IMAGE FILTRATION I

Ole-Johan Skrede

22.02.2017

INF2310 - Digital Image Processing

Department of Informatics
The Faculty of Mathematics and Natural Sciences
University of Oslo

After original slides by Fritz Albregtsen

MESSAGES

- The first mandatory assignment:
  - Tentative posting date: Wednesday 01.03.2017
  - Tentative submission deadline: Friday 17.03.2017

TODAY'S LECTURE

- Neighbourhood operations

TODAY'S LECTURE

- Neighbourhood operations
- Convolution and correlation

- Neighbourhood operations
- Convolution and correlation
- Low pass filtering

TODAY'S LECTURE

- Neighbourhood operations
- Convolution and correlation
- Low pass filtering
- Sections in Gonzales & Woods:
  - 2.6.2: Linear versus Nonlinear Operations
  - 3.1: Background
  - 3.4: Fundamentals of Spatial Filtering
  - 3.5: Smoothing spatial Filtering
  - 5.3: Restoration in the Presence of Noise Only — Spatial Filtering
  - 12.2.1: Matching by correlation

IMAGE FILTERING

OVERVIEW

- A general tool for processing digital images.

OVERVIEW

- A general tool for processing digital images.
- One of the mostly used operations in image processing.

OVERVIEW

- A general tool for processing digital images.
- One of the mostly used operations in image processing.
- Typical utilities:
    - Image improvement
    - Image analysis
    - Remove or reduce noise
    - Improve percieved sharpness
    - Highlight edges
    - Highlight texture

- The filter (or filter kernel) is defined by a matrix, e.g.

$$w = \begin{bmatrix} w[-1,-1] & w[-1,0] & w[-1,1] \\ w[0,-1] & w[0,0] & w[0,1] \\ w[1,-1] & w[1,0] & w[1,1] \end{bmatrix}$$

- The filter (or filter kernel) is defined by a matrix, e.g.

$$w = \begin{bmatrix} w[-1,-1] & w[-1,0] & w[-1,1] \\ w[0,-1] & w[0,0] & w[0,1] \\ w[1,-1] & w[1,0] & w[1,1] \end{bmatrix}$$

- Filter kernels are often square with odd side lenghts.

- The filter (or filter kernel) is defined by a matrix, e.g.

$$w = \begin{bmatrix} w[-1,-1] & w[-1,0] & w[-1,1] \\ w[0,-1] & w[0,0] & w[0,1] \\ w[1,-1] & w[1,0] & w[1,1] \end{bmatrix}$$

- Filter kernels are often square with odd side lenghts.
- In this lecture, we will *always* assume odd side lengths unless otherwise is specified.

- The filter (or filter kernel) is defined by a matrix, e.g.

$$w = \begin{bmatrix} w[-1,-1] & w[-1,0] & w[-1,1] \\ w[0,-1] & w[0,0] & w[0,1] \\ w[1,-1] & w[1,0] & w[1,1] \end{bmatrix}$$

- Filter kernels are often square with odd side lenghts.
- In this lecture, we will *always* assume odd side lengths unless otherwise is specified.
- As with images, we index the filter as $w[x, y]$, where the positive $x-$axis is downward and the positive $y-$axis is to the right.

- The filter (or filter kernel) is defined by a matrix, e.g.

$$w = \begin{bmatrix} w[-1,-1] & w[-1,0] & w[-1,1] \\ w[0,-1] & w[0,0] & w[0,1] \\ w[1,-1] & w[1,0] & w[1,1] \end{bmatrix}$$

- Filter kernels are often square with odd side lenghts.
- In this lecture, we will *always* assume odd side lengths unless otherwise is specified.
- As with images, we index the filter as $w[x, y]$, where the positive $x-$axis is downward and the positive $y-$axis is to the right.
- The origin of the filter is at the filter center.

- The filter (or filter kernel) is defined by a matrix, e.g.

$$w = \begin{bmatrix} w[-1,-1] & w[-1,0] & w[-1,1] \\ w[0,-1] & w[0,0] & w[0,1] \\ w[1,-1] & w[1,0] & w[1,1] \end{bmatrix}$$

- Filter kernels are often square with odd side lenghts.
- In this lecture, we will *always* assume odd side lengths unless otherwise is specified.
- As with images, we index the filter as $w[x,y]$, where the positive $x-$axis is downward and the positive $y-$axis is to the right.
- The origin of the filter is at the filter center.
- We use the names *filter* and *filter kernel* interchangeably. Other names are *filter mask* and *filter matrix*.

- The filter (or filter kernel) is defined by a matrix, e.g.

$$w = \begin{bmatrix} w[-1,-1] & w[-1,0] & w[-1,1] \\ w[0,-1] & w[0,0] & w[0,1] \\ w[1,-1] & w[1,0] & w[1,1] \end{bmatrix}$$

- Filter kernels are often square with odd side lenghts.
- In this lecture, we will *always* assume odd side lengths unless otherwise is specified.
- As with images, we index the filter as $w[x, y]$, where the positive $x-$axis is downward and the positive $y-$axis is to the right.
- The origin of the filter is at the filter center.
- We use the names *filter* and *filter kernel* interchangeably. Other names are *filter mask* and *filter matrix*.
- The result of the filtering is determined by the size of the filter and the values in the filter.

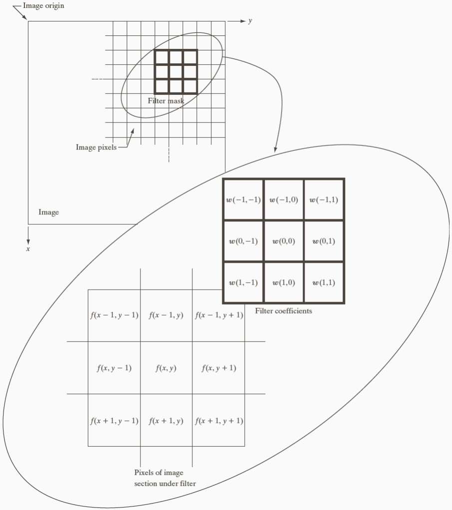


Figure 1: Example of an image and a 3×3 filter kernel.

CONVOLUTION— INTRODUCTION AND EXAMPLE

In image analysis[1], convolution is a binary operation, taking an image f and a filter (also an image) w, and producing an image g. We use an asterisk $*$ to denote this operation

$$f * w = g.$$

[1]This is really just an ordinary *discrete convolution*, the discrete version of a *continuous convolution*.

In image analysis[1], convolution is a binary operation, taking an image f and a filter (also an image) w, and producing an image g. We use an asterisk $*$ to denote this operation

$$f * w = g.$$

For an element $[x, y]$, the operation is defined as

$$g[x,y] = (f * w)[x,y] \coloneqq \sum_{s=-S}^{S} \sum_{t=-T}^{T} f[x-s, y-t]w[s,t].$$

[1]This is really just an ordinary *discrete convolution*, the discrete version of a *continuous convolution*.

Let us walk through a small example, step by step

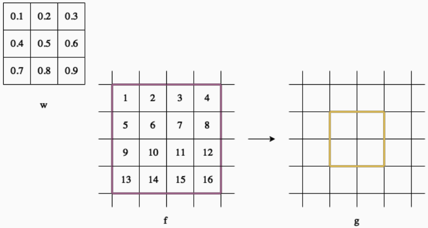


Figure 2: Extract of an image f, a 3×3 filter kernel with values, and a blank result image g. The colored squares indicate which elements will be affected by the convolution.

Notice that we are in the interior of the image, this is because boundaries require some extra attention. We will deal with boundary conditions later.

First, our indices (x, y), will be as indicated by the figure, and we will only affect values inside the coloured squares. In this example. $S = 1$ and $T = 1$.

$$g[x,y] = (f * w)[x,y] := \sum_{s=-1}^{1} \sum_{t=-1}^{1} f[x-s, y-t]w[s,t].$$

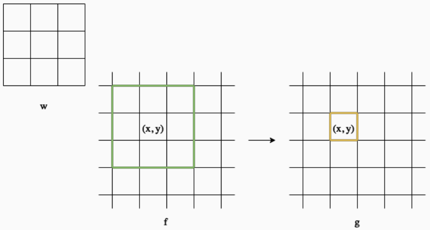
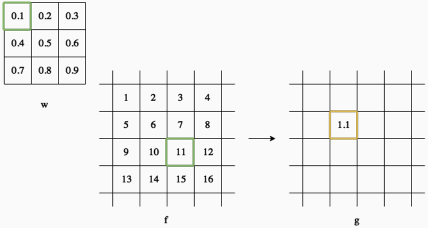


Figure 3: Locations in first convolution.

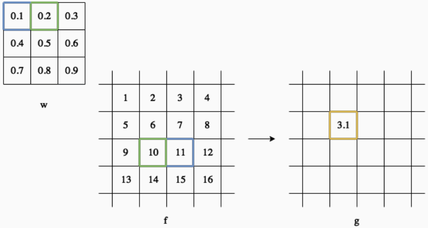
CONVOLUTION 1 — STEP 1: $s = -1, t = -1$



$$g[x,y] = \sum_{s=-1}^{1} \sum_{t=-1}^{1} f[x-s, y-t]w[s,t]$$

$$g[x,y] = 0.1 \cdot 11$$
$$= 1.1$$

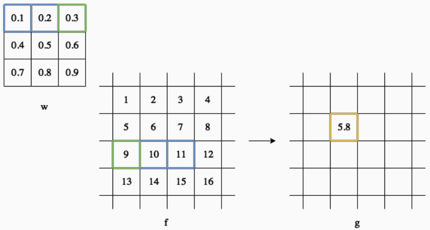
CONVOLUTION 1 — STEP 2: $s = -1, t = 0$



$$g[x,y] = \sum_{s=-1}^{1} \sum_{t=-1}^{1} f[x-s, y-t]w[s,t]$$

$$\begin{aligned} g[x,y] &= 0.1 \cdot 11 + 0.2 \cdot 10 \\ &= 3.1 \end{aligned}$$

