快速入门OpenCV边缘识别

本文档由华南理工大学机器人协会视觉部整理

图片的读取、展示与保存

在OpenCV中，图片是在矩阵`cv::Mat`这种数据结构中储存的。OpenCV在`opencv2/imgcodecs.hpp`头文件中，为我们提供了`cv::imread`函数读取图片；在`opencv2/highgui.hpp`头文件中为我们提供了`cv::imshow`函数用于在操作系统中绘制窗体，展示储存的矩阵；在`opencv2/imgcodecs.hpp`头文件中为我们提供了`cv::imwrite`函数用于将图片以指定格式保存在指定路径。如下所示：

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

int main(void)
{
    // 读取图片
    cv::Mat image = cv::imread("lena.png");
    if (image.empty())
    {
        std::cerr << "Check if the image exists!" << std::endl;
        return 0;
    }

    // 展示图片
    const char* const WINDOW_NAME = "My Image";
    cv::namedWindow(WINDOW_NAME, cv::WINDOW_FREERATIO);
    cv::imshow(WINDOW_NAME, image);
    cv::waitKey(0);

    // 保存图片
    cv::imwrite("lena.jpeg", image);
    return 0;
}
```

在这份代码中，请留意以下几个要点：

1. OpenCV中，所有的函数、模板、数据结构、宏，均被放在`cv`这个命名空间下。为了强调这一点，这份代码使用了作用域解析运算符，在接下来的代码中，出于简洁考虑，我们将使用`using namespace cv`。
2. `cv::imread`函数可以读取指定文件名的文件。这个函数还支持第二个参数，用于设置读取图像的形式，例如我们后文将使用到的`cv::IMREAD_GRAYSCALE`，可以将图片以灰度图像的形式进行读入。参见`cv::ImreadModes`。
3. `cv::namedWindow`函数创建了一个名为`WINDOW_NAME`的窗体。第一个参数`WINDOW_NAME`既是窗体的标题，也是这个窗体的唯一标识符。第二个参数设置这个窗体的属性为`cv::WINDOW_FREERATIO`，允许用户通过鼠标自由拉伸窗体。这个参数的默认值为`cv::WINDOW_AUTOSIZE`。更多的属性请参阅`cv::WindowFlags`。
4. `cv::imshow`这个函数的第一个参数也为`WINDOW_NAME`，用于在指定的`WINDOW_NAME`窗体显示图像。如果该窗体不存在，OpenCV会自行创建一个窗体。换而言之，`cv::namedWindow`不是必须

的。

5. `cv::waitKey`函数可以将程序阻塞，直到发生键盘事件（即键盘的敲击），键盘的按键作为返回值返回。这行语句的作用是，让窗体保持激活并展示图片的状态。如果没有这行语句，当程序会继续运行（注意，`cv::imshow`不会阻塞程序），程序会很快退出，相关窗体此时被销毁。对于用户而言，其效果相当于“看见窗体闪了一下就退出了”。这个函数的参数可以设置阻塞的最长时间（单位为毫秒），`0`表示无限期阻塞，我们将在后续的代码中看到这种用法。
6. `cv::imwrite`函数将第二个参数传入的矩阵（里面存有图像），保存在第一个参数给定的文件中。OpenCV会根据第一个参数的文件名的后缀，自动选择图像的格式。这个函数还允许传入第三个参数，使得用户可以进一步控制保存的图像的一些细节，参见`cv::imwrite`。如果失败，函数会抛出一个异常。

视频的读取、播放

众所周知，视频是有若干帧连续的图像组成。在OpenCV中，处理视频的一种思路是，将每一帧图像保存在矩阵中，按照前文提到的处理图像的方式处理视频。

```
#include <opencv2/videoio.hpp>
#include <opencv2/highgui.hpp>
#include <iostream>

using namespace cv;
using namespace std;

void playVideo(VideoCapture &video);

int main(void)
{
    // 通过摄像头采集视频
    VideoCapture camera(0);
    playVideo(camera);

    // 读取指定的视频文件
    VideoCapture video("video.mp4");
    playVideo(video);

    return 0;
}

void playVideo(VideoCapture &video)
{
    if (!video.isOpened())
    {
        cerr << "Check if the video exists!" << endl;
        return;
    }
    while (true)
    {
        Mat frame;
        video >> frame;

        // 当视频流中不再能读出新的帧，我们认为视频播放完毕，退出循环
        if (frame.empty())
            break;
```

```
        // 用新的帧刷新"Video"窗体，达到图片连续播放的“视频的效果”
        imshow("Video", frame);

