

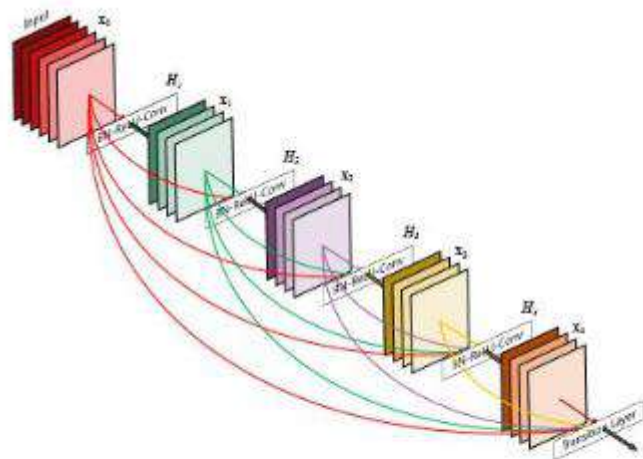
# LF-Net

从图像学习本地特征

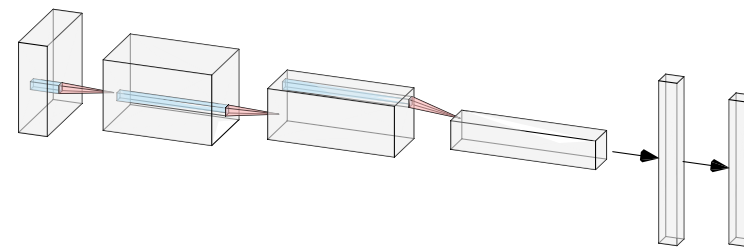
# 模型

detector

descriptor

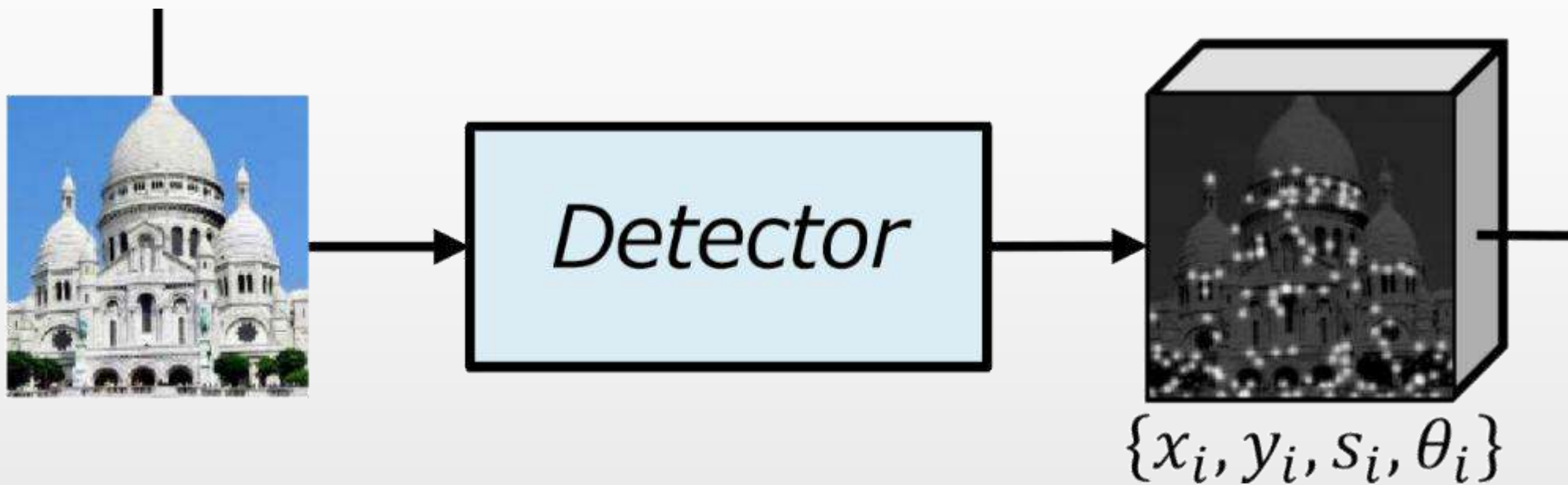


**Figure 1:** A 5-layer dense block with a growth rate of  $k = 4$ . Each layer takes all preceding feature-maps as input.



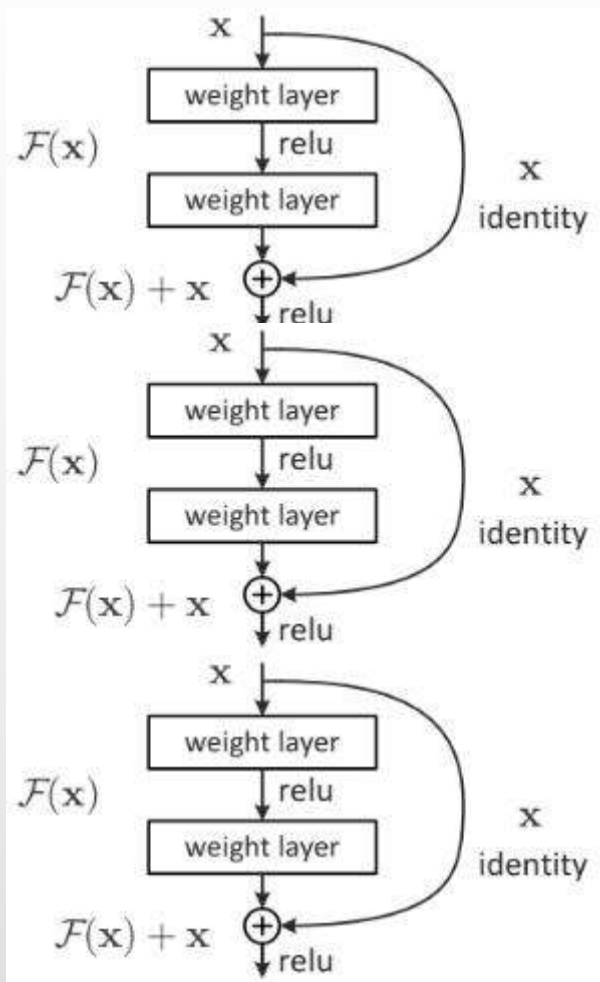
# 构建训练网络——Detector

任务目标



# 构建训练网络——Detector

## 1. Feature map generation



特点:

通过padding使得输入输出的尺寸一样, 这样就能直接得到想要的feature map (代码还叫heat map其实意思一样)。这也是这个网络能够这么高效原因, 层数很少。

据文中说这种方法比SuperPoint里最后一层上采样得到输入输出同尺寸效果好。

Basic Block:

5X5卷积->batch normalization->leaky-ReLu->5X5卷积

detector:

3 X Basic Block

# 构建训练网络——Detector

## 2. Scale-invariant keypoint detection

Input : Feature map -- O

Output : Scale-space score map -- S

### 2.1 Resize feature map N times

目的就是构建尺度金字塔



Feature map(O) ----- 16 channels

每一个



其实有16层



# 构建训练网络——Detector

## 2.1 Resize feature map N times

```
scale_log_factors = np.linspace(np.log(self.max_scale), np.log(self.min_scale), self.num_scales)
scale_factors = np.exp(scale_log_factors)
```

```
--->> array([1.41421356, 1.189207, 1.0, 0.8408964, 0.70710678])
```

$$\frac{1}{\sqrt{2}} \sim \sqrt{2}$$

初始的尺度

```
for i, s in enumerate(scale_factors):
```

```
    inv_s = 1.0 / s
```

```
    feat_height = tf.cast(base_height_f * inv_s + 0.5, tf.int32)
```

```
    feat_width = tf.cast(base_width_f * inv_s + 0.5, tf.int32)
```

```
    rs_feat_maps = tf.image.resize_images(curr_in, tf.stack([feat_height, feat_width]))
```

```
    score_maps = conv2d_fixed_padding(rs_feat_maps, 1,
```

```
        kernel_size=conv_ksize, scope='score_conv_{}'.format(i),
```

```
        use_xavier=use_xavier, use_bias=use_bias)
```

```
    score_maps_list.append(score_maps)
```

Resize N times

不改变原图大小的  
5x5卷积

# 构建训练网络——Detector

## 2.2 Get Score Maps

通过对 FeatureMap(O)卷积得到ScoreMaps

卷积通过padding不改变输入尺寸;

卷积核为5X5 的 **N个独立的**卷积核;

输入为16channel 输出为1channel;

OutPut: N个scoremaps  $h^n$

```
for i, s in enumerate(scale_factors):
```

```
    inv_s = 1.0 / s
```

```
    feat_height = tf.cast(base_height_f * inv_s + 0.5, tf.int32)
```

