计算机视觉实践报告

目录

一、K-Means 聚类算法

1. 基本思想

首先生成 k 个随机样本中心，之后通过迭代的方式，每轮迭代将样本点分配给离它最近的样本中心，从而确定每一个样本所属的类别。之后对于每一类别的所有样本，重新计算它们的中心。重复迭代过程，直至 k 个样本中心的位置不再变化为之。

2.实现过程

```python
def kmeans(features,k,num_iters=200):
    N,D=features.shape  #获取图像特征的像素个数以及每个像素点的通道数
    idxs=np.random.choice(N,size=k,replace=False)   #随机生成K个随机数
    centers=features[idxs]  #使用指定的随机数对应的像素点作为初始聚类中心
    assignments=np.zeros(N) #将每个像素点对应所属的类别初始化为0
    for n in range(num_iters):
        featureMap=np.tile(features,(k,1))  #将整个图像特征重复k次
        centerMap=np.repeat(centers,N,axis=0)
        #将每一个聚类中心重复像素点个数次，从而使得featureMap和centerMap形成了所有的像素-聚类中心对应关系
        dist=np.linalg.norm(featureMap-centerMap,axis=1).reshape(k,N)
        #通过对featureMap和centerMap每一个点（是像素或聚类中心）的做差结果求L2范数的方式计每一组像素和聚类中心之间的距离
        assignments=np.argmin(dist,axis=0)  #对每个像素找出离其最近的聚类中心的下标
        newCenters=np.zeros((k,D))  #初始化新的聚类中心
        for idx in range(k):
            index=np.where(assignments==idx)    #筛选出所有属于第idx类的像素
            newCenters[idx]=np.mean(features[index],axis=0)  #计算第idx类像素的新聚类中心
        if np.allclose(newCenters,centers): #如果新的中心点和上一次迭代的聚类中心位置完全相同则可以提前退出
            break
        else:
            centers=newCenters    #更新新的聚类中心
    return assignments
```

3.性能优化

如果通过循环的方式遍历每个样本，找出与其相距最近的样本中心，则会使得效率十分低下。为了提高效率，可以使用 numpy 的矩阵操作，具体做法为：将整个样本（共 N 个）重复 k 次得到矩阵 A，将所有样本中心（共 k 个）重复 N 次得到矩阵 B，则矩阵 A 和 B 每个位置上的元素对应关系刚好遍历所有“样本-样本中心”对应关系。通过矩阵 A 和矩阵 B 做差，在对得到的结果求 L2 范数，就可以得到所有“样本-样本中心”的距离。之后通过 argmin 函数就可以对每个样本快速筛选出离它最近的样本中心。

二.HAC 聚类算法

1.基本思想

在 HAC 聚类中，每个样本点最初都单独放置在一个 cluster 中。之后通过迭代的方式，在每次迭代中中，将具有最高相似度得分的两个 cluster 合并。从而使得 cluster 数量减少 1。直至最后剩余 cluster 的数量达到了预期数量。其中相似度可以有如下计算方式：

①单链接：将两个 cluster 中距离最近的两个样本点间的距离作为这两个 cluster 的距离。这种方法容易受到极端值的影响。两个不相似的 cluster 点可能由于其中的某个极端的样本点 距离较近而合并在一起。单链接相似度计算方法为：

$$sim(A,B)=\min_{u\in A,v\in B} sim(u,v)$$

②完全链接：将两个 cluster 中距离最远的两个样本点间的距离作为这两个组合数据点的距离。完全链接的问题与单链接相反，两个相似的 cluster 可能由于其中的极端样本点距离较远而无法合并在一起。完全链接相似度计算方法为：

