CS5242 : Neural Networks and Deep Learning

Lecture 8: Convolutional Neural Networks Implementation

Semester 1 2021/22

Xavier Bresson

https://twitter.com/xbresson

Department of Computer Science
National University of Singapore (NUS)

Outline

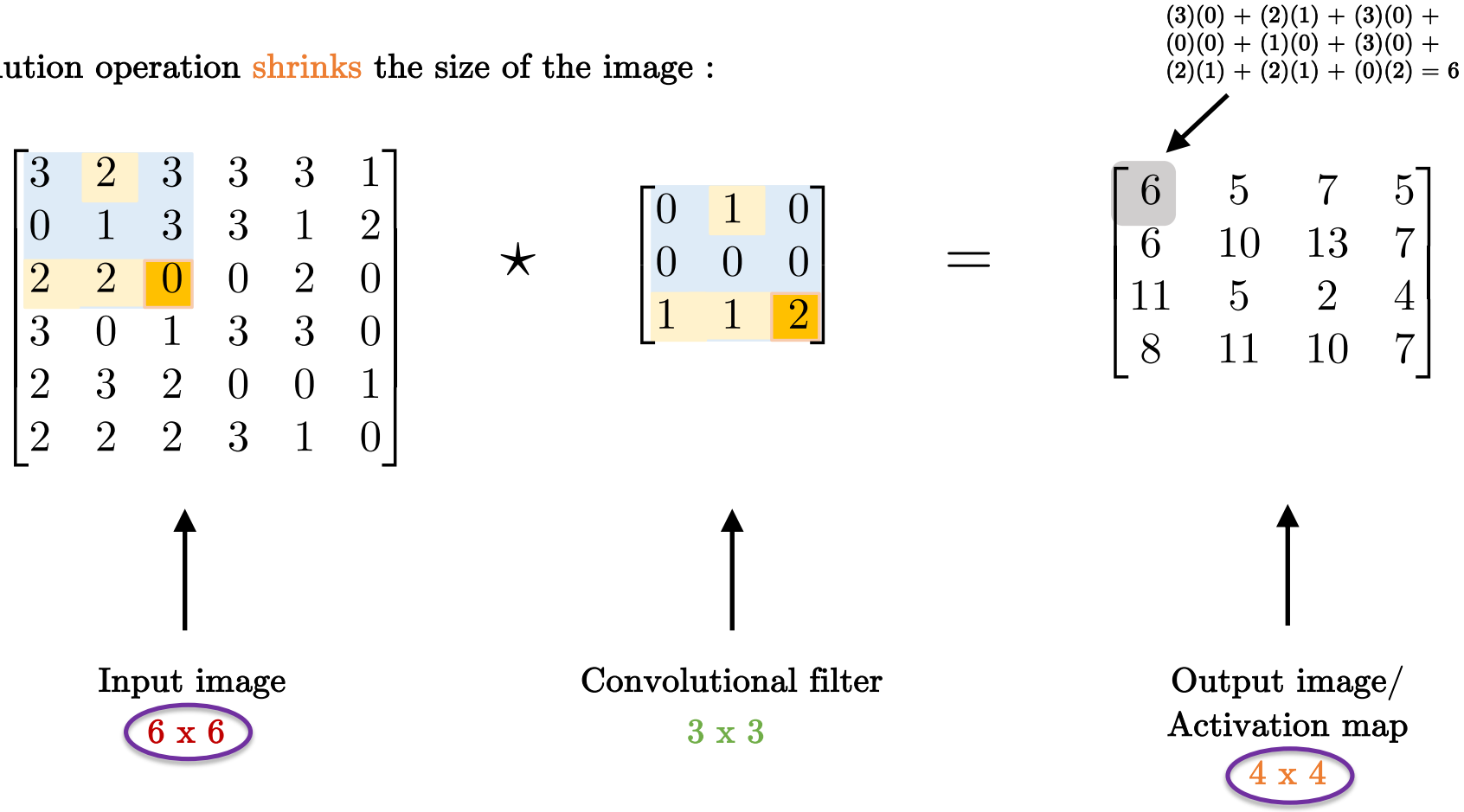
- Padding
- Stride
- Multi-dimensional input
- Convolution as linear operation
- Depth
- Convolutional layer with PyTorch
- Pooling layer with PyTorch
- LeNet5 architecture
- VGG architecture
- Case studies

Outline

- Padding
- Stride
- Multi-dimensional input
- Convolution as linear operation
- Depth
- Convolutional layer with PyTorch
- Pooling layer with PyTorch
- LeNet5 architecture
- VGG architecture
- Case studies

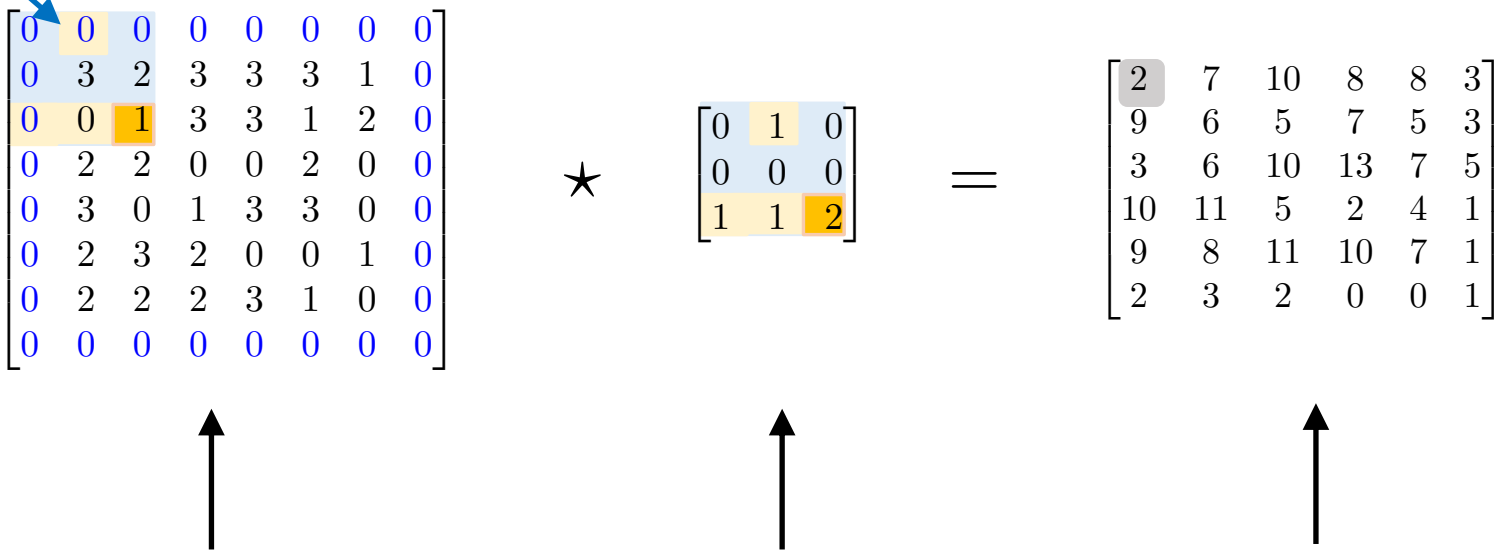
Padding

- Convolution operation shrinks the size of the image :



Padding

- Shrinking the image size is an issue if we want to stack many layers (size will reduce to 1x1 quickly).
  - Padding input image with zeros allows to have the same size for the input and the output.



Input image 6 x 6 padded with zeros

8 x 8

Convolutional filter

3 x 3

Activation map/ output

6 x 6

Padding

- Padding size depends on the filter size :
  - When using 5x5 filters (radius=2), image must be padded with 2 layers of zeros.

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 3 & 2 & 3 & 3 & 3 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 3 & 3 & 1 & 2 & 0 & 0 \\ 0 & 0 & 2 & 2 & 0 & 0 & 2 & 0 & 0 & 0 \\ 0 & 0 & 3 & 0 & 1 & 3 & 3 & 0 & 0 & 0 \\ 0 & 0 & 2 & 3 & 2 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 2 & 2 & 2 & 3 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \star \begin{bmatrix} 1 & 2 & 1 & 1 & 2 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 \\ 1 & 1 & 1 & 2 & 1 \\ 1 & 2 & 1 & 1 & 2 \end{bmatrix} = \begin{bmatrix} 2 & 7 & 10 & 8 & 8 & 3 \\ 9 & 6 & 5 & 7 & 5 & 3 \\ 3 & 6 & 10 & 13 & 7 & 5 \\ 10 & 11 & 5 & 2 & 4 & 1 \\ 9 & 8 & 11 & 10 & 7 & 1 \\ 2 & 3 & 2 & 0 & 0 & 1 \end{bmatrix}$$

  - When using 7x7 filters (radius=3), image must be padded with 3 layers of zeros.
  - Etc.

Outline

- Padding
- Stride
- Multi-dimensional input
- Convolution as linear operation
- Depth
- Convolutional layer with PyTorch
- Pooling layer with PyTorch
- LeNet5 architecture
- VGG architecture
- Case studies

Strided convolution

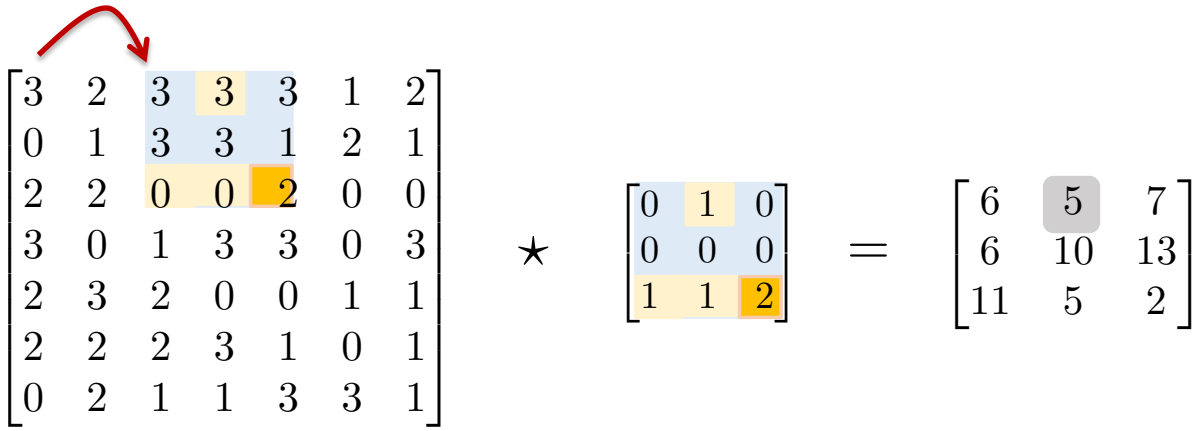
- Strided convolution can be useful for large images (e.g. satellite images 10,000 x 10,000) :

