"Don't expect to be easy to work, because if you're easy to do, your heart will be flattered and arrogant."

_ Buddha

AI: Convolution Method

Presenter: Dr. Ha Viet Uyen Synh.

Part A

CONVOLUTION METHOD

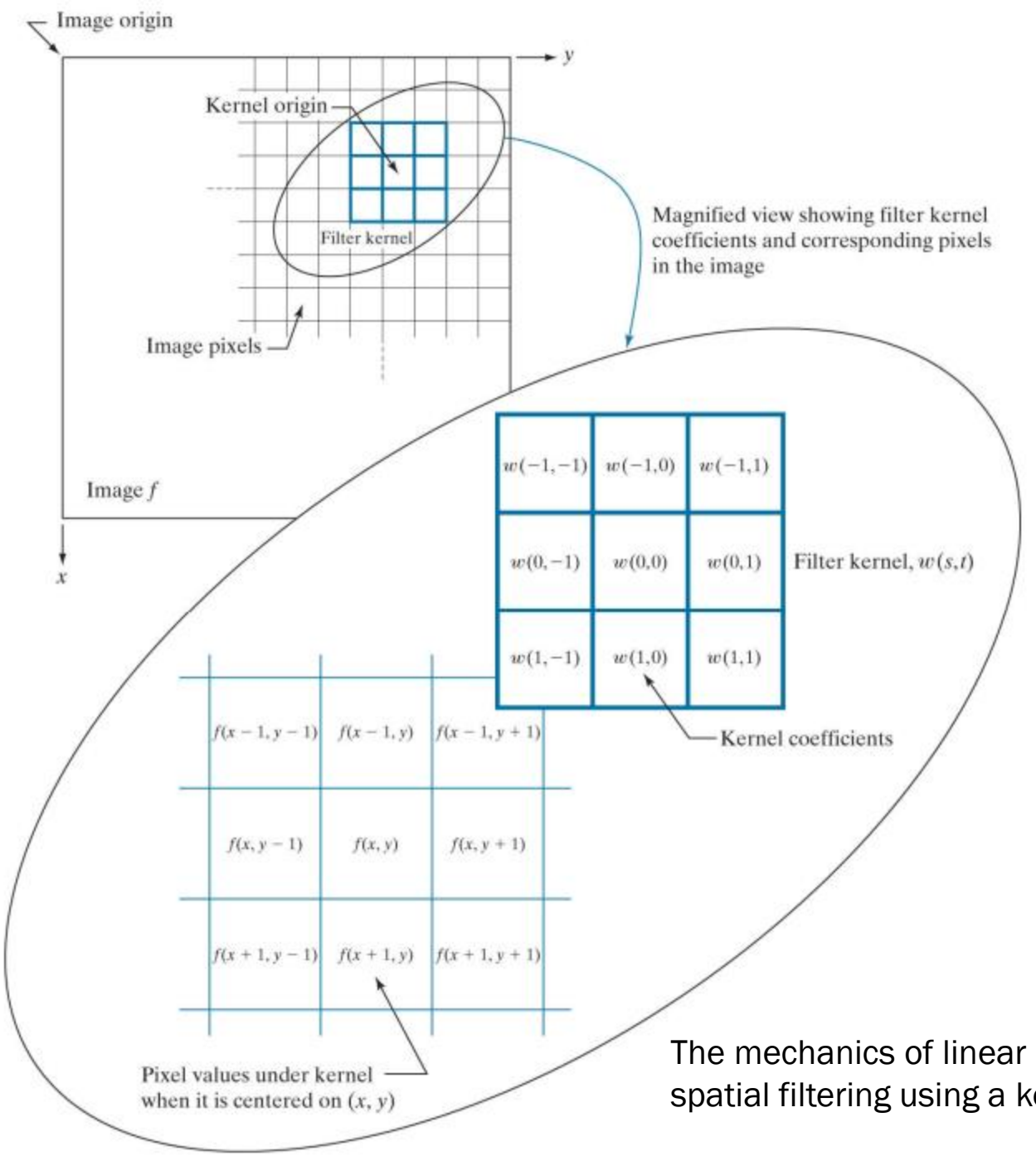
Convolution

Convolution is an important operation in signal and image processing. Convolution operates on two signals (in 1D) or two images (in 2D): one as the “input“ signal (or image), and the other (called the kernel) as a “filter" on the input image, producing an output image (so convolution takes two images as input and produces a third as output).

Let’s start with 1D convolution. Let’s call our input vector f and our kernel g, and say that f has length n, and g has length m. The convolution f ∗ g of f and g is defined as:

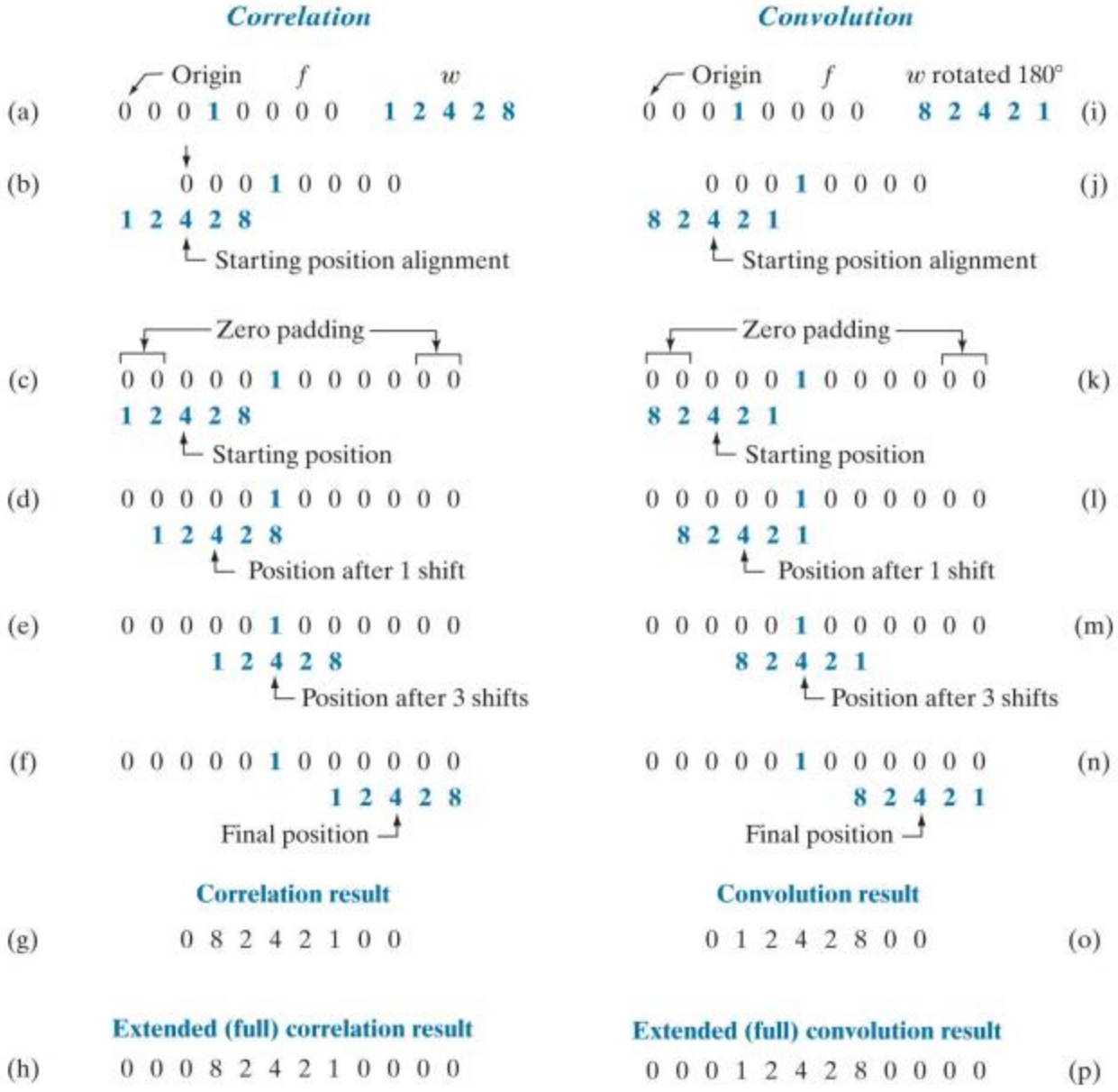
$$(f * g)(i) = \sum_{j=1}^{m} f(j) \cdot g(i - j)$$

One way to think of this operation is that we’re sliding the kernel over the input image. For each position of the kernel, we multiply the overlapping values of the kernel and image together, and add up the results. This sum of products will be the value of the output image at the point in the input image where the kernel is centered.

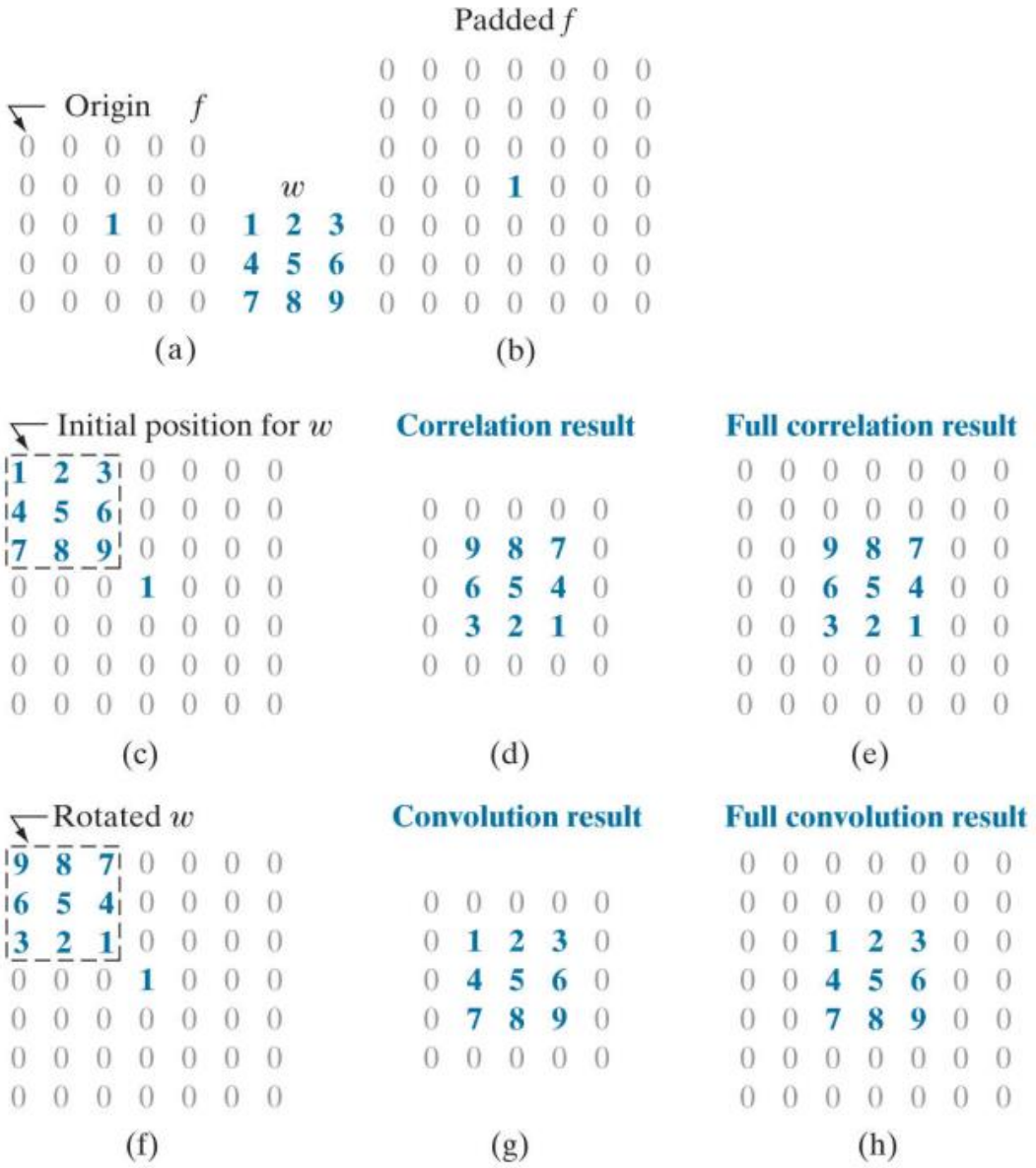


The mechanics of linear spatial filtering using a kernel

Correlation and Convolution of a 1-D kernel



Correlation and Convolution of a 2-D kernel



Basic Operators _ Convolution

$$f(x,y) * w(x,y) = \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} f(m,n) w(x-m, y-n)$$

Or

$$g(x,y) = \sum_{m=-a}^{a} \sum_{n=-b}^{b} w(m,n) f(x+m, y+n)$$

Example:

$$g = f * \begin{bmatrix} 1/2 \\ 0 \\ -1/2 \end{bmatrix}$$

means that

$$g(x,y) = \frac{1}{2} f(x, y+1) - \frac{1}{2} f(x, y-1)$$

Linear spatial filtering

The result is the sum of products of the mask coefficients with the corresponding pixels directly under the mask

