opencv学习笔记

一、核心模块（core）

(一)、Mat - 基本图像容器

1.Mat类由两个数据成员组成：一个是矩阵头，包含矩阵尺寸、存储方法、存储地址等信息；二是一个指向存储所有像素值的矩阵的指针，跟好所选存储方法的不同，矩阵可以是不同的维数，在c++中这个指针相当于智能指针，能够管理动态内存的指针。

2.每个Mat对象有自己的信息头，但是共享同一个矩阵，即矩阵的指针指向同一个地址。因此拷贝构造函数和拷贝赋值运算符都是只拷贝信息头和矩阵指针，不拷贝矩阵，是一种类指针的行为，因此改变某一个对象的内容，其他引用该地址的对象都会改变。

Mat A, C; // 只创建信息头部分，代表一个Mat对象

A = imread(argv[1], CV\_LOAD\_IMAGE\_COLOR); // 这里为矩阵开辟内存

Mat B(A); // 使用拷贝构造函数，拷贝其指针

C = A; // 赋值运算符，拷贝其指针

3.显式地创建一个Mat对象

（1）Mat()构造函数

Mat M(2,2,CV\_8UC3,Scalar(0,0,255));

对于二维多通道图像，首先要定义其尺寸，即行数和列数。然后需要指定存储元素的数据类型以及每个矩阵点的通道数。为此依据下面的规则有多种定义：

CV\_[The number of bits per item][Signed or Unsigned][Type Prefix]C[The channel number]

比如 CV\_8UC3 表示使用8位的 unsigned char 型，每个像素由三个元素组成三通道。预先定义的通道数可以多达四个。 [Scalar](http://opencv.itseez.com/modules/core/doc/basic_structures.html" \l "scalar) 是个short型vector。指定这个能够使用指定的定制化值来初始化矩阵。当然，如果你需要更多通道数，你可以使用大写的宏并把通道数放在小括号中，如下所示

（2）在c/c++中通过构造函数进行初始化

int sz[3]={2,2,2};

Mat L(3,sz,CV\_8UC(1),Scalar::all(0));

上面的例子演示了如何创建一个超过两维的矩阵：指定维数(3)，然后传递一个指向一个数组的指针(sz)，这个数组包含每个维度的尺寸；其余的相同

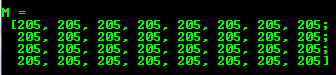
(3)为已存在IplImage指针创建信息头：

IplImage \* img=cvLoadImage(“greatwave.png”,1);

Mat mtx(img);//convert IplImage\*->Mat

(4)create()函数：

M.create(4,4,CV\_8UC(2));



（5）对于小矩阵可以用逗号分隔的初始化函数：

Mat c=(Mat<double>(3,3)<<0,-1,0,-1,5,-1,0,-1,0);

(6)clone()或copyTo(..)创建函数头：

Mat RowClone = C.row(1).clone();

3.对每个Mat对象的默认构造函数，主要是初始化其矩阵头：

Mat D (A, Rect(10, 10, 100, 100) ); // 使用A中矩形矩阵（10，10,100,100）初始化D

Mat E = A(Range:all(), Range(1,3)); // 同上

4.Mat对象保存的图像也可以进行拷贝，使用clone()或copyTo(..)函数：

Mat F=A.clone();

Mat G;

A.copyTo(G);

5.存储方法：存储像素值需要指定颜色空间和数据类型。颜色空间是指对一个给定的颜色，如何组合颜色元素以对其编码。最简单的颜色空间属灰度级空间，只处理黑色和白色，对它们进行组合可以产生不同程度的灰色；对于彩色方式则有更多种类的颜色空间，但不论哪种方式都是把颜色分成三个或者四个基元素，通过组合基元素可以产生所有的颜色。RGB颜色空间是最常用的一种颜色空间，这归功于它也是人眼内部构成颜色的方式，它的基色是红色、绿色和蓝色，有时为了表示透明颜色也会加入第四个元素alpha(A).

RGB是最常见的，这是因为人眼采用相似的工作机制，它也被显示设备所采用；

HSV和HLS把颜色分解为色调、饱和度和亮度/明度。这是描述颜色更自然的方式，比如可以通过抛弃最后一个元素，使算法对输入图像的光照条件不敏感；

YCrCb在JPEG图像格式中广泛使用；

CIE L\*a\*b是一种在感知上均匀的颜色空间，它适合用来度量两个颜色之间的距离。

每个组成元素都有其自己的定义域，取决于其数据类型。如何存储一个元素决定了我们在其定义域上能够控制的精度。最小的数据类型是char，占一个字节或8位。使用三个char型元素已经可以表示1600万种可能的颜色（使用RGB颜色空间），但若使用float(4字节，32位)或double(8字节，64位）则能给出更加精细的颜色分辨能力。但同时也要切记增加元素的尺寸也会增加图像所占的内存空间。

（二）、如何扫描图像、利用查找表和计时

1.颜色空间缩减

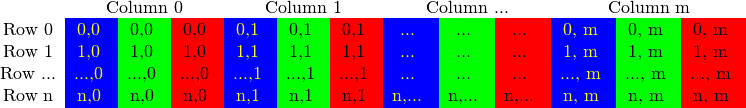
为了提高算法性能即效率，有时用三通道的颜色空间数量太庞大，有时还，可以使用这些颜色的一小部分，可以达到同样的效果。因此，使用颜色空间缩减，将现有颜色空间值除以某一输入值，以获得较少的颜色数。例如，颜色值0到9可取为新值0,10到19可取为10,依次类推。一般用以下公式对颜色进行缩减：

那么，颜色空间缩减算法就可以由以下两步组成：1.遍历图像矩阵的每一个像素（如0～256等）2.对像素应用上述公式。

由于上述公式使用乘除法，运算很费时，因此，我们使用表，来预先计算所有可能的像素值，然后需要这些值的时候，利用查找表直接赋值就可以了。

2.遍历图像的每一个像素，并查找像素对应的表中的缩减值（即扫描图像）

对多通道图像来说，矩阵中的列会包含多个子列，其子列个数与通道数相等。例如，RGB颜色模型的矩阵：



注意到，子列的通道顺序是反过来的：BGR而不是RGB。很多情况下，因为内存足够大，可实现连续存储，因此，图像中的各行就能一行一行地连接起来，形成一个长行。连续存储有助于提升图像扫描速度，我们可以使用 [isContinuous()](http://opencv.itseez.com/modules/core/doc/basic_structures.html" \l "mat-iscontinuous) 来去判断矩阵是否是连续存储的. 相关示例会在接下来的内容中提供。

（1）高效的方法（使用指针访问）

Mat& ScanImageAndReduceC(Mat& I, const uchar\* const table)

{

// accept only char type matrices

CV\_Assert(I.depth() != sizeof(uchar));

int channels = I.channels();

int nRows = I.rows;

int nCols = I.cols\* channels;

if (I.isContinuous())

{

nCols \*= nRows;

nRows = 1;

}

int i,j;

uchar\* p;

for( i = 0; i < nRows; ++i)

{

p = I.ptr<uchar>(i);

for ( j = 0; j < nCols; ++j)

{

p[j] = table[p[j]];

}

}

return I;

}

这里有另外一种方法来实现遍历功能，就是使用 data ， data会从 Mat 中返回指向矩阵第一行第一列的指针。注意如果该指针为NULL则表明对象里面无输入，所以这是一种简单的检查图像是否被成功读入的方法。当矩阵是连续存储时，我们就可以通过遍历 data 来扫描整个图像。例如，一个灰度图像，其操作如下：

uchar\* p = I.data;

for( unsigned int i =0; i < ncol\*nrows; ++i)

\*p++ = table[\*p];

（2）迭代法

在高性能法（the efficient way）中，我们可以通过遍历正确的 uchar 域并跳过行与行之间可能的空缺-你必须自己来确认是否有空缺，来实现图像扫描，迭代法则被认为是一种以更安全的方式来实现这一功能。在迭代法中，你所需要做的仅仅是获得图像矩阵的begin和end，然后增加迭代直至从begin到end。将\*操作符添加在迭代指针前，即可访问当前指向的内容。

Mat& ScanImageAndReduceIterator(Mat& I, const uchar\* const table)

{

// accept only char type matrices

CV\_Assert(I.depth() != sizeof(uchar));

const int channels = I.channels();

switch(channels)

{

case 1:

{

MatIterator\_<uchar> it, end;

for( it = I.begin<uchar>(), end = I.end<uchar>(); it != end; ++it)

\*it = table[\*it];

break;

}

case 3:

{

MatIterator\_<Vec3b> it, end;

for( it = I.begin<Vec3b>(), end = I.end<Vec3b>(); it != end; ++it)

{

(\*it)[0] = table[(\*it)[0]];

(\*it)[1] = table[(\*it)[1]];

(\*it)[2] = table[(\*it)[2]];

}

}

}

return I;

}

于彩色图像中的一行，每列中有3个uchar元素，这可以被认为是一个小的包含uchar元素的vector，在OpenCV中用 Vec3b 来命名。如果要访问第n个子列，我们只需要简单的利用[]来操作就可以。需要指出的是，OpenCV的迭代在扫描过一行中所有列后会自动跳至下一行，所以说如果在彩色图像中如果只使用一个简单的 uchar 而不是 Vec3b 迭代的话就只能获得蓝色通道(B)里的值。

（3）通过相关返回值的On-the-way地址计算

Mat& ScanImageAndReduceRandomAccess(Mat& I, const uchar\* const table)

{

// accept only char type matrices

CV\_Assert(I.depth() != sizeof(uchar));

const int channels = I.channels();

switch(channels)

{

case 1:

{

for( int i = 0; i < I.rows; ++i)

for( int j = 0; j < I.cols; ++j )

I.at<uchar>(i,j) = table[I.at<uchar>(i,j)];

break;

}

case 3:

{

Mat\_<Vec3b> \_I = I;

for( int i = 0; i < I.rows; ++i)

for( int j = 0; j < I.cols; ++j )

{

\_I(i,j)[0] = table[\_I(i,j)[0]];

\_I(i,j)[1] = table[\_I(i,j)[1]];

\_I(i,j)[2] = table[\_I(i,j)[2]];

}

I = \_I;

break;

}

}

return I;

}

该函数输入为数据类型及需求元素的坐标，返回的是一个对应的值-如果用 get 则是constant，如果是用 set 、则为non-constant. 处于程序安全，当且仅当在 debug 模式下 它会检查你的输入坐标是否有效或者超出范围. 如果坐标有误，则会输出一个标准的错误信息. 和高性能法（the efficient way）相比, 在 release模式下，它们之间的区别仅仅是On-the-fly方法对于图像矩阵的每个元素，都会获取一个新的行指针，通过该指针和[]操作来获取列元素.

当你对一张图片进行多次查询操作时，为避免反复输入数据类型和at带来的麻烦和浪费的时间，OpenCV 提供了:basicstructures:Mat\_ <id3> data type. 它同样可以被用于获知矩阵的数据类型，你可以简单利用()操作返回值来快速获取查询结果. 值得注意的是你可以利用 [at()](http://opencv.itseez.com/modules/core/doc/basic_structures.html" \l "mat-at) 函数来用同样速度完成相同操作. 它仅仅是为了让懒惰的程序员少写点 >\_< .

（4）核心函数LUT（The Core Function）

这是最被推荐的用于实现批量图像元素查找和更该操作图像方法。在图像处理中，对于一个给定的值，将其替换成其他的值是一个很常见的操作，OpenCV 提供里一个函数直接实现该操作，并不需要你自己扫描图像，就是:operationsOnArrays:LUT() <lut> ，一个包含于core module的函数. 首先我们建立一个mat型用于查表:

Mat lookUpTable(1, 256, CV\_8U);

uchar\* p = lookUpTable.data;

for( int i = 0; i < 256; ++i)

p[i] = table[i];

然后我们调用函数 (I 是输入 J 是输出):

LUT(I, lookUpTable, J);

3.算法性能比较

性能测试这里用的是彩色图片，结果是数百次测试的平均值.

|  |  |
| --- | --- |
| Efficient Way | 79.4717 milliseconds |
| Iterator | 83.7201 milliseconds |
| On-The-Fly RA | 93.7878 milliseconds |
| LUT function | 32.5759 milliseconds |

我们得出一些结论: 尽量使用 OpenCV 内置函数. 调用LUT 函数可以获得最快的速度. 这是因为OpenCV库可以通过英特尔线程架构启用多线程. 当然,如果你喜欢使用指针的方法来扫描图像，迭代法是一个不错的选择，不过速度上较慢。在debug模式下使用on-the-fly方法扫描全图是一个最浪费资源的方法，在release模式下它的表现和迭代法相差无几，但是从安全性角度来考虑，迭代法是更佳的选择