Convolutional networks

Flowers102

LECTURE 12 | GÉRON CH. 12

Applications of Machine Learning

BSc Mathematics of Artificial Intelligence · Third year

Where we are

Lecture 11 trained a fully connected network with no labels, on 28×28 greyscale images (Fashion-MNIST) flattened to 784 numbers:

- An encoder squeezed each image to a code of 32 numbers and a decoder rebuilt it; the bottleneck stopped the network from copying
- A linear autoencoder reaches the principal subspace of Lecture 6; two ReLU layers went a sixth below that floor
- The code retrieved and clustered as well as the 784 pixels, and no better

WHAT BREAKS WHEN THE IMAGE GETS BIGGER

Flattening a 224×224 colour image gives 150,528 inputs. A single dense layer of 1,000 units on that is **150 million parameters** — for one layer, on one image size. Every one of them has to be learned from data, and none of them knows that two adjacent pixels are related.

Today's method fixes both problems at once, and the derivation says exactly how much it saves.

Today

1. The task, and why a flattened image is the wrong representation
2. **The mathematics** — what weight sharing saves, equivariance, and how pooling buys invariance
3. **The method** — convolution, pooling, an architecture, and training it
4. What it learned: the first layer's filters and the feature maps

Open Lecture 12 in Colab

The stakeholder

A horticultural wholesaler receives mixed pallets of cut stems and has to sort them by species before they reach the auction floor.

- One overhead camera per lane, one photograph per stem
- A human inspector currently does this and is the bottleneck
- They want the machine to name the species from the photograph

Same shape of problem as the garment sorter in Lecture 7. Everything that makes it harder is in the pixels.

What they actually asked for

> **IN THEIR WORDS**
>
> “Tell us which of our species it is. If you cannot tell, say so and we will look at it ourselves — but do not tell us the wrong one.”

Two requirements hiding in one sentence: a classifier over a fixed list of species, and a usable notion of *confidence*.

We build the first today. The second waits until the numbers are good enough to be worth calibrating.

> **FAILURE CONDITION — A CLASSIFIER WITH NO WAY TO ABSTAIN**
>
> The stakeholder asked to be told when the model cannot tell. A softmax always returns a class, and its largest probability is not a confidence you have calibrated. Deliver it as one and you have answered a different requirement while appearing to meet this one.

The dataset

`torchvision.datasets.Flowers102` — 8,189 photographs of 102 flowering species common in the United Kingdom.

```
from torchvision.datasets import Flowers102

train = Flowers102("data", split="train", download=True)
val   = Flowers102("data", split="val",   download=True)
test  = Flowers102("data", split="test",  download=True)
```

About 345 MB. It is one of the datasets the textbook names for this chapter, and the split used here is the one shipped with it — so every result in the literature is on the same 6,149 test images.

What the camera sees

12 of the 102 species, one image each, resized to 128 × 128



Different scales, different backgrounds, different lighting — and several of these species are hard to tell apart by eye.

What is hard here, precisely

- **Position** — the flower is not centred, and not at a fixed scale
- **Background** — leaves, soil, other flowers, a gardener's hand
- **Between-class similarity** — several of the 102 species are separated by petal count and little else
- **Within-class variation** — the same species at two stages of opening

Only the first of those has a structural answer. It is the answer this chapter is about.

The split we are given

Split	Images	Per class	Used for
train	1,020	10, exactly	fitting
val	1,020	10, exactly	choosing
✗ test	✗ 6,149	20–238	reporting, once

✗ There are six times as many test images as training images.