$$sim(A,B)=\max_{u\in A,v\in B} sim(u,v)$$

③平均链接：计算两个 cluster 中的每个样本点与其他所有样本点的距离。将所有距离

的均值作为两个 cluster 间的距离。这种方法虽然结果比前两种方法更合理，但是其计算量非常大。平均链接相似度计算方法为：

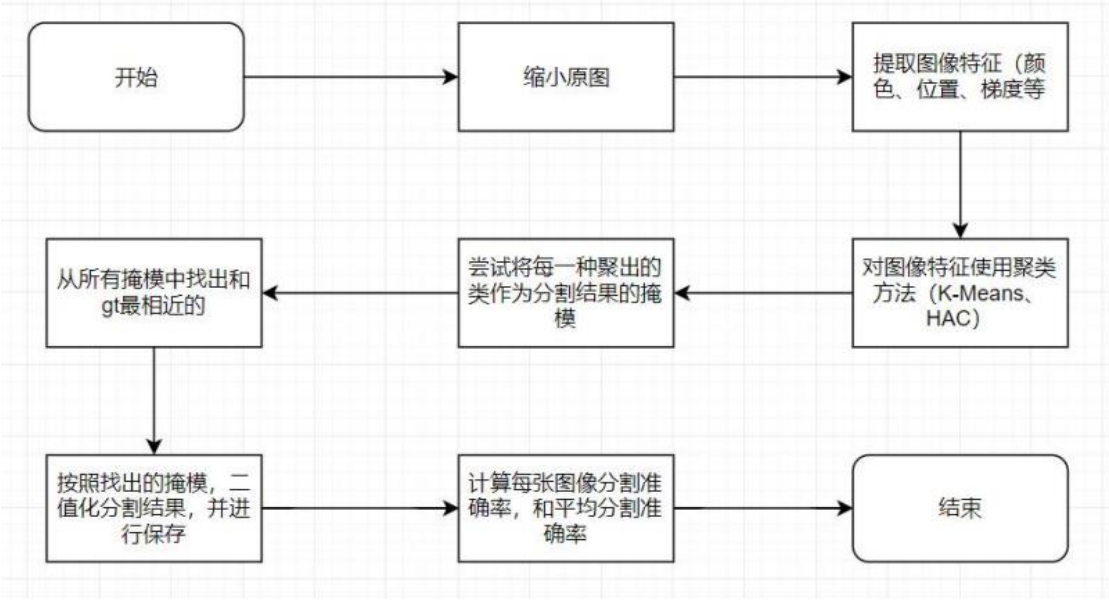
$$sim(A,B)=\frac{\sum_{u\in A,v\in B} sim(u,v))}{size(A)*size(B)}$$

2.实现过程

```
def hierarchical_clustering(features,k):
    N,D=features.shape  #获取图像特征的像素个数以及每个像素点的通道数
    assignments=np.arange(N)    #初始每个像素自身就是一个独立的簇
    centers=np.copy(features)   #初始每个像素自身就是所在簇的中心
    n_clusters=N
    while n_clusters>k:
        dist=pdist(centers) #计算所有簇的中心间的欧氏距离
        distMatrix=squareform(dist) #将对角阵转换为完整矩阵
        distMatrix=np.where(distMatrix!=0.0,distMatrix,1e5)
        # 因为squareform构建的完整矩阵上对角线是零，影响了最短距离的判断，此处将0变为1e5，从而避免影响
        #获取最小值所在的行和列，即距离最近的两个簇的index，取较小的那个作为保留，较大的那个进行合并
        minRow,minCol=np.unravel_index(distMatrix.argmin(),distMatrix.shape)
        saveIdx=min(minRow,minCol)
        mergeIdx=max(minRow,minCol)
        #将簇mergeIdx的点分配给saveIdx所在簇
        assignments=np.where(assignments!=mergeIdx,assignments,saveIdx)
        #因为要删除一个簇mergeIdx，所以下标的变化为:小于mergeIdx的不改变，大于mergeIdx的需要减一
        assignments=np.where(assignments<mergeIdx,assignments,assignments-1)
        #删除被合并的簇所在中心
        centers=np.delete(centers,mergeIdx,axis=0)
        #对新合并得到的簇，计算新的簇的中心
        saveIdxIndecies=np.where(assignments==saveIdx)
        centers[saveIdx]=np.mean(features[saveIdxIndecies],axis=0)
        n_clusters-=1
    return assignments
```

三.聚类算法实现图像分割

1.实验步骤



①缩小原图：

（1）由于图像本身具有一定噪声，因此希望通过将图像缩小的方式减轻噪声对聚类结果的影响。在缩小后的图像上进行聚类，完毕后通过线性插值的方式将图像恢复至原本大小，并对插值出的浮点数类型四舍五入取整，从而将插值出的像素也精确分属到所聚出的某一类之中。

