STEVENS
INSTITUTE OF TECHNOLOGY
1870

PEP 559
Machine Learning in Quantum Physics

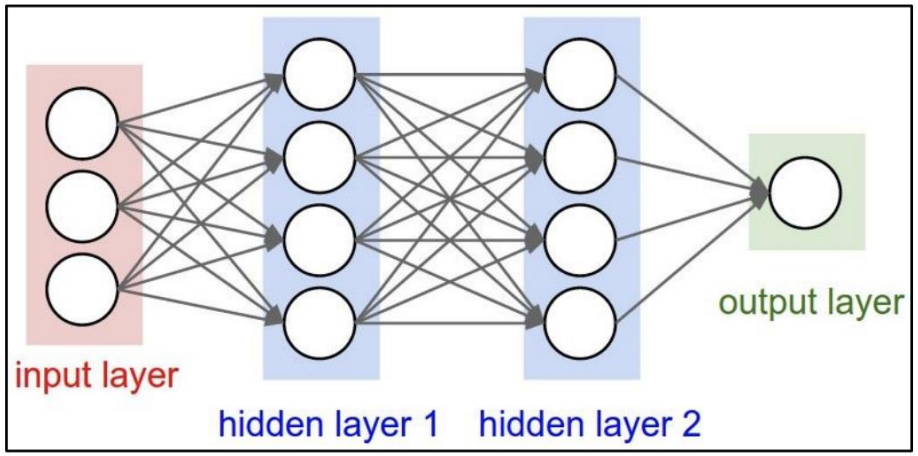
Dr. Chunlei Qu

Spring 2025

Recap: MLP and mini-batch SGD

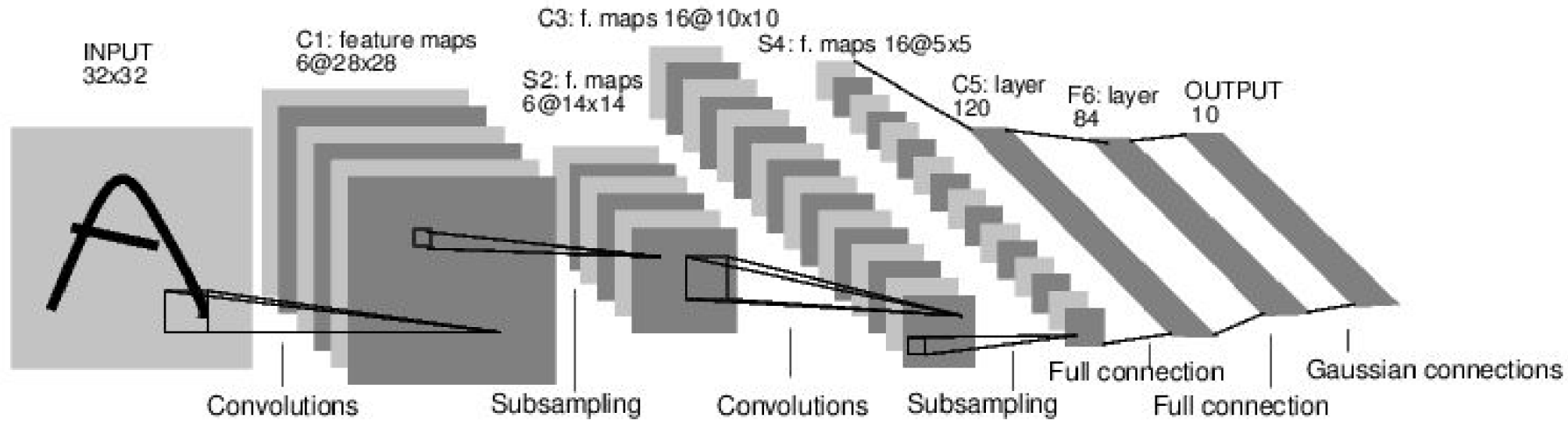
Loop:

1. **Sample** a batch of data
2. **Forward** prop it through the graph, get loss
3. **Backprop** to calculate the gradients
4. **Update** the parameters using the gradient



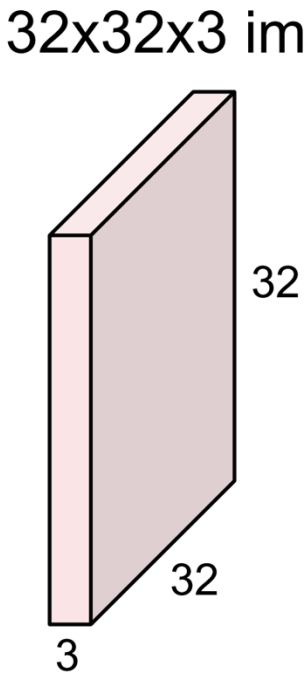
Convolutional Neural Networks (CNNs)

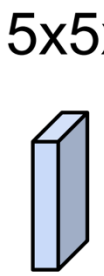
- LeCun Yann et al., 1980
- For classifying handwritten digits from images



Convolution layer: feature extraction layers

- 32=height (pixels)
- 32=width (pixels)
- 3 = number of channels – (RGB color)
- For grayscale image, only one intensity value per pixel, and hence figure size=32x32x1 or just 32x32





