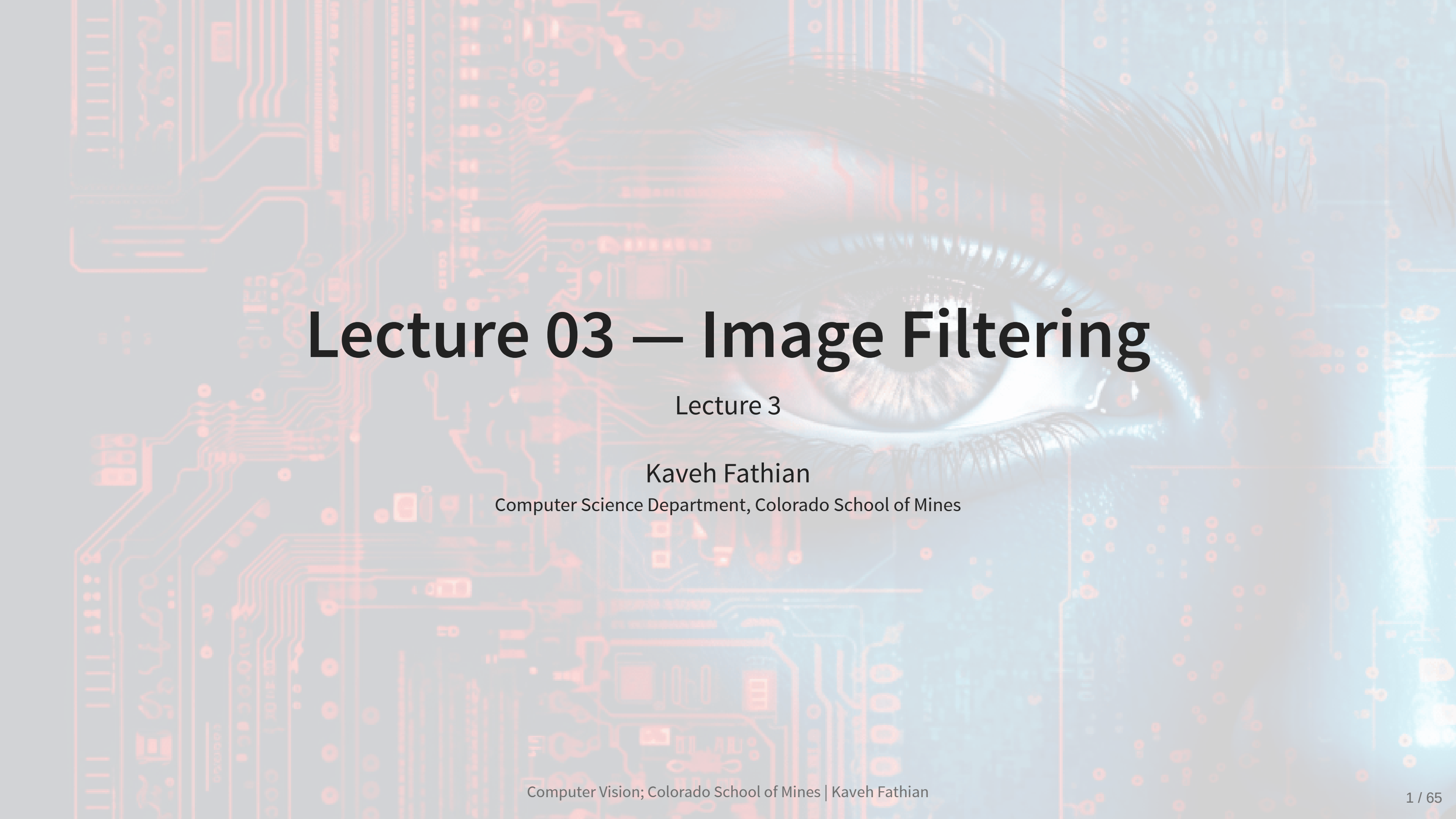




Lecture 03 — Image Filtering

Lecture 3

Kaveh Fathian

Computer Science Department, Colorado School of Mines

Image Filtering



Learning Objectives

By the end of this lecture, you should be able to:

- Explain filtering as a local neighborhood operation.
- Distinguish correlation and convolution operations.
- Predict the behavior of common kernels.
- Explain linearity, shift invariance, boundary handling, and separability.
- Apply Gaussian, Sobel, sharpening, and median filters.
- Use correlation for basic template matching and recognize its limitations.

Fundamental Equations of Computer Vision

1. Image Filtering

$$y[m, n] = \sum_{k, l} h[k, l] I[m + k, n + l]$$

2. Optical Flow

$$I(x, y, t) = I(x + \Delta x, y + \Delta y, t + \Delta t)$$

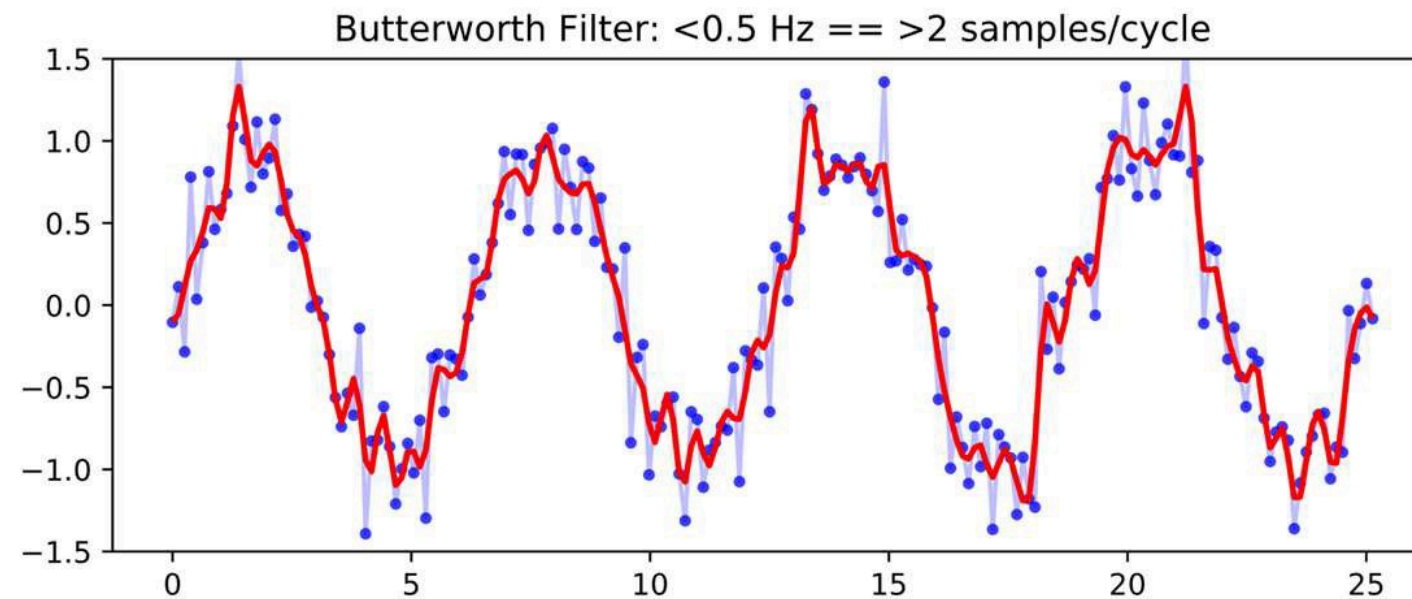
3. Camera Geometry

$$\mathbf{x} = \mathbf{K} \begin{bmatrix} \mathbf{R} & \mathbf{t} \end{bmatrix} \mathbf{X} \quad \mathbf{x}^T \mathbf{F} \mathbf{x}' = 0$$

4. Machine Learning

$$\arg \min_{\mathcal{S}} \sum_{i=1}^k \sum_{\mathbf{x} \in S_i} \|\mathbf{x} - \boldsymbol{\mu}_i\|^2 \quad y = \varphi(\mathbf{w}^T \mathbf{x} + b)$$

Filtering Modifies a Signal



Noisy samples and a smoothed estimate

Filtering transforms a signal to:

- suppress undesired components such as noise
- emphasize useful structure
- extract specific features

Filters can operate on continuous or discrete signals in one or more dimensions.

1D Moving Average Filter

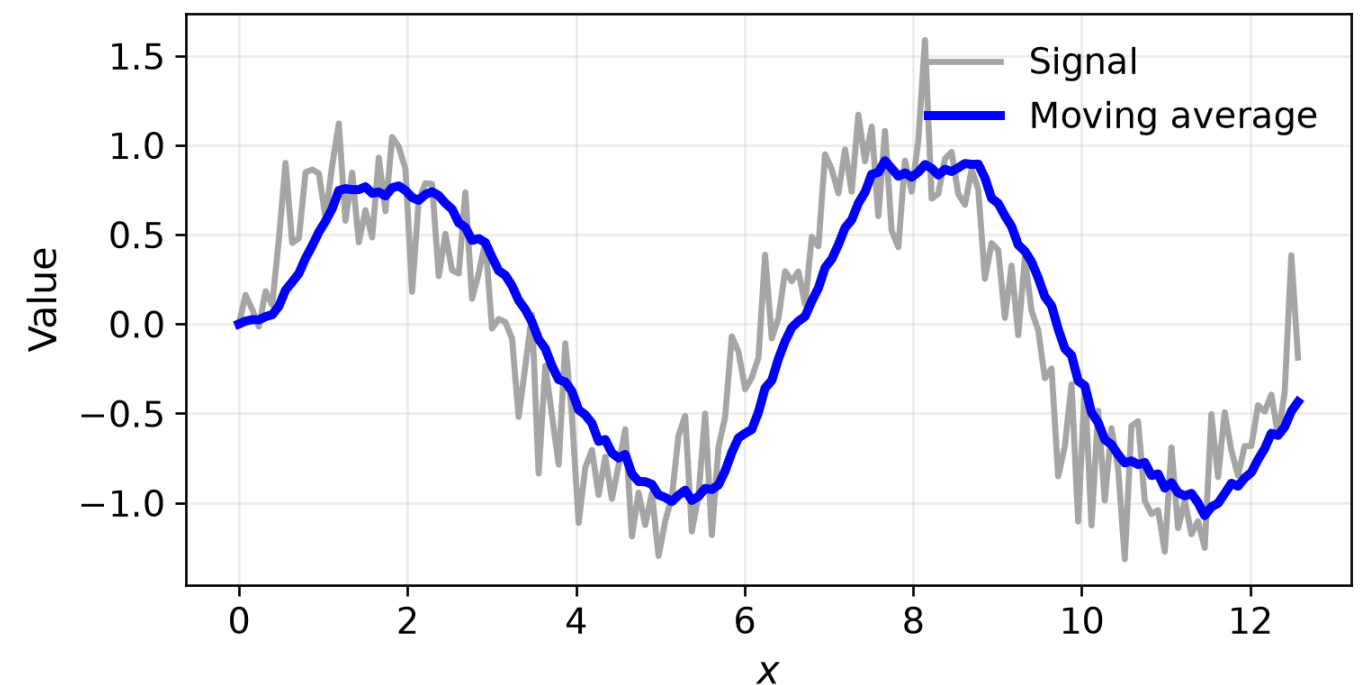
For a window size k :

$$y[n] = \frac{1}{k} \sum_{i=n-k+1}^n (1)I[i]$$

or

$$y[n] = \frac{1}{k} \begin{bmatrix} 1 & 1 & \dots & 1 \end{bmatrix} \begin{bmatrix} I[n-k+1] \\ I[n-k+2] \\ \vdots \\ I[n] \end{bmatrix}$$

```
1 x = np.linspace(0, 4*np.pi, 160)
2 rng = np.random.default_rng(7)
3 noisy = np.sin(x) + 0.28*rng.normal(size=x.size)
4
5 k = 10
6 ker = np.ones(k)/k
7 smoothed = np.convolve(noisy, ker, mode="full")
8
9 fig, ax = plt.subplots(figsize=(6.5, 3.5))
10 ax.plot(x, noisy, c="0.65", lw=2, label="Signal")
11 ax.plot(x, smoothed, c="blue", lw=3, label="Moving average")
12 ax.set(xlabel="$x$", ylabel="Value")
13 ax.legend(frameon=False); ax.grid(alpha=0.2)
14 plt.tight_layout(); plt.show()
```

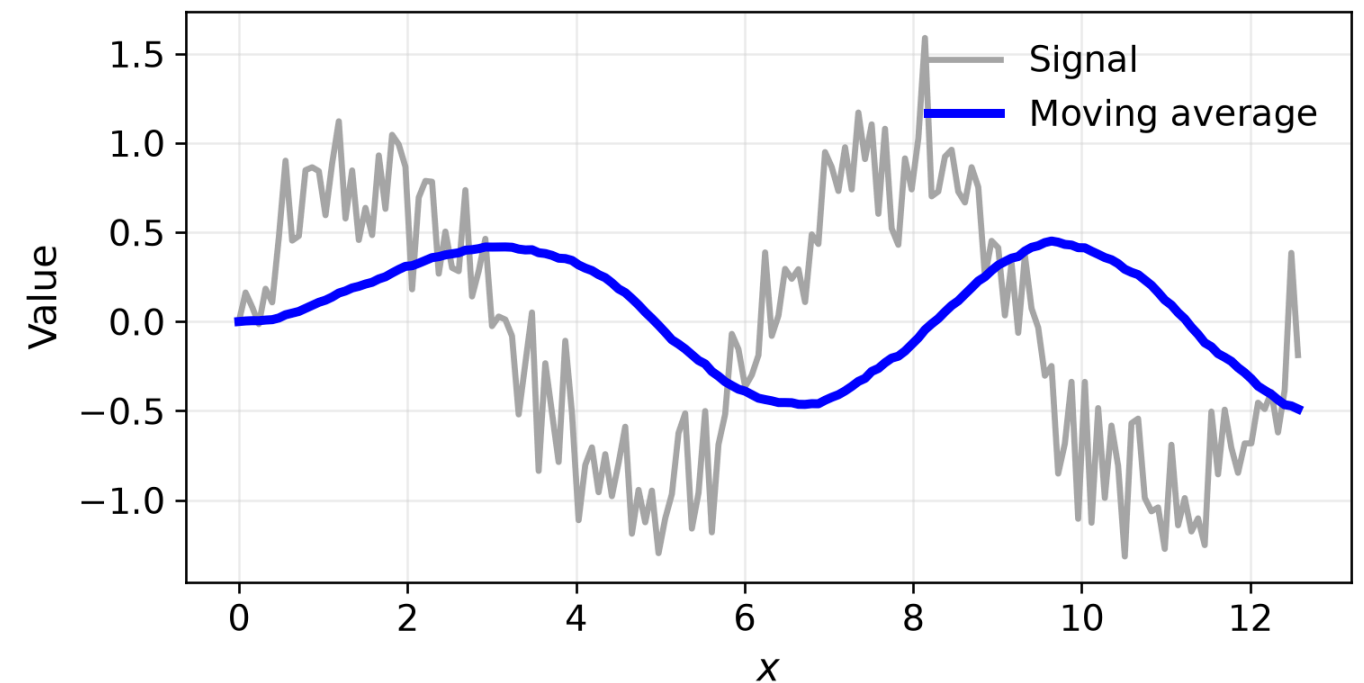


The Window Size k

A larger averaging window:

- removes more rapid variation
- produces a smoother output
- spreads changes over a wider region
- can erase narrow but meaningful features

```
1 x = np.linspace(0, 4*np.pi, 160)
2 rng = np.random.default_rng(7)
3 noisy = np.sin(x) + 0.28*rng.normal(size=x.size)
4
5 k = 50
6 ker = np.ones(k)/k
7 smoothed = np.convolve(noisy, ker, mode="full")
8
9 fig, ax = plt.subplots(figsize=(6.5, 3.5))
10 ax.plot(x, noisy, c="0.65", lw=2, label="Signal")
11 ax.plot(x, smoothed, c="blue", lw=3, label="Moving average")
12 ax.set(xlabel="$x$", ylabel="Value")
13 ax.legend(frameon=False); ax.grid(alpha=0.2)
```



Filter Kernel

What is this filter doing?

$$y[n] = \begin{bmatrix} 0 & 0 & \dots & 0 & 1 \end{bmatrix} \begin{bmatrix} I[n - k + 1] \\ I[n - k + 2] \\ \vdots \\ I[n] \end{bmatrix}$$

```
1 x = np.linspace(0, 4*np.pi, 160)
2 rng = np.random.default_rng(7)
3 noisy = np.sin(x) + 0.28*rng.normal(size=x.size)
4
5 k = 20
6 ker = np.zeros(k); ker[-1] = 1
7 output = np.correlate(np.pad(noisy, (k-1, 0)), ker)
8
9 fig, ax = plt.subplots(figsize=(6.5, 3.5))
10 ax.plot(x, noisy, c="0.65", lw=4, label="Input")
11 ax.plot(x, output, "b--", lw=2.5, label="Output")
12 ax.set(xlabel="$x$", ylabel="Value")
```

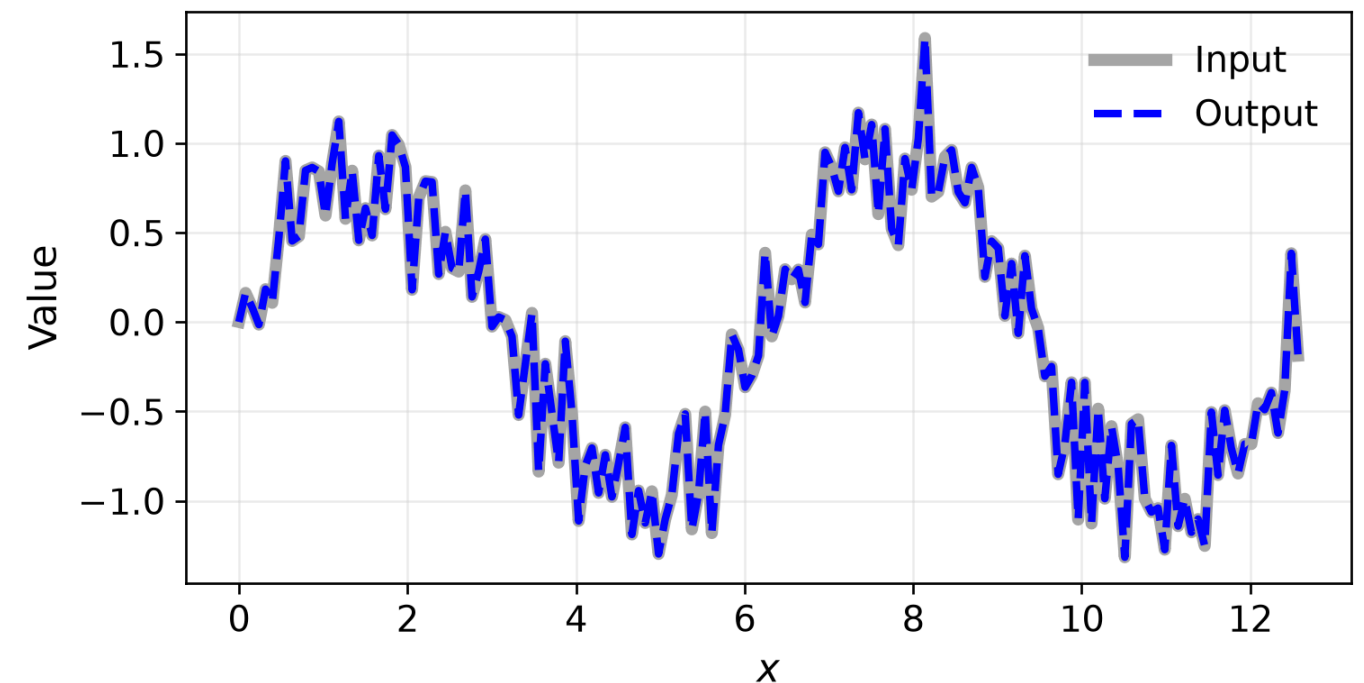


Image Filtering

Image filtering computes a function of the local neighborhood at each pixel position:

$$y[m, n] = \sum_{k, l} h[k, l] I[m + k, n + l]$$

Filter window/kernel size: “ k ” and “ l ”

Image Filtering

Image filtering computes a function of the local neighborhood at each pixel position:

$$\underbrace{y[m, n]}_{\text{output}} = \sum_{k, l} \underbrace{h[k, l]}_{\text{filter kernel weights}} \underbrace{I[m + k, n + l]}_{\text{neighboring image value}}$$