Pixels of image

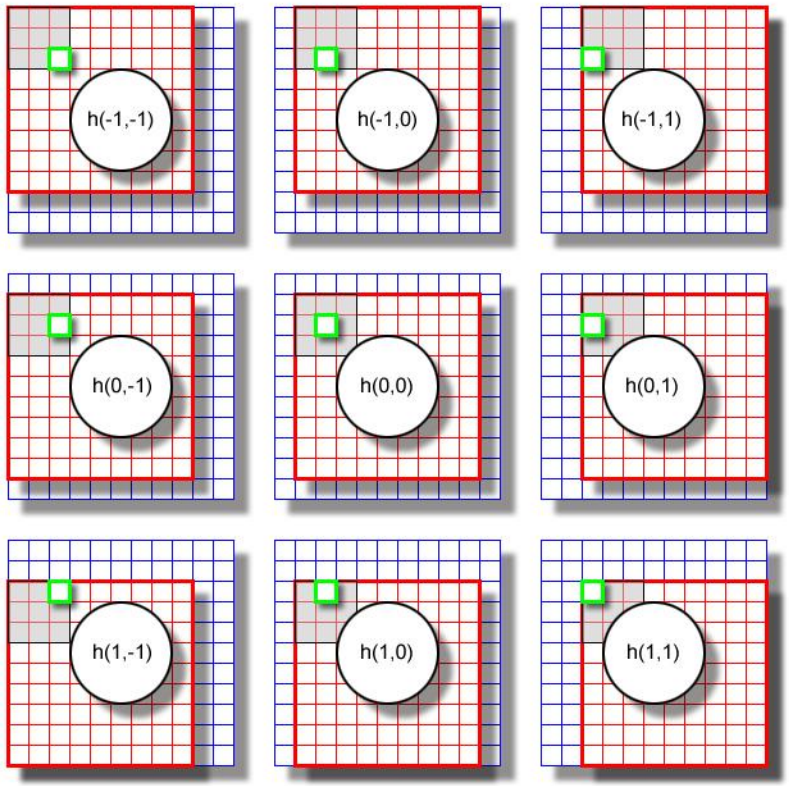
w(-1,-1) f(x-1,y-1)	w(-1,0) f(x-1,y)	w(-1,1) f(x-1,y+1)
w(0,-1) f(x,y-1)	w(0,0) f(x,y)	w(0,1) f(x,y+1)
w(1,-1) f(x+1,y-1)	w(1,0) f(x+1,y)	w(1,1) f(x+1,y+1)

Mask coefficients

w(-1,-1)	w(-1,0)	w(-1,1)
w(0,-1)	w(0,0)	w(0,1)
w(1,-1)	w(1,0)	w(1,1)

$$g(x,y) = w(-1,-1)f(x-1,y-1) + w(-1,0)f(x-1,y) + w(-1,1)f(x-1,y+1) +$$
$$w(0,-1)f(x,y-1) + w(0,0)f(x,y) + w(0,1)f(x,y+1) +$$
$$w(1,-1)f(x+1,y-1) + w(1,0)f(x+1,y) + w(1,1)f(x+1,y+1)$$

Convolution by Copying, Multiplying, and Shifting the Image



original image, **I**

padded image, **P**

effective neighborhood

h(-1,-1)	h(-1,0)	h(-1,1)
h(0,-1)	h(0,0)	h(0,1)
h(1,-1)	h(1,0)	h(1,1)

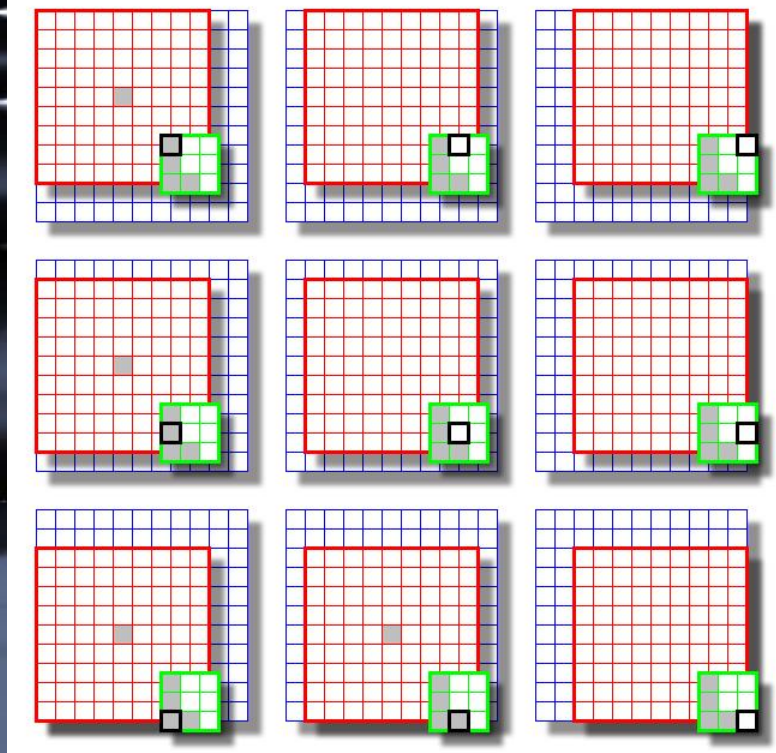
weight matrix

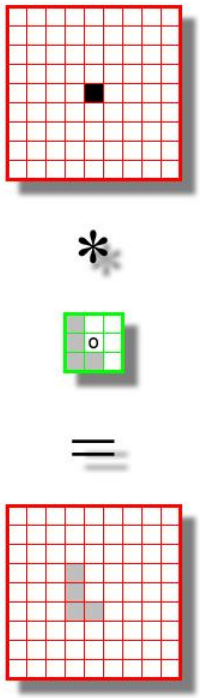
aligned pixels to be summed

weight for image ◯

For each element $\mathbf{h}(r_h,c_h)$ in weight matrix, **h,** image **I** is copied into a zero-padded image, **P**, starting at (r_h,c_h).
Each **P** is multiplied by the corresponding weight, $\mathbf{h}(r_h,c_h)$.
All the **P** images are summed pixel-wise then divided by the sum of the elements of **h**.
The result is cropped out of the center of the accumulated **P**s.

Convolution by Copying, Multiplying, and Shifting the Image





The original image has a black impulse at the center and zeros (white) elsewhere.

The weight matrix has a gray 'L' at its left and zeros (white) elsewhere.

The resulting image has a copy of the weight matrix pegged to the impulse location.

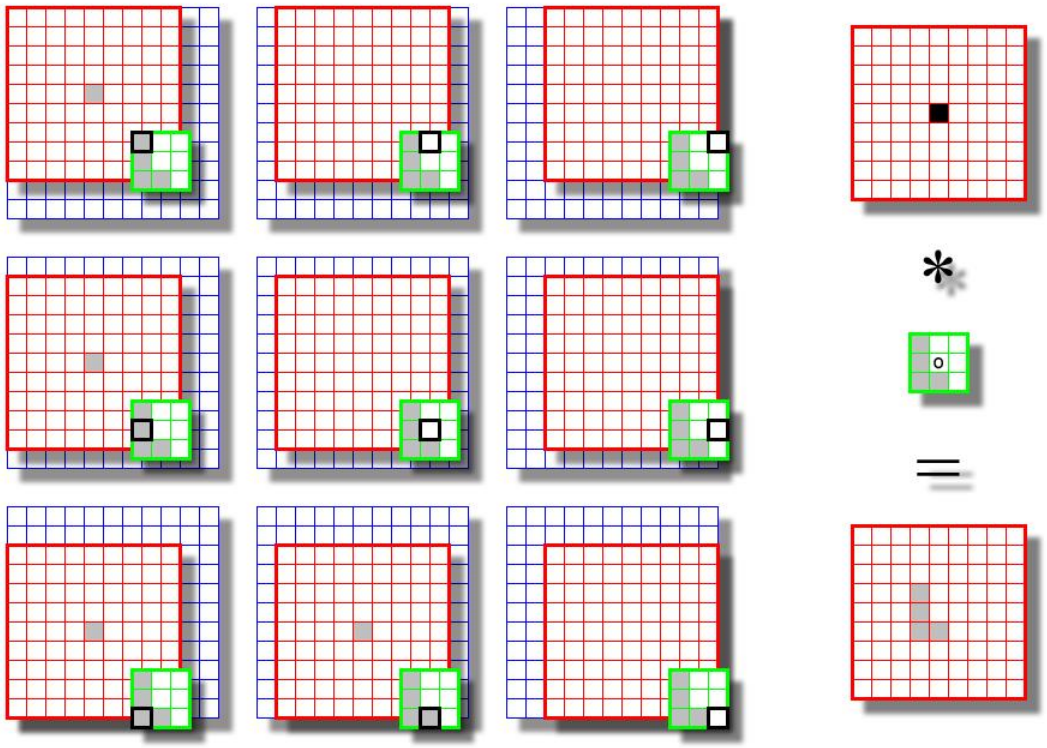
original image, **I**
effective neighborhood
padded image, **P**

In the result, the origin of the weight matrix coincides with the original location of the impulse.

Convolution by Copying, Multiplying, and Shifting the Image

Each copy of the (entire) image is multiplied by the value of the weight matrix in black square (here, white = 0) before being accumulated (pixelwise) in the padded image

The position of the black square relative the center of the weight matrix indicates the shift of the original image relative to the middle of the padded image.

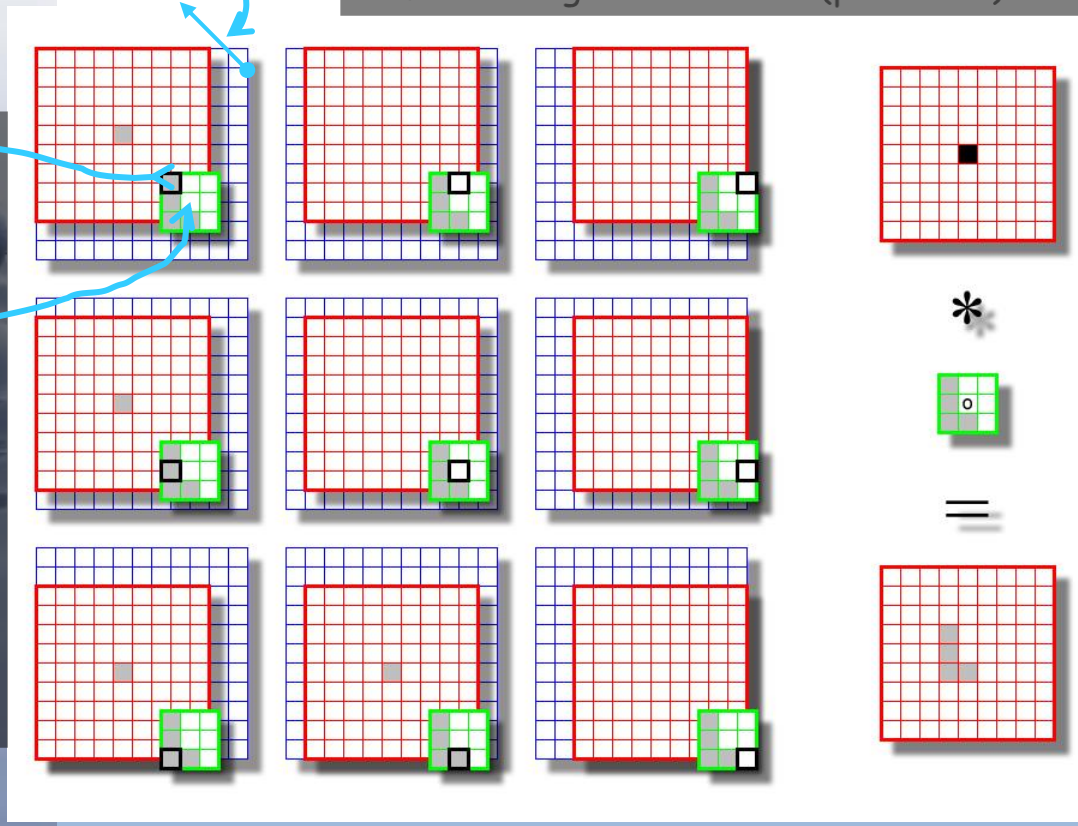


In this image, only the pixel in the center is nonzero so only it shows a result when the image is multiplied by a nonzero value

Convolution by Copying, Multiplying, and Shifting the Image

Each copy of the (entire) image is multiplied by the value of the weight matrix in black square (here, white = 0) before being accumulated (pixelwise) in the padded image

The position of the black square relative the center of the weight matrix indicates the shift of the original image relative to the middle of the padded image.