CONVOLUTION 1 — STEP 3: $s = -1, t = 1$

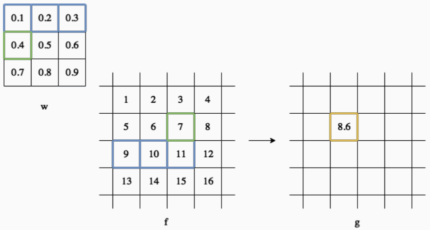


$$g[x,y] = \sum_{s=-1}^{1} \sum_{t=-1}^{1} f[x-s, y-t]w[s,t]$$

$$g[x,y] = 0.1 \cdot 11 + 0.2 \cdot 10 + 0.3 \cdot 9$$

$$= 5.8$$

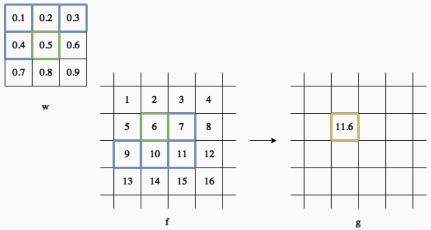
CONVOLUTION 1 — STEP 4: $s = 0, t = -1$



$$g[x,y] = \sum_{s=-1}^{1} \sum_{t=-1}^{1} f[x-s, y-t]w[s,t]$$

$$\begin{aligned} g[x,y] &= 0.1 \cdot 11 + 0.2 \cdot 10 + 0.3 \cdot 9 \\ &+ 0.4 \cdot 7 \\ &= 8.6 \end{aligned}$$

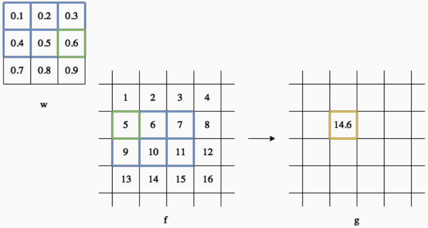
CONVOLUTION 1 — STEP 5: $s = 0, t = 0$



$$g[x,y] = \sum_{s=-1}^{1} \sum_{t=-1}^{1} f[x-s, y-t]w[s,t]$$

$$\begin{aligned} g[x,y] &= 0.1 \cdot 11 + 0.2 \cdot 10 + 0.3 \cdot 9 \\ &+ 0.4 \cdot 7 + 0.5 \cdot 6 \\ &= 11.6 \end{aligned}$$

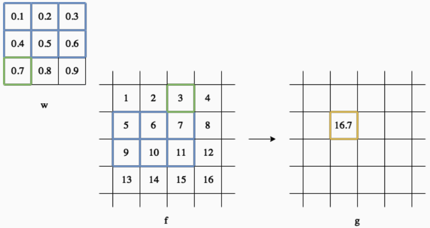
CONVOLUTION 1 — STEP 6: $s = 0, t = 1$



$$g[x,y] = \sum_{s=-1}^{1} \sum_{t=-1}^{1} f[x-s, y-t]w[s,t]$$

$$\begin{aligned} g[x,y] &= 0.1 \cdot 11 + 0.2 \cdot 10 + 0.3 \cdot 9 \\ &+ 0.4 \cdot 7 + 0.5 \cdot 6 + 0.6 \cdot 5 \\ &= 14.6 \end{aligned}$$

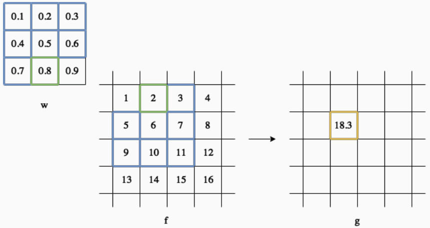
CONVOLUTION 1 — STEP 7: $s = 1, t = -1$



$$g[x,y] = \sum_{s=-1}^{1} \sum_{t=-1}^{1} f[x-s, y-t]w[s,t]$$

$$\begin{aligned} g[x,y] &= 0.1 \cdot 11 + 0.2 \cdot 10 + 0.3 \cdot 9 \\ &+ 0.4 \cdot 7 + 0.5 \cdot 6 + 0.6 \cdot 5 \\ &+ 0.7 \cdot 3 \\ &= 16.7 \end{aligned}$$

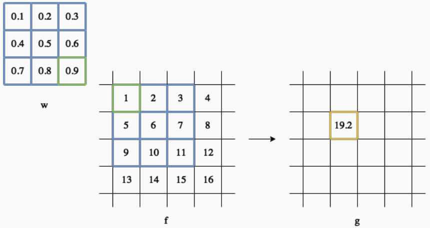
CONVOLUTION 1 — STEP 8: $s = 1, t = 0$



$$g[x,y] = \sum_{s=-1}^{1} \sum_{t=-1}^{1} f[x-s, y-t]w[s,t]$$

$$\begin{aligned} g[x,y] &= 0.1 \cdot 11 + 0.2 \cdot 10 + 0.3 \cdot 9 \\ &+ 0.4 \cdot 7 + 0.5 \cdot 6 + 0.6 \cdot 5 \\ &+ 0.7 \cdot 3 + 0.8 \cdot 2 \\ &= 18.3 \end{aligned}$$

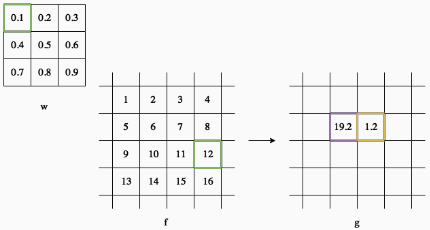
CONVOLUTION 1 — STEP 9: $s = 1, t = 1$



$$g[x,y] = \sum_{s=-1}^{1} \sum_{t=-1}^{1} f[x-s, y-t]w[s,t]$$

$$\begin{aligned} g[x,y] &= 0.1 \cdot 11 + 0.2 \cdot 10 + 0.3 \cdot 9 \\ &+ 0.4 \cdot 7 + 0.5 \cdot 6 + 0.6 \cdot 5 \\ &+ 0.7 \cdot 3 + 0.8 \cdot 2 + 0.9 \cdot 1 \\ &= 19.2 \end{aligned}$$

CONVOLUTION 2 — STEP 1: $s = -1, t = -1$

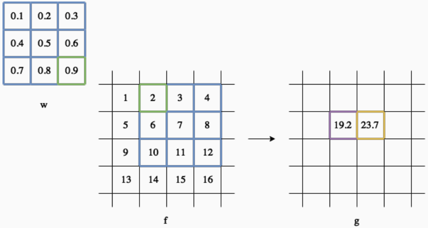


$$g[x,y] = \sum_{s=-1}^{1} \sum_{t=-1}^{1} f[x-s, y-t]w[s,t]$$

$$g[x,y] = 0.1 \cdot 12$$

$$= 1.2$$

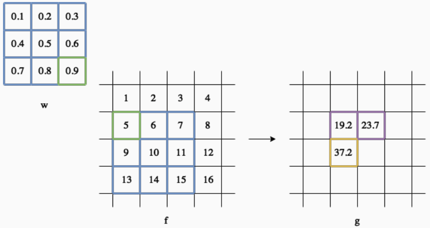
CONVOLUTION 2 — STEP 9: $s = 1, t = 1$



$$g[x, y] = \sum_{s=-1}^{1} \sum_{t=-1}^{1} f[x - s, y - t] w[s, t]$$

$$\begin{aligned} g[x, y] &= 0.1 \cdot 12 + 0.2 \cdot 11 + 0.3 \cdot 10 \\ &+ 0.4 \cdot 8 + 0.5 \cdot 7 + 0.6 \cdot 6 \\ &+ 0.7 \cdot 4 + 0.8 \cdot 3 + 0.9 \cdot 2 \\ &= 23.7 \end{aligned}$$

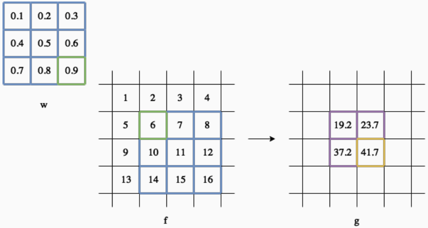
CONVOLUTION 3 — STEP 9: $s = 1, t = 1$



$$g[x,y] = \sum_{s=-1}^{1} \sum_{t=-1}^{1} f[x-s, y-t]w[s,t]$$

$$\begin{aligned} g[x,y] &= 0.1 \cdot 15 + 0.2 \cdot 14 + 0.3 \cdot 13 \\ &+ 0.4 \cdot 11 + 0.5 \cdot 10 + 0.6 \cdot 9 \\ &+ 0.7 \cdot 7 + 0.8 \cdot 6 + 0.9 \cdot 5 \\ &= 37.2 \end{aligned}$$

CONVOLUTION 4 — STEP 9: $s = 1, t = 1$



$$g[x,y] = \sum_{s=-1}^{1} \sum_{t=-1}^{1} f[x-s, y-t]w[s,t]$$

$$\begin{aligned} g[x,y] &= 0.1 \cdot 16 + 0.2 \cdot 15 + 0.3 \cdot 14 \\ &+ 0.4 \cdot 12 + 0.5 \cdot 11 + 0.6 \cdot 10 \\ &+ 0.7 \cdot 8 + 0.8 \cdot 7 + 0.9 \cdot 6 \\ &= 41.7 \end{aligned}$$

USEFUL MENTAL IMAGE

A useful way of thinking about convolution is to

A useful way of thinking about convolution is to

1. Rotate the filter 180 degrees.

USEFUL MENTAL IMAGE

A useful way of thinking about convolution is to

1. Rotate the filter 180 degrees.
2. Place it on top of the image in the image's top left corner.

USEFUL MENTAL IMAGE

A useful way of thinking about convolution is to

1. Rotate the filter 180 degrees.
2. Place it on top of the image in the image's top left corner.
3. "Slide" it across each column until you hit the right boundary of the image.

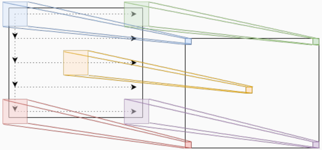
USEFUL MENTAL IMAGE

A useful way of thinking about convolution is to

1. Rotate the filter 180 degrees.
2. Place it on top of the image in the image's top left corner.
3. ”Slide” it across each column until you hit the right boundary of the image.
4. Start from the left again, but now, one row below the previous.

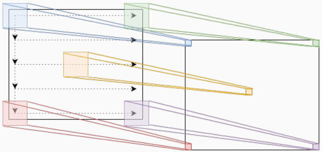
A useful way of thinking about convolution is to

1. Rotate the filter 180 degrees.
2. Place it on top of the image in the image's top left corner.
3. "Slide" it across each column until you hit the right boundary of the image.
4. Start from the left again, but now, one row below the previous.
5. Repeat step 3. followed by 4. until you have covered the whole image.



A useful way of thinking about convolution is to

1. Rotate the filter 180 degrees.
2. Place it on top of the image in the image's top left corner.
3. "Slide" it across each column until you hit the right boundary of the image.
4. Start from the left again, but now, one row below the previous.
5. Repeat step 3. followed by 4. until you have covered the whole image.