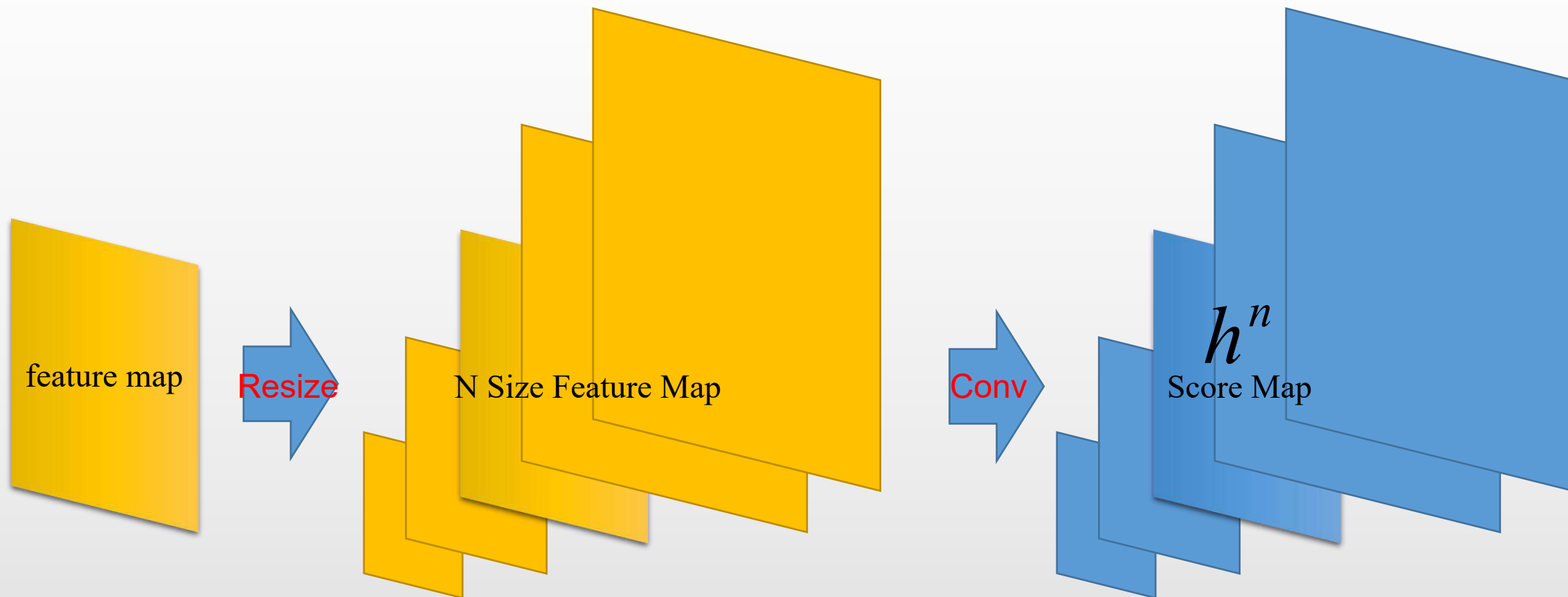
```
    feat_width = tf.cast(base_width_f * inv_s + 0.5, tf.int32)
```

```
    rs_feat_maps = tf.image.resize_images(curr_in, tf.stack([feat_height, feat_width]))
```

```
    score_maps = conv2d_fixed_padding(rs_feat_maps, 1,  
                                     kernel_size=conv_ksize, scope='score_conv_{}'.format(i),  
                                     use_xavier=use_xavier, use_bias=use_bias)
```

```
    score_maps_list.append(score_maps)
```

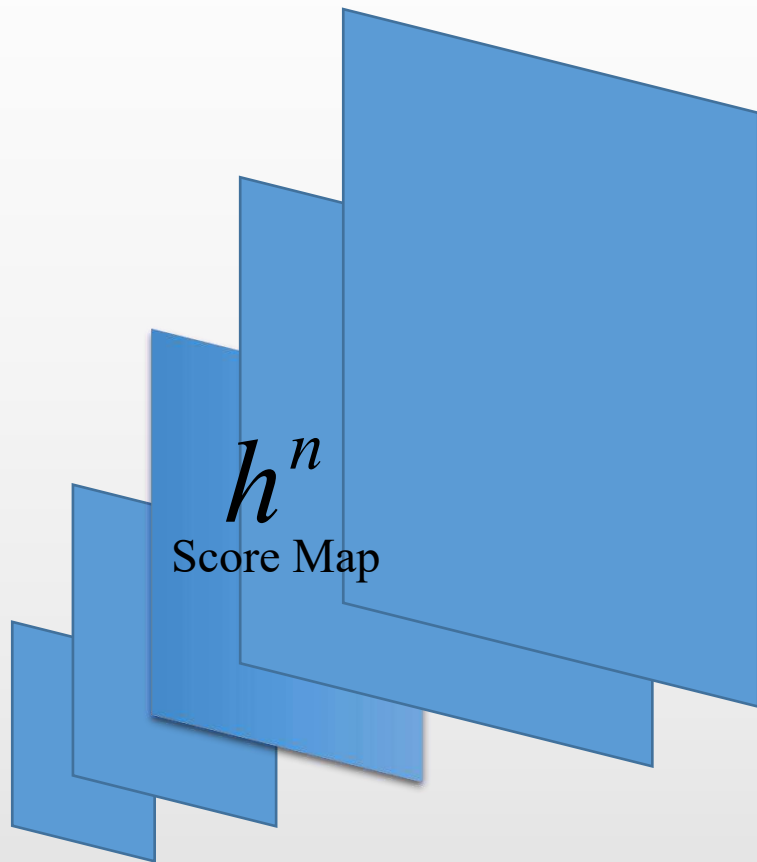
# 构建训练网络——Detector





# 构建训练网络——Detector

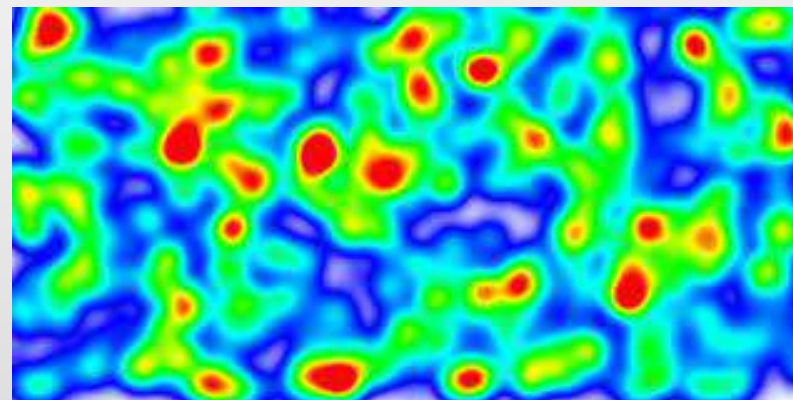
Score Map里是什么?



简单说：里面都是数字

复杂说：

- 热点图可以理解为概率响应图，通过求热点图最大值所在位置坐标，就可以得到该关键点的位置坐标；
- 但又不仅仅是预测某处出现特征点的概率，通过这个值还确定了这个点应有的尺度大小；



# 构建训练网络——Detector

## 2.3 Increase the saliency of keypoints

2.3.1 BatchNormalization进行了**归一化**，可能是想让训练更容易

```
logits = instance_normalization(score_maps_list[i])
```

2.3.2 **恢复尺寸**

```
logits = tf.image.resize_images(logits, (height, width)) # back to original resolution
```

- 这里代码和论文顺序是不一样的，但是问题不大
- 非极大值抑制同时在图片空间和尺度空间进行
- 由于非极大值抑制对尺度影响有依赖，所以我们将他们重新插值到原来的尺度

2.3.3对上面的ScoreMap施加Non-maximum suppression

```
scale_heatmaps = soft_nms_3d(scale_logits, ksize=config.sm_ksize,  
com_strength=config.com_strength)
```

目的：想让得到的ScoreMap特征点更加显著，类似锐化操作

# 构建训练网络——Detector

## soft\_nms\_3d

```
def soft_nms_3d(scale_logits, ksize, com_strength=1.0):  
    # apply softmax on scalespace logits  
    # scale_logits: [B,H,W,S]  
    num_scales = scale_logits.get_shape().as_list()[-1]  
    scale_logits_d = tf.transpose(scale_logits[...,None], [0,3,1,2,4]) # [B,S,H,W,1] in order to apply pool3d  
    max_maps = tf.nn.max_pool3d(scale_logits_d, [1,num_scales,ksize,ksize,1], [1,num_scales,1,1,1], padding='SAME')  
    max_maps = tf.transpose(max_maps[...,0], [0,2,3,1]) # [B,H,W,S]  
    exp_maps = tf.exp(com_strength * (scale_logits-max_maps))  
    exp_maps_d = tf.transpose(exp_maps[...,None], [0,3,1,2,4]) # [B,S,H,W,1]  
    sum_filter = tf.constant(np.ones((num_scales, ksize, ksize, 1, 1)), dtype=tf.float32)  
    sum_ex = tf.nn.conv3d(exp_maps_d, sum_filter, [1,num_scales,1,1,1], padding='SAME')  
    sum_ex = tf.transpose(sum_ex[...,0], [0,2,3,1]) # [B,H,W,S]  
    probs = exp_maps / (sum_ex + 1e-6)  
    return probs
```

- 为了能够训练，这一层使用了Conv来实现SoftMax操作，这样操作就可导了。
- max\_pool3d->exp->conv3d
- **3d**操作是为了在尺度方向也进行，可能也是这样造成了顺序冲突
- 使用的是softmax来做非极大值抑制