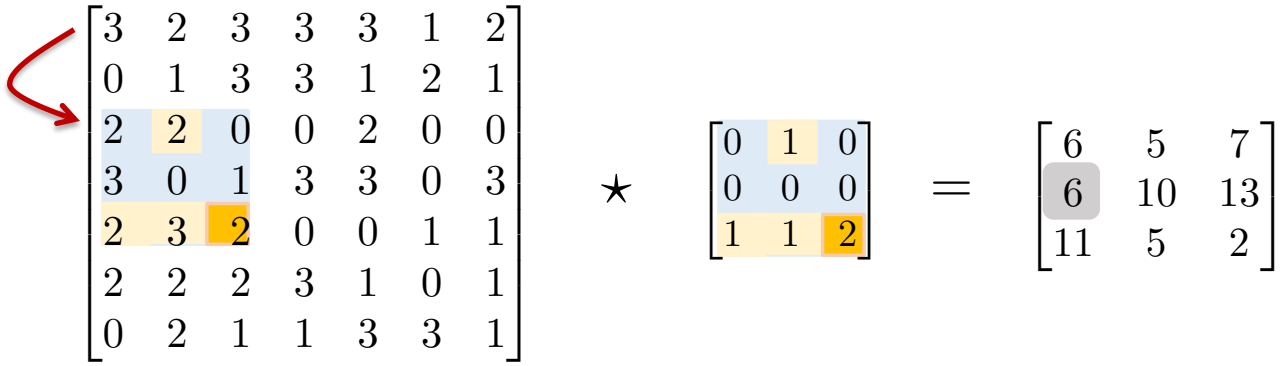
Consider a stride = 2

$$\begin{bmatrix} 3 & 2 & 3 & 3 & 3 & 1 & 2 \\ 0 & 1 & 3 & 3 & 1 & 2 & 1 \\ 2 & 2 & 0 & 0 & 2 & 0 & 0 \\ 3 & 0 & 1 & 3 & 3 & 0 & 3 \\ 2 & 3 & 2 & 0 & 0 & 1 & 1 \\ 2 & 2 & 2 & 3 & 1 & 0 & 1 \\ 0 & 2 & 1 & 1 & 3 & 3 & 1 \end{bmatrix} \star \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 0 \\ 1 & 1 & 2 \end{bmatrix} = \begin{bmatrix} 6 & 5 & 7 \\ 6 & 10 & 13 \\ 11 & 5 & 2 \end{bmatrix}$$

Strided convolution

Consider a stride = 2





Output size

- The output size of a convolution depends on :
  - Input size : n
  - Convolutional filter size : f
  - Padding : p
  - Stride : s

Integer floor value

$$n \times n \quad \Longrightarrow \quad \lfloor \frac{n+2p-f}{s} + 1 \rfloor \times \lfloor \frac{n+2p-f}{s} + 1 \rfloor$$

Input size

Convolution operation
Parameters: f, p, s

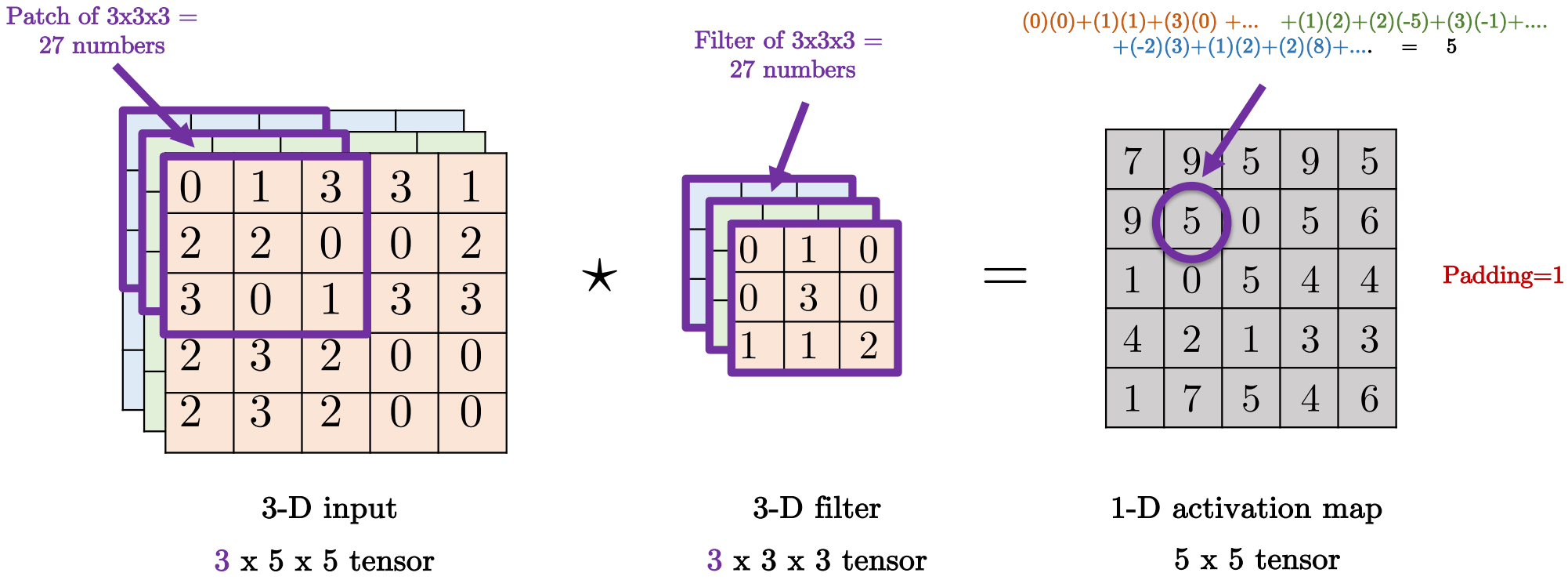
Output size

Outline

- Padding
- Stride
- **Multi-dimensional input**
- Convolution as linear operation
- Depth
- Convolutional layer with PyTorch
- Pooling layer with PyTorch
- LeNet5 architecture
- VGG architecture
- Case studies

Multi-dimensional convolution

- N-D convolution using N-D filters with N-D inputs returns 1-D activation map.
  - Example with 3-D input :



3-D input

3 x 5 x 5 tensor

3-D filter

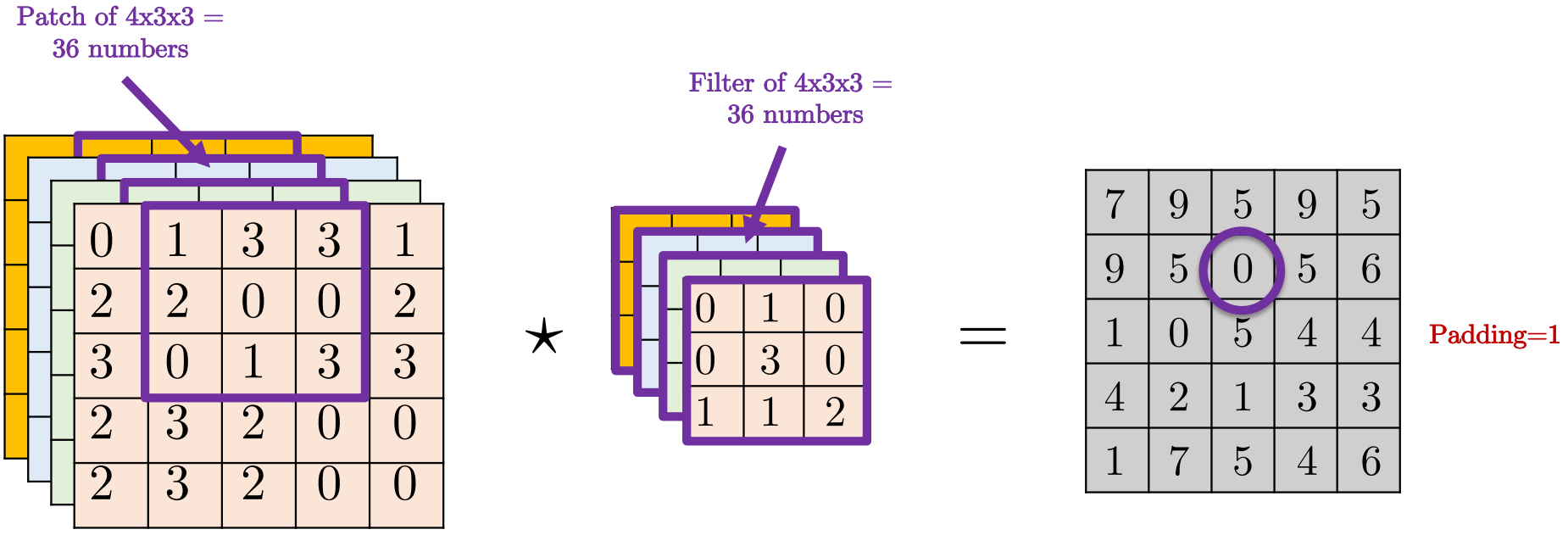
3 x 3 x 3 tensor

1-D activation map

5 x 5 tensor

Multi-dimensional convolution

- N-D convolution using N-D filters with N-D inputs returns 1-D activation map.
  - Example with 4-D input :