Also, check out this nice interactive visualization:
`http://setosa.io/ev/image-kernels/`

BOUNDARY TREATMENT

DIFFERENT MODES

We differentiate between three different *modes* when we filter one image with another:

We differentiate between three different *modes* when we filter one image with another:

Full: We get an output response at every point of overlap between the image and the filter.

We differentiate between three different *modes* when we filter one image with another:

Full: We get an output response at every point of overlap between the image and the filter.

Same: The origin of the filter is always inside the image, and we *pad* the image in order to preserve the image size in the result.

We differentiate between three different *modes* when we filter one image with another:

Full: We get an output response at every point of overlap between the image and the filter.

Same: The origin of the filter is always inside the image, and we *pad* the image in order to preserve the image size in the result.

Valid: We only record a response as long as there is full overlap between the image and the *whole* filter.

This can be written as

$$(f * w)[x, y] = \sum_{s=-\infty}^{\infty} \sum_{t=-\infty}^{\infty} f[x-s, y-t]w[s,t]$$

as long as

$$(x - s) \text{ is a row in } f \text{ and } s \text{ is a row in } w \tag{1}$$

for *some* s, and

$$(y - t) \text{ is a column in } f \text{ and } t \text{ is a column in } w \tag{2}$$

for *some* t.

If f is of size[1] $M \times N$ and w of size $P \times Q$, assuming $M \geq P$ and $N \geq Q$, the output image will be of size $M + P - 1 \times N + Q - 1$. That is,

$$\text{Number of rows: } M + P - 1.$$

$$\text{Number of columns: } N + Q - 1.$$

[1]We use the $rows \times columns$ convention when describing the size of an image.

FULL MODE — EXAMPLE

$$\begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{bmatrix} \overset{\text{full}}{*} \begin{bmatrix} 0.1 & 0.1 & 0.1 \\ 0.1 & 0.1 & 0.1 \\ 0.1 & 0.1 & 0.1 \end{bmatrix} = \begin{bmatrix} 0.1 & 0.3 & 0.6 & 0.9 & 0.7 & 0.4 \\ 0.6 & 1.4 & 2.4 & 3. & 2.2 & 1.2 \\ 1.5 & 3.3 & 5.4 & 6.3 & 4.5 & 2.4 \\ 2.7 & 5.7 & 9. & 9.9 & 6.9 & 3.6 \\ 2.2 & 4.6 & 7.2 & 7.8 & 5.4 & 2.8 \\ 1.3 & 2.7 & 4.2 & 4.5 & 3.1 & 1.6 \end{bmatrix}$$

This can be written as

$$(f * w)[x, y] = \sum_{s=-\infty}^{\infty} \sum_{t=-\infty}^{\infty} f[x - s, y - t]w[s, t]$$

as long as

$$(x - s) \text{ is a row in } f \text{ and } s \text{ is a row in } w \tag{3}$$

for *all* s, and

$$(y - t) \text{ is a column in } f \text{ and } t \text{ is a column in } w \tag{4}$$

for *all* t.

If f is of size $M \times N$ and w of size $P \times Q$, assuming $M \geq P$ and $N \geq Q$, the output image will be of size $M - P + 1 \times N - Q + 1$. That is

$$\text{Number of rows: } M - P + 1.$$
$$\text{Number of columns: } N - Q + 1.$$

VALID MODE — EXAMPLE

$$\begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{bmatrix} \overset{\text{valid}}{*} \begin{bmatrix} 0.1 & 0.1 & 0.1 \\ 0.1 & 0.1 & 0.1 \\ 0.1 & 0.1 & 0.1 \end{bmatrix} = \begin{bmatrix} 5.4 & 6.3 \\ 9. & 9.9 \end{bmatrix}$$

SAME MODE

This is the same as *valid mode*, except that we *pad* the input image (add values outside the original boundary) such that the output size is the same as the original image.

SAME MODE

This is the same as *valid mode*, except that we *pad* the input image (add values outside the original boundary) such that the output size is the same as the original image.

We will cover three types of paddings, and use a 1D example to illustrate. In the 1D example, we assume an original signal of length 5, and we pad with 2 on each side, furthermore we use ”|” to indicate the original boundary.

This is the same as *valid mode*, except that we *pad* the input image (add values outside the original boundary) such that the output size is the same as the original image.
We will cover three types of paddings, and use a 1D example to illustrate. In the 1D example, we assume an original signal of length 5, and we pad with 2 on each side, furthermore we use "|" to indicate the original boundary.

- Zero padding
  $[0, 0|1, 2, 3, 4, 5|0, 0]$

SAME MODE

This is the same as *valid mode*, except that we *pad* the input image (add values outside the original boundary) such that the output size is the same as the original image.
We will cover three types of paddings, and use a 1D example to illustrate. In the 1D example, we assume an original signal of length 5, and we pad with 2 on each side, furthermore we use ”|” to indicate the original boundary.

- Zero padding
  $[0, 0|1, 2, 3, 4, 5|0, 0]$
- Symmetrical padding
  $[2, 1|1, 2, 3, 4, 5|5, 4]$

This is the same as *valid mode*, except that we *pad* the input image (add values outside the original boundary) such that the output size is the same as the original image.
We will cover three types of paddings, and use a 1D example to illustrate. In the 1D example, we assume an original signal of length 5, and we pad with 2 on each side, furthermore we use "|" to indicate the original boundary.

- Zero padding
  $[0, 0|1, 2, 3, 4, 5|0, 0]$
- Symmetrical padding
  $[2, 1|1, 2, 3, 4, 5|5, 4]$
- Circular padding
  $[4, 5|1, 2, 3, 4, 5|1, 2]$

For a filter with size $P \times Q$, we must pad the image with $\frac{P-1}{2}$ rows on each side, and $\frac{Q-1}{2}$ columns on each side. We can check that this will produce an output of size $M \times N$ by calculating the output side from the valid mode:

$$\begin{aligned} M + 2\left[\frac{P-1}{2}\right] - P + 1 = M \\ N + 2\left[\frac{Q-1}{2}\right] - Q + 1 = N \end{aligned} \tag{5}$$

Can also pad with other constant values than 0.

$$\begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{bmatrix} \xrightarrow{\text{zero}} \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 2 & 3 & 4 & 0 \\ 0 & 5 & 6 & 7 & 8 & 0 \\ 0 & 9 & 10 & 11 & 12 & 0 \\ 0 & 13 & 14 & 15 & 16 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{bmatrix} \overset{\text{same}}{*} \begin{bmatrix} 0.1 & 0.1 & 0.1 \\ 0.1 & 0.1 & 0.1 \\ 0.1 & 0.1 & 0.1 \end{bmatrix} = \begin{bmatrix} 1.4 & 2.4 & 3. & 2.2 \\ 3.3 & 5.4 & 6.3 & 4.5 \\ 5.7 & 9. & 9.9 & 6.9 \\ 4.6 & 7.2 & 7.8 & 5.4 \end{bmatrix}$$

SAME MODE — SYMMETRIC PADDING

Also known as mirrored padding or reflective padding.

$$\begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{bmatrix} \xrightarrow{\text{symmetric}} \begin{bmatrix} 1 & 1 & 2 & 3 & 4 & 4 \\ 1 & 1 & 2 & 3 & 4 & 4 \\ 5 & 5 & 6 & 7 & 8 & 8 \\ 9 & 9 & 10 & 11 & 12 & 12 \\ 13 & 13 & 14 & 15 & 16 & 16 \\ 13 & 13 & 14 & 15 & 16 & 16 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{bmatrix} \overset{\text{same}}{*} \begin{bmatrix} 0.1 & 0.1 & 0.1 \\ 0.1 & 0.1 & 0.1 \\ 0.1 & 0.1 & 0.1 \end{bmatrix} = \begin{bmatrix} 2.4 & 3. & 3.9 & 4.5 \\ 4.8 & 5.4 & 6.3 & 6.9 \\ 8.4 & 9. & 9.9 & 10.5 \\ 10.8 & 11.4 & 12.3 & 12.9 \end{bmatrix}$$

Also known as wrapping.

$$\begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{bmatrix} \xrightarrow{\text{circular}} \begin{bmatrix} 16 & 13 & 14 & 15 & 16 & 13 \\ 4 & 1 & 2 & 3 & 4 & 1 \\ 8 & 5 & 6 & 7 & 8 & 5 \\ 12 & 9 & 10 & 11 & 12 & 9 \\ 16 & 13 & 14 & 15 & 16 & 13 \\ 4 & 1 & 2 & 3 & 4 & 1 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{bmatrix} \overset{\text{same}}{*} \begin{bmatrix} 0.1 & 0.1 & 0.1 \\ 0.1 & 0.1 & 0.1 \\ 0.1 & 0.1 & 0.1 \end{bmatrix} = \begin{bmatrix} 6.9 & 6.6 & 7.5 & 7.2 \\ 5.7 & 5.4 & 6.3 & 6. \\ 9.3 & 9. & 9.9 & 9.6 \\ 8.1 & 7.8 & 8.7 & 8.4 \end{bmatrix}$$

PROPERTIES, GENERAL CASE

$$
\begin{aligned}
(f * w)[x, y] &= \sum_{s=-S}^{S} \sum_{t=-T}^{T} f[x-s, y-t] w[s, t] \\
&= \sum_{s=x-S}^{x+S} \sum_{t=y-T}^{y+T} f[s, t] w[x-s, y-t]
\end{aligned}
$$

Commutative

$$f * g = g * f$$

Associative

$$(f * g) * h = f * (f * h)$$

Distributive

$$f * (g + h) = f * g + f * h$$

Associative with scalar multiplication

$$\alpha(f * g) = (\alpha f) * g = f * (\alpha g)$$

CORRELATION AND TEMPLATE MATCHING

In this case, correlation is a binary operation, taking an image f and a filter (also an image) w, and producing an image g. For an element $[x, y]$, the operation is defined as

$$g[x, y] = (f \star w)[x, y] \coloneqq \sum_{s=-S}^{S} \sum_{t=-T}^{T} f[x+s, y+t]w[s, t].$$

In this case, correlation is a binary operation, taking an image f and a filter (also an image) w, and producing an image g. For an element $[x, y]$, the operation is defined as

$$g[x, y] = (f \star w)[x, y] \coloneqq \sum_{s=-S}^{S} \sum_{t=-T}^{T} f[x+s, y+t] w[s, t].$$

- Very similar to convolution. Equivalent if the filter is symmetric.