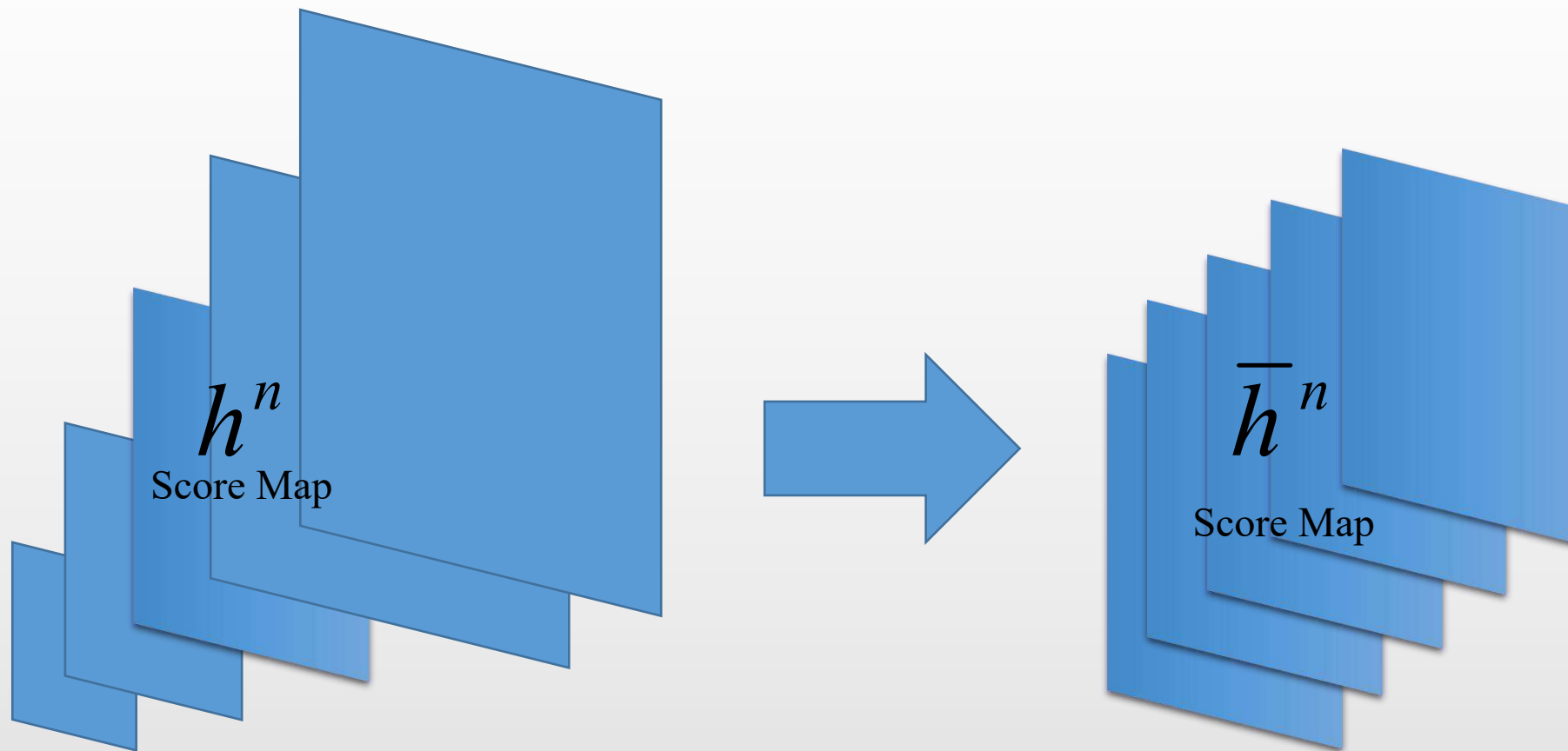
$$h^n \longrightarrow \hat{h}^n$$

由于之前说过的顺序问题  
代码中实际上得到了

$$\longrightarrow \overline{h}^n$$

# 构建训练网络——Detector

## 2.3 Get heatmap $\bar{h}^n$



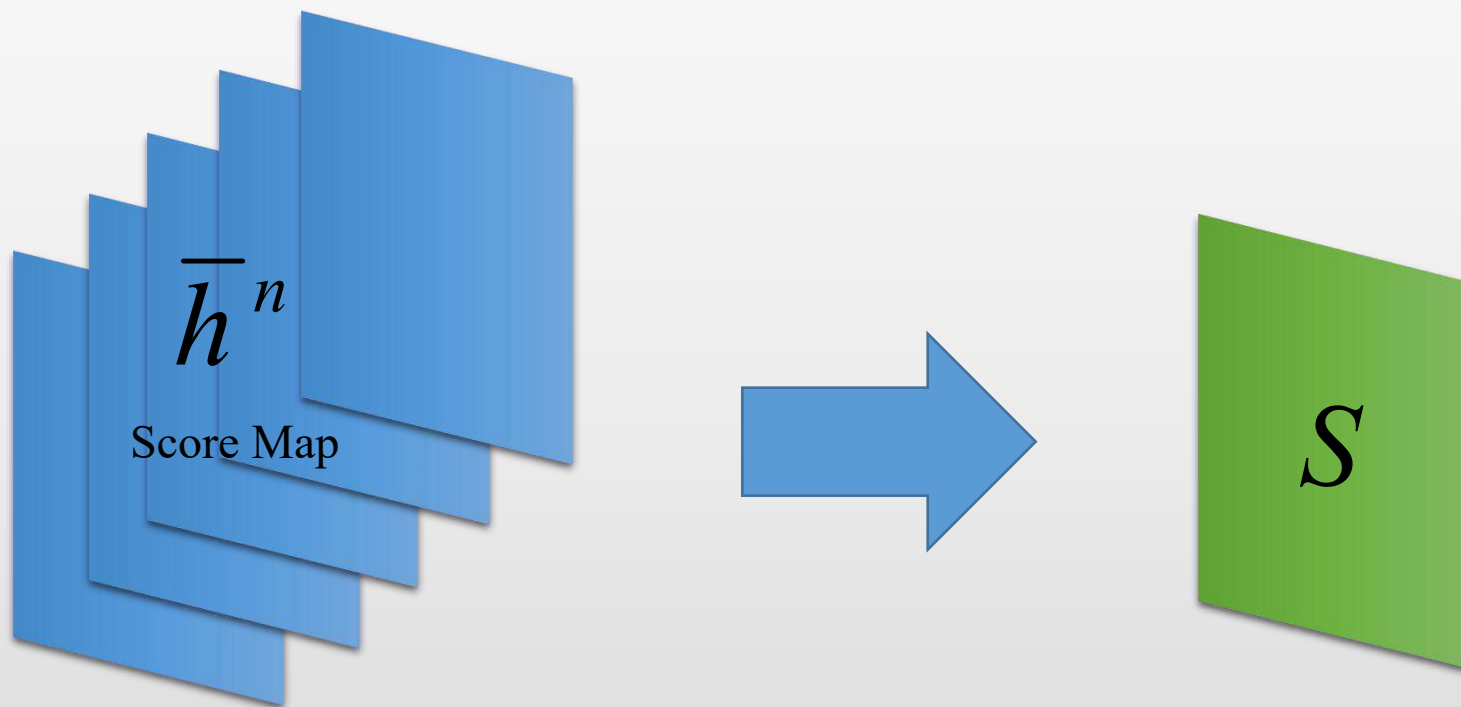
# 构建训练网络——Detector

## 2.4 Get final scale-space score map -- **S**

通过对heatmap进行softmax并进行矩阵Hadamard乘积j求和得到最终S

$$S = \sum_n \bar{h}^n \Theta softmax_n(\bar{h}^n)$$

每一张heatmap先得到极大值然后在  
尺度空间求和得到最终的scoremap  
对于尺度也要乘积得到最终尺度



# 构建训练网络——Detector

## soft\_max\_and\_argmax\_1d

```
def soft_max_and_argmax_1d(inputs, axis=-1, inputs_index=None, keep_dims=False, com_strength1=250.0, com_strength2=250.0):
```

```
    # Safe softmax
```

```
    inputs_exp1 = tf.exp(com_strength1*(inputs - tf.reduce_max(inputs, axis=axis, keep_dims=True)))
```

```
    inputs_softmax1 = inputs_exp1 / (tf.reduce_sum(inputs_exp1, axis=axis, keep_dims=True) + 1e-8)
```

```
    inputs_exp2 = tf.exp(com_strength2*(inputs - tf.reduce_max(inputs, axis=axis, keep_dims=True)))
```

```
    inputs_softmax2 = inputs_exp2 / (tf.reduce_sum(inputs_exp2, axis=axis, keep_dims=True) + 1e-8)
```

```
    inputs_max = tf.reduce_sum(inputs * inputs_softmax1, axis=axis, keep_dims=keep_dims)
```

```
    inputs_index_shp = [1,]*len(inputs.get_shape())
```

```
    inputs_index_shp[axis] = -1
```

```
    if inputs_index is None:
```

```
        inputs_index = tf.range(inputs.get_shape().as_list()[axis], dtype=inputs.dtype) # use 0,1,2,...,inputs.get_shape()[axis]
```

```
    inputs_index = tf.reshape(inputs_index, inputs_index_shp)
```

```
    inputs_amax = tf.reduce_sum(inputs_index * inputs_softmax2, axis=axis, keep_dims=keep_dims)
```

```
    return inputs_max, inputs_amax
```

$$S = \sum_n \bar{h}^n \Theta_{softmax_n}(\bar{h}^n)$$

Get ScoreMap

$$\text{softargmax}(x) = \sum_i \frac{e^{\beta x_i}}{\sum_j e^{\beta x_j}} i$$

Get ScaleMap

# 构建训练网络——Detector

soft\_argmax确定每个像素尺度

$$\text{softargmax}(x) = \sum_i \frac{e^{\beta x_i}}{\sum_j e^{\beta x_j}} i$$


Input: Scale\_heatmap(多尺度热力图)

Output: max\_heatmap+max\_scale

将尺度通道整合在一起，直接得到的是一张地图

softargmax的作用就是要找到每一点的尺度到底是多少，之前不是我们整了一个N个尺度的金字塔么，现在我们得确定某一点他到底在哪个尺度上最显著，所以他先用一个softmax得到heatmap上的值，这样做的结果就是说只想要最显著的尺度然后其他的都抑制到最小，当然有时确实不是0，然后将scale\_factor成上去，这样就能得到比较灵活的尺度信息，不是原来N层那么生硬。

用soft就是想要训练，要不然其实直接用max和argmax也能得到相同结果。

scale	0.707	0.8536	1.0	1.2071	1.414		1.209
score	0	0.362	0.9	0	0		



# 构建训练网络——Detector

## 2.5 Mask

生成各种mask(map上都是0和1)主要是有这几种：

- 各种前面卷积网络padding的map边缘部分要去掉，应当是为了让结果更高，毕竟padding了一大圈0对原始图片肯定有不好影响
- 为了提取patch所以也要去掉map边缘
- 为了提高鲁棒性根据深度图要去掉看不到深度的地方

```
eof_masks_pad = end_of_frame_masks(height, width, det_endpoints['pad_size'])
```

```
eof_masks_crop = end_of_frame_masks(height, width, config.crop_radius)
```





