钱币定位系统 刘帅 2020212267

一、任务介绍：

通过opencv和pytorch框架编写钱币定位系统，通过canny算子实现对钱币图像的轮廓的检测，同时通过hough变换给出各个钱币的圆心坐标与半径大小。

文件组织形式：

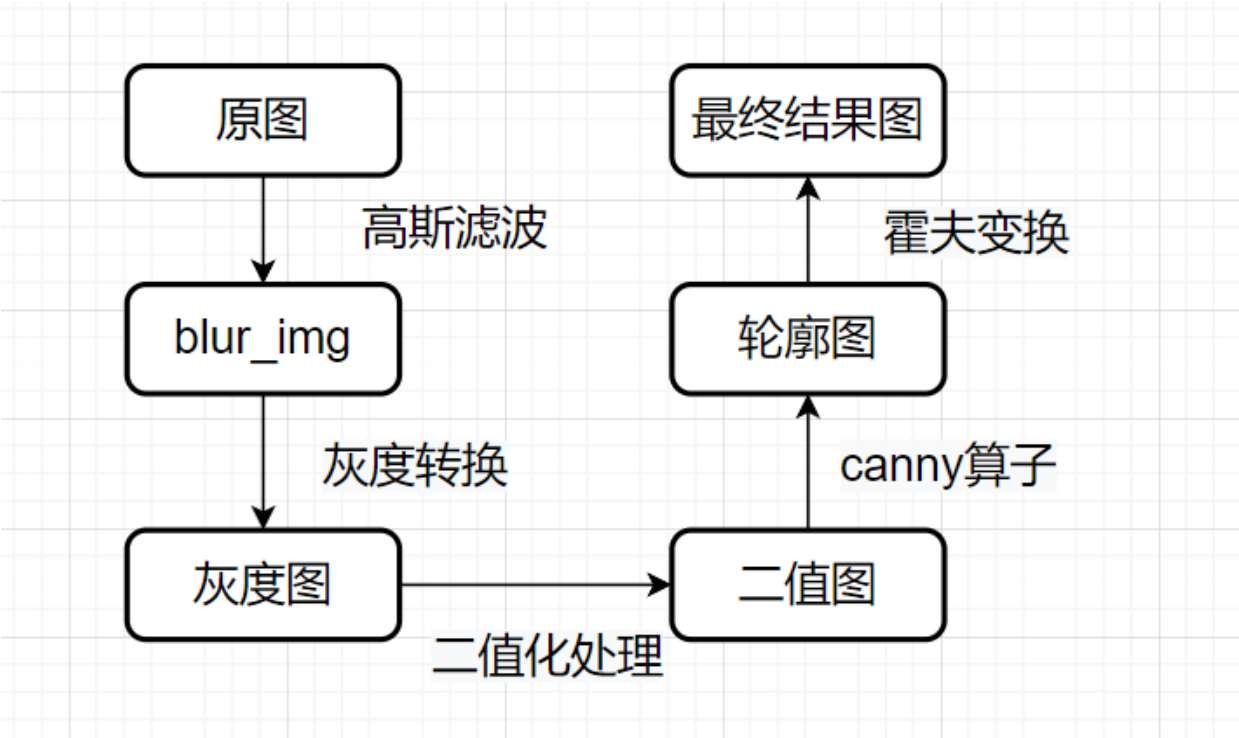
```
|liushuai_2020212267
|-- canny_fromscratch.py #自主实现的canny算法和hough变换
|-- canny_opencv.py#利用opencv实现的轮廓提取
|-- circle_fromscratch.jpg #自主实现的hough圆拟合结果
|-- circle_opencv.jpg #opencv的hough圆拟合结果
|-- coin.jpg#原始coin图片
|-- edge_fromscratch.jpg #自主实现的边缘检测结果
|-- edge_opencv.jpg ##opencv的边缘检测结果
|-- gaussblur.jpg #自主实现和opencv中的gauss模糊比较结果
`-- sobel_operator.jpg #sobel滤波的可视化中间结果
```

