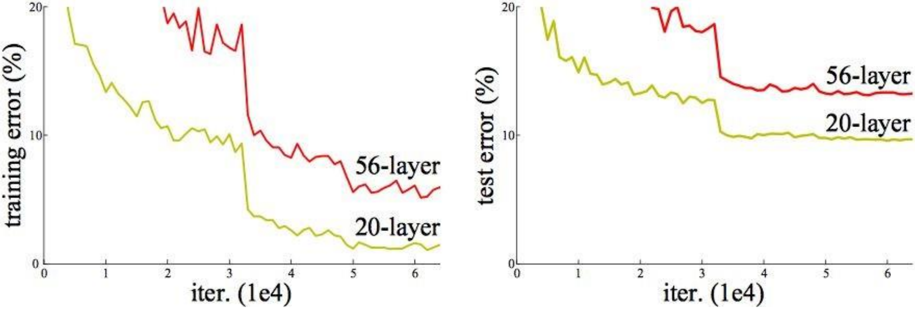
Degradation problem: Definition – Overfitting?

- task is to predict if an image shows a balloon or not
- train a model using a dataset containing many blue colored balloons (and other irrelevant objects)
- test the model on the original dataset: it gives 99% accuracy!
- test the model on a new (“unseen”) dataset containing yellow colored balloons: it gives 20% accuracy!

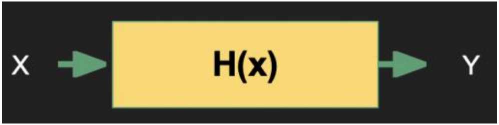
Our model doesn’t *generalise* well from our training data to unseen data. This is known as overfitting.

A model that has learned the noise instead of the signal is considered “overfit” because it fits the training dataset but has poor fit with new datasets.

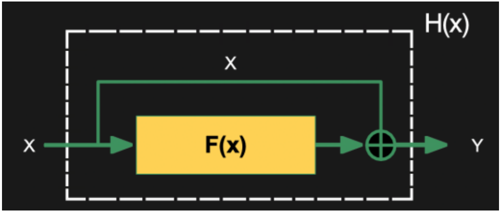
Degradation problem: Definition – Overfitting? No



Residual Learning



H(x) is the true mapping function we want to learn



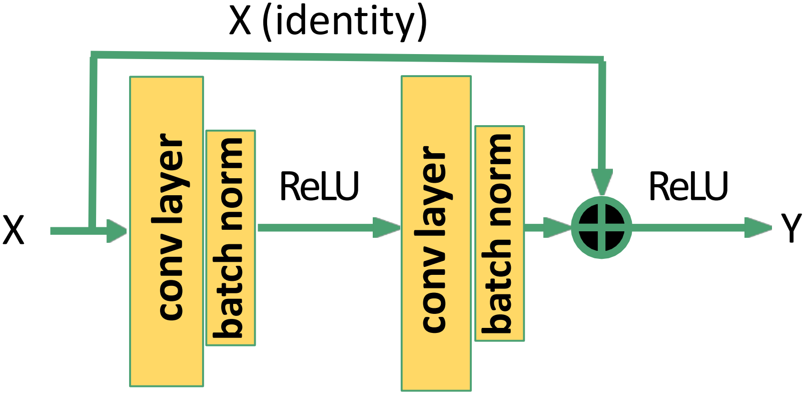
New representation F(x)

$$\boxed{F(x) := H(x) - x}$$

Residual Learning [10]

If F(x) = 0 → identity mapping; if that is a solution the network will be able to find it

Residual Block (1)