# 构建训练网络——Detector

## 2.6 Extract Keypoints

Input: final scale-space score map -- S

Output: K points

对S做一个特殊的**非极大值抑制**  
这里得到的不是score, 而是**PeakMask**  
**没有真的改变scoremap, 不会影响训练**

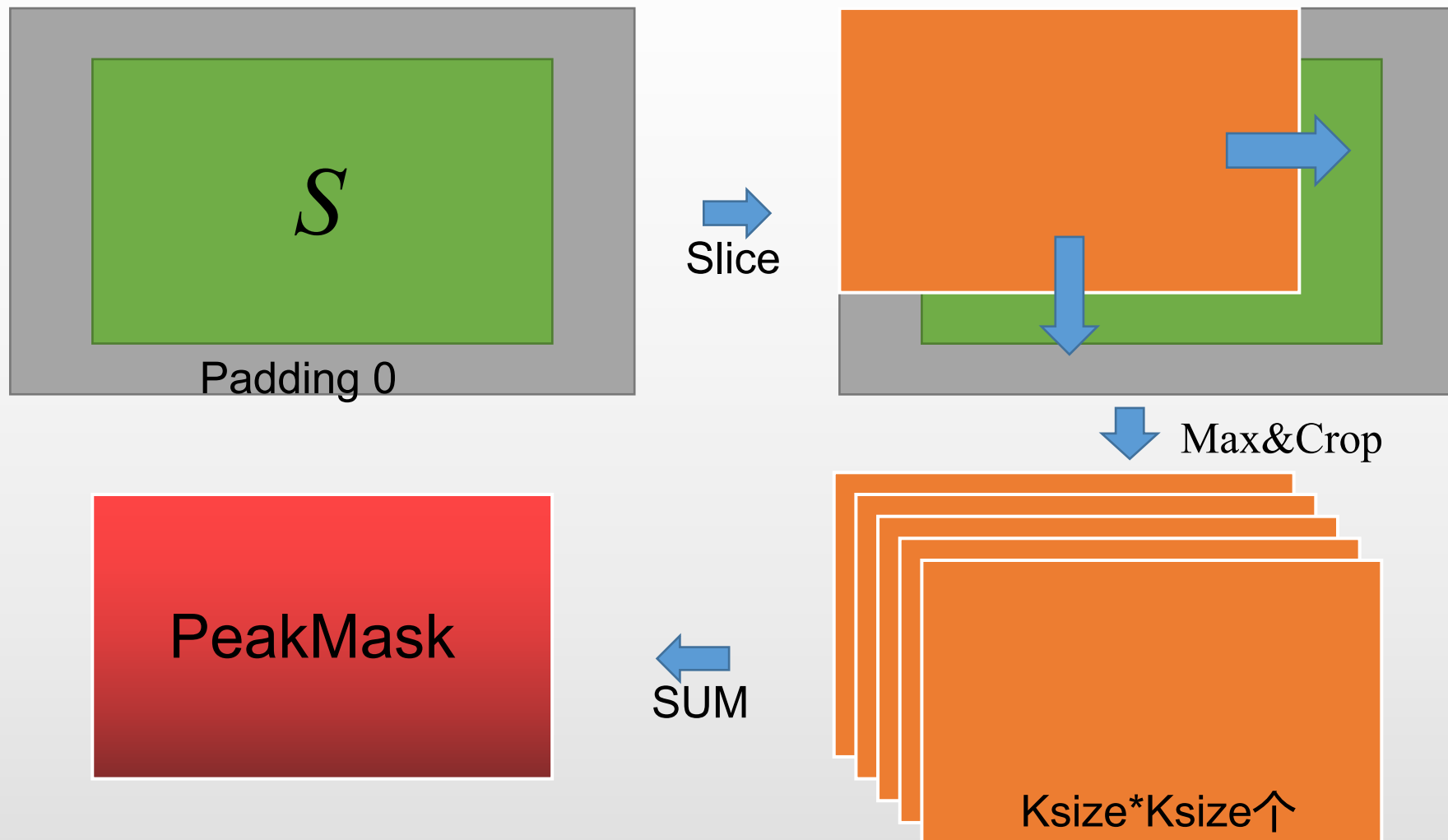
在这个nms中生成的是一个除了极大值其他地方都是0的map  
用的方法是将一个比较大的padding地图进行滑动, 这样只会取得窗口大小的极大值

```
def non_max_suppression(inputs, thresh=0.0, ksize=3, dtype=tf.float32, name='NMS'):
    .....
    hk = ksize // 2
    zeros = tf.zeros_like(inputs)
    works = tf.where(tf.less(inputs, thresh), zeros, inputs)
    works_pad = tf.pad(works, [[0,0], [2*hk,2*hk], [2*hk,2*hk], [0,0]], mode='CONSTANT')
    map_augs = []
    for i in range(ksize):
        for j in range(ksize):
            curr_in = tf.slice(works_pad, [0, i, j, 0], [-1, height+2*hk, width+2*hk, -1])
            map_augs.append(curr_in)

    num_map = len(map_augs) # ksize*ksize
    center_map = map_augs[num_map//2]
    peak_mask = tf.greater(center_map, map_augs[0])
    for n in range(1, num_map):
        if n == num_map // 2:
            continue
        peak_mask = tf.logical_and(peak_mask, tf.greater(center_map, map_augs[n]))
    peak_mask = tf.slice(peak_mask, [0,hk,hk,0],[-1,height,width,-1])
    if dtype != tf.bool:
        peak_mask = tf.cast(peak_mask, dtype=dtype)
    peak_mask.set_shape(inputs.shape) # keep shape information
    return peak_mask
```

# 构建训练网络——Detector

non\_max\_suppression



# 构建训练网络——Detector

## 2.7 Get dxdy

Input: masked final scale-space score map --  $S$

Output:  $K$  keypoints with sub-pixel accuracy

使用二维的soft-argmax可以让局部特征点期望坐标更准确，得到的dxdy就是sub-pixel精度

```
# keypoint refinement
# Use transformer crop to get the patches for refining keypoints to a certain size.
kp_local_max_scores = transformer_crop(max_heatmaps, config.kp_loc_size, batch_inds, kpts,
                                       kpts_scale=kpts_scale) # omit orientation [N, loc_size, loc_size, 1]

dxdy = soft_argmax_2d(kp_local_max_scores, config.kp_loc_size, do_softmax=config.do_softmax_kp_refine, com_strength=config.kp_com_strength) # [N,2]
tf.summary.histogram('dxdy', dxdy)
# Now add this to the current kpts, so that we can be happy!
kpts = tf.to_float(kpts) + dxdy * kpts_scale[:, None] * config.kp_loc_size / 2
```

# 构建训练网络——Detector

## 2.8 Orientation estimaton

- 就是在建立网络时候在feature map后面加了一个输出为2通道的卷积层，两个通道就代表sin和cos
- 为了让点代表真实角度，有加了L2正则化，这样就强制平方和为1

```
ori_maps = conv2d_custom(curr_in, 2,  
                          kernel_size=ori_conv_ksize, scope='ori_conv',  
                          W_initializer=ori_W_init,  
                          b_initializer=ori_b_init)  
ori_maps = tf.nn.l2_normalize(ori_maps, dim=-1)
```

# 构建训练网络——Detector

## 2.8 Orientation estimaton

- 就是在建立网络时候在feature map后面加了一个输出为2通道的卷积层，两个通道就代表sin和cos
- 为了让点代表真实角度，有加了L2正则化，这样就强制平方和为1

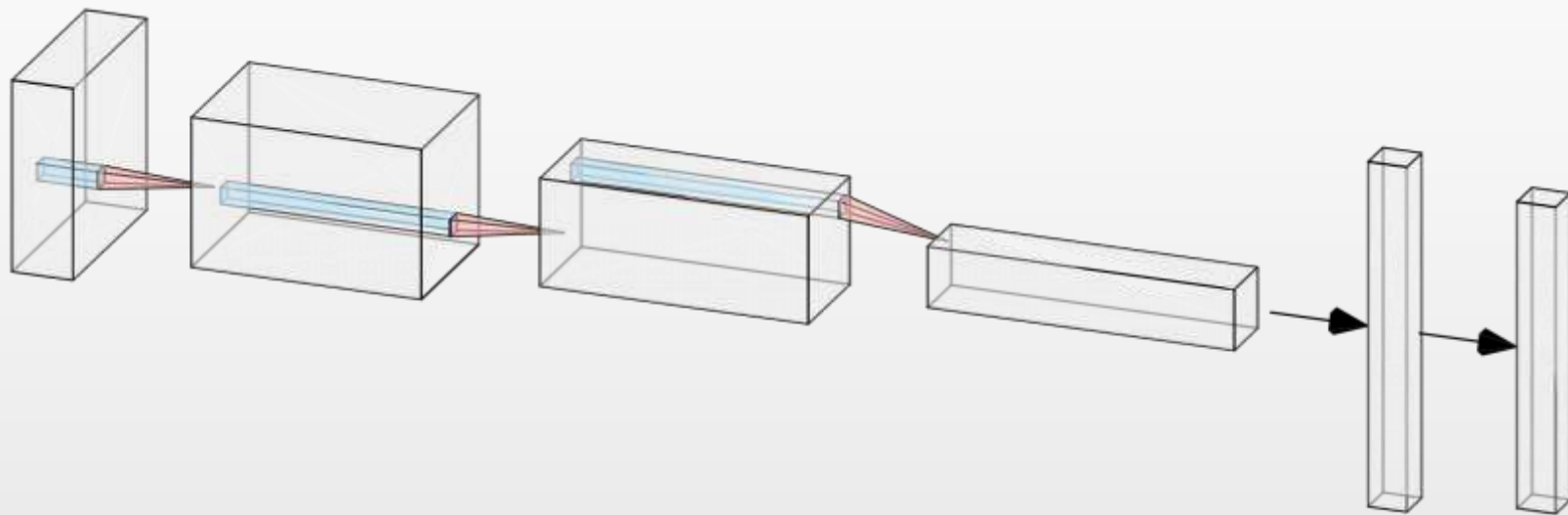
```
ori_maps = conv2d_custom(curr_in, 2,  
                          kernel_size=ori_conv_ksize, scope='ori_conv',  
                          W_initializer=ori_W_init,  
                          b_initializer=ori_b_init)  
ori_maps = tf.nn.l2_normalize(ori_maps, dim=-1)
```