In this image, only the pixel in the center is nonzero so only it shows a result when the image is multiplied by a nonzero value

Example

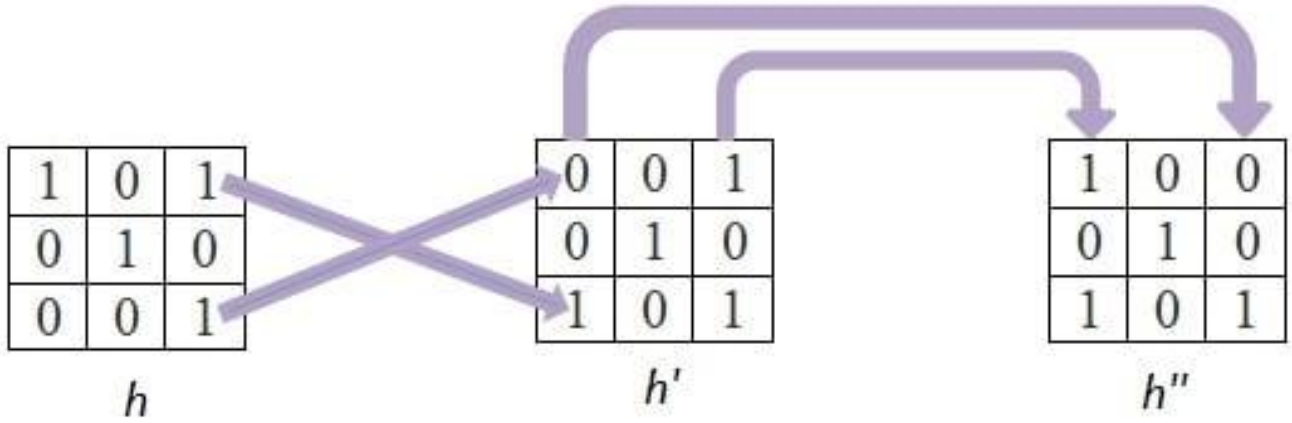
25	100	75	49	130
50	80	0	70	100
5	10	20	30	0
60	50	12	24	32
37	53	55	21	90
140	17	0	23	222

x

1	0	1
0	1	0
0	0	1

h

Matrix inversion

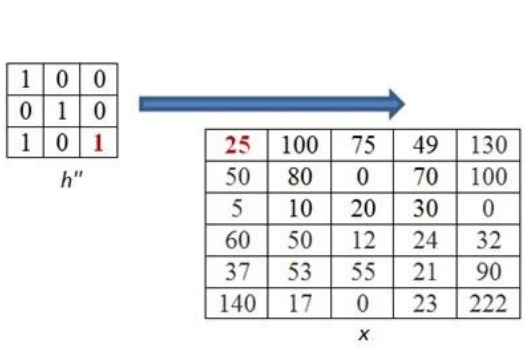


Slide the kernel over the image

$$y[i,j] = \sum_{m=-\infty}^{\infty} \sum_{n=-\infty}^{\infty} h[m,n] \cdot x[i-m, j-n]$$

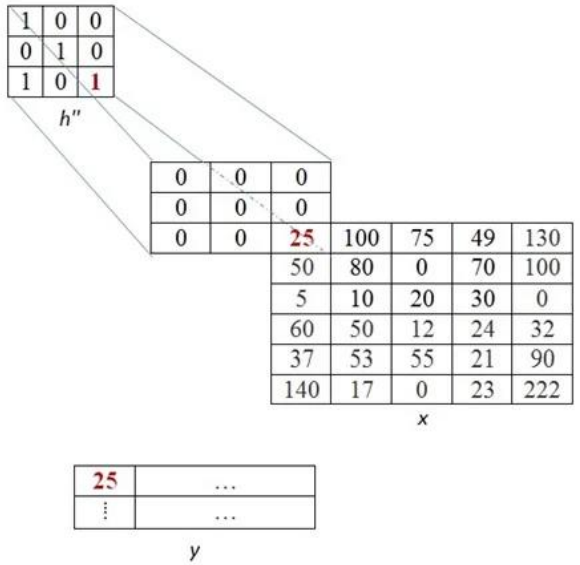
$$y[i,j] = \sum_{m=-\infty}^{\infty} h[m,-1] \cdot x[i-m, j+1] + h[m,0] \cdot x[i-m, j-0] + h[m,1] \cdot x[i-m, j-1]$$

$$\begin{aligned} y[i,j] = {} & h[-1,-1] \cdot x[i+1, j+1] + h[-1,0] \cdot x[i+1, j] + h[-1,1] \cdot x[i+1, j-1] \\ & + h[0,-1] \cdot x[i, j+1] + h[0,0] \cdot x[i, j] + h[0,1] \cdot x[i, j-1] \\ & + h[1,-1] \cdot x[i-1, j+1] + h[1,0] \cdot x[i-1, j] + h[1,1] \cdot x[i-1, j-1] \end{aligned}$$



(a)

Ξ



(b)

1	0	0
0	1	0
1	0	1

h''

25	100	75	49	130
50	**80**	**0**	70	100
5	**10**	**20**	30	0
60	**50**	**12**	24	32
37	53	55	21	90
140	17	0	23	222

x

25	100	100	149	205	49	130
50	105	150	225	149	200	100
5	60	130	140	165	179	130
60	55	**132**	...			
⋮	⋮					

y

$$y(4,3) = 50 \times 1 + 80 \times 0 + 0 \times 0 + 5 \times 0 + 10 \times 1 + 20 \times 0 + 60 \times 1 + 50 \times 0 + 12 \times 1$$
$$= 50 + 0 + 0 + 0 + 10 + 0 + 60 + 0 + 12 = 132$$

25	100	75	49	130
50	80	0	70	100
5	10	20	30	0
60	50	**12**	**24**	**32**
37	53	**55**	**21**	**90**
140	17	**0**	**23**	**222**

x

1	0	0
0	1	0
1	0	1

h''

25	100	100	149	205	49	130
50	105	150	225	149	200	100
5	60	130	140	165	179	130
60	55	132	174	74	94	132
37	113	147	96	189	83	90
140	54	253	145	**255**	...	
⋮	⋮					

y

$$y(6,5) = 12 \times 1 + 24 \times 0 + 32 \times 0 + 55 \times 0 + 21 \times 1 + 90 \times 0 + 0 \times 1 + 23 \times 0 + 222 \times 1$$
$$= 12 + 0 + 0 + 0 + 21 + 0 + 0 + 0 + 222 = 255$$

25	100	75	49	130
50	80	0	70	100
5	10	20	30	0
60	50	12	24	32
37	53	55	21	90
140	17	0	**23**	**222**

x

1	**0**	0
0	1	0
1	0	1

h''

25	100	100	149	205	49	130
50	105	150	225	149	200	100
5	60	130	140	165	179	130
60	55	132	174	74	94	132
37	113	147	96	189	83	90
140	54	253	145	255	137	254
0	140	54	53	78	243	90
0	0	140	17	0	**23**	…

y

$$y\,(8,\,6) = 23 \times 1 + 222 \times 0 = 23 + 0 = 23$$

Convolution operation

Input Volume (+pad 1) (7x7x3)

x[:,:,0]

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

x[:,:,1]

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

x[:,:,2]

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

Filter W0 (3x3x3)

w0[:,:,0]

-1	0	1
0	0	1
1	-1	1

w0[:,:,1]

-1	0	1
1	-1	1
0	1	0

w0[:,:,2]

-1	1	1
1	1	0
0	-1	0

Bias b0 (1x1x1)

b0[:,:,0]

1

Filter W1 (3x3x3)

w1[:,:,0]

0	1	-1
0	-1	0
0	-1	1

w1[:,:,1]

-1	0	0
1	-1	0
1	-1	0

w1[:,:,2]

-1	1	-1
0	-1	-1
1	0	0

Bias b1 (1x1x1)

b1[:,:,0]

0

Output Volume (3x3x2)

o[:,:,0]

2	3	3
3	7	3
8	10	-3

o[:,:,1]

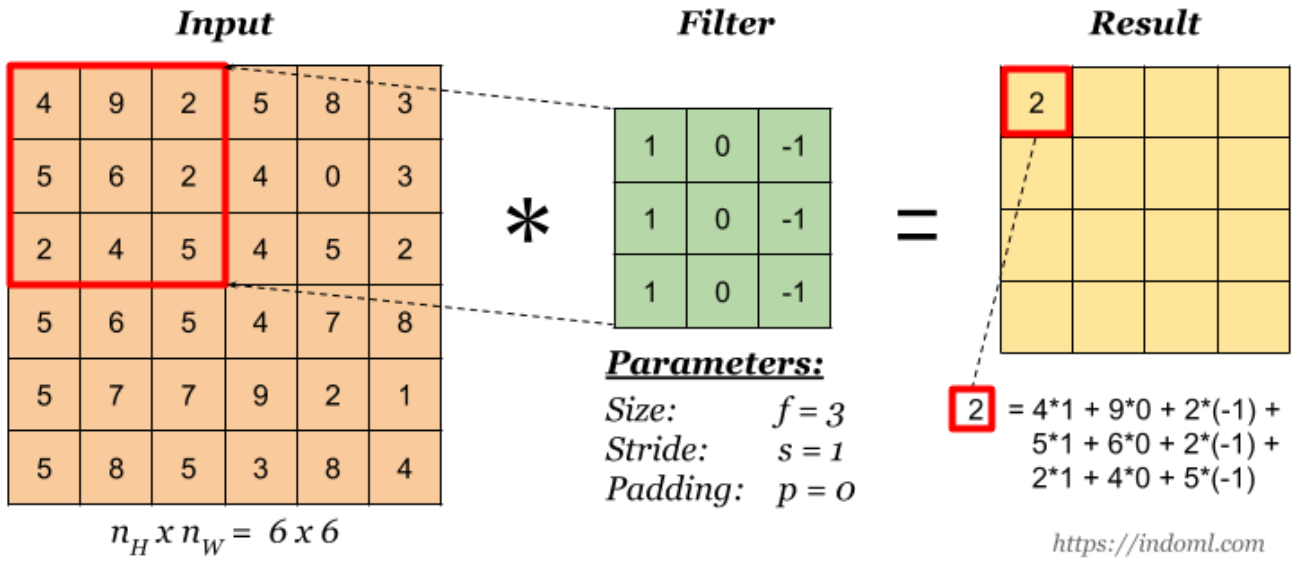
-8	-8	-3
-3	1	0
-3	-8	-5

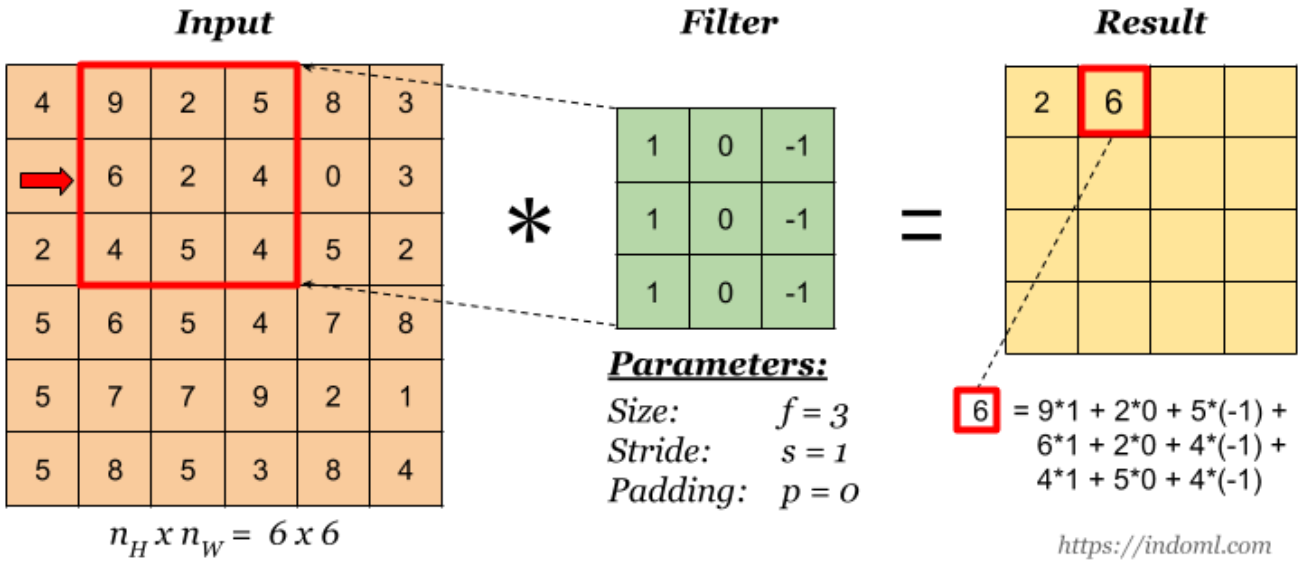
toggle movement

PART B

CONVOLUTION LAYER

Filter

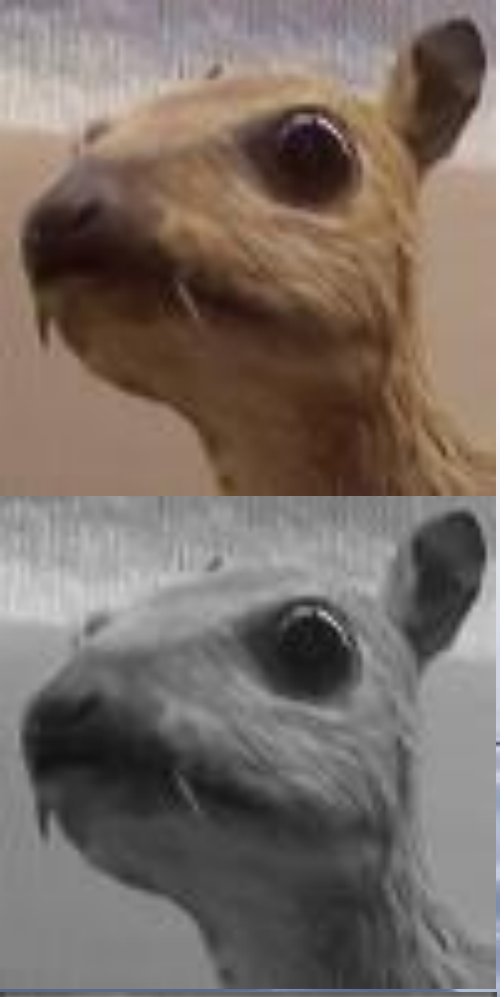

Input
Filter
Result
4 9 2 5 8 3
5 6 2 4 0 3
2 4 5 4 5 2
5 6 5 4 7 8
5 7 7 9 2 1
5 8 5 3 8 4
1 0 -1
1 0 -1
1 0 -1
Parameters:
Size: f = 3
Stride: s = 1
Padding: p = 0
2
2 = 4*1 + 9*0 + 2*(-1) +
5*1 + 6*0 + 2*(-1) +
2*1 + 4*0 + 5*(-1)
n_H x n_W = 6 x 6
https://indoml.com


