# OCR深度实践系列：图像预处理

近一个半月时间没更了，在这段时间里针对OCR业务进行了深入研究，业务也已上线，谨以此篇作为OCR系列的开篇。

目前NLP+OCR的落地应用在市场上愈加火热，如金融领域的研报分析、司法领域的合同审核甚至知识图谱的信息抽取，无不显示着NLP与OCR融合的巨大魅力。

本文将针对OCR的前序-“预处理”从**理论**和**实战**两方面进行详细论述，当然，不会涉及过多的公式，网上对于公式解析已经很全面，若感兴趣可自行查找。

# 1.理论篇

光学字符识别（Optical Character Recognition，OCR）一般包括文本检测（主要用于定位文本的位置）和文本识别（主要用于识别文本的具体内容）两个步骤。而图像质量的好坏对于检测率与识别率的高低影响很大，不容忽视。下面将重点介绍图像预处理中的二值化、去噪和倾斜角检测校正的常用算法。

## 1.1 二值化方法

图像二值化，Image Binarization，即通过将像素点的灰度值设为0或255使得图像呈现明显的黑白效果。在传统方法甚至是现在的流行方法中，高质量的二值化图像仍然可以显著提升OCR效果，一方面减少了数据维度，另一方面排除噪声凸显有效区域。目前，二值化方法主要分为四种：

* 全局阈值方法
* 局部阈值方法
* 基于深度学习的方法
* 基于形态学和阈值的文档图像二值化方法

### 1.1.1 全局阈值方法

（1）固定阈值方法

该方法是对于输入图像中的所有像素点统一使用同一个固定阈值，类似于NLP中相似度计算的阈值选择方法。其基本思想就是个分段函数：

公司中的T就是选择的固定全局阈值。

在NLP领域的相似度计算中，不同领域的文本阈值不同，而在图像领域也是一样，固定阈值方法存在一个致命缺陷：很难为不同的输入图像确定最佳阈值。因此提出了接下来的计算方法。

（2）Ostu方法

Ostu方法又被称为最大类间方差法，是一种自适应的阈值确定方法。

对于图像I(x,y)，前景(即目标)和背景的分割阈值记作T，属于前景的像素点数占整幅图像的比例记为ω0，其平均灰度μ0；背景像素点数占整幅图像的比例为ω1，其平均灰度为μ1。图像的总平均灰度记为U，类间方差记为G。  
 假设图像的背景较暗，并且图像的大小为M×N，图像中像素的灰度值小于阈值T的像素个数记作N0，像素灰度大于阈值T的像素个数记作N1，则有：

将式(5)代入式(6)，得到等价公式：

采用遍历的方法得到使类间方差G最大的阈值T。

注：**opencv**可以直接调用这种算法，threshold(gray, dst, 0, 255, CV\_THRESH\_OTSU);

* **优点：**算法简单，当目标与背景的面积相差不大时，能够有效地对图像进行分割。
* **缺点：**当图像中的目标与背景的面积相差很大时，表现为直方图没有明显的双峰，或者两个峰的大小相差很大，分割效果不佳，或者目标与背景的灰度有较大的重叠时也不能准确的将目标与背景分开。
* **原因：**该方法忽略了图像的空间信息，同时将图像的灰度分布作为分割图像的依据，对噪声也相当敏感。

### 1.1.2 局部阈值方法

（1）自适应阈值算法

自适应阈值算法用到了积分图，是一个快速且有效地对网格的矩形子区域计算和的算法。积分图中任意一点(x,y)的值是从图左上角到该点形成的矩形区域内所有值之和。

（2）Niblack算法

Niblack算法同样是根据窗口内的像素值来计算局部阈值的，不同之处在于它不仅考虑到区域内像素点的均值和方差，还考虑到用一个事先设定的修正系数k来决定影响程度。

（3）Sauvola算法

Sauvola是针对文档二值化进行处理，在Niblack算法基础上进一步改进。在处理光线不均匀或染色图像时，比Niblack算法拥有更好的表现。

### 1.1.3 基于深度学习的方法

2017年提出了一种使用全卷积的二值化方法（Multi-Scale Fully Convolutional Neural Network），利用多尺度全卷积神经网络对文档图像进行二值化，可以从训练数据中学习并挖掘出像素点在空间上的联系，而不是依赖于在局部形状上人工设置的偏置。