In this case, correlation is a binary operation, taking an image f and a filter (also an image) w, and producing an image g. For an element $[x, y]$, the operation is defined as

$$g[x,y] = (f \star w)[x,y] := \sum_{s=-S}^{S} \sum_{t=-T}^{T} f[x+s, y+t]w[s,t].$$

- Very similar to convolution. Equivalent if the filter is symmetric.
- For the mental image, it is the same as convolution, without the rotation of the kernel in the beginning.

In this case, correlation is a binary operation, taking an image f and a filter (also an image) w, and producing an image g. For an element $[x, y]$, the operation is defined as

$$g[x,y] = (f \star w)[x,y] := \sum_{s=-S}^{S} \sum_{t=-T}^{T} f[x+s, y+t]w[s,t].$$

- Very similar to convolution. Equivalent if the filter is symmetric.
- For the mental image, it is the same as convolution, without the rotation of the kernel in the beginning.
- In general *not associative*, which is important in some cases.

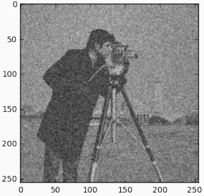
In this case, correlation is a binary operation, taking an image f and a filter (also an image) w, and producing an image g. For an element $[x, y]$, the operation is defined as

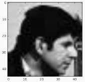
$$g[x,y] = (f \star w)[x,y] \coloneqq \sum_{s=-S}^{S} \sum_{t=-T}^{T} f[x+s, y+t]w[s,t].$$

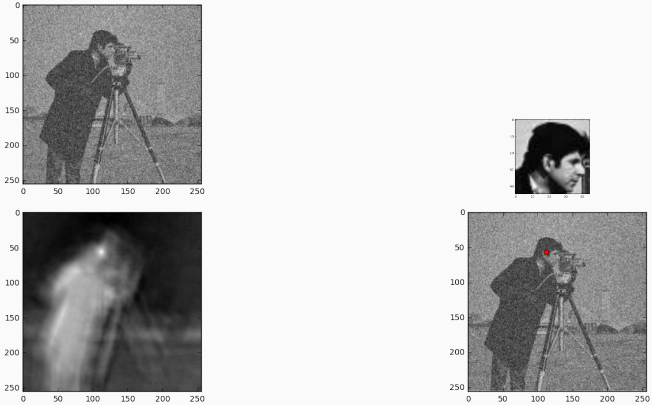
- Very similar to convolution. Equivalent if the filter is symmetric.
- For the mental image, it is the same as convolution, without the rotation of the kernel in the beginning.
- In general *not associative*, which is important in some cases.
- Less important in other cases, such as *template matching*.

- We can use correlation to match patterns in an image.

- We can use correlation to match patterns in an image.
- Remember to normalize the image and the filter with their respective means.







You can find an example implementation in python at `https://ojskrede.github.io/inf2310/lectures/week_06/` (separable_timing.ipynb).

NEIGHBOURHOOD OPERATORS

In general, we can view filtering as the application of an *operator* that computes the result image's value in each pixel (x, y) by utilizing the pixels in the input image in a *neighbourhood* around (x, y)

$$g[x, y] = T(f[\mathcal{N}(x, y)])$$

In general, we can view filtering as the application of an *operator* that computes the result image's value in each pixel (x, y) by utilizing the pixels in the input image in a *neighbourhood* around (x, y)

$$g[x, y] = T(f[\mathcal{N}(x, y)])$$

In the above, we assume that $g[x, y]$ is the value of g at location (x, y) and $f[\mathcal{N}(x, y)]$ is the value(s) in the neighbourhood $\mathcal{N}(x, y)$ of (x, y). T is some operator acting on the pixel values in the neighbourhood.

The neighbourhood of the filter gives the pixels around (x, y) in the input image that the operator (potentially) use.

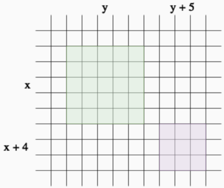


Figure 6: Neighbourhood example, a 5×5 neighbourhood centered at (x, y), and a 3×3 neighbourhood centered at $(x + 4, y + 5)$.

NEIGHBOURHOOD IN PRACTICE

- Squares and rectangles are most common.

NEIGHBOURHOOD IN PRACTICE

- Squares and rectangles are most common.
- For symmetry reasons, the height and width is often odd, making the location of the center point at a pixel.

- Squares and rectangles are most common.
- For symmetry reasons, the height and width is often odd, making the location of the center point at a pixel.
- When nothing else is specified, the origin of the neighbourhood is at the center pixel.

NEIGHBOURHOOD IN PRACTICE

- Squares and rectangles are most common.
- For symmetry reasons, the height and width is often odd, making the location of the center point at a pixel.
- When nothing else is specified, the origin of the neighbourhood is at the center pixel.
- If the neighbourhood is of size 1×1, T is grayscale transform. If T is equal over the whole image, we say that T is a *global* operator.

NEIGHBOURHOOD IN PRACTICE

- Squares and rectangles are most common.
- For symmetry reasons, the height and width is often odd, making the location of the center point at a pixel.
- When nothing else is specified, the origin of the neighbourhood is at the center pixel.
- If the neighbourhood is of size 1×1, T is grayscale transform. If T is equal over the whole image, we say that T is a *global* operator.
- If the neighbourhood size is greater than 1×1, we term T as a *local* operator (even if T is position invariant).

Neighbourhood

Define the set of pixels around (x, y) in the input image where T operates.

Neighbourhood
Define the set of pixels around (x, y) in the input image where T operates.

Operator
Defines the operation done on the values in the pixels in the neighbourhood. Also called a *transform* or an *algorithm*.

Neighbourhood
Define the set of pixels around (x, y) in the input image where T operates.

Operator
Defines the operation done on the values in the pixels in the neighbourhood. Also called a *transform* or an *algorithm*.

Filter
A filter is an operator operating on a neighbourhood, that is, two filters are equivalent *only if* both the neighbourhood and the operator are equal.

ADDITIVITY

A filter is said to be *additive* if

$$T((f_1 + f_2)[\mathcal{N}(x,y)]) = T(f_1[\mathcal{N}(x,y)]) + T(f_2[\mathcal{N}(x,y)])$$

where

- T is the operator.
- $\mathcal{N}(x,y)$ is the neighbourhood around an arbitrary pixel (x,y).
- f_1 and f_2 are arbitrary images.

HOMOGENEITY

A filter is said to be *homogeneous* if

$$T(\alpha f[\mathcal{N}(x,y)]) = \alpha T(f[\mathcal{N}(x,y)])$$

where

- T is the operator.
- $\mathcal{N}(x,y)$ is the neighbourhood around an arbitrary pixel (x,y).
- f is an arbitrary image.
- α is an arbitrary scalar value.

A filter is said to be *linear* if it is both additive and homogeneous, that is, if

$$T((\alpha f_1 + \beta f_2)[\mathcal{N}(x,y)]) = \alpha T(f_1[\mathcal{N}(x,y)]) + \beta T(f_2[\mathcal{N}(x,y)])$$

where

- T is the operator.
- $\mathcal{N}(x,y)$ is the neighbourhood around an arbitrary pixel (x,y).
- f_1 and f_2 are arbitrary images.
- α and β are arbitrary scalar values.

A filter is said to be *position invariant* if

$$T(f[\mathcal{N}(x-s, y-t)]) = g(x-s, y-t)$$

where

- T is the operator.
- $\mathcal{N}(x, y)$ is the neighbourhood around an arbitrary pixel (x, y).
- f_1 and f_2 are arbitrary images.
- $g(x, y) = T(f[\mathcal{N}(x, y)])$ for all (x, y).
- (s, t) is an arbitrary position shift.

In other words, the value of the result image at (x, y) is only dependent of the *values* in the neighbourhood of (x, y), and not dependent on the *positions*.

- A 2D filter w, is said to be *separable* if the filtration can be done with two sequential 1D filtrations, w_V and w_H.

SEPARABLE FILTERS

- A 2D filter w, is said to be *separable* if the filtration can be done with two sequential 1D filtrations, w_V and w_H.
- In other words, if $w = w_V * w_H$

- A 2D filter w, is said to be *separable* if the filtration can be done with two sequential 1D filtrations, w_V and w_H.
- In other words, if $w = w_V * w_H$
- An example is a 5×5 mean filter

$$\frac{1}{5}\begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{bmatrix} * \frac{1}{5}\begin{bmatrix} 1 & 1 & 1 & 1 & 1 \end{bmatrix} = \frac{1}{25}\begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{bmatrix}$$

SEPARABLE FILTERS ARE FASTER

- From the associativity of convolution, with f as an image and $w = w_V * w_H$, we get

$$f * w = f * (w_V * w_H) = (f * w_V) * w_H$$

SEPARABLE FILTERS ARE FASTER

- From the associativity of convolution, with f as an image and $w = w_V * w_H$, we get

$$f * w = f * (w_V * w_H) = (f * w_V) * w_H$$

- As can be seen, we can now do two convolutions with a 1D filter, in stead of one convolution with a 2D filter, that is

$$
\begin{aligned}
g &= f * w, && \text{2D} \\
&\text{vs.} \\
h &= f * w_V && \text{1D} \\
g &= h * w_H && \text{1D}
\end{aligned}
$$

Illustrative (ignores padding etc.) examples of implementations, where

- `f`, input image, size `[M, N]`
- `w`, 2D filter kernel, size `[L, L]`, `(S = T = (L - 1) / 2)`
- `w_V, w_H`, vertical filter kernels, size `[L]`
- `h`, temporary filtered image, size `[M, N]`
- `g`, filtered image, size `[M, N]`

```
# Convolution with 2D filter
g = zeros(f.shape)
for x in range(M):
  for y in range(N):
    for s in range(-S, S):
      for t in range(-T, T):
        g[x, y] += f[x - s, y - t]*w[s, t]
```

```
# Convolution with 2 1D filters
h = zeros(f.shape)
for x in range(M):
  for y in range(N):
    for s in range(-S, S):
      h[x, y] += f[x - s, y]*w_V[s]
g = zeros(f.shape)
for x in range(M):
  for y in range(N):
    for t in range(-T, T):
      g[x, y] += h[x, y - t]*w_H[t]
```

For an $M \times N$ image and a square filter kernel with sidelengths L, a (naive) $2D$ convolution implementation would have complexity of

$$\mathcal{O}(MNL^2),$$

For an $M \times N$ image and a square filter kernel with sidelengths L, a (naive) $2D$ convolution implementation would have complexity of

$$\mathcal{O}(MNL^2),$$

while a (naive) $1D$ convolution implementation would have complexity of

$$\mathcal{O}(MNL).$$

For an $M \times N$ image and a square filter kernel with sidelengths L, a (naive) $2D$ convolution implementation would have complexity of

$$\mathcal{O}(MNL^2),$$

while a (naive) $1D$ convolution implementation would have complexity of

$$\mathcal{O}(MNL).$$

So the speed up should be linear in L.