4-D input

4 x 5 x 5 tensor

4-D filter

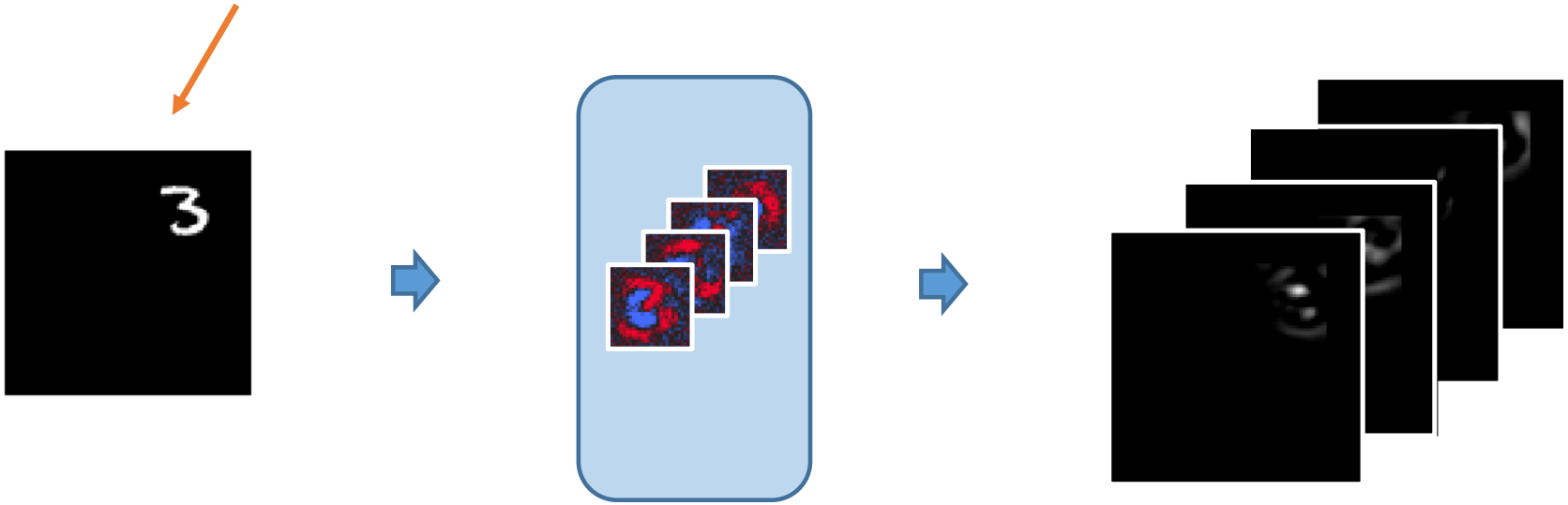
4 x 3 x 3 tensor

1-D activation map

5 x 5 tensor

One-dimensional input

- 1-D input : Gray-scale images



The input has 1 channel → Each filter has 1 channel

128 x 128 tensor

Four filters

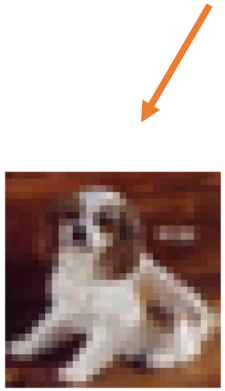
4 x 28 x 28 tensor

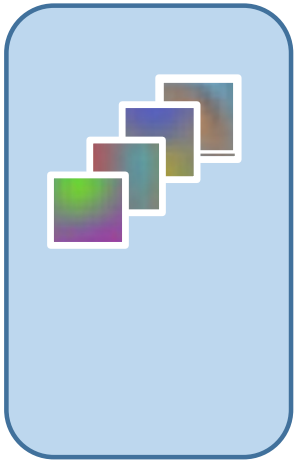
→ Four activation maps

4 x 128 x 128 tensor

Three-dimensional input

- 3-D input : RGB images





The input has N=3 channels → Each filter has 3 channels

Four filters → Four activation maps

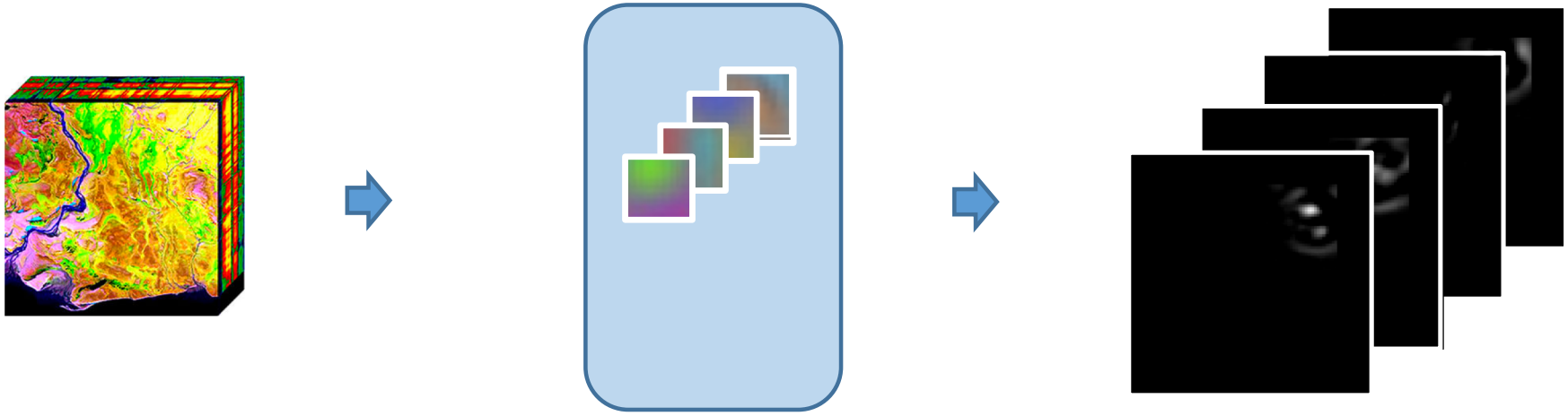
3 x 28 x 28 tensor

4 x 3 x 3 x 3 tensor

4 x 28 x 28 tensor

F-D activations map with N-D input

- Generate F activation maps with N-D inputs :
  - N channels, F filters/activation maps, N x f x f filter size



The input has N channels → Each filter has N channels

F filters → F activation maps

N x n x n tensor

F x N x f x f tensor

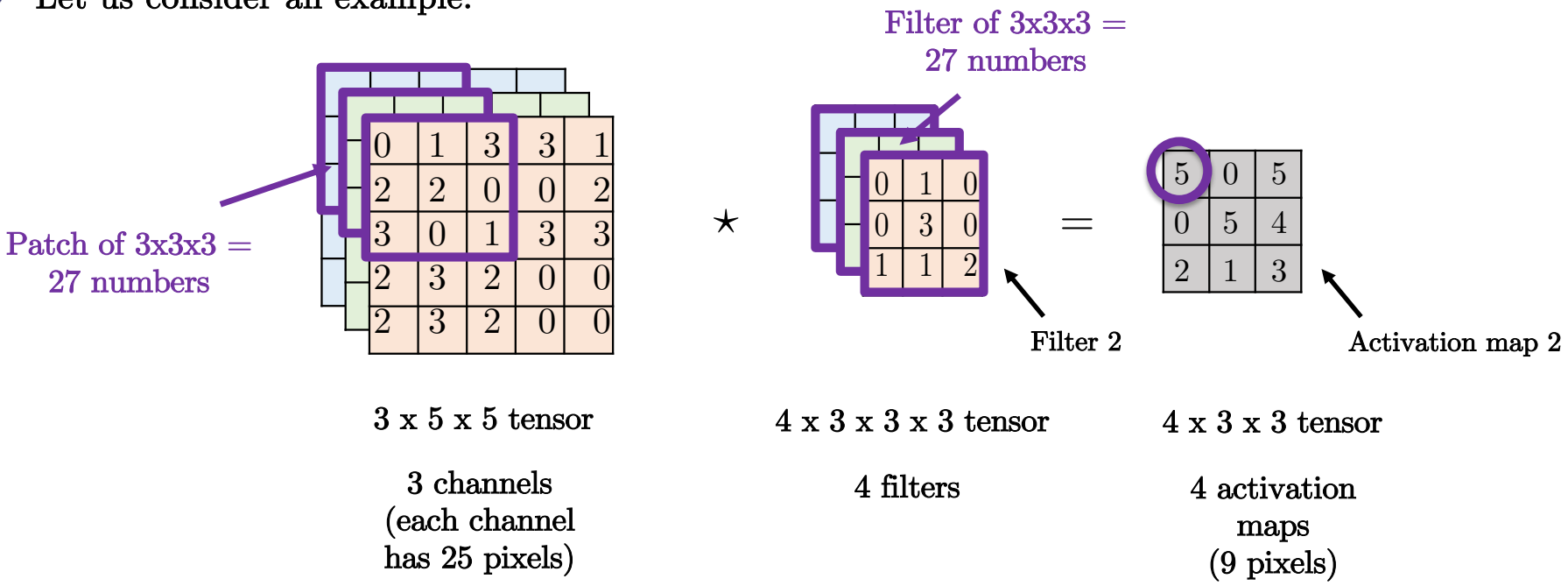
F x n x n tensor

Outline

- Padding
- Stride
- Multi-dimensional input
- Convolution as linear operation
- Depth
- Convolutional layer with PyTorch
- Pooling layer with PyTorch
- LeNet5 architecture
- VGG architecture
- Case studies

Convolution as linear operation

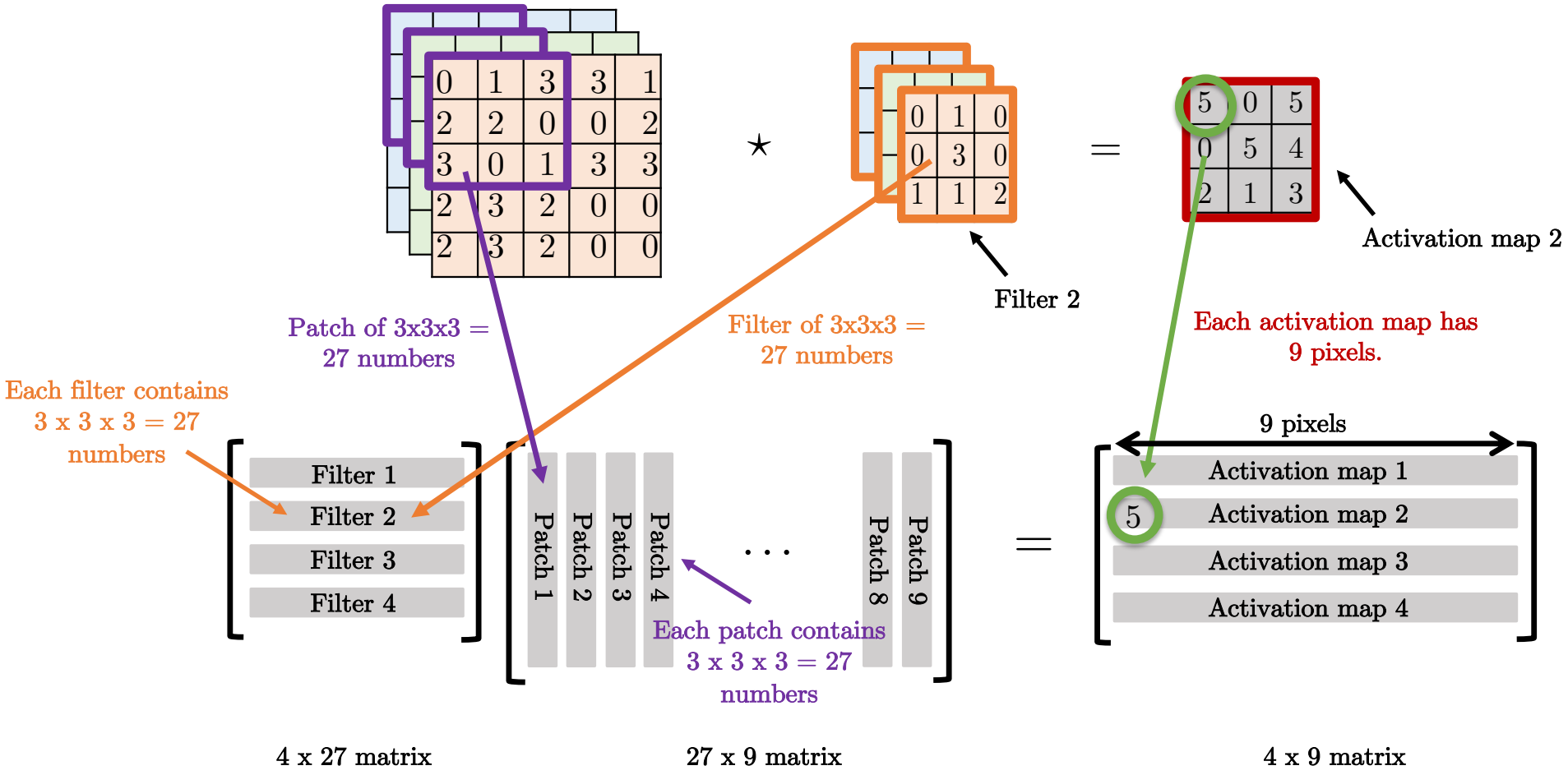
- Convolutional layers are not very different from regular linear layers.
  - They also define a linear operator with a special structure.
- Let us consider an example:



Key observation : Convolution operations are applied on patches !