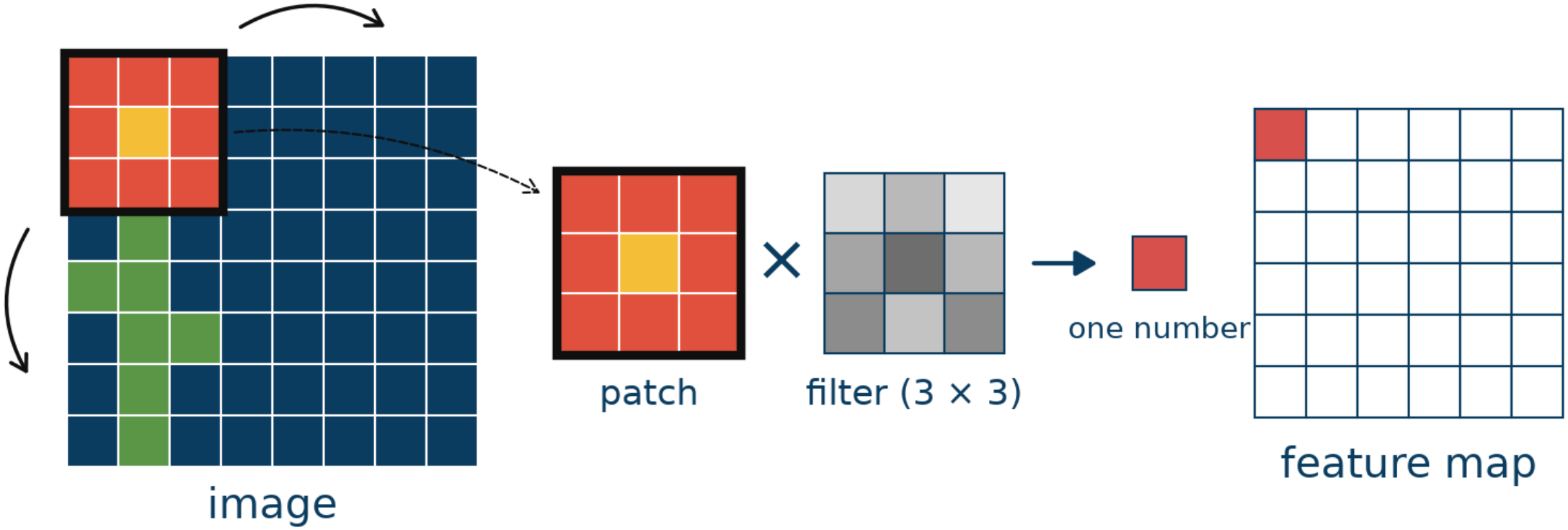
Why the training split is so small

Because somebody had to look at every photograph and name the species.

> **THE STANDING CONSTRAINT OF APPLIED VISION**
>
> Images are cheap and labels are not. Every application in this part of the course has more pixels than it can use and fewer labels than it wants.

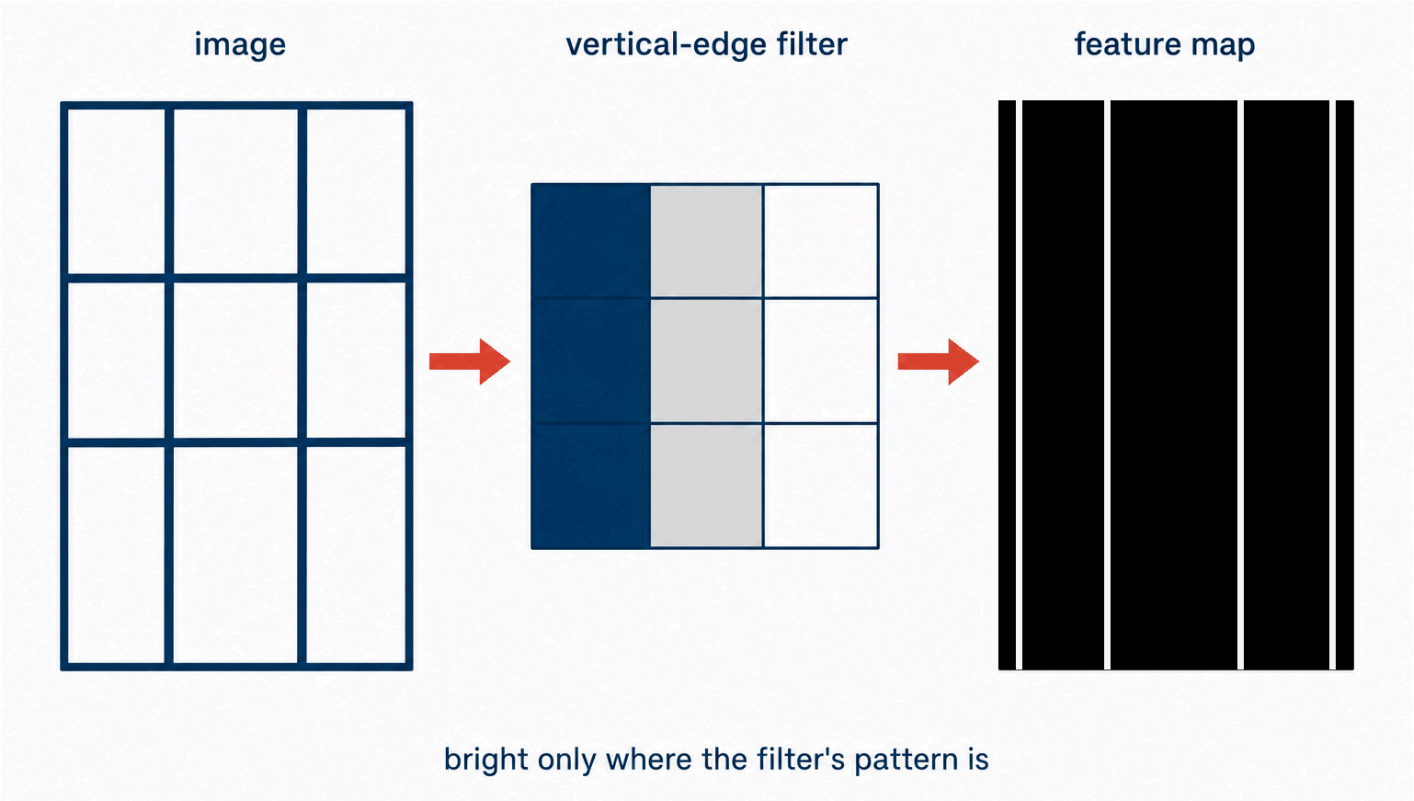
Design for that. A method that needs thousands of labelled examples per class is not a method you can deploy here.