Input
Filter
Result
4 9 2 5 8 3
6 2 4 0 3
2 4 5 4 5 2
5 6 5 4 7 8
5 7 7 9 2 1
5 8 5 3 8 4
1 0 -1
1 0 -1
1 0 -1
Parameters:
Size: f = 3
Stride: s = 1
Padding: p = 0
2 6
6
6 = 9*1 + 2*0 + 5*(-1) +
6*1 + 2*0 + 4*(-1) +
4*1 + 5*0 + 4*(-1)
n_H x n_W = 6 x 6
https://indoml.com

1. One-Dimensional Space

```
import numpy as np

import torch
import torch.nn as nn
import torchvision.transforms

import cv2 as cv

from google.colab.patches import cv2_imshow

import matplotlib.pyplot as plt
import math

from PIL import Image
import requests
from io import BytesIO
```

```
class ConvNet(nn.Module):
    def __init__(self, num_classes=10):
        super(ConvNet, self).__init__()
        # Test layer
        self.conv1_1 = nn.Conv2d(1, 1, kernel_size=(3, 3)
, stride=(1, 1), padding=(1, 1))

    def forward(self, x):
        # Layer 1
        out1_h = self.conv1_1(x)

        return out1_h

# Test image
image = cv.imread('Vd-Orig.png',  cv.IMREAD_GRAYSCALE)

np_array = np.array(image)
cv2_imshow(np_array)
```



```
tensor = torch.from_numpy(np_array)
# Create new model
conv = ConvNet()

# Assign test weight - NOT WORKING!!
weights1_1 = torch.tensor( [[0.0625,0.125,0.0625], [0.125
,0.25,0.125], [0.0625,0.125,0.0625]])
weights1_1 = weights1_1.view(1, 1, 3, 3)

conv.conv1_1.weight.data = weights1_1.float()

# # Run the model
blur = conv(tensor[None, None, ...].float()) #BxCxHxW
blur  = torch.reshape(blur, (blur.shape[2],  blur.shape[3
])) #HxW
blur _features =  blur.detach().numpy()
cv2_imshow(blur_features)
```

2. Two-Dimensional Space

```
class ConvNet(nn.Module):
    def __init__(self, num_classes=10):
        super(ConvNet, self).__init__()
        # Test layer
        self.conv1_1 = nn.Conv2d(1, 1, kernel_size=(3, 3)
, stride=(1, 1), padding=(1, 1))

    def forward(self, x):
        # Layer 1
        out1_h = self.conv1_1(x)
        return out1_h

# Test image
image = cv.imread('Vd-Orig.png')
cv2_imshow(image)
```

```
conv = ConvNet()
# Assign test weight - NOT WORKING!!
weights1_1 = torch.tensor( [[0.0625,0.125,0.0625], [0.125
,0.25,0.125], [0.0625,0.125,0.0625]])
weights1_1 = weights1_1.view(1, 1, 3, 3) #1 input, 1 outp
ut, kernel size (3x3)
conv.conv1_1.weight.data = weights1_1.float()

b,g,r = cv.split(image)
np_array_0 = np.array(b)
np_array_1 = np.array(g)
np_array_2 = np.array(r)

tensor0 = torch.from_numpy(np_array_0)
tensor1 = torch.from_numpy(np_array_1)
tensor2 = torch.from_numpy(np_array_2)
```

```
# Run the model
out0 = conv(tensor0[None, None, ...].float())
out1 = conv(tensor1[None, None, ...].float())
out2 = conv(tensor2[None, None, ...].float())

out0 = torch.reshape(out0, (out0.shape[2], out0.shape[3]))
out0 = out0.detach().numpy()
out1 = torch.reshape(out1, (out1.shape[2], out1.shape[3]))
out1 = out1.detach().numpy()
out2 = torch.reshape(out2, (out2.shape[2], out2.shape[3]))
out2 = out2.detach().numpy()

out = cv.merge((out0, out1, out2))
cv2_imshow(out)
```

3. 3-D Space

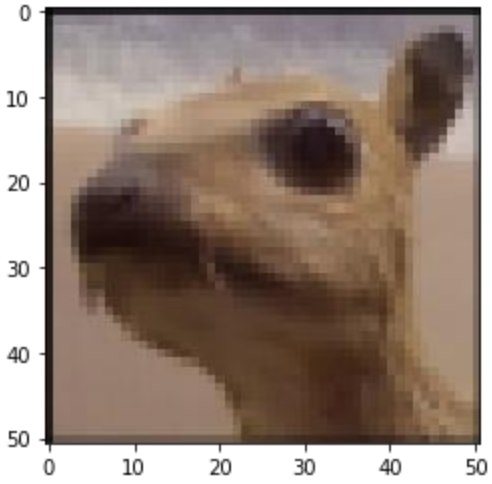
```
import numpy as np
import matplotlib.pyplot as plt

import cv2 as cv

import torch
import torch.nn as nn
import torchvision.transforms

from google.colab.patches import cv2_imshow

from PIL import Image
import requests
from io import BytesIO
```

```
#BxCxHxW
a = np.zeros([3, 3, 3, 3])
a[:, :, 1, 1] = 0.25
a[:, :, 0, 1] = 0.125
a[:, :, 1, 0] = 0.125
a[:, :, 2, 1] = 0.125
a[:, :, 1, 2] = 0.125
a[:, :, 0, 0] = 0.0625
a[:, :, 0, 2] = 0.0625
a[:, :, 2, 0] = 0.0625
a[:, :, 2, 2] = 0.0625
a[0, 1, :, :] = 0
a[0, 2, :, :] = 0
a[1, 0, :, :] = 0
a[1, 2, :, :] = 0
a[2, 0, :, :] = 0
a[2, 1, :, :] = 0
kernel = torch.from_numpy(a)
print(a)
```

```
[[[[0.0625 0.125  0.0625]
   [0.125  0.25   0.125 ]
   [0.0625 0.125  0.0625]]

  [[0.     0.     0.    ]
   [0.     0.     0.    ]
   [0.     0.     0.    ]]

  [[0.     0.     0.    ]
   [0.     0.     0.    ]
   [0.     0.     0.    ]]]


 [[[0.     0.     0.    ]
   [0.     0.     0.    ]
   [0.     0.     0.    ]]

  [[0.0625 0.125  0.0625]
   [0.125  0.25   0.125 ]
   [0.0625 0.125  0.0625]]

  [[0.     0.     0.    ]
   [0.     0.     0.    ]
   [0.     0.     0.    ]]]


 [[[0.     0.     0.    ]
   [0.     0.     0.    ]
   [0.     0.     0.    ]]

  [[0.     0.     0.    ]
   [0.     0.     0.    ]
   [0.     0.     0.    ]]

  [[0.0625 0.125  0.0625]
   [0.125  0.25   0.125 ]
   [0.0625 0.125  0.0625]]]]
```

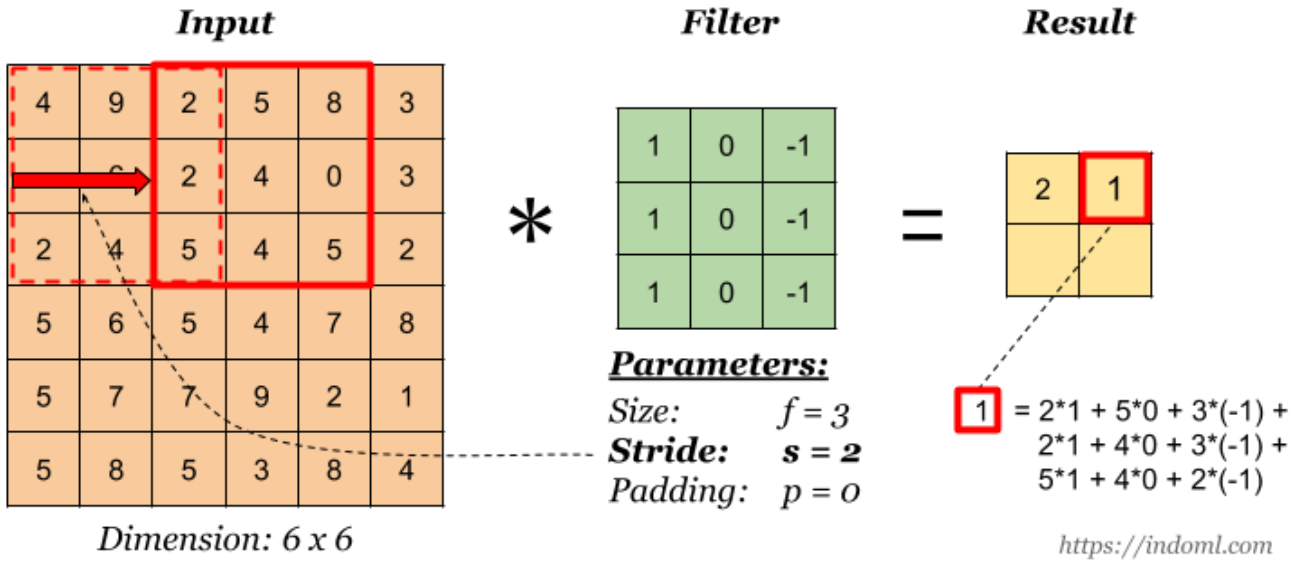
```
class ConvolutionalNetwork(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(in_channels=3, out_channel
s=3,kernel_size=3, stride=2, padding=2)
        self.conv1.weight.data  = kernel.float() # <<<===

    def forward(self, X):
            X = self.conv1(X)
            return X

CNNmodel = ConvolutionalNetwork()

response = requests.get('https://upload.wikimedia.org
/wikipedia/commons/5/50/Vd-Orig.png')
im = Image.open(BytesIO(response.content))

pic = np.array(im)
pic_float = np.float32(pic)
pic_float = np.expand_dims(pic_float,axis=0)

f, axarr = plt.subplots()
axarr.imshow(pic)
plt.show()
```

```
# BxCxHxW
out = CNNmodel(torch.tensor(pic_float).permute(0,3,1,2))
out = out.permute(0,2,3,1).detach().numpy()[0, :, :, :]

f, axarr = plt.subplots()
axarr.imshow(np.uint8(out))
plt.show()
```



Strides

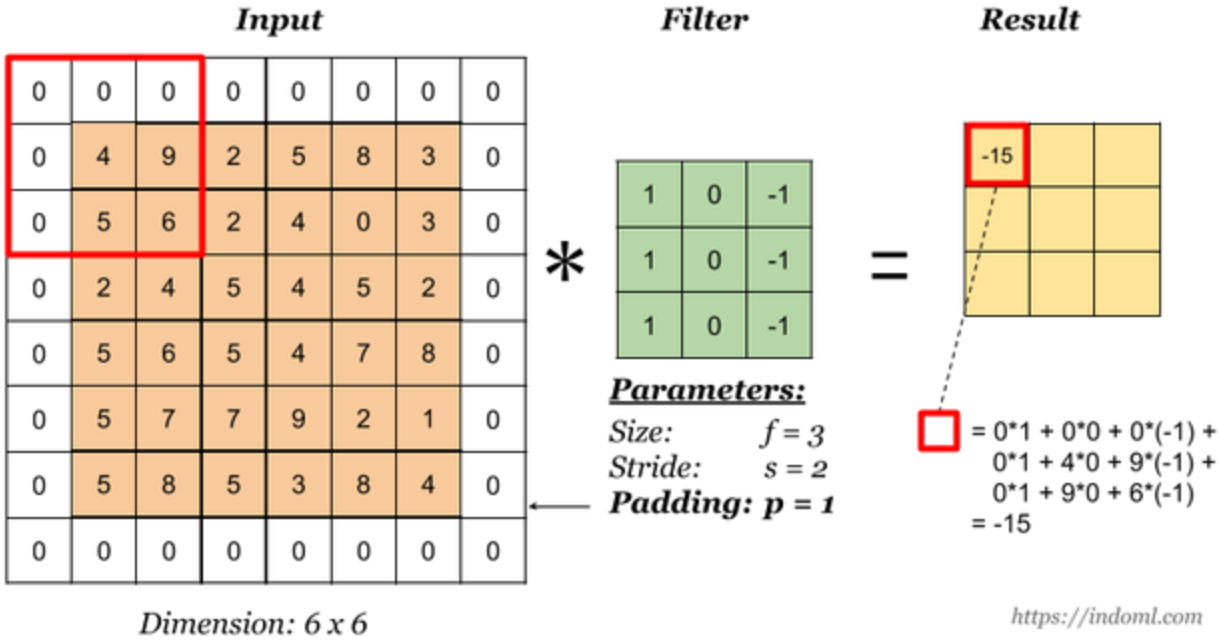
Stride is the number of pixels shifts over the input matrix. When the stride is 1 then we move the filters to 1 pixel at a time. When the stride is 2 then we move the filters to 2 pixels at a time and so on.