二、代码实现

2.1 利用opencv进行图像轮廓提取与硬币边缘拟合

利用opencv进行图像轮廓提取与硬币边缘拟合的流程如下：

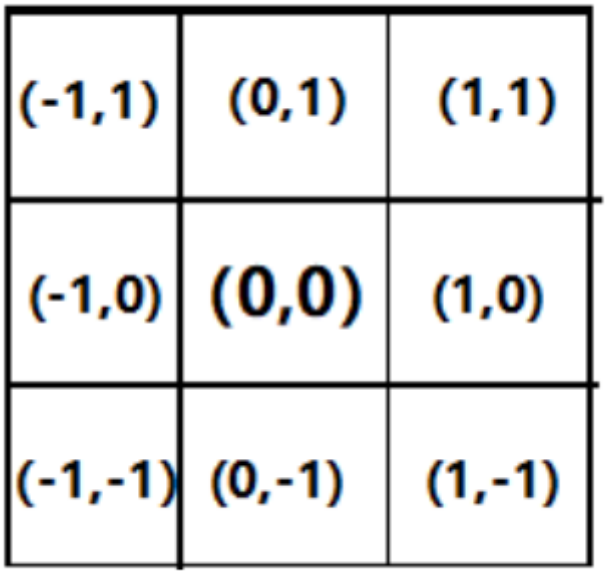
首先将原图进行高斯模糊，得到平滑处理的RGB图像，而后，便于该图像作为canny算子的输入，将其二值化处理，在经过canny算法后得到硬币的轮廓图，最终，再根据轮廓图进行houghcircle的拟合。



为进一步了解高斯模糊的数学原理，作者借助numpy进行了高斯核为3x3的gaussianblur方法的实现。

```
def GaussianBlur(sigma,img):
    sigma = float(sigma)
    num1 = np.around( (2 * np.pi * sigma ** 2) ** (-1),decimals=7)
    num2 = np.around(  (2 * np.pi * sigma ** 2) ** (-1) * np.exp((np.negative(sigma
** 2)) ** (-1) * 0.5),decimals=7)
    num3 = np.around( (2 * np.pi * sigma ** 2) ** (-1) * np.exp((np.negative(sigma
** 2)) ** (-1)),decimals=7)
    GaussMatrix = np.array([[num3,num2,num3],
                            [num2,num1,num2],
                            [num3,num2,num3]])
    total = np.around( ( (num2+num3)*4 + num1),decimals=7) #便于后续归一化处理
    img = cv2.copyMakeBorder(img,1,1,1,1,borderType=cv2.BORDER_REPLICATE)
    (b,g,r) = cv2.split(img)
    b1 = np.zeros(b.shape,dtype="uint8")
    g1 = np.zeros(g.shape,dtype="uint8")
    r1 = np.zeros(r.shape,dtype="uint8")
    temp = list(range(3))
    for i in range(1,b.shape[0]-1):
        for j in range(1,b.shape[1]-1):
            temp[0] = int( (np.dot(np.array([1, 1, 1]), GaussMatrix * b[i - 1:i + 2,
j - 1:j + 2]/total)).dot(np.array([[1], [1], [1]])))
            temp[1] = int((np.dot(np.array([1, 1, 1]),  GaussMatrix * g[i - 1:i + 2,
j - 1:j + 2]/total)).dot(np.array([[1], [1], [1]])))
            temp[2] = int((np.dot(np.array([1, 1, 1]),  GaussMatrix * r[i - 1:i + 2,
j - 1:j + 2]/total)).dot(np.array([[1], [1], [1]])))

            b1[i, j] = temp[0]
            g1[i, j] = temp[1]
            r1[i, j] = temp[2]
    b1=b1[1:-1,1:-1]
    g1=g1[1:-1,1:-1]
    r1=r1[1:-1,1:-1] #把图像还原为原始size
    image = cv2.merge([b1,g1,r1])
    return image
```

为构建3x3的高斯核，我们利用3x3的矩阵的坐标作为index进行构建



服从高斯分布$(1/2\pi\sigma^2)e^{-(x^2+y^2)/2\sigma^2}$，并对高斯核进行归一化处理，同时填充图像的周围一圈像素（利用边缘像素复制），从而便于高斯核对原图边缘像素进行处理，最后经过高斯滤波后再将图像还原为原尺寸大小。