        // 每帧画面持续播放的时间中，若有键盘输入，waitKey返回值不为-1，停止播放
        if (waitKey(1000 / video.get(CAP_PROP_FPS)) != -1)
            break;
    }
}
```

OpenCV在`opencv2/videoio.hpp`头文件为我们提供了`cv::VideoCapture`函数来读取视频，获得视频流，并通过流插入运算符获得每帧画面的图像矩阵（有点像`std::cin`，不是吗？）。其构造函数既可以接受一个字符串，表示视频文件的路径；或者是一个整数，表示摄像头的ID，且这个ID一般从0开始。

`cv::VideoCapture::get`成员函数返回有关视频的一些属性（`double`类型），例如示例代码中传入的`CAP_PROP_FPS`标志参数，可以获取这个视频的每秒传输帧数（*Frames Per Second*）。`1000 / video.get(CAP_PROP_FPS)`可以计算出每一帧所需要播放的毫秒数。其它的标志参数参见`cv::VideoCaptureProperties`。

视频的保存可以通过`cv::VideoWriter`实现。受篇幅限制，不多赘述。有兴趣的读者可以参阅：`cv::VideoWriter`。

图像的缩放

在一些计算机视觉算法中，过大的图像尺寸对运算结果没有太大的帮助，反而会大幅增加运算时间。因此，在一些对性能要求比较高的需求中，人们会考虑缩小图像的尺寸以优化运算时间。OpenCV在`opencv2/imgproc.hpp`头文件中，为我们提供了`cv::resize`函数来进行图像尺寸的缩放，其函数原型如下：

```
void cv::resize (
    InputArray      src,
    OutputArray     dst,
    Size            dsize,
    double          fx = 0,
    double          fy = 0,
    int             interpolation = INTER_LINEAR
);
```

以下是每个参数的含义：

- `src`：输入图像
- `dst`：输入图像，其数据类型应与`src`相同
- `dsize`：输出图像的尺寸。若为0，则以`fx`和`fy`计算出来的长宽比为准。
- `fx`, `fy`：分别表示图像在水平、垂直两轴的缩放比例因子。若比例与`dsize`不一致，以`dsize`为准
- `interpolation`：插值算法

在图像的缩放中，无可避免地会出现像素损失或缺失的情况。因此，我们需要用插值算法处理这些计算`dst`中像素点的值。默认的`cv::INTER_LINEAR`是双线性插值法，在图像放大时有较好的效果和性能；在图像缩小时，`cv::INTER_AREA`是一个不错的选择。其它可用的插值方法参见`cv::InterpolationFlags`。

下面这个例子展示了如何读入一个图像并将其长宽分别缩小一半、放大到原来的2倍。

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

using namespace cv;
using namespace std;

int main(void)
{
    Mat normalImage = imread("lena.png");
    if (normalImage.empty())
    {
        std::cerr << "Check if the image exists!" << std::endl;
        return 0;
    }

    Mat smallImage, bigImage;
    resize(normalImage, smallImage, Size(), 0.5, 0.5);
    resize(normalImage, bigImage, Size(), 2, 2);
    imshow("Normal Image", normalImage);
    imshow("Small Image", smallImage);
    imshow("Big Image", bigImage);
    waitKey(0);
    return 0;
}
```

颜色模型的转换——以RGB模型转GRAY模型为例

OpenCV中支持许多颜色模型，如RGB模型、HSV模型、Lab模型、YUV模型和GRAY模型。

使用RGB模型的图像矩阵中，通常有三个通道，分别表示像素点的颜色，在蓝色、绿色、红色上的“分量”，8位的无符号图像像素取值范围为0-255，浮点型图像取值范围为0-1。通过这三种色光三原色的组合，我们便可以表示一幅图像的每个像素点的所有颜色。另外，在RGB模型中还可以有第四个通道——alpha通道，表示图像的透明度。

GRAY模型是一个灰度图像模型，它只有一个通道，由0到最大值依次表示黑到白。彩色图像和灰度图像的转换，实际上是人眼对于彩色的感觉到亮度感觉的转换。根据心理学研究，直接将RGB三个通道取算术平均得到的灰度图像，并不那么符合人类的感知。事实上，OpenCV中采用的转换公式为：`Y = 0.299R + 0.587G + 0.114B`。