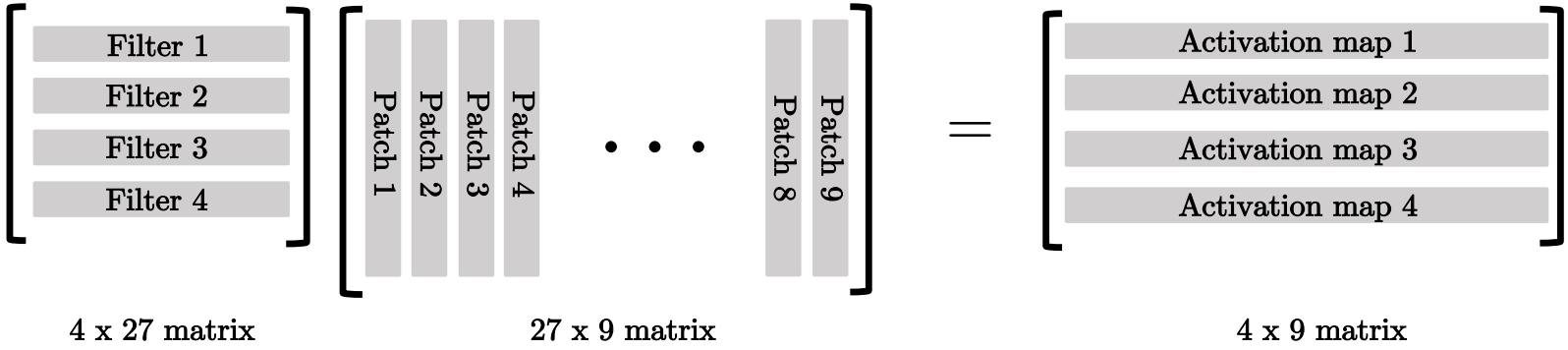
Convolution as linear operation

- Convolutions can be expressed as matrix-matrix multiplications :



Convolution as linear operation

- Convolutions can be expressed as matrix-matrix multiplications :



Convolution = matrix - matrix multiplication

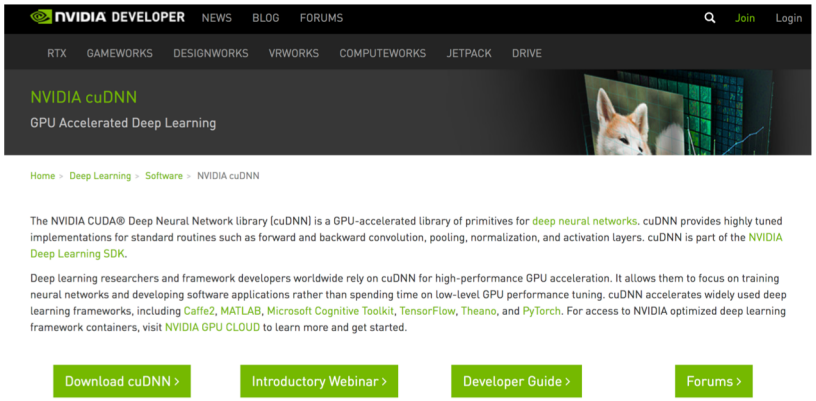
Filters vectorized
(one filter per row)

Input pixels
rearranged as vectorized patches
(one patch per column)

Fast convolution

- Convolutions are simply matrix-matrix multiplications.
  - As matrix-matrix multiplications are standard linear algebra operations that have been highly parallelized.
    - CPU: MKL and BLAS libraries
    - GPU: CUDA library

- Fast convolutions on GPUs with **cuDNN** library.



https://developer.nvidia.com/cudnn

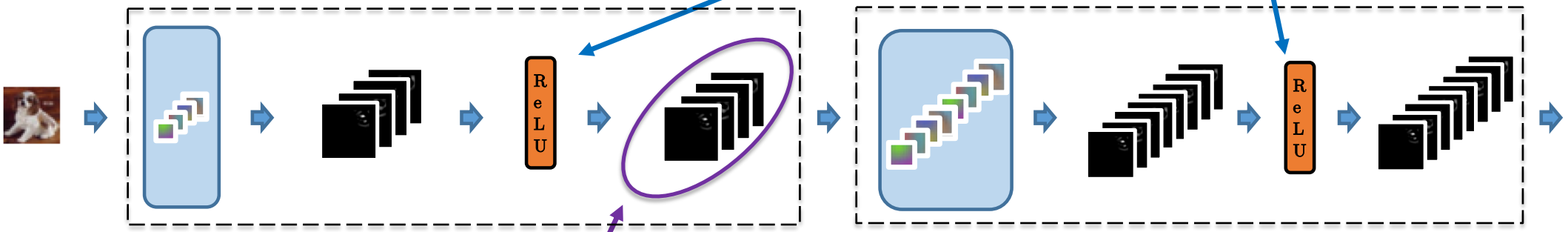
Outline

- Padding
- Stride
- Multi-dimensional input
- Convolution as linear operation
- **Depth**
- Convolutional layer with PyTorch
- Pooling layer with PyTorch
- LeNet5 architecture
- VGG architecture
- Case studies

Depth

- We can put some depth inside the ConvNet architecture :
  - We can stack convolutional layers.

Non-linearity is needed between each layer to prevent the network to collapse into a single convolutional layer.



3 channels Input features

Convolutional layer 1

4 activation maps

4 rectified maps Hidden features

Convolutional layer 2

8 rectified maps Hidden features

Each filter has N=3 channels (RGB) with f=5:
3 x 5 x 5 tensor

CL1 has 4 filters ⇒
#parameters = 4 x 3 x 5 x 5

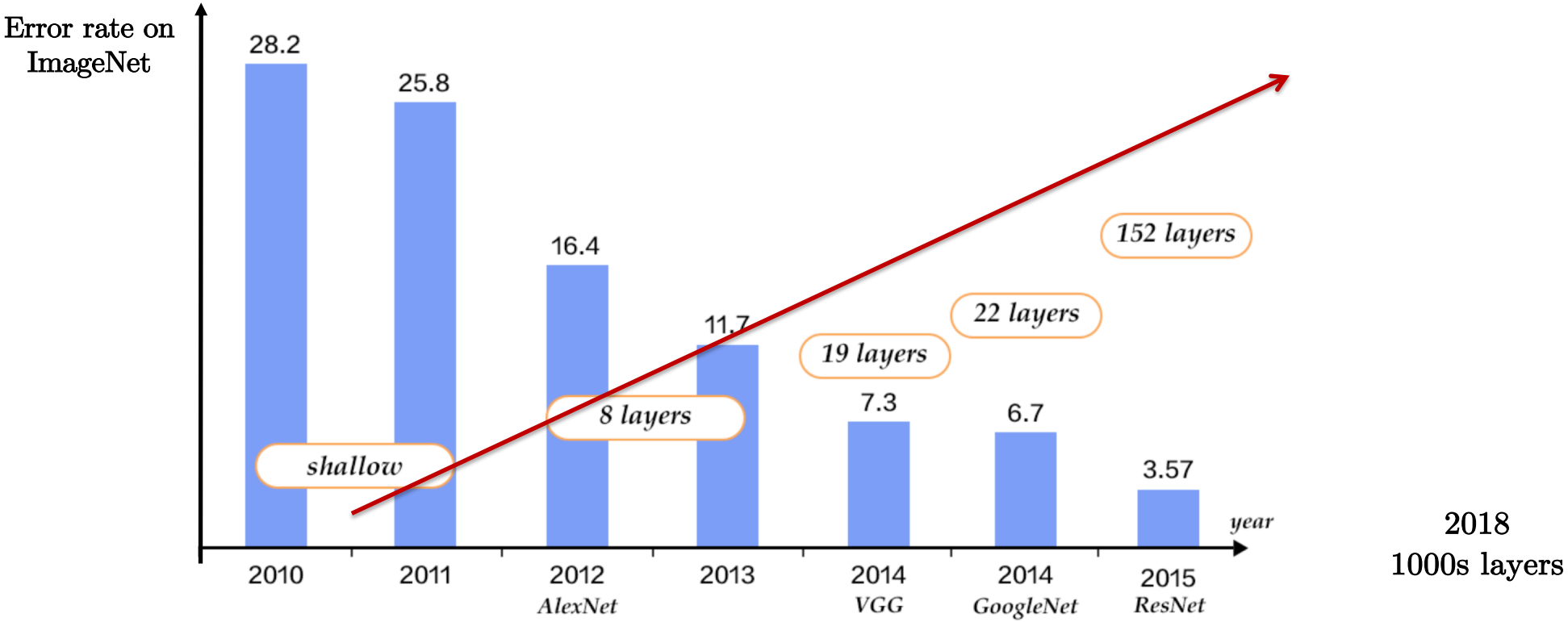
Just think of this as a RGB picture with 4 channels

Each filter has N=4 channels with f=3:
4 x 3 x 3 tensor

CL2 has 4 filters ⇒
#parameters = 8 x 4 x 3 x 3

The deeper the better

- Revolution of depth :



Outline

- Padding
- Stride
- Multi-dimensional input
- Convolution as linear operation
- Depth
- **Convolutional layer with PyTorch**
- Pooling layer with PyTorch
- LeNet5 architecture
- VGG architecture
- Case studies

Convolutional layer in PyTorch

- PyTorch function:

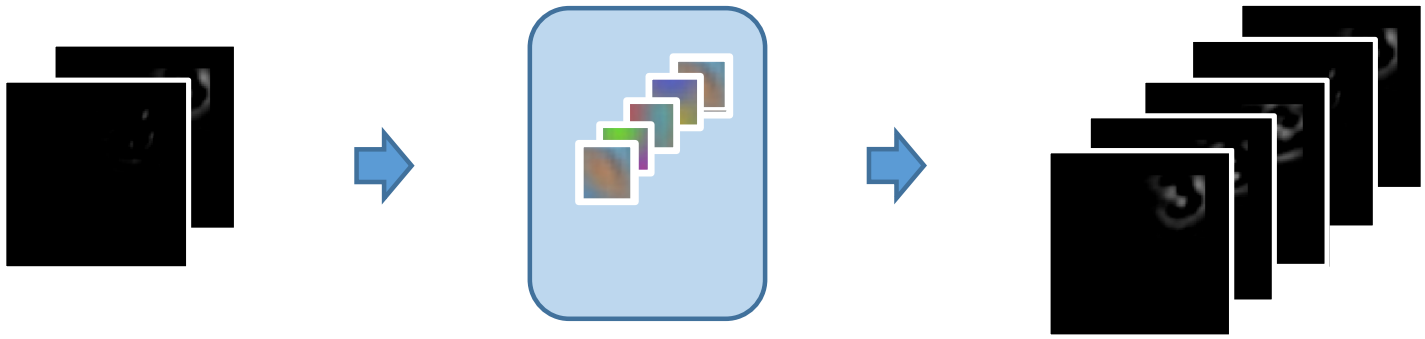
```
mod = nn.Conv2d( 2 , 5 ,  kernel_size=3,  padding=1 )
```