- $y[m, n]$: output value at image coordinate (m, n)
- $h[k, l]$: filter coefficient at local offset (k, l)
- $I[m + k, n + l]$: input image value at the corresponding neighboring location

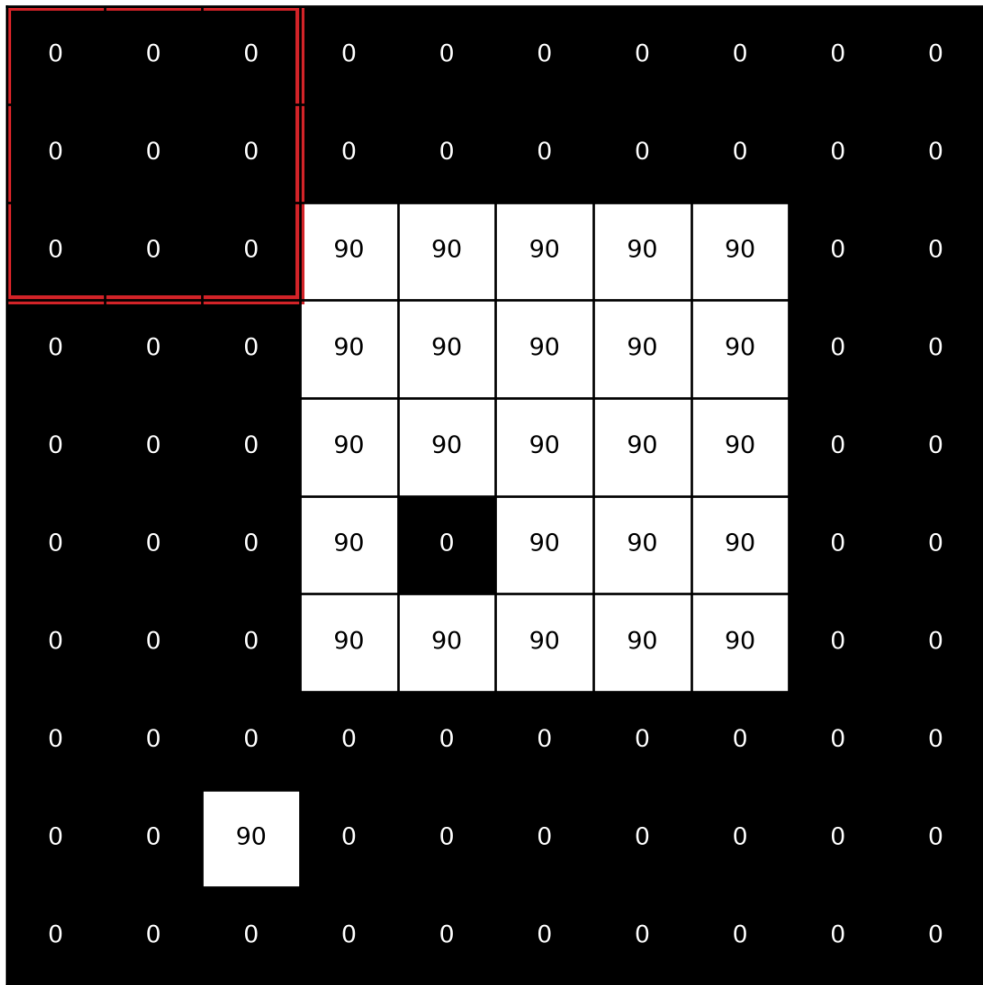
Example: Box Filter Kernel

- The **box filter** replaces each pixel with the average of its neighborhood.
- The size of the neighborhood is determined by the size of the filter kernel
- For example, a 3×3 box filter has the kernel:

$$h[k, l] = \frac{1}{9} \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}.$$

Box Filter: First Position

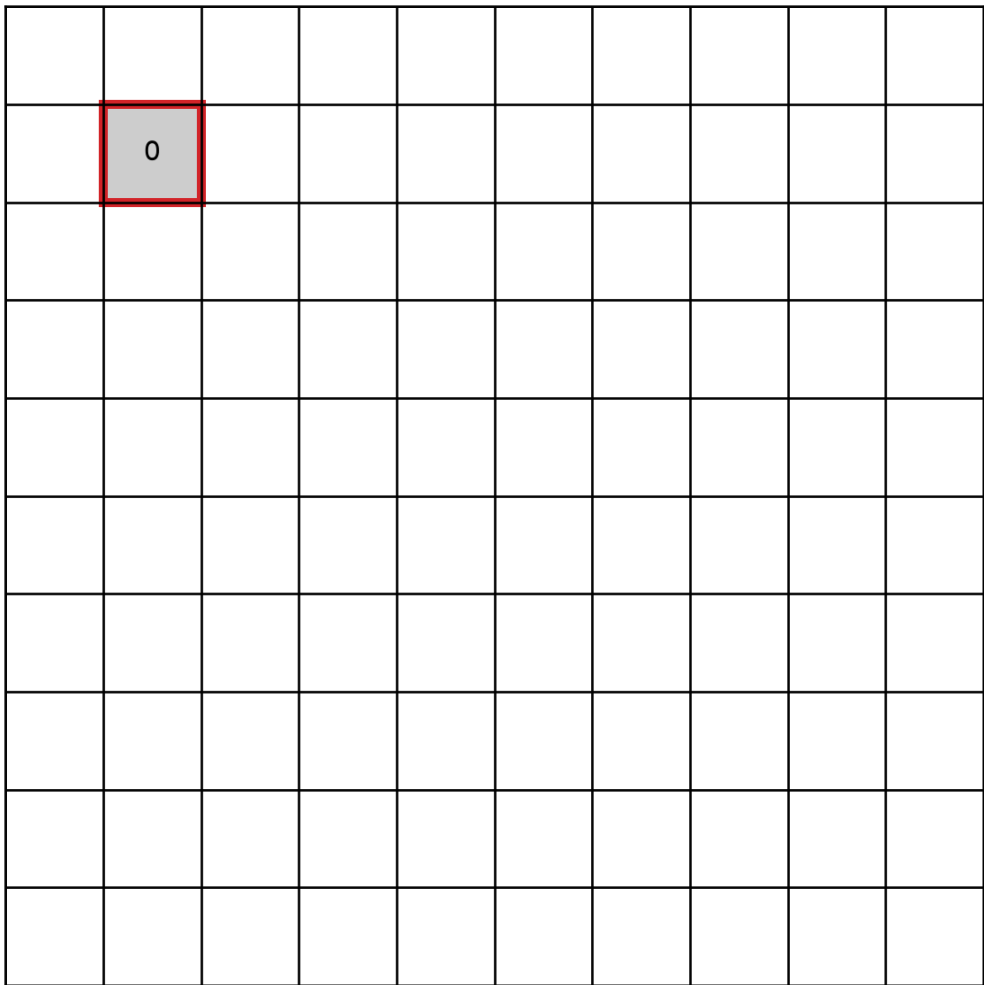
Input $I[m, n]$



$h[k, l]$



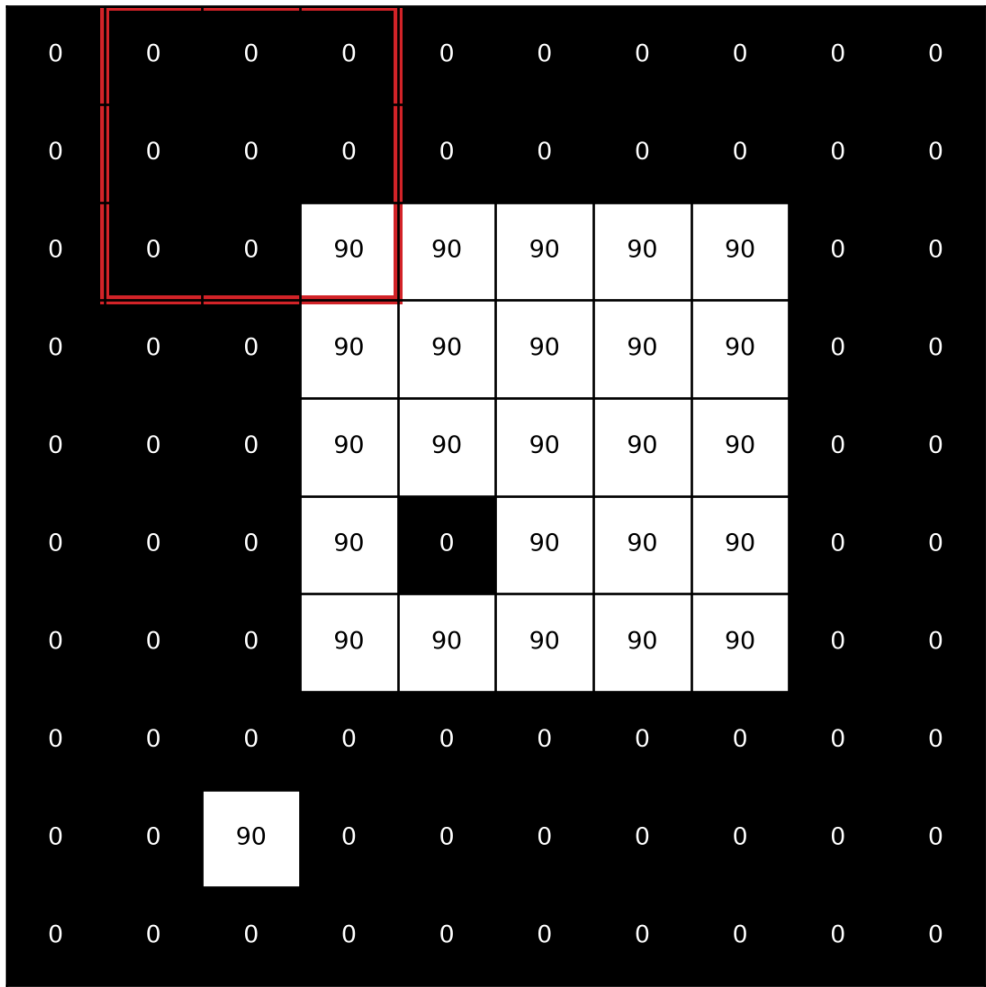
Output $y[m, n]$



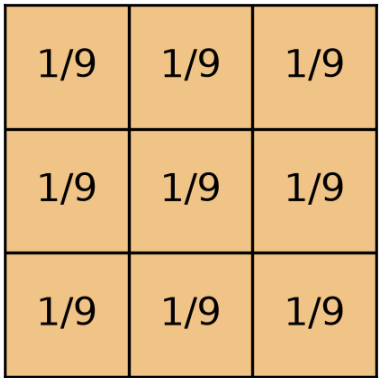
$$y[1, 1] = \sum_{k=-1}^1 \sum_{l=-1}^1 h[k, l] I[1 + k, 1 + l] = 0$$

Box Filter: Move One Pixel Right

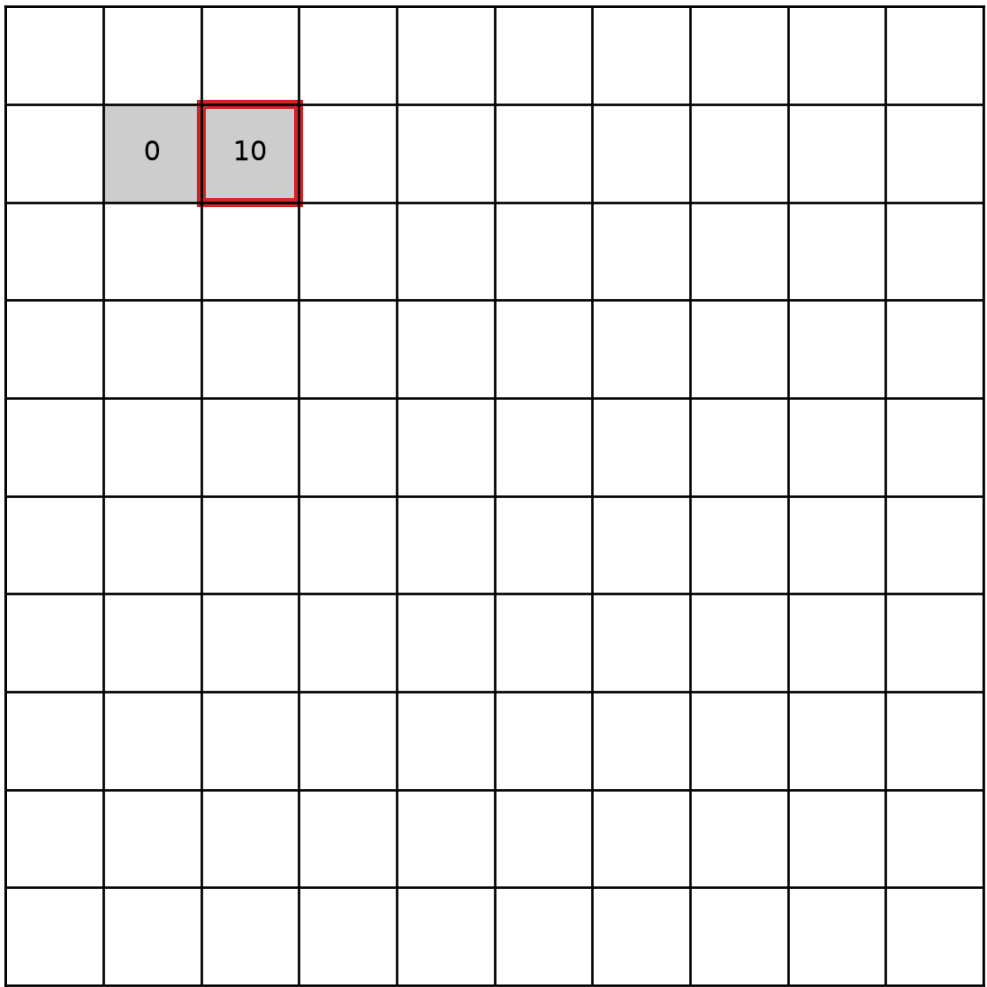
Input $I[m, n]$



$h[k, l]$

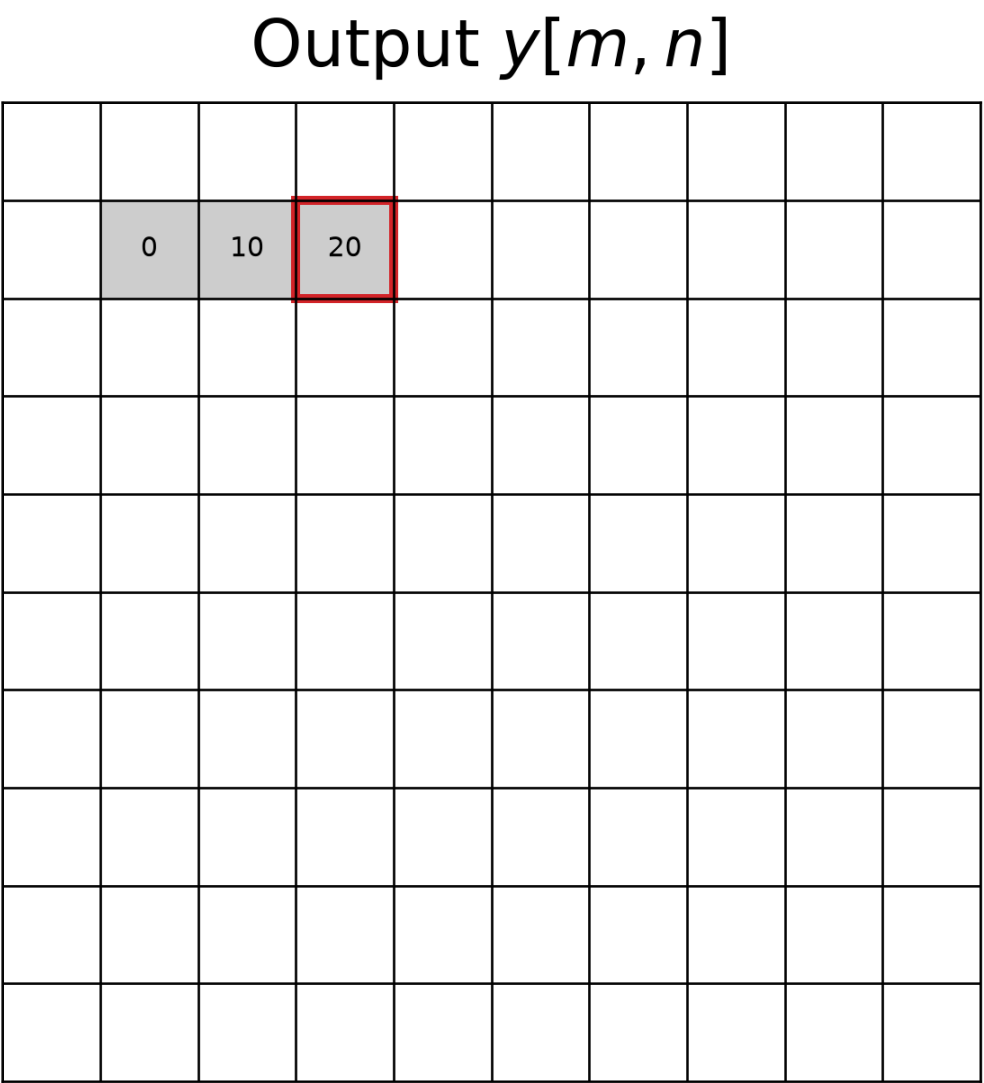
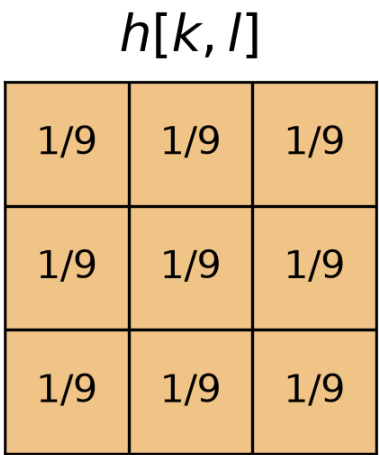
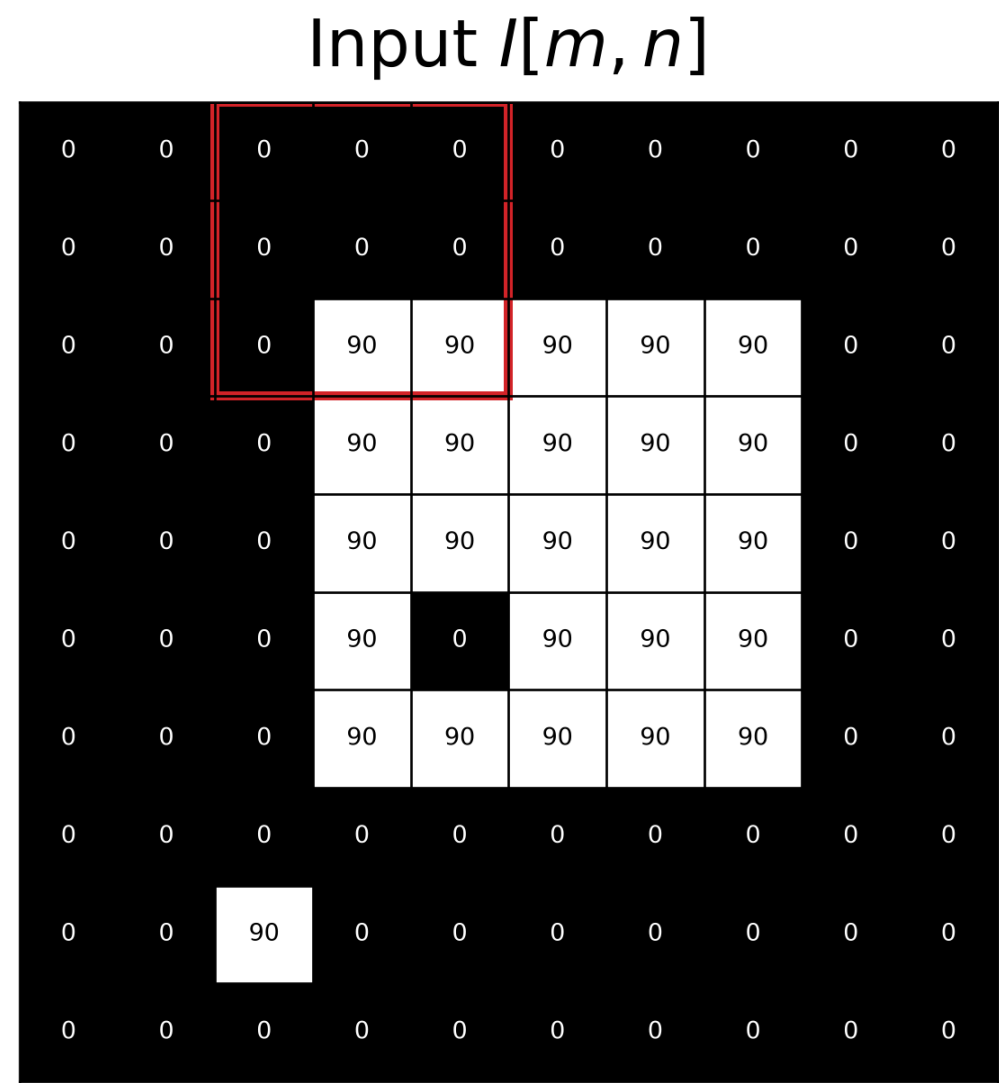


