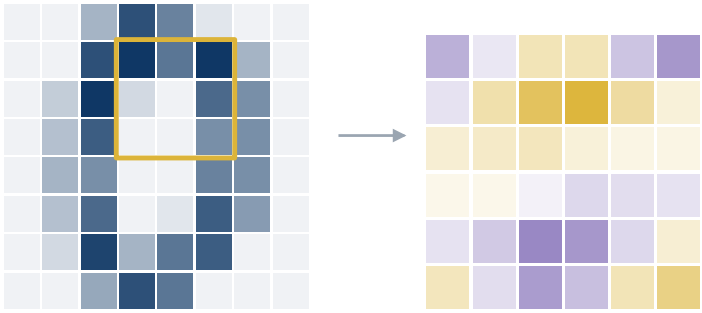


UNIT 03 · CHAPTER 2

Convolutional Neural Networks

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

Dataset - MNIST

police archive (EMNIST): labelled, 28 × 28



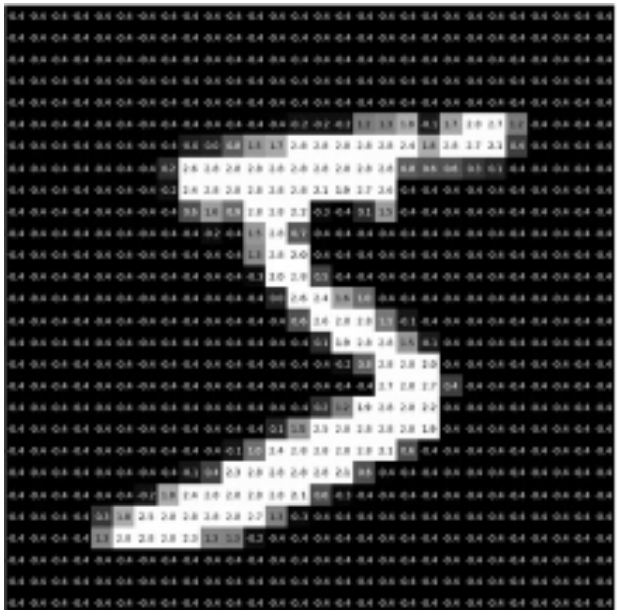


Professor Dingler's note: no labels





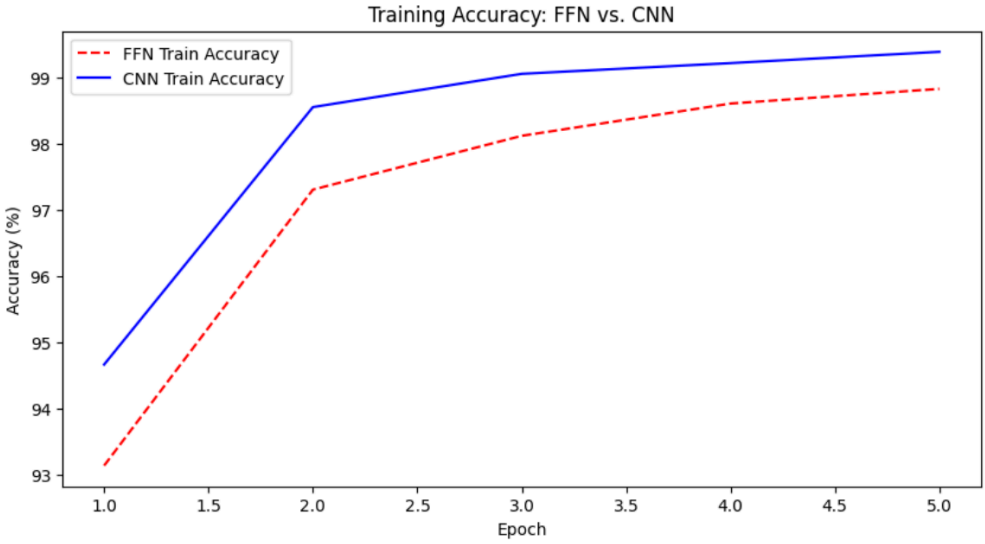
Reviewing Neural Networks



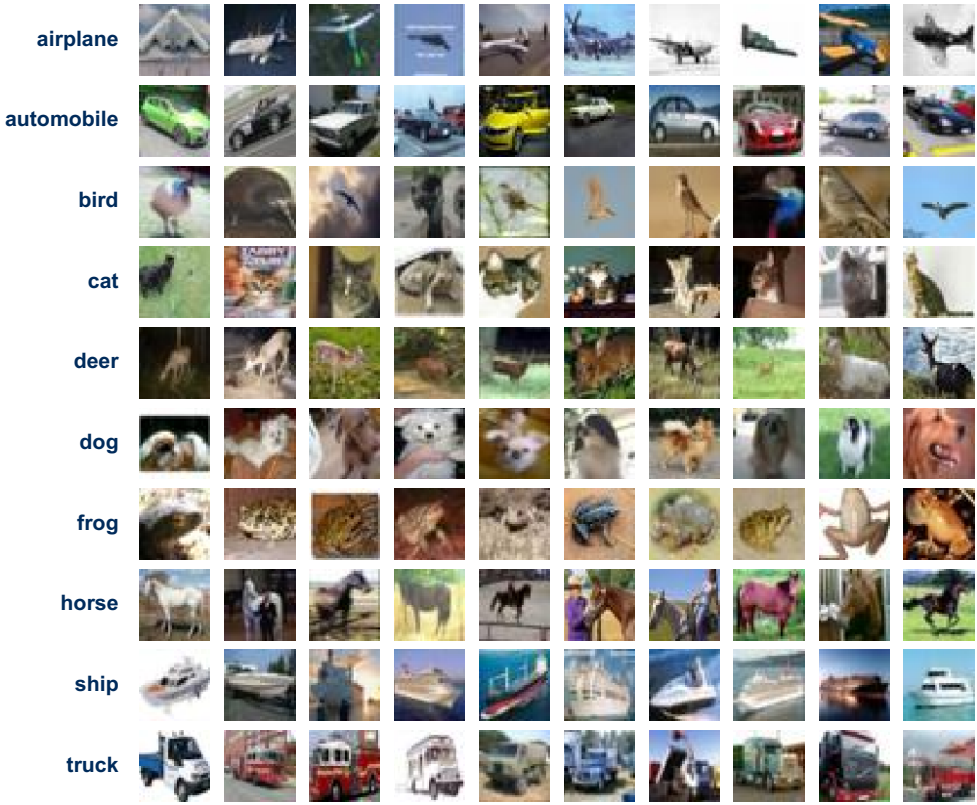
digit	0	1	2	3	4	5	6	7	8	9
$\hat{y}$	0.11	0.14	0.13	0.21	0.10	0.09	0.08	0.06	0.05	0.03
y	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00