Padding

Sometimes filter does not fit perfectly fit the input image. We have two options:

- Pad the picture with zeros (zero-padding) so that it fits
- Drop the part of the image where the filter did not fit. This is called valid padding which keeps only valid part of the image.



Part A

NOISE GENERATION

What is noise?

Noise arises as a result of unmodelled or modellable processes going on in the production and capture of the real signal.

It is not part of the ideal signal and may be caused by a wide range of sources, e.g. variations in the detector sensitivity, environmental variations, the discrete nature of radiation, transmission or quantization errors, etc.

It is also possible to treat irrelevant scene details as if they are image noise (e.g. surface reflectance textures). The characteristics of noise depend on its source, as does the operator which best reduces its effects.

Many image processing packages contain operators to artificially add noise to an image. Deliberately corrupting an image with noise allows us to test the resistance of an image processing operator to noise and assess the performance of various noise filters.

Noise

Noise can generally be grouped into two classes:

Independent noise can often be described by an additive noise model, where the recorded image f(i,j) is the sum of the true image s(i,j) and the noise n(i,j):

$$f(i,j) = s(i,j) + n(i,j)$$

The noise n(i,j) is often zero-mean and described by its variance σ_n^2 . The impact of the noise on the image is often described by the signal to noise ratio (SNR), which is given by

$$SNR = \frac{\sigma_s}{\sigma_n} = \sqrt{\frac{\sigma_f^2}{\sigma_n^2} - 1}$$

where σ_s^2 and σ_f^2 are the variances of the true image and the recorded image, respectively.

Data-dependent noise is possible to model noise with a multiplicative, or non-linear, model. These models are mathematically more complicated; hence, if possible, the noise is assumed to be data independent.

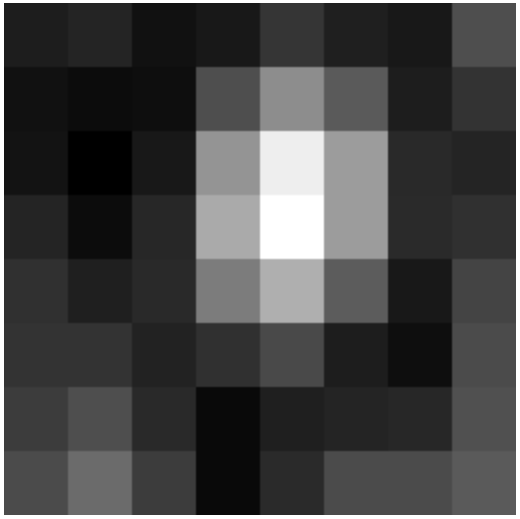
Detector Noise

One kind of noise which occurs in all recorded images to a certain extent is detector noise.

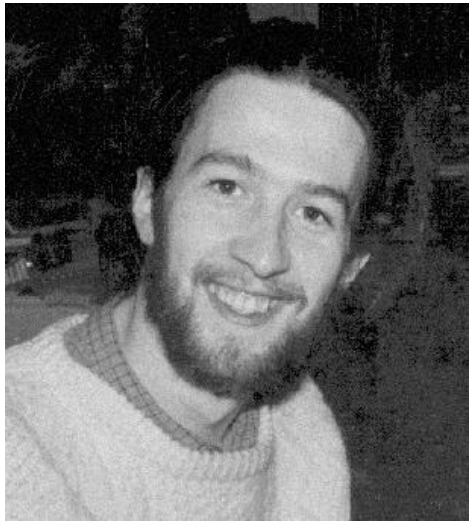
This kind of noise is due to the discrete nature of radiation, i.e. the fact that each imaging system is recording an image by counting photons.

Allowing some assumptions (which are valid for many applications) this noise can be modeled with an independent, additive model, where the noise n(i,j) has a zero-mean Gaussian distribution described by its standard deviation or variance.

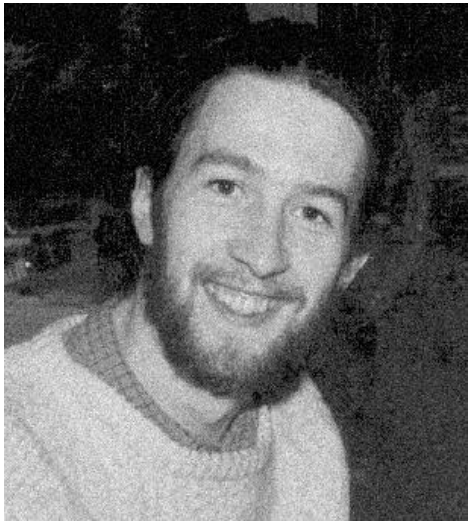
This means that each pixel in the noisy image is the sum of the true pixel value and a random, Gaussian distributed noise value.



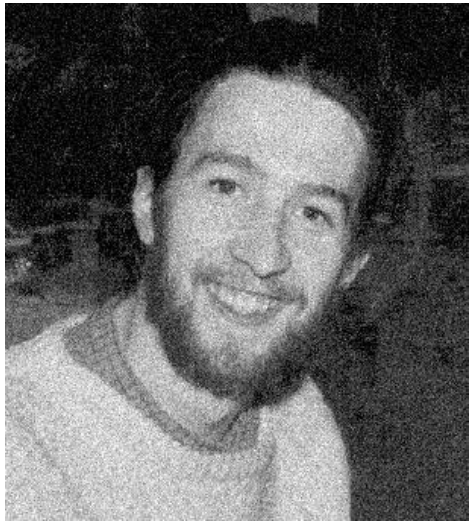
Gaussian Noise



σ=8

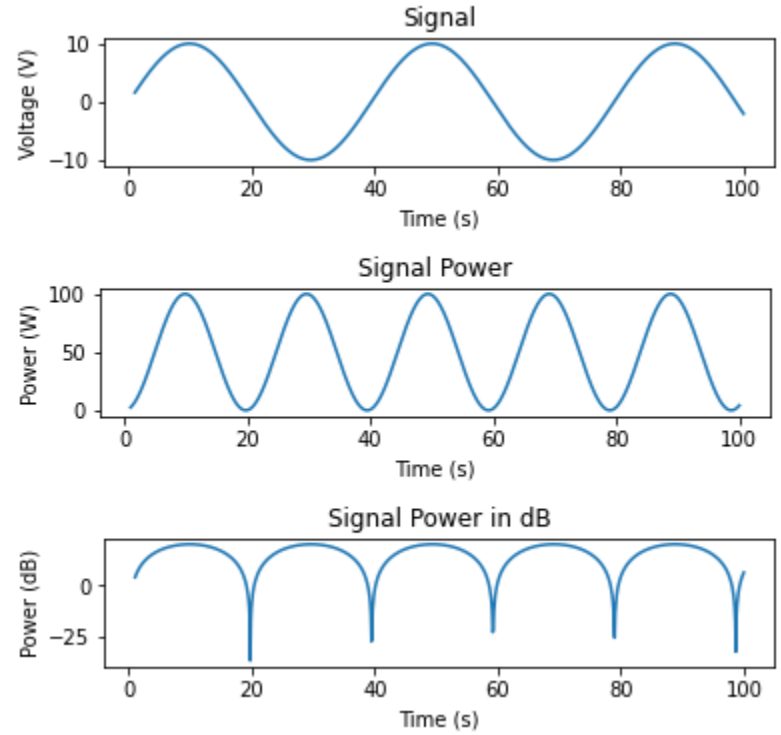


σ=13



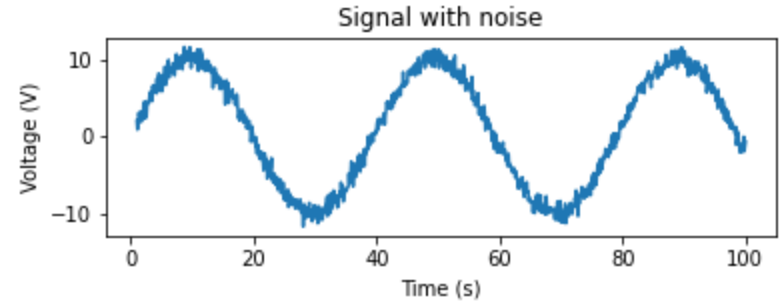
σ=20

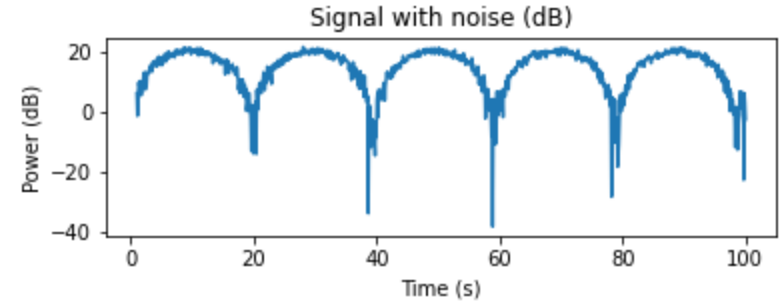
Implementation




```
import numpy as np
import matplotlib.pyplot as plt

t = np.linspace(1, 100, 1000)
x_volts = 10*np.sin(t/(2*np.pi))
plt.subplot(3,1,1)
plt.plot(t, x_volts)
plt.title('Signal')
plt.ylabel('Voltage (V)')
plt.xlabel('Time (s)')
plt.show()

x_watts = x_volts ** 2
plt.subplot(3,1,2)
plt.plot(t, x_watts)
plt.title('Signal Power')
plt.ylabel('Power (W)')
plt.xlabel('Time (s)')
plt.show()

x_db = 10 * np.log10(x_watts)
plt.subplot(3,1,3)
plt.plot(t, x_db)
plt.title('Signal Power in dB')
plt.ylabel('Power (dB)')
plt.xlabel('Time (s)')
plt.show()
```

$$G(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{x^2}{2\sigma^2}}$$



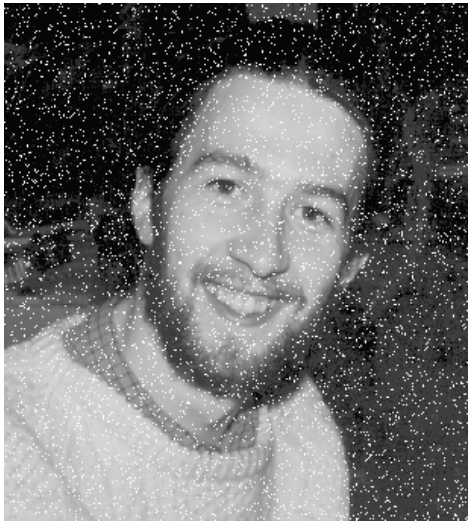


```
# Set a target SNR
target_snr_db = 20
# Calculate signal power and convert to dB
sig_avg_watts = np.mean(x_watts)
sig_avg_db = 10 * np.log10(sig_avg_watts)
noise_avg_db = sig_avg_db - target_snr_db
noise_avg_watts = 10 ** (noise_avg_db / 10)
# Generate an sample of white noise
mean_noise = 0
noise_volts = np.random.normal(mean_noise,
np.sqrt(noise_avg_watts), len(x_watts))
# Noise up the original signal
y_volts = x_volts + noise_volts
# Plot signal with noise
plt.subplot(2,1,1)
plt.plot(t, y_volts)
plt.title('Signal with noise')
plt.ylabel('Voltage (V)')
plt.xlabel('Time (s)')
plt.show()
# Plot in dB
y_watts = y_volts ** 2
y_db = 10 * np.log10(y_watts)
plt.subplot(2,1,2)
plt.plot(t, 10* np.log10(y_volts**2))
plt.title('Signal with noise (dB)')
plt.ylabel('Power (dB)')
plt.xlabel('Time (s)')
plt.show()
```