Convolve the filter with the image
i.e. “slide over the image spatially,
computing dot products”

Key ideas

CNN performs very well on image-related tasks, because

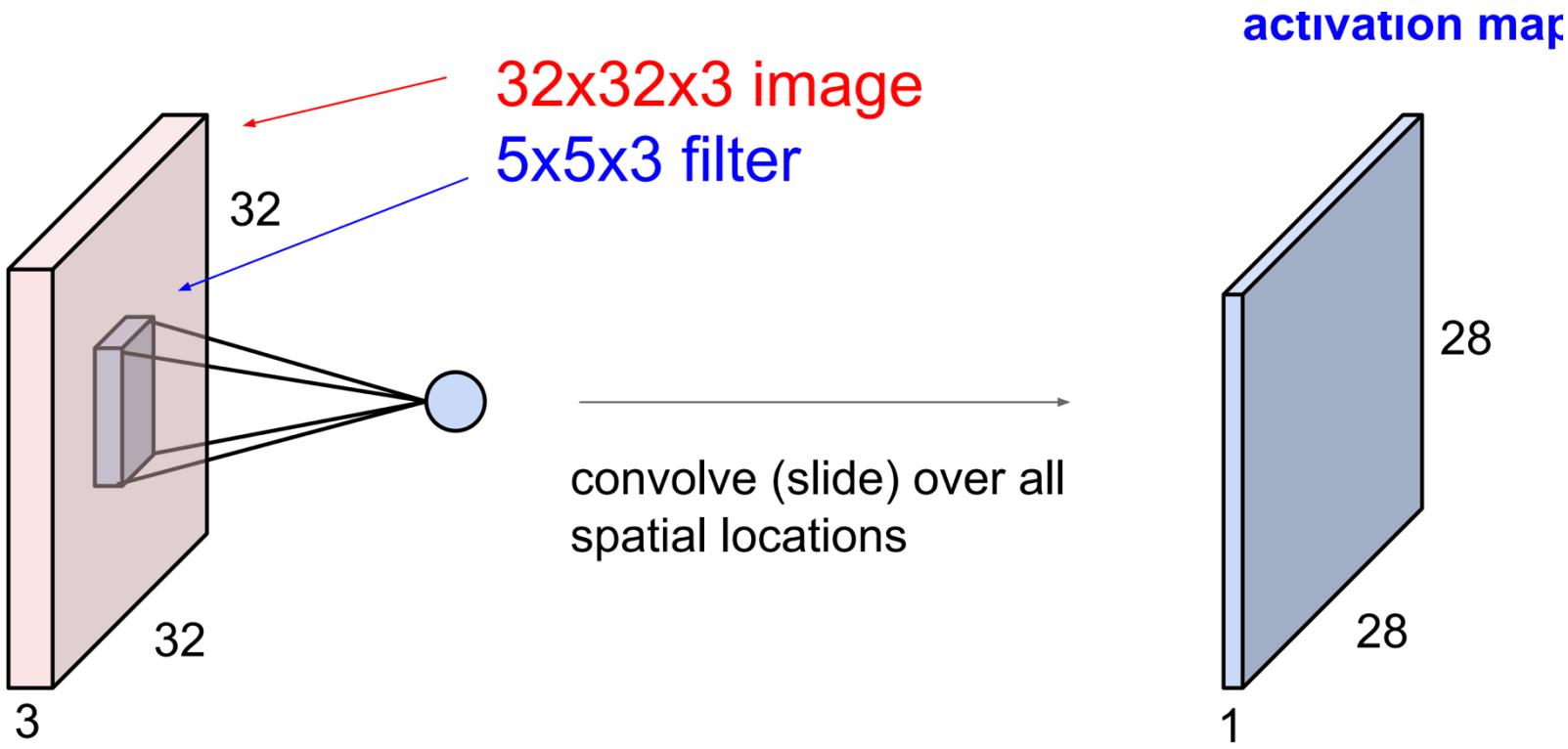
- **Sparse connectivity**: A single element in the feature map is connected to only a small patch of pixels --- local pixels are more correlated --- edges, textures, and features usually appear locally

- **Parameter sharing**: The same filter weights are used for different patches of the input images

Replacing a conventional, fully connected MLP with a convolution layer substantially decreases the number of weights (parameters) in the network

