OPENCV图像处理

resize操作

1.直接裁剪，则直接进行

```
def cv_show(name, image):
    cv2.imshow(name, image)
    cv2.waitKey(0)
    cv2.destoryAllWindows()


image =
cv2.imread(r"C:\Users\jajnw\Desktop\datasets\coco128\images\train2017\000000000009.jpg")
img = image[400:600, 0:200]
cv_show('man', img)
```

这样子就能进行原图直接裁剪

resize

2.进行放大

在原图较小的时候我们可能希望将其放大，而放大的时候就会涉及到填充原本不存在的像素块

3.进行缩小

缩小的时候跟直接裁剪不同，我们需要进行等比例地缩小再进行处理让他像素合理

这两个部分都使用resize

2.1 INTER_NEAREST（最近邻插值）
 最近邻插值是最简单的插值方法，选取离目标点最近的点作为新的插入点，计算公式表示如下：

插值后的边缘效果：由于是以最近的点作为新的插入点，因此边缘不会出现缓慢的渐慢过度区域，这也导致放大的图像容易出现锯齿的现象

2.2 INTER_CUBIC (三次样条插值)

插值后的边缘效果：可以有效避免出现锯齿的现象

2.3 INTER_LINEAR(线性插值)

 线性插值是以距离为权重的一种插值方式。

2.4 INTER_AREA (区域插值)
 区域插值共分三种情况，图像放大时类似于双线性插值，图像缩小(x轴、y轴同时缩小)又分两种情况，此情况下可以避免波纹出现。因此对图像进行缩小时，为了避免出现波纹现象，推荐采用区域插值方法。

总结

OpenGL说明文档有这么解释：To shrink an image, it will generally look best with #INTER_AREA interpolation, whereas to enlarge an image, it will generally look best with #INTER_CUBIC (slow) or #INTER_LINEAR (faster but still looks OK).

如果要缩小图像，通常推荐使用INTER_AREA插值效果最好，而要放大图像，通常使用INTER_CUBIC(速度较慢，但效果最好)，或者使用INTER_LINEAR(速度较快，效果还可以)。

copyMakeBorder操作

如果你想给你的图片设置边界框，就像一个相框一样的东西，你就可以使用`cv2.copyMakeBorder()`函数。但其在卷积操作、零填充等也得到了应用，并且可以用于一些数据增广操作。

cv2.copyMakeBorder(img, top_size, bottom_size, left_size, right_size, borderType=cv2.BORDER_REPLICATE)

BORDER_REPLICATE：复制法，也就是复制最边缘像素。
BORDER_REFLECT：反射法，对感兴趣的图像中的像素在两边进行复制例如：
fedcba|abcdefgh|hgfedcb
BORDER_REFLECT_101：反射法，也就是以最边缘像素为轴，对称，gfedcb|abcdefgh|gfedcba
BORDER_WRAP：外包装法cdefgh|abcdefgh|abcdefg
BORDER_CONSTANT：常量法，常数值填充，需要在函数中填入常数参数。

transpose操作

即将矩阵沿着对角线进行翻转。

transpose()可以实现像素下标的x和y轴坐标进行对调：dst(i,j)=src(j,i)

换句话是逆时针90然后上下翻转

rotate操作

rotate函数能够按照顺时针的方法以三种不同的角度进行旋转，三种角度分别是`90度，180度，270度`。
rotateCode参数可以取枚举类型RotateFlags中的值，`0表示90度，1表示180度，2表示270度`

```
cv_show("rotate_180_frame", cv2.rotate(image, 1))
```

def rotate(src, rotateCode, dst=None)

flip操作

```
def flip(src, flipCode, dst=None)
```

- src：输入图像
- flipCode：flipCode 一个标志来指定如何翻转数组；0表示上下翻转，正数表示左右翻转，负数表示上下左右都翻转。
- dst：输出图像

cvtcolor操作

`cv2.cvtColor()` 方法用于将图像从一种颜色空间转换为另一种颜色空间。

将所有图片一次性转换为 tensor 进行处理（批处理）能更高效地利用硬件加速，减少数据传输时间，适用于深度学习任务。这种方法有助于保持数据一致性并优化内存使用。然而，如果图片尺寸不一致，可能需要在批处理中对图片进行填充或裁剪，这可能导致处理时间增加。

拼接图片后再输入模型可以简化数据处理，但可能需要处理拼接后的图片尺寸问题。如果图片本身较大或拼接后尺寸过大，可能会导致内存消耗增大。整体来说，批处理更具通用性和灵活性，而拼接方法则适用于特定场景下的数据整合。

addWeighted融合

是将两张相同大小，相同类型的图片融合的函数。

void cvAddWeighted(const CvArr* src1, double alpha,const CvArr* src2, double beta,double gamma, CvArr* dst);
参数1：src1，第一个原数组.
参数2：alpha，第一个数组元素权重

参数3：src2第二个原数组
参数4：beta，第二个数组元素权重
参数5：gamma，图1与图2作和后添加的数值。不要太大，不然图片一片白。总和等于255以上就是纯白色了。

参数6：dst，输出图片
如果2个数值直接相加的结果大于255就会赋值为255，在后面介绍图像减法时也会看到如果2个数值直接相减的结果*小于0就会赋值为0*。

新图像的结果可以表示为：dst(I)=saturate(src1(I)∗alpha+src2(I)∗beta+gamma)。

没看明白gamma有啥用，不必理会

拼接操作

纵向合并

vstack

```
image = np.vstack((img1, img2))
```

横向合并

```
image = np.concatenate([img1, img2], axis=1)
```

要拼接之前保证拼接的相同边要尺寸一致

颜色通道提取RGB分离

```
(B,G,R) = cv2.split(image)
image = cv2.merge([B,G,R])
cv2.imshow('image',image)
cv2.imshow("Red",R)
cv2.imshow("Green",G)
cv2.imshow("Blue",B)
```