利用opencv进行完整边缘提取的代码如下：

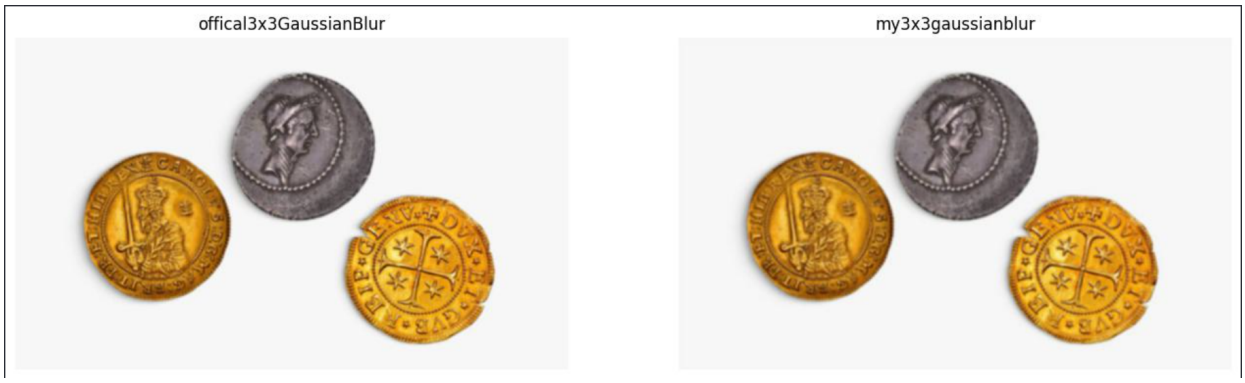
```
dir="/mnt/ve_share/liushuai/Document-Boundary-
Detection/liushuai_2020212267/coin.jpg"
image = cv2.imread(dir)
image=cv2.GaussianBlur(image, (5, 5), 0)#高斯模糊处理
gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY) #转化为灰度图
_, binary = cv2.threshold(gray, 200, 255, cv2.THRESH_BINARY) #二值化处理，将大于200的像
素设为255，反之设为0
edged = cv2.Canny(gray, 75, 200) #利用canny算子进行边缘提取，设置低阈值为75，高阈值为200
cv2.imwrite("/mnt/ve_share/liushuai/Document-Boundary-
Detection/liushuai_2020212267/edge_opencv.jpg",edged)
circles = cv2.HoughCircles(edged,cv2.HOUGH_GRADIENT
,0.1,120,param1=10,param2=30,minRadius=20,maxRadius=100)
if circles is not None:
    circles = np.uint16(np.around(circles))
    i=1
    for x,y,r in circles[0]:
        cv2.circle(image,(x,y),r,(255,0,0),3)
        print(f"第{i}个圆的中心坐标为({x},{y}),半径为:{r}")
        i+=1
cv2.imwrite("/mnt/ve_share/liushuai/Document-Boundary-
Detection/liushuai_2020212267/circle_opencv.jpg",image)
```

