Mask operations on matrices

Prev Tutorial: How to scan images, lookup tables and time measurement with OpenCV

Next Tutorial: Operations with images

Original author	Bernát Gábor
Compatibility	OpenCV >= 3.0

Mask operations on matrices are quite simple. The idea is that we recalculate each pixel's value in an image according to a mask matrix (also known as kernel). This mask holds values that will adjust how much influence neighboring pixels (and the current pixel) have on the new pixel value. From a mathematical point of view we make a weighted average, with our specified values.

Our test case

Let's consider the issue of an image contrast enhancement method. Basically we want to apply for every pixel of the image the following formula:

$$I(i,j) = 5 * I(i,j) - [I(i-1,j) + I(i+1,j) + I(i,j-1) + I(i,j+1)]$$

$$\iff I(i,j) * M, \text{where } M = \begin{array}{c|ccc} {}_i\backslash^j & -1 & 0 & +1 \\ -1 & 0 & -1 & 0 \\ 0 & -1 & 5 & -1 \\ +1 & 0 & -1 & 0 \end{array}$$

The first notation is by using a formula, while the second is a compacted version of the first by using a mask. You use the mask by putting the center of the mask matrix (in the upper case noted by the zero-zero index) on the pixel you want to calculate and sum up the pixel values multiplied with the overlapped matrix values. It's the same thing, however in case of large matrices the latter notation is a lot easier to look over.

Code

C++ | Java | Python

You can download this source code from here or look in the OpenCV source code libraries sample directory at
`samples/cpp/tutorial_code/core/mat_mask_operations/mat_mask_operations.cpp` .

```
#include <opencv2/imgcodecs.hpp>
#include <opencv2/highgui.hpp>
#include <opencv2/imgproc.hpp>
#include <iostream>

using namespace std;
using namespace cv;

static void help(char* progName)
{
    cout << endl
        <<  "This program shows how to filter images with mask: the write it yourself and the"
        << "filter2d way. " << endl
        <<  "Usage:"                                                                    << endl
        << progName << " [image_path -- default lena.jpg] [G -- grayscale] "        << endl << endl;
}


void Sharpen(const Mat& myImage,Mat& Result);

int main( int argc, char* argv[])
{
    help(argv[0]);
    const char* filename = argc >=2 ? argv[1] : "lena.jpg";

    Mat src, dst0, dst1;

    if (argc >= 3 && !strcmp("G", argv[2]))
        src = imread( samples::findFile( filename ), IMREAD_GRAYSCALE);
    else
        src = imread( samples::findFile( filename ), IMREAD_COLOR);

    if (src.empty())
    {
        cerr << "Can't open image ["  << filename << "]" << endl;
        return EXIT_FAILURE;
    }

    namedWindow("Input", WINDOW_AUTOSIZE);
    namedWindow("Output", WINDOW_AUTOSIZE);

    imshow( "Input", src );
    double t = (double)getTickCount();
```

```
    Sharpen( src, dst0 );

    t = ((double)getTickCount() - t)/getTickFrequency();
    cout << "Hand written function time passed in seconds: " << t << endl;

    imshow( "Output", dst0 );
    waitKey();

    Mat kernel = (Mat_<char>(3,3) <<  0, -1,  0,
                                    -1,  5, -1,
                                     0, -1,  0);

    t = (double)getTickCount();

    filter2D( src, dst1, src.depth(), kernel );
    t = ((double)getTickCount() - t)/getTickFrequency();
    cout << "Built-in filter2D time passed in seconds:     " << t << endl;

    imshow( "Output", dst1 );

    waitKey();
    return EXIT_SUCCESS;
}
void Sharpen(const Mat& myImage,Mat& Result)
{
    CV_Assert(myImage.depth() == CV_8U);  // accept only uchar images

    const int nChannels = myImage.channels();
    Result.create(myImage.size(),myImage.type());

    for(int j = 1 ; j < myImage.rows-1; ++j)
    {
        const uchar* previous = myImage.ptr<uchar>(j - 1);
        const uchar* current  = myImage.ptr<uchar>(j    );
        const uchar* next     = myImage.ptr<uchar>(j + 1);

        uchar* output = Result.ptr<uchar>(j);

        for(int i= nChannels;i < nChannels*(myImage.cols-1); ++i)
        {
            *output++ = saturate_cast<uchar>(5*current[i]
                         -current[i-nChannels] - current[i+nChannels] - previous[i] - next[i]);
        }
    }

    Result.row(0).setTo(Scalar(0));
    Result.row(Result.rows-1).setTo(Scalar(0));
    Result.col(0).setTo(Scalar(0));
    Result.col(Result.cols-1).setTo(Scalar(0));
}
```

The Basic Method

C++ | Java | Python

Now let us see how we can make this happen by using the basic pixel access method or by using the **filter2D()** function.

