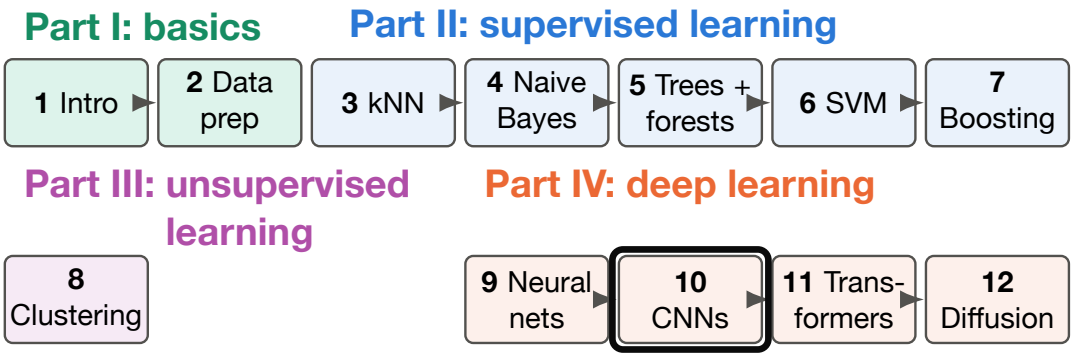
Convolutional neural networks

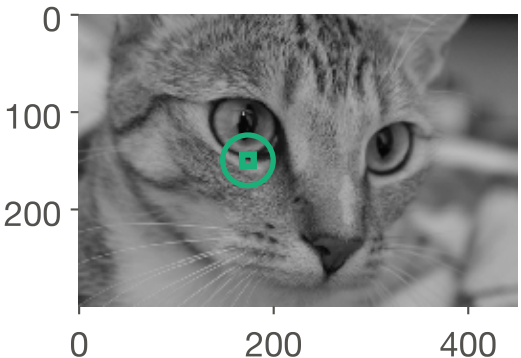
229352 Lecture 10: convolutional neural networks

Course map


Part I: basics
Part II: supervised learning
1 Intro
2 Data prep
3 kNN
4 Naive Bayes
5 Trees + forests
6 SVM
7 Boosting
Part III: unsupervised learning
Part IV: deep learning
8 Clustering
9 Neural nets
10 CNNs
11 Trans-formers
12 Diffusion

A grayscale image is a grid of numbers

Task: classify images, for example cat or dog.



Height $H = 300$ rows, width $W = 451$ columns of **pixels**.

A pixel is one number: 0 (black) to 255 (white).

Circled: a patch of 12×12 pixels.

Photo: scikit-image sample “chelsea” (CC0, S. van der Walt), in grayscale

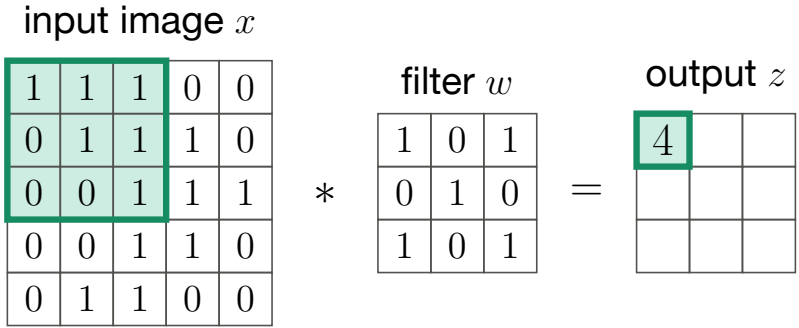
Pixel values of the 12×12 patch

23	23	20	26	25	25	26	27	33	32	30	26
22	21	23	23	21	24	19	20	25	22	22	20
35	30	29	30	26	23	23	22	20	17	22	21
81	60	60	50	37	35	33	28	27	28	25	28
102	97	94	77	66	55	55	53	49	45	41	52
131	130	113	101	100	92	89	71	66	75	73	75
132	120	126	126	114	111	111	102	98	98	97	98
142	137	134	123	122	131	129	127	122	124	111	114
153	152	145	142	143	143	139	139	142	137	130	132
162	160	156	151	150	151	150	148	150	153	142	139
156	158	156	155	155	156	153	154	157	146	136	144
151	150	152	158	156	156	155	156	163	160	154	158

the circled patch: rows 144 to 155, columns 168 to 179

(from 0, as on the axes)

Convolution with a 3×3 filter



Slide the **filter** over the image. At each position, multiply the overlapping numbers and add.