可视化结果如下：

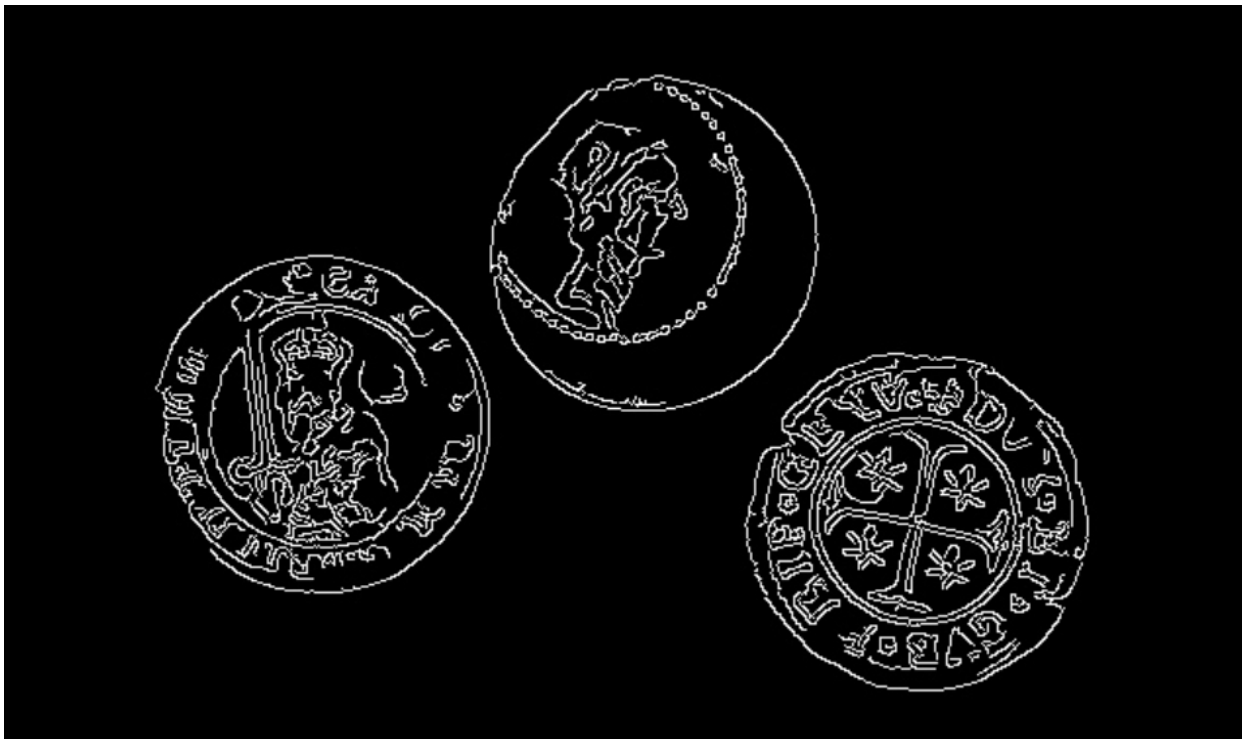
首先，针对自己实现的gauss滤波与opencv中自带的滤波效果比对如下：

利用pyplot将两种算法放到一张图中

```
image = cv2.imread(dir)
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(16,8))
ax1.imshow(cv2.cvtColor(cv2.GaussianBlur(image, (3,3), 0), cv2.COLOR_BGR2RGB))
ax2.imshow(cv2.cvtColor(GaussianBlur(100,image), cv2.COLOR_BGR2RGB))
ax1.set_title('offical3x3GaussianBlur')
ax2.set_title('my3x3gaussianblur')
ax1.axis('off')
ax2.axis('off')
plt.savefig("/mnt/ve_share/liushuai/Document-Boundary-
Detection/liushuai_2020212267/gaussblur.jpg", bbox_inches='tight')
```



利用opencv得到轮廓边缘的效果如下：



霍夫圆效果如下：

同时，python输出三个圆的中心坐标和半径：

```
(cv) root@LUCAS-DEV-b8d59a:/mnt/ve_share/liushuai/Document-Boundary-Detection#  cd /mnt
nv /usr/local/envs/cv/bin/python /root/.vscode-server/extensions/ms-python.python-2023
uncher 59917 -- /mnt/ve_share/liushuai/Document-Boundary-Detection/liushuai_2020212267/
第1个圆的中心坐标为(162,212),半径为:79
第2个圆的中心坐标为(474,262),半径为:77
第3个圆的中心坐标为(330,134),半径为:79
(cv) root@LUCAS-DEV-b8d59a:/mnt/ve_share/liushuai/Document-Boundary-Detection#
```

2.2 自主实现图像轮廓提取与硬币边缘拟合（基于pytorch）

在编写高斯滤波的过程中，我们发现滤波的过程实际上就是卷积运算的过程，同时，sobel算子的滤波等也是卷积操作，因此考虑可以直接利用pytorch对以上算子进行封装，增加代码的可移植性，并且可以利用GPU进行浮点数运算。

2.2.1 cannydetector初始化

cannydetector类继承nn.Module，在后续输入img信息时自动调用nn.Module父类的call函数，执行forward函数。设置gauss滤波、sobel滤波，8个梯度的滤波以及中心的模糊滤波，并利用initiate方法进行初始化。

```
class CannyDetector(nn.Module):
    def __init__(self, filter_size=5, std=1.0, device='cpu'):
        super(CannyDetector, self).__init__()
        self.device = device
        # gaussian模糊
        self.gaussian = nn.Conv2d(in_channels=1, out_channels=1, kernel_size=
(filter_size,filter_size), padding=(filter_size//2,filter_size//2), bias=False)
        # Sobel滤波
        self.sobel_filter_horizontal = nn.Conv2d(in_channels=1, out_channels=1,
kernel_size=3, padding=1, bias=False)
```

```
        self.sobel_filter_vertical = nn.Conv2d(in_channels=1, out_channels=1,
kernel_size=3, padding=1, bias=False)
        # 梯度方向滤波
        self.directional_filter = nn.Conv2d(in_channels=1, out_channels=8,
kernel_size=3, padding=1, bias=False)
        # 中心点模糊滤波
        self.connect_filter = nn.Conv2d(in_channels=1, out_channels=1,
kernel_size=3, padding=1, bias=False)
        # initiate
        params = initiate(filter_size=filter_size, std=std, map_func=lambda
x:torch.from_numpy(x).to(self.device))
        self.load_state_dict(params)
```