# 构建训练网络——Descriptor

使用双线性插值使得输入patch为 $32 \times 32$ 同时能够可微

输入图片需要归一化一下

三层 $3 \times 3$ 卷积+batchnorm+ReLU



得到的最终向量L2正则化

# 训练过程

类似孪生网络

*branch • I*

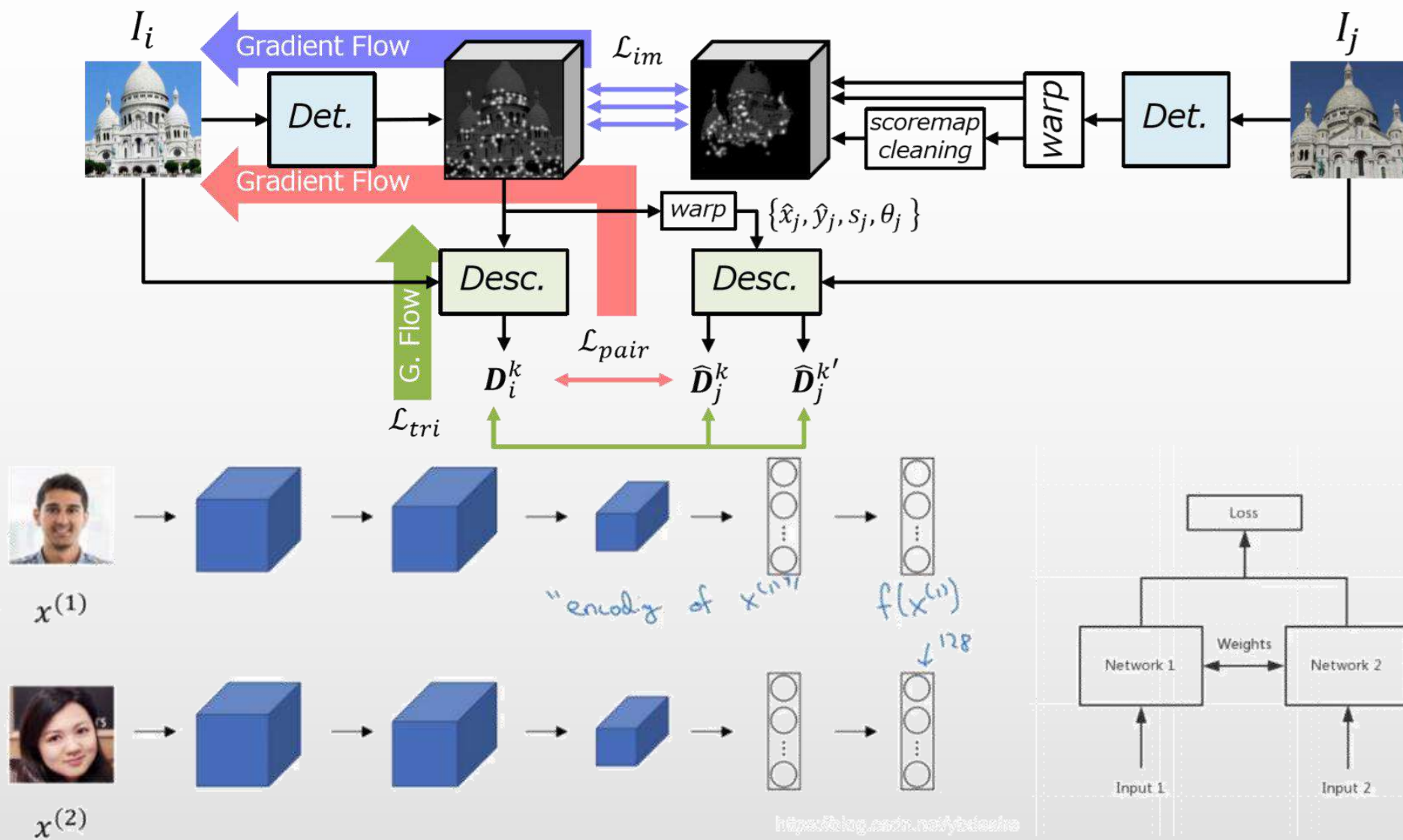


*branch • J*



实际上IJ都参加了训练

# 训练过程

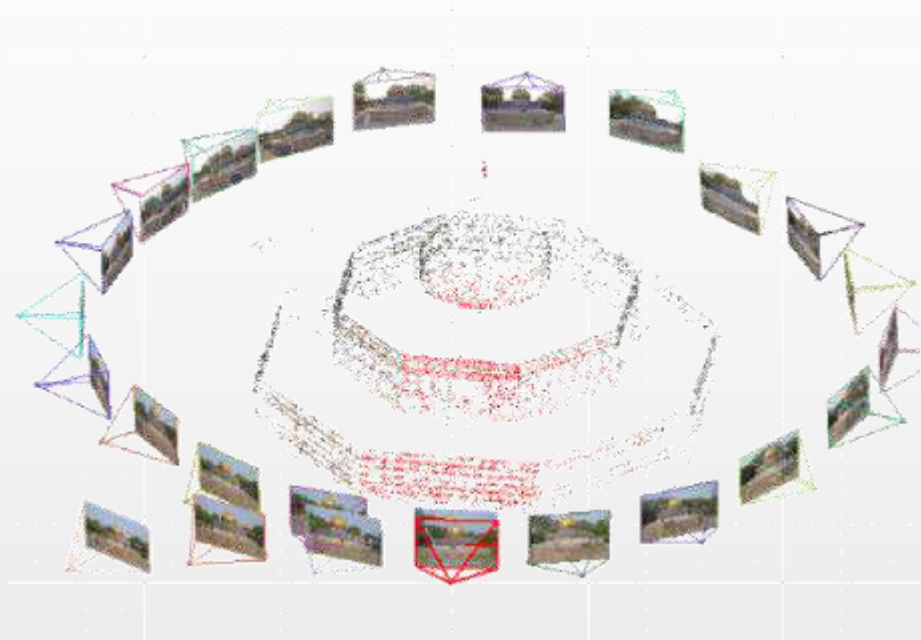




# 训练过程

输入：color图片+depth+相机内参+外参

SFM得到了相机内外参数



有了上述信息我们可以找到两张图片在同场景下的一个对应



# 训练过程



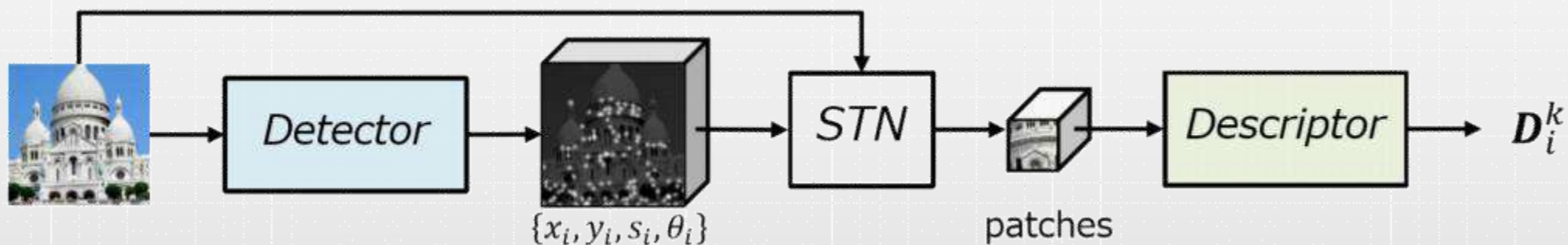
怎么确定对应点  
?

# 策略

## 损失函数

$$\zeta_{\text{det}} = \zeta_{\text{im}} + \lambda_{\text{pair}} \zeta_{\text{pair}} + \zeta_{\text{geom}}$$

$$\zeta_{\text{desc}} = \zeta_{\text{tri}}$$



# 损失函数

$$\zeta_{\text{det}} = \zeta_{im} + \lambda_{pair} \zeta_{pair} + \zeta_{geom}$$



$$\zeta_{im}$$

# 损失函数——Image-level loss

用这个来训练detector知道在哪里生成高的得分

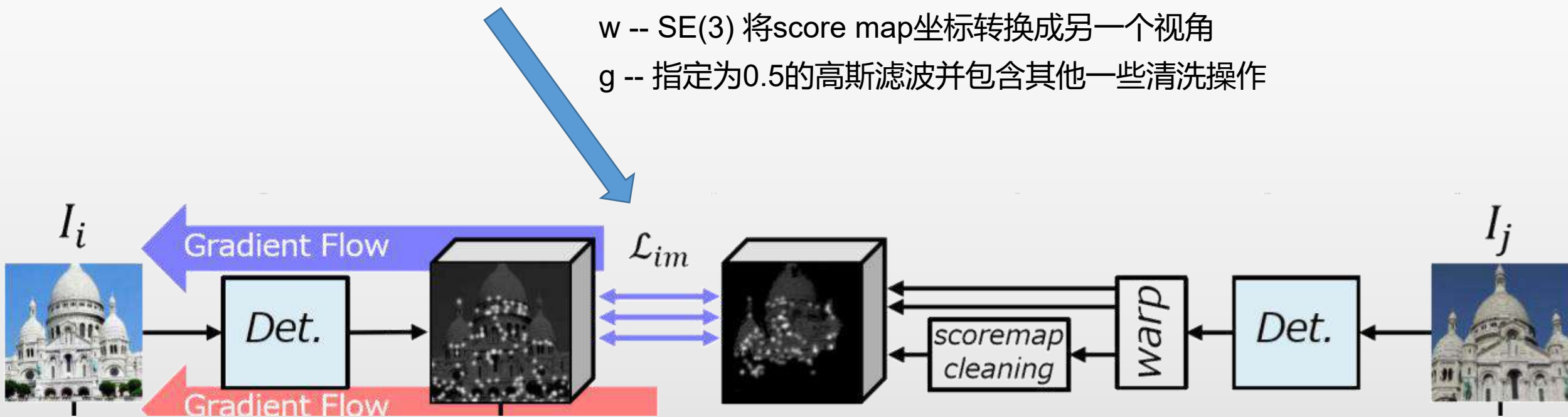
Input : ScoreMaps + depth maps + camera intrinsics + extrinsics