First window:

$$1{\cdot}1 + 1{\cdot}0 + 1{\cdot}1 + 0{\cdot}0 + 1{\cdot}1 + 1{\cdot}0 + 0{\cdot}1 + 0{\cdot}0 + 1{\cdot}1 = 4$$

Convolution as a formula

Image x: pixels (features) x_{rc}, row r, column c.
An $F \times F$ filter with **weights** w_{uv} and a **bias** b.

$$z_{rc} = \sum_{u=1}^{F} \sum_{v=1}^{F} w_{uv}\, x_{r+u-1,\, c+v-1} + b$$

z_{rc}: output of the window with top-left pixel x_{rc}; above, $b = 0$.

The filter is not flipped: mathematics calls this cross-correlation, deep learning calls it **convolution**: $z = x * w + b$, a **feature map**.

Vertical edge detection



$$\begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$$



image x filter w $|z|$: white = large

The **Sobel filter** for vertical edges.

$|z|$ is large where the brightness changes from left to right.

Vertical edge detection by hand

$$\begin{bmatrix} 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 \end{bmatrix}$$

$$*$$

$$\begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$$

Compute z.

Horizontal edge detection



$$\begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$



image x filter w $|z|$: white = large

The Sobel filter for horizontal edges.

In a **convolutional neural network** (CNN), training learns the filter weights (Lecture 9).

A color image has three channels

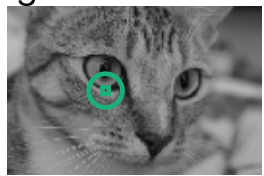
RGB color image: $C = 3$ **channels**, each $H \times W$.

red channel



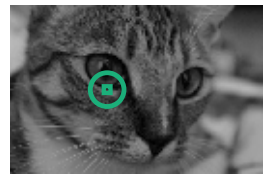
30	30	29	36
31	30	37	37
59	49	47	45
121	98	93	80

green channel



22	22	19	24
21	20	20	20
31	26	26	28
73	53	55	45

blue channel

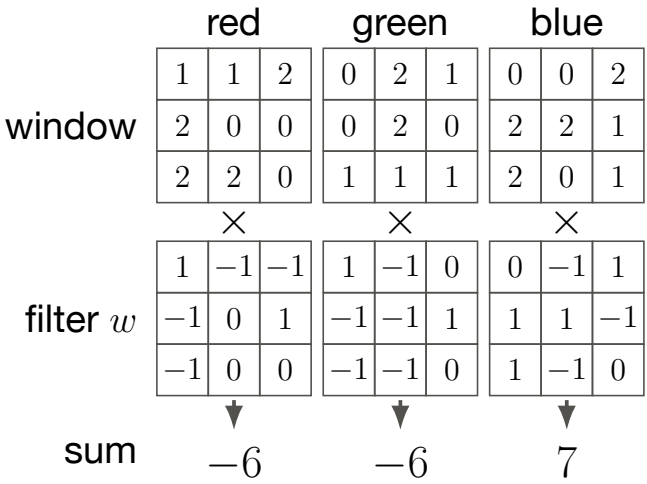


9	9	7	10
9	10	13	13
10	10	5	8
37	22	19	15

The cat: $300 \times 451 \times 3 = 405{,}900$ numbers.

Circled: the patch; values: its top-left 4×4 pixels.

A filter for a color image



One filter (made-up numbers): a 3×3 slice per channel.

Multiply each slice and its window entry by entry, and add.

Three sums plus the bias $b = 1$:

$(-6)+(-6)+7+1 = -4$

Summary: images and filters

- An image: $H \times W$ numbers; $H \times W \times C$ for color.
- Convolution: sum of weights times pixels in a window.
- Filters find patterns, such as vertical or horizontal edges.
- C channels: $F \times F \times C$ weights and a bias per filter.
- One feature map per filter; training learns the weights.

Checkpoint 1

An RGB image of 32×32 pixels;
one filter of size 5×5, with a bias.

How many weights and biases?
How many output maps?

Checkpoint 1

An RGB image of 32×32 pixels;
one filter of size 5×5, with a bias.

How many weights and biases?
How many output maps?

$5 \times 5 \times 3 = 75$ weights, plus 1 bias: 76; one map

Reading ZIP codes (1989)



Yann LeCun

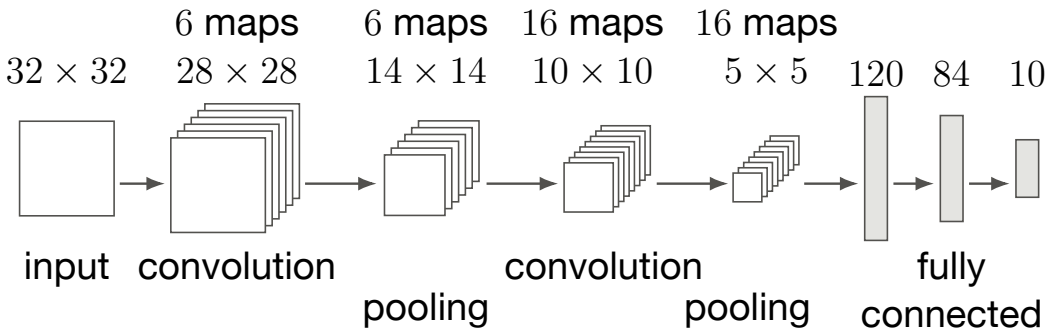
LeCun et al., *Neural Computation*, 1989: a convolutional network, trained with backpropagation, recognizes handwritten ZIP code digits.