2 input channels

5 filters

Each filter has a size 3x3 filters

5 output channels



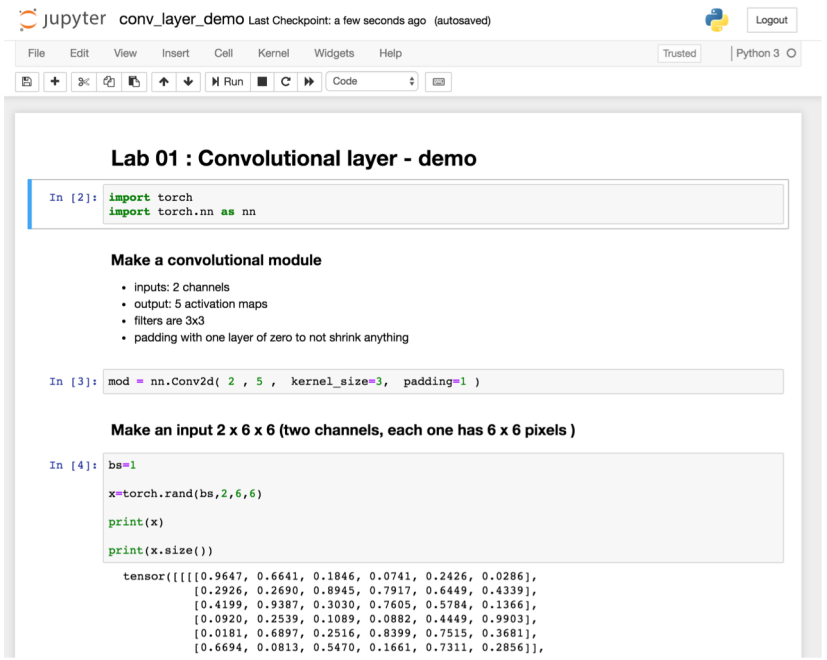
Input is a
2 x 6 x 6 tensor

Conv filters are a
5 x 2 x 3 x 3 tensor

Output is a
5 x 6 x 6 tensor

Lab 01

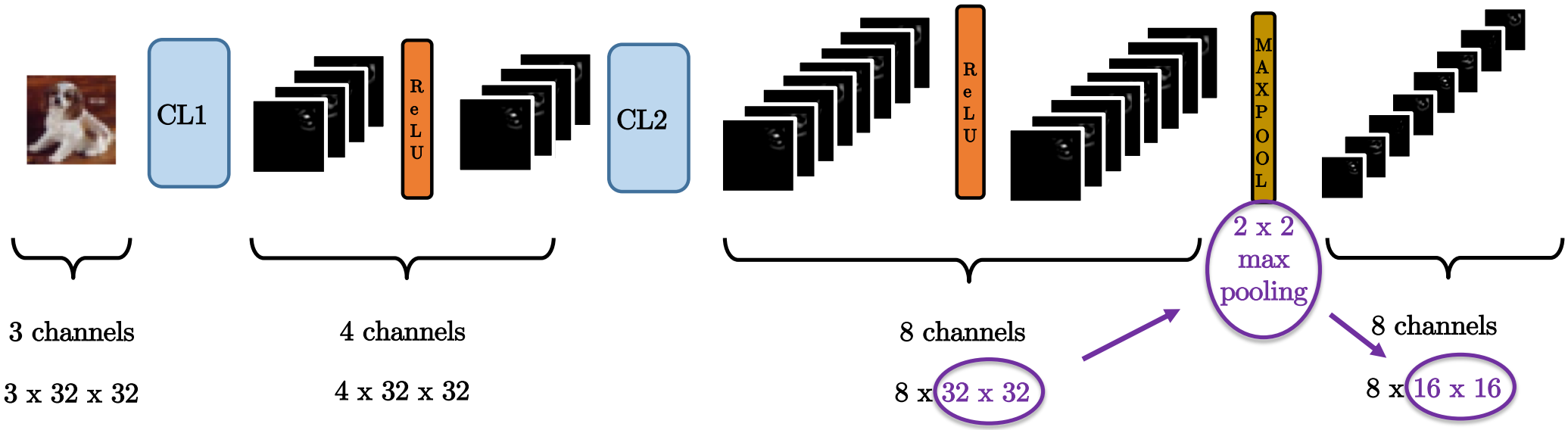
- Convolutional layer



Outline

- Padding
- Stride
- Multi-dimensional input
- Convolution as linear operation
- Depth
- Convolutional layer with PyTorch
- **Pooling layer with PyTorch**
- LeNet5 architecture
- VGG architecture
- Case studies

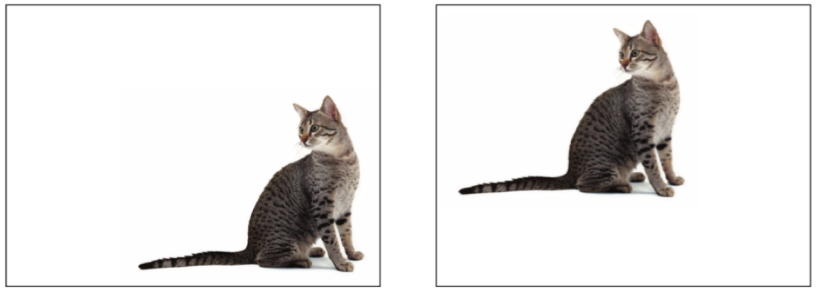
Max pooling



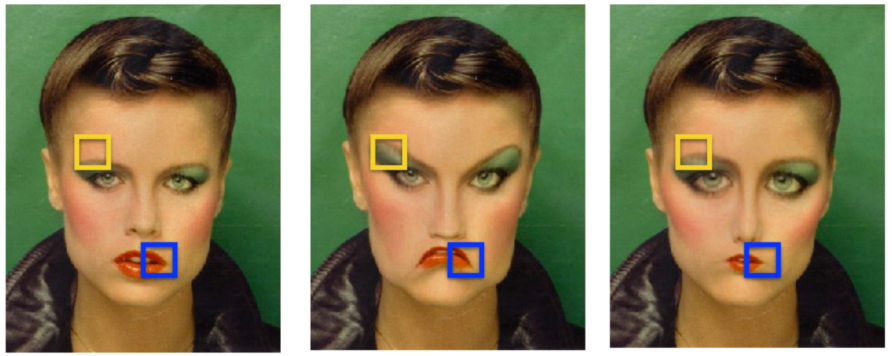
- MP (max pooling) layers are useful for :
  - Trading image resolution with features number (features capture data structures).
  - Controlling the computational complexity (image size /2 and #features x2).

Max pooling

- MP (max pooling) layers are also useful for :
  - Creating global invariance w.r.t. translation (same semantic object independent of location).



  - Providing local invariance w.r.t. object deformations (robust to slight object changes).



Pooling layer in PyTorch

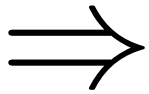
- PyTorch function:
  - Max Pooling with window size = 2 x 2

```
mod = nn.MaxPool2d(2,2)
```

3	3	0	1	1	1
7	2	1	1	5	2
7	9	1	0	5	6
5	6	1	0	3	2
9	7	0	0	0	0
4	7	0	0	0	1

6 x 6 activation map

Down-sampling
+ max pooling

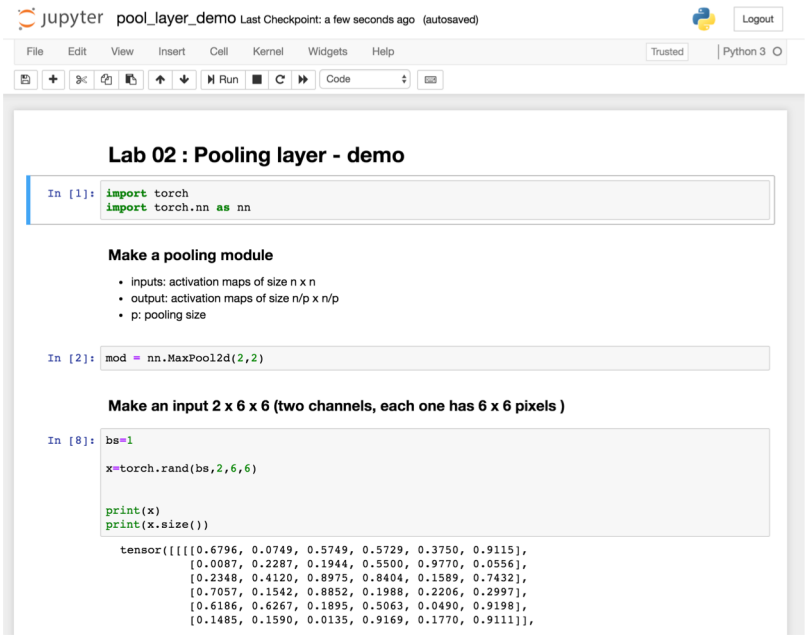


7	1	5
9	1	6
9	0	1

3 x 3 activation map

Lab 02

- Pooling layer :

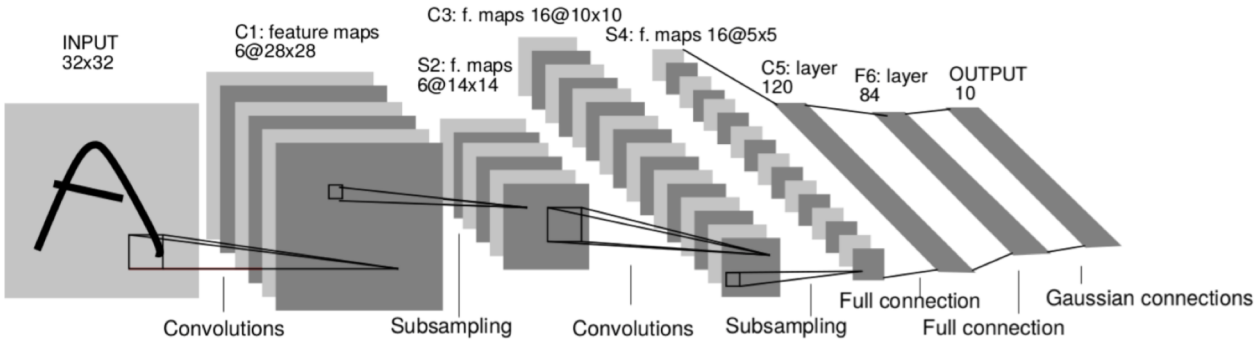


Outline

- Padding
- Stride
- Multi-dimensional input
- Convolution as linear operation
- Depth
- Convolutional layer with PyTorch
- Pooling layer with PyTorch
- **LeNet5 architecture**
- VGG architecture
- Case studies