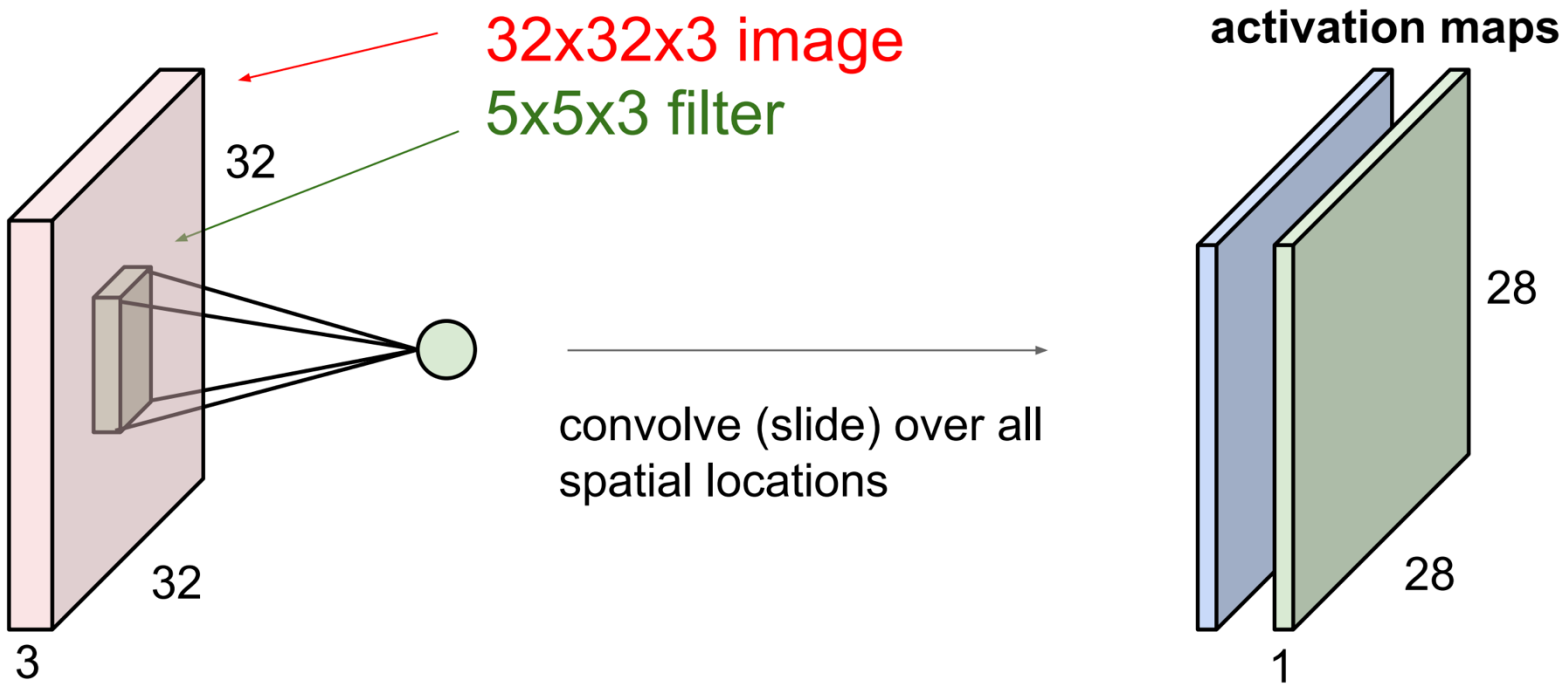
Convolution layer

- After the convolution, each filter generates an activation map



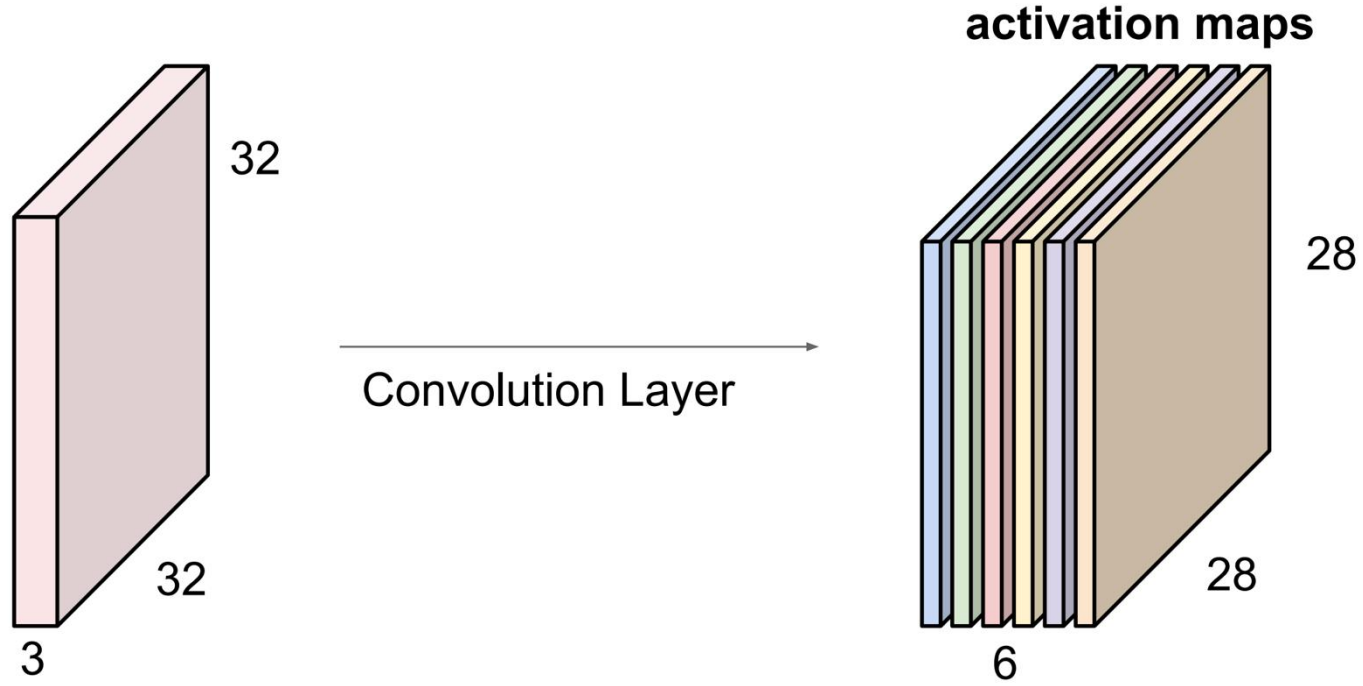
Convolution layer

- A new filter generates a second map