Salt & Pepper Noise

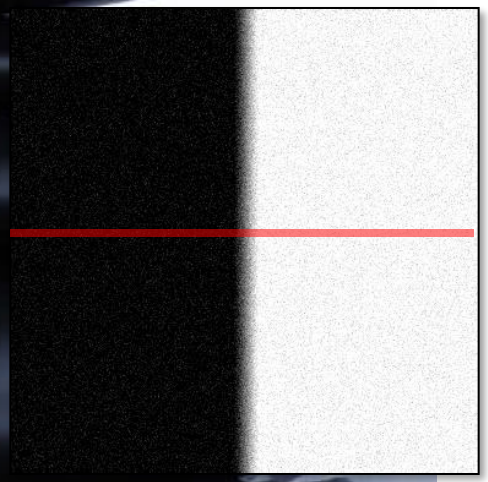
1%

3%



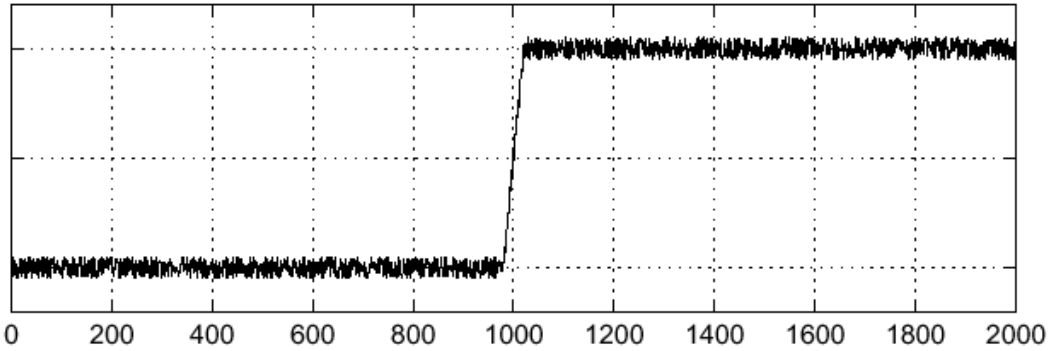
5%

Effects of noise

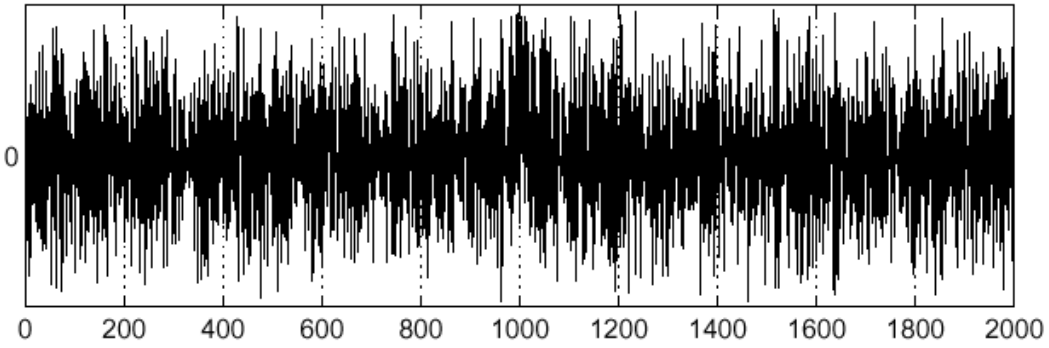


Noisy input image

$f(x)$

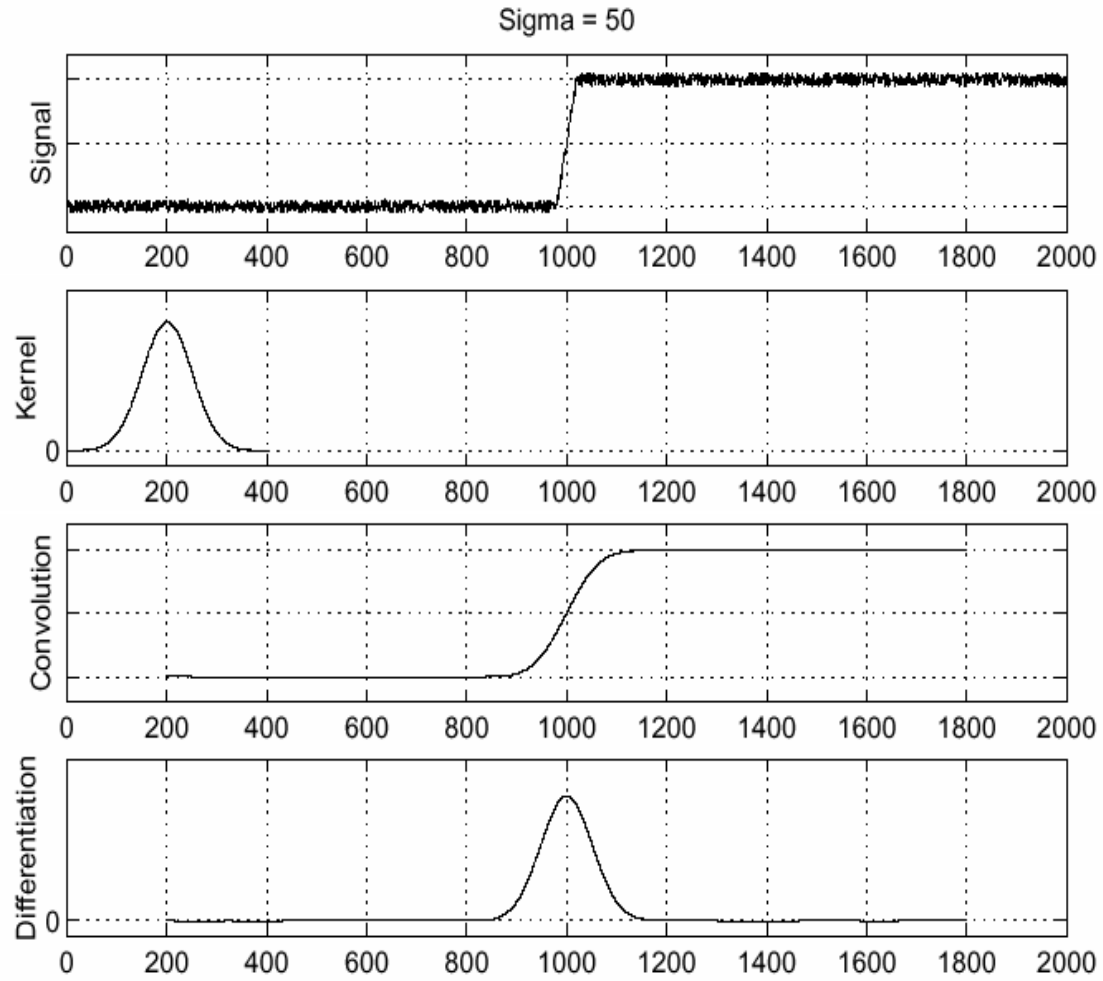


$\frac{d}{dx}f(x)$



Where is the edge?

Solution: smooth first



f

h

$f * h$

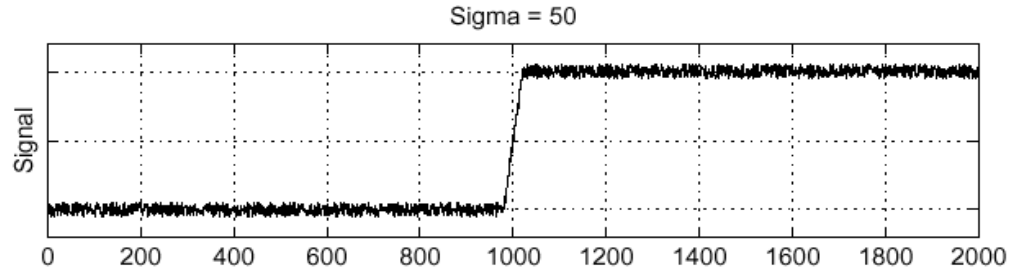
$\frac{d}{dx}(f * h)$

To find edges, look for peaks in $\frac{d}{dx}(f * h)$

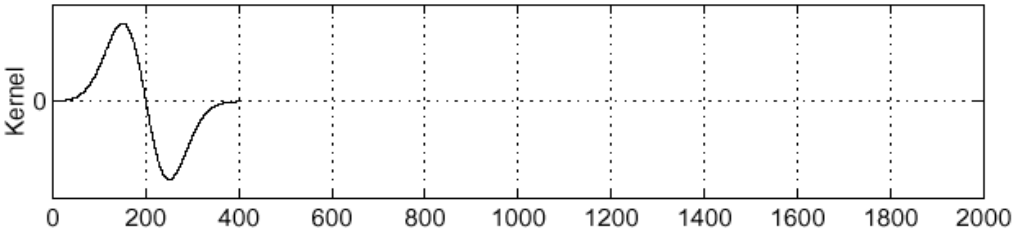
Associative property of convolution

- Differentiation is convolution, and convolution is associative: $$\frac{d}{dx}(f * h) = f * \frac{d}{dx}h$$
- This saves us one operation:

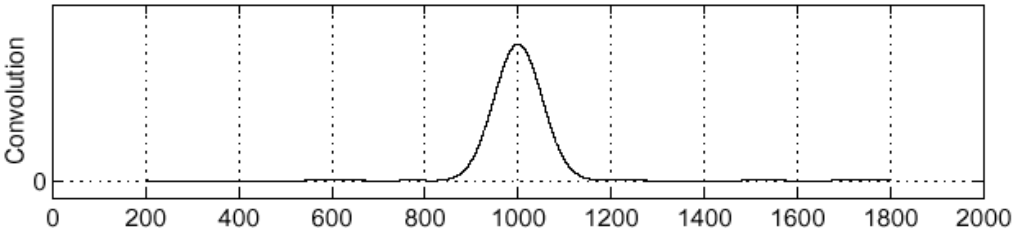
f



$\frac{d}{dx}h$



$f * \frac{d}{dx}h$



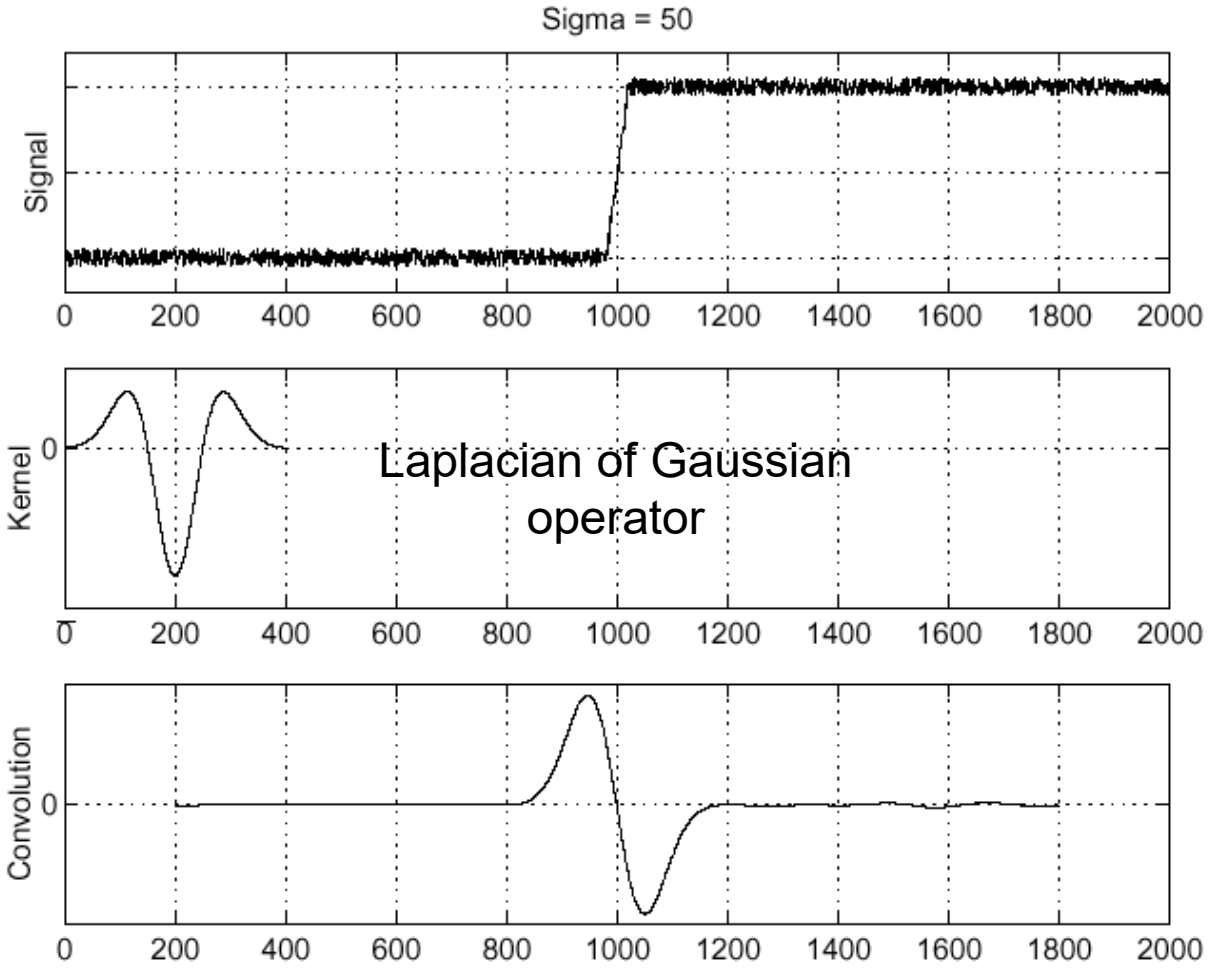
Laplacian of Gaussian

Look for zero-crossings of $\frac{\partial^2}{\partial x^2}(h \star f)$

f

$\frac{\partial^2}{\partial x^2} h$

$(\frac{\partial^2}{\partial x^2} h) \star f$



Example

Apply the Gaussian filter to the image:
Borders: keep border values as they are

15	20	25	25	15	10
20	15	50	30	20	15
20	50	55	60	30	20
20	15	65	30	15	30
15	20	30	20	25	30
20	25	15	20	10	15

Original image

¼* | 1 | 2 | 1 |

¼* | 1 |
| 2 |
| 1 |

Or:

1	2	1
2	4	2
1	2	1

*1/16

15	20	24	23	16	10
20	25	36	33	21	15
20	44	55	51	35	20
20	29	44	35	22	30
15	21	25	24	25	30
20	21	19	16	14	15

15	20	24	23	16	10
19	28	38	35	23	15
20	35	48	43	28	21
19	31	42	36	26	28
18	23	28	25	22	21
20	21	19	16	14	15

Part C

KERNELS

1. Median Filtering

Median filtering is a nonlinear method used to remove noise from images. It is widely used as it is very effective at removing noise while preserving edges. It is particularly effective at removing ‘salt and pepper’ type noise.