One network maps the image of a digit to its class.

Error on the test digits: 5.0%.

Photo (2018): J. Barande / École polytechnique Université Paris-Saclay, CC BY-SA 2.0

LeNet-5 (1998)



Pooling shrinks each map. **Fully connected**: each node uses all values of the previous layer (as in Lecture 9).

Drawing after LeCun et al., *Proc. IEEE*, 1998, Fig. 2

The ImageNet challenge

- **ImageNet**: a large database of photos labeled with object classes.
- ILSVRC 2012 (ImageNet Large Scale Visual Recognition Challenge), classification: 1,000 classes; about 1.2 million training images.
- **Top-5 error**: the share of test images whose true class is not among the model's top five classes.

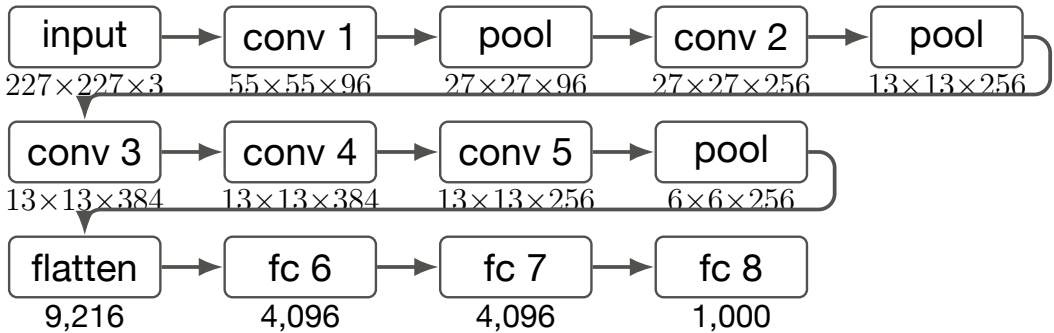
Deng et al., CVPR, 2009; Russakovsky et al., *IJCV*, 2015, Table 2

AlexNet (2012)

- Team *SuperVision* in ILSVRC 2012.
- A CNN with eight layers that have weights: five convolution layers and three fully connected layers; about 61 million weights and biases.
- Top-5 test error 15.3%; another team's best: 26.2%.

Krizhevsky, Sutskever and Hinton, NIPS, 2012

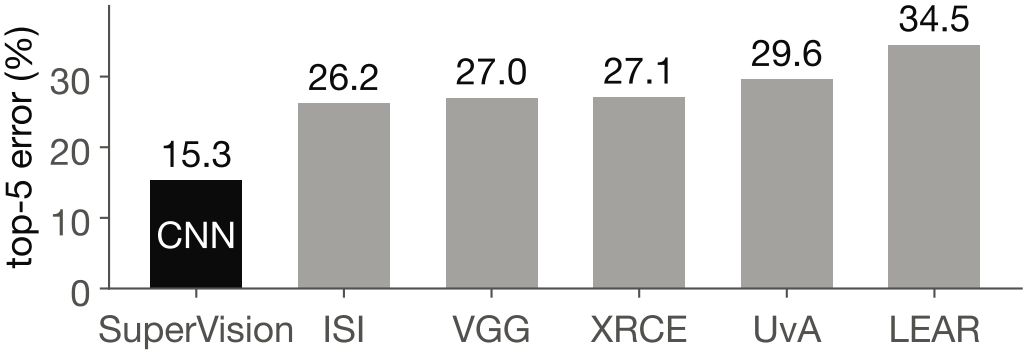
AlexNet: the layers



Sizes: height × width × channels; fc: fully connected.