2.2.2 initiate初始化

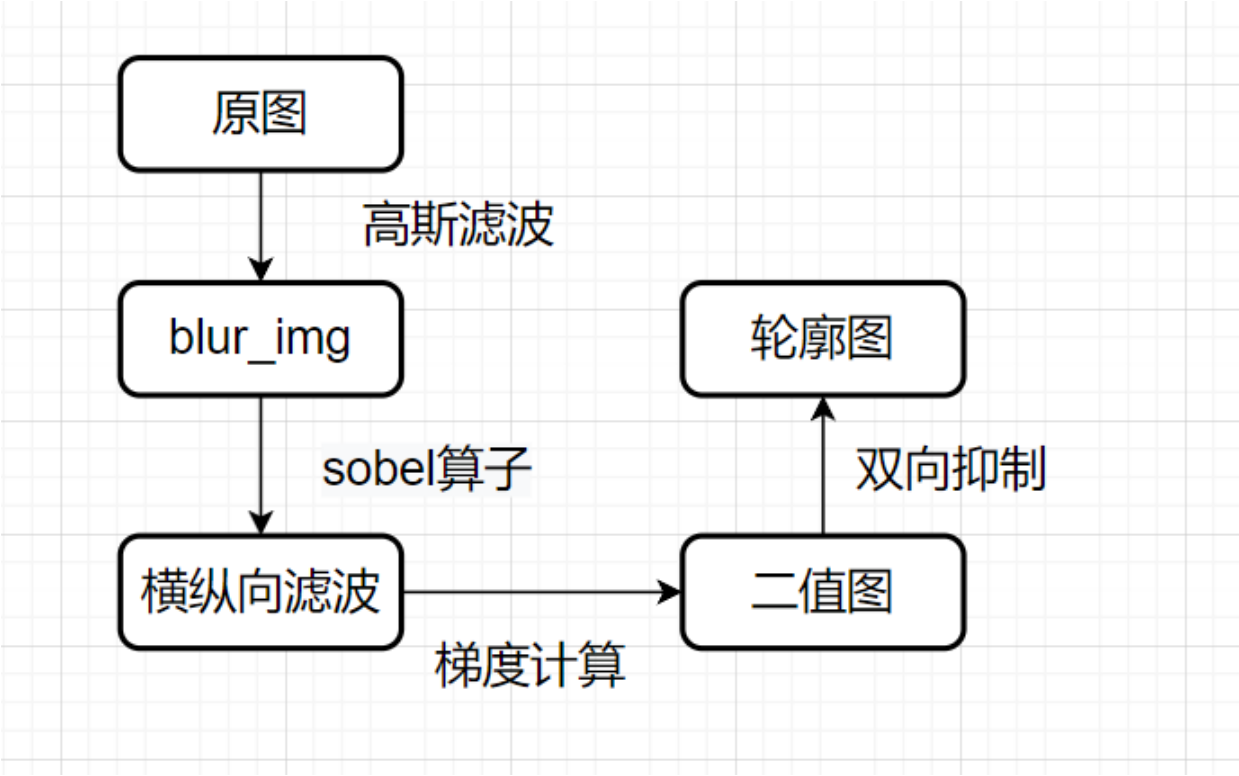
此处主要定义initiate函数，返回state_dict，从而将卷积核的参数存入模型中。

```
def initiate(filter_size=5, std=1.0, map_func=lambda x:x):
    kernel = torch.exp(-torch.arange(- filter_size// 2 + 1., filter_size// 2 + 1.)
** 2 / (2 * std ** 2))

 kernel2d=torch.ger(kernel,kernel).unsqueeze(0).unsqueeze(0).numpy().astype(np.float
32)
    kernel2d/=np.sum(kernel2d)
    sobel_filter_horizontal = np.array([[[
        [1., 0., -1.],
        [2., 0., -2.],
        [1., 0., -1.]]]],
        dtype='float32'
    )
    sobel_filter_vertical = np.array([[[
        [1., 2., 1.],
        [0., 0., 0.],
        [-1., -2., -1.]]]],
        dtype='float32'
    )
    directional_filter = np.array(
        [[[[ 0.,  0.,  0.],
          [ 0.,  1., -1.],
          [ 0.,  0.,  0.]]],
        [[[ 0.,  0.,  0.],
          [ 0.,  1.,  0.],
          [ 0.,  0., -1.]]],
        [[[ 0.,  0.,  0.],
          [ 0.,  1.,  0.],
          [ 0., -1.,  0.]]],
        [[[ 0.,  0.,  0.],
          [ 0.,  1.,  0.],
          [-1.,  0.,  0.]]],


        [[[ 0.,  0.,  0.],
```

```
          [-1.,  1.,  0.],
          [ 0.,  0.,  0.]]],


        [[[-1.,  0.,  0.],
          [ 0.,  1.,  0.],
          [ 0.,  0.,  0.]]],
        [[[ 0., -1.,  0.],
          [ 0.,  1.,  0.],
          [ 0.,  0.,  0.]]],
        [[[ 0.,  0., -1.],
          [ 0.,  1.,  0.],
          [ 0.,  0.,  0.]]]],
        dtype=np.float32
    )
    connect_filter = np.array([[[
        [1., 1., 1.],
        [1., 0., 1.],
        [1., 1., 1.]]]],
        dtype=np.float32
    )
    return {
        'gaussian.weight': map_func(kernel2d),
        'sobel_filter_horizontal.weight': map_func(sobel_filter_horizontal),
        'sobel_filter_vertical.weight': map_func(sobel_filter_vertical),
        'directional_filter.weight': map_func(directional_filter),
        'connect_filter.weight': map_func(connect_filter)
    }
```