（2）缩小图像能够有效减少聚类时的计算量。尤其是在 HAC 聚类算法中，由于初始时每个像素就是一个独立的 cluster，在计算 cluster 之间距离时需要对所有像素两两计算距离。无论是空间还是时间都将花费巨大的开销。通过缩小图像的方式，能够平方级减少计算量。实现如下：

```
def compute_segmentation(img,k,clustering_fn=kmeans,feature_fn=color_position_features,scale=0):
    H,W,C=img.shape #获取图像的高度，宽度和通道数
    if scale>0:
        img=transform.rescale(img,scale,multichannel=True)  #将图像进行缩小处理，加快计算速度
    features=feature_fn(img)    #获取图像的特征图
    assignments=clustering_fn(features,k)   #获取图像的聚类结果
    segments=assignments.reshape((img.shape[:2]))   #将分类结果重塑成和原图像像素点一一对应的形式
    if scale>0:
        segments=transform.resize(segments,(H,W),preserve_range=True)  #将分割后的图像还原至原本大小
        segments=np.rint(segments).astype(int)  #将线性插值得到的聚类结果四舍五入成整型
    return segments
```

②提取图像特征：

（1）颜色特征：仅将图像的每个像素点的 RGB 分量作为样本点的属性进行聚类。

（2）颜色-位置特征：在提取颜色特征的基础上，考虑像素点的坐标位置，共同作为样本点属性进行聚类。同时由于坐标位置数值和 RGB 分量值数量级相差较大，因此需要对特征做归一化处理。

（3）颜色-位置-梯度特征：在提取颜色和位置特征的基础上，考虑像素点水平和垂直方向

上的灰度梯度特征，将两者加和，共同作为样本点的属性进行聚类。同样需要归一化处理。实现如下：

```
def color_features(img):
    H,W,C=img.shape #获取图像的高度，宽度和通道数
    img=img_as_float(img) #将图像像素转换成浮点数表示
    features=np.reshape(img, (H*W, C))  #将图像的每个像素的所有颜色通道作为特征通道，得到对应图像大小（高度、宽度）的图像特征
    return features

def color_position_features(img):
    H,W,C=img.shape #获取图像的高度，宽度和通道数
    color=img_as_float(img) #将图像像素转换成浮点数表示
    features=np.zeros((H*W, C+2))  #预留出两个通道的位置，用于存放像素的位置特征
    position=np.dstack(np.mgrid[0:H,0:W]).reshape((H*W,2))  #得到每个像素的位置信息
    features[:,0:C]=np.reshape(color,(H*W,C))    #将图像的每个像素的所有颜色通道作为特征通道
    features[:,C:C+2]=position  #特征通道再额外拼接上各个像素的位置信息这两个通道
    features=(features-np.mean(features,axis=0))/(np.std(features, axis=0)) #对图像特征中各个元素每个通道的数值进行中心化
    return features

def color_position_gradient_features(img):
    H,W,C=img.shape #获取图像的高度，宽度和通道数
    color=img_as_float(img) #将图像像素转换成浮点数表示
    position=np.dstack(np.mgrid[0:H,0:W]).reshape((H*W,2))  #得到每个像素的位置信息
    grayImg=color.rgb2gray(img) #将图像换成灰度图形式
    gradient=np.gradient(grayImg)   #计算图像每个像素在水平和垂直方向上的梯度值
    gradient=np.abs(gradient[0])+np.abs(gradient[1])    #将两个方向上的梯度值进行相加
    features=np.zeros((H*W,C+3))
    features[:,0:C]=np.reshape(color,(H*W,C))    #将图像的每个像素的所有颜色通道作为特征通道
    features[:,C:C+2]=position  #特征通道再额外拼接上各个像素的位置信息这两个通道
    features[:,C+2]=gradient.reshape((H*W))  #特征通道再额外拼接上各个像素的两个方向上的梯度值之和这个通道
    features=(features-np.mean(features,axis=0))/(np.std(features, axis=0)) #对图像特征中各个元素每个通道的数值进行中心化
    return features
```

③聚类：

使用 K-Means 或 HAC 方法对所提取出的图像特征进行聚类，从而将图像划分出不同的部分。此处存在超参数 k，即将图像所要分成的部分个数。经过试验，取 k=3 时效果较好。 且由于 HAC 聚类算法的计算复杂程度比 K-Means 更高，想要在不超出内存限制的情况下可接受时间内得到图像分割结果，则 HAC 聚类方法接受到的图像特征规模应当比 K-Means 聚类方法所接受的小 1 个数量级左右。此处 K-Means 聚类方法缩小图像为 0.5 倍，HAC 聚类方法缩小图像为 0.01 倍（缩小为 0.05 倍的情况下，16 张图片将有 2 张会在计算过程中超出内存限制）。具体实现方法已在前文中展示。