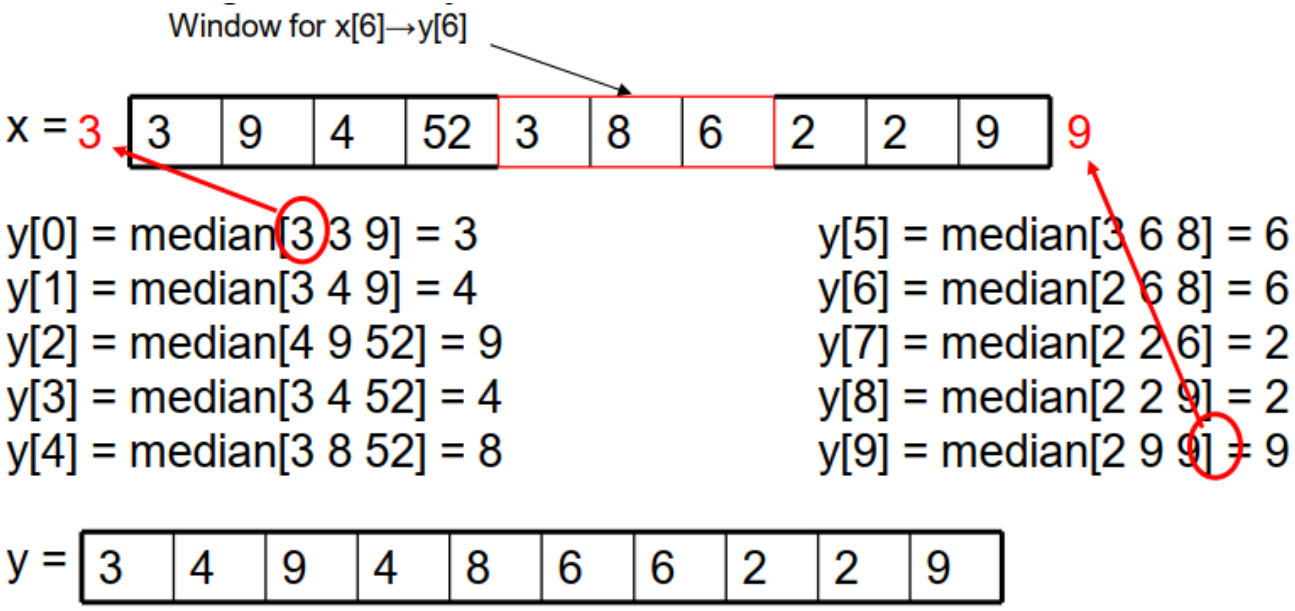
The median filter works by moving through the image pixel by pixel, replacing each value with the median value of neighboring pixels. The pattern of neighbors is called the "window", which slides, pixel by pixel, over the entire image.

The median is calculated by first sorting all the pixel values from the window into numerical order, and then replacing the pixel being considered with the middle (median) pixel value

Example

The following example shows the application of a median filter to a simple one dimensional signal.

A window size of three is used, with one entry immediately preceding and following each entry.



Example

2D Median filtering example using a 3 x 3 sampling window.

Keeping border values unchanged

Sorted: 0,0,1,1,1,2,2,4,4

Input

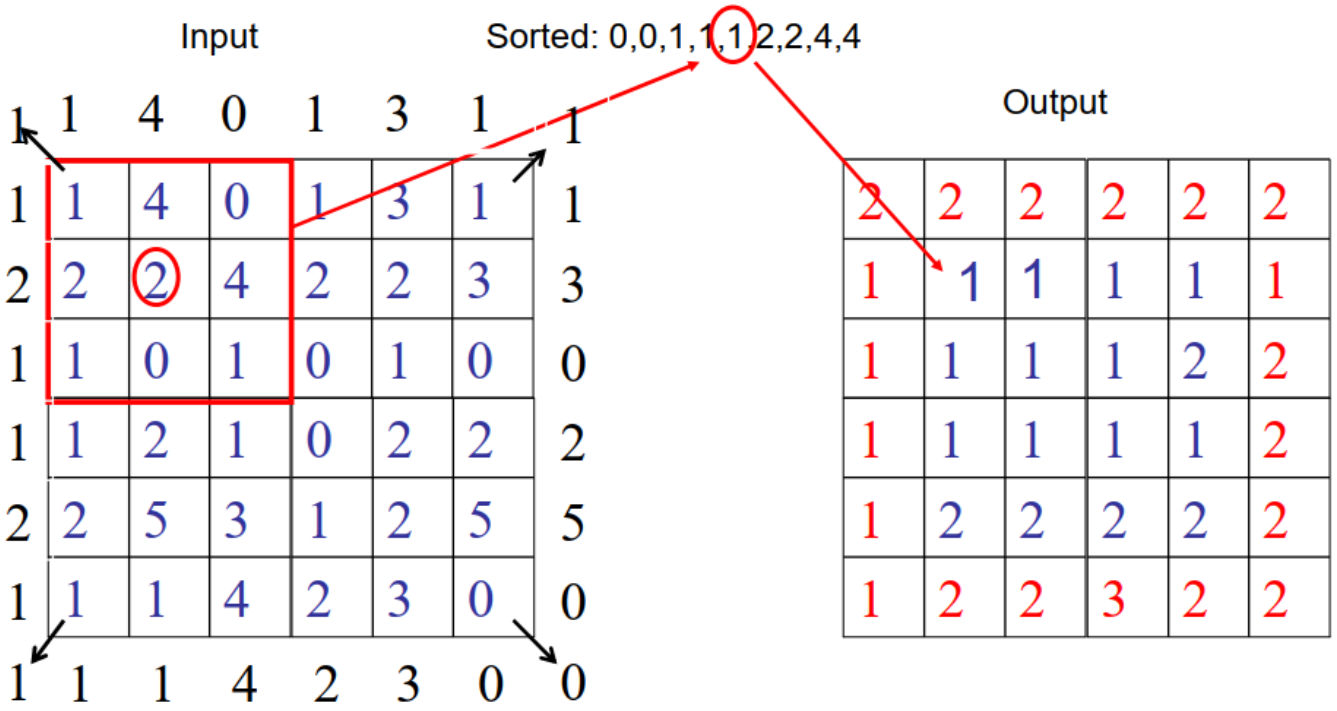
1	4	0	1	3	1
2	2	4	2	2	3
1	0	1	0	1	0
1	2	1	0	2	2
2	5	3	1	2	5
1	1	4	2	3	0

Output

1	4	0	1	3	1
2	1	1	1	1	3
1	1	1	1	2	0
1	1	1	1	1	2
2	2	2	2	2	5
1	1	4	2	3	0

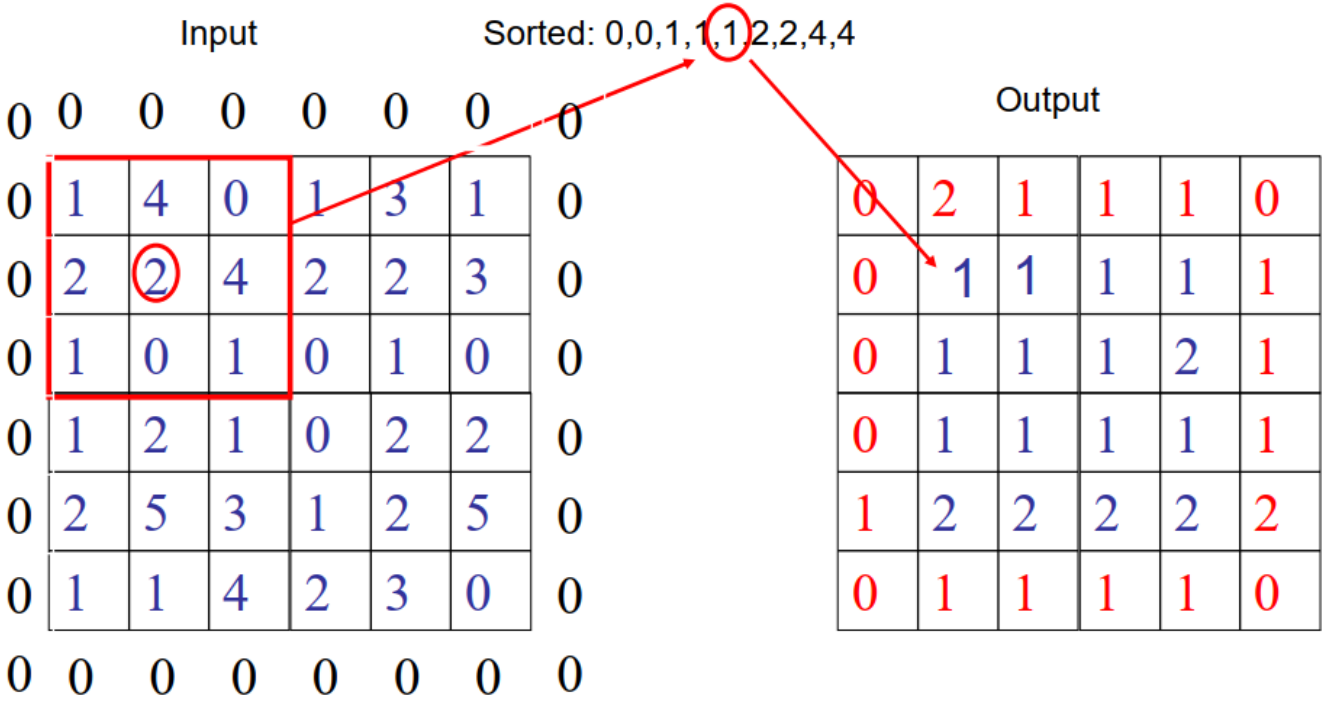
Example

2D Median filtering example using a 3 x 3 sampling window. Extending border values outside with values at boundary.



Example

2D Median filtering example using a 3 x 3 sampling window. Extending border values outside with 0s



2. Average Filtering

Average (or mean) filtering is a method of ‘smoothing’ images by reducing the amount of intensity variation between neighbouring pixels.

The average filter works by moving through the image pixel by pixel, replacing each value with the average value of neighbouring pixels, including itself.

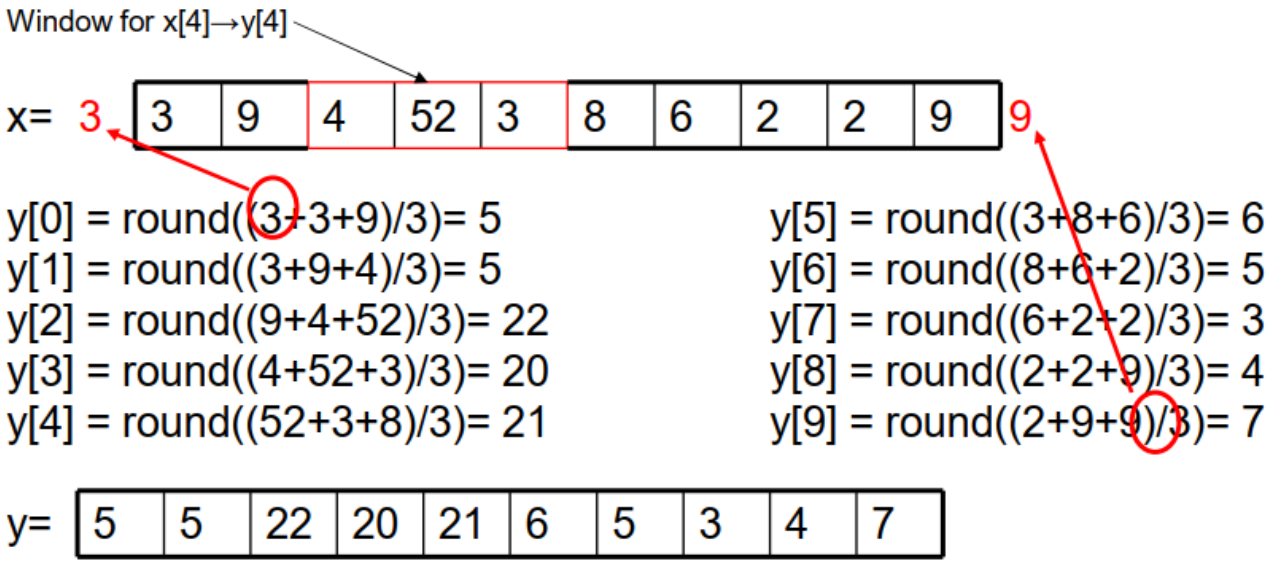
There are some potential problems:

- A single pixel with a very unrepresentative value can significantly affect the average value of all the pixels in its neighborhood.
- When the filter neighborhood straddles an edge, the filter will interpolate new values for pixels on the edge and so will blur that edge. This may be a problem if sharp edges are required in the output.

Example

The following example shows the application of an average filter to a simple one dimensional signal.

A window size of three is used, with one entry immediately preceding and following each entry.