### 1.1.4 基于形态学和阈值的文档图像二值化方法

该方法大体分为四步操作：

* 1）将RGB图像转化为灰度图；
* 2）图像滤波处理；
* 3）数学形态学运算；
* 4）阈值计算。

其中，数学形态学运算包括腐蚀、膨胀、开运算和闭运算。

## 1.2 图像去噪

在图像的采集、量化或传输过程中会导致图像噪声的出现，这对图像的后处理、分析会产生极大的影响。目前去噪方法分为4种：

* 空间滤波
* 小波阈值去噪
* 非局部方法
* 基于神经网络的方法

（1）空间滤波

空间滤波由一个邻域和对该邻域内像素执行的预定义操作组成，滤波器中心遍历输入图像的每个像素点之后就得到了处理后的图像。其中线性空间滤波器指的是在图像像素上执行的是线性操作，非线性空间滤波器的执行操作则与之相反。

线性空间滤波器包括平滑线性滤波、高斯滤波器。

非线性空间滤波器包括中值滤波、双边滤波。

（2）小波阈值去噪

基本思路包括3个步骤：1）二维图像的小波分解；2）对高频系数进行阈值量化；3）二维小波重构。

（3）非局部方法

该类型方法包括NL-means和BM3D。其中BM3D是当前效果最好的算法之一，具体可参考[this repo](https://github.com/ericmjonas/pybm3d)。

（4）基于神经网络的方法

目前已经逐渐流行使用这种方法进行降噪处理，从简单的MLP发展到LLNet。

## 1.3 倾斜角检测校正

在扫描过程中，很容易出现文档旋转和位移的情况，因此后续的OCR处理与倾斜角检测校正步骤密不可分。常见的方法有：霍夫变换、Randon变换以及基于PCA的方法。

针对霍夫变换的使用一般分为3个步骤：

1）用霍夫变换探测出图像中的所有直线；

2）计算出每条直线的倾斜角，求它们的平均值；

3）根据倾斜角旋转矫正图片。

# 2.实战篇

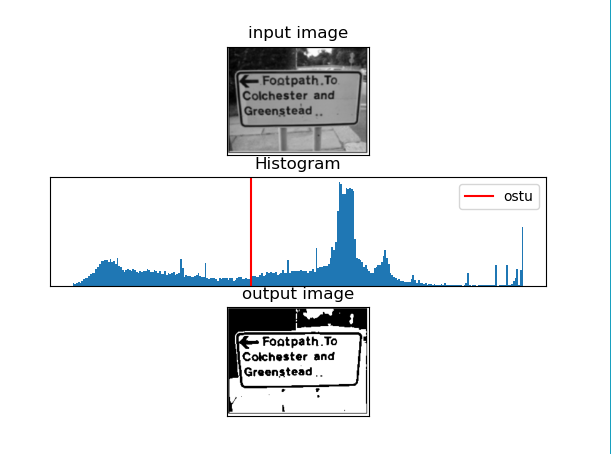
接下来本文将针对图像二值化、去噪、水平矫正三个模块进行实战演示。

## 2.1 图像二值化

以Ostu算法为例，展示实际效果，核心代码如下：

#1.将图像转换为灰度图  
gray\_img = cv2.cvtColor(img, cv2.COLOR\_BGR2GRAY)  
#2.对灰度图使用ostu算法  
ret1, th1 = cv2.threshold(gray\_img, 0, 255, cv2.THRESH\_OTSU)

输出效果如下：



## 2.2 图像去噪

以图像处理领域广为人知的**Lena**图片为例，展示高斯滤波、NL-means非局部均值、小波阈值以及BM3D四种方法的效果。

核心代码如下所示：

#1.读取噪声图像  
noisy\_img = np.float32(imread('../img/lena\_noise.bmp'))  
noisy\_img = cv2.cvtColor(noisy\_img, cv2.COLOR\_BGR2GRAY) / 255  
  