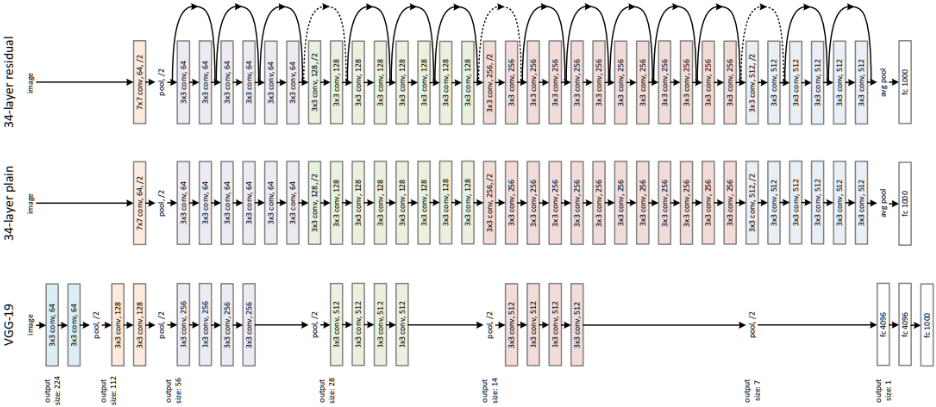
Residual Block (2)

- with this approach, the network will decide how deep it needs to be

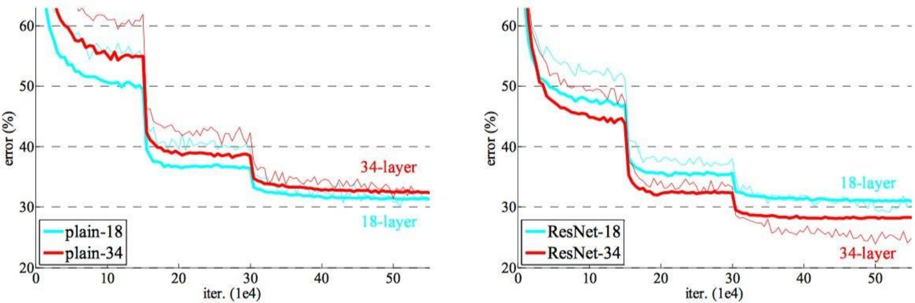
- the identity connections introduce no new parameter to the network architecture, hence it will not add any computational burden

- this method allows us to design deeper networks in order to deal with much complicated problems and tasks

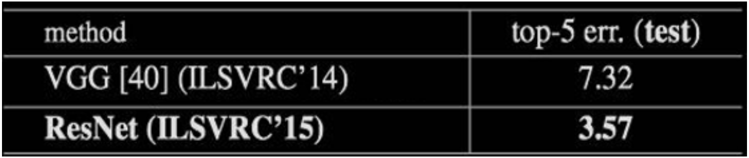
ResNet-34: Plain vs With Skip Connections vs VGG (4)



No Degradation for ResNet on ImageNet



Results on ImageNet



method	top-5 err. (**test**)
VGG [40] (ILSVRC'14)	7.32
ResNet (ILSVRC'15)	**3.57**

1. J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li and L. Fei-Fei. **ImageNet: A Large-Scale Hierarchical Image Database.** *IEEE Computer Vision and Pattern Recognition (CVPR), 2009*

2. O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg and L. Fei-Fei. **ImageNet Large Scale Visual Recognition Challenge**. *International Journal of Computer Vision (IJCV),* 2015

3. A. Krizhevsky, I. Sutskever, and G. E Hinton. **Imagenet classification with deep convolutional neural networks**. *Advances in neural information processing systems (NIPS)*, 2012

4. K. Simonyan and A. Zisserman. **Very deep convolutional networks for large-scale image recognition.** *International Conference on Learning Representations (ICLR)*, 2015.

5. https://www.deeplearningbook.org/contents/mlp.html#pf25

6. https://en.wikipedia.org/wiki/Sigmoid_function#cite_note-:0-1

7. https://en.wikipedia.org/wiki/Activation_function

8. https://en.wikipedia.org/wiki/Rectifier_(neural_networks)

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

10. K. He, X. Zhang, S. Ren, and J. Sun. **Deep residual learning for image recognition.** *IEEE Computer Vision and Pattern Recognition (CVPR), 2016.*