2.2.3 forward函数

轮廓提取的流程示意如下



高斯滤波：

```
def forward(self, img, lowvalue=10.0, highvalue=100.0):
    # 拆分图像通道
    img_r = img[:,:,0] # red channel
    img_g = img[:,:,1] # green channel
    img_b = img[:,:,2] # blue channel
    tofloat=lambda x:torch.tensor(x).float().unsqueeze(0)

    # gaussianblured preprocess
    blurred_img_r = self.gaussian(tofloat(img_r))
    blurred_img_g = self.gaussian(tofloat(img_g))
    blurred_img_b = self.gaussian(tofloat(img_b))
```

首先将rgb通道拆分，针对每一个通道进行高斯滤波操作

sobel滤波：

在经过高斯滤波后，再利用sobel算子对横向和纵向进行滤波

```
grad_x_r = self.sobel_filter_horizontal(blurred_img_r)
grad_y_r = self.sobel_filter_vertical(blurred_img_r)
grad_x_g = self.sobel_filter_horizontal(blurred_img_g)
grad_y_g = self.sobel_filter_vertical(blurred_img_g)
grad_x_b = self.sobel_filter_horizontal(blurred_img_b)
grad_y_b = self.sobel_filter_vertical(blurred_img_b)
```

记录梯度，并求最值：

以45度为划分，将梯度方向近似为其中的一个，并且设置负方向（从而在下面index索引时能够在同梯度方向下找到最大值），**该算法筛选最大像素的核心在于channel_select_filtered.min(dim=0)[0] > 0.0，对于中心为a，梯度方向相同的三个像素点a,b,c，即如果a-b与a-c同时大于零，说明a是该梯度的邻域内最大的，记录此时的边缘。**

```
calgrad=lambda x,y:torch.sqrt(x**2+y**2) #the function of grad calculation
grad_mag = calgrad(grad_x_r,grad_y_r)
grad_mag += calgrad(grad_x_g,grad_x_g)
grad_mag += calgrad(grad_x_b,grad_y_b)
grad_orientation = (torch.atan2(grad_y_r+grad_y_g+grad_y_b,
grad_x_r+grad_x_g+grad_x_b) * (180.0/math.pi))
grad_orientation += 180.0
grad_orientation =  torch.round(grad_orientation / 45.0) * 45.0

all_filtered = self.directional_filter(grad_mag) #八个梯度方向的featuremap
inidices_positive = (grad_orientation / 45) % 8 #梯度方向
inidices_negative = ((grad_orientation / 45) + 4) % 8 #加上180度 进行反向索引
_, height, width = inidices_positive.shape
num_pixel = height * width
pixel_range = torch.Tensor([range(num_pixel)]).to(self.device)
indices = (inidices_positive.reshape((-1, )) * num_pixel + pixel_range).squeeze()
channel_select_filtered_positive = all_filtered.reshape((-1, ))
[indices.long()].reshape((1, height, width))
#实际在哪个梯度有效
indices = (inidices_negative.reshape((-1, )) * num_pixel + pixel_range).squeeze()
channel_select_filtered_negative = all_filtered.reshape((-1, ))
[indices.long()].reshape((1, height, width))
channel_select_filtered = torch.stack([channel_select_filtered_positive,
channel_select_filtered_negative])
is_max = channel_select_filtered.min(dim=0)[0] > 0.0 #找出两个向量中相同位置较小的那个值，并将其与0进行比较（如果为true说明中间比他邻域的相同方向都大）
final_edge = grad_mag.clone()
final_edge[is_max==0] = 0.0
```

双阈值筛选：

设置一大一小两个阈值，对大于阈值的部分进行模糊处理，对于在阈值间的像素，将其设为0

```
low = min(lowvalue, highvalue)
high = max(lowvalue, highvalue)
thresholded = final_edge.clone()
lower = final_edge<low
thresholded[lower] = 0.0 #小于较小阈值的设为0
higher = final_edge>high
thresholded[higher] = 1.0#大于较大阈值的设为1
connect_map = self.connect_filter(higher.float())#对大于阈值的部分进行模糊处理
middle = torch.logical_and(final_edge>=low, final_edge<=high)#居中的
thresholded[middle] = 0.0 #把居中的设为0
connect_map[torch.logical_not(middle)] = 0
thresholded[connect_map>0] = 1.0
thresholded[..., 0, :] = 0.0
```

```
thresholded[..., -1, :] = 0.0
thresholded[..., :, 0] = 0.0
thresholded[..., :, -1] = 0.0 #去除四个边缘
thresholded = (thresholded>0.0).float()
```