Convolution layer

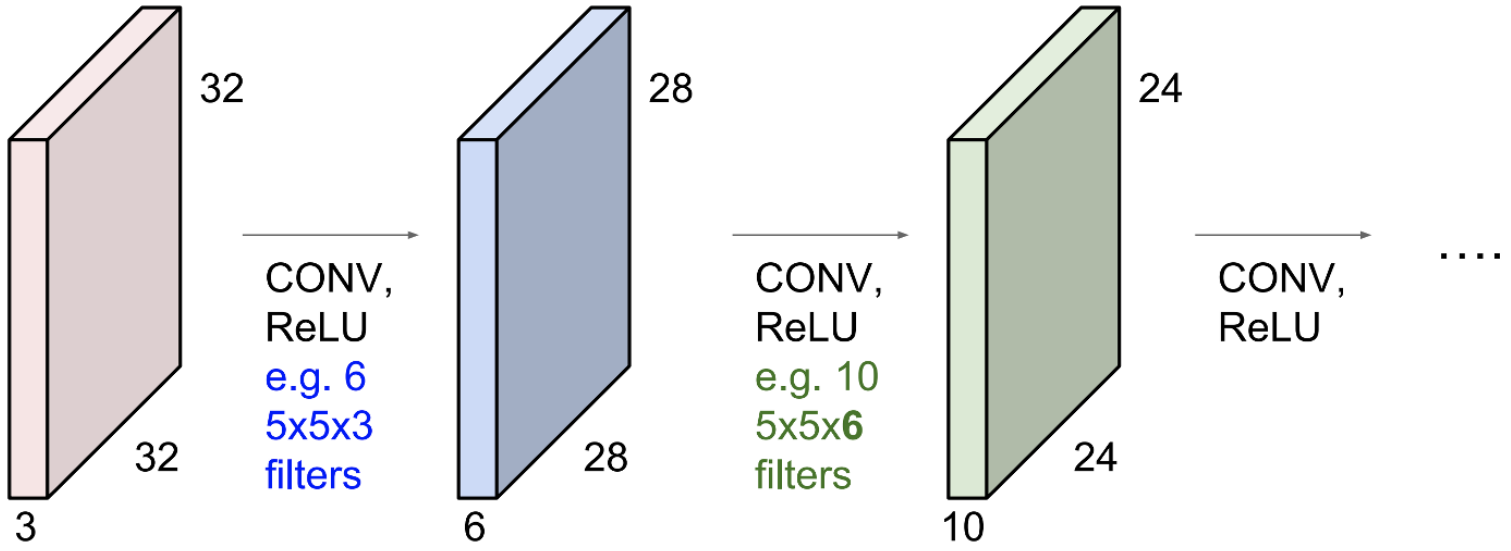
For example, if we had 6 5x5 filters, we'll get 6 separate activation maps:



We stack these up to get a "new image" of size 28x28x6!