对于一些计算机视觉算法来说，彩色图像的色彩信息并没有那么重要。因此，针对这种情况，人们往往会将彩色图像转换为灰度图像，提高计算速度。

OpenCV在`opencv2/imgproc.hpp`头文件中，为我们提供了`cv::cvtColor`函数用于实现颜色模型的转换，接受四个参数，前两个参数分别为输入图像的矩阵和保存结果的矩阵。第三个参数为颜色空间转换的标志，如下文中用到的`cv::COLOR_BGR2GRAY`标记，表示将彩色图像转为灰度图像，其它颜色模型转换的标志参见`cv::ColorConversionCodes`。第四个参数为目标图像中的通道数，若为0，则函数会自行推断。RGB模型转GRAY模型代码如下所示：

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

using namespace cv;
```

```
using namespace std;

int main(void)
{
    Mat rgbImage = imread("lena.png");
    if (rgbImage.empty())
    {
        std::cerr << "Check if the image exists!" << std::endl;
        return 0;
    }

    Mat grayImage;
    cvtColor(rgbImage, grayImage, COLOR_BGR2GRAY);

    imshow("RGB", rgbImage);
    imshow("GRAY", grayImage);
    waitKey(0);

    return 0;
}
```

二值化

固定阈值的二值化

在灰度图像的基础上更进一步，如果我们只需要让整个图像呈现出明显的黑白效果，那么可以考虑使用二值化。二值化，指的是按照某种规则，将图像中的每一个像素点，（通常）置为0或最大值（对与我们常用的8位无符号图像，最大值是255）。图像的二值化使图像中数据量大为减少，从而能凸显出目标的轮廓。如处理二维码、文稿、板书，或者是识别形状，都可以使用二值化，这让文字、图像的形状得以凸显，便于后期处理。因此，在数字图像处理中，二值图像占有非常重要的地位。

OpenCV在`opencv2/imgproc.hpp`头文件中为我们提供了`cv::threshold`函数完成二值化操作，其原型如下：

```
double cv::threshold (
    InputArray      src,
    OutputArray     dst,
    double          thresh,
    double          maxval,
    int             type
);
```

- `src`, `dst`：分别表示输入和输出的二值化后的图像矩阵，后者要与前者有相同的尺寸、数据类型和通道数
- `thresh`：二值化的阈值
- `maxval`：二值化过程中的最大值，它只在`cv::THRESH_BINARY`和`cv::threSH_BINARY_INV`两种二值化方法中使用
- `type`：选择图像二值化方法的标志，如`cv::THRESH_BINARY`和`cv::THRESH_BINARY_INV`

其中，`cv::THRESH_BINARY`是一种图像二值化的方法。这个算法需要选定一个阈值`thresh`，然后遍历图像中的每一个像素，如果这个像素值大于阈值，则将其置为最大值`maxval`，反之，则置为0。而`cv::THRESH_BINARY_INV`则与之相反，将大于阈值的像素置为0，反之则置为最大值`maxval`。

$$
\begin{aligned}
THRESH\_BINARY(x,y) &= \begin{cases} maxval & \text{if } src(x,y) > thresh \\ 0 & \text{otherwise} \end{cases} \\
THRESH\_BINARY\_INV(x,y) &= \begin{cases} 0 & \text{if } src(x,y) > thresh \\ maxval & \text{otherwise} \end{cases}
\end{aligned}
$$

此外，这个图像二值化也有别的标志，有兴趣的读者可以参阅`cv::ThresholdTypes`。另外，如果有多个阈值需要比较，则可以使用显示查找表（*Look-Up-Table, LUT*），参见`cv::LUT`。

下面的这个样例程序展示了如何使用OpenCV对图像进行二值化处理：

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

using namespace cv;
using namespace std;

int main(void)
{
    Mat image = imread("blackboard.png", IMREAD_GRAYSCALE);
    if (image.empty())
    {
        std::cerr << "Check if the image exists!" << std::endl;
        return 0;
    }

    Mat binaryImg, binaryInvImg;
    threshold(image, binaryImg, 128, 255, THRESH_BINARY);
    threshold(image, binaryInvImg, 128, 255, THRESH_BINARY_INV);

    imshow("ORIGINAL", image);
    imshow("THRESH_BINARY", binaryImg);
    imshow("THRESH_BINARY_INV", binaryInvImg);

    waitKey(0);
    return 0;
}
```

自动选择阈值的二值化

至此，一个问题出现了——我们该如何选择合适的阈值呢？在一些场景中，光线、颜色等因素对我们图像有着很大的影响，不合理的阈值甚至是所有固定的阈值，往往无法在所有的情况下起到很好的效果。