Looking back at our naive implementations, we see that in the case of the 2D filter, we have about MNL^2 multiplications and $MN(L^2 - 1)$ additions, so about

$$\text{flops}_{nonsep} = MNL^2 + MN(L^2 - 1) = MN(2L^2 - 1)$$

floating point operations (FLOPS).

Looking back at our naive implementations, we see that in the case of the 2D filter, we have about MNL^2 multiplications and $MN(L^2-1)$ additions, so about

$$\text{flops}_{nonsep} = MNL^2 + MN(L^2-1) = MN(2L^2-1)$$

floating point operations (FLOPS).
For the case with 2 1D filters, we have about $2MNL$ multiplications and $2MN(L-1)$, which is then

$$\text{flops}_{sep} = 2MNL + 2MN(L-1) = 2MN(2L-1).$$

how much faster? flops

We can get an idea of the speedup by looking at

$$\frac{\text{flops}_{nonsep}}{\text{flops}_{sep}} = \frac{2L^2 - 1}{4L - 2} \sim \frac{2L + 1}{4}$$

So the 2D case should be about $\frac{L}{2} + \frac{1}{4}$ times slower than the separable case.
For a concrete example, take a look here:
`https://ojskrede.github.io/inf2310/lectures/week_06/`

I do not like math magic in my slides, so I will now show how

$$\frac{2L^2-1}{4L-2} \sim \frac{2L+1}{4}.$$

We say that they are *asymptotically equivalent*, and write $\sim$ to symbolise this. This simply means that the one approaches the other as L increases.

$$\begin{aligned}
\frac{2L^2-1}{4L-2} &= \frac{L^2}{2L-1} - \frac{1}{4L-2} \\
&= \frac{L^2(2L+1)}{(2L-1)(2L+1)} - \frac{1}{4L-2} \\
&= \frac{L^2}{(2L)^2-1^2}(2L+1) - \frac{1}{4L-2} \\
&= \frac{L^2}{(4L^2-1)}(2L+1) - \frac{1}{4L-2} \\
&= \frac{1}{(4-\frac{1}{L^2})}(2L+1) - \frac{1}{4L-2} \\
&\xrightarrow{L\,\to\,\infty} \frac{2L+1}{4}.
\end{aligned}$$

Convolution filters with identical columns or rows can be implemented efficiently by updating already computed values.

1. Compute the response $r_{x,y}$ at a pixel (x, y).

CONVOLUTION WITH UPDATE

Convolution filters with identical columns or rows can be implemented efficiently by updating already computed values.

1. Compute the response $r_{x,y}$ at a pixel (x, y).
2. Compute the response in the next pixel from r_{xy}:

CONVOLUTION WITH UPDATE

Convolution filters with identical columns or rows can be implemented efficiently by updating already computed values.

1. Compute the response $r_{x,y}$ at a pixel (x, y).
2. Compute the response in the next pixel from r_{xy}:
    - Identical columns: $r_{x,y+1} = r_{x,y} - C_1 + C_L$, where C_1 is the "column response" from the first column when the kernel origin is at (x, y), and C_L is the "column response" from the last column when the kernel origin is at $(x, y + 1)$.

CONVOLUTION WITH UPDATE

Convolution filters with identical columns or rows can be implemented efficiently by updating already computed values.

1. Compute the response $r_{x,y}$ at a pixel (x, y).
2. Compute the response in the next pixel from r_{xy}:
   - Identical columns: $r_{x,y+1} = r_{x,y} - C_1 + C_L$, where C_1 is the "column response" from the first column when the kernel origin is at (x, y), and C_L is the "column response" from the last column when the kernel origin is at $(x, y+1)$.
   - Identical rows: $r_{x+1,y} = r_{x,y} - R_1 + R_L$, where R_1 is the "row response" from the first row when the kernel origin is at (x, y), and R_L is the "row response" from the last row when the kernel origin is at $(x+1, y)$.

Convolution filters with identical columns or rows can be implemented efficiently by updating already computed values.

1. Compute the response $r_{x,y}$ at a pixel (x, y).
2. Compute the response in the next pixel from r_{xy}:
    - Identical columns: $r_{x,y+1} = r_{x,y} - C_1 + C_L$, where C_1 is the "column response" from the first column when the kernel origin is at (x, y), and C_L is the "column response" from the last column when the kernel origin is at $(x, y + 1)$.
    - Identical rows: $r_{x+1,y} = r_{x,y} - R_1 + R_L$, where R_1 is the "row response" from the first row when the kernel origin is at (x, y), and R_L is the "row response" from the last row when the kernel origin is at $(x + 1, y)$.
3. Repeat from 2.

CONVOLUTION WITH UPDATE — ILLUSTRATION

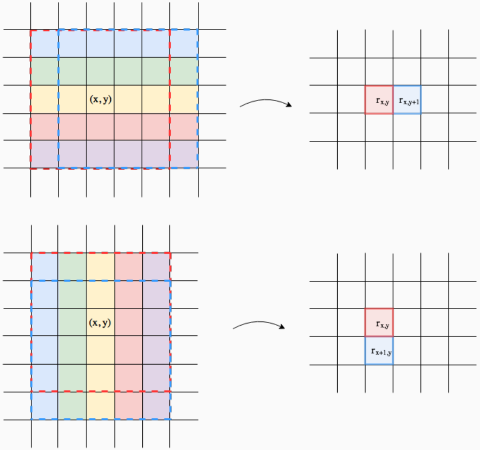


Figure 7: Top: reusing columns; bottom: reusing rows.

CONVOLUTION WITH UPDATE — SPEED GAIN

- C_1, C_L, R_1 and R_L can (individually) be computed with L multiplications and $L-1$ additions. This means that computing the new response requires $2L$ multiplications and $2L$ additions.

- C_1, C_L, R_1 and R_L can (individually) be computed with L multiplications and $L-1$ additions. This means that computing the new response requires $2L$ multiplications and $2L$ additions.
- This is as fast as separability.

CONVOLUTION WITH UPDATE — SPEED GAIN

- C_1, C_L, R_1 and R_L can (individually) be computed with L multiplications and $L-1$ additions. This means that computing the new response requires $2L$ multiplications and $2L$ additions.
- This is as fast as separability.
    - If we ignore the initialization; we must compute one full convolution to restart the updates at every new row or column.

CONVOLUTION WITH UPDATE — SPEED GAIN

- C_1, C_L, R_1 and R_L can (individually) be computed with L multiplications and $L-1$ additions. This means that computing the new response requires $2L$ multiplications and $2L$ additions.
- This is as fast as separability.
  - If we ignore the initialization; we must compute one full convolution to restart the updates at every new row or column.
  - All filters that can be updated in this way, is also separable. Conversly, separable filters need not be ”updatable”.

- C_1, C_L, R_1 and R_L can (individually) be computed with L multiplications and $L-1$ additions. This means that computing the new response requires $2L$ multiplications and $2L$ additions.
- This is as fast as separability.
  - If we ignore the initialization; we must compute one full convolution to restart the updates at every new row or column.
  - All filters that can be updated in this way, is also separable. Conversly, separable filters need not be ”updatable”.
  - Can be combined with separability when a 1D filter is uniform.
- Uniform filters can be computed even faster.

CONVOLUTION WITH UPDATE — SPEED GAIN

- C_1, C_L, R_1 and R_L can (individually) be computed with L multiplications and $L-1$ additions. This means that computing the new response requires $2L$ multiplications and $2L$ additions.
- This is as fast as separability.
  - If we ignore the initialization; we must compute one full convolution to restart the updates at every new row or column.
  - All filters that can be updated in this way, is also separable. Conversly, separable filters need not be ”updatable”.
  - Can be combined with separability when a 1D filter is uniform.
- Uniform filters can be computed even faster.
  - Each update only needs 2 subtractions and 1 addition.

CONVOLUTION WITH UPDATE — SPEED GAIN

- C_1, C_L, R_1 and R_L can (individually) be computed with L multiplications and $L-1$ additions. This means that computing the new response requires $2L$ multiplications and $2L$ additions.
- This is as fast as separability.
    - If we ignore the initialization; we must compute one full convolution to restart the updates at every new row or column.
    - All filters that can be updated in this way, is also separable. Conversly, separable filters need not be ”updatable”.
    - Can be combined with separability when a 1D filter is uniform.
- Uniform filters can be computed even faster.
    - Each update only needs 2 subtractions and 1 addition.
    - Only filters that are proportional to the mean value filter are uniform.

LOW-PASS FILTERS

Low-pass filters

Lets through low frequencies, stops high frequencies. ”Blurs” the image.

Low-pass filters
: Lets through low frequencies, stops high frequencies. ”Blurs” the image.

High-pass filters
: Lets through high frequencies, stops low frequencies. ”Sharpens” the image.

Low-pass filters
Lets through low frequencies, stops high frequencies. ”Blurs” the image.

High-pass filters
Lets through high frequencies, stops low frequencies. ”Sharpens” the image.

Band-pass filters
Lets through frequencies in a certain range.

Low-pass filters
: Lets through low frequencies, stops high frequencies. ”Blurs” the image.

High-pass filters
: Lets through high frequencies, stops low frequencies. ”Sharpens” the image.

Band-pass filters
: Lets through frequencies in a certain range.

Feauture detection filters
: Used for detection of features in an image. Features such as edges, corners and texture.

- Lets through low frequencies (large trends and slow variations in an image).

LOW-PASS FILTER — OVERVIEW

- Lets through low frequencies (large trends and slow variations in an image).
- Stops or damps high frequencies (noise, details and sharp edges in an image).

LOW-PASS FILTER — OVERVIEW

- Lets through low frequencies (large trends and slow variations in an image).
- Stops or damps high frequencies (noise, details and sharp edges in an image).
- We will learn more about frequencies in the lectures about Fourier transforms.

LOW-PASS FILTER — OVERVIEW

- Lets through low frequencies (large trends and slow variations in an image).
- Stops or damps high frequencies (noise, details and sharp edges in an image).
- We will learn more about frequencies in the lectures about Fourier transforms.
- The resulting image is a ”smeared”, ”smoothed” or ”blurred” version of the original image.