Why do we care about CNNs?

We get **99%** on **MNIST** without using anything fancy, why do we care?



CIFAR-10



CIFAR-10 is `60,000` tiny color photos, `32` × `32` pixels, of `10` kinds of thing. We train on `50,000` and keep `10,000` aside as _____. A lot **harder than handwritten digits**.

A Feed-Forward Network on Photos

To hand a photo to a feed-forward network we _____ it: `32` × `32` pixels with `3` colors each is _____ inputs.

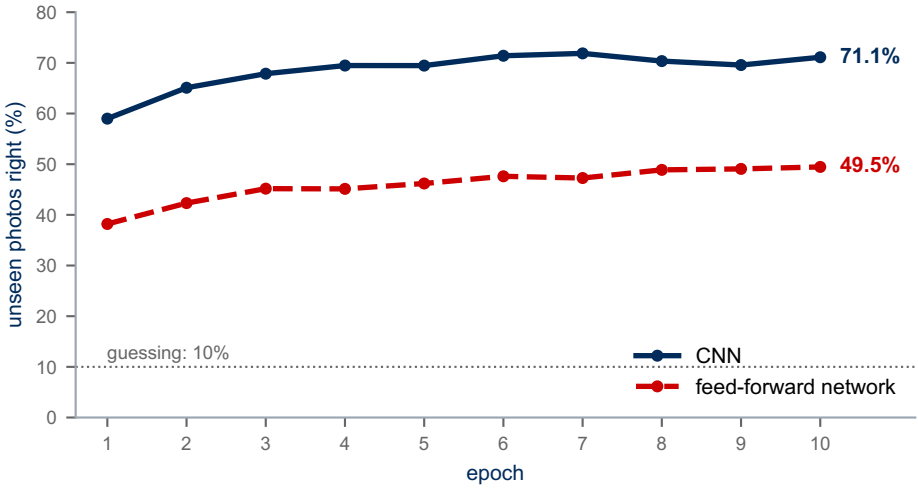
```python
import torch
from torch import nn

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
```

CNN vs. FFN on CIFAR-10



Same photos, same training: the **FFN** gets **49.5%** of unseen photos right, a **CNN** gets **71.1%**.

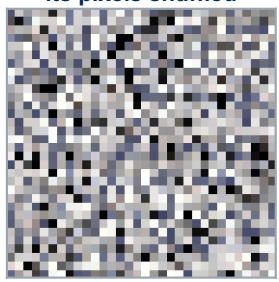
Shuffling the Pixels

Shuffle every photo's pixels into the same random order (nothing is lost, it's just rearranged) and train both networks again:

a photo



its pixels shuffled

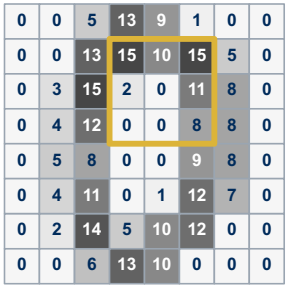


unseen photos right	feed-forward	CNN
photos as they are	**49.5%**	**71.1%**
pixels shuffled	**48.8%**	**50.4%**

The **FFN doesn't use the layout**. The **CNN's** whole advantage is making use of the overall structure of the image

Filters

A photo is 3,072 numbers, too many to work by hand, so we'll build our first one on an 8 × 8 handwritten zero (pixels from 0 to 16). We want a **neuron** that looks for _____ in an image





Lay a 3 × 3 grid of weights over a 3 × 3 patch, multiply each pixel by the weight on top of it, and _____:

15 + 10 + 15 − 8 = 32.

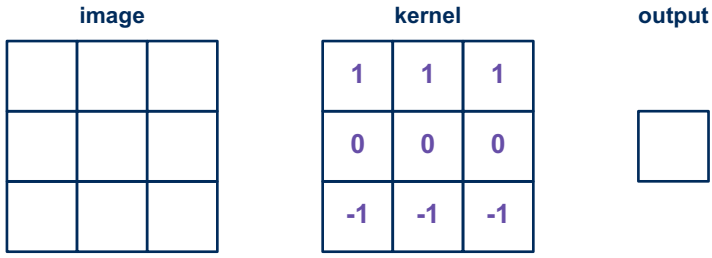
A small matrix of weights **encoding some pattern** we're looking for in an image is called a **filter** or a _____.

A Horizontal Edge Filter

`1`s along the top, `0`s through the middle, `-1`s along the bottom: it adds up the top row and

_____.

- ink on top, blank below: _____
- a blank patch: _____
- ink only at the bottom: _____



Exercise: with pixels from `0` to `16`, what `3` × `3` patch gives this filter its largest output, and what is it?