After Krizhevsky et al., NIPS, 2012, Fig. 2, as one network

ILSVRC 2012: classification results

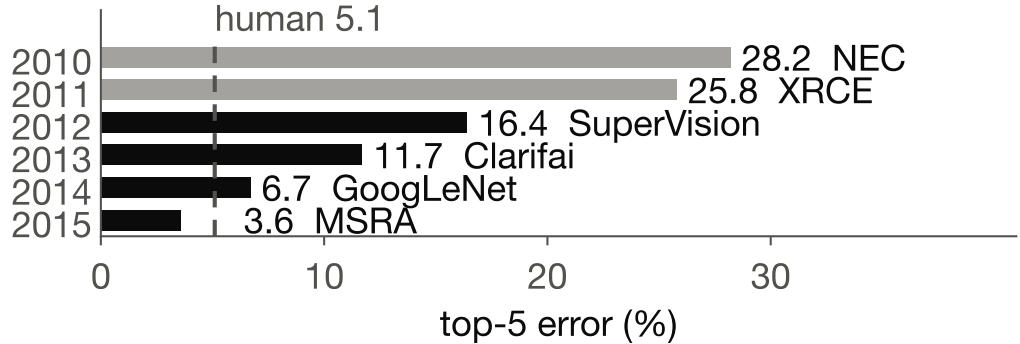


best entry of each team; gray: other methods

SuperVision: 15.3% (16.4% without extra training images)

Russakovsky et al., *IJCV*, 2015, Table 5

ImageNet winners, 2010 to 2015

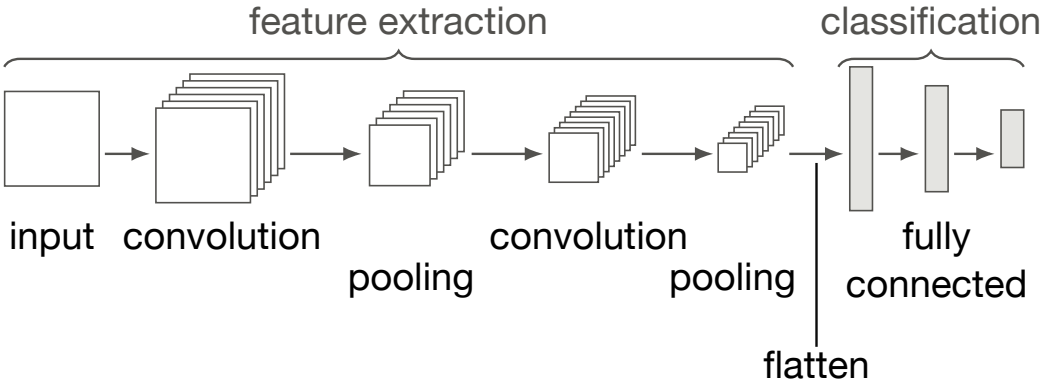


winner of each year, no extra training images; black: CNNs

Russakovsky et al., *IJCV*, 2015, Tables 5 to 7 and 9; He et al., CVPR, 2016

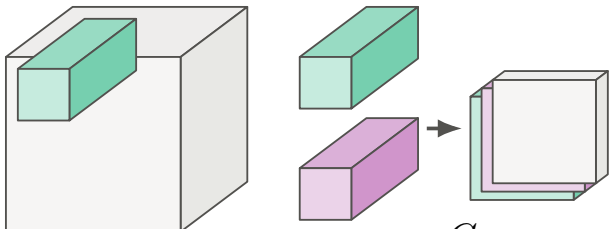
Main layers of a CNN

Flatten: the last maps become one vector.



Drawing after LeCun et al., *Proc. IEEE*, 1998, Fig. 2

A convolution layer

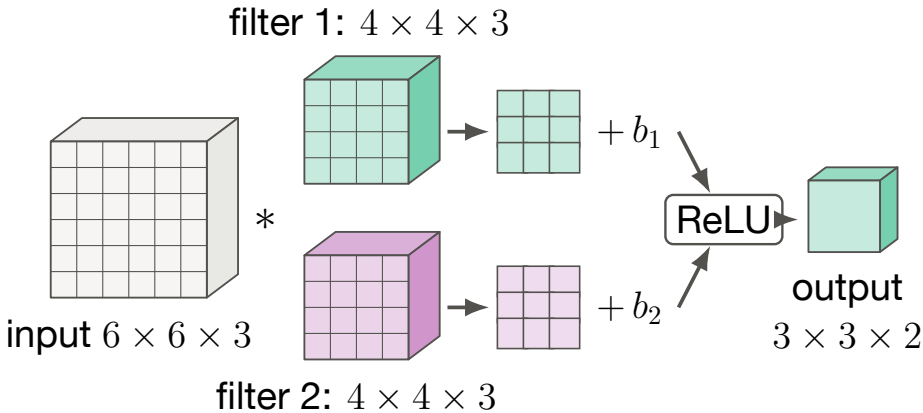


Input: C_{in} channels.

C_{out} filters, each $F \times F \times C_{\text{in}}$ with a bias.

One feature map per filter; stacked: C_{out} channels.

A convolution layer with two filters



Each map gets its own bias, then **ReLU** $\max(0, z)$ (Lecture 9).