Output $y[m, n]$



$$y[1, 2] = \sum_{k=-1}^1 \sum_{l=-1}^1 h[k, l] I[1 + k, 2 + l] = 10$$

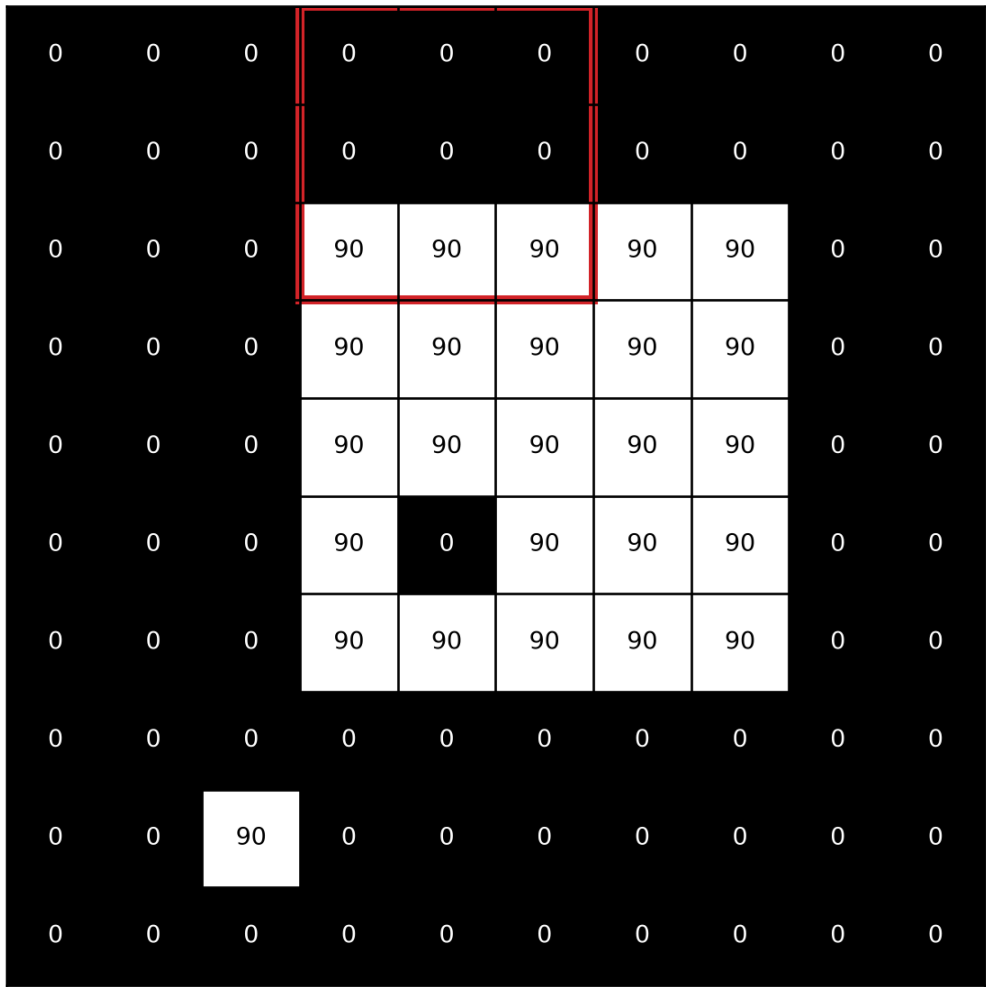
Box Filter: Continue the Sweep



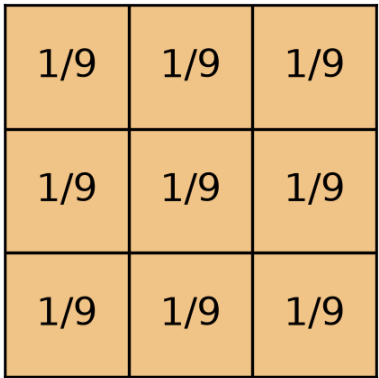
$$y[1, 3] = \sum_{k=-1}^1 \sum_{l=-1}^1 h[k, l] I[1 + k, 3 + l] = 20$$

Box Filter: Continue the Sweep

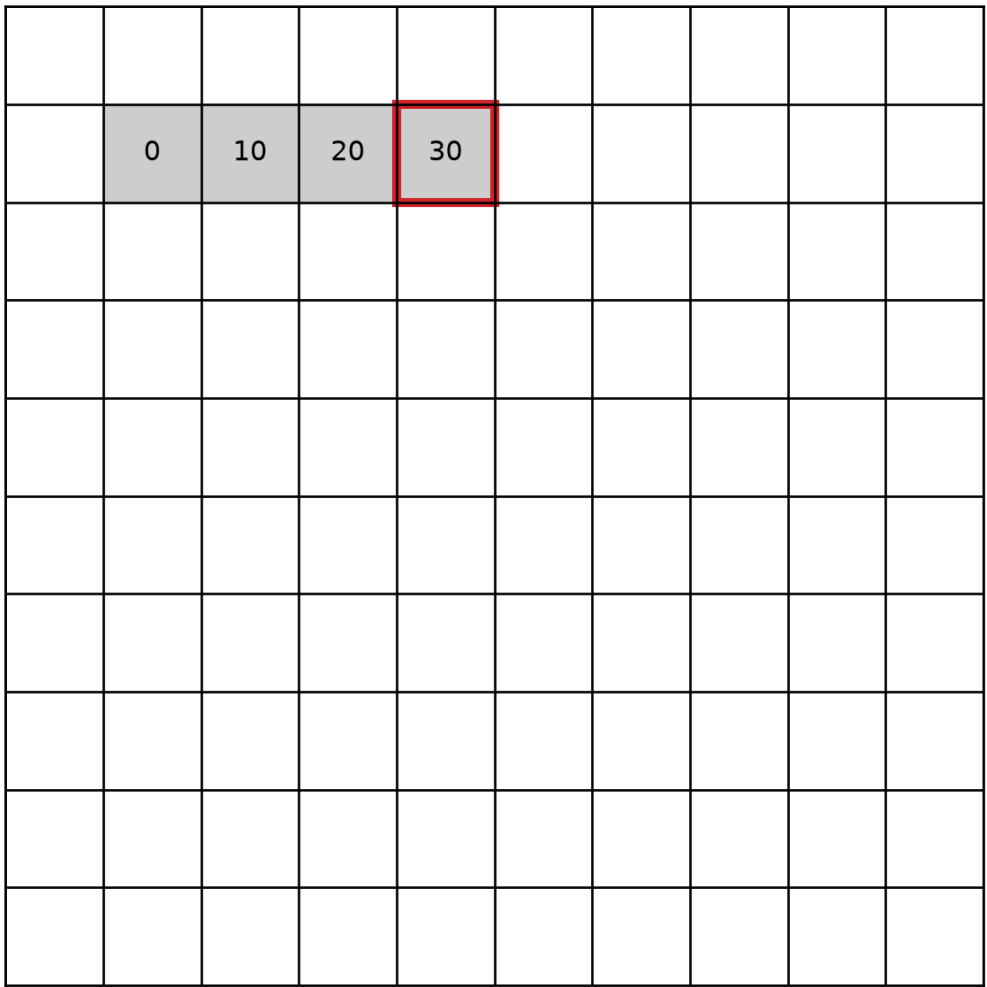
Input $I[m, n]$



$h[k, l]$



Output $y[m, n]$



$$y[1, 4] = \sum_{k=-1}^1 \sum_{l=-1}^1 h[k, l] I[1 + k, 4 + l] = 30$$

Box Filter: Continue the Sweep

Input $I[m, n]$

0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	90	90	90	90	90	0	0
0	0	0	90	90	90	90	90	0	0
0	0	0	90	90	90	90	90	0	0
0	0	0	90	0	90	90	90	0	0
0	0	0	90	90	90	90	90	0	0
0	0	0	0	0	0	0	0	0	0
0	0	90	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0

$h[k, l]$

1/9	1/9	1/9
1/9	1/9	1/9
1/9	1/9	1/9

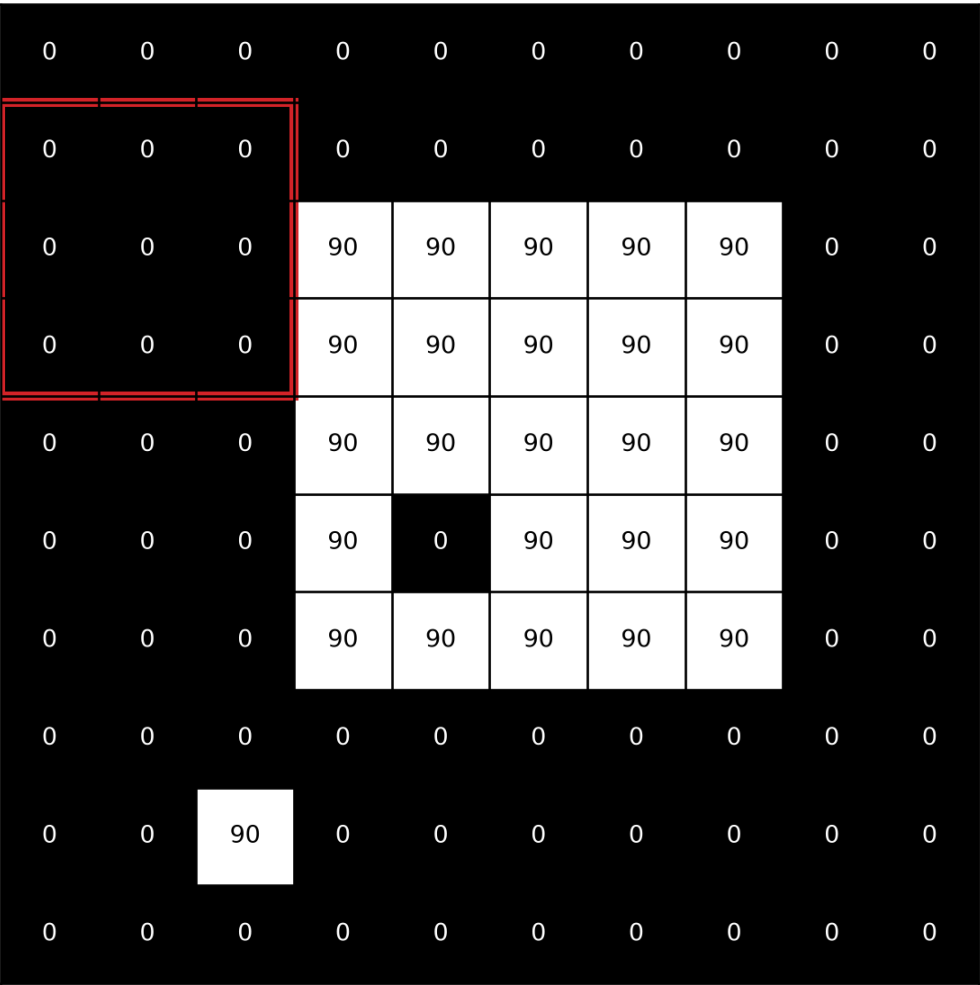
Output $y[m, n]$

	0	10	20	30	30	30	20	10	

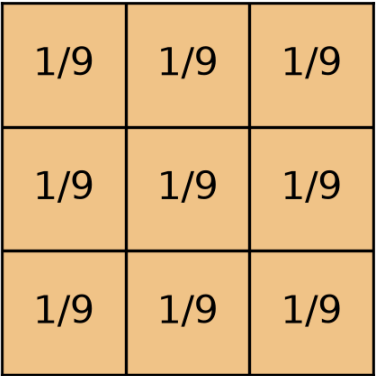
$$y[1, 8] = \sum_{k=-1}^1 \sum_{l=-1}^1 h[k, l] I[1 + k, 8 + l] = 10$$

Box Filter: Continue Across the Column

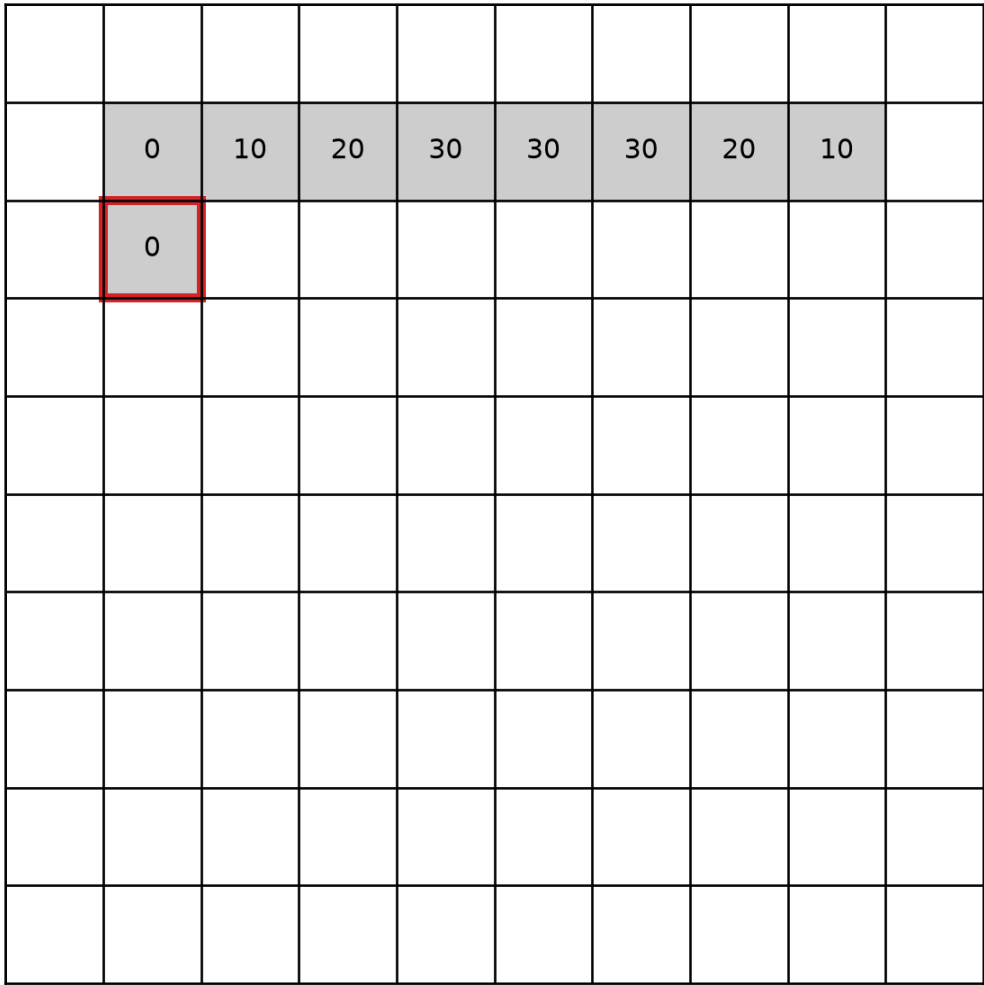
Input $I[m, n]$



$h[k, l]$

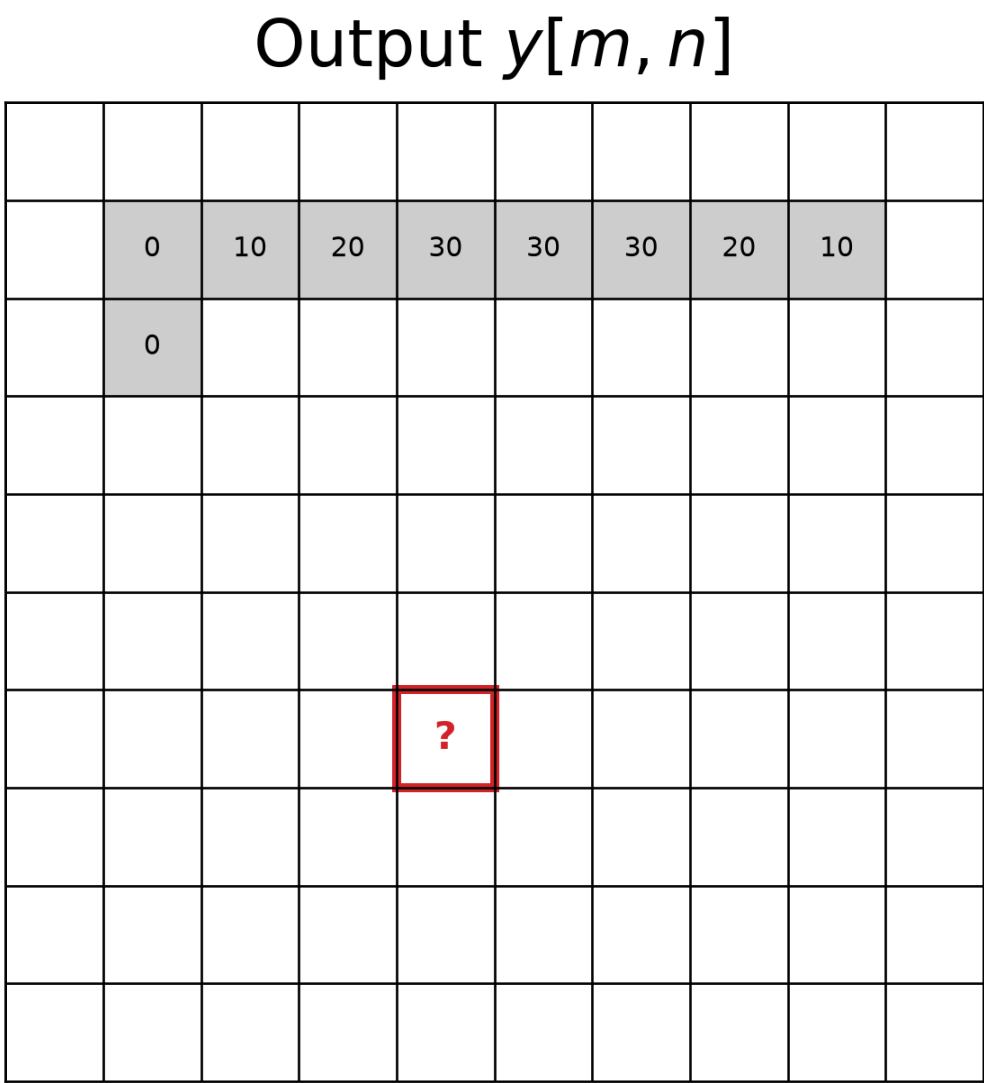
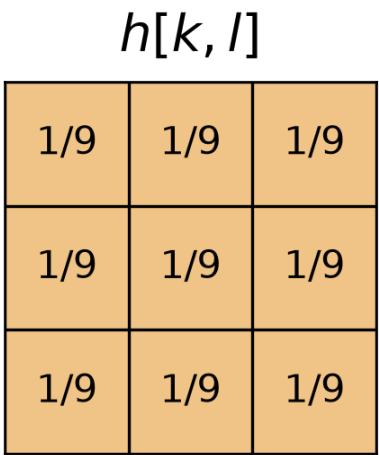
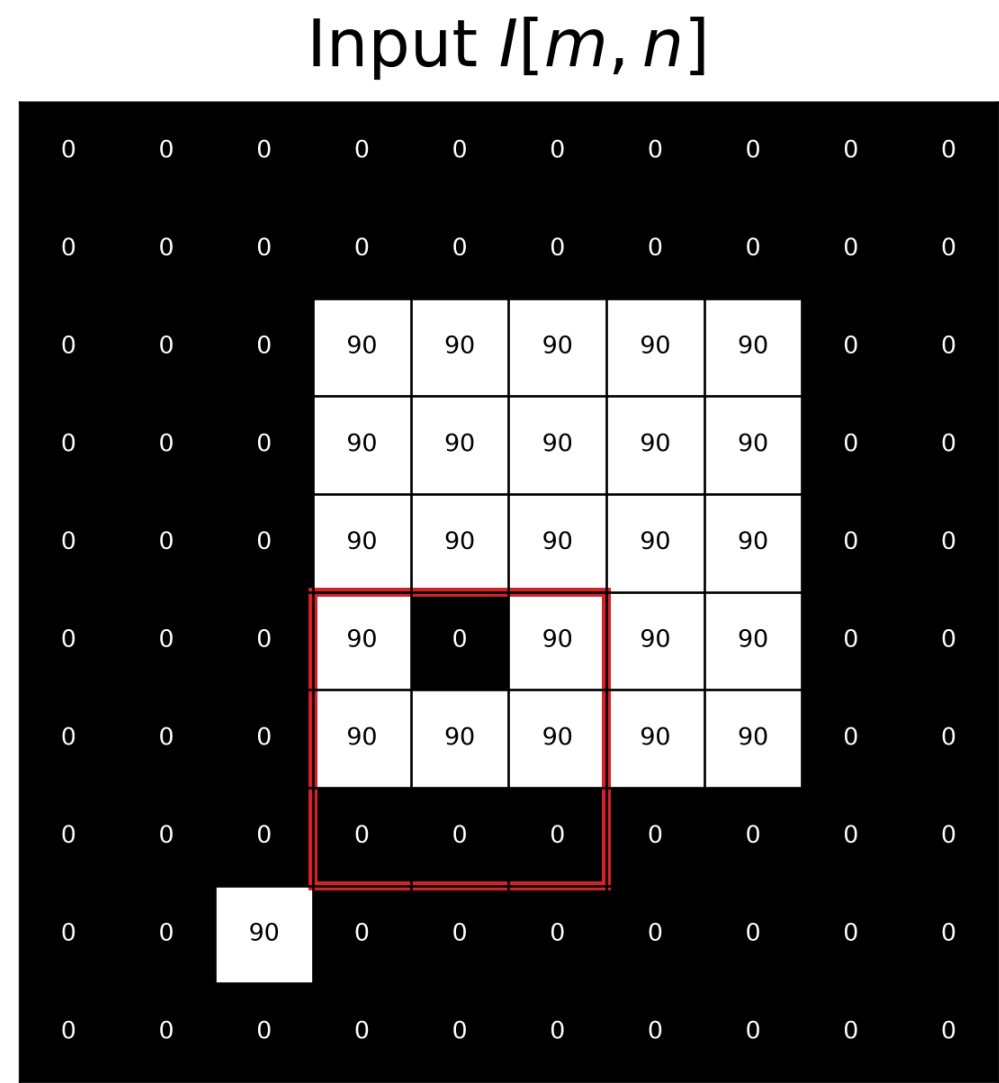