Convolution

Slide the filter across the image _____, doing the same multiply and add at every stop. A 3 × 3 filter fits in 6 places across an 8 pixel row:

```python
import torch
from sklearn.datasets import load_digits

zero = torch.tensor(load_digits().images[0],
                    dtype=torch.float32)
K = torch.tensor([[ 1.,  1.,  1.],
                  [ 0.,  0.,  0.],
                  [-1., -1., -1.]])

out = torch.zeros(6, 6)
for i in range(6):
    for j in range(6):
        patch = zero[i:i+3, j:j+3]
        out[i, j] = (patch * K).sum()
print(out.long())
```

OUTPUT

```
tensor([[-13,  -2,  10,  10,  -9, -18],
        [ -3,  12,  26,  32,  14,   4],
        [  5,   7,   9,   4,   2,   2],
        [  1,   1,   0,  -5,  -4,  -3],
        [ -3,  -8, -21, -18,  -5,   5],
        [  9,  -4, -17, -10,  10,  19]])
```

Cross-Correlation

CNNs are named after **convolution**, which combines two functions to create a third by _____ before sliding it:

Without the flip it's called **cross-correlation**, and that's what we just did:

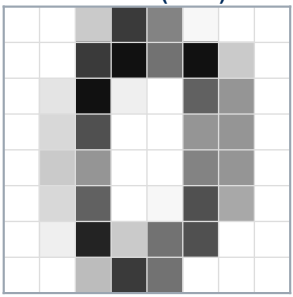
cross-correlation: what CNNs actually compute

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

Nearly every library computes cross-correlation and **calls it convolution**. The kernel is learned anyway, so a network that needed it flipped would _____.

Feature Maps

the zero (8 × 8)



horizontal-edge filter

-13	-2	10	10	-9	-18
-3	12	26	32	14	4
5	7	9	4	2	2
1	1	0	-5	-4	-3
-3	-8	-21	-18	-5	5
9	-4	-17	-10	10	19

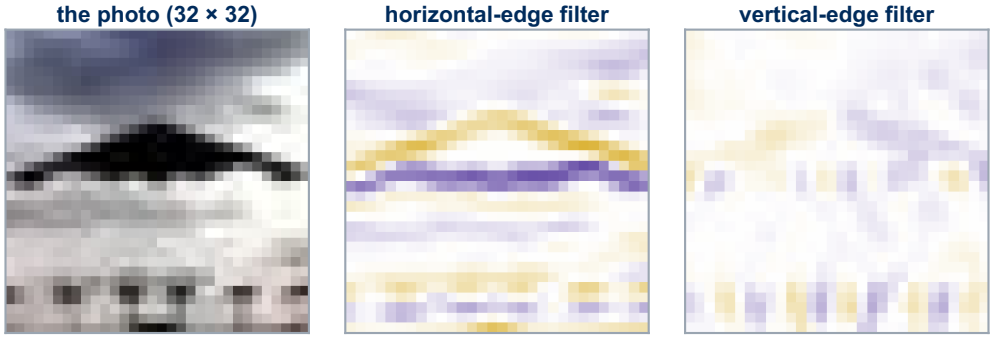
vertical-edge filter

-33	-27	14	3	6	27
-40	-10	30	-17	-11	34
-35	10	35	-26	-24	28
-31	13	30	-29	-22	29
-33	6	22	-28	-4	33
-31	-12	10	-6	14	24

The grid a filter makes is a **feature map**, one number per position saying _____.

Turned on its side, the filter picks out the zero's sides instead (`-40` and `35`). **Different filters find different patterns**, so a layer applies _____ and keeps every map.

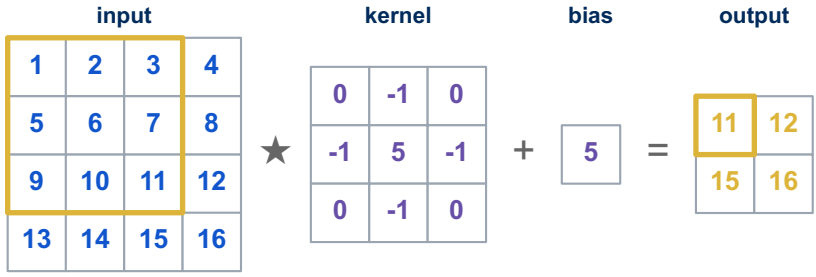
Edges in a Photo



The same two filters on a CIFAR photo (turned grey by averaging its three colors): the horizontal-edge filter traces _____, the vertical-edge filter finds _____.

A real CNN's first filters are `3` × `3` × `3`, so they can look for _____ too.

Output Size



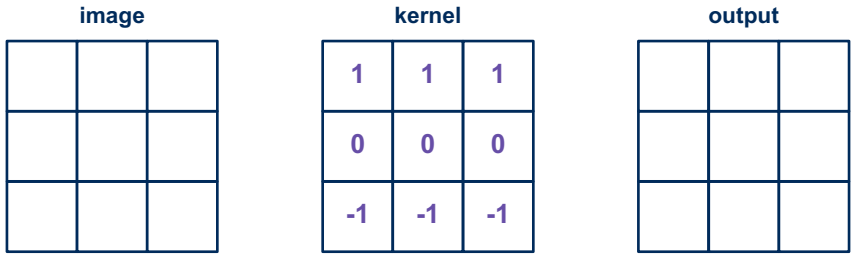
A $k \times k$ filter on an $n \times n$ image fits in _____ places across, so a 4×4 image gives a 2×2 output.

Each output also gets a **bias**, the same job it did in our neurons: $5 \cdot 6 - 2 - 5 - 7 - 10 + 5 = 11$.

Padding

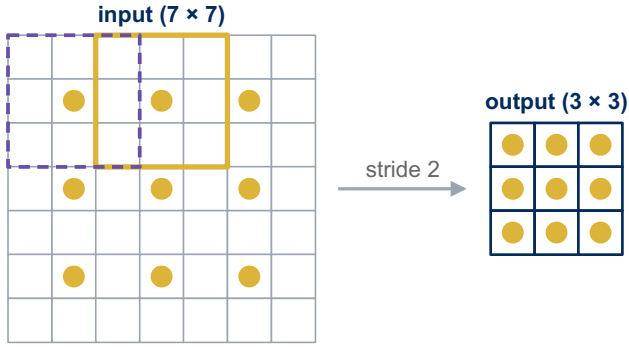
Shrinking is a nuisance: two 3×3 layers take our digit down to 4×4, and _____ get looked at less.

Padding is _____. One ring makes our zero 10×10, the filter fits in 8 places, and the map is _____.



Exercise: how would we get a 3×3 output from a 3×3 image and a 3×3 filter?

Stride



The **stride** is

_____.

Stride 2 over a 7 × 7 input makes 3 stops across, so the output is 3 × 3.

A cheap way to shrink a map, at the price of **skipping positions where a pattern might have matched**.

Output Size, in General

An $n \times n$ input, a $k \times k$ filter, padding p and stride s give an output of size

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

- our zero, padding 1, stride 1: $(8 + 2 - 3)/1 + 1 =$ _____
- stride 2: $(8 + 2 - 3)/2 = 3.5$, rounded down, plus $1 =$ _____
- an MNIST digit, padding 1, stride 1: $(28 + 2 - 3)/1 + 1 =$ _____

Learned Filters

Instead of hard-coding what the filters look for, the numbers in each filter are **weights**,

like every other weight.

A filter is a neuron with a few weights and a bias, **used at every position**. The first layer of our CIFAR CNN against a fully connected layer with the same `32,768` outputs (`32` maps of `32` × `32`):

layer	weights
convolutional, 32 filters of 3 × 3 × 3	896
fully connected, 3,072 → 32,768	100,696,064

This is called **weight sharing**.

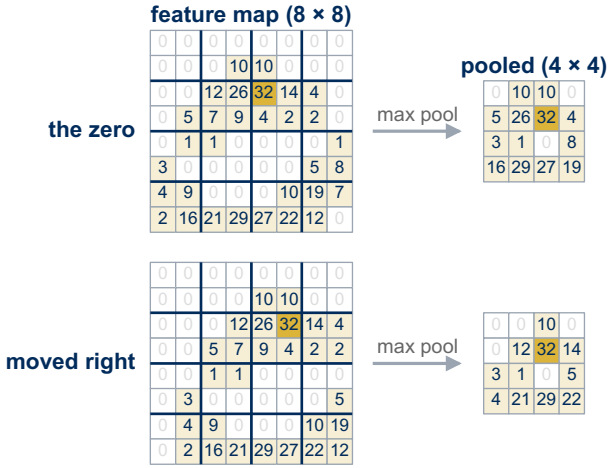
Pooling

Pooling, or _____, is a method through which we can

_____ to **allow filters to match** even if **what we're looking for isn't perfectly aligned**

The most common version keeps _____ in each 2×2 window, **halving the map both ways**.

Pooling the Zero

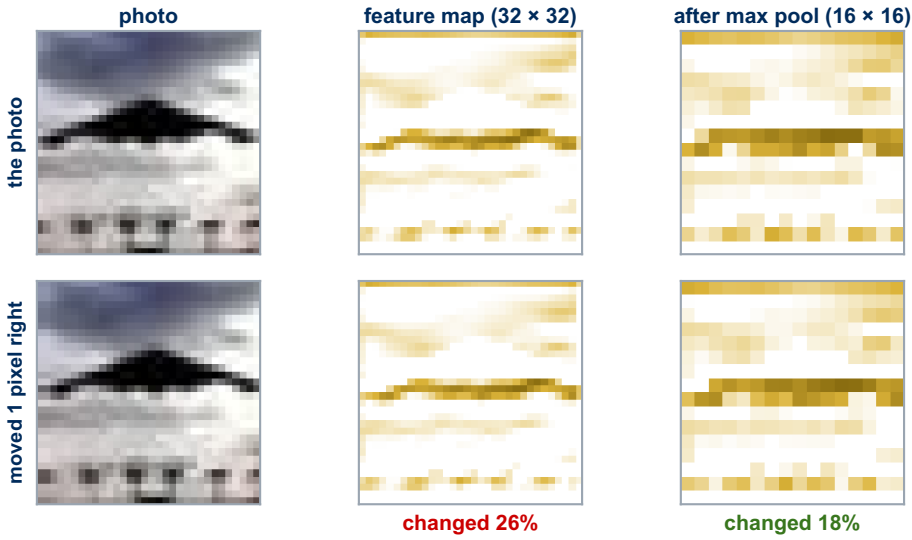


The top edge's 32 moved in the map but

_____.

The bottom edge's 29 crossed into the next window, so pooling only forgives a shift **that stays inside a window**.

Pooling a Moved Photo



Over all 32 first-layer filters, a one-pixel move changes the maps by 37% and the pooled maps by 25%. Pooling **softens a small shift**, it doesn't _____.

Max and Average Pooling

The two common types are **max pooling** and **average pooling**. The window of the zero's map that holds the `32`:

[2.2] max pooling

$$\max(32, 14, 4, 2) = 32$$

[2.3] average pooling

$$\frac{1}{4}(32 + 14 + 4 + 2) = 13$$

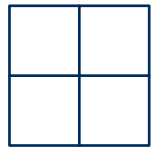
Networks that use **max pooling** _____ than those using **average pooling**, so **when people speak of pooling they usually mean max pooling**

Pooling has _____, and cuts every map to _____ of its size.

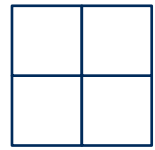
Pooling Layers (exercise)

Compute what the average and max pooling output would be.

3	2	−3	2
1	6	20	5
4	−13	2	6
−2	3	9	3



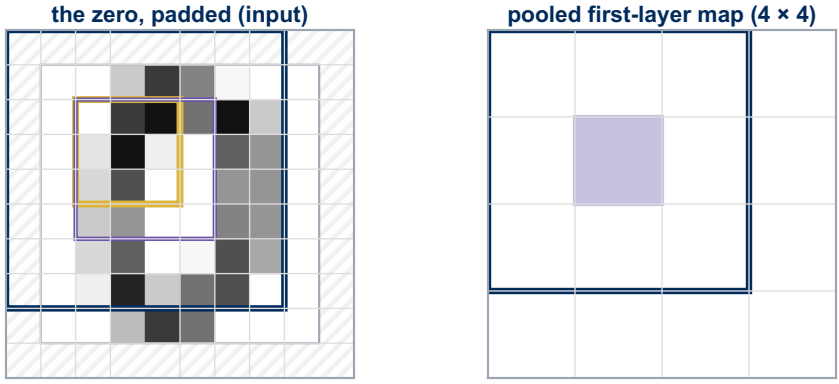
Average



Max

Exercise: compute the 2×2 average pooling and max pooling outputs (window 2×2, stride 2).

Receptive Fields



One 3 × 3 filter sees 9 pixels, enough for _____ but not for a zero. After a 2 × 2 pool each cell summarizes a 4 × 4 patch, so a 3 × 3 filter on top sees _____, the whole digit.

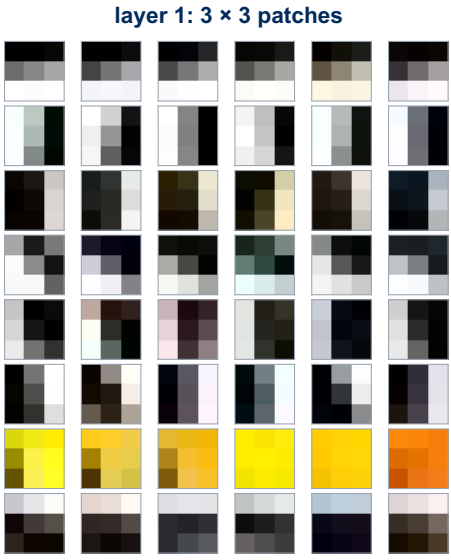
The patch one output can see is its **receptive field**, and it _____.

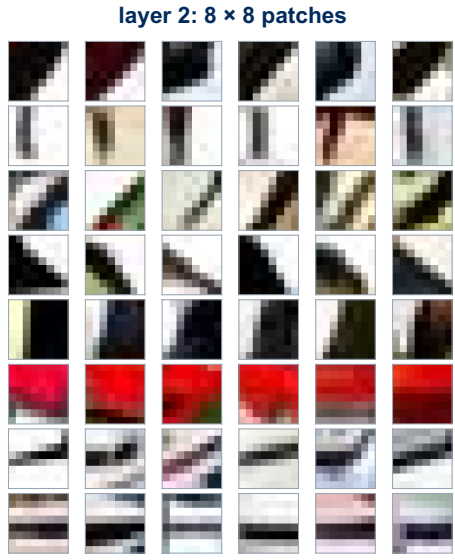
Hierarchical Filters

The **intuition** is that earlier filters learn

_____

like edges which then **get passed into** _____ filters that **learn to recognize more complex things** like corners, stripes and parts of objects from the lower level filters output





What excites our CIFAR CNN most: a layer-1 unit sees 3 × 3 pixels and finds _____, a layer-2 unit sees 8 × 8 and finds

_____.

Where the Idea Came From

In 1959 Lettvin, Maturana, McCulloch and Pitts found cells in a frog's eye that fire for

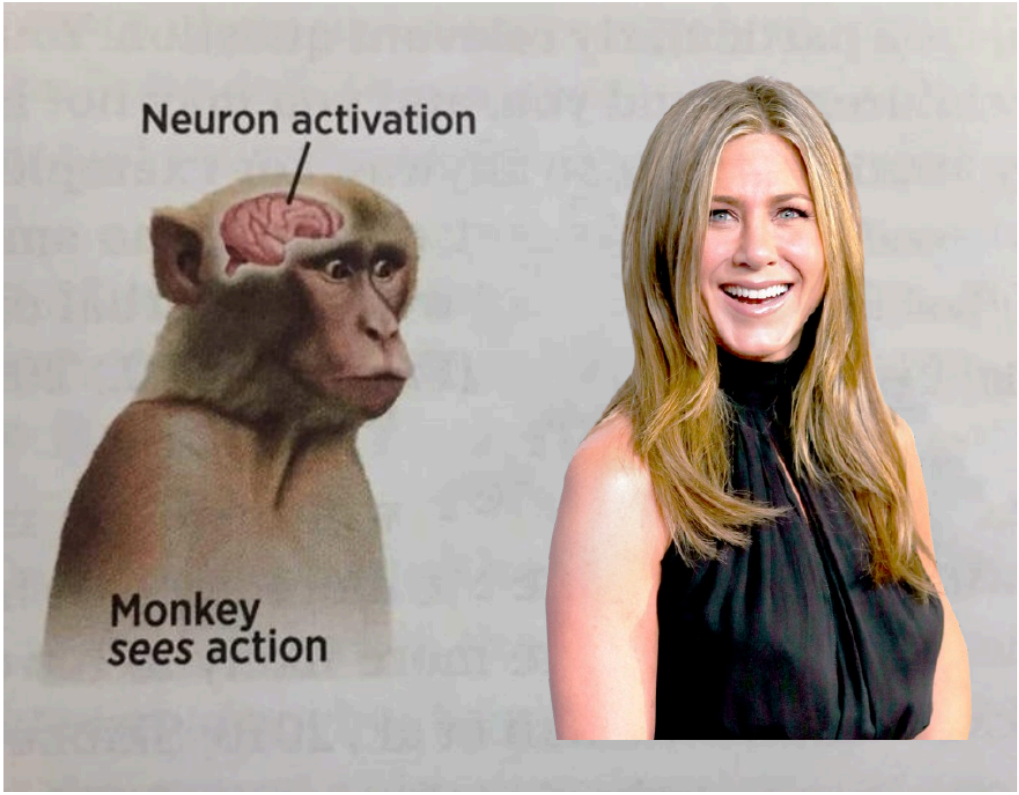
_____

(bug detectors, more or less).

It was *somewhat* replicated in humans: Quiroga et al. (2005) found a single neuron that fired for

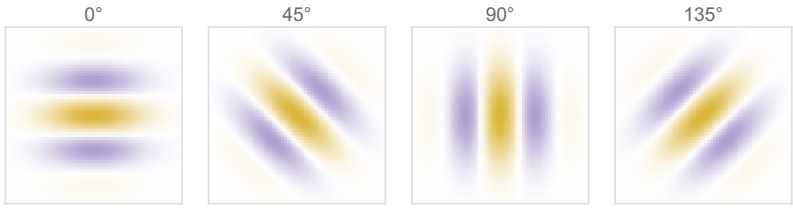
_____

and very little else.



Simple and Complex Cells

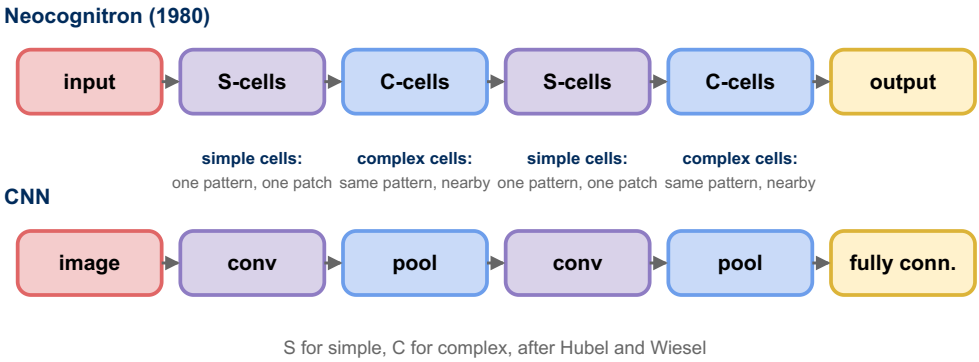
In **1959** Hubel and Wiesel recorded single neurons in a cat's visual cortex while showing it bars of light. Each **simple cell** fired for **an edge or bar at one _____, in one small patch** of what the cat saw (its **receptive field**).



Complex cells fired for the same angle **anywhere in a larger patch** (a bit like pooling). Hubel and Wiesel shared the **1981** Nobel Prize for this.

The Neocognitron (1980)

Kunihiko Fukushima took Hubel and Wiesel's two kinds of cells and **stacked them into a network**: layers of **S-cells** (one pattern, one small patch) alternating with layers of **C-cells** (the same pattern, anywhere nearby).



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

Swap S for _____ and C for _____ and it's a CNN. It learned its filters without labels. LeCun et al. (1989) trained one with **backprop** instead, to read handwritten zip codes.

CNN Dimensions (exercise)

Exercise: starting from a $3 \times 32 \times 32$ photo, give the shape after every layer: conv 3×3 with 32 filters and padding 1, ReLU, max pool 2×2, conv 3×3 with 64 filters and padding 1, ReLU, max pool, flatten, then fully connected to 512, 256 and 10. How many weights does each layer have?

CNN Architecture

Blocks of _____ learn a **learned representation**, then a _____ classifies it. Our CIFAR CNN:

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**

CNNs in PyTorch

The convolutional block: two rounds of _____.

```
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),
```

Then the fully connected block, which _____:

```
    # 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
```

Where the Weights Are

network	weights	unseen photos right
feed-forward	1,707,274	49.5%
CNN	2,250,954	71.1%

The CNN has _____ weights than the feed-forward network, and `2,231,562` of them (`99%`) are in its fully connected layers. Its two convolutional layers have only `19,392`.

So it isn't winning by size, it's winning by _____.

Training a CNN

Softmax, cross-entropy and backprop, same as before. The one new piece: each filter weight was used _____, so its gradient _____:

[2.4]

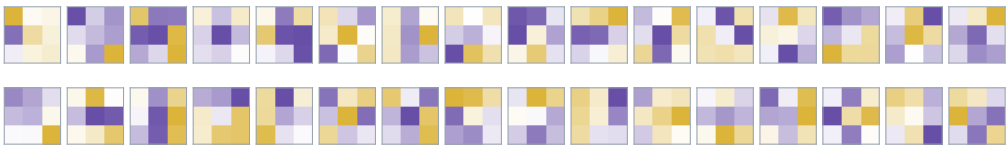
summed over every position (i, j) the filter visited

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

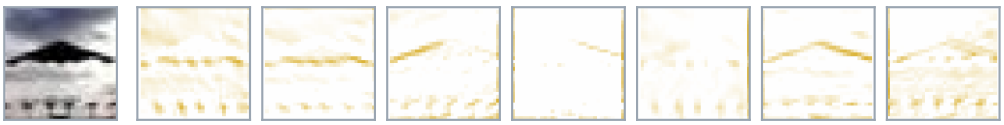
Which is itself a **cross-correlation**, of the image with the feature map's gradient.

What Our CNN Learned

32 learned first-layer filters (3 × 3)



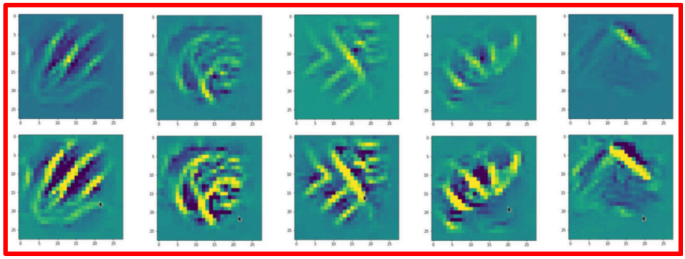
what 7 of them find in the photo (after the ReLU)



Trained on `50,000` photos it gets **71.1%** of unseen photos right. Nobody told its filters what to look for, but most of them _____.

Learned Filters (MNIST)







CNNs and the Visual Cortex

Trained on 1.2 million photos, AlexNet's 96 first-layer filters came out looking a lot like **V1's simple cells**: **edges at every** _____, plus a few blobs of color (Krizhevsky, Sutskever and Hinton, 2012). Nobody designed them.

Further up, a CNN trained only to recognize objects had middle layers that predicted neurons in **V4** and top layers that predicted neurons in **IT**, the brain's object area (Yamins et al., 2014).

Built to copy the visual cortex, then trained to do its job, CNNs **ended up a lot like it**.

Why the CNN Wins

1. Each filter looks at _____, because neighboring pixels belong together.
2. The same filter is used _____, because an edge is an edge wherever it is.
3. Layers build _____.

All three depend on **where each pixel is**, which is exactly what flattening threw away. Shuffle the pixels and the CNN falls from `71.1%` to `50.4%`,

_____.

Efficiency in CNN Implementations (homework)

These **nested loops** are really really really slow! It took me **~9 hours** to run my CNN this way.

If we're particularly clever, we can collapse it all into a **single matrix multiplication!**

(this took me from ~9 hours to ~5 minutes)

Python CNN by Hand - Convolutional Layer

Same three methods as last lecture. Each filter is _____, holding its `c_in * k * k` weights.

```python
import torch
import torch.nn.functional as F

class ConvLayer:
    def __init__(self, c_in, c_out, k):
        self.k = k
        n = c_in * k * k  # weights in one filter
        self.W = torch.randn(c_out, n) * (2 / n)**0.5
        self.b = torch.zeros(c_out)
```

Python CNN by Hand - Convolutional Layer, Forward

The trick is **im2col**: cut every patch out as _____, and the whole layer is _____.

```python
    def forward(self, X):
        N, C, H, W = X.shape
        k, Ho, Wo = self.k, H - self.k + 1, W - self.k + 1
        self.shape = X.shape  # keep for backward
        # im2col: every k x k patch becomes one row
        cols = F.unfold(X, kernel_size=k)  # (N, C*k*k, L)
        self.cols = cols.transpose(1, 2).reshape(
            N * Ho * Wo, C * k * k)
        Z = self.cols @ self.W.T + self.b  # every patch
        Z = Z.reshape(N, Ho, Wo, -1)
        return Z.permute(0, 3, 1, 2)  # (N, C_out, Ho, Wo)
```

Python CNN by Hand - Convolutional Layer, Backward

`F.fold` is `F.unfold` in reverse: it adds every patch's gradient back _____, summing where patches overlap.

```
    def backward(self, dZ):
        N, C, H, W = self.shape
        k, Ho, Wo = self.k, H - self.k + 1, W - self.k + 1
        dZ = dZ.permute(0, 2, 3, 1).reshape(N * Ho * Wo,
                                            -1)
        self.dW = dZ.T @ self.cols  # dL/dW
        self.db = dZ.sum(dim=0)     # dL/db
        dcols = (dZ @ self.W).reshape(N, Ho * Wo, -1)
        # col2im: each patch's gradient back in place
        dX = F.fold(dcols.transpose(1, 2), (H, W),
                    kernel_size=k)
        return dX  # dL/dX, sent back

    def update(self, lr):
        self.W -= lr * self.dW
        self.b -= lr * self.db
```

Python CNN by Hand - Max Pooling, Forward

```python
class MaxPool:
    def forward(self, X):
        N, C, H, W = X.shape
        self.shape = X.shape  # keep for backward
        # each 2 x 2 window in its own last dimension
        win = X.reshape(N, C, H // 2, 2, W // 2, 2)
        win = win.permute(0, 1, 2, 4, 3, 5).reshape(
            N, C, H // 2, W // 2, 4)
        out, self.idx = win.max(dim=-1)  # keep the winner
        return out
```