2D Average filtering

- Consider the following 3 by 3 average filter:
- We can write it mathematically as:

$$\begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix}$$

$$I\_new(x,y) = \sum_{j=-1}^{1}\sum_{i=-1}^{1} 1 \times I\_old(x+i, y+j)$$

$$I\_new\_normalized(x,y) = \frac{1}{\sum_{j=-1}^{1}\sum_{i=-1}^{1} 1} \sum_{j=-1}^{1}\sum_{i=-1}^{1} 1 \times I\_old(x+i, y+j)$$

Example

2D Average filtering example using a 3 x 3 sampling window. Keeping border values unchanged

Average = round(1+4+0+2+2+4+1+0+1)/9 = 2

Input

1	4	0	1	3	1
2	2	4	2	2	3
1	0	1	0	1	0
1	2	1	0	2	2
2	5	3	1	2	5
1	1	4	2	3	0

Output

1	4	0	1	3	1
2	2	2	2	1	3
1	2	1	1	1	0
1	2	1	1	1	2
2	2	2	2	2	5
1	1	4	2	3	0

2D Average filtering example using a 3 x 3 sampling window. Extending border values outside with values at boundary

Input

Average = round(1+4+0+1+4+0+2+2+4)/9 = 2

1	1	4	0	1	3	1	1
1	1	4	0	1	3	1	1
2	2	2	4	2	2	3	3
1	1	0	1	0	1	0	0
1	1	2	1	0	2	2	2
2	2	5	3	1	2	5	5
1	1	1	4	2	3	0	0
1	1	1	4	2	3	0	0

Output

2	2	2	2	2	2
2	2	2	2	1	2
1	2	1	1	1	2
2	2	1	1	1	2
2	2	2	2	2	2
2	2	3	3	2	2

2D Median filtering example using a 3 x 3 sampling window. Extending border values outside with 0s (Zero-padding)

Input

Average = round(2+5+0+3+0+0+0+0+0)/9 = 1

0	0	0	0	0	0	0	0
0	1	4	0	1	3	1	0
0	2	2	4	2	2	3	0
0	1	0	1	0	1	0	0
0	1	2	1	0	2	2	0
0	2	5	3	1	2	5	0
0	1	1	4	2	3	0	0
0	0	0	0	0	0	0	0

Output

1	1	1	1	1	1
1	2	2	2	1	1
1	2	1	1	1	1
1	2	1	1	1	1
1	2	2	2	2	2
1	2	2	2	1	1

3. Gaussian Filter

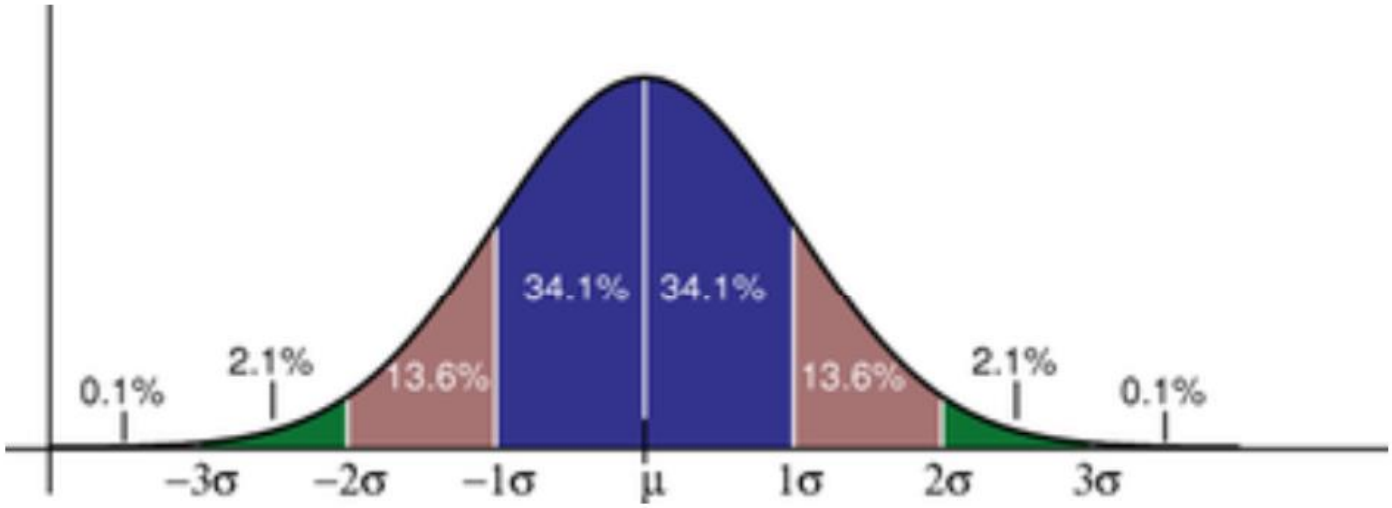
Gaussian filtering is used to blur images and remove noise and detail.

In one dimension, the Gaussian function is:

$$G(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{x^2}{2\sigma^2}}$$

where σ is the standard deviation of the distribution The distribution is assumed to have a mean of 0.

The Standard deviation of the Gaussian function plays an important role in its behavior.

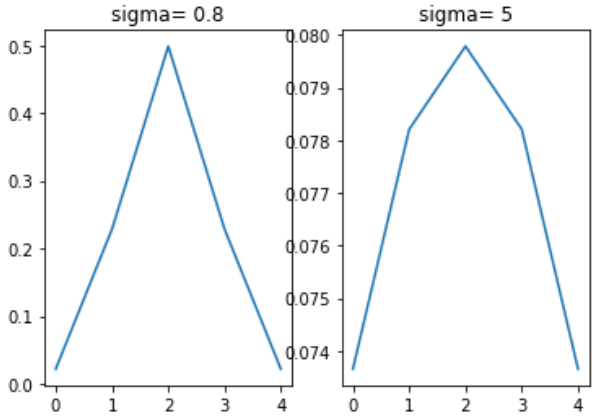


Implementation

```
import numpy as np
import matplotlib.pyplot as plt

sigma1 = 0.8
sigma2 = 5
globalsize = 5

def gaussian_kernel1d(size,sigma):
    filter_range = np.linspace(-int(size/2),int(size/2),size)
    gaussian_filter = [1 / (sigma * np.sqrt(2*np.pi)) * np.exp(-
x**2/(2*sigma**2)) for x in filter_range]
    return gaussian_kernel

fig,ax = plt.subplots(1,2)

filter1 = gaussian_kernel1d(size=globalsize,sigma=sigma1)
print(filter1)
ax[0].plot(filter1)
ax[0].set_title(f'sigma= {sigma1}')

filter2 = gaussian_kernel1d(size=globalsize,sigma=sigma2)
ax[1].plot(filter2)
ax[1].set_title(f'sigma= {sigma2}')
plt.show()
```

Implementation

```
[0.02191037561696068, 0.22831135673627742, 0.49867785050179086, 0.22831135673627742, 0.02191037561696068]
```



$$G_{1D} = \frac{1}{4}[1 \quad 2 \quad 1]$$

$$G_{2D} = G_{1DT} * G_{1D} = \frac{1}{4}[1 \quad 2 \quad 1]^T * \frac{1}{4}[1 \quad 2 \quad 1]$$

$$G_{2D} = \frac{1}{16}\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$$

4. Sobel Filter

The Sobel filter is used for edge detection.

It works by calculating the gradient of image intensity at each pixel within the image. It finds the direction of the largest increase from light to dark and the rate of change in that direction.

The result shows how abruptly or smoothly the image changes at each pixel, and therefore how likely it is that that pixel represents an edge. It also shows how that edge is likely to be oriented.

The result of applying the filter to a pixel in a region of constant intensity is a zero vector.

The result of applying it to a pixel on an edge is a vector that points across the edge from darker to brighter values.

Edge Detection

What is an edge:

- A location in the image where is a sudden change in the intensity/colour of pixels.
- A transition between objects or object and background.
- From a human visual perception perspective it attracts attention.

Problem: *Images contain noise, which also generates sudden transitions of pixel values*.

Usually there are three steps in the edge detection process:

1) Noise reduction

Suppress as much noise as possible without removing edges.

2) Edge enhancement

Highlight edges and weaken elsewhere (high pass filter).

3) Edge localization

Look at possible edges (maxima of output from previous filter) and eliminate spurious edges (often noise related).

Edge Detection

Gradient Estimation

Estimation of the intensity gradient at a pixel in the x and y direction, for an image f, is given by:

$$\frac{\partial f}{\partial x} = f(x+1, y) - f(x-1, y)$$

$$\frac{\partial f}{\partial y} = f(x, y+1) - f(x, y-1)$$

The gradient calculation (g_x,g_y) can be expressed as:

$$g_x = h_x * f(x, y)$$

$$g_y = h_y * f(x, y)$$

Sobel Filter

The Sobel filter uses two 3 x 3 kernels. One for changes in the horizontal direction, and one for changes in the vertical direction.

The two kernels are convolved with the original image to calculate the approximations of the derivatives.

If we define Gx and Gy as two images that contain the horizontal and vertical derivative approximations respectively, the computations are:

$$G_x = \begin{pmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{pmatrix} * A \quad \text{and} \quad G_y = \begin{pmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{pmatrix} * A$$

where A is the original source image.

Sobel Filter

To compute Gx and Gy we move the appropriate kernel (window) over the input image, computing the value for one pixel and then shifting one pixel to the right. Once the end of the row is reached, we move down to the beginning of the next row.

The example below shows the calculation of a value of Gx:

a11	a12	a13	…	
a21	a22	a23	…	
a31	a32	a33	…	
…	…	…	…	

Input image

kernel =

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

b11	b12	b13	…	
b21	b22	b23	…	
b31	b32	b33	…	
…	…	…	…	

Output image (G_x)

$$b_{22} = a_{13} - a_{11} + 2a_{23} - 2a_{21} + a_{33} - a_{31}$$

Example

Convolve the Sobel kernels to the original image (use 0 padding)

10	50	10	50	10
10	55	10	55	10
10	65	10	65	10
10	50	10	50	10
10	55	10	55	10

Original image

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

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

155	0	0	0	-155
225	0	0	0	-225
235	0	0	0	-235
220	0	0	0	-220
160	0	0	0	-160

-75	-130	-130	-130	-75
-15	-30	-10	-30	-15
5	10	10	10	5
10	20	20	20	10
70	120	120	120	70

Compute Gx and Gy, gradients of the image performing the convolution of Sobel kernels with the image. Use zero-padding to extend the image

0	0	10	10	10
0	0	10	10	10
0	0	10	10	10
0	0	10	10	10
0	0	10	10	10

y

x

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

h_x

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

h_y

G_x

0	30	30	0	-30
0	40	40	0	-40
0	40	40	0	-40
0	40	40	0	-40
0	30	30	0	-30

G_y

-10	-30	-40	-30	-10
0	0	0	0	0
0	0	0	0	0
0	0	0	0	0
10	30	40	30	10

Compute Gx and Gy, gradients of the image performing the convolution of Sobel kernels with the image. Use border values to extend the image.

0	0	10	10	10
0	0	10	10	10
0	0	10	10	10
0	0	10	10	10
0	0	10	10	10

y

x

G_x

0	40	40	0	0
0	40	40	0	0
0	40	40	0	0
0	40	40	0	0
0	40	40	0	0

G_y

0	0	0	0	0
0	0	0	0	0
0	0	0	0	0
0	0	0	0	0
0	0	0	0	0

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

h_x

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

h_y

$$\Theta = \arctan\left(\frac{G_y}{G_x}\right)$$

	0	0		
	0	0		
	0	0		
	0	0		
	0	0		

5. Prewitt Filter

The Prewitt filter is similar to the Sobel in that it uses two 3 x 3 kernels. One for changes in the horizontal direction, and one for changes in the vertical direction.

The two kernels are convolved with the original image to calculate the approximations of the derivatives.

If we define Gx and Gy as two images that contain the horizontal and vertical derivative approximations respectively, the computations are:

$$G_x = \begin{pmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{pmatrix} * A \qquad \text{and} \qquad G_y = \begin{pmatrix} -1 & -1 & -1 \\ 0 & 0 & 0 \\ 1 & 1 & 1 \end{pmatrix} * A$$

where A is the original source image.

Prewitt Filter

To compute Gx and Gy we move the appropriate kernel (window) over the input image, computing the value for one pixel and then shifting one pixel to theright. Once the end of the row is reached, we move down to the beginning of the next row.

The example below shows the calculation of a value of Gx:

a11	a12	a13	…	
a21	a22	a23	…	
a31	a32	a33	…	
…	…	…	…	

Input image

kernel =

1	0	-1
1	0	-1
1	0	-1

b11	b12	b13	…	
b21	b22	b23	…	
b31	b32	b33	…	
…	…	…	…	

Output image (G_x)

$$b_{22} = -a_{11} + a_{13} - a_{21} + a_{23} - a_{31} + a_{33}$$

Example

Convolve the Prewitt kernels to the original image (0 padding)

10	50	10	50	10
10	55	10	55	10
10	65	10	65	10
10	50	10	50	10
10	55	10	55	10

Original image

1	0	-1
1	0	-1
1	0	-1

-1	-1	-1
0	0	0
1	1	1

105	0	0	0	-105
170	0	0	0	-170
170	0	0	0	-170
170	0	0	0	-170
105	0	0	0	-105

-65	-75	-120	-75	-65
-15	-15	-30	-15	-15
5	5	10	5	5
10	10	20	10	10
50	70	110	75	65


Any Questions?