Output : Loss


$$\zeta_{im}(S_i, S_j) = |S_i - g(w(S_j))|^2$$

$w$  -- SE(3) 将score map坐标转换成另一个视角

$g$  -- 指定为0.5的高斯滤波并包含其他一些清洗操作



# 损失函数——Image-level loss

$w(S_j)$   `inverse_warp_view_2_to_1`  
(heatmaps2, depths2, depths1, c2到c1相机变换矩阵 降4\*4 `c2_to_c1`, 内参 `K1`, `K2`, 平面内转换 `inv_thetas1`, `thetas2`, `depth_thresh=0.5`, `get_warped_depth=False`)

通过SE(3)转换了map, 还用depth形成了一个mask遮去小于depth\_thresh的地方

```
nonocc_masks = tf.cast(tf.less(tf.squared_difference(depths1w, depths1),  
depth_thresh**2), tf.float32)
```

返回变换好的heatmap以及mask

以上操作不可避免会截段求导流所以J分支不会进行训练

# 损失函数——Image-level loss

$$g(w(S_j))$$


```
gt_heatmaps1 = tf.nn.conv2d(top_k1w, psf, [1,1,1,1], padding='SAME')
gt_heatmaps1 = tf.minimum(gt_heatmaps1, 1.0)
```

```
psf ← tf.constant(get_gauss_filter_weight(config.hm_ksize,  
config.hm_sigma)[:,:,:None,None], dtype=tf.float32)
```

Input : topk map\*heatmap

Output : clean score map

就是只有想要的特征点的score存在，其他位置都是0，然后对这个map进行了高斯卷积

g操作不仅把mask后的map用高斯在特征点周围扩散，还将最大值压制在了1

- 极大值抑制对训练比较好
- 并且产生了一个更纯净更高的好的结果

0	0	0	0	0
0	0	0	0	0
0	0	.9	0	0
0	0	0	0	0
0	0	0	0	0



0	0	0	0	0
0	.4	.4	.4	0
0	.4	1	.4	0
0	.4	4	.4	0
0	0	0	0	0

# 损失函数——Image-level loss

具体训练过程代码中其实是：

```
l2diff1 = tf.squared_difference(tgt_heatmaps1, gt_heatmaps1)
loss1 = tf.reduce_mean( tf.reduce_sum(l2diff1 * visible_masks1, axis=axis123) / Nvis1 )
```

并且同时在左右batch上求了平均值，来进一步使得训练稳定

$$\zeta_{im}(S_i, S_j) = \frac{|S_i - g(w(S_j))|^2 + |S_j - g(w(S_i))|^2}{2}$$

分析：

- 进行clean 步骤原因可能是想强化一下学习的目的，因为我们想学习的是特征点坐标，如果说不进行clean，在L2loss时候网络有可能在把不是特征点的地方给加大（最后结果也是loss减小但这不是我们想要的）
- 求平均值目的就是要相互学习，提高训练效率

Loss的目的：

- 就是为了能训练出特征点检测器
- 两张图像相互使得score map相近可能还不足以得到真正的特征点，但是当数据足够多的时候网络就会在足够鲁棒地地方生成特征点，比如一下不变的角点，而天空等空白区域则没有



# 损失函数——Image-level loss

```
l2diff1 = tf.squared_difference(tgt_heatmaps1, gt_heatmaps1)  
loss1 = tf.reduce_mean( tf.reduce_sum(l2diff1 * visible_masks1, axis=axis123) / Nvis1 )
```

```
l2diff2 = tf.squared_difference(tgt_heatmaps2, gt_heatmaps2)  
loss2 = tf.reduce_mean( tf.reduce_sum(l2diff2 * visible_masks2, axis=axis123) / Nvis2 )
```

```
det_loss = (loss1 + loss2) / 2.0
```



# 损失函数

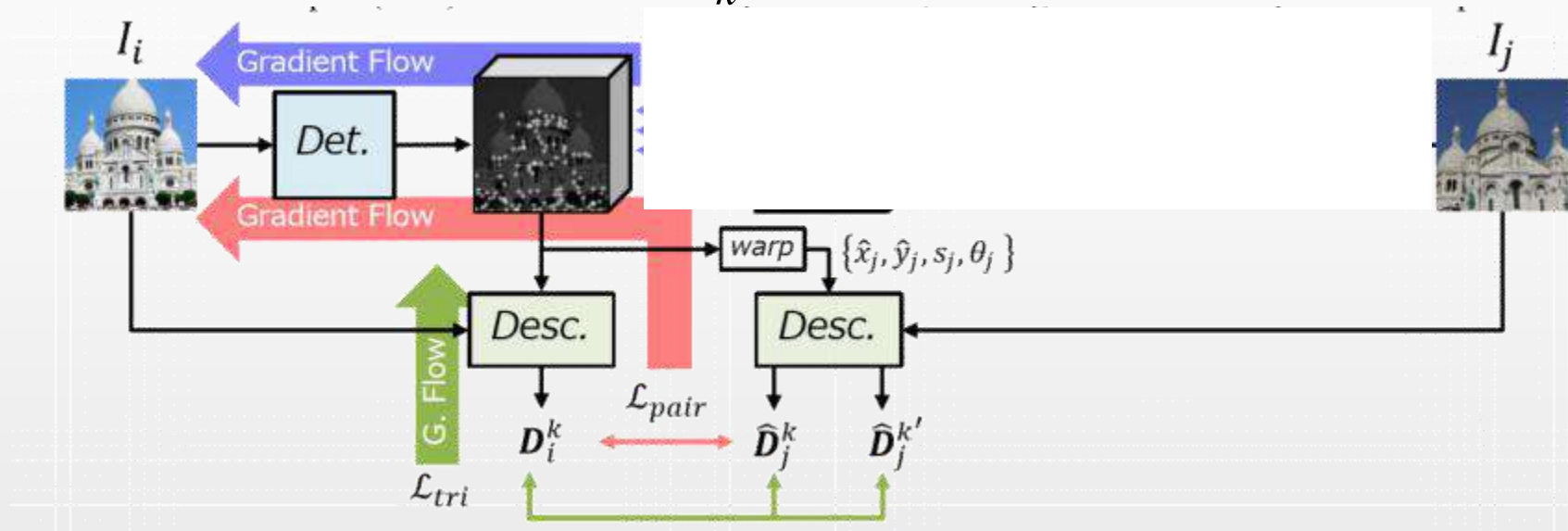
$$\zeta_{\text{det}} = \zeta_{\text{im}} + \lambda_{\text{pair}} \zeta_{\text{pair}} + \zeta_{\text{geom}}$$



# 损失函数——Patch-wise loss

Input: 将k个特征点以scale、ori的信息提出的patch(原始图片)

Output:  $\zeta_{pair}(D_i^k, \hat{D}_j^k) = \sum_k |D_i^k - \hat{D}_j^k|^2$



Patch-wise 为了训练detector warp的顺序相反是把 $I_i$ 的patch warp到 $I_i$ 上来对比des, 因为这样才能让得到的loss反向流到det中去改进。

# 损失函数——Patch-wise loss

$$\zeta_{pair}(D_i^k, \hat{D}_j^k) = \sum_k |D_i^k - \hat{D}_j^k|^2$$

```
d_pos = tf.reduce_sum(tf.square(desc_feats1-desc_feats1_pos), axis=1) # [B*K,]
```

```
desc_pair_loss = tf.reduce_mean(d_pos)
```

D是特征值网络从patch中提取的特征向量

Loss的目的:

- 训练特征点尺度、角度去做匹配
- 因为其实这里关于特征点位置信息已经没法流回去了，而又没有流到descriptor网络中，这样做的目的应该是让得到的patch大小方向信息更加适合
- descriptor能够帮助提升detector表现
- 但是文中说取消这个loss影响很小，这么看其实detector和des在训练中其实是独立的，可以不依赖des去训练det

# 损失函数

$$\zeta_{\text{det}} = \zeta_{\text{im}} + \lambda_{\text{pair}} \zeta_{\text{pair}} + \zeta_{\text{geom}}$$



# 损失函数——Patch-wise loss

$$\zeta_{pair}(s_i^k, \theta_i^k, \hat{s}_j^k, \hat{\theta}_j^k) = \lambda_{ori} \sum_k |\theta_i^k - \hat{\theta}_j^k|^2 + \lambda_{scale} \sum_k |s_i^k - \hat{s}_j^k|^2$$

```
ori_loss1 = tf.squared_difference(ori_maps1, ori_maps1w)
ori_loss1 = tf.reduce_mean( tf.reduce_sum(ori_loss1 * visible_masks1, axis=axis123) / Nvis1 )
ori_loss2 = tf.squared_difference(ori_maps2, ori_maps2w)
ori_loss2 = tf.reduce_mean( tf.reduce_sum(ori_loss2 * visible_masks2, axis=axis123) / Nvis2 )
ori_loss = (ori_loss1 + ori_loss2) * 0.5
```

```
scale_loss1 = tf.squared_difference(tf.log(scale_maps1), tf.log(scale_maps1w))
max_scale_loss1 = tf.reduce_max(scale_loss1)
scale_loss1 = tf.reduce_mean(tf.reduce_sum(scale_loss1 * visible_masks1, axis=axis123) / Nvis1)
scale_loss2 = tf.squared_difference(tf.log(scale_maps2), tf.log(scale_maps2w))
max_scale_loss2 = tf.reduce_max(scale_loss2)
scale_loss2 = tf.reduce_mean(tf.reduce_sum(scale_loss2 * visible_masks2, axis=axis123) / Nvis2)
scale_loss = (scale_loss1 + scale_loss2) * 0.5
```

# 损失函数——Patch-wise loss

$$\zeta_{pair}(s_i^k, \theta_i^k, \hat{s}_j^k, \hat{\theta}_j^k) = \lambda_{ori} \sum_k |\theta_i^k - \hat{\theta}_j^k|^2 + \lambda_{scale} \sum_k |s_i^k - \hat{s}_j^k|^2$$