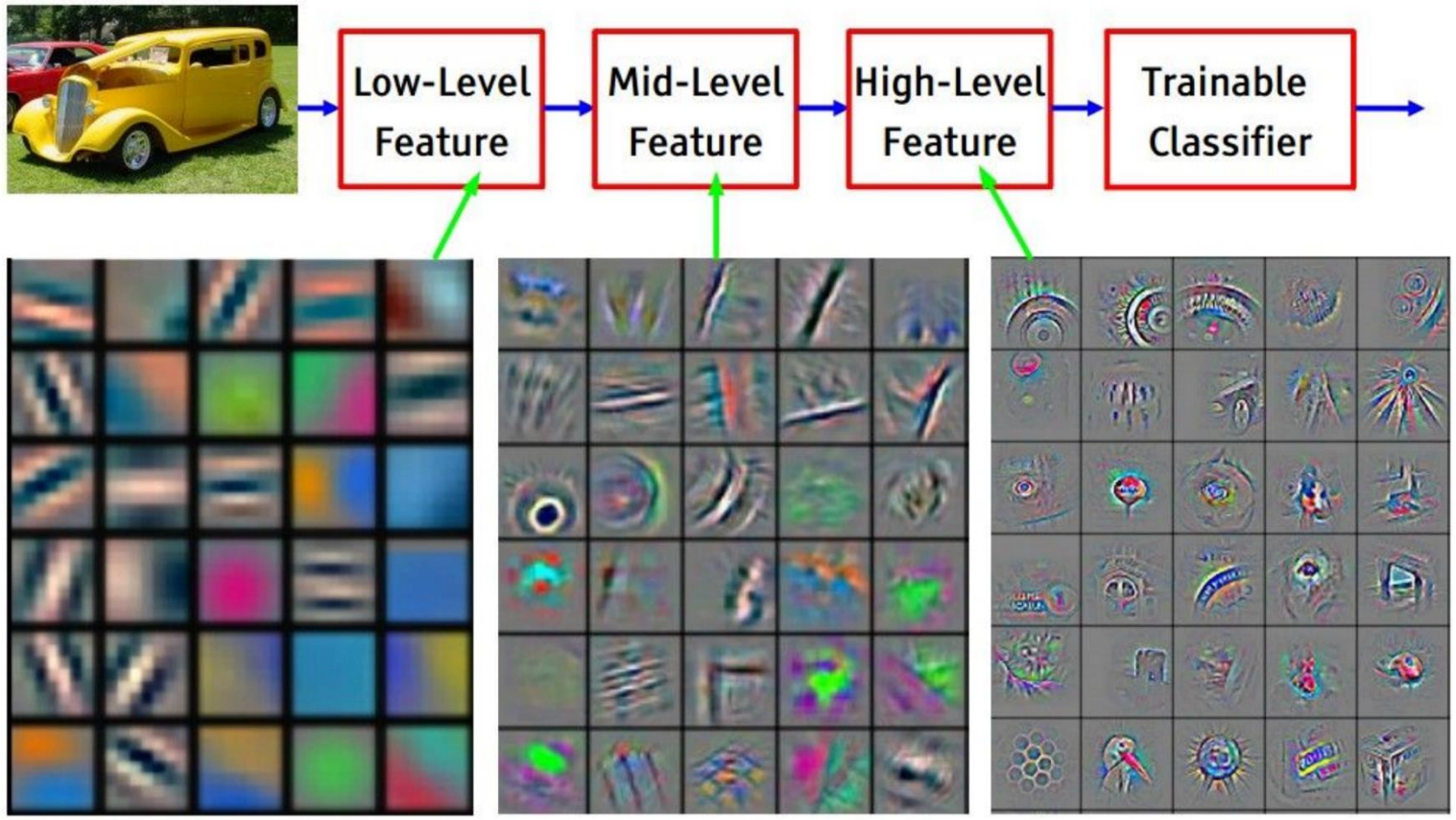
ConvNet architecture

- A sequence of convolutional layers, interspersed with activation functions

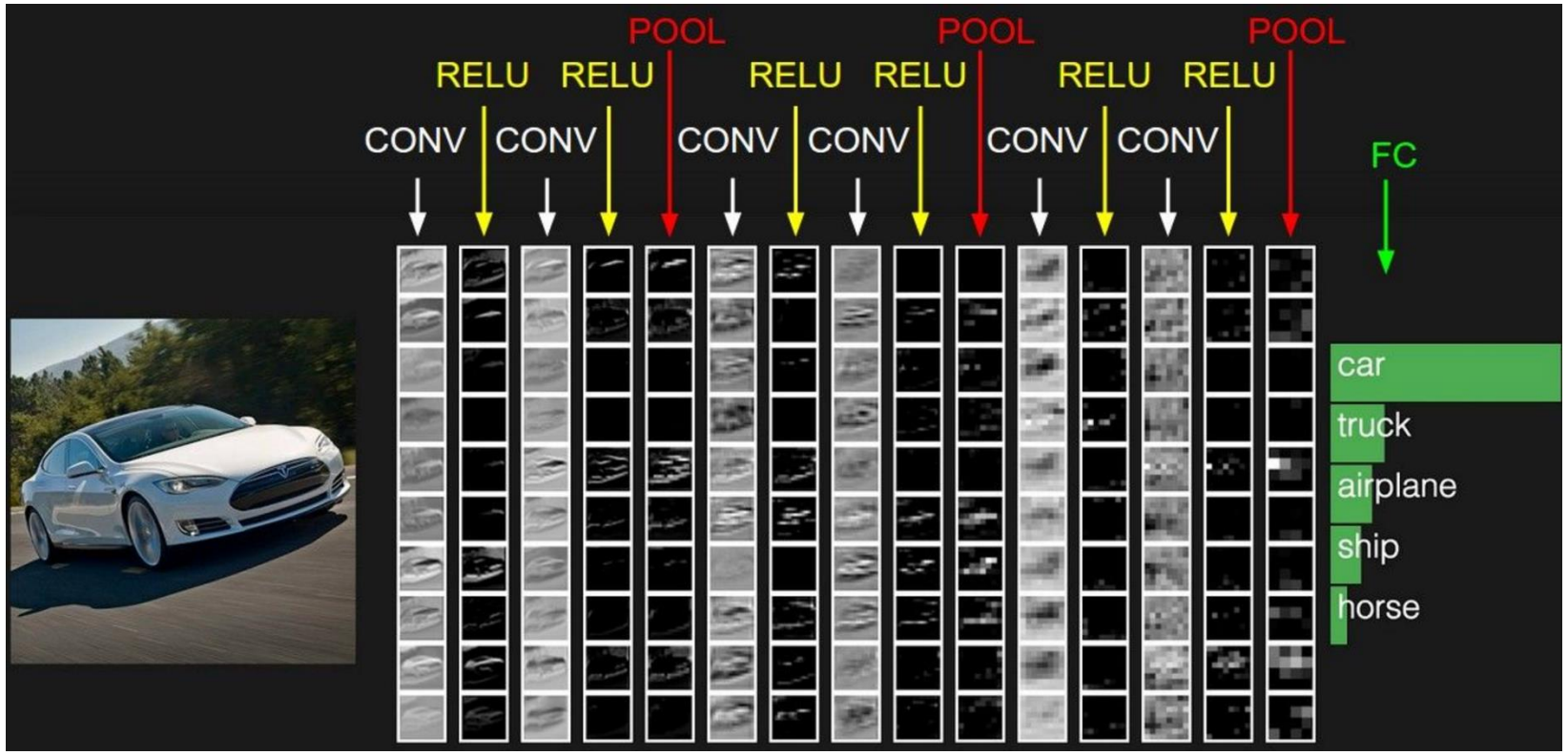


Example: image recognition

- Different kernels extract different levels of features (edge, textures, shape, parts, etc)