Output $y[m, n]$



$$y[2, 1] = \sum_{k=-1}^1 \sum_{l=-1}^1 h[k, l] I[2 + k, 1 + l] = 0$$

Box Filter: Try This Position

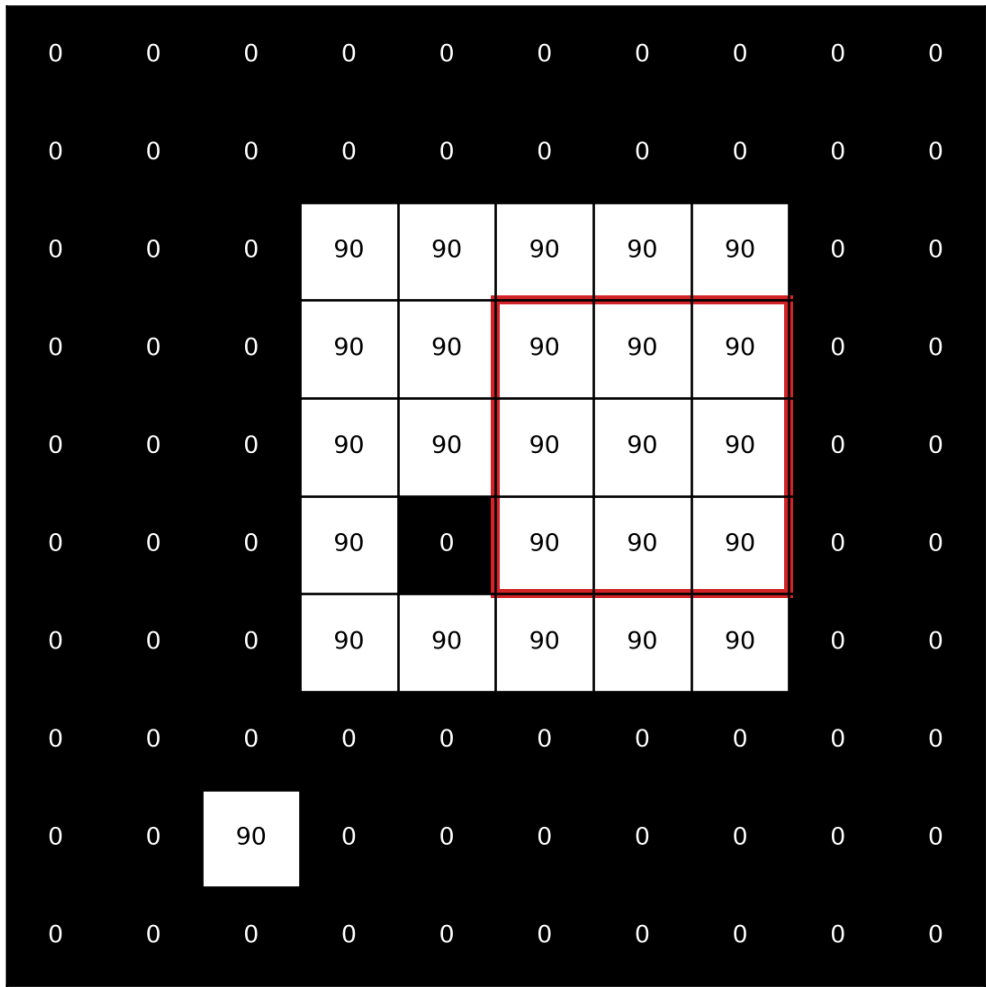


What is $y[6, 4]$?

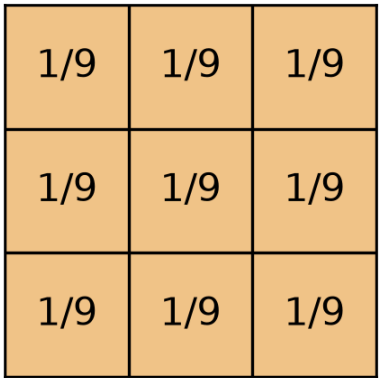
Add the nine pixels under the red window, then divide by nine.

Box Filter: Another Position

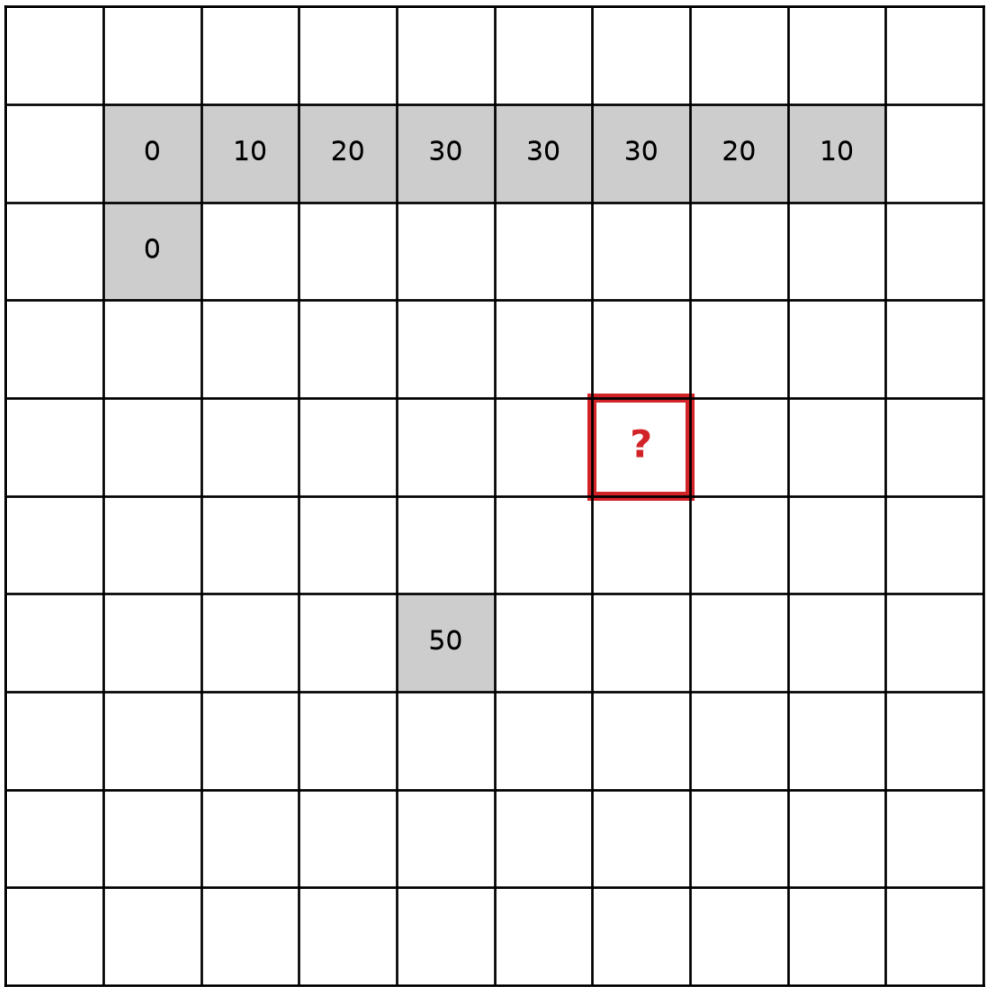
Input $I[m, n]$



$h[k, l]$

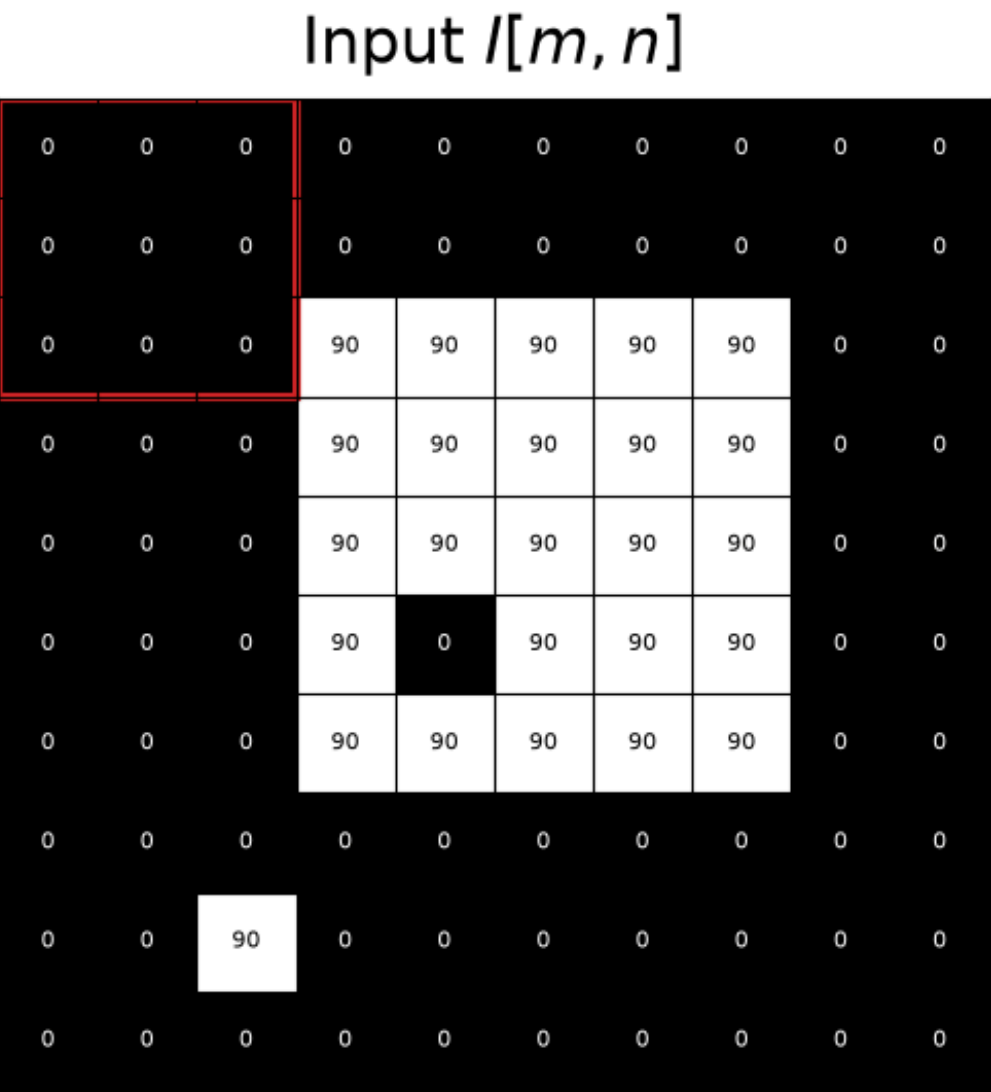


Output $y[m, n]$

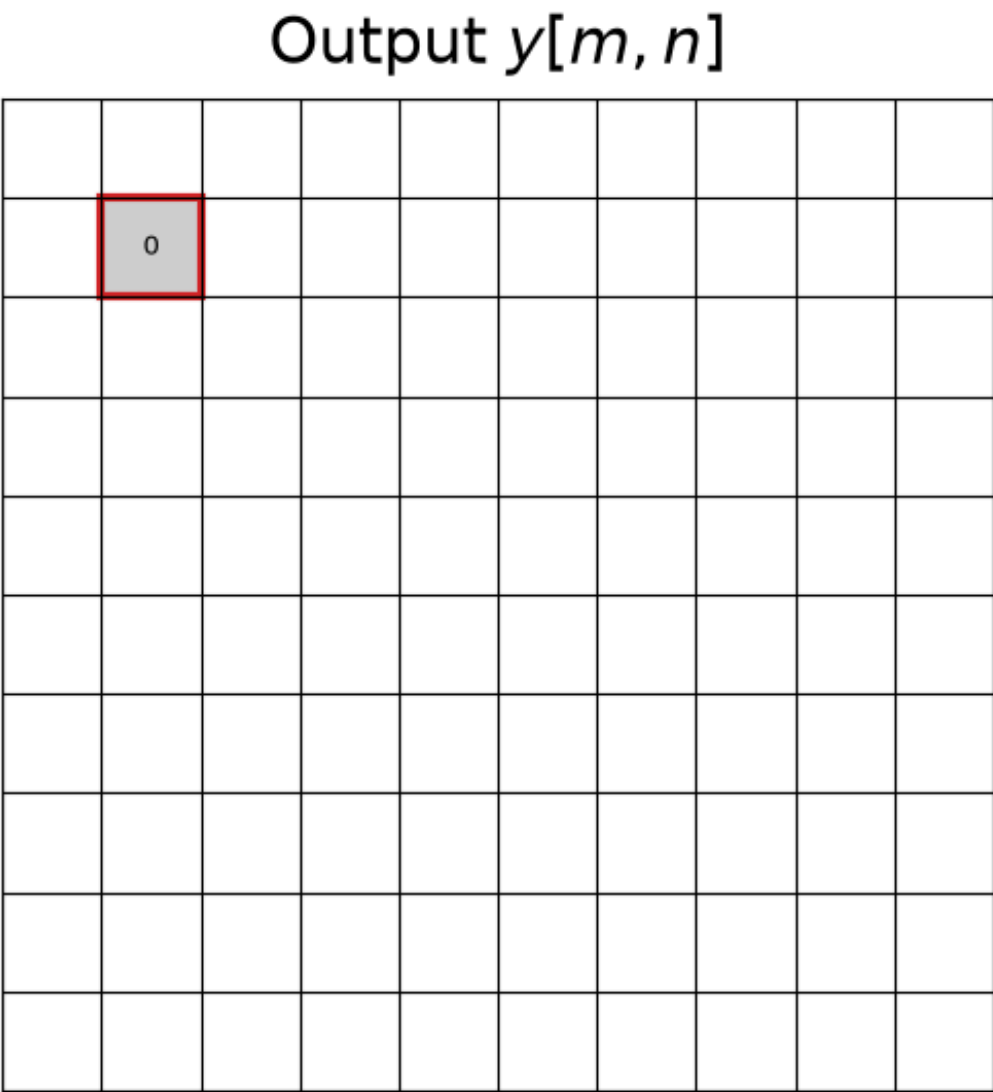


What is $y[4, 6]$?

Box Filter: Complete Output



Kernel $h[k, l]$



Position $(m, n) = (1, 1)$: $y[1, 1] = 0$

Box Filter: Complete Output

Filtered output $y[m, n]$

	0	10	20	30	30	30	20	10	
	0	20	40	60	60	60	40	20	
	0	30	60	90	90	90	60	30	
	0	30	50	80	80	90	60	30	
	0	30	50	80	80	90	60	30	
	0	20	30	50	50	60	40	20	
	10	20	30	30	30	30	20	10	
	10	10	10	0	0	0	0	0	

$$y[m, n] = \sum_{k=-1}^1 \sum_{l=-1}^1 h[k, l] I[m + k, n + l]$$

Box Filter: What Does It Do?

- Replaces each pixel by the average of its local neighborhood.
- Smooths small intensity variations and noise.
- Suppresses sharp edges and fine details.
- Why does the kernel sum to one?

$$h[k, l] = \frac{1}{9} \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

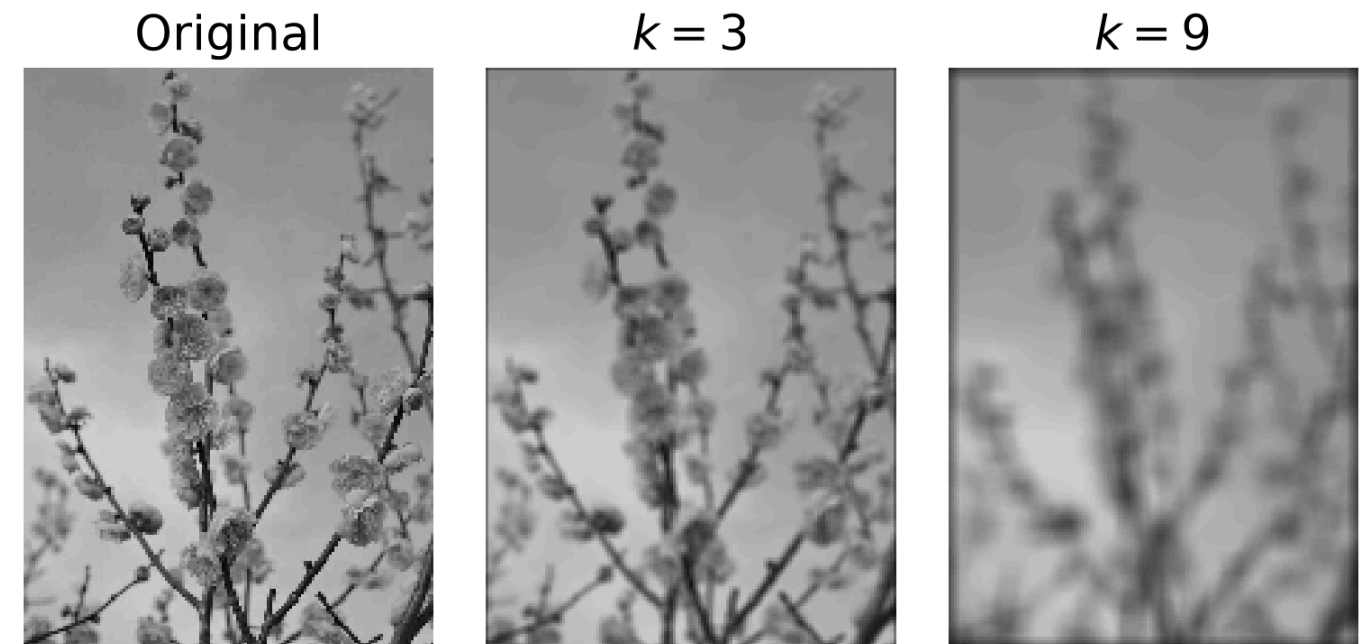
! Important

The coefficients sum to one, so the filter preserves constant image regions.