- Lets through low frequencies (large trends and slow variations in an image).
- Stops or damps high frequencies (noise, details and sharp edges in an image).
- We will learn more about frequencies in the lectures about Fourier transforms.
- The resulting image is a ”smeared”, ”smoothed” or ”blurred” version of the original image.
- Typical applications: Remove noise, locate larger objects in an image.

LOW-PASS FILTER — OVERVIEW

- Lets through low frequencies (large trends and slow variations in an image).
- Stops or damps high frequencies (noise, details and sharp edges in an image).
- We will learn more about frequencies in the lectures about Fourier transforms.
- The resulting image is a ”smeared”, ”smoothed” or ”blurred” version of the original image.
- Typical applications: Remove noise, locate larger objects in an image.
- Challenging to preserve edges.

MEAN VALUE FILTER

- Computes the mean value in the neighbourhood.

MEAN VALUE FILTER

- Computes the mean value in the neighbourhood.
    - All weights (values in the filter) are equal.

MEAN VALUE FILTER

- Computes the mean value in the neighbourhood.
  - All weights (values in the filter) are equal.
  - The sum of the weights is equal to 1.

MEAN VALUE FILTER

- Computes the mean value in the neighbourhood.
  - All weights (values in the filter) are equal.
  - The sum of the weights is equal to 1.
- The neighbourhood size determines the ”blurring magnitude”.

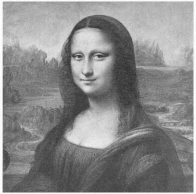
MEAN VALUE FILTER

- Computes the mean value in the neighbourhood.
  - All weights (values in the filter) are equal.
  - The sum of the weights is equal to 1.
- The neighbourhood size determines the ”blurring magnitude”.
  - Large filter: Loss of details, more blurring.

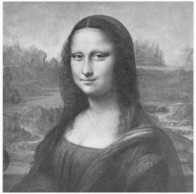
MEAN VALUE FILTER

- Computes the mean value in the neighbourhood.
  - All weights (values in the filter) are equal.
  - The sum of the weights is equal to 1.
- The neighbourhood size determines the ”blurring magnitude”.
  - Large filter: Loss of details, more blurring.
  - Small filter: Preservation of details, less blurring.

$$\frac{1}{9}\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}, \quad \frac{1}{25}\begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{bmatrix}, \quad \frac{1}{49}\begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix}$$



(a) Original



(b) 3×3

(c) 9×9

(d) 25×25

Figure 8: Gray level Mona Lisa image filtered with a mean value filter of different sizes.

Objective: Locate large, bright objects.

Possible solution: 15×15 mean value filtering followed by a global thresholding.

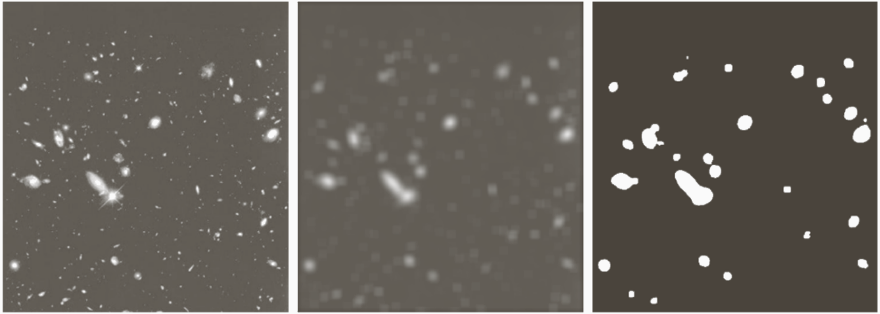


Figure 9: Left: Image obtained with the Hubble Space Telescope. Middle: Result after mean value filtering. Right: Result after global thresholding of the filtered image.

- For integer valus x, y, let

$$w[x,y] = A \exp\left\{-\frac{x^2+y^2}{2\sigma^2}\right\}$$

where A is usually such that $\sum_x \sum_y w[x,y] = 1$.

GAUSSIAN FILTER

- For integer valus x, y, let

$$w[x,y] = A \exp\left\{-\frac{x^2+y^2}{2\sigma^2}\right\}$$

  where A is usually such that $\sum_x \sum_y w[x,y] = 1$.
- This is a *non-uniform low-pass filter*.

- For integer valus x, y, let

$$w[x,y] = A \exp\left\{-\frac{x^2+y^2}{2\sigma^2}\right\}$$

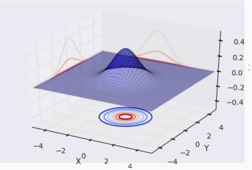
where A is usually such that $\sum_x \sum_y w[x,y] = 1$.
- This is a *non-uniform low-pass filter*.
- The parameter σ is the standard deviation and controls the amount of smoothing.
    - Small σ: Less smoothing
    - Large σ: More smoothing

GAUSSIAN FILTER

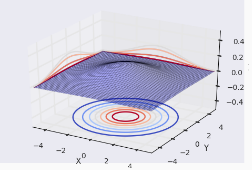
- For integer valus x, y, let

$$w[x, y] = A \exp\left\{-\frac{x^2 + y^2}{2\sigma^2}\right\}$$

  where A is usually such that $\sum_x \sum_y w[x, y] = 1$.
- This is a *non-uniform low-pass filter.*
- The parameter σ is the standard deviation and controls the amount of smoothing.
  - Small σ: Less smoothing
  - Large σ: More smoothing
- A Gaussian filter smooths less than a uniform filter of the same size.



(a) $\sigma = 1$



(b) $\sigma = 2$

Figure 10: Continuous bivariate Gaussian with different σ. Implementation can be found at `https://ojskrede.github.io/inf2310/lectures/week_06/`

APPROXIMATION OF GAUSSIAN FILTER

A 3×3 Gaussian filter can be approximated as

$$w = \frac{1}{16}\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 1 \\ 1 & 2 & 1 \end{bmatrix}.$$

APPROXIMATION OF GAUSSIAN FILTER

A 3×3 Gaussian filter can be approximated as

$$w = \frac{1}{16} \begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 1 \\ 1 & 2 & 1 \end{bmatrix}.$$

This is separable as 4 1D filters $\frac{1}{2}[1, 1]$:

$$w = \frac{1}{2} \begin{bmatrix} 1 \\ 1 \end{bmatrix} * \frac{1}{2} \begin{bmatrix} 1 & 1 \end{bmatrix} * \frac{1}{2} \begin{bmatrix} 1 \\ 1 \end{bmatrix} * \frac{1}{2} \begin{bmatrix} 1 & 1 \end{bmatrix} = \frac{1}{4} \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix} * \frac{1}{4} \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}.$$

APPROXIMATION OF GAUSSIAN FILTER

A 3×3 Gaussian filter can be approximated as

$$w = \frac{1}{16}\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 1 \\ 1 & 2 & 1 \end{bmatrix}.$$

This is separable as 4 1D filters $\frac{1}{2}[1, 1]$:

$$w = \frac{1}{2}\begin{bmatrix} 1 \\ 1 \end{bmatrix} * \frac{1}{2}\begin{bmatrix} 1 & 1 \end{bmatrix} * \frac{1}{2}\begin{bmatrix} 1 \\ 1 \end{bmatrix} * \frac{1}{2}\begin{bmatrix} 1 & 1 \end{bmatrix} = \frac{1}{4}\begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix} * \frac{1}{4}\begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}.$$

Or two 1D filters $\frac{1}{4}[1, 2, 1]$:

$$w = \frac{1}{4}\begin{bmatrix} 1 \\ 2 \\ 1 \end{bmatrix} * \frac{1}{4}\begin{bmatrix} 1 & 2 & 1 \end{bmatrix}.$$

EDGE-PRESERVING NOISE FILTERING

- We often use low-pass filters to reduce noise, but at the same time, we want to keep edges.

EDGE-PRESERVING NOISE FILTERING

- We often use low-pass filters to reduce noise, but at the same time, we want to keep edges.
- There exists a lot of edge-preserving filters.

- We often use low-pass filters to reduce noise, but at the same time, we want to keep edges.
- There exists a lot of edge-preserving filters.
- Many works by utilising only a sub-sample of the neighbourhood.

- We often use low-pass filters to reduce noise, but at the same time, we want to keep edges.
- There exists a lot of edge-preserving filters.
- Many works by utilising only a sub-sample of the neighbourhood.
- This could be implemented by sorting pixels *radiometrically* (by pixel value), and/or *geometrically* (by pixel location).

In the example from fig. 11, we could choose to only include contributions from within the blue ball.

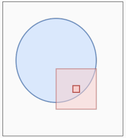


Figure 11: Blue ball on white background. Red square to illustrate filter kernel.

- We create a one-dimensional list of all pixel values around (x, y).

- We create a one-dimensional list of all pixel values around (x, y).
- We then sort this list.

- We create a one-dimensional list of all pixel values around (x, y).
- We then sort this list.
- Then, we compute the response from (x, y) from one element in the sorted list, or by some weighted sum of the elements.

- We create a one-dimensional list of all pixel values around (x, y).
- We then sort this list.
- Then, we compute the response from (x, y) from one element in the sorted list, or by some weighted sum of the elements.
- This is a non-uniform filter.

MEDIAN FILTERING

- A rank filter where we choose the middle value in the sorted list of values in the neighbourhood of (x, y).

MEDIAN FILTERING

- A rank filter where we choose the middle value in the sorted list of values in the neighbourhood of (x, y).
- One of the most frequently used edge-preserving filters.

MEDIAN FILTERING

- A rank filter where we choose the middle value in the sorted list of values in the neighbourhood of (x, y).
- One of the most frequently used edge-preserving filters.
- Especially well-suited for reducing *impulse-noise* (aka. ”salt-and-pepper-noise”).

MEDIAN FILTERING

- A rank filter where we choose the middle value in the sorted list of values in the neighbourhood of (x, y).
- One of the most frequently used edge-preserving filters.
- Especially well-suited for reducing *impulse-noise* (aka. ”salt-and-pepper-noise”).
- Not unusual with non-rectangular neighbourhood, e.g. plus-shaped neighbourhoods.

