4.4 图像处理算法

4.4.1 文字方向检测算法

实现了不同图片文字方向的算法，当文字方向在0，90，180，270°时识别效果较好

*predict.py l38注释错误*

函数的具体实现如下：

def predict(path=None,img=None):

"""

图片文字方向预测

"""

ROTATE = [0,90,180,270]

if path is not None:

im = Image.open(path).convert('RGB')

elif img is not None:

im = Image.fromarray(img).convert('RGB')

w,h = im.size

xmin,ymin,xmax,ymax = int(0.1\*w),int(0.1\*h),w-int(0.1\*w),h-int(0.1\*h)

im = im.crop((xmin,ymin,xmax,ymax))##剪切图片边缘，清除边缘噪声

im = im.resize((224,224))

img = np.array(im)

img = preprocess\_input(img.astype(np.float32))

pred = model.predict(np.array([img]))

index = np.argmax(pred,axis=1)[0]

return ROTATE[index]

对算法解释如下：

对于输入的图片，

1. 首先剪切W[int(0.1 \* w): w - int(0.1 \* w)], H[int(0.1 \* h), h - int(0.1 \* h)]的范围，然后将图片大小调整为(224, 224)，通过剪切边界的方式清除图片的边缘噪声，并产生VGG16模型的输入
2. 去除边缘噪声后的图片再利用VGG16提取特征，并返回一个Keras的模型对象功供训练好的模型进行识别偏转角度

*VGG16模型：VGG16模型的权重是由ImageNet训练而来的，默认输入尺寸是224×224*

得到文字偏转角度后，将图片根据对应角度旋转回正向即可：

算法分析：

1. 识别的准确度很大程度上由训练模型的训练结果有关
2. 为了保证训练速度，只取了具有代表性的0,90,180,270四个角度作为训练结果；若时间和条件允许，可以加入更多角度。

4.4.2 text\_detect

定义了一个函数用于初步检测文字边界，但经过该算法只能进行初步的识别，还需对检测的结果进行进一步调整以得到更精确的结果。

from ctpn.model import ctpn

from ctpn.detectors import TextDetector

from ctpn.other import draw\_boxes

import numpy as np

def text\_detect(img):

scores, boxes,img = ctpn(img)

textdetector = TextDetector()

boxes = textdetector.detect(boxes,scores[:, np.newaxis],img.shape[:2])

text\_recs,tmp = draw\_boxes(img, boxes, caption='im\_name', wait=True,is\_display=False)

return text\_recs,tmp,img

对算法解释如下：

把训练好的模型放在ctpn/model目录下后，即可使用该函数。

首先建立了一个TextDetector的对象，TextDetector类的定义如下：

class TextDetector:

"""

Detect text from an image

"""

def \_\_init\_\_(self):

"""

pass

"""

self.text\_proposal\_connector=TextProposalConnector()

def detect(self, text\_proposals,scores,size):

"""

Detecting texts from an image

:return: the bounding boxes of the detected texts

"""

#text\_proposals, scores=self.text\_proposal\_detector.detect(im, cfg.MEAN)

keep\_inds=np.where(scores>cfg.TEXT\_PROPOSALS\_MIN\_SCORE)[0]

text\_proposals, scores=text\_proposals[keep\_inds], scores[keep\_inds]

sorted\_indices=np.argsort(scores.ravel())[::-1]

text\_proposals, scores=text\_proposals[sorted\_indices], scores[sorted\_indices]

# nms for text proposals

keep\_inds=nms(np.hstack((text\_proposals, scores)), cfg.TEXT\_PROPOSALS\_NMS\_THRESH)

text\_proposals, scores=text\_proposals[keep\_inds], scores[keep\_inds]

scores=normalize(scores)

text\_lines=self.text\_proposal\_connector.get\_text\_lines(text\_proposals, scores, size)

keep\_inds=self.filter\_boxes(text\_lines)

text\_lines=text\_lines[keep\_inds]

if text\_lines.shape[0]!=0:

keep\_inds=nms(text\_lines, cfg.TEXT\_LINE\_NMS\_THRESH)

text\_lines=text\_lines[keep\_inds]

return text\_lines

def filter\_boxes(self, boxes):

heights=boxes[:, 3]-boxes[:, 1]+1

widths=boxes[:, 2]-boxes[:, 0]+1

scores=boxes[:, -1]

return np.where((widths/heights>cfg.MIN\_RATIO) & (scores>cfg.LINE\_MIN\_SCORE) &

(widths>(cfg.TEXT\_PROPOSALS\_WIDTH\*cfg.MIN\_NUM\_PROPOSALS)))[0]

建立对象后，将传入图片的尺寸，初步分割的字体图片等信息传递给TextDetector.detect函数，TextDetector.detect函数的定义已在TextDetector类的代码中给出，detect函数的重点是通过一个fast\_rcnn的模型来进行文字识别，然后返回初步识别出的文字图片的边界，可以看出，文字检测的重中之重还是模型的训练。

算法分析：

我们选取的训练集为英文字母，数字和常见的汉字，经过足够量的训练后，可以产生较理想的结果，若扩大训练集和训练量，可以得到更好的结果。

4.4.3 调整文字识别结果

该算法用于对文字识别的效果进行初步调整，核心算法的实现很简单

for index,rec in enumerate(text\_recs):

results[index] = [rec,]

xlength = int((rec[6] - rec[0])\*0.1)

ylength = int((rec[7] - rec[1])\*0.2)

if adjust:

pt1 = (max(1,rec[0]-xlength),max(1,rec[1]-ylength))

pt2 = (rec[2],rec[3])

pt3 = (min(rec[6]+xlength,xDim-2),min(yDim-2,rec[7]+ylength))

pt4 = (rec[4],rec[5])

else:

pt1 = (max(1,rec[0]),max(1,rec[1]))

pt2 = (rec[2],rec[3])

pt3 = (min(rec[6],xDim-2),min(yDim-2,rec[7]))

pt4 = (rec[4],rec[5])

算法分析：

对初步检测出的文字图片进行数学上的优化，最终检测图像倾斜角度的算式为

degree = degrees(atan2(pt2[1]-pt1[1],pt2[0]-pt1[0]))##图像倾斜角度

因此，若rec[0] - xlength < 1,rc[1] - ylength，实际计算角度的算式为actan2(rec[3] - 1, rec[2] - 1)，即对text\_detect初步检测的结果进行了初步的调整。

因为该步优化只是数学上的优化，因此很难再有大的优化。

4.4.4 相似图片检验

检验相似图片的关键算法是**感知哈希算法**，感知哈希算法简单地说就是对每张图片生成一个指纹字符串，然后比较不同图片的指纹，结果越接近说明图片越相似。

对算法的实现做一个简单的说明：

1. 缩小图片尺寸

将输入的图片缩小为8\*8共64个像素点，缩小的目的是为了去除大图中的细节，只保留一些图片结构，图像明暗度之类的基本信息，避免图像尺寸，图像比例等带来的图片差异。

b. 简化色彩

将8\*8的图片转为64灰度级，即所有的像素点只用最多64种颜色表示。

c. 计算平均值

计算所有像素灰度的平均值，计算的目的是为了之后的二值化。

d. 图像二值化

将每个像素的灰度值与平均值进行比较，大于或等于平均值的记为1；小于平均值的记为0。

e. 计算哈希值

将上一步的计算结果组合在一起构成一个64位整数就是图片的指纹，可以看出，组合的次序并不重要，只要在比较时让所有的图片都采用相同次序即可。

得到图片指纹后，即可比较两张图片的哈希值，通过设置阈值判断两张图片的相似度。

4.4.5