可视化结果：

1）sobel滤波的可视化结果

2）canny算法检测边缘的可视化结果

结果分析

可见，利用pytorch处理后的边缘比canny效果更好，原因可能如下：

1）高斯核大小和sigma：

较小的核大小和sigma分布更加集中，感受野更小，会导致边缘检测结果更加精细，但是对噪声更加敏感。较大的 ksize 和 sigma感受野更大，分布更加离散，则会使边缘检测结果更加平滑，但也会导致边缘的模糊和丢失细节。

2）双阈值的选择：

两个阈值的大小会影响边缘的检测结果。如果两个阈值之间的差异较小，则检测到的边缘将较少，但其强度较大。如果两个阈值之间的差异较大，则检测到的边缘数量将增加，但其强度可能较弱。

2.3 霍夫圆的实现

2.3.1 霍夫圆的初始化

在canny算子中，我们利用sobel算子实现了梯度提取，并得到了轮廓的图片，将其作为hough的输入

```
class Hough_transform:
    def __init__(self, img, grad_mag, mindistance, step, circle_threshold):
        self.img = img
        self.grad_mag = grad_mag.squeeze(0)
        self.height, self.width = img.shape[0:2]
        self.radius = math.ceil(math.sqrt(self.height**2 + self.width**2))   #圆直径
的最大长度
        self.mindistance = mindistance
        self.step = step
        self.vote_matrix = np.zeros([math.ceil(self.height / self.step),
math.ceil(self.width / self.step), math.ceil(self.radius / self.step)])
        self.circle_threshold = circle_threshold
        self.circles = []
```

2.3.2 霍夫变换

对图像进行循环遍历，如果像素值大于0，则将该像素作为圆心，并以步长为半径向外扩散。在扩散的过程中，对每个圆周上的像素进行投票，增加对应的计数器值。最终返回投票矩阵

```
def Hough_transform_algorithm(self):
    for height in range(1, self.height - 1):
        for width in range(1, self.width - 1):
            if self.img[height][width] > 0: #像素值大于0，则作为圆心，以步长为r扩散半径
                x = width
                y = height
                r = 0
                while x > 0 and y > 0 and x < self.width and y < self.height:
                    self.vote_matrix[math.floor(y / self.step)][math.floor(x /
self.step)][math.floor(r / self.step)] += 1
                    x = x + self.step
                    y = y + self.grad_mag[height][width] * self.step
                    r = r + math.sqrt(self.step ** 2 + (self.grad_mag[height][width]
* self.step) ** 2)
                x = width - self.step
                y = height - self.grad_mag[height][width] * self.step
                r = math.sqrt(self.step ** 2 + (self.grad_mag[height][width] *
self.step) ** 2)
    return self.vote_matrix
```