MEDIAN FILTERING

- A rank filter where we choose the middle value in the sorted list of values in the neighbourhood of (x, y).
- One of the most frequently used edge-preserving filters.
- Especially well-suited for reducing *impulse-noise* (aka. ”salt-and-pepper-noise”).
- Not unusual with non-rectangular neighbourhood, e.g. plus-shaped neighbourhoods.
- Some challenges:
    - Thin lines can dissappear

- A rank filter where we choose the middle value in the sorted list of values in the neighbourhood of (x, y).
- One of the most frequently used edge-preserving filters.
- Especially well-suited for reducing *impulse-noise* (aka. ”salt-and-pepper-noise”).
- Not unusual with non-rectangular neighbourhood, e.g. plus-shaped neighbourhoods.
- Some challenges:
  - Thin lines can dissappear
  - Corners can be rounded

MEDIAN FILTERING

- A rank filter where we choose the middle value in the sorted list of values in the neighbourhood of (x, y).
- One of the most frequently used edge-preserving filters.
- Especially well-suited for reducing *impulse-noise* (aka. ”salt-and-pepper-noise”).
- Not unusual with non-rectangular neighbourhood, e.g. plus-shaped neighbourhoods.
- Some challenges:
    - Thin lines can dissappear
    - Corners can be rounded
    - Objects can be shrinked

- A rank filter where we choose the middle value in the sorted list of values in the neighbourhood of (x, y).
- One of the most frequently used edge-preserving filters.
- Especially well-suited for reducing *impulse-noise* (aka. ”salt-and-pepper-noise”).
- Not unusual with non-rectangular neighbourhood, e.g. plus-shaped neighbourhoods.
- Some challenges:
  - Thin lines can dissappear
  - Corners can be rounded
  - Objects can be shrinked
- The size and shape of the neighbourhood is important

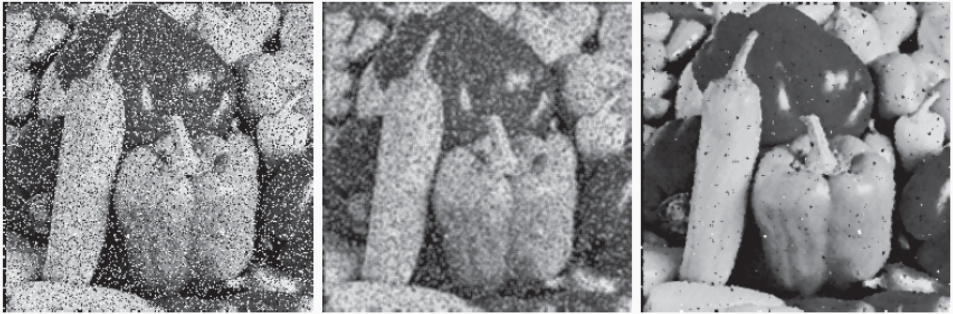


Figure 12: Left: Image with salt and pepper noise. Middle: Mean value filtering. Right: Median filtering.

Mean value filter: The mean value inside the neighbourhood.

- Smooths local variations and noise, but also edges.
- *Especially* well suited on local variations, e.g. mild noise in many pixel values.

MEAN VALUE- AND MEDIAN- FILTERS

Mean value filter: The mean value inside the neighbourhood.

- Smooths local variations and noise, but also edges.
- *Especially* well suited on local variations, e.g. mild noise in many pixel values.

Median value filter: The median value inside the neighbourhood

- Better for certain types of noise and preserves edges better.
- Worse on local variations and other kinds of noise.
- Especially well suited for salt-and-pepper-noise.

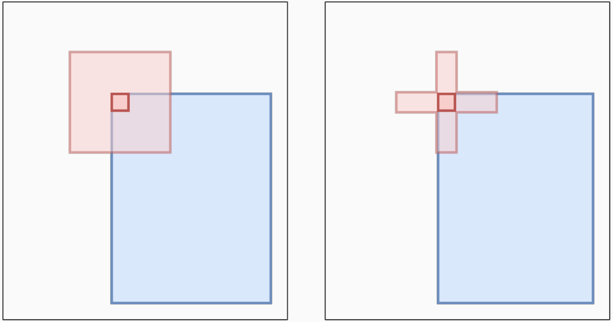


Figure 13: Left: Quadratic neighbourhood rounds corners. Right: pluss-shaped neighbourhood preserves corners.

- Sorting pixel values are slow (general worst case: $\mathcal{O}(n\log(n))$, where n is the number of elements (here, $n = L^2$)).
- Using histogram-updating techniques, we can achieve $\mathcal{O}(L)$[1]
- Utilizing histogram-updating even more, we can achieve $\mathcal{O}(1)$[1]

[1]Huang, T.S., Yang, G.J., Tang, G.Y.: *A Fast Two-Dimensional Median Filtering Algorithm*, EEE TASSP 27(1), 13-18, 1979

[2]Perreault and Hébert: *Median filtering in constant time*, IEEE TIP 16(9), 2389-2394, 2007.

ALPHA-TRIMMED MEAN VALUE FILTER

- The response is computed as the mean value of the $PQ - d$ middle values (after sorting) in the $P \times Q$ neighbourhood around (x, y).

- The response is computed as the mean value of the $PQ - d$ middle values (after sorting) in the $P \times Q$ neighbourhood around (x, y).
- Let $\Omega_{x,y}$ be the set of pixel positions of the $PQ - d$ middle values after sorting, then the response is given by

$$g[x, y] = \frac{1}{PQ - d} \sum_{(s,t) \in \Omega_{x,y}} f[s, t].$$

ALPHA-TRIMMED MEAN VALUE FILTER

- The response is computed as the mean value of the $PQ - d$ middle values (after sorting) in the $P \times Q$ neighbourhood around (x, y).
- Let $\Omega_{x,y}$ be the set of pixel positions of the $PQ - d$ middle values after sorting, then the response is given by

$$g[x, y] = \frac{1}{PQ - d} \sum_{(s,t) \in \Omega_{x,y}} f[s, t].$$

- For $d = 0$ this becomes the mean value filter.

- The response is computed as the mean value of the $PQ - d$ middle values (after sorting) in the $P \times Q$ neighbourhood around (x, y).
- Let $\Omega_{x,y}$ be the set of pixel positions of the $PQ - d$ middle values after sorting, then the response is given by

$$g[x, y] = \frac{1}{PQ - d} \sum_{(s,t) \in \Omega_{x,y}} f[s, t].$$

- For $d = 0$ this becomes the mean value filter.
- For $d = PQ - 1$ this becomes the median filter.

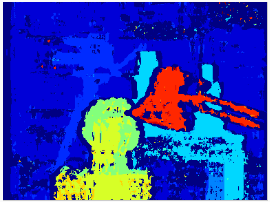
MODE FILTER (CURSORY READING)

- The response $g[x, y]$ is equal to the most frequent pixel value in $\mathcal{N}(x, y)$.

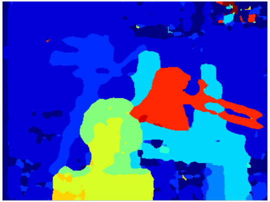
- The response $g[x, y]$ is equal to the most frequent pixel value in $\mathcal{N}(x, y)$.
- The number of unique pixel values must be small compared to the number of pixels in the neighbourhood.

MODE FILTER (CURSORY READING)

- The response $g[x, y]$ is equal to the most frequent pixel value in $\mathcal{N}(x, y)$.
- The number of unique pixel values must be small compared to the number of pixels in the neighbourhood.
- Mostly used on segmented images containing only a few color levels to remove isolated pixels.



(a) Segmented image



(b) After mode filtering

Figure 14: Segmented image before and after mode filtering.

K-NEAREST-NEIGHBOUR FILTER

- $g[x, y]$ is the mean value of the K nearest pixel values in $\mathcal{N}(x, y)$.

- $g[x, y]$ is the mean value of the K nearest pixel values in $\mathcal{N}(x, y)$.
- Here, nearest is in terms of absolute difference in value.

- $g[x, y]$ is the mean value of the K nearest pixel values in $\mathcal{N}(x, y)$.
- Here, nearest is in terms of absolute difference in value.
- Problem: K is constant for the entire image.
    - Too small K: We remove too little noise.
    - Too large K: We remove edges and corners.

- $g[x, y]$ is the mean value of the K nearest pixel values in $\mathcal{N}(x, y)$.
- Here, nearest is in terms of absolute difference in value.
- Problem: K is constant for the entire image.
  - Too small K: We remove too little noise.
  - Too large K: We remove edges and corners.
- How to choose K for a $L \times L$ neighbourhood, where $L = 2S + 1$.
  - $K = 1$: no effect.
  - $K \leq L$: preserves thin lines.
  - $K \leq (S + 1)^2$: preserves corners.
  - $K \leq (S + 1)L$: preserves straight lines.

K-NEAREST-CONNECTED-NEIGHBOUR FILTER

- The neighbourhood $\mathcal{N}(x, y)$ is the entire image.

K-NEAREST-CONNECTED-NEIGHBOUR FILTER

- The neighbourhood $\mathcal{N}(x, y)$ is the entire image.
- Thin lines, corners and edges are preserved if K is smaller or equal the number of pixels in the object.

K-NEAREST-CONNECTED-NEIGHBOUR FILTER

- The neighbourhood $\mathcal{N}(x, y)$ is the entire image.
- Thin lines, corners and edges are preserved if K is smaller or equal the number of pixels in the object.

The implementation is something like this

```
# K nearest connected neighbours (pseudocode)
# f is original image, and g is filtered image, both of shape [M, N]
for x in range(M):
  for y in range(N):
    chosen_vals = []
    chosen_pixel = (x, y)
    candidate_vals = []
    candidate_pixels = []
    while len(chosen_vals) <= K:
      candidate_pixels.append_unique(neighbourhood(chosen_pixel))
      candidate_vals = f[candidate_pixels]
      chosen_pixel = candidate_pixels.pop(argmin(abs(candidate_vals - f[x, y])))
      chosen_vals.append(f[chosen_pixel])
    g[x, y] = mean(chosen_vals)

```

MINIMAL MEAN SQUARE ERROR (MMSE) FILTER

- For a $P \times Q$ neighbourhood $\mathcal{N}(x,y)$ of (x,y), we can compute the sample mean $\mu(x,y)$ and variance[1] $\sigma^2(x,y)$

$$\mu(x,y) = \frac{1}{PQ} \sum_{(s,t)\in\mathcal{N}(x,y)} f[s,t]$$

$$\begin{aligned}\sigma^2(x,y) &= \frac{1}{PQ} \sum_{(s,t)\in\mathcal{N}(x,y)} (f[s,t] - \mu(x,y))^2 \\ &= \frac{1}{PQ} \sum_{(s,t)\in\mathcal{N}(x,y)} f^2[s,t] - \mu^2(x,y)\end{aligned}$$

[1] For an *unbiased* estimator of the variance, the denominator is $PQ - 1$.