Here's a function that will do this:

```
void Sharpen(const Mat& myImage,Mat& Result)
{
    CV_Assert(myImage.depth() == CV_8U);  // accept only uchar images

    const int nChannels = myImage.channels();
    Result.create(myImage.size(),myImage.type());

    for(int j = 1 ; j < myImage.rows-1; ++j)
    {
        const uchar* previous = myImage.ptr<uchar>(j - 1);
        const uchar* current  = myImage.ptr<uchar>(j    );
        const uchar* next     = myImage.ptr<uchar>(j + 1);

        uchar* output = Result.ptr<uchar>(j);

        for(int i= nChannels;i < nChannels*(myImage.cols-1); ++i)
        {
            *output++ = saturate_cast<uchar>(5*current[i]
                         -current[i-nChannels] - current[i+nChannels] - previous[i] - next[i]);
        }
    }

    Result.row(0).setTo(Scalar(0));
    Result.row(Result.rows-1).setTo(Scalar(0));
    Result.col(0).setTo(Scalar(0));
    Result.col(Result.cols-1).setTo(Scalar(0));
}
```

At first we make sure that the input images data is in unsigned char format. For this we use the **cv::CV_Assert** function that throws an error when the expression inside it is false.

```
    CV_Assert(myImage.depth() == CV_8U);  // accept only uchar images
```

We create an output image with the same size and the same type as our input. As you can see in the **storing** section, depending on the number of channels we may have one or more subcolumns.

We will iterate through them via pointers so the total number of elements depends on this number.

```
    const int nChannels = myImage.channels();
    Result.create(myImage.size(),myImage.type());
```

We'll use the plain C [] operator to access pixels. Because we need to access multiple rows at the same time we'll acquire the pointers for each of them (a previous, a current and a next line). We need another pointer to where we're going to save the calculation. Then simply access the right items with the [] operator. For moving the output pointer ahead we simply increase this (with one byte) after each operation:

```
    for(int j = 1 ; j < myImage.rows-1; ++j)
    {
        const uchar* previous = myImage.ptr<uchar>(j - 1);
        const uchar* current  = myImage.ptr<uchar>(j    );
        const uchar* next     = myImage.ptr<uchar>(j + 1);

        uchar* output = Result.ptr<uchar>(j);

        for(int i= nChannels;i < nChannels*(myImage.cols-1); ++i)
        {
            *output++ = saturate_cast<uchar>(5*current[i]
                         -current[i-nChannels] - current[i+nChannels] - previous[i] - next[i]);
        }
    }
```

On the borders of the image the upper notation results inexistent pixel locations (like minus one - minus one). In these points our formula is undefined. A simple solution is to not apply the kernel in these points and, for example, set the pixels on the borders to zeros:

```
    Result.row(0).setTo(Scalar(0));
    Result.row(Result.rows-1).setTo(Scalar(0));
    Result.col(0).setTo(Scalar(0));
    Result.col(Result.cols-1).setTo(Scalar(0));
```

The filter2D function

C++ | Java | Python

Applying such filters are so common in image processing that in OpenCV there is a function that will take care of applying the mask (also called a kernel in some places). For this you first need to define an object that holds the mask:

```
    Mat kernel = (Mat_<char>(3,3) <<  0, -1,  0,
                                     -1,  5, -1,
                                      0, -1,  0);
```

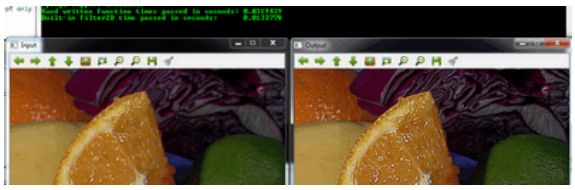
Then call the **filter2D()** function specifying the input, the output image and the kernel to use:

```
    filter2D( src, dst1, src.depth(), kernel );
```

The function even has a fifth optional argument to specify the center of the kernel, a sixth for adding an optional value to the filtered pixels before storing them in K and a seventh one for determining what to do in the regions where the operation is undefined (borders).

This function is shorter, less verbose and, because there are some optimizations, it is usually faster than the *hand-coded method*. For example in my test while the second one took only 13 milliseconds the first took around 31 milliseconds. Quite some difference.

For example:



Check out an instance of running the program on our YouTube channel .