LeNet5 architecture

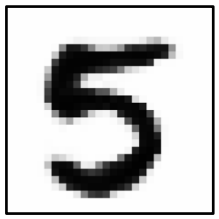
- LeCun, Bottou, Bengio, Haffner, 1998:
  - 1st ConvNet architecture



- The dataset:
  - MNIST
- The network:
  - Convolutional/MP layers
  - Linear layers

MNIST dataset

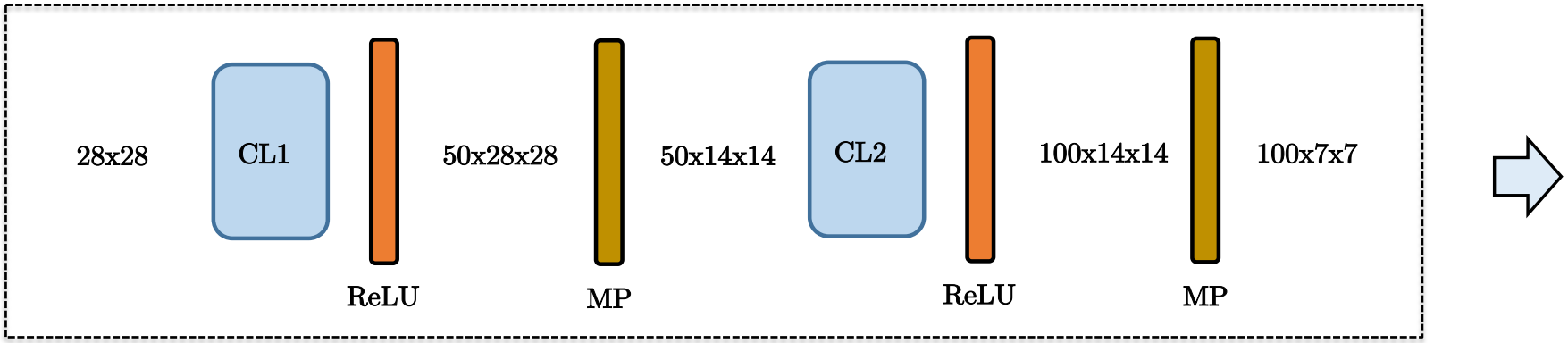
- Number of classes: 10
- Training set: 50,000 labelled images
- Test set: 10,000 labelled images



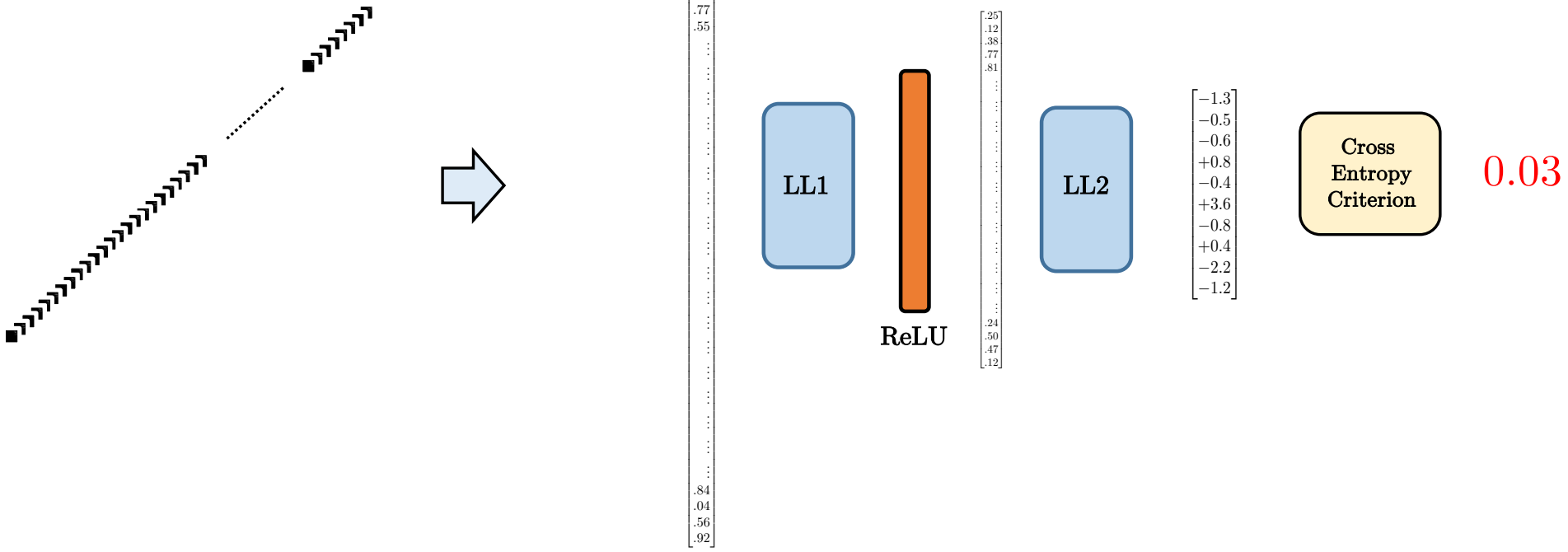
28 x 28 grayscale image

Two convolutional layers

- Convolutional/MP layers:



Linear layers



CL features

100 (small) activation maps

(7 x 7 pixel each)

Vector with 4,900 entries

(100 x 7 x 7 = 4,900)

Vectors with 100 entries

10 Scores

Loss

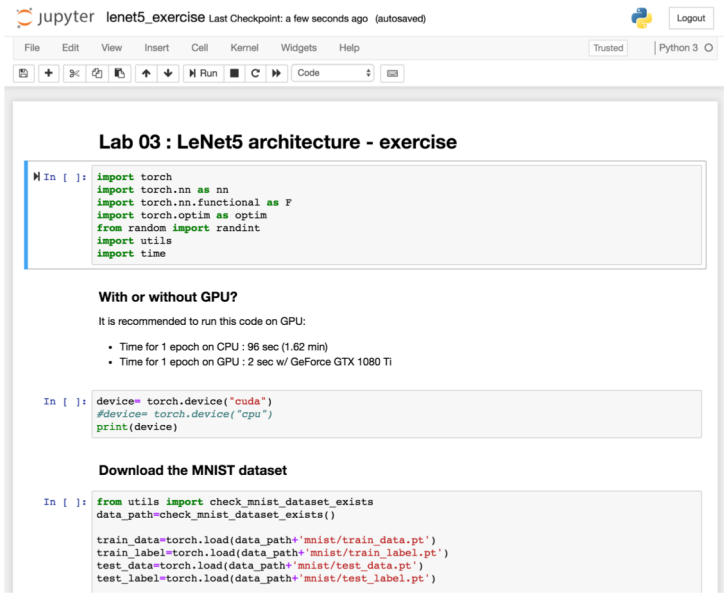
Number of architecture parameters

- First CL:
  - 28x28 $\Rightarrow$ CL1+ReLU $\Rightarrow$ 50x28x28
  - 50x28x28 $\Rightarrow$ MP $\Rightarrow$ 50x14x14
  - #Parameters = 50x3x3 + 50 = 500
- Second CL:
  - 50x14x14 $\Rightarrow$ CL1+ReLU $\Rightarrow$ 100x14x14
  - 100x14x14 $\Rightarrow$ MP $\Rightarrow$ 100x7x7
  - #Parameters = 100x50x3x3+100 = 45,100
- First LL:
  - 100x7x7 $\Rightarrow$ Reshape $\Rightarrow$ 4,900
  - 4,900 $\Rightarrow$ LL1 $\Rightarrow$ 100
  - #Parameters = 4900x100+100 = 490,100
- Second LL:
  - 100 $\Rightarrow$ LL2 $\Rightarrow$ 10
  - #Parameters = 100x10+10 =1,010

Total for 4 Layers $\Rightarrow$ 536,710 / 0.54 Millions of parameters

Lab 03

- LeNet5 on MNIST :
  - Learning rate schedule:
    - Start with lr=0.25
    - Divide by 1.5 every 5 epochs



Results

- Results with a 4-layer ConvNet:
  - Number of parameters: 0.54 million
  - Error on training set: 0%
  - Error on test set: 0.78%

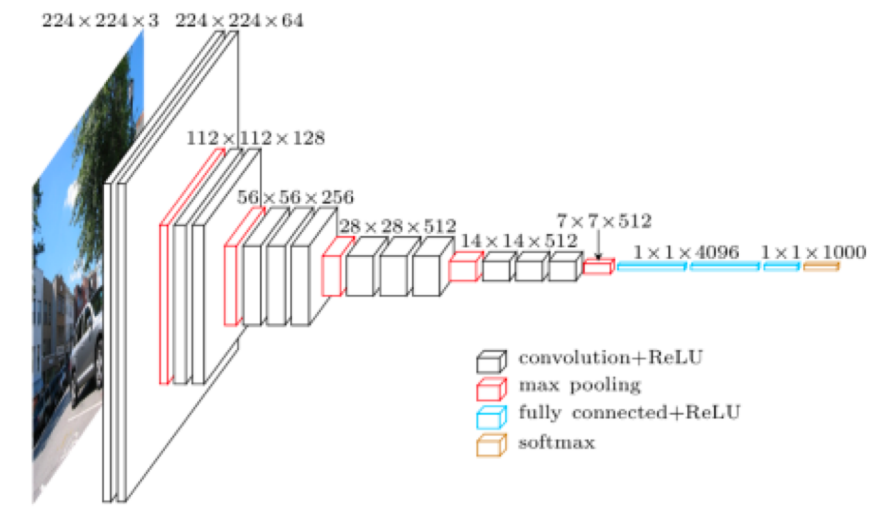
- Timing for one epoch:
  - CPU : 1.6 min
  - Tesla K80 GPU (Google Cloud) : 6 sec
  - GTX 1080 GPU ($600) : 2 sec

Outline

- Padding
- Stride
- Multi-dimensional input
- Convolution as linear operation
- Depth
- Convolutional layer with PyTorch
- Pooling layer with PyTorch
- LeNet5 architecture
- **VGG architecture**
- Case studies

VGG architecture

- Simonyan and Zisserman, 2014 :
  - 2nd place at the ImageNet challenge in 2014
  - All convolutional features are 3x3