MINIMAL MEAN SQUARE ERROR (MMSE) FILTER

- For a $P \times Q$ neighbourhood $\mathcal{N}(x, y)$ of (x, y), we can compute the sample mean $\mu(x, y)$ and variance[1] $\sigma^2(x, y)$

$$\begin{aligned} \mu(x, y) &= \frac{1}{PQ} \sum_{(s,t) \in \mathcal{N}(x,y)} f[s, t] \\ \sigma^2(x, y) &= \frac{1}{PQ} \sum_{(s,t) \in \mathcal{N}(x,y)} (f[s, t] - \mu(x, y))^2 \\ &= \frac{1}{PQ} \sum_{(s,t) \in \mathcal{N}(x,y)} f^2[s, t] - \mu^2(x, y) \end{aligned}$$

- Assume that we have an estimate of the noise-variance σ_η^2

[1] For an *unbiased* estimator of the variance, the denominator is $PQ - 1$.

MINIMAL MEAN SQUARE ERROR (MMSE) FILTER

- For a $P \times Q$ neighbourhood $\mathcal{N}(x,y)$ of (x,y), we can compute the sample mean $\mu(x,y)$ and variance[1] $\sigma^2(x,y)$

$$\mu(x,y) = \frac{1}{PQ} \sum_{(s,t)\in\mathcal{N}(x,y)} f[s,t]$$

$$\sigma^2(x,y) = \frac{1}{PQ} \sum_{(s,t)\in\mathcal{N}(x,y)} (f[s,t] - \mu(x,y))^2$$

$$= \frac{1}{PQ} \sum_{(s,t)\in\mathcal{N}(x,y)} f^2[s,t] - \mu^2(x,y)$$

- Assume that we have an estimate of the noise-variance σ_η^2
- Then, the $MMSE$-response at (x,y) is given as

$$g[x,y] = \begin{cases} f[x,y] - \frac{\sigma_\eta^2}{\sigma^2(x,y)}(f[x,y] - \mu(x,y)), & \sigma_\eta^2 \leq \sigma^2(x,y) \\ \mu(x,y), & \sigma_\eta^2 > \sigma^2(x,y). \end{cases}$$

[1]For an *unbiased* estimator of the variance, the denominator is $PQ - 1$.

MINIMAL MEAN SQUARE ERROR (MMSE) FILTER

- For a $P \times Q$ neighbourhood $\mathcal{N}(x,y)$ of (x,y), we can compute the sample mean $\mu(x,y)$ and variance[1] $\sigma^2(x,y)$

$$\mu(x,y) = \frac{1}{PQ} \sum_{(s,t)\in\mathcal{N}(x,y)} f[s,t]$$

$$\sigma^2(x,y) = \frac{1}{PQ} \sum_{(s,t)\in\mathcal{N}(x,y)} (f[s,t] - \mu(x,y))^2$$

$$= \frac{1}{PQ} \sum_{(s,t)\in\mathcal{N}(x,y)} f^2[s,t] - \mu^2(x,y)$$

- Assume that we have an estimate of the noise-variance σ_η^2
- Then, the $MMSE$-response at (x,y) is given as

$$g[x,y] = \begin{cases} f[x,y] - \frac{\sigma_\eta^2}{\sigma^2(x,y)}(f[x,y] - \mu(x,y)), & \sigma_\eta^2 \leq \sigma^2(x,y) \\ \mu(x,y), & \sigma_\eta^2 > \sigma^2(x,y). \end{cases}$$

- In ”homogeneous” areas, the response will be close to $\mu(x,y)$

[1] For an *unbiased* estimator of the variance, the denominator is $PQ - 1$.

MINIMAL MEAN SQUARE ERROR (MMSE) FILTER

- For a $P \times Q$ neighbourhood $\mathcal{N}(x, y)$ of (x, y), we can compute the sample mean $\mu(x, y)$ and variance[1] $\sigma^2(x, y)$

$$\mu(x,y) = \frac{1}{PQ} \sum_{(s,t)\in\mathcal{N}(x,y)} f[s,t]$$

$$\sigma^2(x,y) = \frac{1}{PQ} \sum_{(s,t)\in\mathcal{N}(x,y)} (f[s,t] - \mu(x,y))^2$$

$$= \frac{1}{PQ} \sum_{(s,t)\in\mathcal{N}(x,y)} f^2[s,t] - \mu^2(x,y)$$

- Assume that we have an estimate of the noise-variance σ_η^2
- Then, the $MMSE$-response at (x, y) is given as

$$g[x,y] = \begin{cases} f[x,y] - \frac{\sigma_\eta^2}{\sigma^2(x,y)}(f[x,y] - \mu(x,y)), & \sigma_\eta^2 \leq \sigma^2(x,y) \\ \mu(x,y), & \sigma_\eta^2 > \sigma^2(x,y). \end{cases}$$

- In ”homogeneous” areas, the response will be close to $\mu(x, y)$
- Close to edges will $\sigma^2(x, y)$ be larger than σ_η^2, resulting in a response closer to $f[x, y]$.

[1] For an *unbiased* estimator of the variance, the denominator is $PQ - 1$.

SIGMA FILTER (CURSORY READING)

- The filter result $g[x, y]$ is equal to the mean value of the pixels in the neighbourhood $\mathcal{N}(x, y)$ with values in the interval $f[x, y] \pm k\sigma$.

SIGMA FILTER (CURSORY READING)

- The filter result $g[x, y]$ is equal to the mean value of the pixels in the neighbourhood $\mathcal{N}(x, y)$ with values in the interval $f[x, y] \pm k\sigma$.
- σ is a standard deviation estimated from ”homogeneous” regions in f.

- The filter result $g[x, y]$ is equal to the mean value of the pixels in the neighbourhood $\mathcal{N}(x, y)$ with values in the interval $f[x, y] \pm k\sigma$.
- σ is a standard deviation estimated from "homogeneous" regions in f.
- k is a parameter with an appropriate problem-dependent value.

$$g[x, y] = \frac{\sum_{s=-S}^{S} \sum_{t=-t}^{T} w_{xy}[s, t] f[x + s, y + t]}{\sum_{s=-S}^{S} \sum_{t=-t}^{T} w_{xy}[s, t]},$$

where

$$w_{xy}[s, t] = \begin{cases} 1, & \text{if } |f[x, y] - f[x + s, y + t]| \leq k\sigma \\ 0, & \text{if not .} \end{cases}$$

MAX-HOMOGENEITY FILTER

- An edge-preserving filter.

MAX-HOMOGENEITY FILTER

- An edge-preserving filter.
- Form multiple overlapping sub-neighbourhoods from the original neighbourhood.

MAX-HOMOGENEITY FILTER

- An edge-preserving filter.
- Form multiple overlapping sub-neighbourhoods from the original neighbourhood.
- There exist many different ways to split the neighbourhood.

MAX-HOMOGENEITY FILTER

- An edge-preserving filter.
- Form multiple overlapping sub-neighbourhoods from the original neighbourhood.
- There exist many different ways to split the neighbourhood.
- Every sub-neighbourhood should contain the original centre pixel.

MAX-HOMOGENEITY FILTER

- An edge-preserving filter.
- Form multiple overlapping sub-neighbourhoods from the original neighbourhood.
- There exist many different ways to split the neighbourhood.
- Every sub-neighbourhood should contain the original centre pixel.
- The most homogeneous (e.g. with smallest variance) sub-neighbourhood contains the least edges.

MAX-HOMOGENEITY FILTER

- An edge-preserving filter.
- Form multiple overlapping sub-neighbourhoods from the original neighbourhood.
- There exist many different ways to split the neighbourhood.
- Every sub-neighbourhood should contain the original centre pixel.
- The most homogeneous (e.g. with smallest variance) sub-neighbourhood contains the least edges.
- Computation:
    - Compute the mean value and variance in each sub-neighbourhood.
    - Set $g[x, y]$ equal to the mean value of the sub-neighbourhood with lowes variance.

- For every symmetrical pixel-pair in the neighbourhood of (x, y):
  Choose the pixel with the closest value to $f[x, y]$

SYMMETRICAL NEAREST NEIGHBOUR (SNN) FILTER

- For every symmetrical pixel-pair in the neighbourhood of (x, y):
  Choose the pixel with the closest value to $f[x, y]$
- $g[x, y]$ is then the mean value of the chosen pixel values and $f[x, y]$.

SYMMETRICAL NEAREST NEIGHBOUR (SNN) FILTER

- For every symmetrical pixel-pair in the neighbourhood of (x, y):
  Choose the pixel with the closest value to $f[x, y]$
- $g[x, y]$ is then the mean value of the chosen pixel values and $f[x, y]$.
- The number of values that are meaned in a $P \times Q$ neighbourhood is then $(PQ - 1)/2 + 1$.

Most important

- Mean value filter
- Gaussian filter
- Median filter

TABLE OF SOME LOW-PASS FILTERS

Most important

- Mean value filter
- Gaussian filter
- Median filter

Other examples covered today

- Alpha-trimmed mean value filter
- Mode filter
- K-nearest-neighbour filter
- K-nearest-connected-neighbour filter
- MMSE filter
- Sigma filter
- Max-homogeneity filter
- Symmetrical nearest neighbour filter

Most important

- Mean value filter
- Gaussian filter
- Median filter

Other examples covered today

- Alpha-trimmed mean value filter
- Mode filter
- K-nearest-neighbour filter
- K-nearest-connected-neighbour filter
- MMSE filter
- Sigma filter
- Max-homogeneity filter
- Symmetrical nearest neighbour filter

Other examples not covered today

- Family of image guided adaptive filters (e.g. anisotropic diffusion filter)

SUMMARY

- Spatial filter: a neighbourhood and an operator.
    - The operator defines the action taken on the input image.
    - The neighbourhood defines which pixels the operator use.

SUMMARY

- Spatial filter: a neighbourhood and an operator.
    - The operator defines the action taken on the input image.
    - The neighbourhood defines which pixels the operator use.
- Convolution is linear spatial filtering
    - Knowing how this works is essential in this course.
    - Knowing how to implement this is also essential.

SUMMARY

- Spatial filter: a neighbourhood and an operator.
    - The operator defines the action taken on the input image.
    - The neighbourhood defines which pixels the operator use.
- Convolution is linear spatial filtering
    - Knowing how this works is essential in this course.
    - Knowing how to implement this is also essential.
- Correlation can be used for template matching

- Spatial filter: a neighbourhood and an operator.
    - The operator defines the action taken on the input image.
    - The neighbourhood defines which pixels the operator use.
- Convolution is linear spatial filtering
    - Knowing how this works is essential in this course.
    - Knowing how to implement this is also essential.
- Correlation can be used for template matching
- Low-pass filters can reduce noise, and there are many different low-pass filters.

Questions?