Convolution layers in PyTorch

```
import torch
from torch import nn

conv = nn.Conv2d(in_channels=3, out_channels=16,
                 kernel_size=3)
print(conv.weight.shape)  # torch.Size([16, 3, 3, 3])
print(conv.bias.shape)    # torch.Size([16])
```

16 filters, each 3×3 over 3 input channels, and 16 biases.

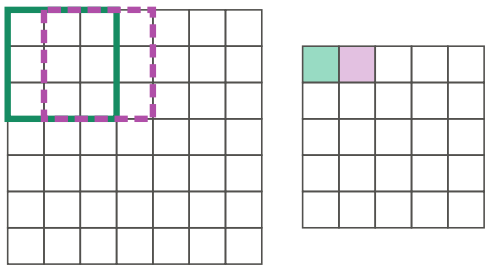
PyTorch stores the weights as (filters, channels, height, width).

Each code slide continues the code of the slides before it.

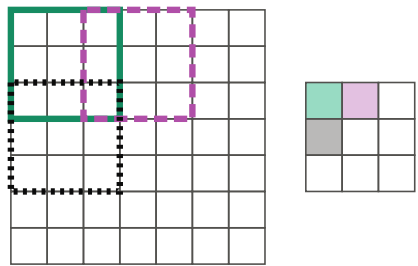
Stride

Stride S: the number of pixels the filter moves at each step.

$S = 1$: output 5×5



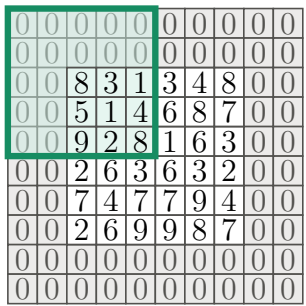
$S = 2$: output 3×3



Input 7×7, filter 3×3; a window and its output share a mark.

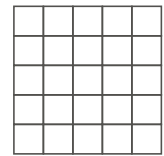
Zero padding

Padding P: P rings of zeros around the input. Here $P = 2$.



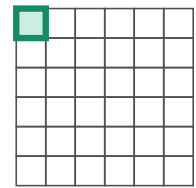
input with padding

$*$



filter 5×5

$=$



output 6×6

Stride 1. Output without padding: 2×2.

Stride and padding in PyTorch

```
conv = nn.Conv2d(in_channels=3, out_channels=16,
                 kernel_size=3, stride=2, padding=1)
x = torch.rand(1, 3, 300, 451)  # 1 image, 3 channels
print(conv(x).shape)
# torch.Size([1, 16, 150, 226])
```

A minibatch of 1 image of 300×451 pixels (the cat's size), in PyTorch's order (images, channels, height, width).

The output maps are 150×226. Where do these numbers come from?

Computing the size of the output

Input $H \times W$, filter $F \times F$, stride S, padding P.

Output height $=$

Output size: example 1

What is the output size?

- Input $227 \times 227 \times 3$
- 96 filters $11 \times 11 \times 3$
- Stride $S = 2$
- Padding $P = 1$

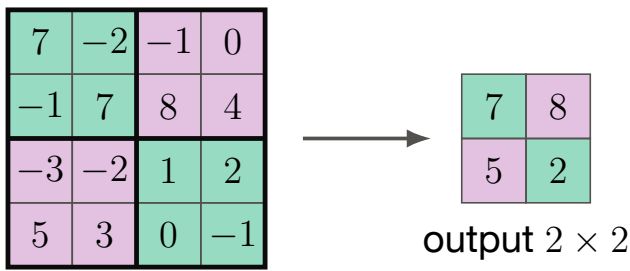
Output size: example 2

Which padding P gives an output with the height and width of the input?

- Input $25 \times 25 \times 3$
- 96 filters $3 \times 3 \times 3$
- Stride $S = 1$

Max pooling

Max pooling with a 2×2 window and stride 2: each output cell is the largest value in its window.



Why pooling helps

- Max pooling has no weights: there is nothing to learn.
- 2×2 pooling halves the height and width of each channel (rounded down).
- In the MNIST CNN below: 16 maps of 5×5, 400 values, instead of 16 maps of 10×10, 1,600.
- Less computation, and the first fully connected layer needs 48,120 weights and biases instead of 192,120.

