原 基于感知哈希算法的视觉目标跟踪

2013年12月21日 20:17:42 zouxy09 阅读数：48962

基于感知哈希算法的视觉目标跟踪

zouxy09@qq.com

http://blog.csdn.net/zouxy09

偶然看到这三篇博文[1][2][3]，提到图片检索网站TinEye和谷歌的相似图片搜索引擎的技术原理。以图搜图搜索引擎的使命是：你上传一张图片，然后他们尽全力帮你把互联网上所有与它相似的图片搜索出来。当然了，这只是他们认为的相似，所以有时候搜索结果也不一定对。事实上，以图搜图三大搜索引擎除了上面的老牌的TinEye和Google外，还有百度上线不算很久的新生儿：百度识图。之前听余凯老师的一个Deep Learning的讲座，里面很大一部分就介绍了百度识图这个产品，因为它是Deep Learning在百度成功上线的一个应用。里面详尽的把百度识图和谷歌的PK了一番。如果我没有听错和记错的话，余凯老师所介绍的百度识图也是应用了卷积神经网络CNN的，还有非常霸气的一点是：余凯老师说百度几乎都是监督学习！在厦门还是哪，有200人每天给百度标数据。这财力，气度全在上面了，没什么好说的了。

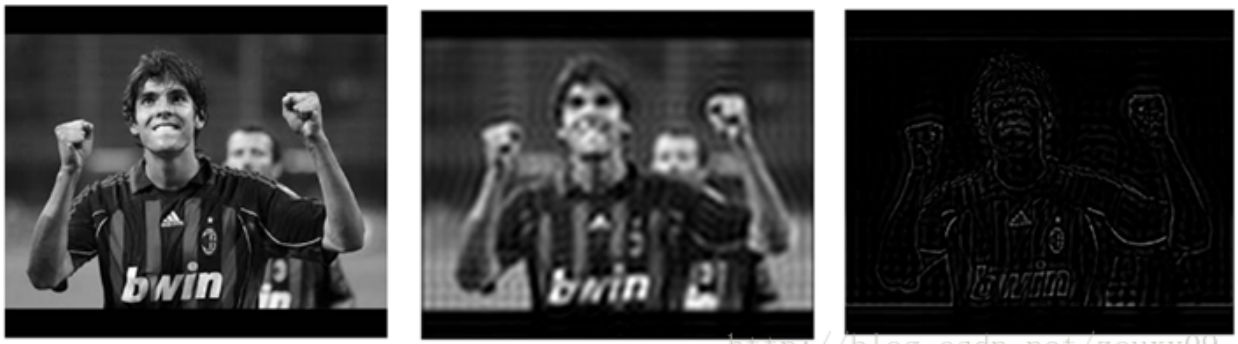
跑题了，我们回到这三篇博文提到的谷歌的以图搜图搜索引擎，博文中提到，这个网站提到了该引擎实现相似图片搜素的关键技术叫做“感知哈希算法”（Perceptual hash algorithm），它的作用是对每张图片生成一个“指纹”（fingerprint）字符串，然后比较不同图片的指纹。结果越接近，就说明图片越相似。（不知道是不是真的那么简单，哈哈）

但在这里，我考虑的不是图片检索，而是跟踪。因为既然它可以衡量两个图片的相似性，那么我就在想，那它就可以拿来做目标跟踪了，只要在每一帧找到和目标最相似的地方，那个就是目标了。这个和之前写的模板匹配的原理是差不多的，只是之前模板匹配采用的相似度度量是两个图片的相关性，这里用的是“hash指纹”。另外，详细的描述请参考上面三篇博文，这里先稍微总结下感知哈希算法的实现过程，然后给出自己简单实现目标跟踪的代码。

一、感知哈希算法

1、基于低频的均值哈希

一张图片就是一个二维信号，它包含了不同频率的成分。如下图所示，亮度变化小的区域是低频成分，它描述大范围的信息。而亮度变化剧烈的区域（比如物体的边缘）就是高频的成分，它描述具体的细节。或者说高频可以提供图片详细的信息，而低频可以提供一个框架。



原图　　低频成分　　高频成分

而一张大的，详细的图片有很高的频率，而小图片缺乏图像细节，所以都是低频的。所以我们平时的下采样，也就是缩小图片的过程，实际上是损失高频信息的过程。

均值哈希算法主要是利用图片的低频信息，其工作过程如下：

（1）缩小尺寸：去除高频和细节的最快方法是缩小图片，将图片缩小到8x8的尺寸，总共64个像素。不要保持纵横比，只需将其变成8*8的正方形。这样就可以比较任意大小的图片，摒弃不同尺寸、比例带来的图片差异。

（2）简化色彩：将8*8的小图片转换成灰度图像。

（3）计算平均值：计算所有64个像素的灰度平均值。

（4）比较像素的灰度：将每个像素的灰度，与平均值进行比较。大于或等于平均值，记为1；小于平均值，记为0。

（5）计算hash值：将上一步的比较结果，组合在一起，就构成了一个64位的整数，这就是这张图片的指纹。组合的次序并不重要，只要保证所有图片都采用同样次序就行了。(我设置的是从左到右，从上到下用二进制保存)。

 = 8f373714acfcf4d0

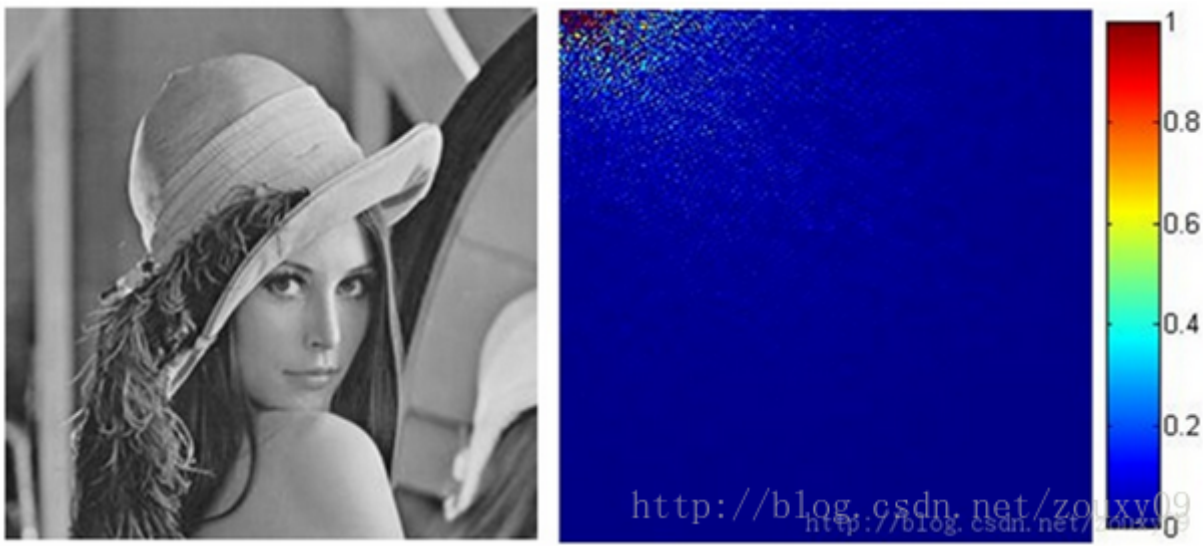
计算一个图片的hash指纹的过程就是这么简单。刚开始的时候觉得这样就损失了图片的很多信息了，居然还能有效。简单的算法也许存在另一种美。如果图片放大或缩小，或改变纵横比，结果值也不会改变。增加或减少亮度或对比度，或改变颜色，对hash值都不会太大的影响。最大的优点：计算速度快！

这时候，比较两个图片的相似性，就是先计算这两张图片的hash指纹，也就是64位0或1值，然后计算不同位的个数(汉明距离)。如果这个值为0，则表示这两张图片非常相似，如果汉明距离小于5，则表示有些不同，但比较相近，如果汉明距离大于10则表明完全不同的图片。

2、增强版：pHash

均值哈希虽然简单，但受均值的影响非常大。例如对图像进行伽马校正或直方图均衡就会影响均值，从而影响最终的hash值。存在一个更健壮的算法叫pHash。它将均值的方法发挥到极致。使用离散余弦变换(DCT)来获取图片的低频成分。

离散余弦变换（DCT）是种图像压缩算法，它将图像从像素域变换到频率域。然后一般图像都存在很多冗余和相关性的，所以转换到频率域之后，只有很少的一部分频率分量的系数才不为0，大部分系数都为0（或者说接近于0）。下图的右图是对lena图进行离散余弦变换（DCT）得到的系数矩阵图。从左上角依次到右下角，频率越来越高，由图可以看到，左上角的值比较大，到右下角的值就很小很小了。换句话说，图像的能量几乎都集中在左上角这个地方的低频系数上面了。



pHash的工作过程如下：

（1）缩小尺寸：pHash以小图片开始，但图片大于8*8，32*32是最好的。这样做的目的是简化了DCT的计算，而不是减小频率。

（2）简化色彩：将图片转化成灰度图像，进一步简化计算量。

（3）计算DCT：计算图片的DCT变换，得到32*32的DCT系数矩阵。

（4）缩小DCT：虽然DCT的结果是32*32大小的矩阵，但我们只要保留左上角的8*8的矩阵，这部分呈现了图片中的最低频率。

（5）计算平均值：如同均值哈希一样，计算DCT的均值。

（6）计算hash值：这是最主要的一步，根据8*8的DCT矩阵，设置0或1的64位的hash值，大于等于DCT均值的设为”1”，小于DCT均值的设为“0”。组合在一起，就构成了一个64位的整数，这就是这张图片的指纹。

结果并不能告诉我们真实性的低频率，只能粗略地告诉我们相对于平均值频率的相对比例。只要图片的整体结构保持不变，hash结果值就不变。能够避免伽马校正或颜色直方图被调整带来的影响。

与均值哈希一样，pHash同样可以用汉明距离来进行比较。(只需要比较每一位对应的位置并算计不同的位的个数)

二、基于感知哈希算法的视觉跟踪

和前面说的那样，对于感知哈希算法的视觉跟踪，思想很简单，我们把要跟踪的目标保存好，计算它的hash码，然后在每一帧来临的时候，我们扫描整个图像，计算每个扫描窗口的hash码，比较它和目标的hash码的汉明距离，汉明距离距离最小的扫描窗口就是和目标最相似的，也就是该帧的目标所在位置。为了加速，我们只在上一帧目标的周围图像区域进行扫描。为了适应目标的变化，我们还需要在成功跟踪后的每一帧更新我们要跟踪的目标。

当时看到这个东西的时候，感觉很简单，然后就花了点时间动手写了下代码，不知道代码是否正确，如有错误，还望大家不吝指点。我的代码是基于VS2010+ OpenCV2.4.2的。基础的均值哈希和pHash都实现了，切换只需要在改变代码里面跟踪的那个函数的flag即可。代码可以读入视频，也可以读摄像头，两者的选择只需要在代码中稍微修改即可。对于视频来说，运行会先显示第一帧，然后我们用鼠标框选要跟踪的目标，然后跟踪器开始跟踪每一帧。对摄像头来说，就会一直采集图像，然后我们用鼠标框选要跟踪的目标，接着跟踪器开始跟踪后面的每一帧。具体代码如下：

hashTracker.cpp

```
// Object tracking algorithm using Hash or pHash code
// Author : zouxy
// Date   : 2013-12-21
// HomePage : http://blog.csdn.net/zouxy09
// Email  : zouxy09@qq.com
```

```
#include <opencv2/opencv.hpp>

using namespace cv;
using namespace std;

// Global variables
Rect box;
bool drawing_box = false;
bool gotBB = false;

// bounding box mouse callback
void mouseHandler(int event, int x, int y, int flags, void *param){
  switch( event ){
  case CV_EVENT_MOUSEMOVE:
    if (drawing_box){
        box.width = x-box.x;
        box.height = y-box.y;
    }
    break;
  case CV_EVENT_LBUTTONDOWN:
    drawing_box = true;
    box = Rect( x, y, 0, 0 );
    break;
  case CV_EVENT_LBUTTONUP:
    drawing_box = false;
    if( box.width < 0 ){
        box.x += box.width;
        box.width *= -1;
    }
    if( box.height < 0 ){
        box.y += box.height;
        box.height *= -1;
    }
    gotBB = true;
    break;
  }
}

// calculate the hash code of image
Mat calHashCode(Mat image)
{
        resize(image, image, Size(8, 8));
        Scalar imageMean = mean(image);
        return (image > imageMean[0]);
}
```

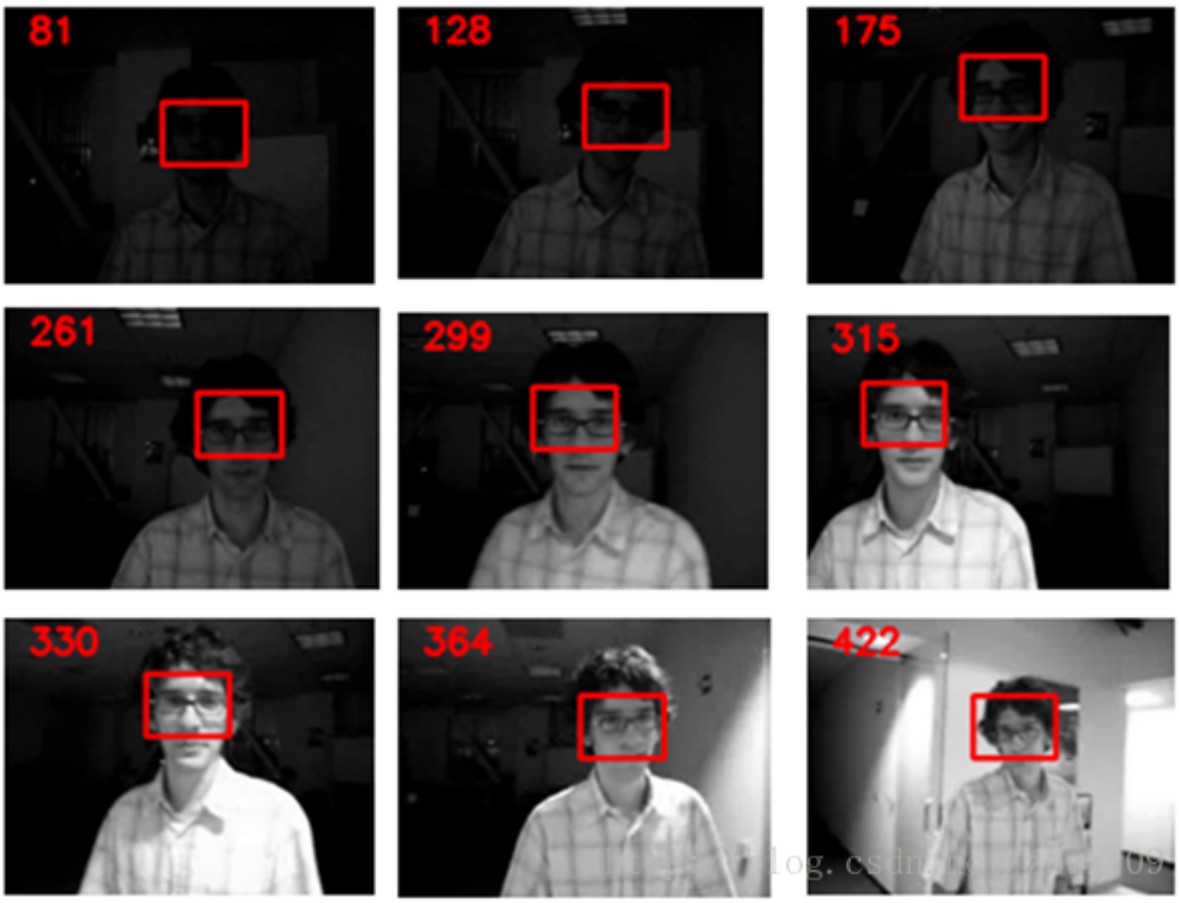
```
// calculate the pHash code of image
Mat calPHashCode(Mat image)
{
	Mat floatImage, imageDct;
	resize(image, image, Size(32, 32));
	image.convertTo(floatImage, CV_32FC1);
	dct(floatImage, imageDct);
	Rect roi(0, 0, 8, 8);
	Scalar imageMean = mean(imageDct(roi));
	return (imageDct(roi) > imageMean[0]);
}

// get hamming distance of two hash code
int calHammingDistance(Mat modelHashCode, Mat testHashCode)
{
	return countNonZero(modelHashCode != testHashCode);
}

// tracker: get search patches around the last tracking box,
// and find the most similar one using hamming distance
void hashTrack(Mat frame, Mat &model, Rect &trackBox, int flag = 0)
{
	Mat gray;
	cvtColor(frame, gray, CV_RGB2GRAY);

	Rect searchWindow;
	searchWindow.width = trackBox.width * 3;
	searchWindow.height = trackBox.height * 3;
	searchWindow.x = trackBox.x + trackBox.width * 0.5 - searchW
	searchWindow.y = trackBox.y + trackBox.height * 0.5 - searcl
	searchWindow &= Rect(0, 0, frame.cols, frame.rows);

	Mat modelHashCode, testHashCode;
	if (flag)
		modelHashCode = calHashCode(model);
	else
		modelHashCode = calPHashCode(model);
	int step = 2;
	int min = 1000;
	Rect window = trackBox;
	for (int i = 0; i * step < searchWindow.height - trackBox.he
	{
		window.y = searchWindow.y + i * step;
		for (int j = 0; j * step < searchWindow.width - trac
		{
			window.x = searchWindow.x + j * step;
			if (flag)
				testHashCode = calHashCode(gray(wir
```

```
                        else
                                testHashCode = calPHashCode(gray(windo
                    int distance = calHammingDistance(modelHashCode
                    if (distance < min)
                                trackBox = window;
                                min = distance;
                        }
                }
        }
        model = gray(trackBox);
        cout << "The min hanming distance is: " << min << endl;
}

int main(int argc, char * argv[])
{
        VideoCapture capture;
        // from video
        capture.open("david.mpg");
        bool fromfile = true;

        // from camera
        //capture.open(0);
        //bool fromfile = false;

        //Init camera
        if (!capture.isOpened())
        {
                cout << "capture device failed to open!" << endl;
                return -1;
        }
        //Register mouse callback to draw the bounding box
        cvNamedWindow("hashTracker", CV_WINDOW_AUTOSIZE);
        cvSetMouseCallback("hashTracker", mouseHandler, NULL );

        Mat frame, model;
        capture >> frame;
        while(!gotBB)
        {
                if (!fromfile)
                        capture >> frame;

                imshow("hashTracker", frame);
                if (cvWaitKey(20) == 'q')
                        return 1;
        }
        //Remove callback
```

```
        cvSetMouseCallback("hashTracker", NULL, NULL );
        Mat gray;
        cvtColor(frame, gray, CV_RGB2GRAY);
        model = gray(box);

        int frameCount = 0;
        while (1)
        {
                capture >> frame;
                if (frame.empty())
                        return -1;
                double t = (double)cvGetTickCount();
                frameCount++;

                // tracking
                hashTrack(frame, model, box, 0);

                // show
                stringstream buf;
                buf << frameCount;
                string num = buf.str();
                putText(frame, num, Point(20, 30), FONT_HERSHEY_SIM
                rectangle(frame, box, Scalar(0, 0, 255), 3);
                imshow("hashTracker", frame);


                t = (double)cvGetTickCount() - t;
                cout << "cost time: " << t / ((double)cvGetTickFrequenc

                if ( cvWaitKey(1) == 27 )
                        break;
        }

        return 0;
}
```

三、实验结果

我们还是和之前一样，用在目标跟踪领域一个benchmark的视频-david来测试下代码的效果。如下图所以，每帧的帧号在左上角所示。这里的初始框是我随意画的，所以你的结果和我的有可能不同。下图的结果是使用pHash的，pHash比均值hash要好，但耗时也增加了不少。另外，我的代码没有经过优化的，写着玩嘛，哈哈。



四、思考

看到这个算法的时候，第一个感觉就是，这太简单了吧，它真的有效吗？像下图左那样，它的hash值的图压根就看不出是个什么东西了，居然还能做相似的匹配，而且一定情况下，还是挺有效的。

这种简单的比较得到0和1编码还让我想到了经典的LBP特征，如上图右，不同在于LBP是每个像素点与邻域比较，而hash是与整幅图的均值比较。所以LBP可以保存明暗这种过渡的边缘，而hash保存的是图像整体的精简版的低频分量。

这也让人困惑在简单与复杂的抉择之间，它们的考量也非三言两语能避之。也许算法之美一定程度上能从其简单和有效处得以瞥见吧。

另外，我还特意检索了一下，暂时还没有搜到基于感知哈希算法的视觉跟踪，不知道会不会对大家有所启发。（发论文的，求挂名哦，哈哈^-^）

五、参考文献：

[1] Google 以图搜图 - 相似图片搜索原理 -Java实现

[2] 看起来像它——图像搜索其实也不难

[3] 相似图片搜索的原理

[4] 最简单的目标跟踪（模版匹配）