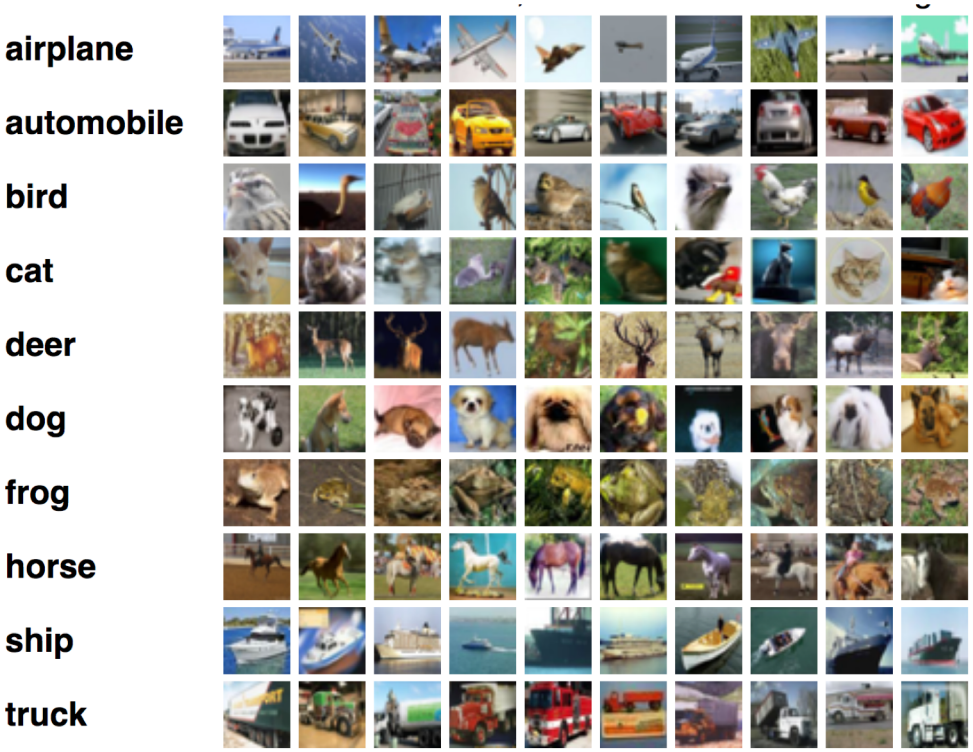


- The dataset:
  - CIFAR
- The network:
  - Convolutional/MP layers
  - Linear layers

CIFAR dataset

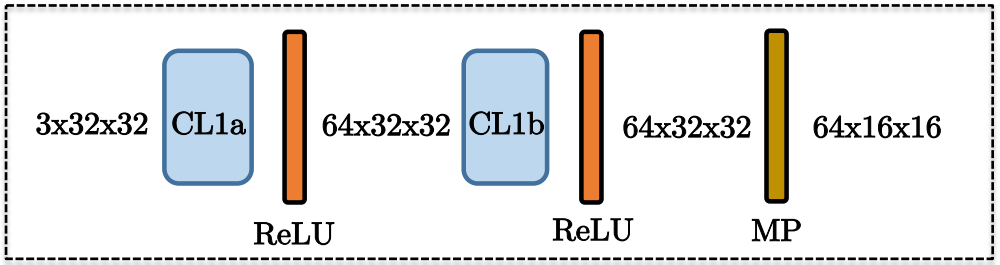
- Number of classes: 10
- Training set: 50,000 labelled images
- Test set: 10,000 labelled images

32 x 32 RGB image

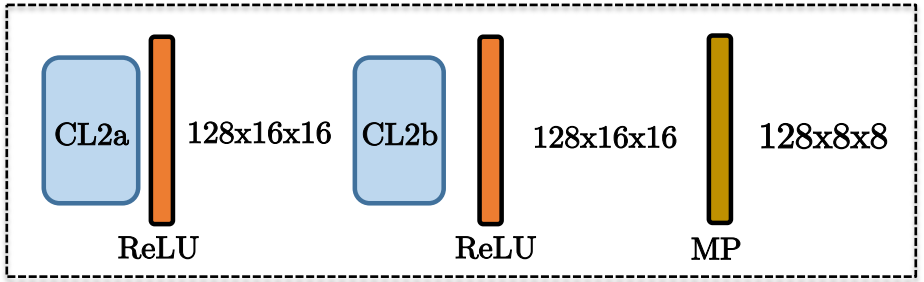


Four convolutional layers

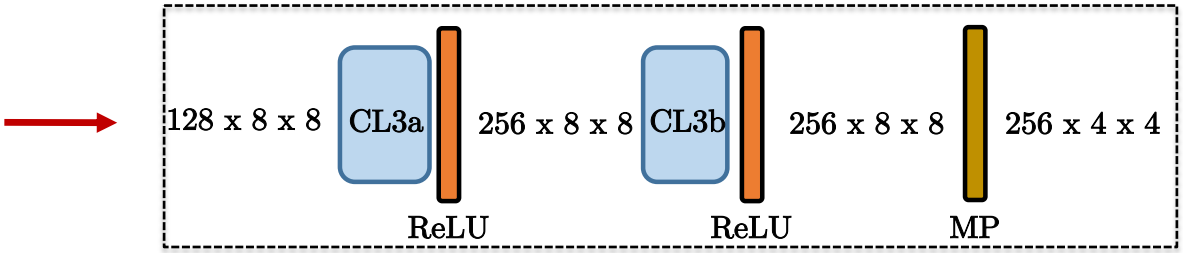
- Convolutional/MP layers:
  - The number of features/channels increases by a factor 2.
  - The resolution of the activation maps decreases by a factor 2.



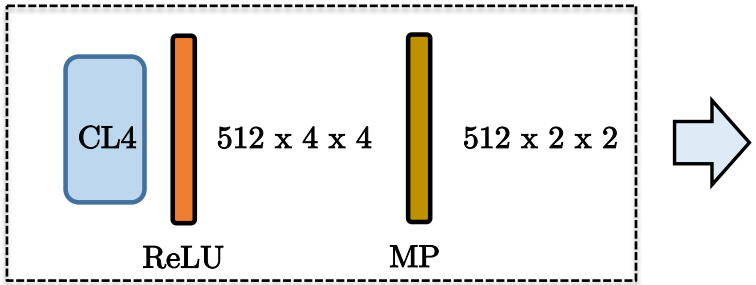
First block: from 3x32x32 to 64x16x16



Second block: from 64 x 16 x 16 to 128 x 8 x 8

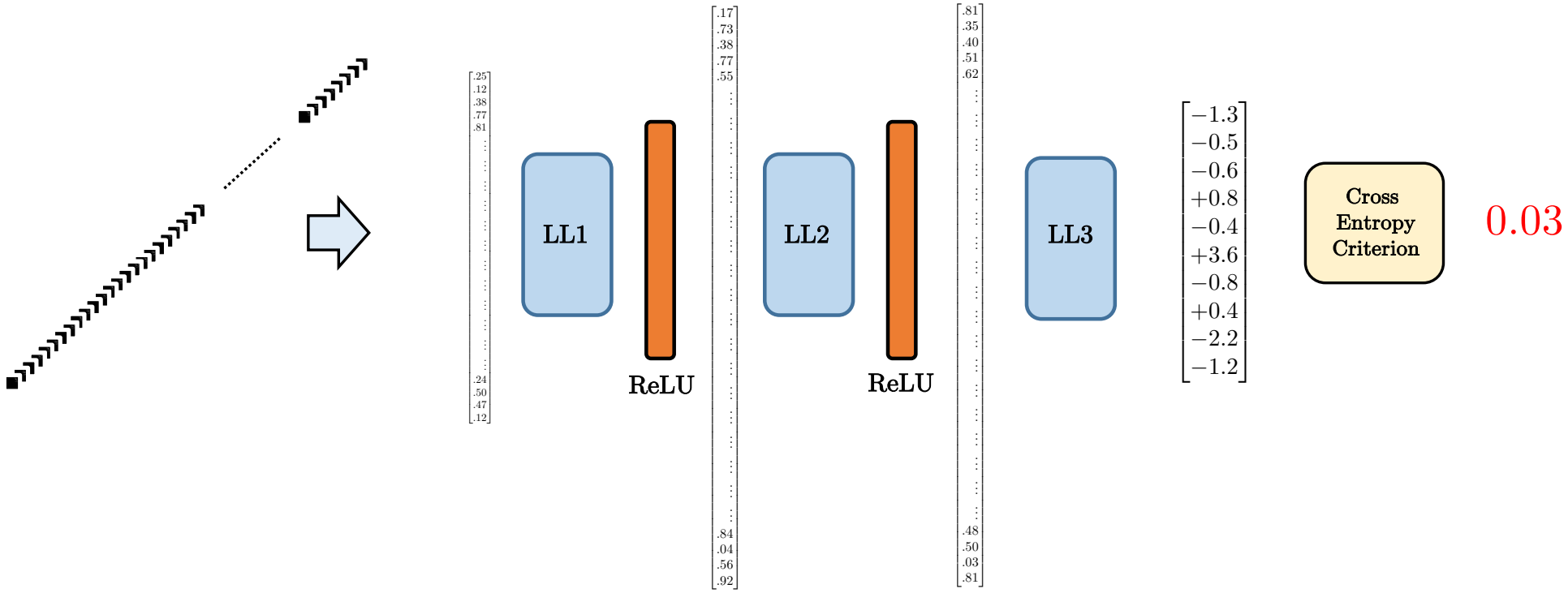


Third block: from 128 x 8 x 8 to 256 x 4 x 4



Fourth block: from 256 x 4 x 4 to 512 x 2 x 2

Linear layers



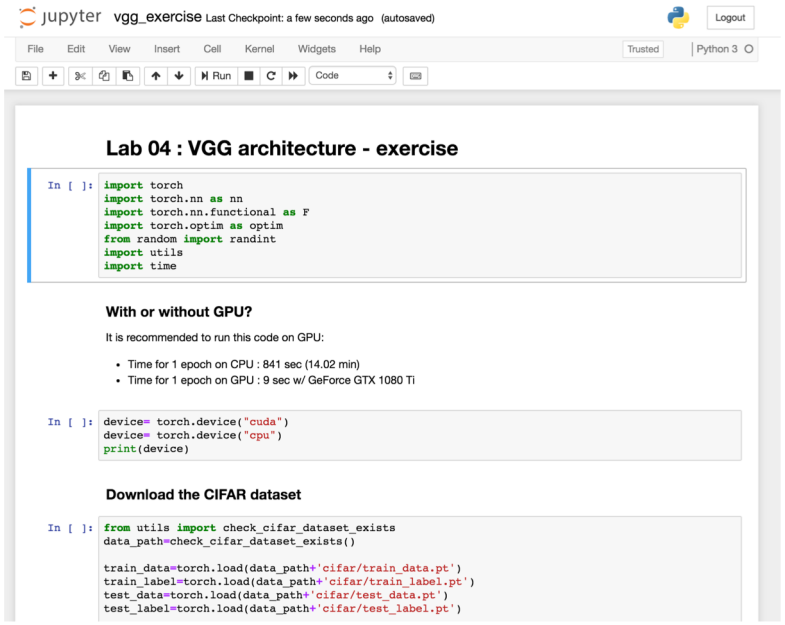
Number of architecture parameters

- First Block:
  - 3x32x32 ⇒ CL1a+ReLU ⇒ 64x32x32
  - 64x32x32 ⇒ CL1b+ReLU ⇒ 64x32x32
  - 64x32x32 ⇒ MP ⇒ 64x16x16
  - #Parameters = 64x3x3x3+64+64x64x3x3+64 = 38K
- Second Block:
  - 64x16x16 ⇒ CL2a+ReLU ⇒ 128x16x16
  - 128x16x16 ⇒ CL2b+ReLU ⇒ 128x16x16
  - 128x16x16 ⇒ MP ⇒ 128x8x8
  - #Parameters = 128x64x3x3+128+128x128x3x3+128 = 221K
- Third Block:
  - 128x8x8 ⇒ CL3a+ReLU ⇒ 256x8x8
  - 256x8x8 ⇒ CL3b+ReLU ⇒ 256x8x8
  - 256x8x8 ⇒ MP ⇒ 256x4x4
  - #Parameters = 256x128x3x3+256+256x256x3x3+256 = 885K
- Fourth Block:
  - 256x4x4 ⇒ CL4+ReLU ⇒ 512x4x4
  - 512x4x4 ⇒ MP ⇒ 512x2x2
  - #Parameters = 512x216x3x3+512+512x512x3x3+512 = 3.3M
- LL:
  - 512x2x2 ⇒ Reshape ⇒ 2,048
  - 2,048 ⇒ LL1+ReLU ⇒ 4,096
  - 4,096 ⇒ LL2+ReLU ⇒ 4,096
  - 4,096 ⇒ LL3 ⇒ 10
  - #Parameters = 2048x4096+4096+4096x4096+4096+4096x10+10 = 25.2M