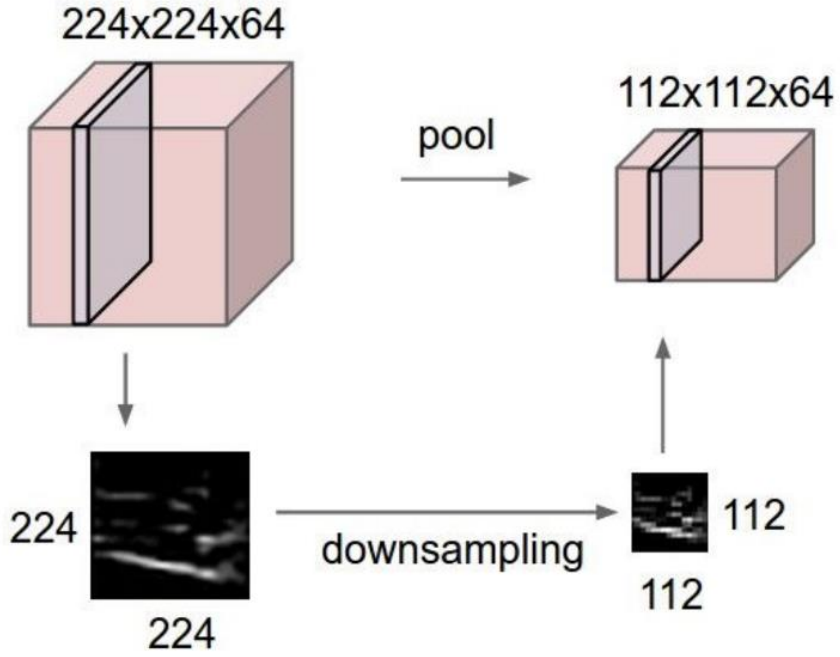


Example: ConvNet architecture for image recognition



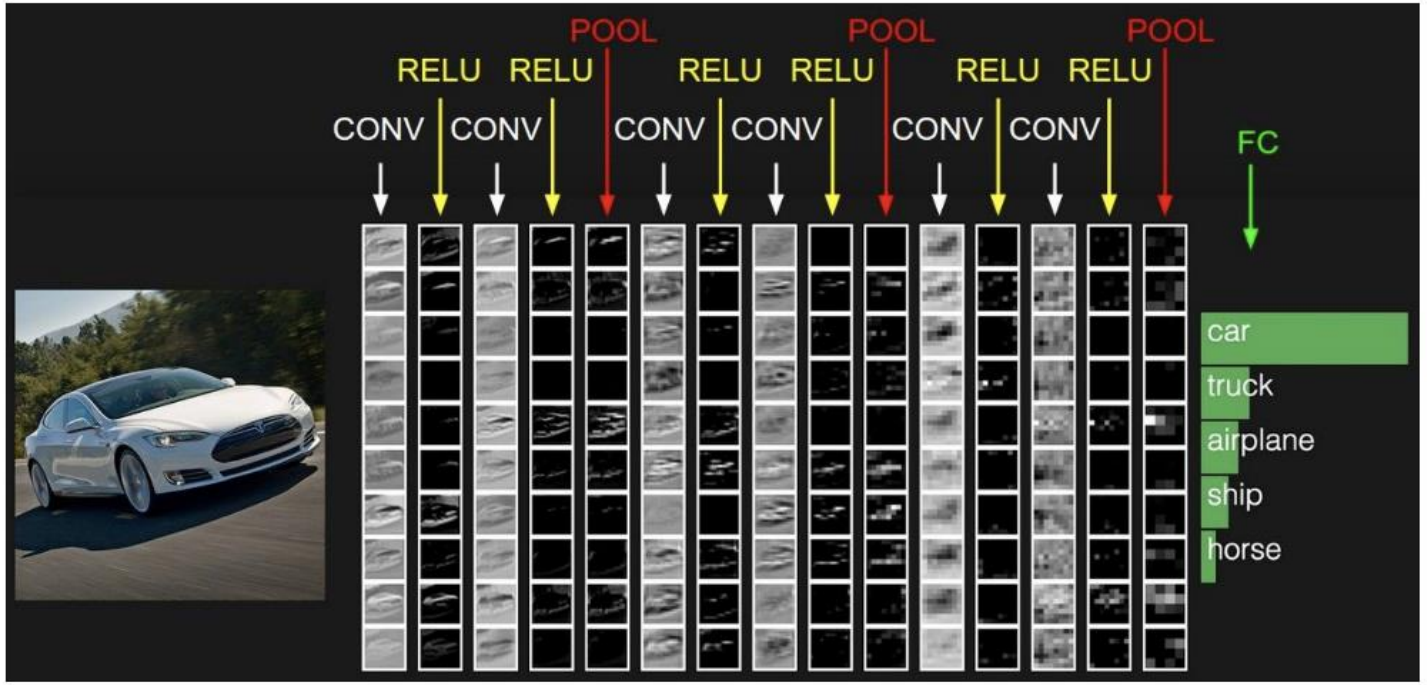
Pooling layer

- makes the representations smaller and more manageable
- operates over each activation map independently:



Fully-connected layer

- Contains neurons that connect to the entire input volume, as in ordinary Neural Networks



Convolutional Neural Network

- Discrete convolution in one dimension

$\vec{x}$: input vector (or signal) — n-elements

$\vec{w}$: filter or kernel — m-elements

The convolution is defined as

$$\vec{y} = \vec{x} * \vec{w}$$

The i-th element of the output is

$$y[i] = \sum_{k=-\infty}^{\infty} x[i-k]\, w[k]$$

let $l = i-k$ ⇒ $k = i-l$

or equivalently, $$= \sum_{k=-\infty}^{\infty} x[l]\, w[i-l]$$

As k increases, the index of $\vec{x}$ decreases while the index of $\vec{w}$ increases

This is effectively to flip the kernel and shift it along the signal

For example,

$\vec{x} = [1, 2, 3, 4, 5]$

$\vec{w} = [1, 0, -1]$ (This kernel is an edge-detector)

Step 1. $\vec{w}^{\,flipped} = [-1, 0, 1]$

Step 2: $\vec{x} = [1, 2, 3, 4, 5]$

$\vec{w}^{f} = [-1, 0, 1]$ → shift to the right for the next step

⇒ first output element $1 \cdot (-1) + 2 \cdot 0 + 3 \cdot 1 = 2$

second output element $2 \cdot (-1) + 3 \cdot 0 + 4 \cdot 1 = 2$

third output element $3 \cdot (-1) + 4 \cdot 0 + 5 \cdot 1 = 2$

Thus, output vector $\vec{y} = [2, 2, 2]$

The number of elements in the output vector is

$$n - m + 1$$

Here, $5 - 3 + 1 = 3$

(n: input vector size; m: filter or kernel size)

The size of the output vector is smaller than the input. If you want to keep output size equal to input, which is popular in CNNs, you can apply the "Zero-Padding" technique.

Note: when performing the summation, we assume the vectors $\vec{x}$ and $\vec{w}$ are filled with zeros. The output vector $\vec{y}$ thus also has infinite size with lots of zeros. This is not useful in practical situations. Instead, $\vec{x}$ is usually padded with a finite number of zeros. The number of zeros padded on each side is denoted by p.

For example, if $p = 2$

$$\vec{x} = [0, 0, 1, 2, 3, 4, 5, 0, 0]$$

$$\vec{w}^f = [-1, 0, 1]$$

$$y[0] = 0 \cdot (-1) + 0 \cdot 0 + 1 \cdot 1 = 1$$

$$y[1] = 0 \cdot (-1) + 1 \cdot 0 + 2 \cdot 1 = 2$$

$$y[2] = 1 \cdot (-1) + 2 \cdot 0 + 3 \cdot 1 = 2$$

$$y[3] = \dots$$

$y[6] = 3 \cdot (-1) + 4 \cdot 0 + 5 \cdot 1 = 2$

$y[7] = 4 \cdot (-1) + 5 \cdot 0 + 0 \cdot 1 = -4$

$y[8] = 5 \cdot (-1) + 0 \cdot 0 + 0 \cdot 1 = -5$

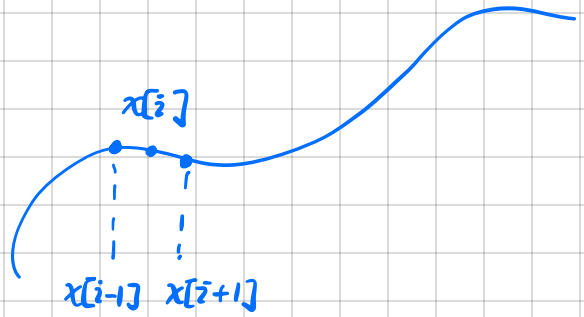
Thus, $\vec{y} = [1, 2, 2, 2, -4, -5]$

same size as the input vector.

About the kernel $\vec{w} = [1, 0, -1]$ or $\vec{w}^{flipped} = [-1, 0, 1]$.

This is a simple first-order difference kernel, which computes the "Right value" − "Left value"

In each 3-element window, it ignores the center (0), subtracts the left element from the right. This approximates a derivative i.e, the rate of change — which is why it's used as a basic edge detector.



$$\frac{x[i+1] - x[i-1]}{2}$$

Stride

When sliding the kernel over the input vector, we may take different number of steps.

$S = 1$: move the kernel one-element a time

$S = 2$: move the kernel two-elements a time

For example, $p=2$, $s=2$

$$\vec{x} = [3, 2, 1, 7, 1, 2, 5, 4]$$

$$\vec{w} = [\tfrac{1}{2}, \tfrac{3}{4}, 1, \tfrac{1}{4}]$$

$$\vec{w}^f = [\tfrac{1}{4}, 1, \tfrac{3}{4}, \tfrac{1}{2}]$$

$$y[0] = 3\cdot\tfrac{1}{4} + 2\cdot 1 + 1\cdot\tfrac{3}{4} + 7\cdot\tfrac{1}{2} = \tfrac{3}{4} + 2 + \tfrac{3}{4} + \tfrac{7}{2} = 7$$

$$y[1] = 1\cdot\tfrac{1}{4} + 7\cdot 1 + 1\cdot\tfrac{3}{4} + 2\cdot\tfrac{1}{2} = \tfrac{1}{4} + 7 + \tfrac{3}{4} + 1 = 9$$

$$y[2] = 1\cdot\tfrac{1}{4} + 2\cdot 1 + 5\cdot\tfrac{3}{4} + 4\cdot\tfrac{1}{2} = \tfrac{1}{4} + 2 + \tfrac{15}{4} + 2 = 8$$

Hence $\vec{y} = [7, 9, 8]$

Note: strides affect both computation and the output size

For $s=1$, the output is large. Even though it preserves detail, the computation could be slow.

For $s=2+$, the output is smaller (downsampling), and the computation is faster, but it may lose detail.

If input size is n, kernel size is m, padding is p, stride is s, then output size

$$\left\lfloor \frac{n + 2p - m}{s} \right\rfloor + 1$$

↓ floor function

For example, $\vec{x} = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]$ $n=10$

$\vec{w} = [1, 0, -1]$ $m=3$

$P = 2$

$S = 2$

$\Rightarrow$

$\vec{x}^P = [0, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 0, 0]$

$\vec{w}^f = [-1, 0, 1]$

The output should be a vector with six elements.

$$\left\lfloor \frac{n+2p-m}{s} \right\rfloor + 1 = \left\lfloor \frac{10+4-3}{2} \right\rfloor + 1 = \lfloor 5.5 \rfloor + 1$$

$$= 5 + 1 = 6$$

Discrete convolution in 2D

For input matrix (image)

$$X_{n_1 \times n_2}$$

$m_1 < n_1$

$m_2 < n_2$

Filter matrix

$$W_{m_1 \times m_2}$$

output matrix

$$Y = X * W$$

or $$Y[i,j] = \sum_{k_1} \sum_{k_2} X[i-k_1, j-k_2]\, W[k_1, k_2]$$

All the previously mentioned techniques, such as zero padding, rotating the kernel matrix, and the use of strides, are also applicable to 2D convolutions.

Note: in 2D, the kernel matrix is rotated by 180°: which is equivalent to first flip it left-right (horizontal flip), and then flip it up-down (vertical flip)

e.g. $$W = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} \Rightarrow W' = \begin{bmatrix} 3 & 2 & 1 \\ 6 & 5 & 4 \\ 9 & 8 & 7 \end{bmatrix}$$

$$\Rightarrow W^r = \begin{bmatrix} 9 & 8 & 7 \\ 6 & 5 & 4 \\ 3 & 2 & 1 \end{bmatrix}$$

Subsampling layers or Pooling layers

This is another important operation in CNNs.

- It reduces the size of feature vector or matrix
- It keeps the most important information
- It makes the model faster and more robust to small changes (like small shifts in an image which usually corresponds to the same label)

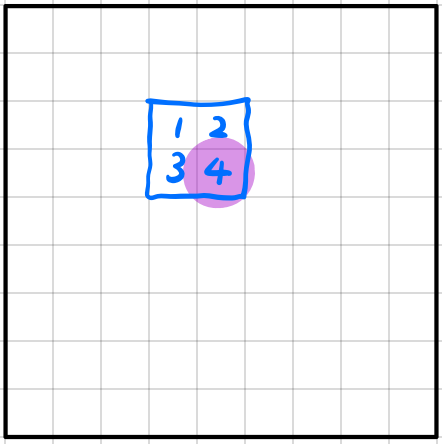
Two common approaches.

(1) Max Pooling

It takes the maximum value in a small window.

Imagine sanning over the image with a little square, like 2x2.

For each window, you keep the largest number.



4

(2) Mean Pooling

It takes the average value in a window.

2.5