FilterFlow - transformation reference

Dominik Lau, Łukasz Głazik, Tadeusz Brzeski

June 18, 2024

Nomenclature

In_i - i-th input of a transformation
Out - ouput of a transformation
$I(x, y)$ - image's I pixel value with coordinates x, y; $I(x, y) \in [0, 255]^3$, $I(x, y) = (r, g, b)^T$
K - kernel (for convolutions)
$size(K)$ - kernel size
$K(i, j)$ - kernel value with coordinates (i, j), where $i, j \in 0..size(K) - 1$
$\oplus$ - XOR

1 Source

1.1 Source

User-uploaded source

1.2 Red

Only red color i.e.

$$Out(x, y) = (1, 0, 0)^T$$

Image size is 512×512

1.3 White noise

White noise as output with a random seed. Image size is 512×512. Pattern changes on refresh.

1.4 perlin noise

Perlin noise as output with 5 octaves, persistence 0.5 and a random seed [1]. Image size is 512×512. Pattern changes on refresh.

2 Linear

2.1 Custom kernel

A convolution with user-defined kernel. Available kernel sizes 2, 3 and 4. Let $N = size(K)$. The convolution does the following

$$Out(x, y) = \sum_{i=0}^{N-1} \sum_{j=0}^{N-1} I(x + i - \lfloor N/2 \rfloor, y + j - \lfloor N/2 \rfloor) * K(i, j)$$

The pixel we take neighborhood for is determined by the above equation and it's the top left pixel for kernel size 2, middlemost for kernel size 3 and pixel with coordinates corresponding to $K(1, 1)$.

2.2 4-connected Laplace

4-connected Laplace is like Custom kernel with kernel

$$K = \begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

2.3 8-connected Laplace

8-connected Laplace is like Custom kernel with kernel

$$K = \begin{bmatrix} 1 & 1 & 1 \\ 1 & -8 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

2.4 Gaussian

Gaussian is like Custom kernel with kernel

$$K = \begin{bmatrix} 0.0625 & 0.125 & 0.0625 \\ 0.125 & 0.25 & 0.125 \\ 0.0625 & 0.125 & 0.0625 \end{bmatrix}$$

2.5 Sobel X

Sobel X is like Custom kernel with kernel

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

2.6 Sobel Y

Sobel Y is like Custom kernel with kernel

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

3 Pooling

Pooling takes a block of pixels of size $N \times N$ and calculates reduction functions on it. Then it moves *step* pixels in one direction. The step value is (called stride) also customizable. For instance, if the pooling size is 3 and step is 4 it is going to apply a reduction function on a block of size 3×3 and move 4 pixels to the next block of size 3×3. Each pooling changes the size of the output image

$$Out.width = \left\lfloor \frac{In.width - 1}{stride + 1} \right\rfloor$$
$$Out.height = \left\lfloor \frac{In.height - 1}{stride + 1} \right\rfloor$$

Available strides: 1, 2, 3, 4. Available pooling sizes: 2, 3, 4.

Let $N =$ pooling size. The pooling operation expressed mathematically

$$out(x, y) = \odot_{i=0..N-1} \odot_{i=0..N-1} I(x + i - \lfloor N/2 \rfloor, y + j - \lfloor N/2 \rfloor)$$

where $\odot$ is the reduction function i.e. $\odot : \mathbb{R}^N \longrightarrow \mathbb{R}$

3.1 Max pooling

It is a pooling operation with $\odot = max$

3.2 Min pooling

It is a pooling operation with $\odot = min$

3.3 Avg pooling

It is a pooling operation with $\odot = average$

4 Logical

4.1 And

For a given argument p

$$Out(x, y) = p \wedge In(x, y)$$

4.2 Or

For a given argument p

$$Out(x, y) = p \vee In(x, y)$$

4.3 Xor

For a given argument p

$$Out(x, y) = p \oplus In(x, y)$$

4.4 Binary And

Binary as in two arguments.