Python CNN by Hand - Max Pooling, Backward

Only the pixel that won the max _____, every other pixel in its window gets `0`.

```python
    def backward(self, dA):
        N, C, H, W = self.shape
        g = torch.zeros(N, C, H // 2, W // 2, 4)
        # each gradient goes to the pixel that won
        g.scatter_(-1, self.idx.unsqueeze(-1),
                   dA.unsqueeze(-1))
        g = g.reshape(N, C, H // 2, W // 2, 2, 2)
        g = g.permute(0, 1, 2, 4, 3, 5)
        return g.reshape(N, C, H, W)

    def update(self, lr):
        pass  # no weights to learn
```

Python CNN by Hand - Linear Layer

The same `LinearLayer` as last lecture:

```python
class LinearLayer:
    def __init__(self, n_in, n_out):
        self.W = torch.randn(n_out, n_in) * (2/n_in)**0.5
        self.b = torch.zeros(n_out)

    def forward(self, X):
        self.X = X                  # keep for backward
        return X @ self.W.T + self.b

    def backward(self, dY):
        self.dW = dY.T @ self.X     # dL/dW
        self.db = dY.sum(dim=0)     # dL/db
        return dY @ self.W          # dL/dX, sent back

    def update(self, lr):
        self.W -= lr * self.dW
        self.b -= lr * self.db
```

Python CNN by Hand - ReLU and Flatten

```python
class ReLU:
    def forward(self, Z):
        self.Z = Z  # saved
        return Z.clamp(min=0)

    def backward(self, dA):
        return dA * (self.Z > 0).float()  # dL/dZ

    def update(self, lr):
        pass  # no weights to learn
```

```python
class Flatten:
    def forward(self, X):
        self.shape = X.shape  # keep for backward
        return X.reshape(X.shape[0], -1)

    def backward(self, dY):
        return dY.reshape(self.shape)  # un-flatten

    def update(self, lr):
        pass  # no weights to learn
```

Python CNN by Hand - Network

The same `Network` as last lecture, it doesn't care _____.

```python
class Network:
    def __init__(self, layers):
        self.layers = layers

    def forward(self, X):
        for layer in self.layers:
            X = layer.forward(X)
        return X  # logits

    def backward(self, y_hat, y):
        grad = y_hat - y  # softmax + cross-entropy
        for layer in reversed(self.layers):
            grad = layer.backward(grad)

    def update(self, lr):
        for layer in self.layers:
            layer.update(lr)
```

Training a CNN by Hand - The Digits

```
from sklearn.datasets import load_digits
from sklearn.model_selection import train_test_split

d = load_digits()
X_tr, X_te, y_tr, y_te = train_test_split(
    d.images / 16, d.target, test_size=0.25,
    random_state=0)
# (N, 1, 8, 8): one channel per digit
X_tr = torch.tensor(X_tr).float().unsqueeze(1)
X_te = torch.tensor(X_te).float().unsqueeze(1)
Y_tr = torch.eye(10)[torch.tensor(y_tr)]  # one-hot
```

Training a CNN by Hand

```
torch.manual_seed(0)
model = Network([ConvLayer(1, 8, 3), MaxPool(), ReLU(),
                 Flatten(), LinearLayer(8 * 3 * 3, 10)])
for epoch in range(30):
    for i in range(0, len(X_tr), 32):
        logits = model.forward(X_tr[i:i + 32])
        y_hat = torch.softmax(logits, dim=1)
        model.backward(y_hat, Y_tr[i:i + 32])
        model.update(0.01)

pred = model.forward(X_te).argmax(dim=1)
print('unseen digits right:',
      (pred == torch.tensor(y_te)).float().mean())
```

OUTPUT

```
unseen digits right: tensor(0.9778)
```

That's the browser's run. Desktop PyTorch starts from different random weights and gets **98.7%**. A feed-forward network on the same digits gets **97.3%**.

Training CNNs (homework bonus)

We could ask the same four questions I asked at the start of class for a CNN too!

```python
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim

class CNN(nn.Module):
    def __init__(self):
        super(CNN, self).__init__()
        self.conv_layers = nn.Sequential(
            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)
        )
```

```python
        self.fc_layers = nn.Sequential(
            nn.Flatten(),
            nn.Linear(64 * 8 * 8, 512),
            nn.Sigmoid(),
            nn.Linear(512, 256),
            nn.Sigmoid(),
            nn.Linear(256, 10)
        )

    def forward(self, x):
        x = self.conv_layers(x)
        return self.fc_layers(x)
```

Training CNNs: A and B

A. Draw the computation graph for the network.

B. Write the nested equation you'd use to calculate the loss.

Training CNNs: C and D

C. Give the derivative chain for calculating the gradient of the kernels in the first layer.

D. Give the equation you'd use to actually calculate the gradients.

Computer Vision

Prior to 2020 and the paper "An image is worth 16x16 words" basically all of **computer vision** was done using:

CNNs are still widely used today and were partly **an inspiration for transformers**

Vision Tasks

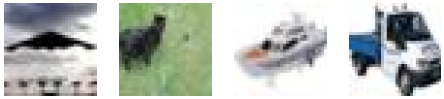
There are a ton of **new tasks** in **computer vision** we haven't seen so far!

These include: Image Classification, Object Detection, Semantic Segmentation, Instance Segmentation, Panoptic Segmentation, Keypoint Detection, Optical Character Recognition (OCR), Image Captioning, Visual Question Answering (VQA), Image-to-Image Translation, Super-Resolution, Image Restoration, 3D Reconstruction, Depth Estimation

Next: Sequences

When our CNN labels a photo, **it doesn't matter what photo came before it or comes after**. Shuffle the photos and every one gets the same label. Shuffle the words of a sentence and _____.

photos in order



airplane cat ship truck

shuffled: same labels



ship airplane truck cat

words in order

cheer cheer for old notre dame

shuffled: a different sentence

notre cheer for cheer dame old