④图像掩模：

由于 GT 中的图像，将图像分割成了黑白两个部分，而我在聚类时选取的 k=3，因此需要从聚类结果中选取一个类作为“目标”，和 GT 中的“目标”（即图像的白色部分）进行比对。对我所聚出的类进行遍历，对于每个类，将该类下的像素变成白色，其余像素变成黑色，从而得到“目标”的掩模，使用该掩模可以和 GT 进行准确度对比。

实现如下：

```
def evaluate_segmentation(mask_gt, segments):
    num_segments=np.max(segments)+1 #将图像分割成的部分总数
    max_accuracy=0  #初始化准确率最大的分割部分的准确率
    for i in range(num_segments):
        mask=(segments==i).astype(int)  #获取所有被分割成第i部分的像素形成的掩模，转换成相当于只有这部分是白色，其余为黑色的分割结果
        accuracy=compute_accuracy(mask_gt,mask) #对比选取分割出的白的为当前掩模部分的情况下，和gt的重合率
        max_accuracy=max(accuracy,max_accuracy) #和gt重合率最大的即为准确率最高的分割方案
    for i in range(num_segments):
        mask=(segments==i).astype(int)  #获取所有被分割成第i部分的像素形成的掩模，转换成相当于只有这部分是白色，其余为黑色的分割结果
        accuracy=compute_accuracy(mask_gt,mask) #对比选取分割出的白的为当前掩模部分的情况下，和gt的重合率
        if (accuracy==max_accuracy):
            return max_accuracy,mask
```

⑤分割准确率： 为了定量化显示用提取的图像特征序列进行图像分割的实际效果，将获得

的各个类下的“目标”掩模和 GT“目标”进行重合，统计二者对应相同位置的像素颜色一致的部分所占整个图像的比例。从各个类的“目标”掩模中选取该占比最高的数值作为图像分割的准确率。实现如下（其中取最高值操作是在 evaluate_segmentation 函数中完成）：

```
def compute_accuracy(mask_gt,mask):
    accuracy=np.mean(mask==mask_gt)    #计算分割方案和gt的重合度，作为该种分割方案的准确度
    return accuracy
```

2.实验说明

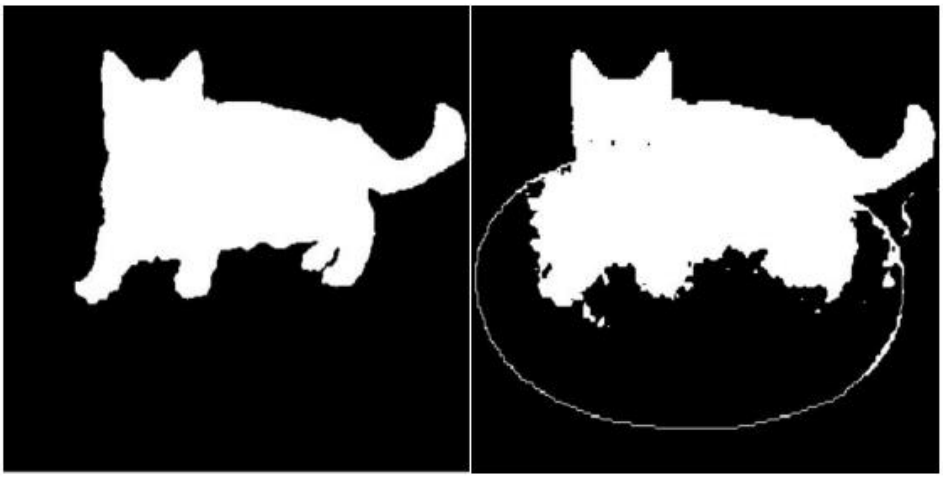
本次作业通过在 Visual Studio Code 使用 Anaconda 环境对编写的代码进行解释执行，其中使用到的 numpy 等库均已在 Anaconda 环境中安装。通过控制台命令 python -u “文件路径 \文件名.py” 可以对源程序进行解释执行。从 data/imgs 目录和 data/gt 目录分别获取到原图像和 GT，通过 zip 函数组成一组“输入-标准输出”对，在运行图像分割后，将自己得到的输出与标准输出（GT）进行比对，从而能够得到分割准确率。并且将自己得到的输出，按照准确率最高的掩模，以黑白图像的形式保存在 data/my/Kmeans 和 data/my/HAC 目录下作为结果记录。 通过所保存的图像文件来看，HAC 聚类容易聚出横平竖直的块状和条状区域，并且由于 HAC 聚类计算量大而导致对输入的缩小比例更高，最终导致 HAC 聚类所获得的图像风格结果结构较为简单，不如 K-Means 聚类算法获得的结果好（并且从准确率上来看也是如此，然而如果内存充足，可以预见在缩小同样比例的情况下，HAC 聚类比 K-Means 聚类得到的图像风格结果准确率应该更高）。