Total for 10 Layers ⇒ 30 Millions of parameters

Lab 04

- Learning rate schedule:
  - Start with lr=0.25
  - Divide by 2 at epoch 10
  - Divide by 2 again at epoch 14
  - Divide by 2 again at epoch 18



Results

- Results with a 1-layer MLP :
  - Number of parameters: 0.03 million
  - Error on training set: 59%
  - Error on test set: 65%

- Results with a 3-layer MLP :
  - Number of parameters: 2 millions
  - Error on training set: 0%
  - Error on test set: 46%

- Results with a 10-layer ConvNet :
  - Number of parameters: 30 millions
  - Error on training set: 0%
  - Error on test set: 20%

Timing for one epoch:

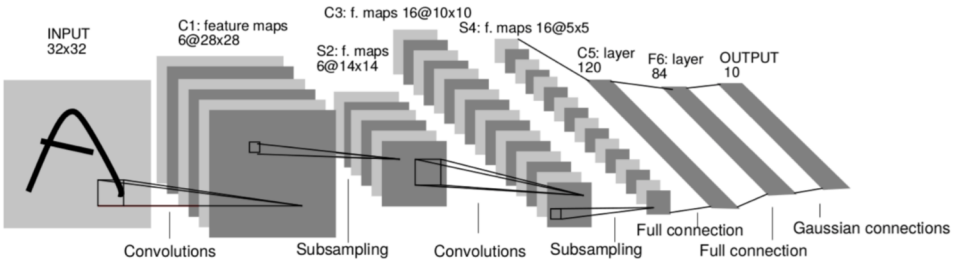
- CPU : 14 min
- Tesla K80 GPU (Google Cloud): 1.2 min
- GTX 1080 GPU ($600) : 9 sec

Outline

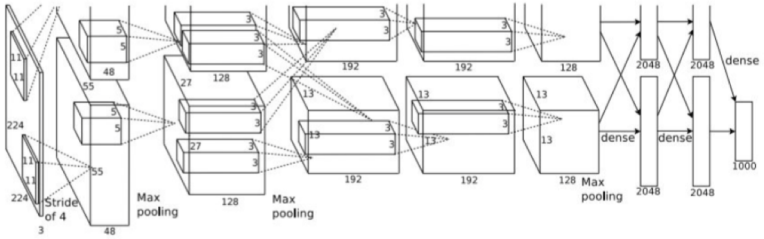
- Padding
- Stride
- Multi-dimensional input
- Convolution as linear operation
- Depth
- Convolutional layer with PyTorch
- Pooling layer with PyTorch
- LeNet5 architecture
- VGG architecture
- Case studies

ConvNet architectures

- LeNet5 [LeCun, Bottou, Bengio, Haffner, 1998]:
  - Input is 28x28
  - Architecture is CL-MP-CL-MP-LL-LL
  - Accuracy on MNIST is 99.6%
  - #Parameters=0.6M

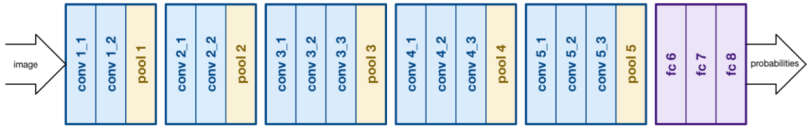


- AlexNet [Krizhevsky, Hinton, Sutskever, 2012]:
  - Input is 227x227x3
  - Architecture is 7CL-3MP-2FC
  - Predication error on ImageNet is 15.4%
  - 1st rank
  - #Parameters=62M

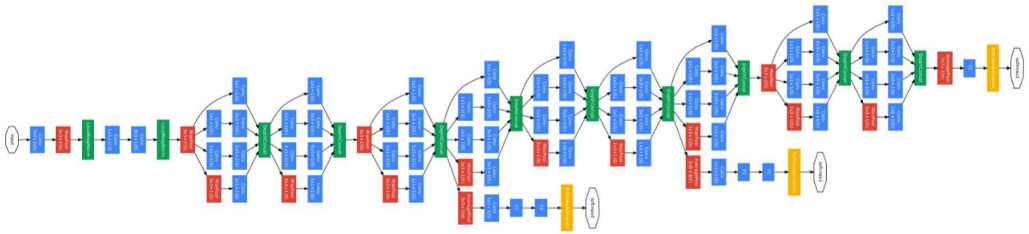


ConvNet architectures

- VGGNet [Simonyan, Zisserman, 2014]:
  - Input is 227x227x3
  - Architecture has 16 layers
  - Predication error on ImageNet is 7.3%
  - 2nd rank
  - #Parameters=138M

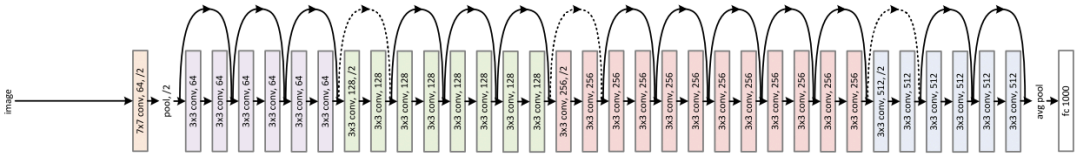


- GoogleNet/Inception [Szegedy-et.al, 2014]:
  - Input is 227x227x3
  - Architecture has 22 layers
  - Predication error on ImageNet is 6.7%
  - 1st rank
  - #Parameters=5M



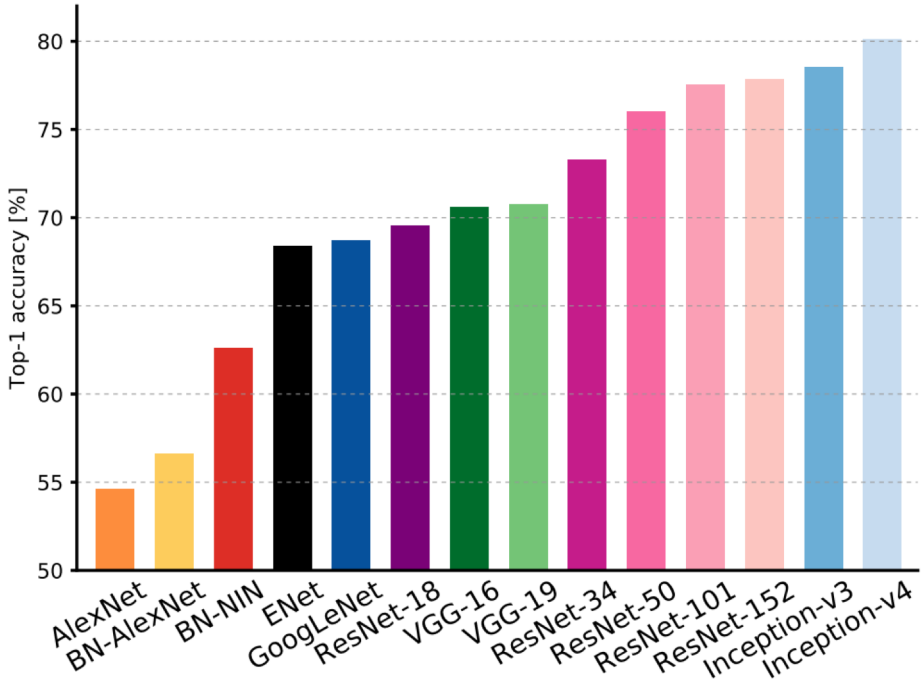
ConvNet architectures

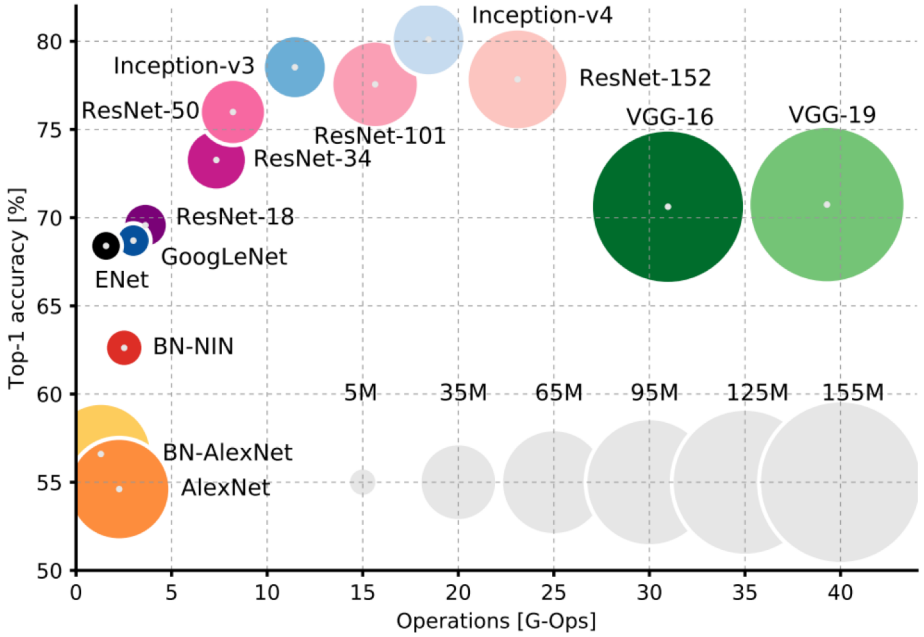
- Microsoft/ResNet [He-et.al'15]:
  - Input is 227x227x3
  - Architecture has 152 layers
  - Predication error on ImageNet is 3.6%
  - 1st rank
  - #Parameters=60M



- Etc.

ConvNet architectures





Questions?