为了解决这个问题。OpenCV为我们提供了`cv::THRESH_OTSC`和`cv::THRESH_TRIANGLE`标志，分别表示使用大津法和三角形法结合图像灰度值的分布特性获取二值化的阈值。它们可以与我们前文提到的`cv::THRESH_BINARY`等标志进行按位或运算，此时传入的第三个参数`thresh`将被忽略。

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

using namespace cv;
using namespace std;

int main(void)
{
```

```
    Mat image = imread("blackboard.png", IMREAD_GRAYSCALE);
    if (image.empty())
    {
        std::cerr << "Check if the image exists!" << std::endl;
        return 0;
    }

    Mat binaryImgOtsu, binaryImgTri, binaryInvImgOtsu, binaryInvImgTri;
    threshold(image, binaryImgOtsu, 0, 255, THRESH_BINARY | THRESH_OTSU);
    threshold(image, binaryImgTri, 0, 255, THRESH_BINARY | THRESH_TRIANGLE);
    threshold(image, binaryInvImgOtsu, 0, 255, THRESH_BINARY_INV | THRESH_OTSU);
    threshold(image, binaryInvImgTri, 0, 255, THRESH_BINARY_INV |
THRESH_TRIANGLE);

    imshow("THRESH_BINARY OTSU", binaryImgOtsu);
    imshow("THRESH_BINARY TRIANGLE", binaryImgTri);
    imshow("THRESH_BINARY_INV OTSU", binaryInvImgOtsu);
    imshow("THRESH_BINARY_INV TRIANGLE", binaryInvImgTri);

    waitKey(0);
    return 0;
}
```

自适应的二值化

无论是人为设置的阈值，还是自动选择的阈值，`cv::threshold`函数处理每张图片时，都只会使用一个阈值。但实际情况中，由于光照不均匀以及阴影的存在，全局只有一个阈值可能会使得在阴影处的白色区域也会被二值化为黑色。因此，在`opencv2/imgproc.hpp`头文件中，OpenCV为我们提供了`cv::adaptiveThreshold`函数，让我们可以在局部使用自适应的阈值。其函数原型如下：

```
void cv::adaptiveThreshold (
    InputArray  src,
    OutputArray dst,
    double      maxValue,
    int         adaptiveMethod,
    int         thresholdType,
    int         blockSize,
    double      C
);
```

- `src`, `dst`：分别表示输入和输出的图像。前者只能8位单通道图像，后者要与前者有相同的尺寸、数据类型和通道数
- `maxValue`：二值化的最大值
- `adaptiveMethod`：自适应确定阈值的方法，可以为均值法`cv::ADAPTIVE_THRESH_MEAN_C`和高斯法`ADAPTIVE_THRESH_GAUSSIAN_C`。
- `thresholdType`：图像二值化的方法，只能为`cv::THRESH_OTSC`或`cv::THRESH_TRIANGLE`
- `blockSize`：自适应确定阈值的像素邻域大小，一般为3、5、7等奇数。函数通过均值法或高斯法在此`blockSize * blockSize`的邻域内计算阈值进行二值化
- `C`：从平均值或者加权平均数中减去的常数，通常为正数，但也可以是0或负数

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

```
using namespace cv;
using namespace std;

int main(void)
{
    Mat image = imread("blackboard.png", IMREAD_GRAYSCALE);
    if (image.empty())
    {
        std::cerr << "Check if the image exists!" << std::endl;
        return 0;
    }
    Mat imgMean, imgGauss;
    adaptiveThreshold(image, imgMean, 255, ADAPTIVE_THRESH_MEAN_C,
THRESH_BINARY_INV, 75, 0);
    adaptiveThreshold(image, imgGauss, 255, ADAPTIVE_THRESH_GAUSSIAN_C,
THRESH_BINARY_INV, 75, 0);
    imshow("THRESH_MEAN", imgMean);
    imshow("THRESH_GAUSSIAN", imgGauss);
    waitKey(0);
    return 0;
}
```

轮廓检测与绘制

高斯滤波

在图像处理的过程中，有时候为了减少图像中的噪声造成的不良影响，我们可以考虑滤波。高斯滤波是一种常见的滤波方法。在`opencv2/imgproc.hpp`中，OpenCV提供了`cv::GaussianBlur`函数，能够根据输入参数自动生成高斯滤波器，实现对图像的高斯滤波。