3.实验结果

①使用 K-Means 聚类方法实现的图像分割，准确率如下图所示：

（最好情况下分割准确率 81.04%）

部分原图和 GT-分割结果对比展示如下：

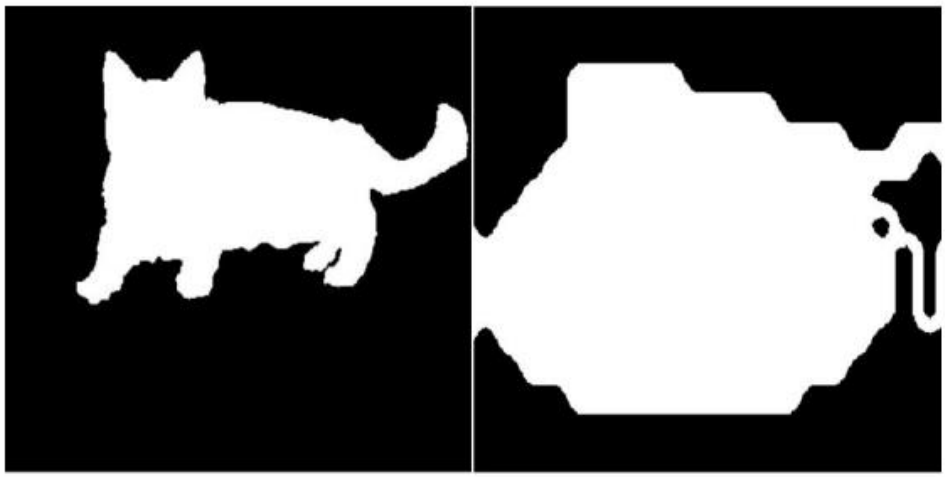


②使用 HAC 聚类方法（使用单链接方式）实现的图像分割，准确率如下图所示：

```
第1张图像的准确率为：0.7743
第2张图像的准确率为：0.7686
第3张图像的准确率为：0.8695
第4张图像的准确率为：0.9590
第5张图像的准确率为：0.7780
第6张图像的准确率为：0.6888
第7张图像的准确率为：0.5982
第8张图像的准确率为：0.7317
第9张图像的准确率为：0.8485
第10张图像的准确率为：0.8523
第11张图像的准确率为：0.6709
第12张图像的准确率为：0.6402
第13张图像的准确率为：0.7406
第14张图像的准确率为：0.6089
第15张图像的准确率为：0.8039
第16张图像的准确率为：0.5382
所有图像的平均准确率为：0.7420
```

（最好情况下分割准确率 74.20%）

部分 GT-分割结果展示如下：



四．图像拼接

1．SIFT 特征点检测，得到特征点以及特征向量

如果要实现图像之间的特征点匹配，要通过特征描述子集之间比对完成。常见的匹配器有暴力匹配器和快速近似最邻近算法匹配器。暴力匹配器就是将两幅图像中的特征描述子全都匹配一遍，得到最优的结果，它的优点是精度高，但是缺点也是显而易见的，在大量的匹配时，匹配时间会很长。快速近似最邻近算法匹配器，故名思意，它只搜索邻近的点，找到邻近的最优匹配，它的匹配准确度会比暴力匹配器低，但是它的匹配时间大大的缩减了。

- **cv2.xfeatures2d.SIFT_create(nfeatures=None,nOctaveLayers=None,contrastThreshold=None, edgeThreshold=None, sigma=None)-->descriptor**

函数用于创建一个用于提取 SIFT 特征的描述符
* nfeatures，保留的最佳特性的数量。特征按其得分进行排序(以 SIFT 算法作为局部对比度进行测量)；
* nOctavelLayers，高斯金字塔最小层级数，由图像自动计算出；
* constrastThreshold，对比度阈值用于过滤区域中的弱特征。阈值越大，检测器产生的特征