$$Out(x, y) = In_1(x, y) \wedge In_2(x, y)$$

4.5 Binary Or

$$Out(x, y) = In_1(x, y) \vee In_2(x, y)$$

4.6 Binary Xor

$$Out(x, y) = In_1(x, y) \oplus In_2(x, y)$$

5 Binary

5.1 Addition

$$Out(x, y) = In_1(x, y) + In_2(x, y)$$

5.2 Multiply

$$Out(x, y) = In_1(x, y) * In_2(x, y)$$

5.3 Substraction

$$Out(x,y) = In_1(x,y) - In_2(x,y)$$

6 Point

6.1 Brightness

For a given argument p

$$Out(x,y) = pIn(x,y)$$

6.2 Threshold

For a given argument p

$$\begin{cases} Out(x,y) = 255 \text{ for } I(x,y) > p \\ Out(x,y) = 0 \text{ for } I(x,y) \leq p \end{cases}$$

6.3 Grayscale

$$Out(x,y) = In(x,y).r * 0.299 + In(x,y).g * 0.587 + In(x,y).b * 0.114;$$

6.4 To YCbCr

$$Out(x,y) = \begin{bmatrix} 0.299 & 0.587 & 0.114 \\ -0.169 & -0.331 & 0.5 \\ 0.5 & -0.419 & -0.081 \end{bmatrix} In(x,y) + (0, 127.5, 127.5)^T$$

6.5 From YCbCr

$$Out(x,y) = \begin{bmatrix} 1 & 0 & 1.4 \\ 1 & -0.343 & -0.711 \\ 1 & 1.765 & 0 \end{bmatrix} (In(x,y) - (0, 127.5, 127.5)^T)$$

6.6 R channel

$$Out(x,y) = (In(x,y).r, 0, 0)^T$$

6.7 G channel

$$Out(x,y) = (0, In(x,y).g, 0)^T$$

6.8 B channel

$$Out(x,y) = (0, 0, In(x,y).b)^T$$

6.9 Channel combination

This transformation combines three images, it takes r channel of the first one, g channel of the second one and b channel of the third one and sets them as output channels

$$Out(x,y) = (In_1(x,y).r, In_2(x,y).g, In_3(x,y).b)^T$$

7 Morphologic

7.1 Erosion

For each neighborhood of pixels (that means for a kernel of $size(K) = N$) it performs a reduction together with logical and

$$\begin{aligned}
Out(x,y).r &= \bigwedge_{i=0..N-1} \bigwedge_{j=0..N-1} In(x+i, y+j).r > t \\
Out(x,y).g &= \bigwedge_{i=0..N-1} \bigwedge_{j=0..N-1} In(x+i, y+j).g > t \\
Out(x,y).b &= \bigwedge_{i=0..N-1} \bigwedge_{j=0..N-1} In(x+i, y+j).b > t
\end{aligned}$$

where $t = 0.5$.

7.2 Dilatation

For each neighborhood of pixels (that means for a kernel of $size(K) = N$) it performs a reduction together with logical and

$$\begin{aligned}
Out(x,y).r &= \bigvee_{i=0..N-1} \bigvee_{j=0..N-1} In(x+i, y+j).r > t \\
Out(x,y).g &= \bigvee_{i=0..N-1} \bigvee_{j=0..N-1} In(x+i, y+j).g > t \\
Out(x,y).b &= \bigvee_{i=0..N-1} \bigvee_{j=0..N-1} In(x+i, y+j).b > t
\end{aligned}$$

where $t = 0.5$.

7.3 Skeletonization

The image is skeletonized using the simplified algorithm[2]. Only 2 iterations (due to time complexity and us wanting to keep it real-time). Pseudocode of one iteration:

```
eroded = erodsion(img)
temp = dilation(eroded)
temp = img - temp
skel = bitwise_or(skel, temp)
img = eroded
```

8 Other

8.1 Mux

A transformation that passes selected input forward. Let i be the selected image index, then

$$Out(x,y) = In_i(x,y)$$

References

[1] https://adrianb.io/2014/08/09/perlinnoise.html

[2] https://gist.github.com/jsheedy/3913ab49d344fac4d02bcc887ba4277d