Smoothing with a Box Filter

- For a $k \times k$ box filter, $h[k, k] = 1/k^2$.
- A larger k produces stronger smoothing

```
1 import cv2
2
3 img_path = Path("assets/03_filtering/bloom_o
4 img_col = cv2.imread(str(img_path))
5
6 img_gray = cv2.cvtColor(img_col, cv2.COLOR_B
7 img_flt = img_gray.astype(np.float32) / 255.0
8 img = cv2.resize(img_flt, None, fx=0.2, fy=0
9
10 box3 = cv2.blur(img, (3, 3), borderType=cv2.B
11 box9 = cv2.blur(img, (9, 9), borderType=cv2.B
12
13 fig, axes = plt.subplots(1, 3, figsize=(6.5,
14 for ax, result, title in zip(
15     axes, [img, box3, box9], ["Original", "$
16     ax.imshow(result, cmap="gray", vmin=0, vi
17     ax.set_title(title) + ax.axis('off')
```

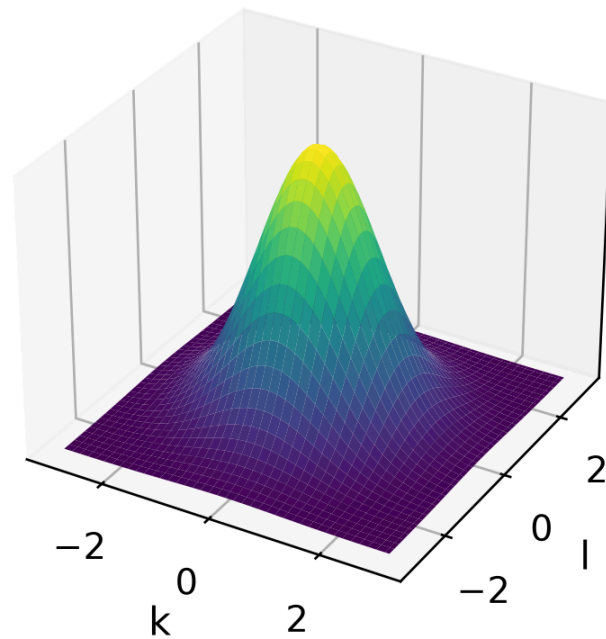


Gaussian Filter Kernel

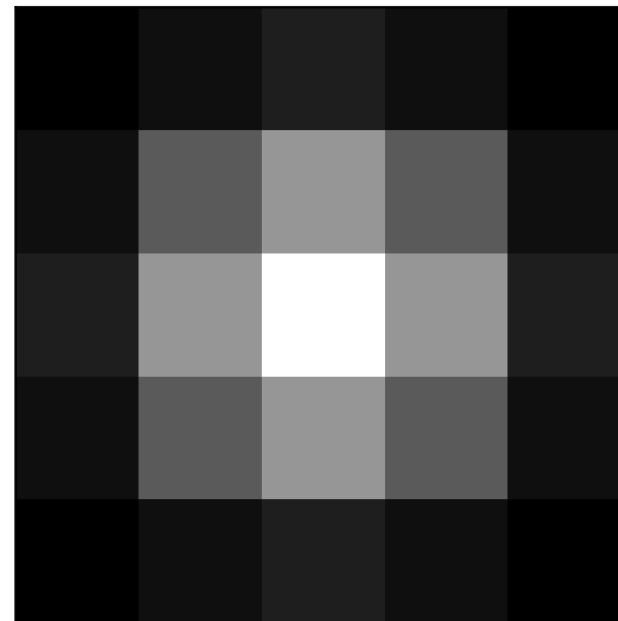
A Gaussian kernel gives the largest weight to the center pixel and smoothly decreases with distance:

$$h_{\sigma}[k, l] = \frac{1}{2\pi\sigma^2} \exp\left(-\frac{k^2 + l^2}{2\sigma^2}\right).$$

Continuous Gaussian



5 × 5 sampled kernel

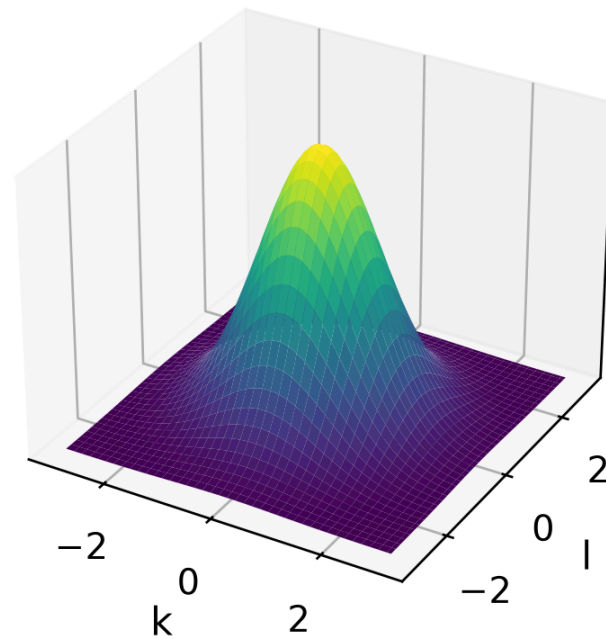


5 × 5 kernel, $\sigma = 1$

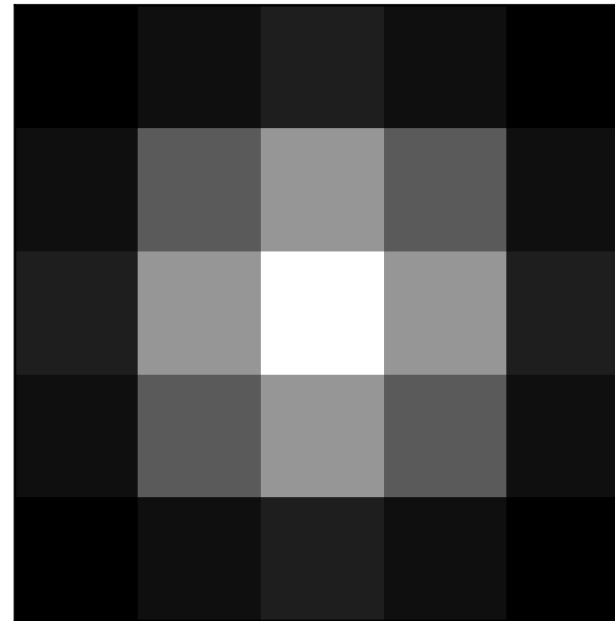
0.003	0.013	0.022	0.013	0.003
0.013	0.060	0.098	0.060	0.013
0.022	0.098	0.162	0.098	0.022
0.013	0.060	0.098	0.060	0.013
0.003	0.013	0.022	0.013	0.003

Gaussian Filter Kernel

Continuous Gaussian



5 × 5 sampled kernel



5 × 5 kernel, $\sigma = 1$

0.003	0.013	0.022	0.013	0.003
0.013	0.060	0.098	0.060	0.013
0.022	0.098	0.162	0.098	0.022
0.013	0.060	0.098	0.060	0.013
0.003	0.013	0.022	0.013	0.003

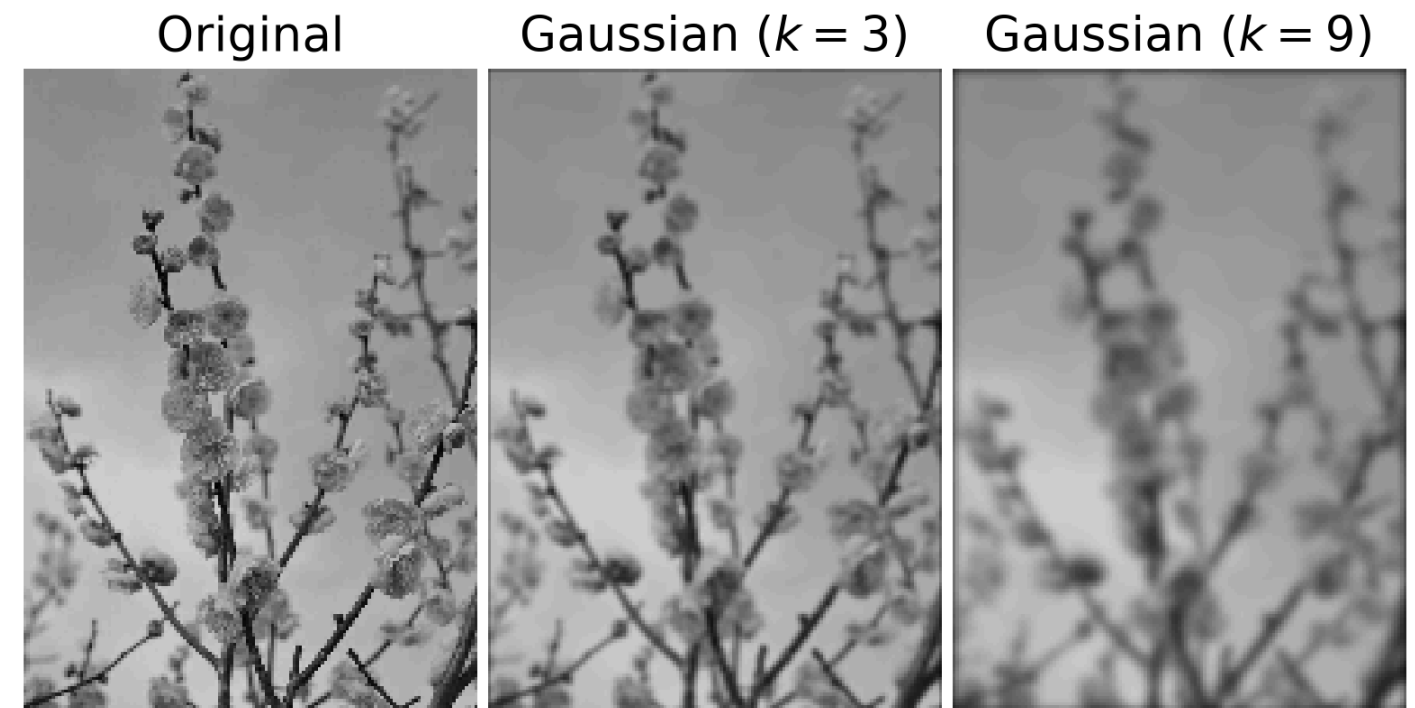
i Note

- Gaussian kernel weights sum to one (aside from negligible floating-point errors) – **why?**
- The continuous Gaussian integrates to one over the infinite plane. A truncated (5) sample does not automatically sum to one, so the explicit normalization line is necessary.
- Sampled kernel size should be large enough to include the meaningful support of the Gaussian—often several standard deviations on each side.

Smoothing with a Gaussian Filter

- A Gaussian filter smooths noises.

```
1 import cv2
2
3 gau3 = cv2.GaussianBlur(img, (3, 3), sigmaX=0, sigmaY=0)
4 gau9 = cv2.GaussianBlur(img, (9, 9), sigmaX=0, sigmaY=0)
5
6 fig, axes = plt.subplots(1, 3, figsize=(6.5, 4.5))
7 for ax, im, title in zip(axes, [img, gau3, gau9],
8                             ["Original", r"Gaussian ($k=3$)", r"Gaussian ($k=9$)"]):
9     ax.imshow(im, cmap="gray", vmin=0, vmax=255)
10    ax.set_title(title)
11    ax.axis("off")
12
13 plt.tight_layout(pad=0.3)
14 plt.show()
```



Box versus Gaussian Filter Smoothing

Original



Box filter ($k = 9$)



Gaussian filter ($k = 9$)



- The box filter weights every pixel in the window equally.
- The Gaussian filter changes weights smoothly, which produces fewer artifacts.

Linear Filter Properties

Linearity:

$$\text{filter}(I, h_1 + h_2) = \text{filter}(I, h_1) + \text{filter}(I, h_2)$$

h_1

1	0	1
0	1	0
1	0	1

$+ \quad h_2$

0	1	0
1	1	1
0	1	0

$= \quad h_1 + h_2$

1	1	1
1	2	1
1	1	1

h at position (m, n)

1	1	1	0
1	2	1	0
1	1	1	0
0	0	0	0

shift
→

h at position $(m + 1, n + 1)$

0	0	0	0
0	1	1	1
0	1	2	1
0	1	1	1

Shift/translation invariance:

$$\text{filter}(I, \text{shift}(h)) = \text{shift}(\text{filter}(I, h))$$

Separable Filters

A 2D kernel h is **separable** if

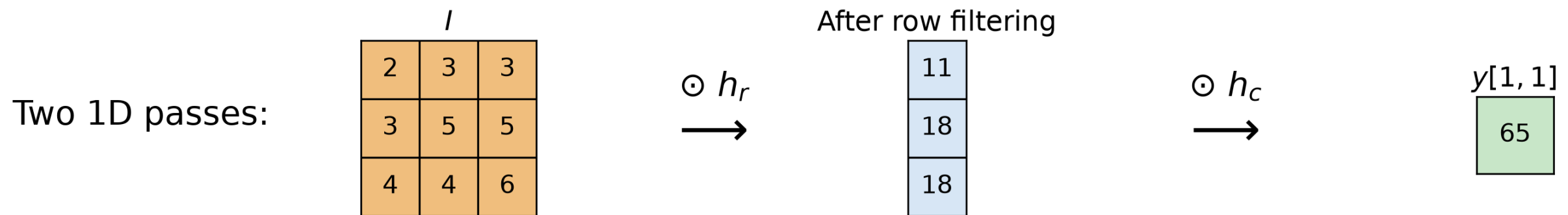
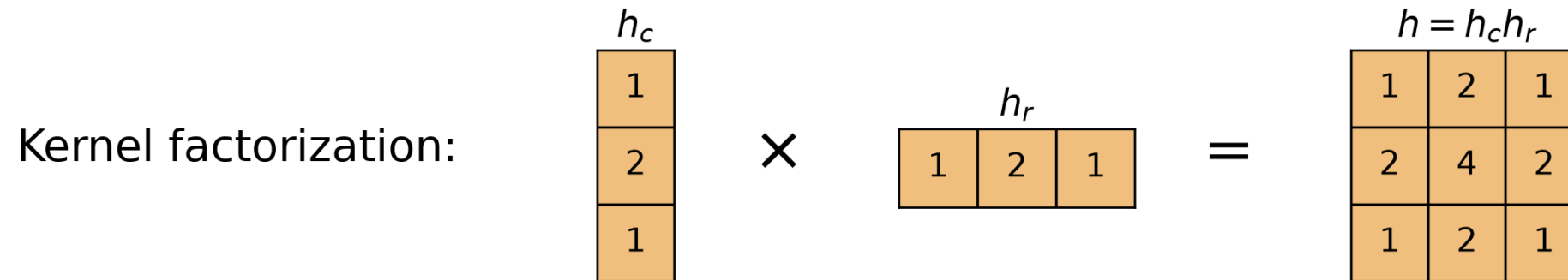
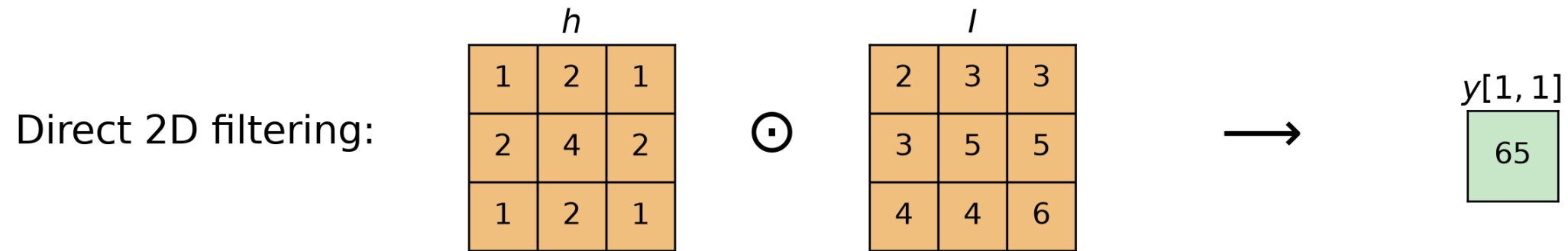
$$h = v u^T,$$

where v is a vertical 1D kernel and u^T is a horizontal 1D kernel.

For example,

$$\frac{1}{16} \begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix} = \frac{1}{4} \begin{bmatrix} 1 \\ 2 \\ 1 \end{bmatrix} \frac{1}{4} [1 \quad 2 \quad 1].$$

Separability Example



Separability Reduces Computation

For an $M \times N$ image and a $P \times Q$ kernel:

Method	Approximate multiply-adds
Full 2D filtering	$MNPQ$
Separable filtering	$MN(P + Q)$

For a square $K \times K$ kernel, the *per-pixel* cost drops from K^2 to $2K$.

Example: A separable 9×9 filter needs about $81/18 = 4.5$ times fewer multiply-adds than direct 2D filtering.

The Gaussian Filter Is Separable

Because

$$\frac{1}{2\pi\sigma^2} e\left(-\frac{x^2+y^2}{2\sigma^2}\right) = \left(\frac{1}{\sqrt{2\pi}\sigma} e\left(-\frac{x^2}{2\sigma^2}\right)\right) \left(\frac{1}{\sqrt{2\pi}\sigma} e\left(-\frac{y^2}{2\sigma^2}\right)\right) \implies$$
$$G_\sigma(x, y) = G_\sigma(x) G_\sigma(y),$$

we can:

1. Filter every row with a 1D Gaussian.
 2. Filter every column with the same 1D Gaussian.
- Repeated filtering with smaller Gaussian kernels can produce the same result as filtering once with a larger Gaussian kernel.
 - Applying a Gaussian filter with standard deviation σ twice is equivalent to applying one Gaussian filter with $\sigma_{\text{combined}} = \sqrt{2} \sigma$.
 - This produces the same result with substantially fewer computations.

Filtering: Correlation and Convolution

2D Correlation:

The kernel is applied without changing its orientation:

$$y[m, n] = \sum_{k, l} h[k, l] I[m+k, n+l].$$

OpenCV's `filter2D` computes correlation:

```
1 y_corr = cv2.filter2D(  
2     I, -1, h, borderType=cv2.BORDER_CONSTANT
```

2D Convolution:

The kernel indices are reversed:

$$y[m, n] = \sum_{k, l} h[k, l] I[m-k, n-l].$$

For convolution, **rotate the kernel by 180° first**:

```
1 h_rotated = cv2.flip(h, -1)  
2  
3 y_conv = cv2.filter2D(  
4     I, -1, h_rotated, borderType=cv2.BORDER_
```

- Convolution is **equivalent to correlation with a 180° rotated kernel**:

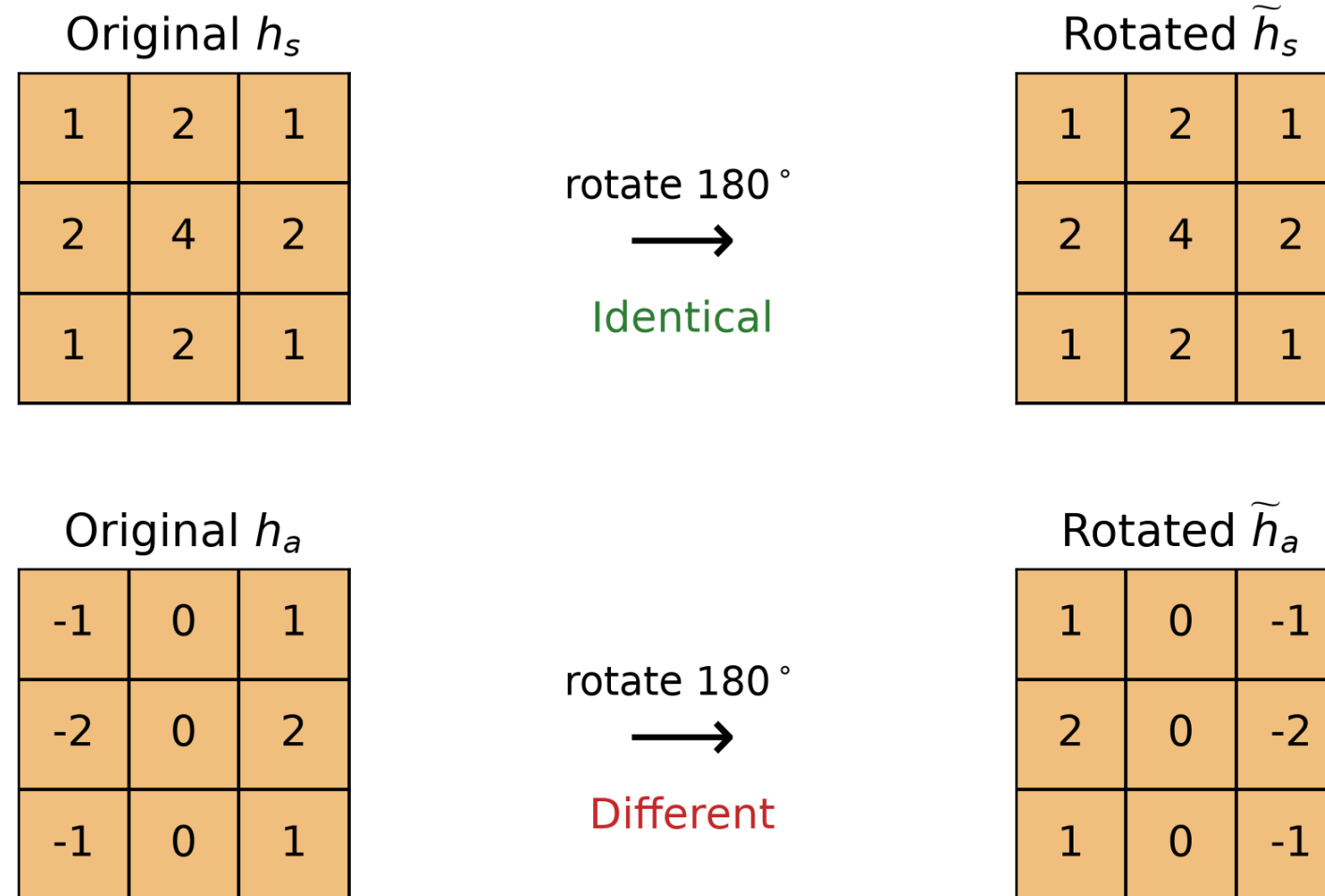
$$\tilde{h}[k, l] = h[-k, -l].$$

- Correlation and convolution are **identical** when the kernel has **180° rotational symmetry**:

$$h[k, l] = h[-k, -l].$$

Correlation and Convolution Differ for Asymmetric Kernels

A kernel rotated by 180° : $\tilde{h}[k, l] = h[-k, -l]$.