这个损失函数的目的就是让得到的ori和scale要一样

方向和尺度本来就是为了得到图像中的不变性

- 参数在限制角度和尺度变化速度，太快有可能什么都学不到，因为这里面descriptor得到的结果并不好，需要限制一下；
- 事实上Ori这个预测器在其他论文中也说可能存在很多极小值点使得训练起来困难，本文使用强化学习会更难训练；



# 损失函数

$$\zeta_{\text{desc}} = \zeta_{tri}$$

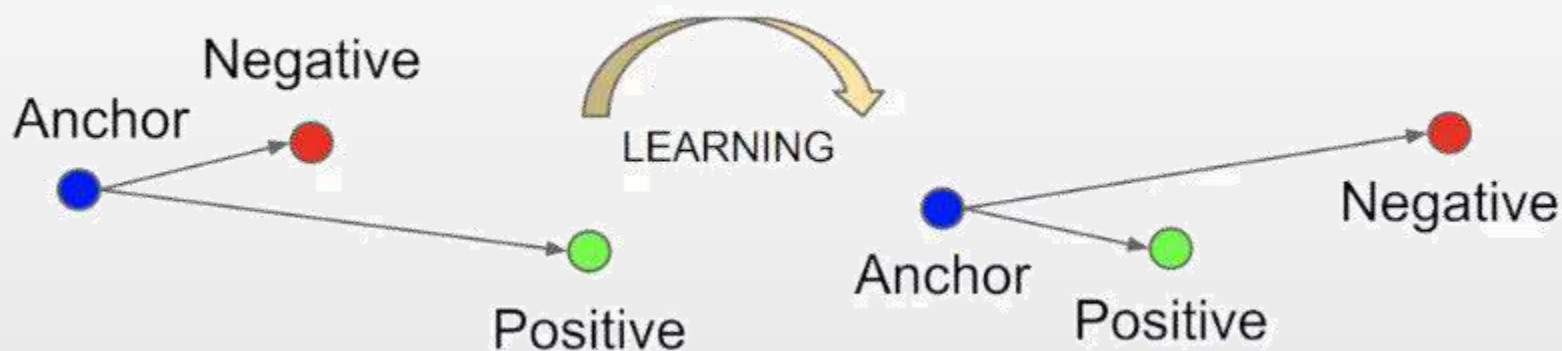

$$\zeta_{tri}$$



# 损失函数——Triplet loss

$$\zeta_{tri}(D_i^k, \hat{D}_j^k, \hat{D}_j^{k'}) = \sum_k \max(0, |D_i^k - \hat{D}_j^k|^2 - |D_i^k - \hat{D}_j^{k'}|^2 + C)$$

Triplet Loss的核心是锚示例、正示例、负示例共享模型，通过模型，将锚示例与正示例聚类，远离负示例。



# 损失函数——Triplet loss

$$\zeta_{tri}(D_i^k, \hat{D}_j^k, \hat{D}_j^{k'}) = \sum_k \max(0, |D_i^k - \hat{D}_j^k|^2 - |D_i^k - \hat{D}_j^{k'}|^2 + C)$$

正示例: Ground-truth

负示例: Sampling informative patches

```
find_random_hard_negative_from_myself_with_geom_constrain_less_memory():  
    feat_dists = tf.reduce_sum(tf.squared_difference(feats1_mat, feats2_mat), axis=-1) # [K,K]  
    geom_dists = tf.reduce_sum(tf.squared_difference(kp1_mat, kp2_mat), axis=-1) # [K,K]  
    neighbor_penalty = tf.cast(tf.less_equal(geom_dists, geom_sq_thresh), tf.float32) * 1e5  
    feat_dists = feat_dists + neighbor_penalty # avoid to pickup from neighborhood  
    topk_dist, topk_inds = tf.nn.top_k(-feat_dists, k=num_pickup, sorted=False) # take the  
    smallest value
```

负样本在featmap负样本中排序选择最相近M(5~64)个patch中选择  
同时还注意到了几何限制，不让特征点距离太近，这样子不利于生成不同的描述子

# 最终损失函数

$$\zeta_{\text{det}} = \zeta_{\text{im}} + \lambda_{\text{pair}} \zeta_{\text{pair}} + \zeta_{\text{geom}}$$

$$\zeta_{\text{desc}} = \zeta_{\text{tri}}$$

# 训练

- 在训练中每个branch他都把图片i和j反转一下，这也是为什么会出现loss函数求和之后再求平均值
- 整个网络其实只有一份，并没有两个网络交替训练，训练时候左右两侧网络是一摸一样的

## 主要参数：

### 提取特征点数量：

- ScanNet -- 512
- Outdoor -- 1024

Learning rate -- 学习率

hm\_ksize -- 提取l\_j特征点时高斯核大小

nms\_thresh -- 提取l\_j特征点时的抑制程度

nms\_ksize -- 抑制的核大小，取决取多大范围的局部极大值5

weight\_det\_loss -- pairwise-loss的重要程度

scale\_weight -- 损失函数中scale的重要程度(default:0.1)

desc\_margin -- 描述子损失函数中正负样本拉伸程度(default:1.0)

crop\_radius -- 图像边缘要去的宽度(default:16)

patch\_size -- 提取patch大小(default:32)

depth\_thresh -- 要mask掉的部分(default:1.0)

sm\_ksize/com\_strength -- heatmap进行极大值抑制时候的程度

scale\_com\_strength -- 确定最终scalemap的程度值

kp\_loc\_size -- 在确定kp后抑制纯净

# Dataset

## ScanNet数据集

- 数据集中Groundtruth的pose等信息
- 为了使得两张图片间有共同的物体，限制在15帧距离

## 室外数据集

- 使用SFM程序来生成pose
- 怎么找到有共同点的两张图片
- 在sfm结果中进行可见性检测

## 深度图片中的噪声

我们通过三维重建找到对应的深度，如果同depth图片中的值差距过大，我们就认为深度图中那个像素是错的

# Dataset

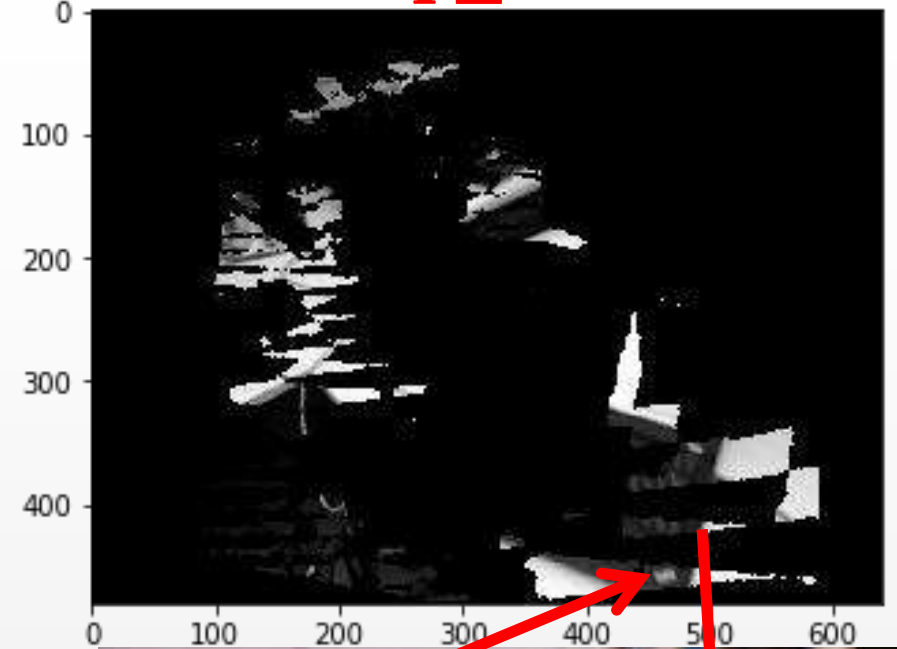
inverse\_warp\_view\_2\_to\_1

(heatmaps2, depths2, depths1, c2Tc1s, K1, K2, inv\_thetas1, thetas2, depth\_thresh=0.5, get\_warped\_depth=False)