```
void cv::GaussianBlur (
    InputArray  src,
    OutputArray dst,
    Size        ksize,
    double      sigmaX,
    double      sigmaY = 0,
    int         borderType = BORDER_DEFAULT
);
```

- `src`, `dst`：输入和输出的图像的矩阵
- `ksize`：高斯滤波器的尺寸，可以不为正方形，但边长必须为正奇数。若尺寸位0，则由标准偏差计算尺寸
- `sigmaX`,`sigmaY`：x、y轴方向的标准差，若后者位0，则置为与前者相同；若均为0，则由滤波器尺寸计算
- `borderType`：像素外推法选择标志，默认的`cv::BORDER_DEFAULT`表示不含边界值倒序填充，其余标志参见`cv::BorderTypes`

一般而言，高斯滤波对高斯噪声去除效果较高，但会造成图像模糊，且滤波器尺寸越大，滤波后图像变得越模糊。

轮廓检测

OpenCV为在`opencv2/imgproc.hpp`头文件中为我们提供了`cv::findContours`函数帮助我们检测图像的轮廓，其一个函数原型如下：

```
void cv::findContours (
    InputArray          image,
    OutputArrayOfArrays contours,
    int                 mode,
    int                 method,
    Point               offset = Point()
);
```

- `image`：输入的图像
- `contours`：输出的轮廓，每个轮廓存放着像素的坐标
- `mode`：检测轮廓的模式，可以为`cv::RETR_EXTERNAL`，`cv::RETR_LIST`，`cv::RETR_CCOMP`，`cv::RETR_TREE`，具体含义参见`cv::RetrievalModes`
- `method`：轮廓逼近方法标志，可以为`cv::CHAIN_APPROX_NONE`，`cv::CHAIN_APPROX_SIMPLE`，`cv::CHAIN_APPROX_TC89_L1`，`cv::CHAIN_APPROX_TC89_KOS`，具体含义参见`cv::ContourApproximationModes`
- `offset`：每个轮廓点移动的可选偏移量。这个参数主要用在从ROI图像中找出轮廓并基于整个图像分析轮廓的场景中

为了方便我们展示图像，OpenCV为我们提供了`cv::drawContours`函数。其原型如下：

```
void cv::drawContours (
    InputOutputArray    image,
    InputArrayOfArrays  contours,
    int                 contourIdx,
    const Scalar &      color,
    int                 thickness = 1,
    int                 lineType = LINE_8,
    InputArray          hierarchy = noArray(),
    int                 maxLevel = INT_MAX,
    Point               offset = Point()
)
```

- `image`：待绘制轮廓的目标图像
- `contours`：将要绘制的所有轮廓
- `contourId`：绘制轮廓的输入，若为负数，则绘制所有的轮廓
- `color`：绘制轮廓的颜色
- `thickness`：绘制轮廓线条的粗细，若为负数，则绘制轮廓的内部
- `lineType`：边界线的连接类型，可以为四连通线型`cv::LINE_4`，8联通线型`cv::LINE_8`，抗锯齿线型`cv::LINE_AA`
- `hierarchy`：可选的轮廓结构关系信息
- `maxLevel`：绘制轮廓的最大等级
- `offset`：可选的轮廓偏移参数，按指定的移动距离绘制所有的轮廓

我们处理的图像经常会有一些噪声，这对我们边缘检测的效果影响很大。一个小技巧是，在进行边缘检测前，先进行滤波。这样，我们就能减少噪声对图像的影响。

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

```
#include <iostream>
#include <vector>

using namespace cv;
using namespace std;

int main(void)
{
    Mat image = imread("Contour.png");
    if (image.empty())
    {
        std::cerr << "Check if the image exists!" << std::endl;
        return 0;
    }
    Mat grayImage, binaryImage;
    cvtColor(image, grayImage, COLOR_BGR2GRAY);
    GaussianBlur(grayImage, grayImage, Size(5, 5), 2, 2);
    threshold(grayImage, binaryImage, 0, 255, THRESH_BINARY | THRESH_OTSU);

    vector< vector<Point> > contours;
    vector<Vec4i> hierarchy;
    findContours(binaryImage, contours, hierarchy, RETR_TREE,
CHAIN_APPROX_SIMPLE, Point());

    for (int i = 0; i < contours.size(); ++i)
        drawContours(image, contours, i, Scalar(0, 0, 255), 2, LINE_AA);

    imshow("Result", image);
    waitKey(0);
    return 0;
}
```

当然，如果您需要获取轮廓之间的层级关系，`cv::drawContours`函数还有一个重载，能够帮助我们获取轮廓间的层级关系，有兴趣的朋友可以参见`cv::drawContours`。

参考文献

1. OpenCV 4.5.4官方文档
2. 冯振,郭延宁,吕悦勇.OpenCV 4快速入门[M].北京:人民邮电出版社,2020.