#高斯滤波  
img\_Guassian = cv2.GaussianBlur(noisy\_img, (5, 5), 0)  
  
# NL-means（非局部均值）  
def nlm(X, N, K, sigma):  
 H, W = X.shape  
 pad\_len = N + K  
 Xpad = np.pad(X, pad\_len, 'constant', constant\_values=0)  
 yy = np.zeros(X.shape)  
 B = np.zeros([H, W])  
 for ny in range(-N, N + 1):  
 for nx in range(-N, N + 1):  
 ssd = np.zeros((H, W))  
 # 根据邻域内像素间相似性确定权重  
 for ky in range(-K, K + 1):  
 for kx in range(-K, K + 1):  
 ssd += np.square(  
 Xpad[pad\_len + ny + ky:H + pad\_len + ny + ky, pad\_len + nx + kx:W + pad\_len + nx + kx] - Xpad[  
 pad\_len + ky:H + pad\_len + ky,  
 pad\_len + kx:W + pad\_len + kx])  
 ex = np.exp(-ssd / (2 \* sigma \*\* 2))  
 B += ex  
 yy += ex \* Xpad[pad\_len + ny:H + pad\_len + ny, pad\_len + nx:W + pad\_len + nx]  
 return yy / B  
img\_nlm = nlm(noisy\_img, 10, 4, 0.6)  
  
# 小波阈值  
def wavelet(X, levels, lmain):  
 def im2wv(img, nLev):  
 # pyr array  
 pyr = []  
 h\_mat = np.array([[1, 1, 1, 1],  
 [-1, 1, -1, 1],  
 [-1, -1, 1, 1],  
 [1, -1, -1, 1]])  
 for i in range(nLev):  
 n, mid = len(img), len(img) // 2  
 # split image up for HWT  
 a = img[:n:2, :n:2]  
 b = img[1:n:2, :n:2]  
 c = img[:n:2, 1:n:2]  
 d = img[1:n:2, 1:n:2]  
 vec = np.array([a, b, c, d])  
 # reshape vector to perform mat mult  
 D = 1 / 2 \* np.dot(h\_mat, vec.reshape(4, mid \* mid))  
 L, H1, H2, H3 = D.reshape([4, mid, mid])  
 pyr.append([H1, H2, H3])  
 img = L  
 pyr.append(L)  
 return pyr  
  
 def wv2im(pyr):  
 h\_mat = np.array([[1, 1, 1, 1],  
 [-1, 1, -1, 1],  
 [-1, -1, 1, 1],  
 [1, -1, -1, 1]])  
 h\_mat\_inv = np.linalg.inv(h\_mat)  
  
 L = pyr[-1]  
 for [H1, H2, H3] in reversed(pyr[:-1]):  
 n, n2 = len(L), len(L) \* 2  
 vec = np.array([L, H1, H2, H3])  
  
 D = 2 \* np.dot(h\_mat\_inv, vec.reshape(4, n \* n))  
 a, b, c, d = D.reshape([4, n, n])  
  
 img = np.empty((n2, n2))  
 img[:n2:2, :n2:2] = a  
 img[1:n2:2, :n2:2] = b  
 img[:n2:2, 1:n2:2] = c  
 img[1:n2:2, 1:n2:2] = d  
 L = img  
 return L  
  
 def denoise\_coeff(y, lmbda):  
 x = np.copy(y)  
 x[np.where(y > lmbda / 2.0)] -= lmbda / 2.0  
 x[np.where(y < -lmbda / 2.0)] += lmbda / 2.0  
 x[np.where(np.logical\_and(y > -lmbda / 2.0, y < lmbda / 2.0))] = 0  
 return x  
  