- Since $h_s = \tilde{h}_s$, correlation and convolution produce the same result.
- But $h_a \neq \tilde{h}_a$, so correlation and convolution produce different responses.

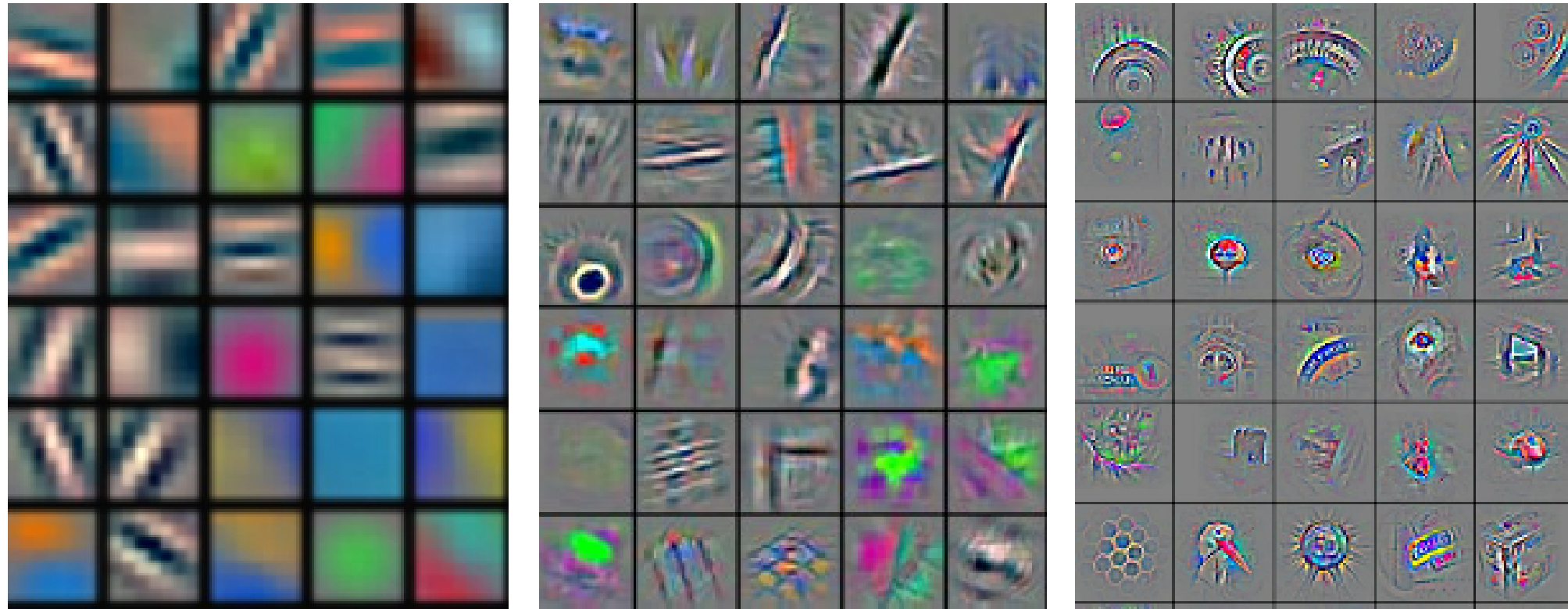
Convolution (*) Properties

- **Commutative:** $I * h = h * I$
 - Mathematically, the input and kernel can be interchanged.
- **Associative:** $(I * h_1) * h_2 = I * (h_1 * h_2)$
 - A sequence of filters can be combined into a single kernel.
 - This can reduce the number of filtering operations.
 - Note that **correlation is not associative** in general.
- **Distributive:** $I * (h_1 + h_2) = I * h_1 + I * h_2$
- **Scalar factor out:** $(\alpha I) * h = \alpha(I * h)$
- **Identity:** $I * \delta = I$, where δ is the unit impulse
 - The identity kernel is the unit impulse:
$$\delta[m, n] = \begin{cases} 1, & m = 0, n = 0, \\ 0, & \text{otherwise.} \end{cases}$$
 - Convolving with the unit impulse leaves the image unchanged

⚠ Important

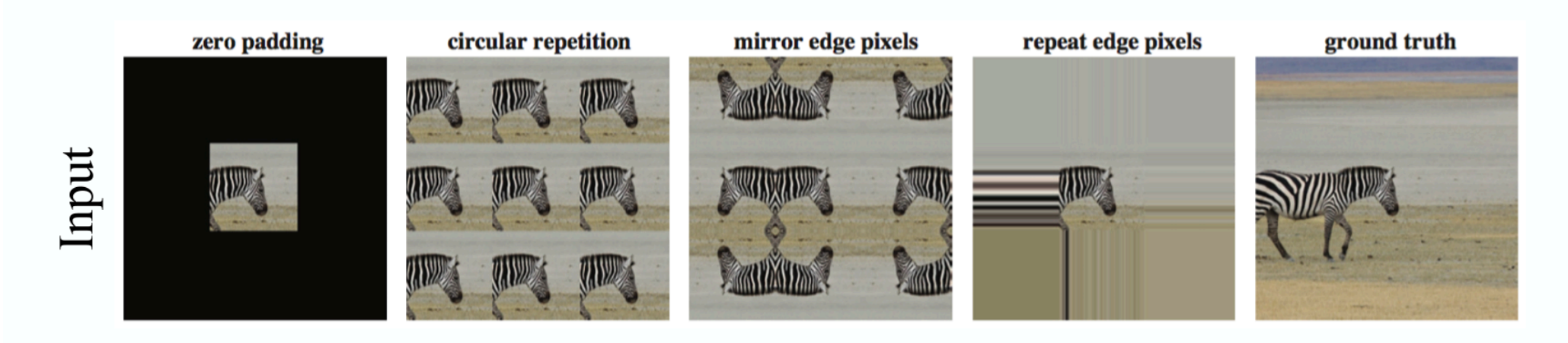
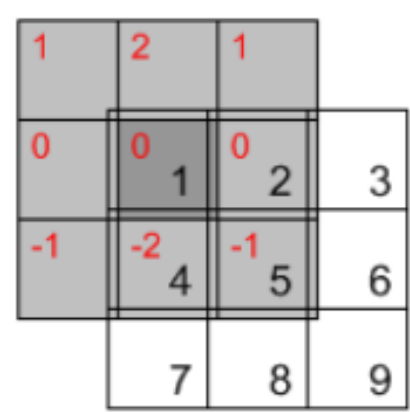
- Finite images and boundary rules can make practical implementations appear to violate these identities near the edges.
- Convolution filters are both linear and shift-invariant – **why?**
- Every liner filter can be represented as convolution with some kernel – **why?**

“Convolution” Layers Usually Compute Correlation



- Convolutional neural networks learn their kernel weights from data rather than using predefined filters.
- Despite the name, most deep-learning libraries apply learned kernels without rotating them, which is, correlation.
- This does not reduce the network’s expressive power since the kernel is learned, and the network can learn either a kernel or its rotated version.

Boundary Handling



Near an image boundary, part of the kernel falls outside the available data. Need to extrapolate (padding) to compute the output edge pixels.

Strategy	Assumption outside the image	Common consequence
Constant / zero	Missing pixels have a fixed value	Dark or bright border artifacts
Replicate	Repeat the closest boundary pixel	Flat extension at the edge
Reflect	Mirror the image across the boundary	Often smoothest for natural images
Wrap	Opposite edges are adjacent	Appropriate for periodic data
Valid only	Skip incomplete neighborhoods	Smaller output image

i Note

Two implementations using the same kernel can disagree near the boundary if their padding modes differ.

Common Output Modes

For an $M \times N$ image and a $P \times Q$ kernel:

Mode	Output size	Interpretation
<code>full</code>	$(M + P - 1) \times (N + Q - 1)$	Every partial overlap, with padding
<code>same</code>	$M \times N$	Center portion matching the input size
<code>valid</code>	$(M - P + 1) \times (N - Q + 1)$	Only complete overlaps; no padding

Example: convolving a 275×175 image with another 275×175 image produces:

- `full`: 549×349
- `same`: 275×175
- `valid`: 1×1

 Note

Library defaults differ. Specify the mode and boundary behavior explicitly when reproducibility matters.

Three Common Output Modes

```
1 import cv2
2
3 x = cv2.imread("assets/03_filtering/colorado")
4 x = cv2.resize(x, (200,120), interpolation=cv2.INTER_LINEAR)
5 h, w = x.shape
6
7 p = cv2.copyMakeBorder(x, h-1,0,w-1,0, cv2.BORDER_REPLICATE)
8 full = cv2.filter2D(p, cv2.CV_32F, x, anchor=cv2.CV_32F)
9 same = cv2.filter2D(x, cv2.CV_32F, x, borderMode=cv2.BORDER_REPLICATE)
10 valid = full[h-1:h, w-1:w]
11
12 ims, names = [x,full,same,valid], ["Input","Full","Same","Valid"]
13 H, W = full.shape
14 fig, ax = plt.subplots(2,2,figsize=(6.5,5))
15
16 for a, im, name in zip(ax.flat, ims, names):
17     ih, iw = im.shape
18     a.imshow(im, cmap="gray", extent=(0,iw,ih,0))
19     a.set(xlim=(0,W), ylim=(H,0), title=fr"{name}")
20     a.axis("off")
21
22 plt.tight_layout(pad=.4)
```

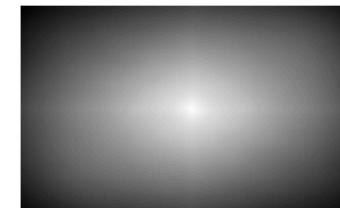
Input: 200 × 120



full: 399 × 239



same: 200 × 120



valid: 1 × 1



Practice with Linear Filters

Predict the effect of each kernel before viewing the following examples.

Input



Kernel

0	0	0
0	1	0
0	0	0

Kernel

0	0	0
0	0	1
0	0	0

Kernel

1	0	-1
2	0	-2
1	0	-1

Kernel

$-1/9$	$-1/9$	$-1/9$
$-1/9$	$8/9$	$-1/9$
$-1/9$	$-1/9$	$-1/9$

Practice:

What will this kernel do to the image?

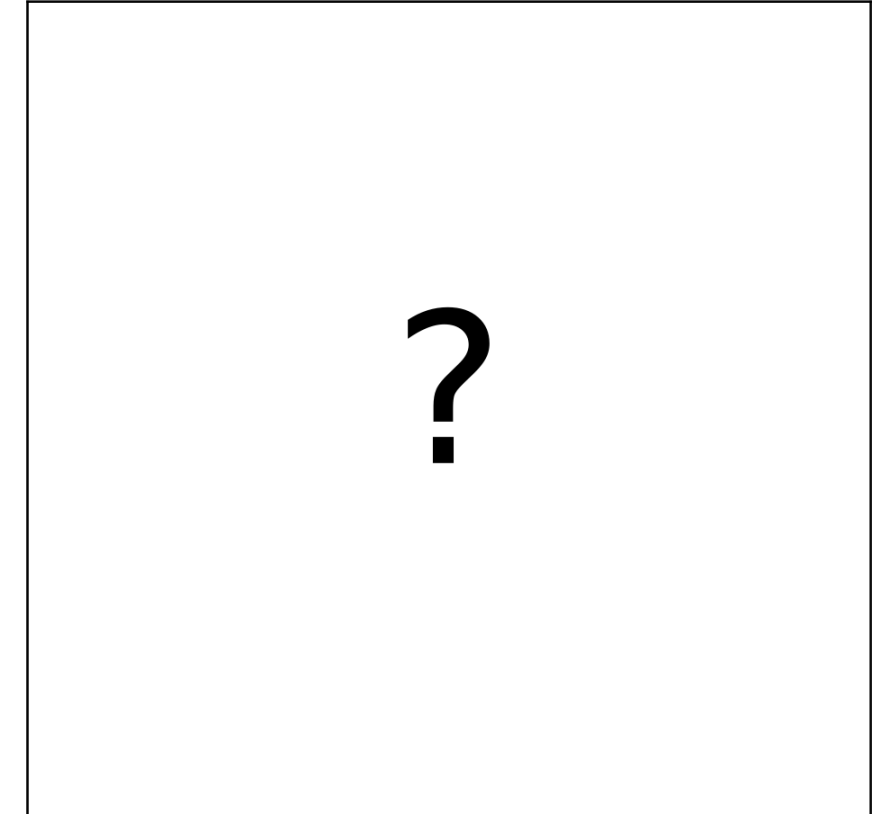
Input



Kernel

0	0	0
0	1	0
0	0	0

Predict the output



Identity Kernel Leaves the Image Unchanged

The center coefficient copies the current pixel:

$$y[m, n] = I[m, n].$$

Input



Kernel

0	0	0
0	1	0
0	0	0

Output: unchanged



Practice:

What will happen when the kernel selects the pixel immediately to the right?

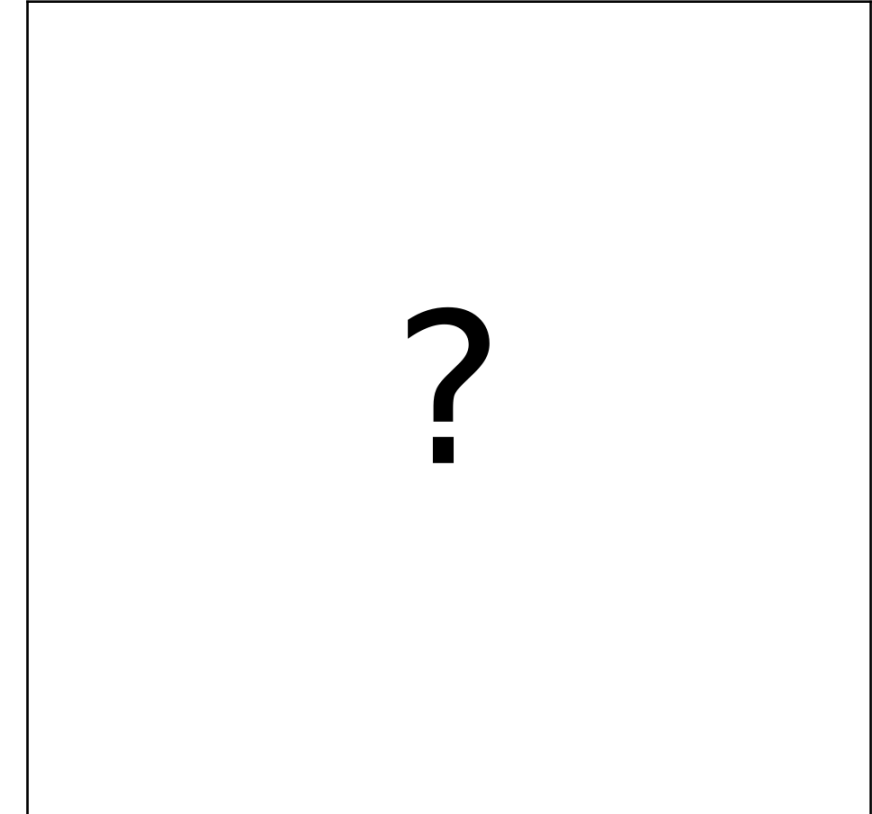
Input



Kernel

0	0	0
0	0	1
0	0	0

Predict the output



Shift Kernel Copies a Neighboring Pixel

For correlation, the kernel produces

$$y[m, n] = I[m, n + 1],$$

so the image content shifts left by one pixel.

Input



Kernel

0	0	0
0	0	1
0	0	0

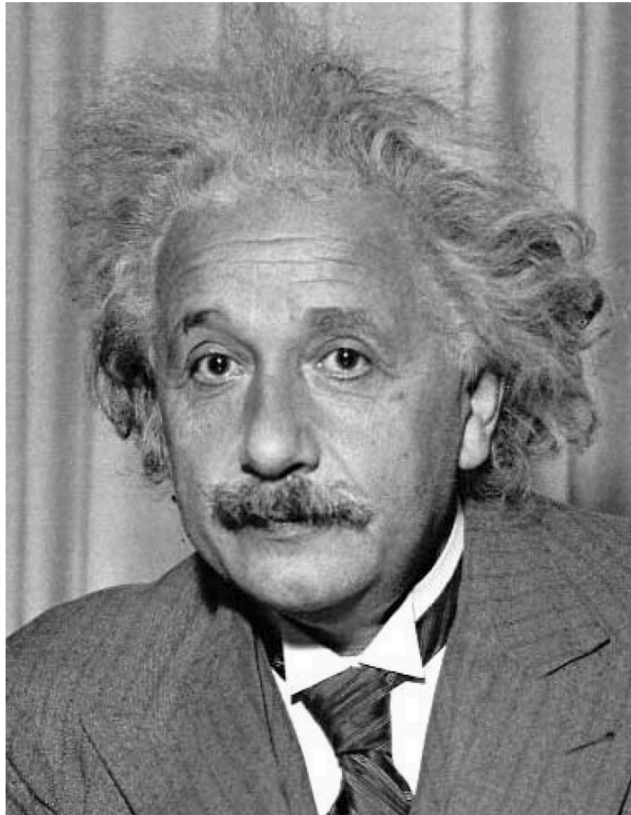
Output: shifted left



Practice:

Examine the kernel and predict which image structures will produce the strongest response.