越少；
* edgeThreshold ，用于过滤掉类似边缘特征的阈值。 请注意，其含义与 contrastThreshold 不同，即 edgeThreshold 越大，滤出的特征越少
* sigma，高斯输入层级， 如果图像分辨率较低，则可能需要减少数值

```
def detectAndDescribe(self, image):
        # 转换为灰度图
        gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)

        # 建立SIFT生成器
        descriptor = cv2.xfeatures2d.SIFT_create()
        # 检测特征点并计算描述子
        kps, features = descriptor.detectAndCompute(gray, None)

        kps = np.float32([kp.pt for kp in kps])

        return kps, features
```

通过 SIFT_create()实例化 SIFT 对象，调用 detectAndCompute 方法得到特征点和特征向量

2. 使用 KNN 得到和图像 A 中一个点最接近的前 K 的点在 B 中的位置

这里使用 RANSAC 方法，找到和一个点在另一张图片上最相似的两个点，如果第一个点的近似程度远远大于第二个点，那么才认为第一个点是可靠的匹配点。

- **rawMatches = matcher.knnMatch(featuresA, featuresB, 2)**

参数：
* featureA 和 featureB 表示两个特征向量集；
* K 表示按 knn 匹配规则输出的最优的 K 个结果
在该算法中，输出的结果是：featureA（检测图像）中每个点与被匹配对象 featureB（样本图像）中特征向量进行运算的匹配结果。
则，rawMatches 中共有 featureA 条记录，每一条有最优 K 个匹配结果。
每个结果中包含三个非常重要的数据分别是 queryIdx，trainIdx，distance

- queryIdx：特征向量的特征点描述符的下标（第几个特征点描述符），同时也是描述符对应特征点的下标
- trainIdx：样本特征向量的特征点描述符下标,同时也是描述符对应特征点的下标
- distance：代表匹配的特征点描述符的欧式距离，数值越小也就说明俩个特征点越相近

```python
def matchKeypoints(self, kpsA, kpsB, featureA, featureB, ratio, reprojThresh):
        # 建立暴力匹配器
        matcher = cv2.BFMatcher()

        # 使用KNN检测来自AB图的SIFT特征匹配
        rawMatches = matcher.knnMatch(featureA, featureB, 2)

        # 过滤
        matches = []
        for m in rawMatches:
            if len(m) == 2 and m[0].distance < m[1].distance * ratio:
                matches.append((m[0].trainIdx, m[0].queryIdx))

        if len(matches) > 4:
            # 获取匹配对的点坐标
            ptsA = np.float32([kpsA[i] for (_, i) in matches])
            ptsB = np.float32([kpsB[i] for (i, _) in matches])
```

3. 通过点对计算 H 矩阵

- **cv2.findHomography(srcPoints,dstPoints,method=None,ransacReprojThreshold=None, mask=None, maxIters=None, confidence=None) --> (H, status)**

计算多个二维点对之间的最优单映射变换矩阵 H（3*3），status 为可选的输出掩码，0/1 表示在变换映射中无效/有效
* method（0， RANSAC, LMEDS, RHO）
* ransacReprojThreshold 最大允许冲投影错误阈值（限方法 RANSAC 和 RHO）
* mask 可选输出掩码矩阵，通常由鲁棒算法（RANSAC 或 LMEDS）设置，是不需要设置的
* maxIters 为 RANSAC 算法的最大迭代次数，默认值为 2000
* confidence 可信度值，取值范围为 0 到 1.

```python
# 计算H矩阵
            H, status = cv2.findHomography(ptsA, ptsB, cv2.RANSAC, reprojThresh)
```

4. 利用单应矩阵对图片进行变换拼接

- **cv.warpPerspective(src, M, dsize, dst=None, flags=None, borderMode=None, borderValue=None) --> result**
  将图像按照变换映射 M 执行后返回变换后的图像 result。

```python
# 将图片A进行视角变换 中间结果
result = cv2.warpPerspective(imageA, H, (imageA.shape[1] + imageB.shape[1], imageA.shape[0]))
# 将图片B传入]
result[0:imageB.shape[0], 0:imageB.shape[1]] = imageB
self.cv_show('result', result)
```

5.实验结果

源图像:

目标图像: