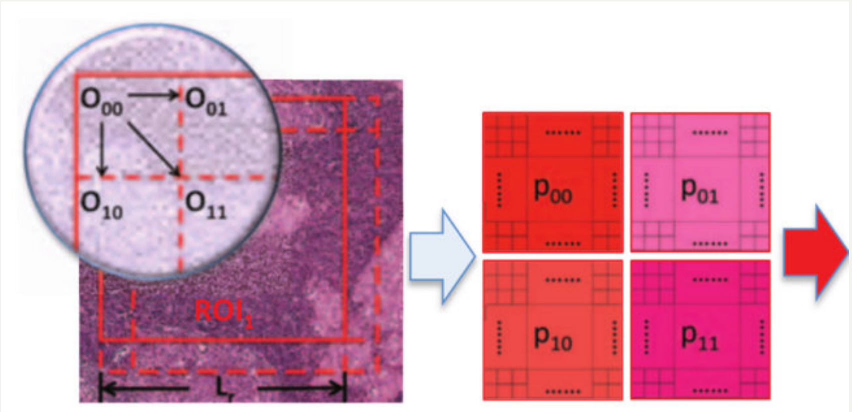
ScanNet流程的详细说明

给定大小为$H*W$ 的WSI图像，将其分割为多块区域$BLOCK[i,j]$，

$BLOCK$区域需满足条件$260+32*k$,原因如下:

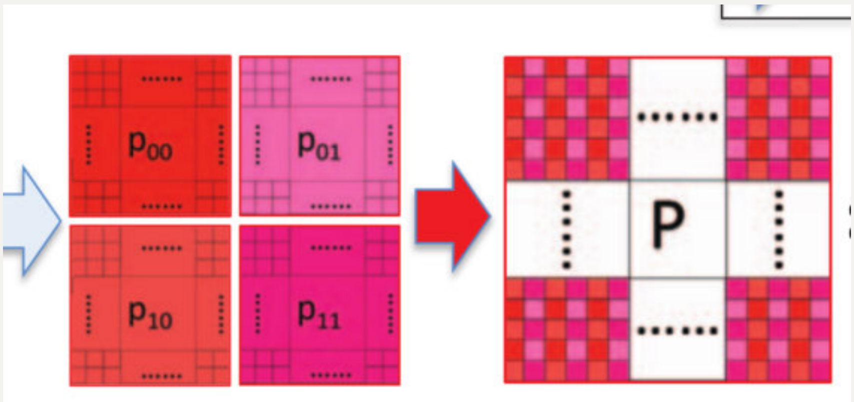
- 输入ScanNet图像大小为,$244\times244\times3$ ==> 输出概率的大小为 $2\times1\times1$.标记$L_f=244$
- ScanNet是一类FCN,可以通过滑动得到更大的概率矩阵,同时需要输入更大的图像. 由于ScanNet有5个pool层,则需要滑动$2^5=32$(记为S_f)个像素才能增加输出尺寸的大小,既ROI区域的尺寸(H,W)必须满足 $244+32\times k$的条件,此时生成的概率图OPT大小为$(k+1)\times(k+1)$
- 假设用于offset概率密度为$alpha=2$,则需要2*2个大小一样,偏移量为$32/alpha=16$(记为S_d)的$ROI[i,j],i,j\in[0,1]$,既可得到一个2×2个$(k+1)\times(k+1)$的概率矩阵OPTS.同时由于偏移量是16,$BLOCK$的尺寸(H,W),需满足$H,W=244+16+32\times k,k\in[0,1,2,...]$.



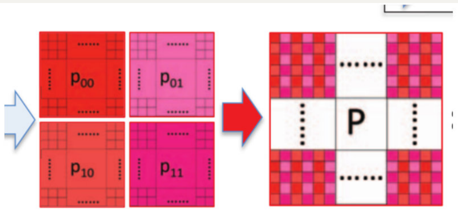

```python
def inner_scan(self,opts,roi_list):
    '''(测试通过)
    设定的roi是PIL.Image类
    Lr = Lf + (Lp -1) * Sf; Sr = Sf *Lp
    假设Lr = 2868, Sf=32, Lf=244, 则Lp=83(吻合, 此处ok),此时opt大小为LpXLpX2,经过softmax转换成LpXLpX1的p值
    :param roi: 单个ROI区域
    :return: opt矩阵
    '''

    roi_batch=torch.cat(roi_list,0)
      print('roi batch size',roi_batch.shape)
    sample_size=roi_batch.shape[0]
    Iteration = int(sample_size/self.mini_batch)
    opt_list=[]
    rows=0
    while(rows*self.mini_batch+self.mini_batch<sample_size):
        mini_batch=roi_batch[rows*self.mini_batch:(rows+1)*self.mini_batch]
        opt = self.model(mini_batch)
        opt =F.softmax(opt)[:,1].cpu().detach()
        opt_list.append(opt)
        rows+=1
    mini_batch=roi_batch[rows*self.mini_batch:sample_size]
    opt = self.model(mini_batch)
    opt =F.softmax(opt)[:,1].cpu().detach()
    opt_list.append(opt)
    opt_list=torch.cat(opt_list,0)
      opt = self.model(roi_batch)
      opt =F.softmax(opt)[:,1].cpu().detach()
    print('opt_list size',opt_list.shape)
    count=0
    for i in range(self.alpha):
        for j in range(self.alpha):
            opts[i,j,:,:]=opt_list[count]
      print('opts shape',opts.shape)
    return opts
```

- $ROI[i,j], i,j \in [0,1]$ 的对应的$OPTS[i,j], i,j \in [0,1]$交织在一起得到整个$BLOCK$区域的的概率矩阵M,大小为$(2k+2) \times (2k+2)$. 交织的计算方法过程如下:

$$\begin{cases} p(h', w') = p_{ij}(h, w) \\ i = h' \ mod \ \alpha, \quad j = w' \ mod \ \alpha \\ h = \lfloor h'/\alpha \rfloor, \quad w = \lfloor w'/\alpha \rfloor. \end{cases} \tag{3}$$




```python
def reconstruction(dpt,opts):
    '''
    After scan image, we can reconstuct DPT by interweaving.
    :return: dpt
    '''
    W, H = dpt.shape
    for h_ in range(H):
        for w_ in range(W):
            i = h_ % self.alpha
            j = w_ % self.alpha
            h = int(h_ / self.alpha)
            w = int(w_ / self.alpha)
            dpt[w_, h_] = opts[i, j][w, h]
    return dpt
```

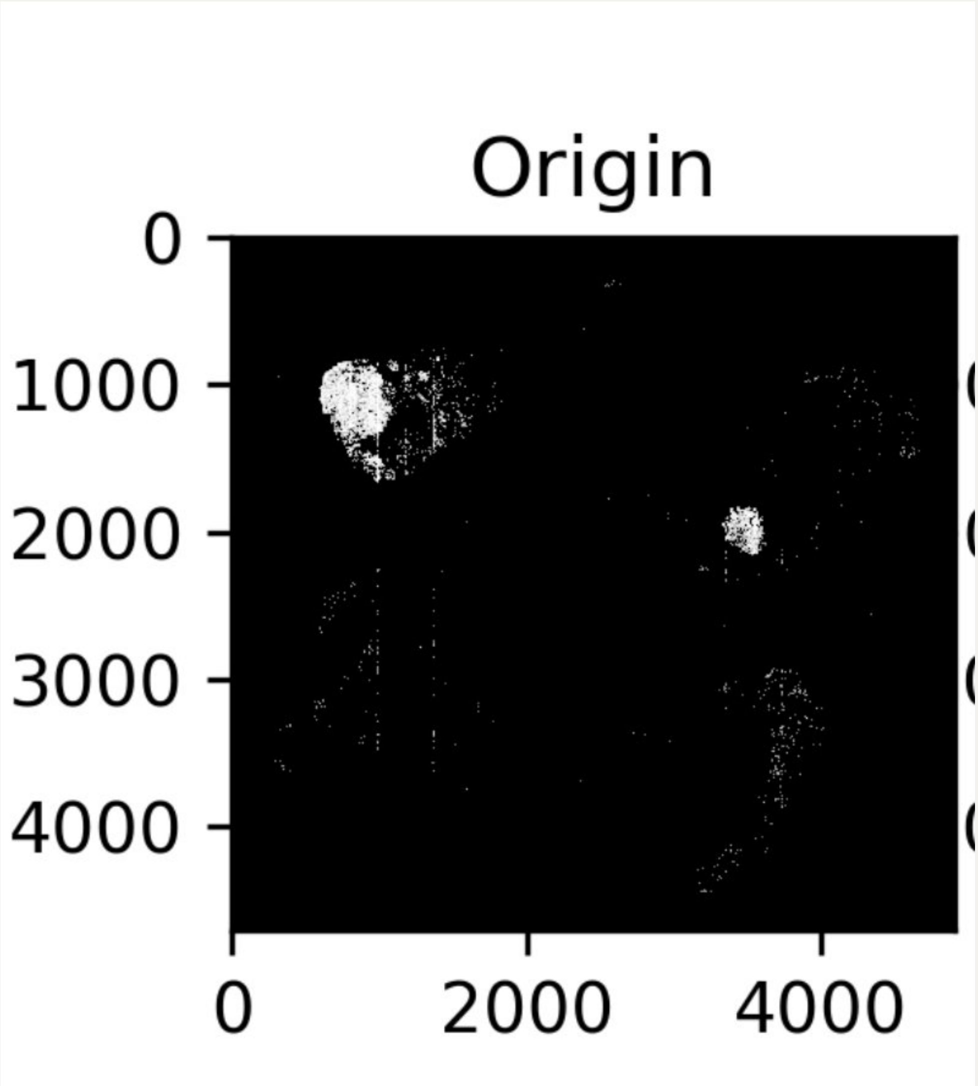


设$BLOCK[i,j], i,k \in A$的概率矩阵为M, 根据i,j的坐标即可拼接成整个WSI区域的概率图,大小为$(i \times M_w) \times (j \times M_h)$, 此时满足的关系式为:

$$H_I = H_P * S_d + L_f/2$$

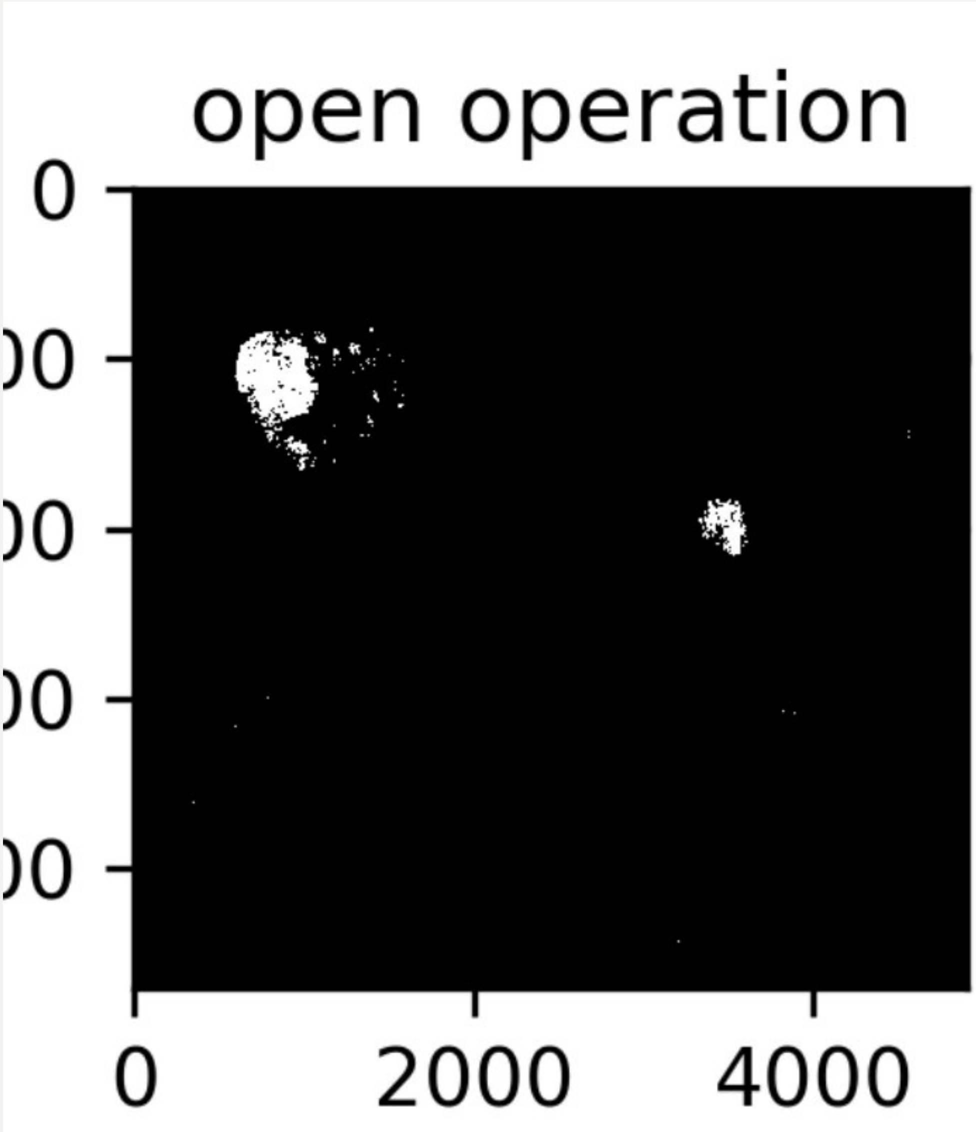
H_I:为WSI原坐标(X,Y), H_P:生成概率矩阵的坐标(x,y), S_d:偏移量值为16, L_f : 输入图像的patch尺寸值为244.

最终的概率图如下:

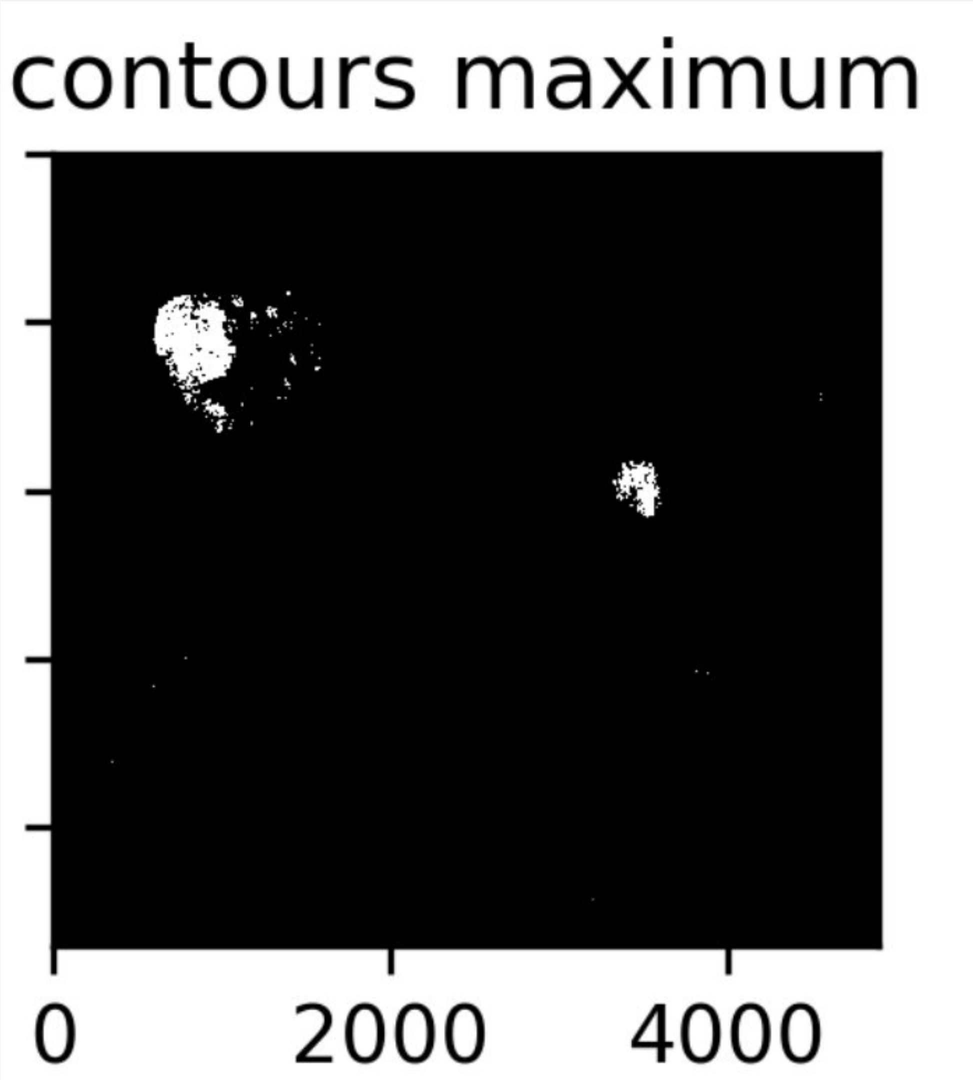


考虑到初始生成得概率图还有许多噪声点的存在:

文章对结果进行Open操作,效果如下: 设置不同的kernel可以去掉不同大小的离群点.



最后,概率图的每个连通区域的概率值由该连通域的最大概率表示,结果如下:



对应的代码如下:

```python
def fix(self,fpm,save_path=None,img=False,show=False):
    '''
    进行后处理的操作
    :param fpm:
    :return:
    '''

    logging.info('Handling %s'%save_path)
    st =time.time()
    bifpm = np.where(fpm>self.threshhold,1,0) # 二分类
    time1=time.time()
    logging.info('calculate bifpm time consuming: %.4f'%(time1-st))
    fpm = np.where(fpm>self.threshhold,fpm,0)
    opening = morphology.opening(bifpm, self.kernel)
    time2= time.time()
    logging.info('calculate opening time consuming: %.4f'%(time2-time1))
    labels,num_label = measure.label(opening, connectivity=self.connectivity,return_num=True)
    logging.info('num_label = %d'%num_label)
    fpm_filter = fpm *opening
    csvRows=[] #保存中心点和概率值
    for i in range(num_label):
          pdb.set_trace()
        st = time.time()
        max_p = np.max(fpm_filter[labels==i])
        ed = time.time()
        logging.info('each Iter, max_p st-ed: %.4f'%(ed-st))
        if max_p == 0: #无须再添加normal的值
            continue
        fpm_filter[labels==i] =  max_p#取最大概率
        ed2 = time.time()
        logging.info('each Iter, filter st-ed: %.4f'%(ed2-ed))
        indexes=np.argwhere(labels==i)
        x, y = np.sum(indexes,axis=0)/indexes.shape[0]
        x = int(x*self.Sd+self.Lf/2)   # 使用中间值map
        y = int(y*self.Sd+self.Lf/2)   # 使用中间值map
        csvRows.append([max_p, y, x])  # Transpose location
        ed3 =time.time()
        logging.info('each Iter, index st-ed: %.4f'%(ed3-ed2))
```