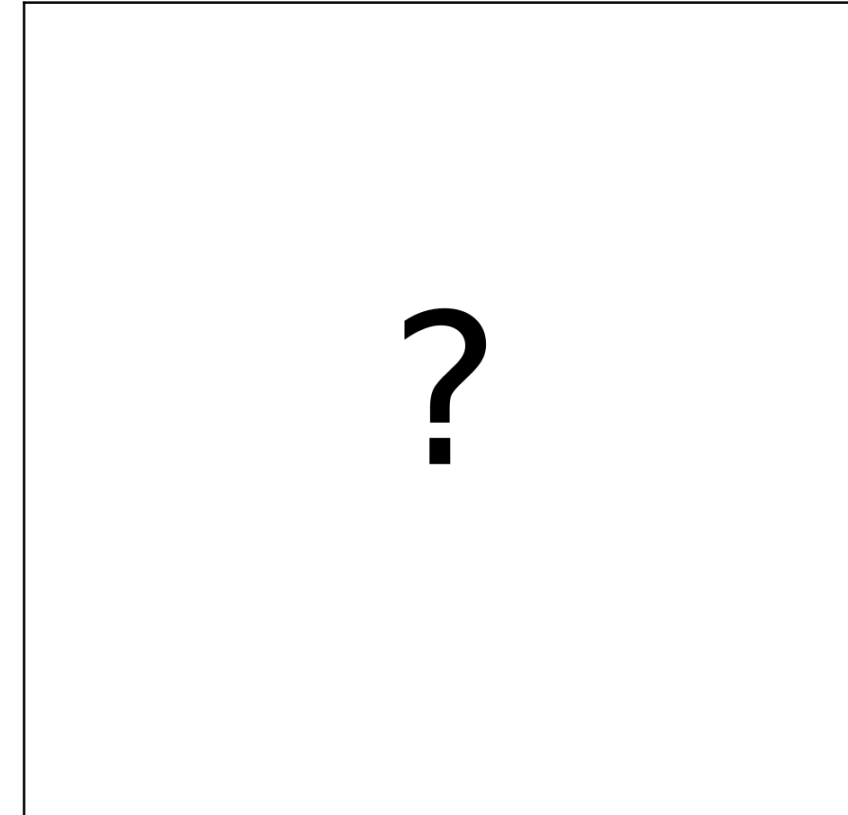
Input



Kernel

1	0	-1
2	0	-2
1	0	-1

Predict the output

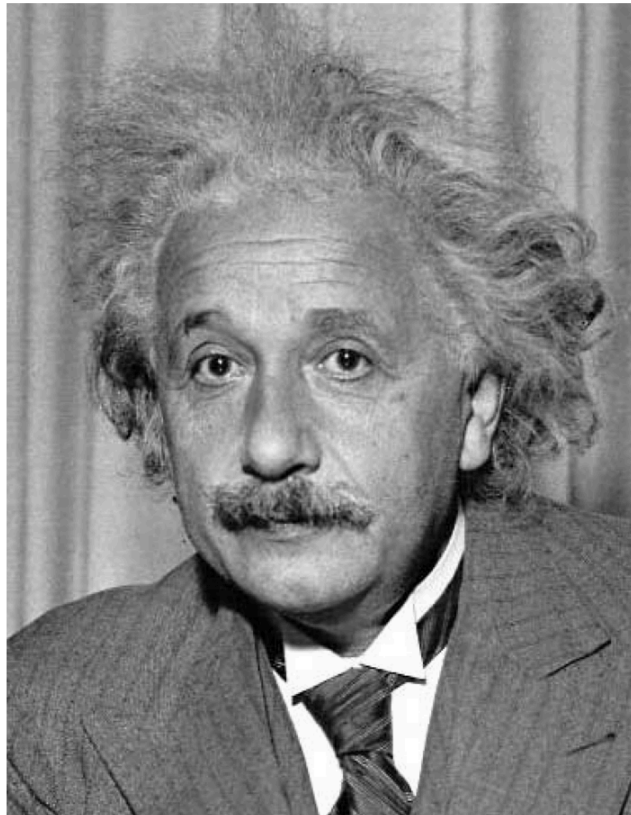


- Will the kernel respond to horizontal or vertical edges?
- Which regions of the image will have a response near zero?

Sobel x : Detecting Vertical Edges

This kernel responds strongly to vertical edges.

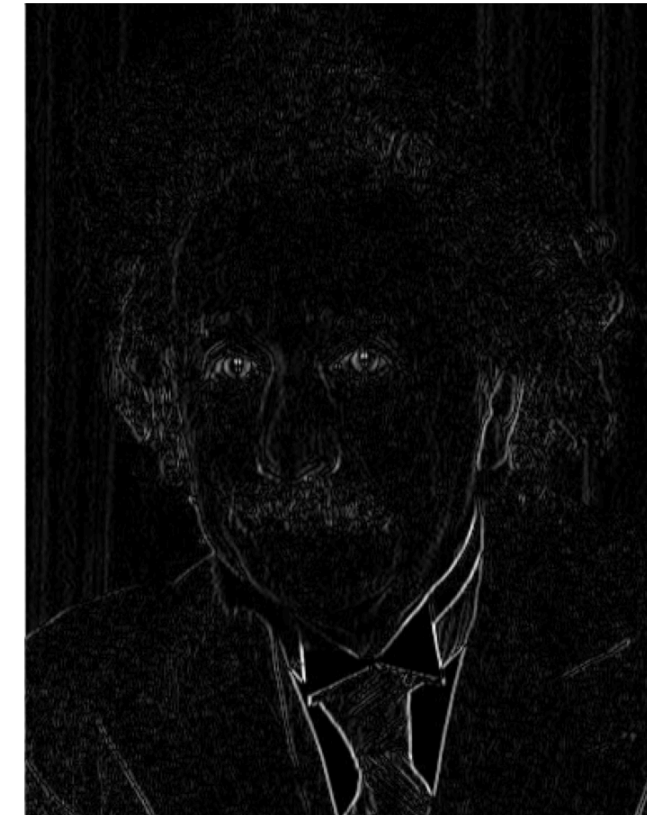
Input



Kernel

1	0	-1
2	0	-2
1	0	-1

Absolute response

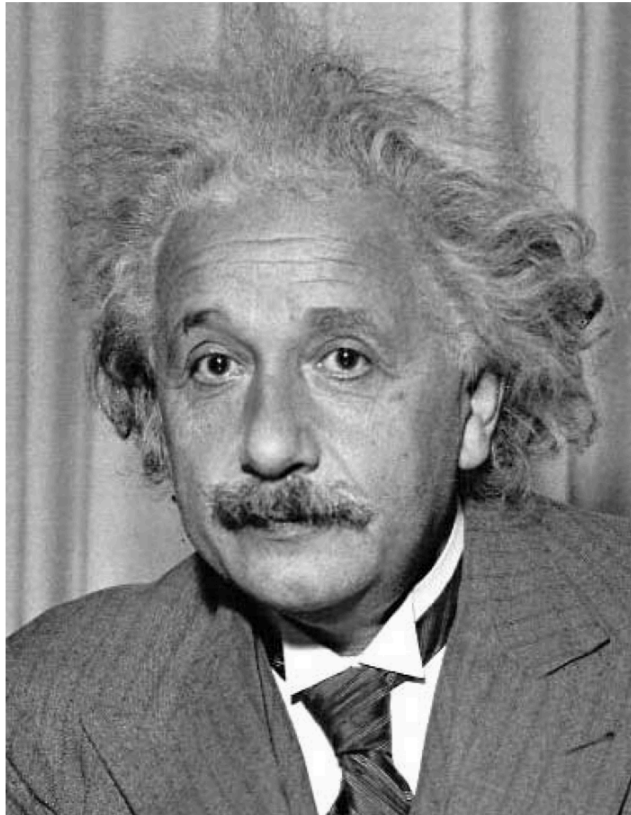


The absolute value displays both dark-to-light and light-to-dark transitions as bright edges.

Practice:

Examine the kernel and predict which image structures will produce the strongest response.

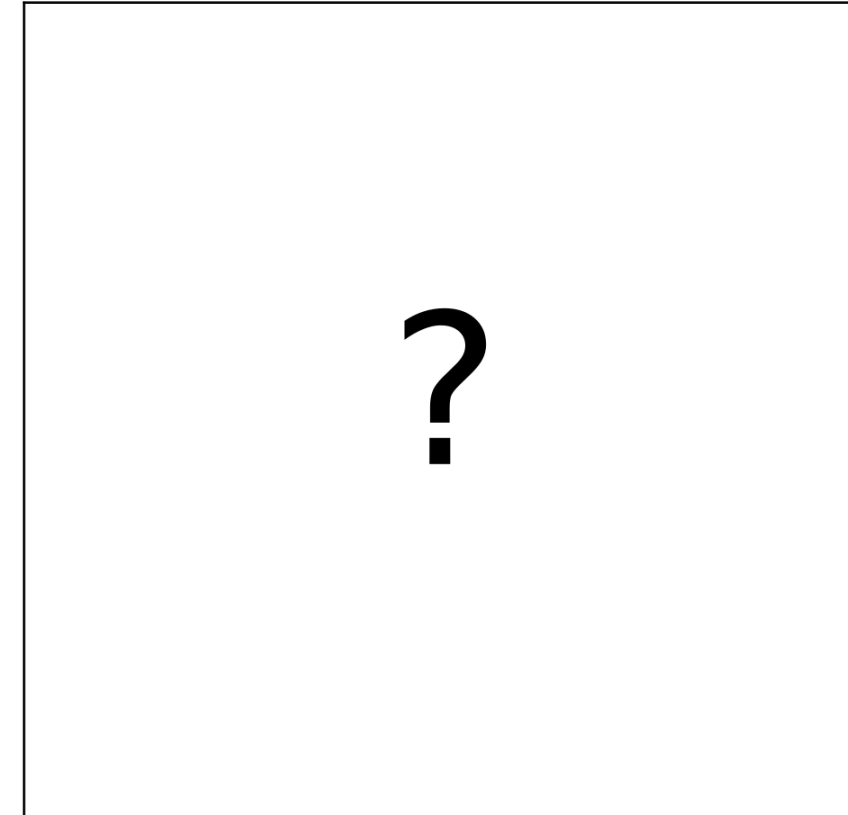
Input



Kernel

1	2	1
0	0	0
-1	-2	-1

Predict the output

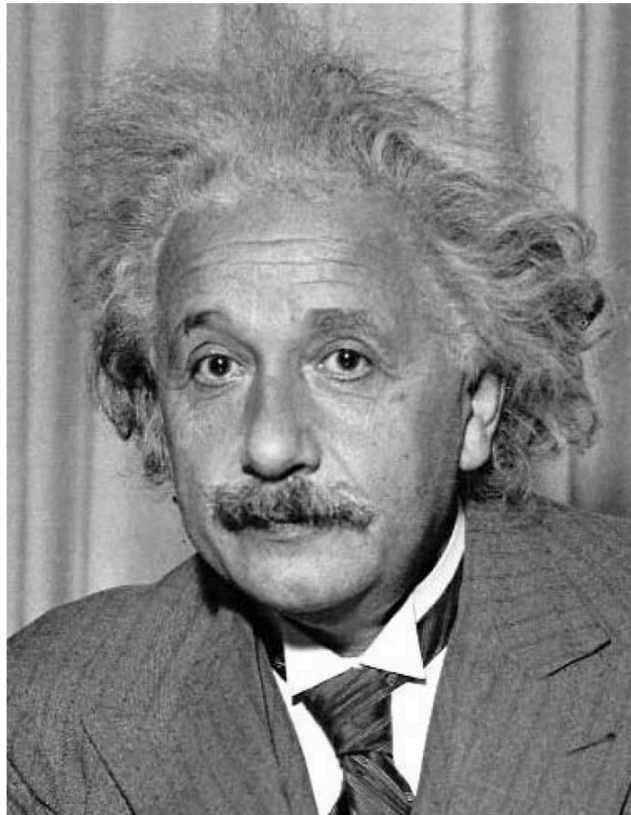


- Will the kernel respond to horizontal or vertical edges?
- Which regions of the image will have a response near zero?

Sobel y : Detecting Horizontal Edges

This kernel responds strongly to horizontal edges.

Input



Kernel

1	2	1
0	0	0
-1	-2	-1

Absolute response



The sign of the original response indicates the direction of the intensity transition.

Practice: Subtracting the Local Average

What remains after subtracting the local 3×3 average from the center pixel?

$$h_{\text{detail}} = \delta - h_{\text{box}}.$$

Input



Kernel

-1/9	-1/9	-1/9
-1/9	8/9	-1/9
-1/9	-1/9	-1/9

Predict the output

?

Local-Detail Kernel Extracts High Frequencies

The kernel removes smooth content and preserves rapid intensity changes:

$$y = I - \text{boxblur}(I).$$

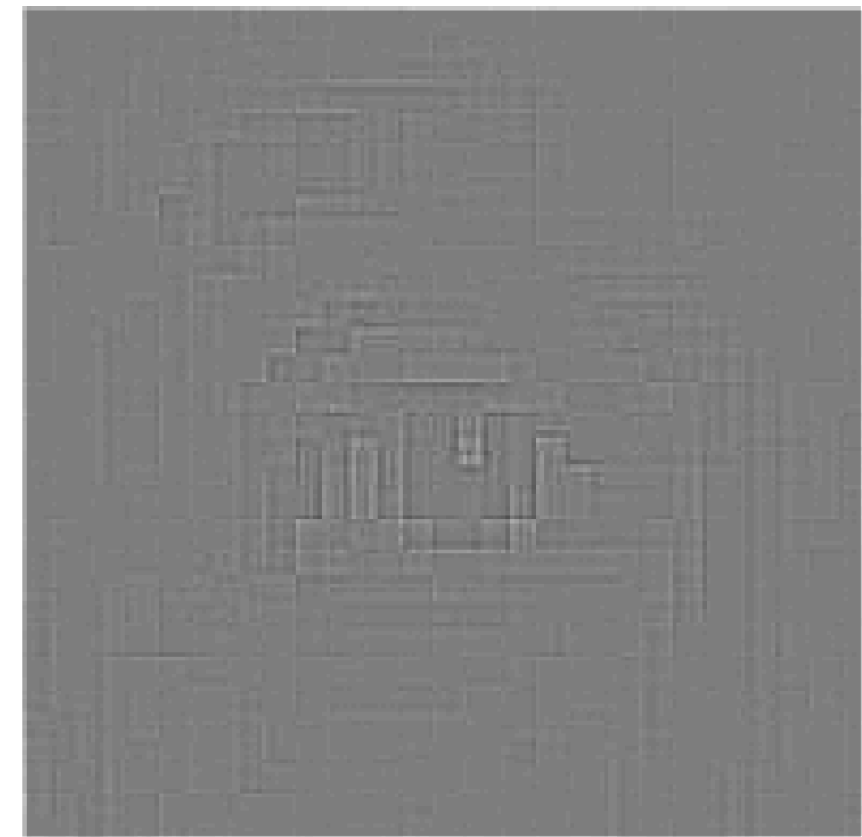
Input



Kernel

-1/9	-1/9	-1/9
-1/9	8/9	-1/9
-1/9	-1/9	-1/9

Signed local detail



Mid-gray represents weak response; brighter or darker values represent strong filter response.

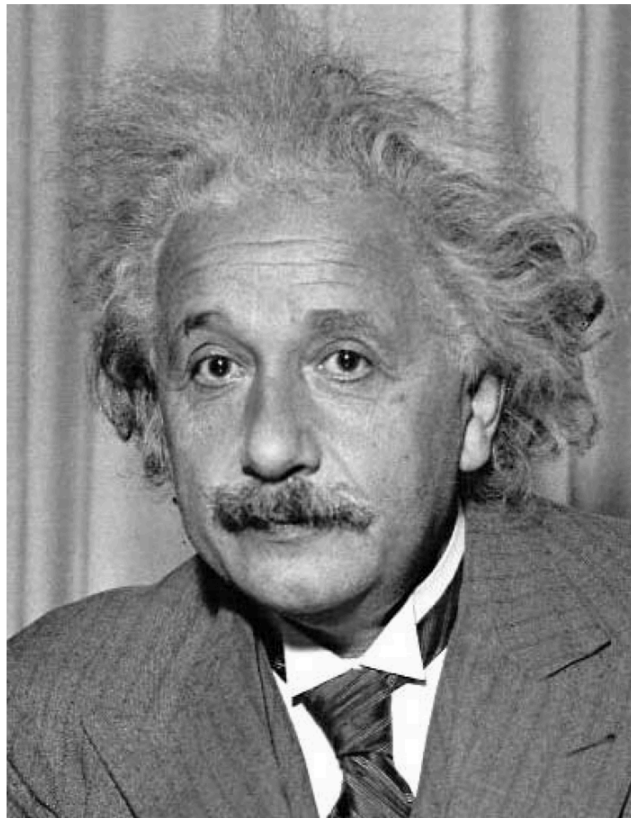
Adding Local Detail Sharpens the Image

Sharpening adds the extracted high-frequency detail back to the original image:

$$h_{\text{sharp}} = \delta + (\delta - h_{\text{box}})$$

$$y = I + (I - \text{boxblur}(I))$$

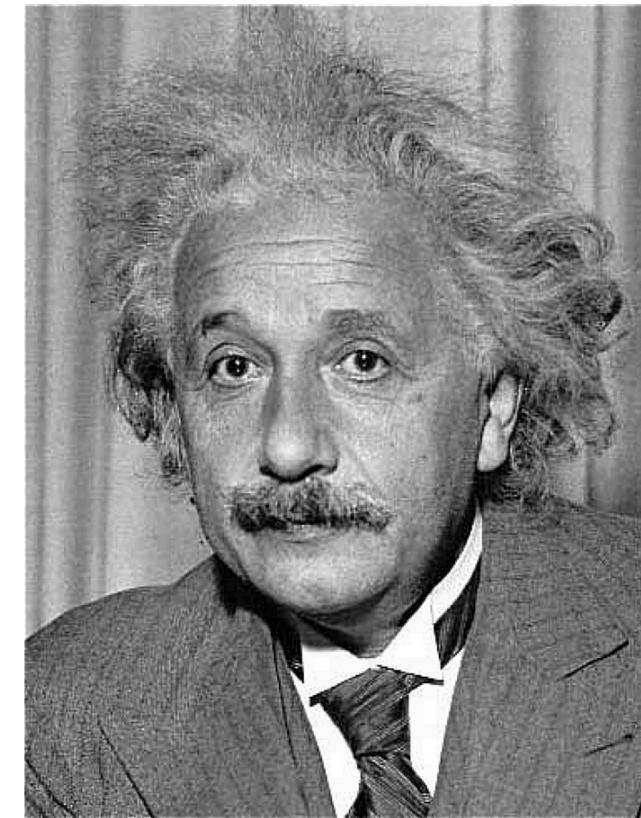
Input



Kernel

-1/9	-1/9	-1/9
-1/9	17/9	-1/9
-1/9	-1/9	-1/9

Sharpened output



Filter Responses Can Be Negative

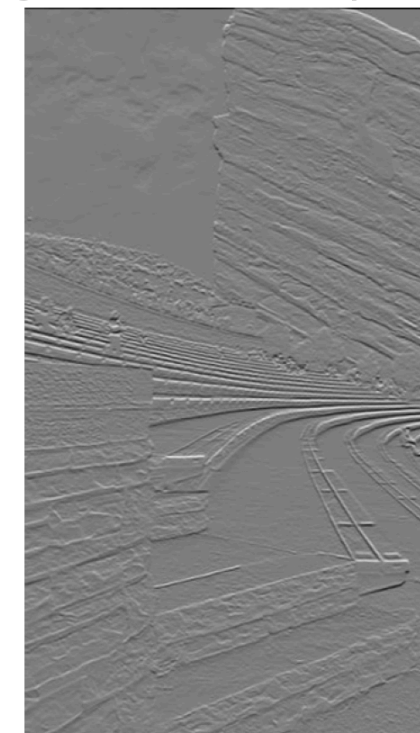
- Consider the horizontal-edge Sobel kernel h .
- A Sobel filter estimates an image derivative, so output is not restricted to the range $[0, 255]$ or $[0, 1]$.
- Positive and negative values represent opposite edge polarities.
- Values near zero indicate little intensity change.

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

```
1 img = cv2.imread("assets/03_filtering/red-rocks.jpg")
2 h_y = np.array([[1, 2, 1],
3                 [0, 0, 0],
4                 [-1, -2, -1]], dtype=np.float32)
5 img_out = cv2.filter2D(img, cv2.CV_32F, h_y,
6 limit = np.abs(img_out).max())
7
8 fig, ax = plt.subplots(1, 2, figsize=(6.5, 4))
9 ax[0].imshow(img, cmap="gray"); ax[0].set_title("Input")
10 ax[1].imshow(img_out, cmap="gray", vmin=-limit, vmax=limit)
11 ax[1].set_title("Signed filter response")
12 for a in ax: a.axis("off")
13 plt.tight_layout(); plt.show()
```



Signed filter response



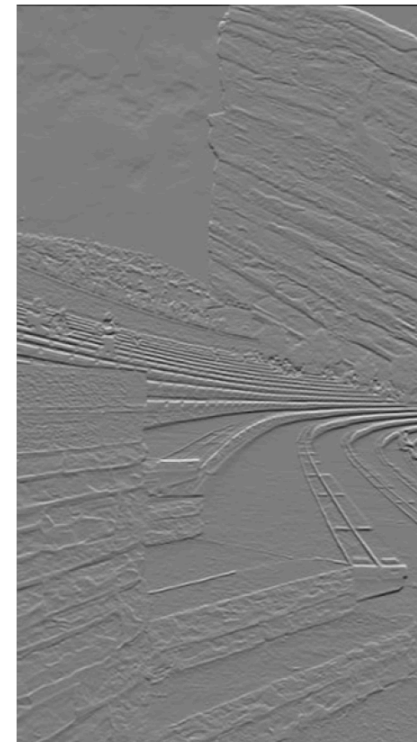
Visualizing Signed Filter Responses

The signed filter response can be transformed differently depending on what we want to visualize.

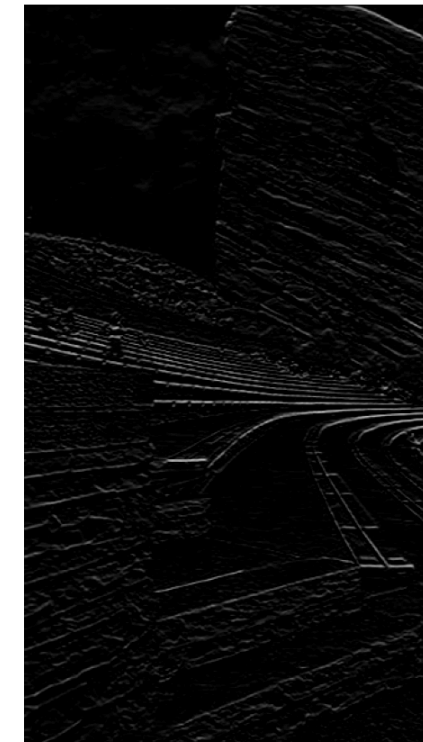
- In the signed response, zero is centered at gray.
- Clipping negative values removes one edge polarity.
- The absolute response shows edge strength but discards polarity.

```
1 limit = max(float(np.abs(img_out).max()), 1)
2
3 images = [img_out, np.clip(img_out, 0, None)]
4
5 titles = ["Signed", "Neg clipped", "Absolute"]
6 fig, ax = plt.subplots(1, 3, figsize=(6.5, 4))
7 for a, image, title in zip(ax, images, titles):
8     signed = title == "Signed"
9     a.imshow(image, cmap="gray", vmin=-limit if signed else 0)
10    a.set_title(title); a.axis("off")
11 plt.tight_layout(); plt.show()
```

Signed



Neg clipped



Absolute val



Practice: Match Each Filter to Its Output

Select E, F, G, H, or I for each filter. One candidate is not used.

1. $A * B = ?$ 2. $B * C = ?$ 3. $A * D = ?$ 4. $A * A = ?$

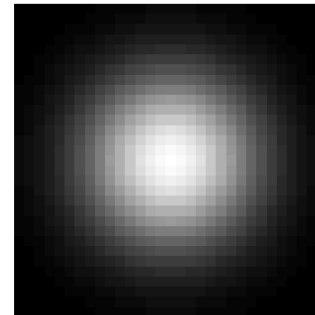
A: input image



B: derivative kernel



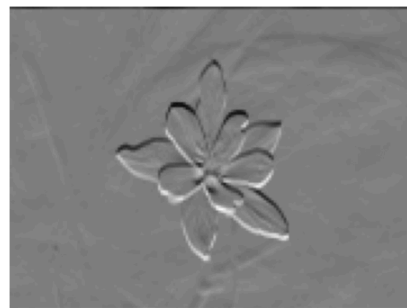
C: Gaussian kernel



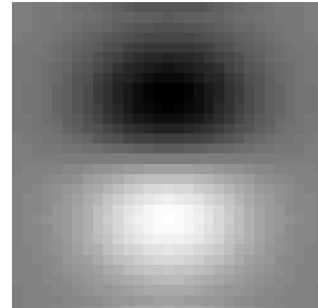
D: shifted impulse



E



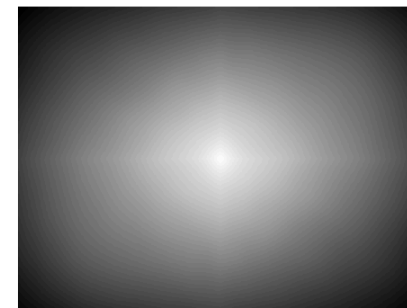
F



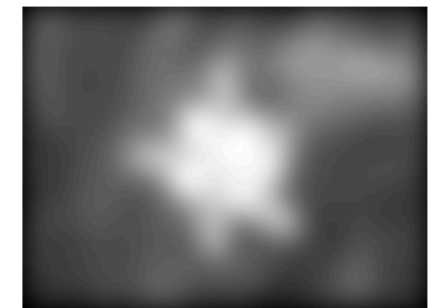
G



H

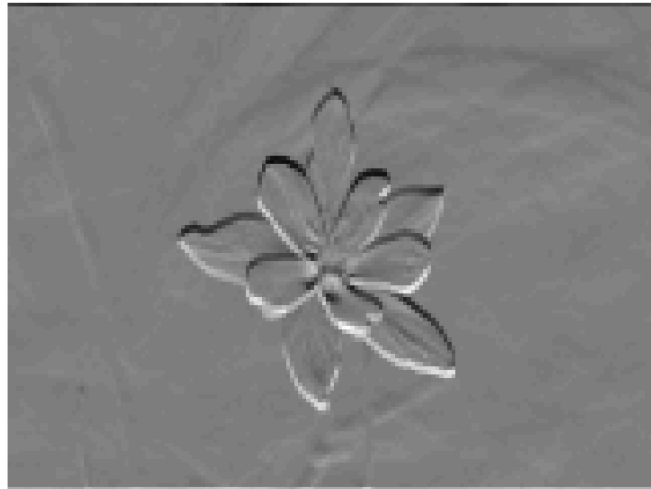


I

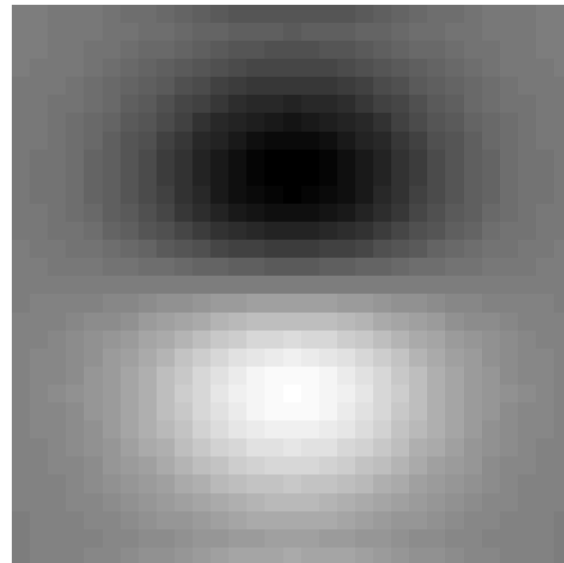


Practice: Answers

$$A * B = E$$



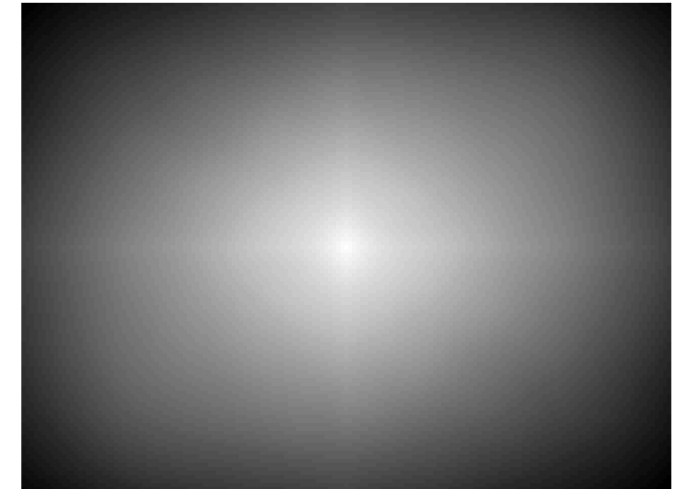
$$B * C = F$$



$$A * D = G$$



$$A * A = H$$



- $A * B = E$: the derivative kernel produces an edge response.
- $B * C = F$: smoothing the derivative kernel produces a derivative-of-Gaussian kernel.
- $A * D = G$: correlation with the shifted impulse shifts the image.
- $A * A = H$: correlation the image with itself produces a broad self-correlation response.

The unused candidate is $I = A * C$, the Gaussian-smoothed image.

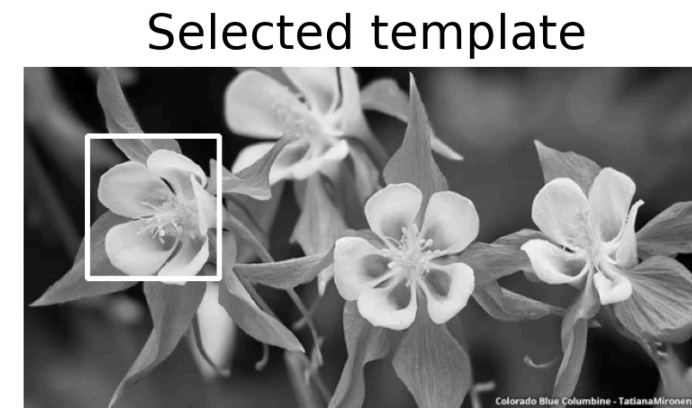
Correlation as Template Matching

- Use an image patch as the filter kernel and correlate it across the image:

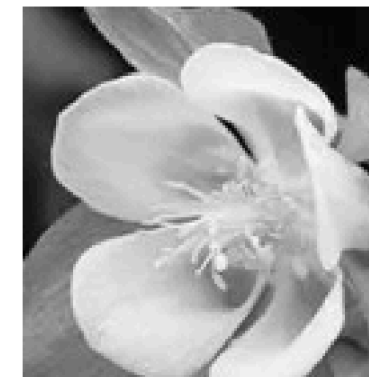
$$y[m, n] = \sum_k \sum_l h[k, l] I[m + k, n + l]$$

- The largest response is the location that best matches the template.

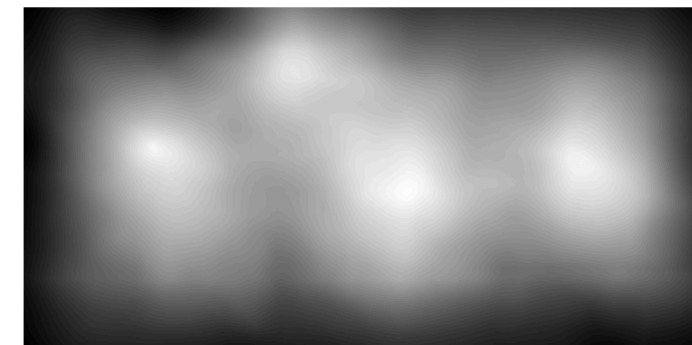
```
1 import cv2
2
3 path = "assets/03_filtering/colorado-flowers
4 box = (60, 65, 125, 135) # x0, y0, width, height
5
6 I = cv2.imread(path, cv2.IMREAD_GRAYSCALE).astype(np.float32)
7 x0, y0, w, ht = box
8 h = I[y0:y0+ht, x0:x0+w]
9
10 y = cv2.filter2D(I, cv2.CV_32F, h, borderType=cv2.BORDER_REPLICATE)
11 _, score, _, (px, py) = cv2.minMaxLoc(y)
12
13 selected = I.copy(); cv2.rectangle(selected, (x0, y0), (x0+w, y0+ht),
14 match = I.copy()
```



Template h



Correlation output y



Largest response



Brightness Can Dominate Correlation

- Raw correlation computes a dot product, so brighter regions can produce larger responses.
- Subtract the template mean: $f_0 = f - \bar{f}$
- Now positive and negative values describe the **pattern relative to its local brightness**.

```
1 h0 = h - h.mean()
2 y = cv2.filter2D(I, cv2.CV_32F, h0, borderType=cv2.BORDER_REPLICATE)
3
4 _, score, _, (px,py) = cv2.minMaxLoc(y)
5 match = I.copy()
6 cv2.rectangle(match, (px-w//2,py-h//2), (px+w//2,py+h//2), color=(0,0,255), thickness=2)
7
8 fig, ax = plt.subplots(2,2,figsize=(6.5,4.5))
9 ax[0,0].imshow(h0,cmap="gray"); ax[0,0].set_title("Zero-mean h")
10 ax[1,0].imshow(y,cmap="gray"); ax[1,0].plot(px,py, 'r'); ax[1,0].set_title("Correlation output y")
11 ax[1,0].set_title("Correlation output $y$")
12 ax[1,1].imshow(match,cmap="gray"); ax[1,1].set_title("Match image")
13 for a in ax.flat: a.axis("off")
14 plt.tight_layout(pad=.3); plt.show()
```

Zero-mean h



Correlation output y



Peak: (122, 132)



Use Correlation, Not Convolution, for Template Matching

- **Correlation** compares the template in its original orientation.
- **Convolution** compares against a template rotated by 180° .
- For **template matching**, use **correlation** so the template orientation is preserved.

```
1 kernels = [h0, cv2.flip(h0, -1)]
2 names = ["Correlation", "Convolution"]
3
4 fig, ax = plt.subplots(2, 2, figsize=(6.5, 4.4))
5
6 for r, (hk, name) in enumerate(zip(kernels, names)):
7     y = cv2.filter2D(I, cv2.CV_32F, hk, borderType=cv2.BORDER_REPLICATE)
8     _, score, _, (px, py) = cv2.minMaxLoc(y)
9
10    match = I.copy()
11    cv2.rectangle(match, (px-w//2, py-h//2), (px+w//2, py+h//2), (0, 255, 0, 2))
12
13    ax[r, 0].imshow(hk, cmap="gray")
14    ax[r, 1].imshow(match, cmap="gray")
15    ax[r, 0].set_title(f"{name} kernel")
16    ax[r, 1].set_title(f"Peak response = {score}")
17
```

Correlation kernel



Convolution kernel



Peak response = 982.2



Peak response = 583.8



Nonlinear Filters

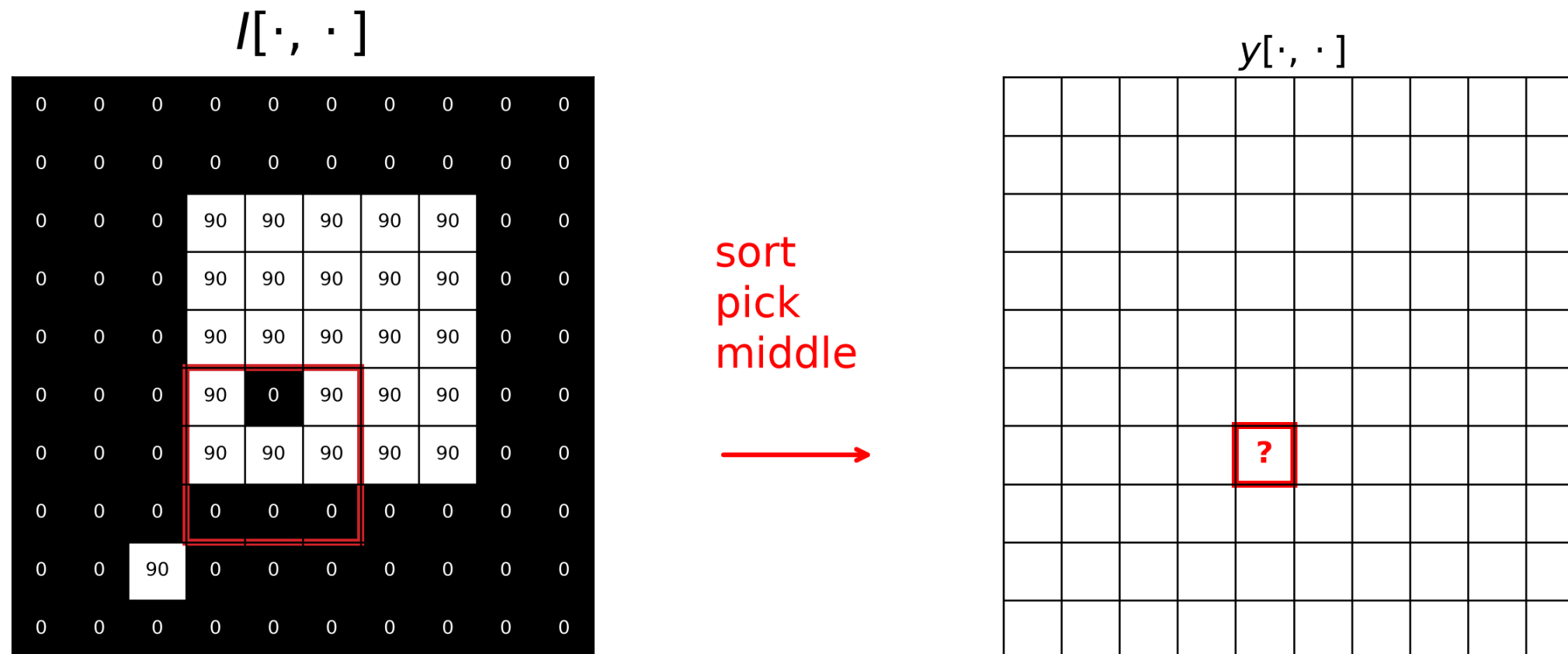
$$y = \text{filter}(I, h)$$

Median Filter

- A median filter is a nonlinear **rank filter**:

$$y[m, n] = \text{median}\{I[m + k, n + l]\}_{(k,l) \in \Omega}.$$

- It replaces the center pixel by the middle value in the local neighborhood.
- Unlike a linear filter, there is **no kernel** $h[k, l]$ such that $y[m, n] = \sum_k \sum_l h[k, l] I[m + k, n + l]$.



Salt-and-Pepper Noise: Mean vs. Median

- Impulse noise replaces some pixels with 0 or 255.
- A mean filter averages these outliers with their neighbors, while a median filter can reject them.

```
1 path = "assets/03_filtering/colorado-flower.  
2 I0 = cv2.imread(path, cv2.IMREAD_GRAYSCALE)  
3 I0 = cv2.resize(I0, (380,255))  
4  
5 # Add salt-and-pepper noise  
6 rng, p = np.random.default_rng(7), 0.03  
7 Isp = I0.copy(); M = rng.random(I0.shape)  
8 Isp[M < p] = 0; Isp[M > 1-p] = 255  
9  
10 # Compare mean and median filtering  
11 Imean3 = cv2.blur(Isp, (3,3), borderType=cv2  
12 Imed3 = cv2.medianBlur(Isp, 3)  
13  
14 fig, ax = plt.subplots(2,2,figsize=(6.5,4.4)  
15 for a, im, t in zip(ax.flat,  
16     [I0, Isp, Imean3, Imed3],  
17     ["Original $I$", "Salt-and-pepper noise"  
18     panel(a, im, t)  
19 plt.tight_layout(pad=.3); plt.show()
```

Original /



Salt-and-pepper noise



3×3 mean



3×3 median



Window Size Matters

- Larger windows remove more noise, but they also remove more image detail.

```
1 # Mean filters
2 Imean3 = cv2.blur(Isp, (3,3), borderType=
3 Imean11 = cv2.blur(Isp, (11,11), borderType=
4
5 # Median filters
6 Imed3 = cv2.medianBlur(Isp, 3)
7 Imed11 = cv2.medianBlur(Isp, 11)
8
9 fig, ax = plt.subplots(2,2,figsize=(6.5,4.4)
10 for a, im, t in zip(ax.flat,
11     [Imean3, Imean11, Imed3, Imed11],
12     ["3×3 mean", "11×11 mean", "3×3 median",
13     panel(a, im, t)
14 plt.tight_layout(pad=.3); plt.show()
```

3×3 mean



11×11 mean



3×3 median



11×11 median



Median Filters

- The median filter operates over a window by selecting the median intensity in the window.
- Because ordering is nonlinear, in general,

$$\text{median}(I_1 + I_2) \neq \text{median}(I_1) + \text{median}(I_2)$$

Note

- What advantage does a median filter have over a mean filter?
- Is a median filter a kind of convolution?

Filtering Applications

Improve an image:

- Remove noise
- Smooth before downsampling
- Sharpen local detail
- Estimate illumination or background

Extract information:

- Detect edges and corners
- Measure texture
- Find repeated patterns
- Match a template to image regions

Goal	Useful starting point	Main tradeoff
Reduce noise	Gaussian smoothing	Blurs fine details
Remove impulse noise	Median filtering	Can remove small structures
Detect edges	Sobel or derivative kernels	Amplifies noise
Enhance local detail	Sharpening / high-pass	May create noise or halos
Find a known pattern	Normalized correlation	Sensitive to scale and rotation

Key Takeaways

- A filter maps each output location to a function of a local input neighborhood.
- A kernel's weights determine whether the filter smooths, differentiates, shifts, or sharpens.
- Correlation uses the kernel as written; convolution rotates it by 180° .
- Gaussian filters are composable and separable.
- Sobel filters estimate signed spatial derivatives and edge strength.
- Boundary rules and output modes are part of the algorithm.
- Median filtering is nonlinear and robust to isolated extreme values.