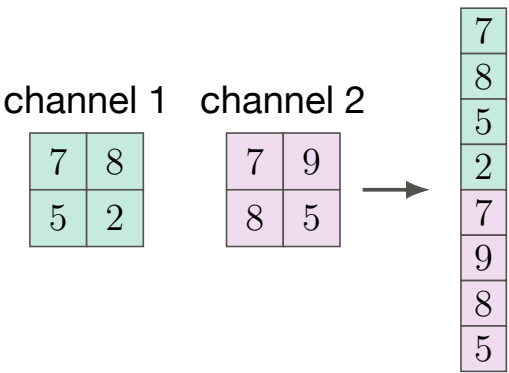
Max pooling in PyTorch

```
pool = nn.MaxPool2d(kernel_size=2, stride=2)
x = torch.tensor([[ 7, -2, -1,  0],
                  [-1,  7,  8,  4],
                  [-3, -2,  1,  2],
                  [ 5,  3,  0, -1]]).float()
print(pool(x.reshape(1, 1, 4, 4)))
# tensor([[[[7., 8.],
#           [5., 2.]]]])
print(sum(p.numel() for p in pool.parameters()))  # 0
```

The input is the 4×4 grid of the max-pooling slide, and the output is its 2×2 result.

The layer has 0 weights.

Flatten

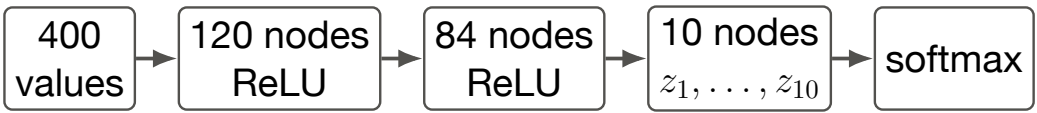


Flatten: write all values of the feature maps in one vector.

Order (PyTorch's `nn.Flatten`): channel 1 row by row, then channel 2.

Flatten has no weights: it only reshapes.

Fully connected layers and softmax



Each node: a weighted sum of all values of the previous layer, plus a bias.

Softmax: 10 probabilities that sum to 1, one per digit (Lecture 9); the loss is the cross-entropy.

Summary: layers of a CNN

- Convolution: C_{out} filters $F \times F \times C_{\text{in}}$ with biases; then ReLU.
- Stride S: the filter's step; padding P: rings of zeros.
- Max pooling: the largest value per window; no weights.
- Smaller maps: less computation in the next layers, and fewer weights in the first fully connected layer.
- Flatten, fully connected layers, softmax: class probabilities.

Checkpoint 2

A convolution layer: 6 input channels, 16 filters of size 5×5, each with a bias.

How many weights and biases?

Checkpoint 2

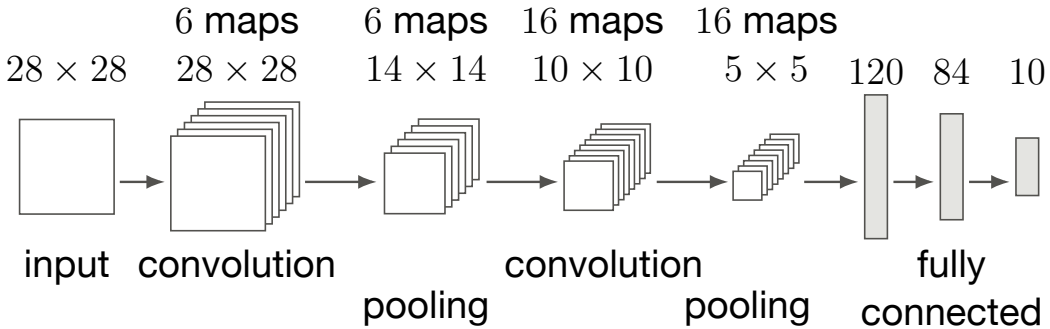
A convolution layer: 6 input channels, 16 filters of size 5×5, each with a bias.

How many weights and biases?

$$16 \times (5 \times 5 \times 6 + 1) = 16 \times 151 = 2{,}416$$

A CNN for the MNIST digits

MNIST (Lecture 3): 60,000 training and 10,000 test images of 28×28 pixels, divided by 255; image $x^{(i)}$, label $y^{(i)}$, pixel $x_{rc}^{(i)}$.



5×5 filters, stride 1, $P = 2$ then 0; 2×2 pooling, stride 2.

Shapes, weights and biases

layer	output shape ($C \times H \times W$)	weights and biases
convolution, 6 filters 5×5, padding 2, ReLU	$6 \times 28 \times 28$	156
max pooling	$6 \times 14 \times 14$	0
convolution, 16 filters 5×5, ReLU	$16 \times 10 \times 10$	2,416
max pooling	$16 \times 5 \times 5$	0
flatten	400	0
fully connected, ReLU	120	48,120
fully connected, ReLU	84	10,164
fully connected (output)	10	850
total		**61,706**

Convolution layer 1: $F = 5$, $P = 2$ (padding slide); $6 \times (5 \times 5 \times 1 + 1) = 156$.

The CNN in PyTorch

```
torch.manual_seed(0)
model = nn.Sequential(
    nn.Conv2d(1, 6, kernel_size=5, padding=2),  # 6x28x28
    nn.ReLU(), nn.MaxPool2d(2, stride=2),       # 6x14x14
    nn.Conv2d(6, 16, kernel_size=5),            # 16x10x10
    nn.ReLU(), nn.MaxPool2d(2, stride=2),       # 16x5x5
    nn.Flatten(),                               # 400
    nn.Linear(400, 120), nn.ReLU(),
    nn.Linear(120, 84), nn.ReLU(),
    nn.Linear(84, 10),                          # z_1..z_10
)
print(model(torch.zeros(64, 1, 28, 28)).shape)  # 64 images
# torch.Size([64, 10])
```

Comments: sizes $C \times H \times W$; $z_1, \ldots, z_{10}$: softmax inputs.

Training the CNN

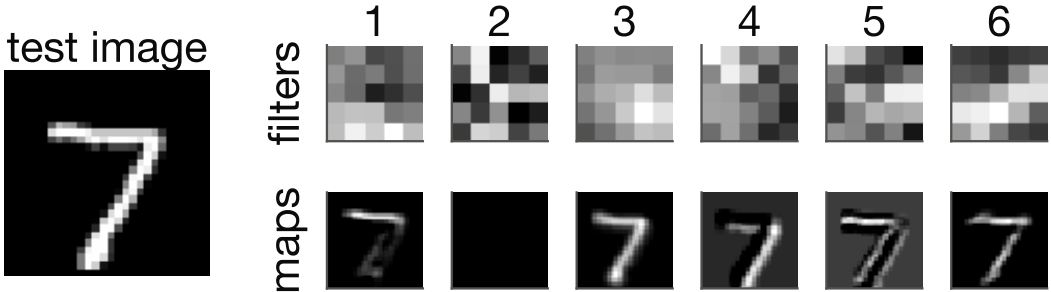
```
loss_fn = nn.CrossEntropyLoss()          # cross-entropy
opt = torch.optim.SGD(model.parameters(), lr=0.1)
for epoch in range(2):
    for xb, yb in train_loader:          # minibatches of 64
        loss = loss_fn(model(xb), yb)    # minibatch average
        opt.zero_grad()
        loss.backward()                  # backpropagation: g
        opt.step()                       # w <- w - 0.1 g
```

`train_loader`: the 60,000 training images, shuffled.

Accuracy on the test set

- Test accuracy 98.2%: 179 of 10,000 test images wrong.
- Training accuracy: 98.3%.

First convolution layer: filters and maps

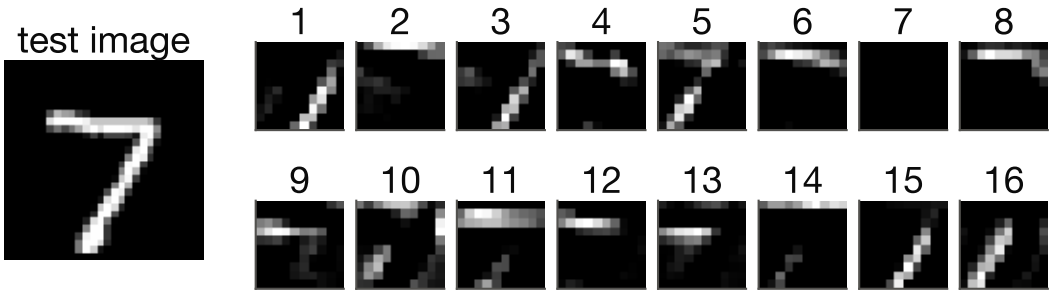


top: 6 learned filters; bottom: their maps after ReLU

map 2 is all 0 here

MNIST (LeCun, Cortes, Burges)

Second convolution layer: feature maps



its 16 maps for the same image, after ReLU

most maps are bright on the top or on the slanted stroke

Strengths and weaknesses of CNNs

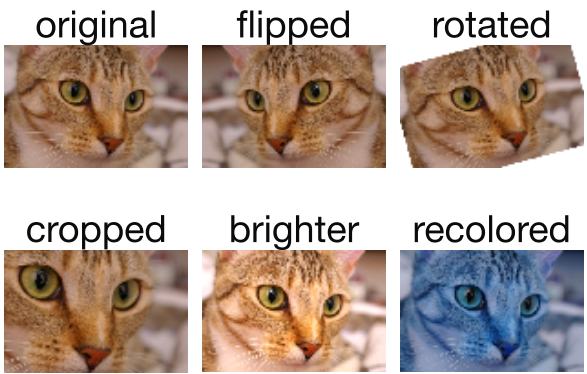
Strengths

- The ImageNet winners of 2012 to 2015 were CNNs; 98.2% on MNIST here.
- Layers can be added, removed or changed for a task.

Weaknesses

- Millions of weights: much computation (GPUs) and many labeled images.
- Little theory explains why such large networks predict well on new images.

Data augmentation

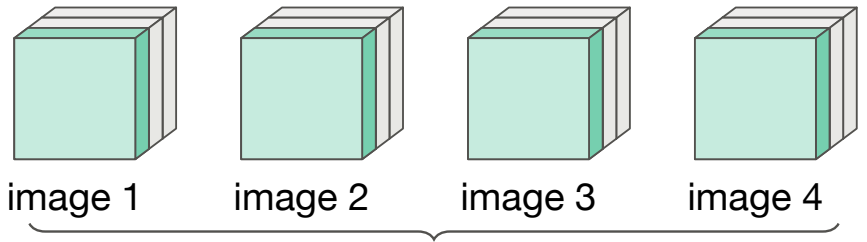


Changed copies of the training images, with the label of the original.

Only changes that keep the class: a 6 rotated by $180°$ becomes a 9.

More varied images usually reduce overfitting.

Batch normalization: which values



Batch normalization standardizes each channel separately.

Shaded: one channel in all images, all positions. They share one mean μ and one variance σ^2.

Batch normalization: the formula

For one channel, each value z becomes

$$\hat{z} = \frac{z - \mu}{\sqrt{\sigma^2 + \epsilon}} \ (\epsilon = 10^{-5}), \qquad \text{output } \gamma\,\hat{z} + \beta$$

μ, σ^2: the channel's mean and variance over the minibatch and all positions.

γ (scale) and β (shift): learned, one pair per channel.

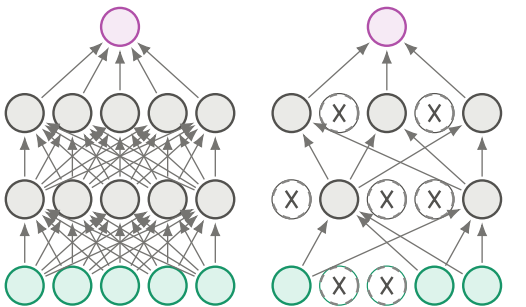
At test time, PyTorch uses running averages of μ and σ^2.

Ioffe and Szegedy, ICML, 2015

Batch normalization in PyTorch

```
torch.manual_seed(0)
bn = nn.BatchNorm2d(num_features=64)          # 64 channels
z = 3 + 5 * torch.randn(32, 64, 14, 14)       # 32 images
out = bn(z)                                   # training mode
print(f"{z.mean():.2f} {z.std():.2f}")        # 2.98 5.00
print(f"{out.mean():.2f} {out.std():.2f}")    # 0.00 1.00
print(bn.weight.shape)
# torch.Size([64])
```

Dropout



all nodes one training pass

In training, each node's output is 0 with probability p, anew in each forward pass.

Crossed: dropped in this pass.

At test time: no node is dropped. Dropout often reduces overfitting.

Drawing after Srivastava et al., *JMLR*, 2014, Fig. 1

Dropout in PyTorch

```
torch.manual_seed(0)
drop = nn.Dropout(p=0.2)
h = torch.ones(6)
print(drop(h))   # training mode
# tensor([0.0000, 1.2500, 1.2500, 0.0000, 1.2500, 1.2500])
drop.eval()
print(drop(h))   # test mode
# tensor([1., 1., 1., 1., 1., 1.])
```

$p = 0.2$: the probability of dropping.

Summary: training a CNN

- Training as in Lecture 9: cross-entropy, minibatches.
- MNIST: 98.2% test accuracy, 61,706 weights and biases.
- Data augmentation: changed copies, same labels.
- Batch normalization: standardize, then $\gamma\hat{z} + \beta$.
- Dropout: in training, outputs are 0 with probability p.

The full model in PyTorch

```
model = nn.Sequential(
    nn.Conv2d(1, 6, kernel_size=5, padding=2),
    nn.BatchNorm2d(6), nn.ReLU(), nn.MaxPool2d(2, stride=2),
    nn.Conv2d(6, 16, kernel_size=5), nn.BatchNorm2d(16),
    nn.ReLU(), nn.MaxPool2d(2, stride=2),
    nn.Flatten(),
    nn.Linear(400, 120), nn.ReLU(), nn.Dropout(p=0.2),
    nn.Linear(120, 84), nn.ReLU(), nn.Dropout(p=0.2),
    nn.Linear(84, 10),
)
model.eval()  # dropout off; batch norm: running averages
print(model(torch.zeros(64, 1, 28, 28)).shape)
# torch.Size([64, 10])
```

`model.eval()`: testing; `model.train()`: training.