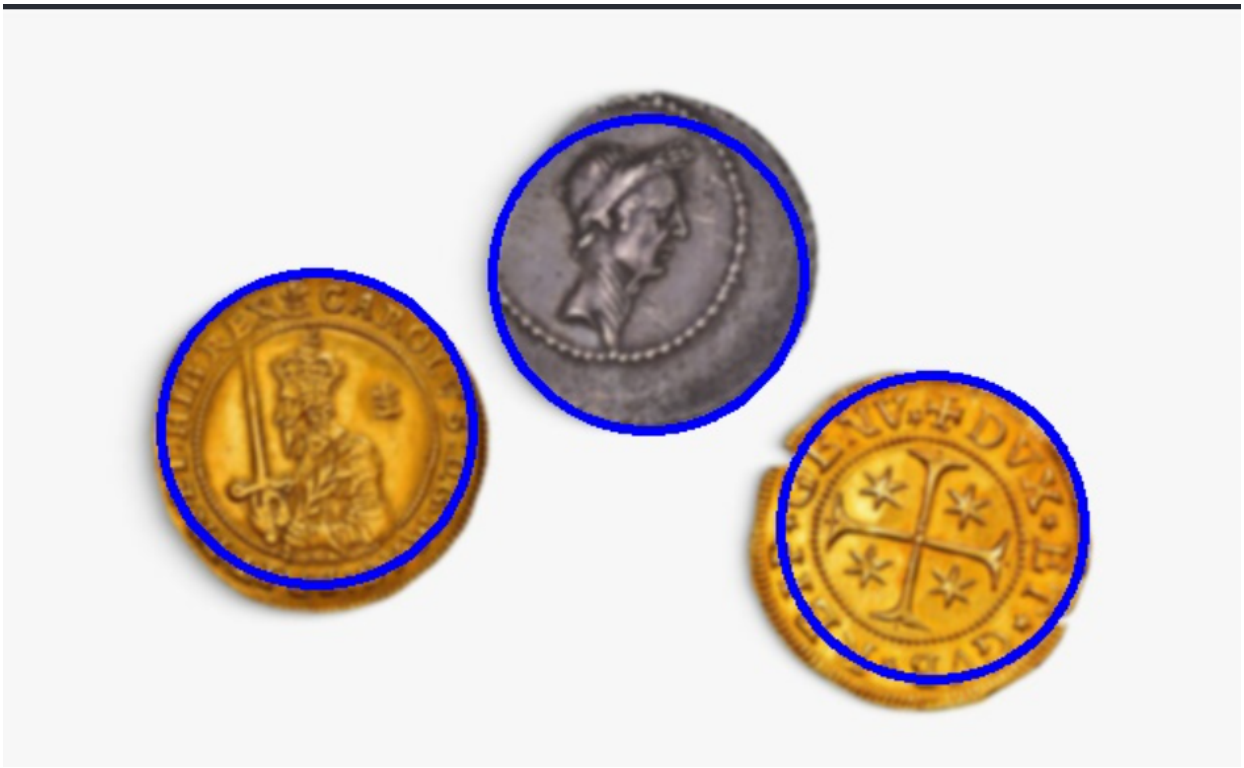
2.3.3 最大值抑制

遍历投票矩阵中的所有元素，如果新的圆和当前圆的距离小于预定义的距离阈值 self.mindistance，则将其添加到可能圆的列表中。如果距离大于阈值，则将所有可能圆的平均位置计算出来，并将其添加到最终圆的列表中。接下来，利用mindistance将候选圆分组，并计算每组圆的平均值。

```
    def Select_Circle(self):
        candidateCircles = []
        for i in range(0, self.vote_matrix.shape[0]):
            for j in range(0, self.vote_matrix.shape[1]):
                for k in range(0, self.vote_matrix.shape[2]):
                    if self.vote_matrix[i][j][k] > self.circle_threshold:
                        y = i * self.step + (self.step / 2)
                        x = j * self.step + (self.step / 2)
                        r = k * self.step + (self.step / 2)
                        candidateCircles.append([math.ceil(x), math.ceil(y),
math.ceil(r)])

        x, y, r = candidateCircles[0]
        possibleCircles = []
        middleCircles = []
        for circle in candidateCircles:
            if math.sqrt((x - circle[0])**2 + (y - circle[1])**2) <=
self.mindistance: #如果两个圆的距离小于阈值,则可能是同一个圆的候选点
                possibleCircles.append([circle[0], circle[1], circle[2]])
            else:
                result = np.array(possibleCircles).mean(axis=0)#如果大于，将所有圆的平均
位置计算出来
                middleCircles.append([result[0], result[1], result[2]])
                possibleCircles.clear()
                x, y, r = circle
                possibleCircles.append([x, y, r])
        result = np.array(possibleCircles).mean(axis=0)
        middleCircles.append([result[0], result[1], result[2]])

        middleCircles.sort(key=lambda x:x[0], reverse=False)
        x, y, r = middleCircles[0]
        possibleCircles = []
        for circle in middleCircles:
            if math.sqrt((x - circle[0])**2 + (y - circle[1])**2) <=
self.mindistance:
                possibleCircles.append([circle[0], circle[1], circle[2]])
            else:
                result = np.array(possibleCircles).mean(axis=0)
                print("Circle core: (%f, %f), Radius: %f" % (result[0], result[1],
result[2]))
                self.circles.append([result[0], result[1], result[2]])
                possibleCircles.clear()
                x, y, r = circle
                possibleCircles.append([x, y, r])
        result = np.array(possibleCircles).mean(axis=0)
        self.circles.append([result[0], result[1], result[2]])
```

2.3.4 可视化

```
def printcircle(self):
    for circle in self.circles:
        x,y,r=circle
        cv2.circle(img, (x.astype('uint8'), y.astype('uint8')), r.astype('uint8'),
(255, 0, 0), 2)
    cv2.imwrite('/mnt/ve_share/liushuai/Document-Boundary-
Detection/liushuai_2020212267/circle_fromscratch.jpg', img)
```



霍夫圆结果分析：

step：步长越小，能够检测到的圆的位置精度越高，但是计算量也会增加。

circle_threshold：圆心阈值是指在累加器中达到多少次才认为是一个圆心，阈值越大，检测到的圆更加可靠，但是可能会漏检一些较小的圆。

mindistance：如果两个圆心之间的距离小于该阈值，则认为这两个圆可能是同一个圆，会将它们合并为一个圆。阈值越小，能够检测到的圆的数量越多，但是可能会出现误检的情况。

同时，在调试过程中观察canny算子中的梯度分布，发现数值较为离散，最大值已经到达了8000，对后续步长的设定造成了不利的影响，可能的优化方案是对梯度分布进行归一化，从而便于后续的迭代处理。在时间复杂度方面，由于自己实现的霍夫圆复杂度为O(n^3)，因此选择合适的步长对圆的拟合尤为重要。