What a convolution computes



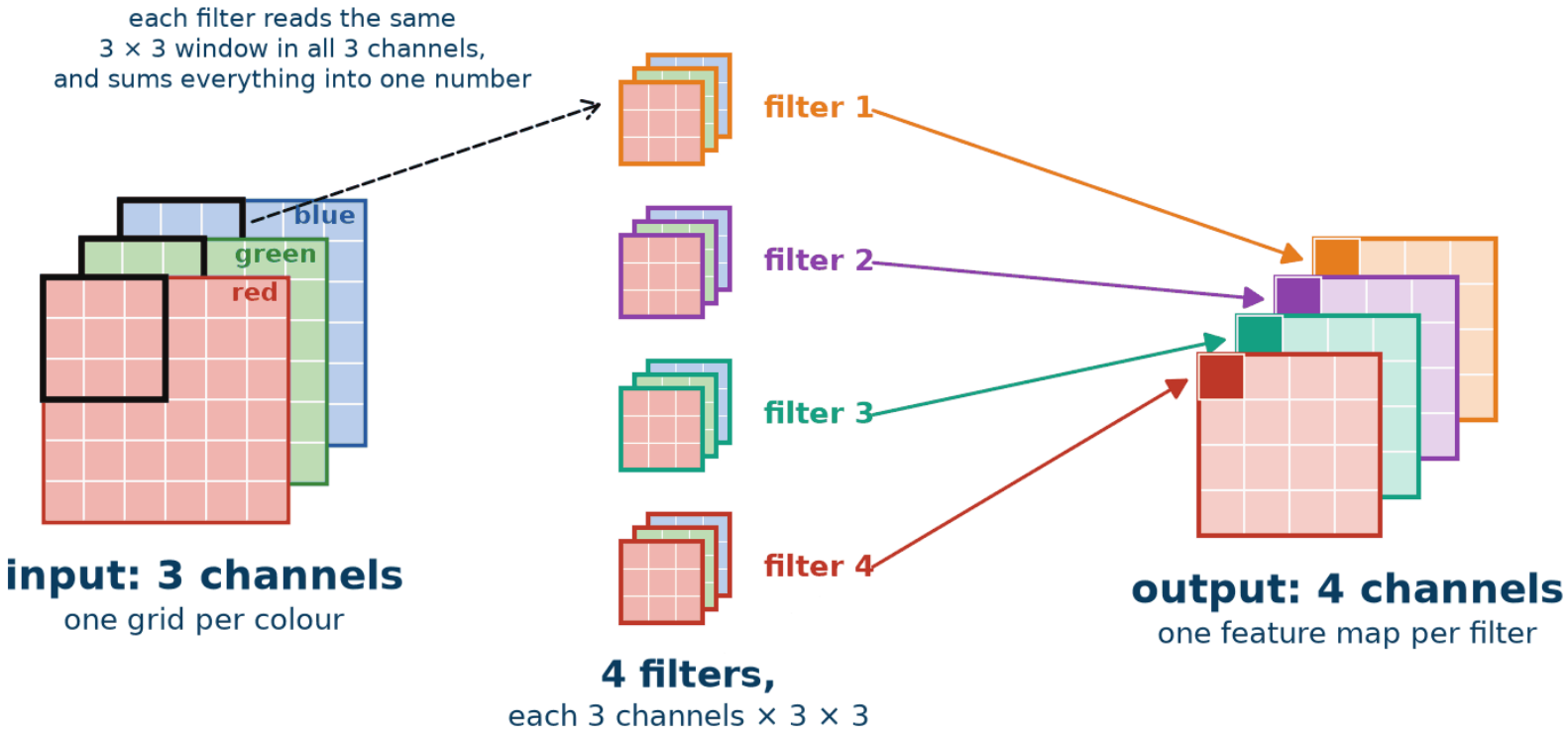
Lay a small grid of weights on a 3×3 patch, multiply and add: one number. Slide one pixel and repeat. The numbers form the **feature map**, 6×6 from an 8×8 image.

What a filter looks for



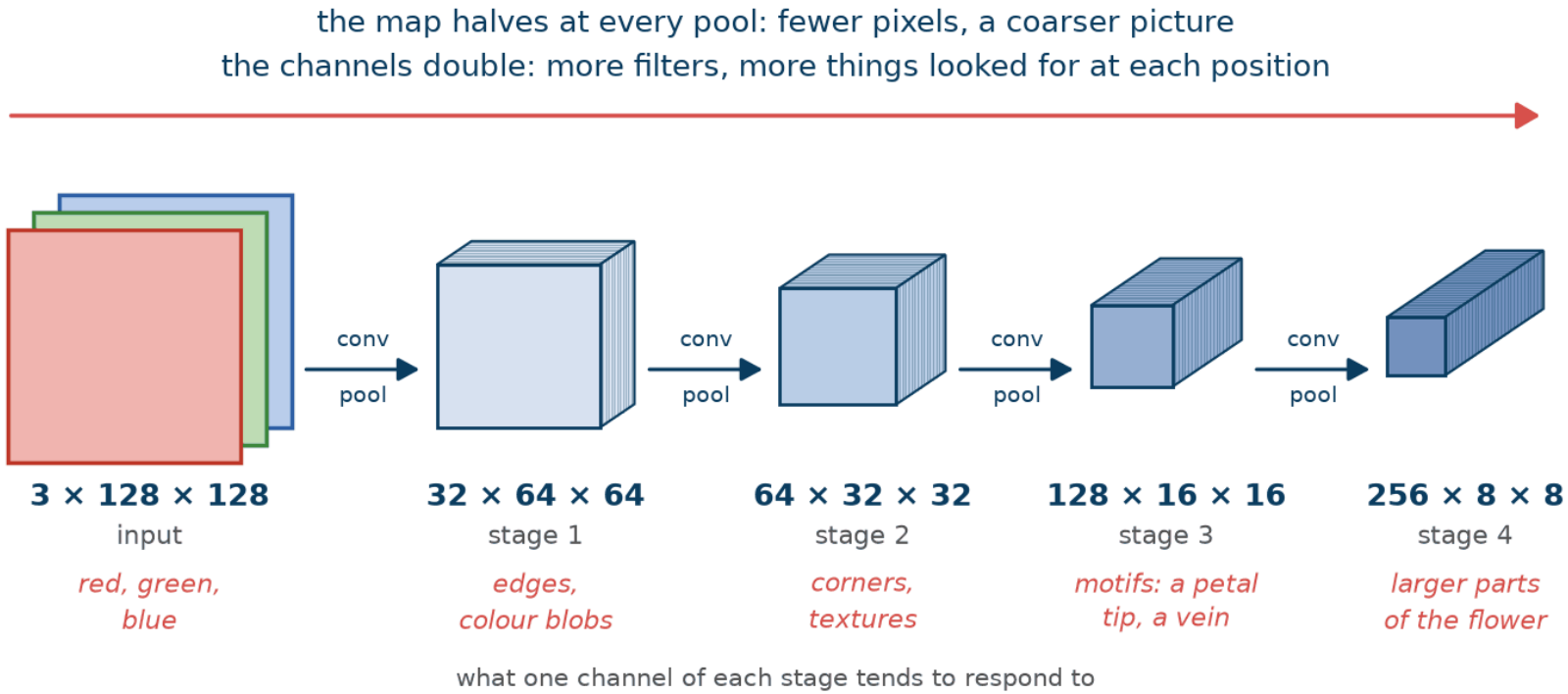
A filter is a pattern: the sum is large where the patch looks like it. This one sees vertical edges, not horizontal ones. A network learns its filters.

Input channels and output channels



A filter spans every input channel: $C_{\text{in}} \times k \times k$ weights, one output map. Use C_{out} filters and the output has C_{out} channels, the input of the next layer.

Inside the network, a channel is not a colour



After the first layer a channel is one filter’s map, not a colour. Each pool halves the map and each stage doubles the channels: fewer positions, more aspects of the input described at each one.

The obvious thing to try first

You have a multilayer perceptron that works. Resize every photograph to 128 × 128, flatten it, and feed it in.

	Count
Inputs, 128 × 128 × 3	49,152
Outputs, 32 values at every pixel position	524,288
✗ **Weights in one fully connected layer between them**	✗ **25,769,803,776**

✗ **That is 96 GB of float32 for one layer. The full argument is this lecture's mathematics block; here it is just a reason to do something else.**

What weight sharing saves

The same output, $C_{\text{out}} \times H \times W$, two ways. A **dense layer** joins each of the $C_{\text{in}}HW$ input values to each of the $C_{\text{out}}HW$ output values. A **convolution** has C_{out} filters of $C_{\text{in}} \times k \times k$ weights, **the same at every position**.

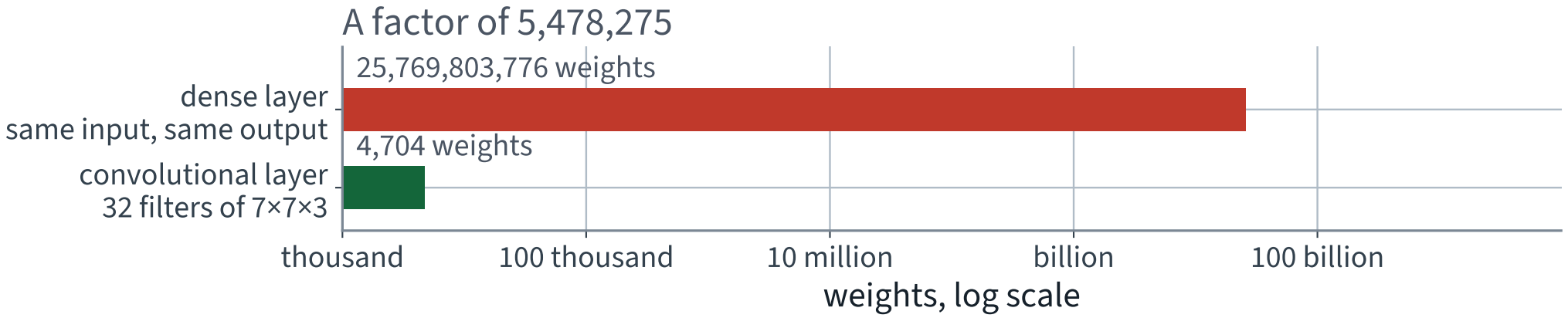
$$\frac{\overbrace{(C_{\text{in}}HW)\,(C_{\text{out}}HW)}^{\text{dense layer}}}{\underbrace{C_{\text{out}}C_{\text{in}}k^2}_{\text{convolution}}} = \frac{H^2W^2}{k^2}$$

THE CHANNELS CANCEL: THE IMAGE SIZE, SQUARED, OVER THE WINDOW

5,478,275

times fewer weights at $H = W = 128$, $k = 7$: 4,704 against 25,769,803,776.

The same output shape, two ways



A logarithmic axis, and the two bars still barely fit on the same figure.

Equivariance, precisely

Sharing the weights assumes that a pattern means the same thing wherever it appears. Let $T_{\mathbf{v}}$ be the operator that shifts an image by the vector $\mathbf{v}$:

$$(T_{\mathbf{v}} x)(\mathbf{p}) \;=\; x(\mathbf{p} - \mathbf{v})$$

A map f is **equivariant** to translation when

$$f(T_{\mathbf{v}}\, x) \;=\; T_{\mathbf{v}}\, f(x) \qquad \text{for every } \mathbf{v}$$

Shift then compute, or compute then shift — the same answer. A convolution has this property because its weights do not depend on the position; it holds exactly away from the border, where zero padding invents input that was not there.

Pooling

After a couple of convolutions, halve the height and width by keeping only the largest value in each 2×2 block.

- Three quarters of the activations disappear, and with them three quarters of the memory and the arithmetic in every later layer
- It has **no parameters** — nothing about it is learned
- Each later unit therefore sees a larger piece of the original image

There is a fourth consequence — invariance — and it comes two slides on.

Max pooling, on numbers

4 × 4 feature map

5	3	8	1
2	9	4	0
1	6	7	3
4	2	5	8

after 2 × 2 max pool

9	8
6	8

Sixteen numbers in, four out, and the four are the largest of each block.

How pooling manufactures invariance

A map g is **invariant** to translation when

$$g(T_{\mathbf{v}}\, x) \;=\; g(x) \qquad \text{for every } \mathbf{v}$$

The output does not move at all, so it no longer says *where*. Classification wants that; equivariance alone does not give it.

A 2×2 max pool is invariant to any shift that keeps the maximum inside its own window — so half the one-pixel shifts, in each direction.

- Stack four pooling stages and the tolerated shift grows to ± 8 pixels in the last feature map
- A global pool over the whole map is invariant to *every* shift — there is no position left to be sensitive to

The invariance is approximate and it is built, one factor of two at a time. Nothing in the architecture declares it.

A convolution, as arithmetic

For an input x with C channels and a kernel w of size $k \times k$, the output at position (i, j) of filter f is

$$z_{ij}^{(f)} = b^{(f)} + \sum_{c=1}^{C} \sum_{u=0}^{k-1} \sum_{v=0}^{k-1} w_{c,u,v}^{(f)} \, x_{c,\, i+u,\, j+v}$$

A dot product between the kernel and a window of the input, repeated at every position. The sum over c is why a filter reads *all* the input channels at once.

`nn.Conv2d`, and the defaults you did not ask for

```
nn.Conv2d(
    in_channels=3,        # must match the previous layer's output
    out_channels=32,      # how many filters -> how many output maps
    kernel_size=7,
    stride=1,             # the default
    padding=3,            # NOT the default; the default is 0
    bias=False,           # a batch-norm follows, and it has its own shift
)
```

The default you did not ask for: **padding defaults to zero**, which silently shrinks every feature map by $k - 1$ and will eventually give you a shape error twelve layers later.

The architecture

Stage	Layers	Output
1	conv 7×7 ×32, conv 3×3 ×32, pool	32 × 64 × 64
2	conv ×64, conv ×64, pool	64 × 32 × 32
3	conv ×128, conv ×128, pool	128 × 16 × 16
4	conv ×256, pool	256 × 8 × 8
✓ head	**✓ flatten, dense 256, dropout, dense 102**	**✓ 102**

Channels double as the map halves — the textbook's own rule of thumb: what pooling takes away in resolution, the extra filters give back in channels.

The architecture, in code

```
def conv_block(c_in, c_out, k=3):
    return [nn.Conv2d(c_in, c_out, k, padding=k // 2, bias=False),
            nn.BatchNorm2d(c_out), nn.ReLU()]

net = nn.Sequential(
    *conv_block(3, 32, k=7), *conv_block(32, 32), nn.MaxPool2d(2),
    *conv_block(32, 64),     *conv_block(64, 64), nn.MaxPool2d(2),
    *conv_block(64, 128),  *conv_block(128, 128), nn.MaxPool2d(2),
    *conv_block(128, 256),                        nn.MaxPool2d(2),
    nn.Flatten(),
    nn.Linear(256 * 8 * 8, 256), nn.ReLU(),
    nn.Dropout(0.5),
    nn.Linear(256, 102),
)
```

Thirty modules. Convolution and pooling are today's; everything else was on a slide in Lectures 7 and 8.

Count the parameters, and look at where they are

Part	Parameters	Share
Every convolution and batch-norm layer together	586,720	12.2%
✗ One dense layer, 16,384 → 256	**✗ 4,194,560**	**✗ 87.3%**
Output layer, 256 → 102	26,214	0.5%
✓ Total	**✓ 4,807,494**	100%

✗ Nine parameters in ten are in the layer that is not convolutional.

The training loop — unchanged

```
opt   = torch.optim.Adam(net.parameters(), lr=3e-4)
lossf = nn.CrossEntropyLoss()

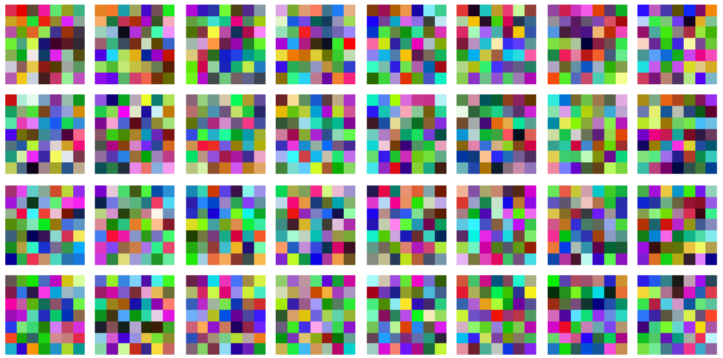
for epoch in range(30):
    net.train()
    for xb, yb in train_loader:
        opt.zero_grad()
        loss = lossf(net(xb), yb)
        loss.backward()
        opt.step()
```

Exactly the five lines from Lecture 8. Convolution changed the model; it did not change the loop, because autograd does not care what the modules are.

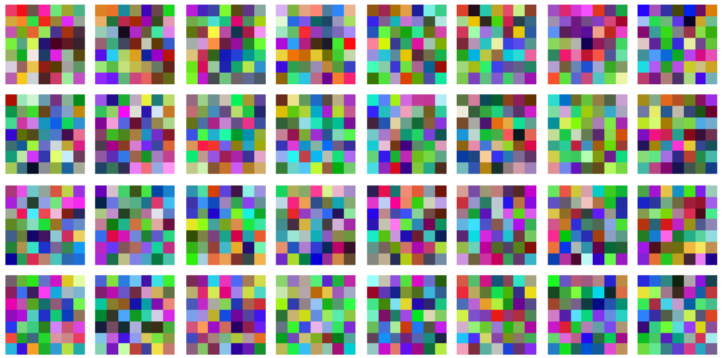
Thirty-two filters, before and after

All 32 filters of the first layer, 7 × 7 × 3, each rescaled to its own range

at initialisation



after 30 epochs



Look at the two blocks for ten seconds, and say what changed.

It moved a little. It did not become anything

```
import torch.nn.functional as F

# init_filters: a copy of net[0].weight, taken before the first step
before, after = init_filters.flatten(), net[0].weight.flatten()
print(f"cosine similarity {F.cosine_similarity(before, after, dim=0):.4f}")
print(f"mean |change| / mean |weight| "
      f"{(after - before).abs().mean() / before.abs().mean():.1%}")
```

	Value
✓ cosine similarity, all 4,704 weights before against after	**0.9880 ✓**
mean absolute change, as a share of the mean weight	14.1%
weights that changed sign	160

✗ The weights moved. No structure appeared — the right-hand block is still noise, and thirty epochs of training bought it.

Two reasons, and both are worth knowing

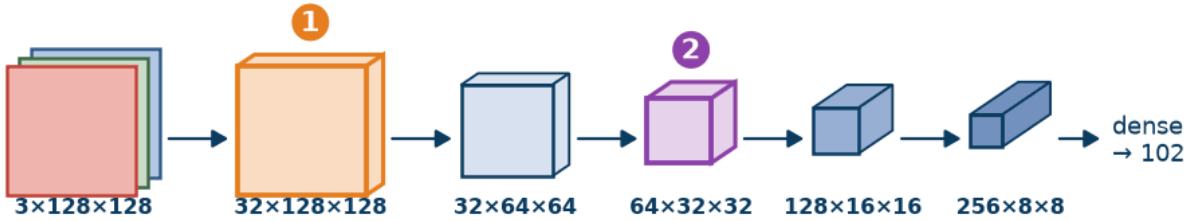
1. **The gradient that reaches it is small.** Ten layers of chain rule separate the loss from these weights, and 1,020 images give 960 optimiser steps in total
2. **Its scale does not matter.** A batch norm follows immediately, and normalising the output makes the overall size of these weights irrelevant to the function — so there is no gradient pressure on it at all

The textbook picture of a first layer — colour blobs, oriented edges — is real, and you will see it next lecture. It comes from a corpus a thousand times this size.

What a filter does to one photograph

One test photograph, through the trained network.

Each tile is one output channel of a layer:
the map that one filter makes of the photograph.
Bright = the filter responds strongly there.



the network of this lecture: convolutions and pooling in four stages, then the dense head



After the first convolution each channel is still a picture of the flower. After stage 2 the maps are coarse: each keeps one aspect of it, not the picture.

Reading a feature map

- Bright means *this filter fired strongly here* — nothing more mystical than that
- The spatial layout survives: a bright patch is where in the image the pattern was found
- After four pooling stages the map is 8×8, so one cell answers for a 256-pixel square of the original

That last line is a loss of information we chose. It is the right choice for naming a species, and the wrong one for any task that must say *where*.

The notebook for this lecture

Open Lecture 12 in Colab

- **Runs on free CPU.** It trains the network for 30 epochs, and every number on these slides is one it printed
- The dense-against-convolutional parameter count, computed rather than quoted — the arithmetic you can do on paper first
- The network built, its parameters counted by part, trained, and the first layer's filters and feature maps

> **WHAT TO CHANGE**
>
> Set `padding=0` on every convolution and push one image through the network. Every map shrinks, and the first linear layer no longer matches what reaches it.

NOT PRESENTED — FOR AFTERWARDS

Exercises Lecture 12

five, in the style of the written paper

Exercises — Lecture 12

1 A dense layer mapping a $3 \times 128 \times 128$ input to a $32 \times 128 \times 128$ output has $n_{\text{in}} n_{\text{out}}$ weights. Give the convolutional count for a 7×7 kernel, and say which symbols are absent from it. [5]

2 State the difference between equivariance and invariance, say which of the two pooling builds, and why a species classifier wants it. [4]

3 Convolution is equivariant to translation on the interior of an image but not at the border. Give the reason. [3]

Solutions on Lecture 13’s deck.

Exercises — Lecture 12 (continued)

4 The lecture's network has 4,807,494 parameters. Name the layer that holds most of them, give its count from its shapes, and say why the ten convolutions together hold so few. [4]

5 After thirty epochs the first layer's 32 filters are still almost the vector they started as: cosine similarity 0.9880. Give the two reasons, and say what a first layer needs before colour blobs and oriented edges appear. [4]

Solutions on Lecture 13's deck.

THE FIVE SET AT THE END OF LECTURE 11

Solutions Lecture 11

not part of this lecture

Solutions — Lecture 11

1. At a code of 32 numbers, the test error per pixel is 0.01519 for PCA with 32 components, 0.01523 for a linear…

✓ Without ReLUs the network compresses along flat directions only, and PCA is already the best flat compression, so training lands on it. The ReLUs let the map curve: the sixth below PCA is what no flat compression gets.

- A linear encoder and decoder make one linear map, the class PCA is the best of
- The gain is the whole case for the non-linear layers

2. At a code of 128 numbers the non-linear autoencoder's test error is 0.01179, above PCA's 0.00641. A colleague…

✓ It compares an exact optimum with a trained approximation. PCA is computed exactly; the network, one hidden layer of 256 and twenty epochs, has not found the best of its larger class. More capacity is not more accuracy until the optimiser has found it.

- The error is the run's, not the class's
- The fix is a budget: a wider hidden layer, more epochs

Solutions — Lecture 11 (2)

3. The code size is set equal to the number of pixels and nothing else constrains the network. State what the…

✓ The identity, with zero loss. The code is a re-encoding of the pixels that loses nothing and learns nothing.

- With $d = n$ copying fits through the code, so the identity reaches zero loss and training finds it
- A bottleneck narrower than the input is the only thing that forbids copying; without it the code carries exactly the pixels, differently arranged

4. On noisy test images, the error against the clean image is 0.05264 for the noisy image itself, 0.02408 for…

✓ The denoising one: phone photographs are noisy, and there it cuts the plain one's error by a third. The plain one copies part of the noise; the denoising one learned noisy-in, clean-out. On clean images it is worse.

- The plain autoencoder already removes about half the noise: noise does not fit through 32 numbers
- If the photographs turn out clean, the plain one is the better choice

Solutions — Lecture 11 (3)

5. Nearest-neighbour retrieval in a 32-number autoencoder code scores the same precision at ten as in the 784…

✓ A twenty-fourfold saving in storage and in the cost of every comparison, at equal precision. It could not beat the pixels because the reconstruction loss contains no information about which images are similar: the code is a compressed account of pixel distance.

- A nearest-neighbour search compares the query with every item in the catalogue, so the length of the code is the cost of the search
- A representation that is better than its input at a task needs a training signal about that task, and reconstructing the input supplies none

LECTURE 13 · GÉRON CH. 12

Transfer learning

Why features learned on a large labelled corpus transfer to these flowers
Freeze the backbone and train a head; then let the last block move, at its own learning rate
Augmentation — and why it belongs to the training loader only

Run the Lecture 12 notebook first