 pyr = im2wv(X, levels)  
 for i in range(len(pyr) - 1):  
 for j in range(2):  
 pyr[i][j] = denoise\_coeff(pyr[i][j], lmain / (2 \*\* i))  
 pyr[i][2] = denoise\_coeff(pyr[i][2], np.sqrt(2) \* lmain / (2 \*\* i))  
 im = wv2im(pyr)  
 return im  
# BM3D算法  
def run\_bm3d(noisy\_im, sigma,  
 n\_H, k\_H, N\_H, p\_H, tauMatch\_H, useSD\_H, tau\_2D\_H, lambda3D\_H,  
 n\_W, k\_W, N\_W, p\_W, tauMatch\_W, useSD\_W, tau\_2D\_W):  
 k\_H = 8 if (tau\_2D\_H == 'BIOR' or sigma < 40.) else 12  
 k\_W = 8 if (tau\_2D\_W == 'BIOR' or sigma < 40.) else 12  
  
 noisy\_im\_p = symetrize(noisy\_im, n\_H)  
 img\_basic = bm3d\_1st\_step(sigma, noisy\_im\_p, n\_H, k\_H, N\_H, p\_H, lambda3D\_H, tauMatch\_H, useSD\_H, tau\_2D\_H)  
 img\_basic = img\_basic[n\_H: -n\_H, n\_H: -n\_H]  
  
 assert not np.any(np.isnan(img\_basic))  
 img\_basic\_p = symetrize(img\_basic, n\_W)  
 noisy\_im\_p = symetrize(noisy\_im, n\_W)  
 img\_denoised = bm3d\_2nd\_step(sigma, noisy\_im\_p, img\_basic\_p, n\_W, k\_W, N\_W, p\_W, tauMatch\_W, useSD\_W, tau\_2D\_W)  
 img\_denoised = img\_denoised[n\_W: -n\_W, n\_W: -n\_W]  
  
 return img\_basic, img\_denoised

输出效果如下：









从以上效果中可以看出，BM3D算法去噪效果最好。

## 2.3 水平矫正

演示以霍夫变换为例的算法，核心代码：

# 度数转换  
def DegreeTrans(theta):  
 res = theta / np.pi \* 180  
 return res  
  
# 逆时针旋转图像degree角度（原尺寸）  
def rotateImage(src, degree):  
 # 旋转中心为图像中心  
 h, w = src.shape[:2]  
 # 计算二维旋转的仿射变换矩阵  
 RotateMatrix = cv2.getRotationMatrix2D((w / 2.0, h / 2.0), degree, 1)  
 print(RotateMatrix)  
 # 仿射变换，背景色填充为白色  
 rotate = cv2.warpAffine(src, RotateMatrix, (w, h), borderValue=(255, 255, 255))  
 return rotate  
  
# 通过霍夫变换计算角度  
def CalcDegree(srcImage):  
 midImage = cv2.cvtColor(srcImage, cv2.COLOR\_BGR2GRAY)  
 dstImage = cv2.Canny(midImage, 50, 200, 3)  
 lineimage = srcImage.copy()  
 # 通过霍夫变换检测直线  
 # 第4个参数就是阈值，阈值越大，检测精度越高  
 lines = cv2.HoughLines(dstImage, 1, np.pi / 180, 200)  
 # 由于图像不同，阈值不好设定，因为阈值设定过高导致无法检测直线，阈值过低直线太多，速度很慢  
 sum = 0  
 # 依次画出每条线段  
 for i in range(len(lines)):  
 for rho, theta in lines[i]:  
 # print("theta:", theta, " rho:", rho)  
 a = np.cos(theta)  
 b = np.sin(theta)  
 x0 = a \* rho  
 y0 = b \* rho  
 x1 = int(round(x0 + 1000 \* (-b)))  
 y1 = int(round(y0 + 1000 \* a))  
 x2 = int(round(x0 - 1000 \* (-b)))  
 y2 = int(round(y0 - 1000 \* a))  
 # 只选角度最小的作为旋转角度  
 sum += theta  
 cv2.line(lineimage, (x1, y1), (x2, y2), (0, 0, 255), 1, cv2.LINE\_AA)  
 cv2.imshow("Imagelines", lineimage)  
  
 # 对所有角度求平均，这样做旋转效果会更好  
 average = sum / len(lines)  
 angle = DegreeTrans(average) - 90  
 return angle

输出效果如下：

