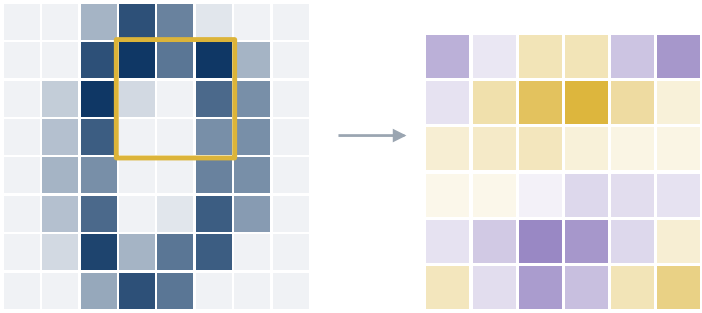


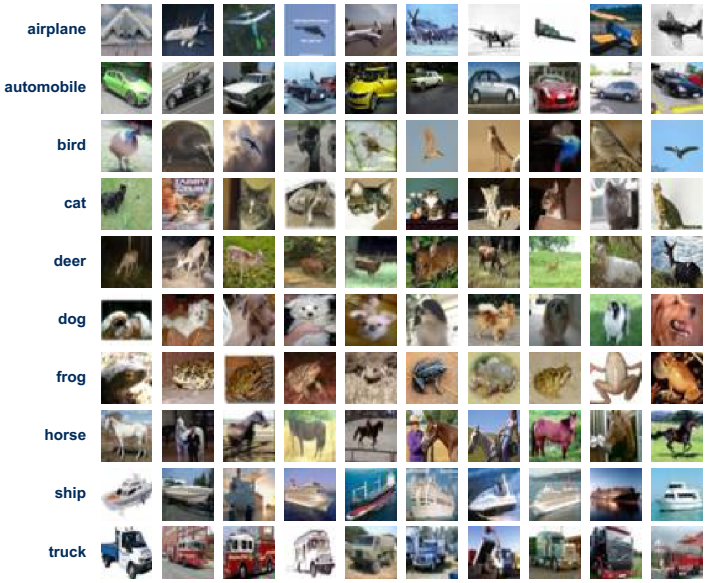
UNIT 03 · CHAPTER 2

Convolutional Neural Networks

CSE 30124 · Introduction to Artificial Intelligence
William Theisen · University of Notre Dame

Images as Inputs

In the last two chapters our networks took a handful of numbers from each game's box score and told us whether Notre Dame won, and backpropagation found their weights. Nothing about a network says its inputs have to come from football. A photo is a grid of numbers too, three of them at every pixel for how much red, green and blue there is.



Pictured above is **CIFAR-10**, `60,000` tiny color photos, `32` × `32` pixels each, of `10` kinds of thing from airplanes to trucks. We train on `50,000` of them and keep the other `10,000` aside as unseen photos.

The simplest way to give a photo to a feed-forward network is to **flatten** it, reading the pixels off row by row (all three colors of each) into one long list of $32 \times 32 \times 3 = 3{,}072$ inputs.

Feed-Forward Networks on Photos

So let's do exactly that, with hidden layers of `512` and `256` neurons and one output for each of the `10` classes:

```
import torch
from torch import nn

ffn = nn.Sequential(
    nn.Flatten(),  # 3 x 32 x 32 -> 3,072
    nn.Linear(3072, 512), nn.ReLU(),
    nn.Linear(512, 256), nn.ReLU(),
    nn.Linear(256, 10),
)
x = torch.zeros(5, 3, 32, 32)  # 5 blank photos
print(ffn(x).shape)
print(sum(p.numel() for p in ffn.parameters()))
```

OUTPUT

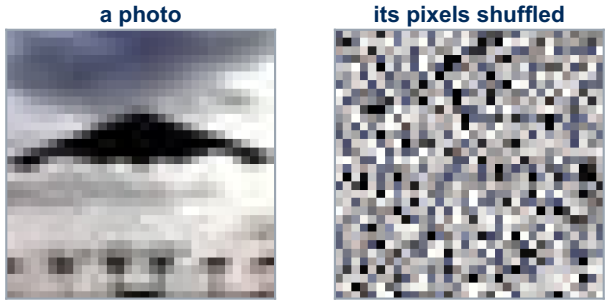
```
torch.Size([5, 10])
1707274
```

That's `1,707,274` weights, `1,573,376` of them in the first layer alone, where every one of the `3,072` inputs connects to every one of the `512` neurons. Trained for `10` epochs, it gets **`49.5%` of the unseen photos right**. Guessing would get `10%`, so it has learned something.

Unfortunately, the same kind of network gets `97.3%` of sklearn's `8` × `8` handwritten digits right. Small, centered, grey digits are easy, photos of cats and dogs taken from every angle **are not**.

Spatial Structure

Part of the problem is that flattening hands the network `3,072` numbers in a row, and nothing tells it that one pixel sits next to another. To check, we can shuffle the pixels of every photo into one fixed random order (the same order for every photo, so nothing is lost) and train again.



Pictured on the right is our airplane after the shuffle, unreadable to us. The feed-forward network gets `48.8%` of the shuffled photos right, about what it got before (`49.5%`). To a fully connected network **a photo is just 3,072 unrelated numbers**, so scrambling them costs it nothing.

Spatial Structure, Continued

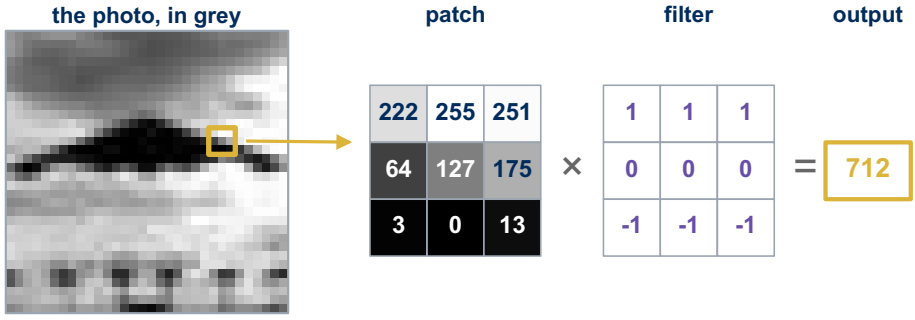
Unfortunately, that's also why it struggles. An edge is an edge wherever it is in a photo, but weights learned for an edge in one corner say nothing about the same edge anywhere else, so **every pattern has to be learned again at every place it could appear**.

It's also expensive. Our first layer already has `1,573,376` weights for a `32` × `32` photo. A `12`-megapixel photo off your phone has `36,578,304` inputs, and the same `512` neurons would need about `18.7` billion weights, for one layer.

What we'd like instead is a model that looks at **a small neighborhood of pixels at a time**, and uses the same weights wherever that neighborhood happens to be.

Filters

So take a `3 × 3` grid of weights, lay it over a `3 × 3` patch of the photo, multiply each pixel by the weight on top of it, and add up the nine products. To keep the numbers simple we'll use the airplane in grey for now (the average of its three colors, from `0` for black to `255` for white).



This grid has `1`s along the top, `0`s through the middle and `-1`s along the bottom, so it adds up the top row of the patch and takes away the bottom row. The patch above sits where bright sky meets the plane's dark wing, so it gets a big positive number ($222 + 255 + 251 - 3 - 0 - 13 = 712$). A patch of plain sky would get about `0`, and one with the dark part on top would come out negative.

A small matrix of weights encoding some pattern we're looking for in an image is called a **filter** or a kernel (the words are interchangeable). This one is a horizontal edge filter, it looks for places where the photo gets darker as we move down.

Convolution

One patch gives one number. To find every horizontal edge in the photo, we slide the filter across it one pixel at a time and do the same multiply and add at every stop. A `3 × 3` filter fits in `30` places across a `32` pixel row and `30` down, so we get a `30 × 30` grid of numbers:

```
import numpy as np
import torch

photo = torch.tensor(np.load('cifar-plane.npy'))
grey = photo.float().mean(dim=2).round()  # 32 x 32
K = torch.tensor([[ 1.,  1.,  1.],
                  [ 0.,  0.,  0.],
                  [-1., -1., -1.]])

out = torch.zeros(30, 30)
for i in range(30):
    for j in range(30):
        patch = grey[i:i+3, j:j+3]
        out[i, j] = (patch * K).sum()
print(out.shape)
print(out[11, 20:26].long())  # row 11, through the wing
```

OUTPUT

```
torch.Size([30, 30])
tensor([369, 528, 658, 712, 685, 558])
```

The `712` from the last page is in row `11`, in a run of big numbers along the top of the wing.

Cross-Correlation

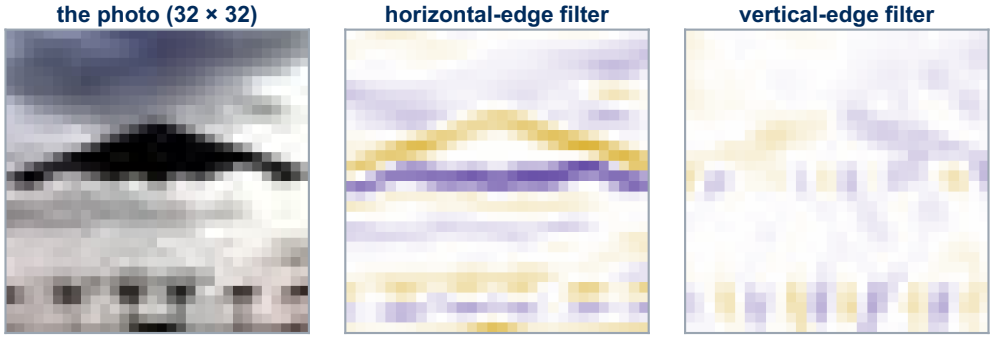
Written out, the output at row i and column j for an image I and a kernel K adds up the kernel's weights times the pixels under them, with m and n running over the kernel's rows and columns:

cross-correlation: what CNNs actually compute

$$S(i, j) = (I \star K)(i, j) = \sum_m \sum_n I(i + m,\ j + n)\, K(m, n)$$

CNNs are named after a mathematical operation called a convolution, in which two functions are combined to create a third. A true convolution flips the kernel before sliding it ($I(i - m,\ j - n)$ instead of $I(i + m,\ j + n)$), and what we've done, without the flip, is called **cross-correlation**. In practice nearly every library computes cross-correlation and calls it convolution. Since the kernel's weights are learned, a network that needed a flipped kernel would simply learn it flipped.

Feature Maps

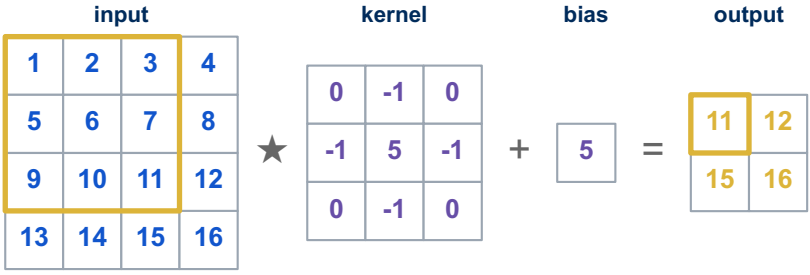


The grid a filter makes is called a **feature map**, one number for every position, saying how strongly the patch there matches the filter's pattern. Pictured above, the horizontal edge filter lights up (gold) along the top of the plane, where sky sits above the dark wing, and goes negative (purple) along its underside, where the dark is above the light. Its most negative value, `-712`, is two rows below its largest.

Turning the filter on its side gives a vertical edge filter, with `1`s down the left and `-1`s down the right. Its map has little to say about the plane, whose edges are mostly flat, but picks out the columns of the building underneath. **Different filters find different patterns in the same image**, so a layer of a CNN usually applies several filters and keeps every map.

Output Size and Padding

Notice that our 32 × 32 photo gave a 30 × 30 map. Applying a filter typically shrinks it, since there are only so many places we can fit the whole filter. A $k \times k$ filter on an $n \times n$ image fits in $n - k + 1$ places across.

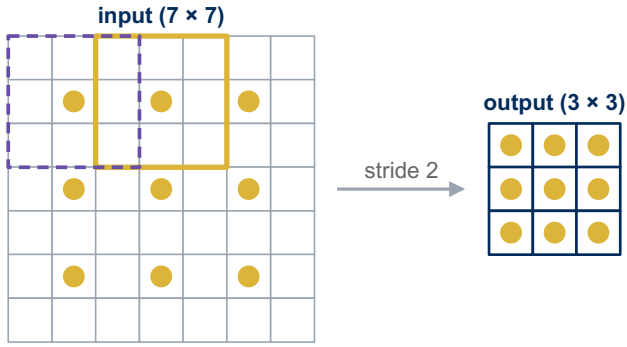


Pictured above is a smaller case, a 4 × 4 image holding the numbers 1 to 16. The kernel fits in only four places, so the output is 2 × 2. Each output also gets a **bias** of 5 added, the same job the bias did in our neurons. The top left output is $5 \cdot 6 - 2 - 5 - 7 - 10 + 5 = 11$.

Shrinking is often a nuisance. Ten layers of 3 × 3 filters would take our photo down to 12 × 12, and pixels on the border get looked at fewer times than pixels in the middle. The usual fix is **padding**, a ring of zeros around the image before the filter slides over it. With one ring our photo becomes 34 × 34, the filter fits in 32 places across, and the feature map comes out the same size as the photo.

Stride

So far the filter has moved one pixel at a time. It can also skip, and the number of pixels it moves at each step is called the **stride**. Pictured below, a 3×3 filter with a stride of 2 makes only three stops across a 7×7 input, so the output is 3×3.



Putting padding p and stride s together, an n × n input and a k × k filter give an output of size

$$o = \left\lfloor \frac{n + 2p - k}{s} \right\rfloor + 1 \quad \text{[2.1]}$$

rounded down

For our photo with padding 1 and stride 1 that's $(32 + 2 - 3)/1 + 1 = 32$. With stride 2, $(32 + 2 - 3)/2 = 15.5$, which rounds down to 15 (the filter can't make half a stop), so the output is 16×16. A bigger stride is a cheap way to shrink a map, at the price of **skipping over positions where a pattern might have matched**.

Channels

So far our filter has only looked at the grey photo. A real photo has three numbers at every pixel, so it's really three 32×32 grids stacked up, one each for red, green and blue, called **channels**.

A filter for a color photo is $3 \times 3 \times 3$, a 3×3 grid of weights for each channel. At each stop it multiplies all 27 pixel values by their weights and adds them up, plus a bias, so it still gives one number per position and one feature map. Because it sees all three colors at once, it can look for **color patterns as well as light and dark ones**, like a red edge against a blue sky.

The same goes for later layers. A layer with 32 filters makes 32 feature maps, which together are a $32 \times 32 \times 32$ stack with 32 channels, and a filter in the next layer is $3 \times 3 \times 32$, looking at a patch of every map at once.

Learned Filters

Our two edge filters were designed by hand, and for decades computer vision did exactly that, with filters for edges, corners and blobs feeding a classifier. Unfortunately it's hard to know in advance which patterns tell a cat from a dog. So instead of hard-coding what the filters look for, a CNN treats the numbers in each filter as weights and **lets them be learned during training**, by the same gradient descent and backpropagation as every other weight.

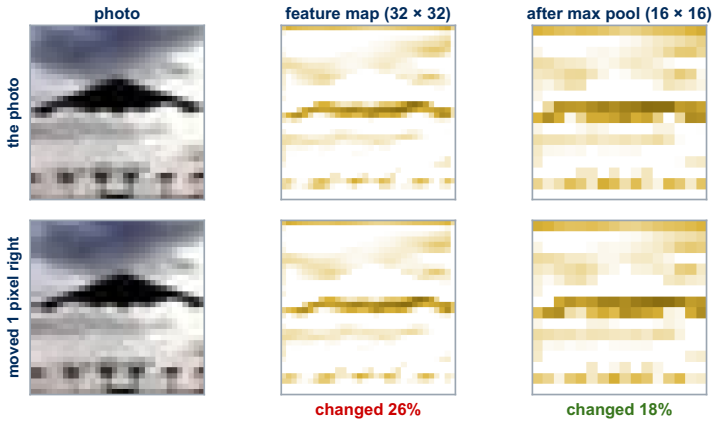
That makes a filter a strange sort of neuron. It's a weighted sum of `27` inputs plus a bias like the neurons of the last two chapters, but it's used at every position with the same weights. Compare a convolutional layer of `32` filters (with padding `1`) against a fully connected layer with the same number of outputs, `32` maps of `32` × `32`, or `32,768`:

layer	weights	biases	total
convolutional, 32 filters of 3 × 3 × 3	864	32	896
fully connected, 3,072 → 32,768	100,663,296	32,768	100,696,064

Sharing weights across positions like this is called **weight sharing**, and here it means about `112,000` times fewer weights. It also means a filter that has learned to find an edge will **find it wherever it is**, because the same filter is tried everywhere.

Pooling

Finding an edge wherever it is still leaves the feature map saying where it was. Move the photo one pixel right and its maps move one pixel right too, so the next layer sees its numbers in different places. **Pooling**, or downsampling, summarizes small neighborhoods of a feature map so that a filter can match even if what we're looking for isn't perfectly aligned. The most common version keeps the largest number in each 2 × 2 window, halving the map both ways.



Pictured above is one of the filters from the CNN we'll train at the end of this chapter, on the photo and on the photo moved one pixel right. Its feature map changed by 26% (adding up how far every number moved), its pooled map by 18%. Over all 32 of the first layer's filters it's 37% against 25%, so **pooling softens a small shift but doesn't remove it**, since a feature that crosses into the next window still moves.

Max and Average Pooling

Keeping the largest number in each window is called **max pooling**. The other common choice, average pooling, keeps the average instead. Take the window of our horizontal edge map (with padding `1`, negatives set to `0` by a ReLU) that holds the `712`, with `685`, `80` and `217` beside and below it:

[2.2] max pooling

$$\max(712, 685, 80, 217) = 712$$

[2.3] average pooling

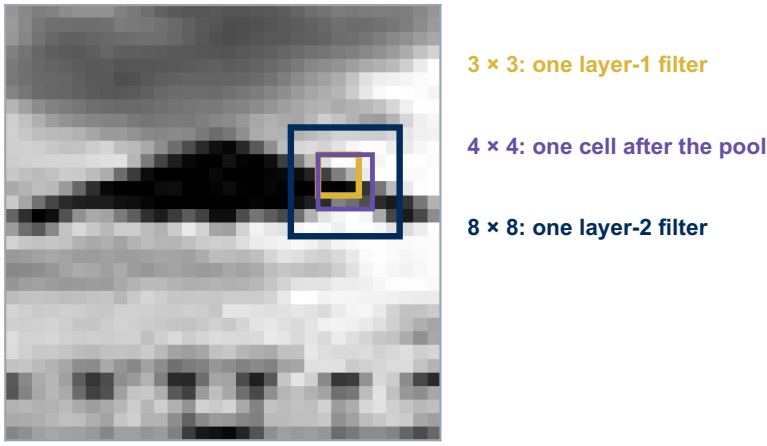
$$\tfrac{1}{4}(712 + 685 + 80 + 217) = 423.5$$

The `712` says "there's an edge in here", which is what we wanted to know. The `423.5` says the same thing more quietly, watered down by the patches that didn't match as well. Networks that use max pooling tend to perform better than those using average pooling, so **when people speak of pooling they usually mean max pooling**.

Either way, pooling has no weights, so there's nothing to learn. It also cuts every map to a quarter of its size, which makes every later layer about four times cheaper.

Receptive Fields

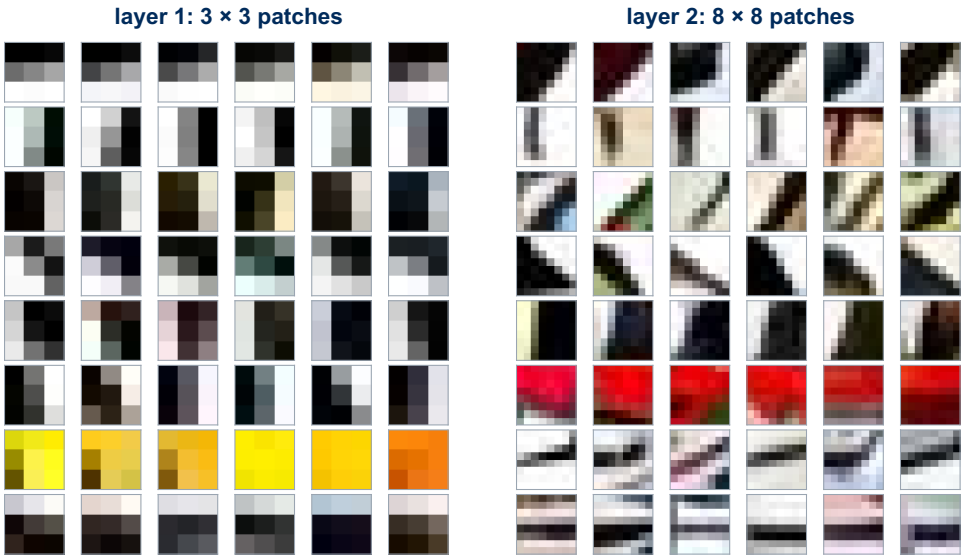
A single 3 × 3 filter only ever sees nine pixels, enough for an edge but not for a wing, let alone a plane. So we stack them, filters on the photo, then pooling, then a second set of filters on the pooled maps. Each cell of a pooled map summarizes a 4 × 4 patch of the photo, so a 3 × 3 filter on the pooled map sees an 8 × 8 patch.



The patch of the image that one output can see is called its **receptive field**, and it grows with every layer. Another round of filters and pooling would take it to 18 × 18, more than half the photo.

Hierarchical Filters

The intuition behind stacking is that earlier filters learn low-level features like edges, which get passed into later filters that learn to recognize more complex things (a corner, a stripe, eventually part of a wing) from the lower level filters' output. Pictured below, for the CNN we'll train later, are the patches of unseen photos that excite some of its units most.



Layer 1's units, seeing 3 × 3 patches, respond to edges at different angles and to single colors. Layer 2's, seeing 8 × 8 patches, respond to **corners, diagonal strokes, stripes and patches of red**. Nobody designed either set.

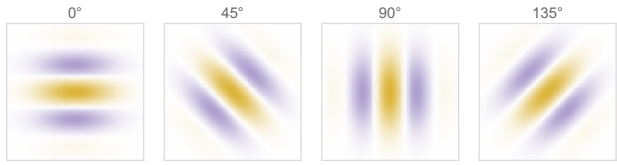
Hierarchies in the Brain

The idea of a hierarchy comes from biology. In `1959` Lettvin, Maturana, McCulloch and Pitts reported cells in a frog's eye that respond to small dark moving spots (bug detectors, more or less), and the same year Hubel and Wiesel found cells in a cat's visual cortex that respond to an edge at one angle in one small part of the visual field. Something like the top of the hierarchy turned up in `2005`, when Quiroga and colleagues recorded single neurons in epilepsy patients and found one that responded to photos of Jennifer Aniston and very little else.

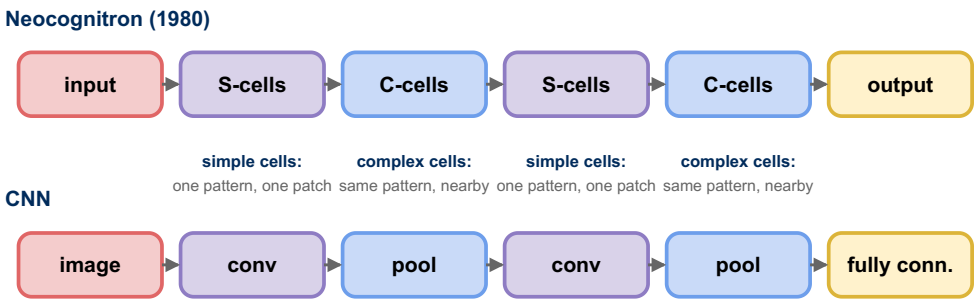
Hubel and Wiesel's cells are the bottom of the hierarchy, each one looking at a small patch, and Quiroga's neuron is somewhere near the top, responding to one person across many different photos of her. A CNN is built the same way, **small patches first, bigger and more specific things later**.

The Neocognitron

Hubel and Wiesel found two kinds of cells. A **simple cell** fires for an edge at one angle in one small patch of what the cat sees, and a **complex cell** fires for the same angle anywhere in a somewhat bigger patch, as if it had pooled a group of simple cells. (They shared a Nobel Prize for this in 1981.) The usual model of a simple cell's patch is stripes under a soft round window:



In 1980 Kunihiko Fukushima stacked the two into a network, the **Neocognitron**, with layers of S-cells (filters) alternating with layers of C-cells (pooling):



S for simple, C for complex, after Hubel and Wiesel

It learned its filters without labels. LeCun and colleagues trained the design with backpropagation in 1989, to read zip codes, which is **more or less the CNN of this chapter**.

CNN Architecture

A CNN for classification has two halves. The first is blocks of convolution, ReLU and pooling, which learn a way to map from the photo to a smaller set of useful features (a **learned representation**). The second flattens the last maps into one list and hands it to a feed-forward network like the ones in the last two chapters, which does the classifying. For CIFAR-10 two blocks are enough:

layer	output shape	weights
photo	3 × 32 × 32	
conv, 32 filters of 3 × 3 × 3, padding 1, ReLU	32 × 32 × 32	896
max pool 2 × 2	32 × 16 × 16	0
conv, 64 filters of 3 × 3 × 32, padding 1, ReLU	64 × 16 × 16	18,496
max pool 2 × 2	64 × 8 × 8	0
flatten	4,096	0
fully connected, ReLU	512	2,097,664
fully connected, ReLU	256	131,328
fully connected, softmax	10	2,570
total		**2,250,954**

Notice that the second layer's filters are $3 \times 3 \times 32$, a 3×3 patch of all 32 maps at once, so each has 288 weights and a bias.

CNNs in PyTorch

In PyTorch our network is a few lines:

```python
import torch
from torch import nn

cnn = nn.Sequential(
    # the convolutional block
    nn.Conv2d(3, 32, kernel_size=3, padding=1),
    nn.ReLU(),
    nn.MaxPool2d(2),
    nn.Conv2d(32, 64, kernel_size=3, padding=1),
    nn.ReLU(),
    nn.MaxPool2d(2),
    # the fully connected block
    nn.Flatten(),
    nn.Linear(64 * 8 * 8, 512), nn.ReLU(),
    nn.Linear(512, 256), nn.ReLU(),
    nn.Linear(256, 10),
)
x = torch.zeros(5, 3, 32, 32)  # 5 blank photos
print(cnn(x).shape)
print(sum(p.numel() for p in cnn.parameters()))
```

OUTPUT

```
torch.Size([5, 10])
2250954
```

That's more weights than our feed-forward network (`1,707,274`), so the CNN isn't small. `2,231,562` of its weights (`99%`) are in the fully connected layers at the end, and its two convolutional layers have only `19,392` between them. The filters are cheap, it's **the flattened maps that are expensive**.

Convolution as Matrix Multiplication

Our convolution ran two nested loops, and a real layer adds loops over filters, channels and photos. That's really really really slow (your author once spent about 9 hours training a CNN that way). The trick is to cut out every patch first, one row per position, and make each filter a column:

```
import numpy as np
import torch
import torch.nn.functional as F
photo = torch.tensor(np.load('cifar-plane.npy'))
grey = photo.float().mean(dim=2).round()
H = torch.tensor([[1., 1, 1], [0, 0, 0], [-1, -1, -1]])
V = H.T  # the vertical-edge filter
patches = F.unfold(grey[None, None], 3)[0].T  # (900, 9)
filters = torch.stack([H.reshape(-1), V.reshape(-1)],
                      dim=1)
maps = patches @ filters  # every position, both filters
print(maps[:, 0].reshape(30, 30)[11, 20:26].long())
print(maps[:, 1].reshape(30, 30)[11, 20:26].long())
```

OUTPUT

```
tensor([369, 528, 658, 712, 685, 558])
tensor([-144, -150, -163, -150, -148, -180])
```

The 900 × 9 patch matrix times the 9 × 2 filter matrix gives both maps, the first matching our loops. This is called **im2col** ("image to columns"), and PyTorch's `F.unfold` does the cutting out. With it the same CNN trained in about 5 minutes, since PyTorch (and even more so a GPU) does one big matrix multiplication very fast.

GEMM

Matrix multiplication is one of the most common operations in scientific computing, so it has its own name, **GEMM** (GEneral Matrix Multiply), after the routine in the BLAS libraries that nearly every numerical program calls. People have spent decades tuning it for every processor. A multiplication **uses each number many times**, so a good library keeps numbers in the processor's fastest memory while it reuses them, and does many multiply-adds at once.

Our loops get none of that. Running all 32 filters of our CNN's first layer over the airplane photo took about `0.35` seconds with the loops in Python, and about `0.06` milliseconds as one GEMM, roughly `6,000` times faster (on the computer this chapter was written on). GPUs widen the gap further (recent NVIDIA cards have **tensor cores** that do almost nothing but small matrix multiplications).

A linear layer is already a GEMM, and im2col turns a convolution into one too, so **nearly all the work of training a CNN ends up as GEMMs**. Unfortunately im2col pays for it in memory. Every pixel lands in up to `9` rows of the patch matrix, so for the airplane's three colors it holds `24,300` numbers where the photo had `3,072`. Libraries like NVIDIA's cuDNN get around this with an "implicit GEMM", which cuts out each patch only when the multiplication needs it.

Training a CNN

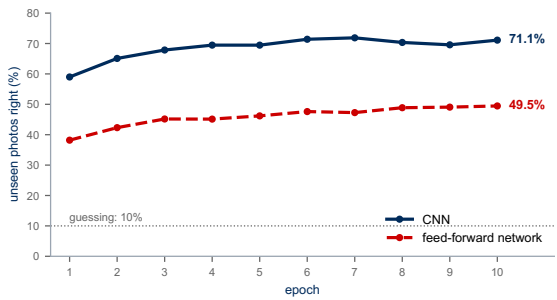
Training a CNN works like training any other network, softmax, cross-entropy and backpropagation. The one new piece is a filter's gradient. Each filter weight $K(m, n)$ was used at every position, so its gradient adds up what it contributed at all of them:

[2.4]

$$\frac{\partial \mathcal{L}}{\partial K(m,n)} = \sum_i \sum_j \frac{\partial \mathcal{L}}{\partial S(i,j)} I(i+m, j+n)$$

summed over every position (i, j) the filter visited

(itself a cross-correlation, of the image with the feature map's gradient). We trained both networks the same way, `10` epochs of Adam in batches of `64` on the `50,000` training photos:



The CNN gets **71.1% of the unseen photos right**, against `49.5%` for the feed-forward network, and after one epoch it's already past anything the feed-forward network reaches. Unfortunately it also gets `97.8%` of its training photos right, and its unseen accuracy peaked at `71.9%` in epoch `7`, so it's **starting to memorize**.

The Layout Matters

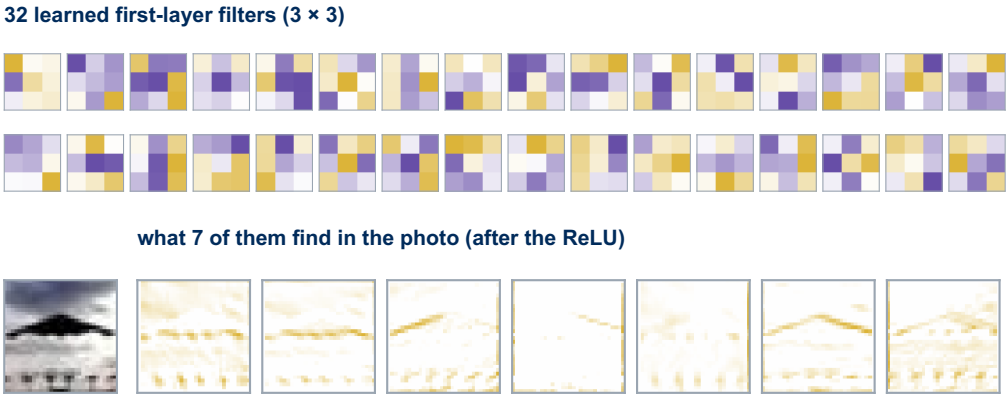
So what does the CNN know that the feed-forward network doesn't? Shuffle the pixels of every photo into the same fixed order, like we did for the feed-forward network, and train the CNN again:

unseen photos right	feed-forward	CNN
photos as they are	49.5%	71.1%
pixels shuffled	48.8%	50.4%
moved one pixel right	46.3%	54.1%

With its pixels shuffled the CNN does about as well as the feed-forward network. **The CNN's whole advantage is where each pixel is**. Its filters look at neighboring pixels, and once the neighbors are scattered there's nothing for them to find.

Unfortunately the layout cuts both ways. Move every unseen photo one pixel right (with a black stripe down the left) and the CNN drops to `54.1%`. The pools only forgive a shift that stays inside a window, the fully connected layers at the end still care where things are, and the network **has never seen a moved photo**. The usual fix is to show it some. Training on extra copies of each photo, moved, flipped or rotated a little, is called **data augmentation**, and almost every CNN trained on real photos uses it.

What Our CNN Learned



Pictured above are our CNN's 32 first-layer filters (each 3 × 3 × 3, drawn with its three colors added up) and what seven of them make of the airplane. Nobody told them to look for edges, but most of them **find an edge at some angle**, along the top or underside of the plane or the building's columns.

CNNs and the Visual Cortex

Our filters are only 3×3, too small to look like much. Bigger networks on real photos are more interesting. AlexNet, trained on 1.2 million photos in 2012, learned 96 first-layer filters, and they came out looking a lot like V1's simple cells: **edges at every angle and thickness**, plus a few blobs of color. Nobody told the network that's what eyes do.

The resemblance carries on up the hierarchy. Yamins et al. (2014) trained CNNs to recognize objects and compared them with recordings from monkeys looking at the same photos. The networks' middle layers predicted neurons in V4, their top layers predicted neurons in IT, and the networks that were better at the task matched the brain better.

So the CNN went full circle. It was built to copy the visual cortex, then trained only to recognize things, and **ended up working a lot like the thing it copied**.

Computer Vision

Prior to `2020` and the paper "An Image is Worth 16x16 Words" (which cut images into `16 × 16` patches and fed them to a transformer), basically all of **computer vision** was done with CNNs. They're still widely used, and they do much more than classify, from finding and outlining objects in a photo to estimating depth and sharpening blurry images.

Unfortunately, a CNN expects its input to be a grid of a fixed size. A sentence isn't one. Its words come one after another, there can be any number of them, and what each one means depends on the words that came before it.