view2

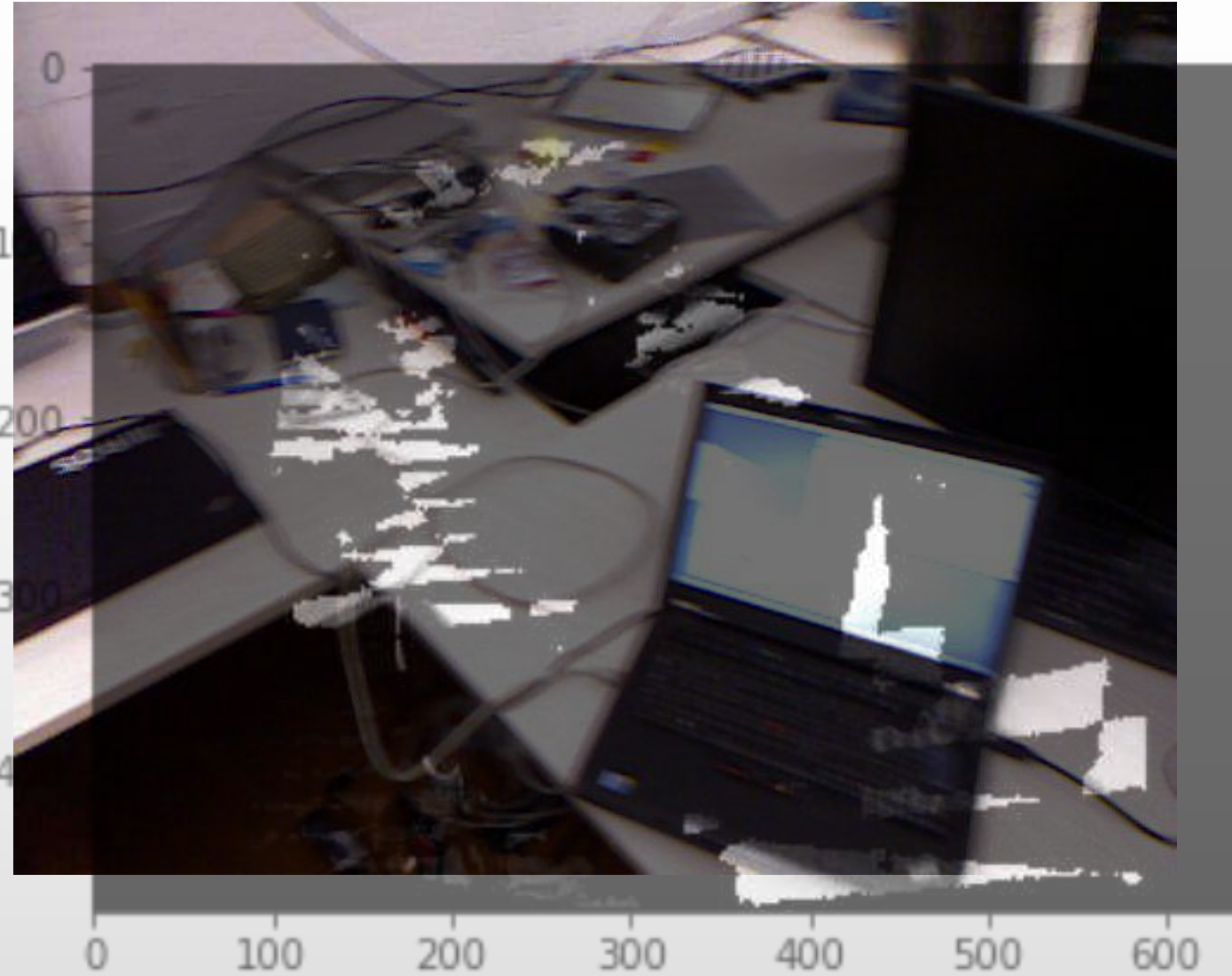


wrap\_view2



view1

# Dataset





# Dataset

## inverse\_warp\_view\_2\_to\_1

(heatmaps2, depths2, depths1, c2Tc1s, K1, K2, inv\_thetas1, thetas2, depth\_thresh=0.5, get\_warped\_depth=False)

norm\_meshgrid

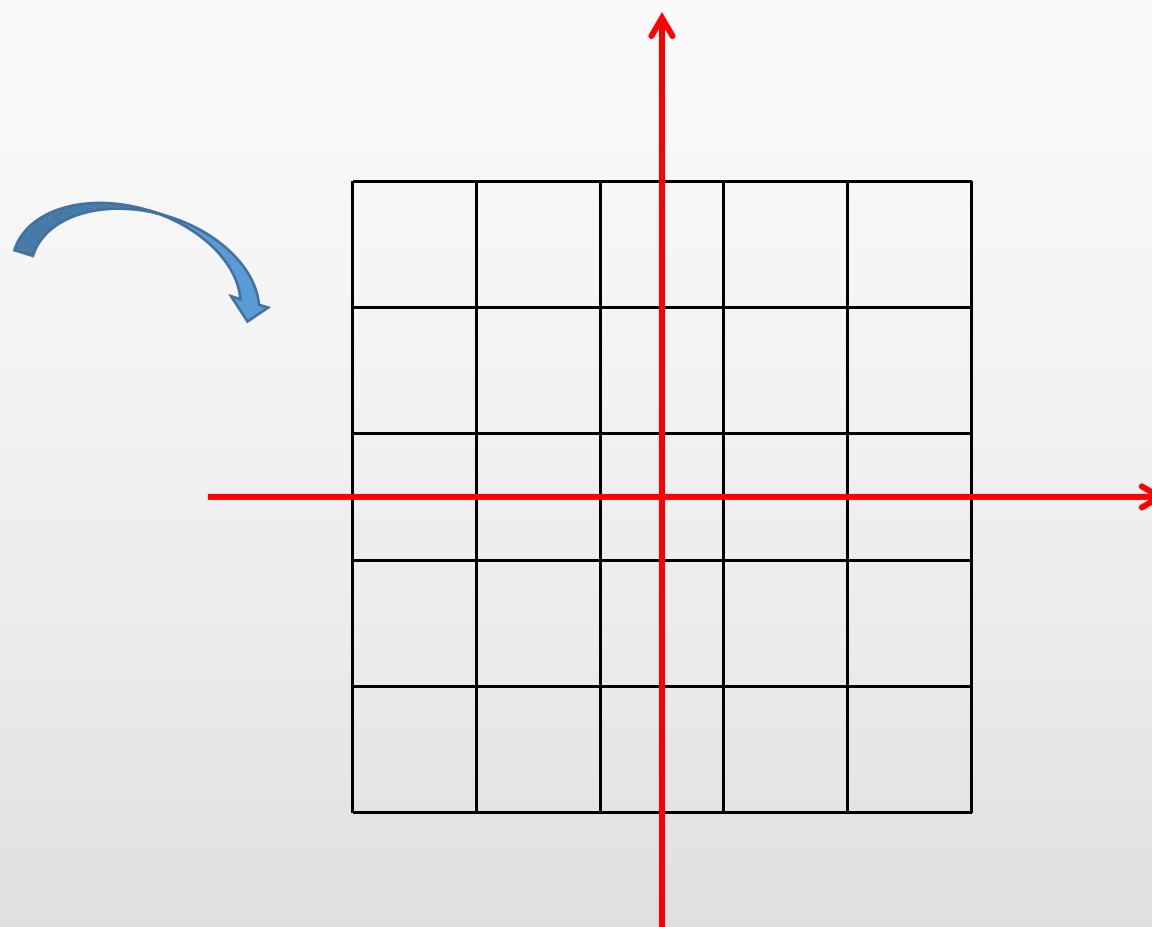
转换为齐次坐标的基本形式，方便以后处理

```
xy_n1 = norm_meshgrid(batch_size, height1, width1)
# [B,3,H,W]
xy_n1 = tf.reshape(xy_n1, [batch_size, 3, -1])
# [B,3,N], N=H*W
```

在这个归一化的坐标系中进行平面转换

if inv\_thetas1 is not None:

```
xy_n1 = tf.matmul(inv_thetas1, xy_n1)
z_n1 = tf.slice(xy_n1, [0,2,0],[-1,1,-1])
xy_n1 = xy_n1 / (z_n1+eps)
```





# Dataset

## inverse\_warp\_view\_2\_to\_1

(heatmaps2, depths2, depths1, c2Tc1s, K1, K2, inv\_thetas1,  
thetas2, depth\_thresh=0.5, get\_warped\_depth=False)

# SE(3) transformation : pixel1 to camera1

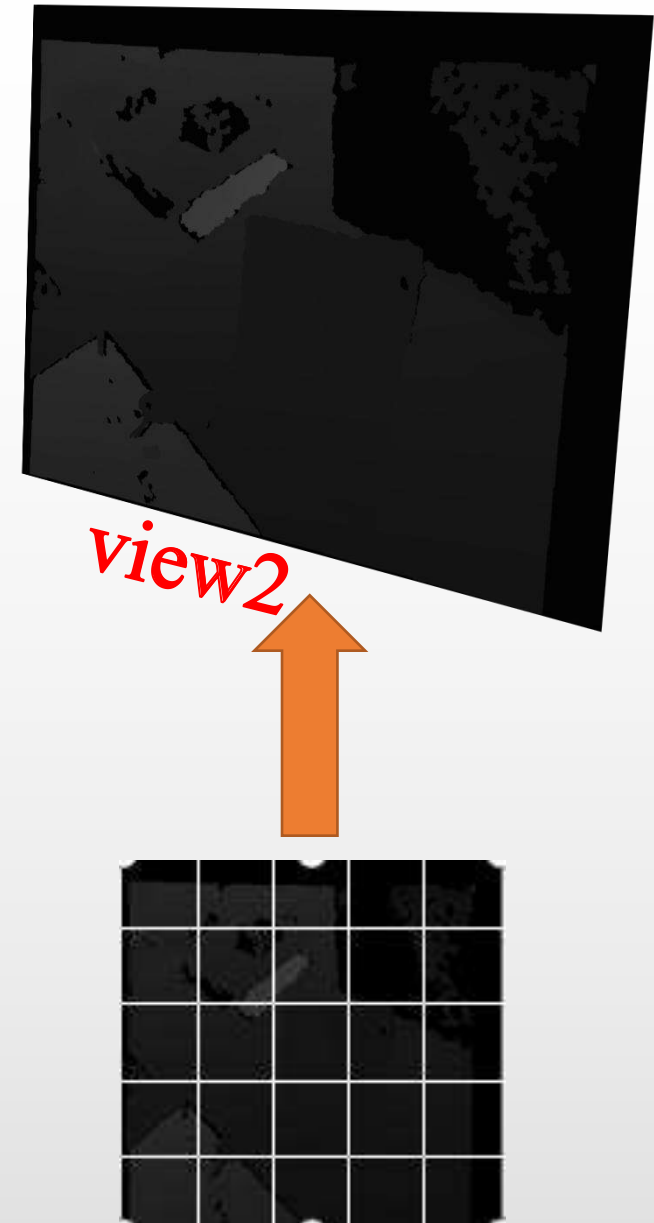
Z1 = tf.reshape(depths1, [batch\_size, 1, -1])

xy\_u1 = unnorm\_xy\_coords(xy\_n1, height=height1, width=width1)

XYZ1 = tf.matmul(inv\_K1, xy\_u1) \* Z1

ones = tf.ones([batch\_size, 1, height1\*width1])

XYZ1 = tf.concat([XYZ1, ones], axis=1)



# Dataset

## inverse\_warp\_view\_2\_to\_1

(heatmaps2, depths2, depths1, c2Tc1s, K1, K2, inv\_thetas1, thetas2, depth\_thresh=0.5, get\_warped\_depth=False)

# SE(3) transformation : camera1 to camera2 to pixel2

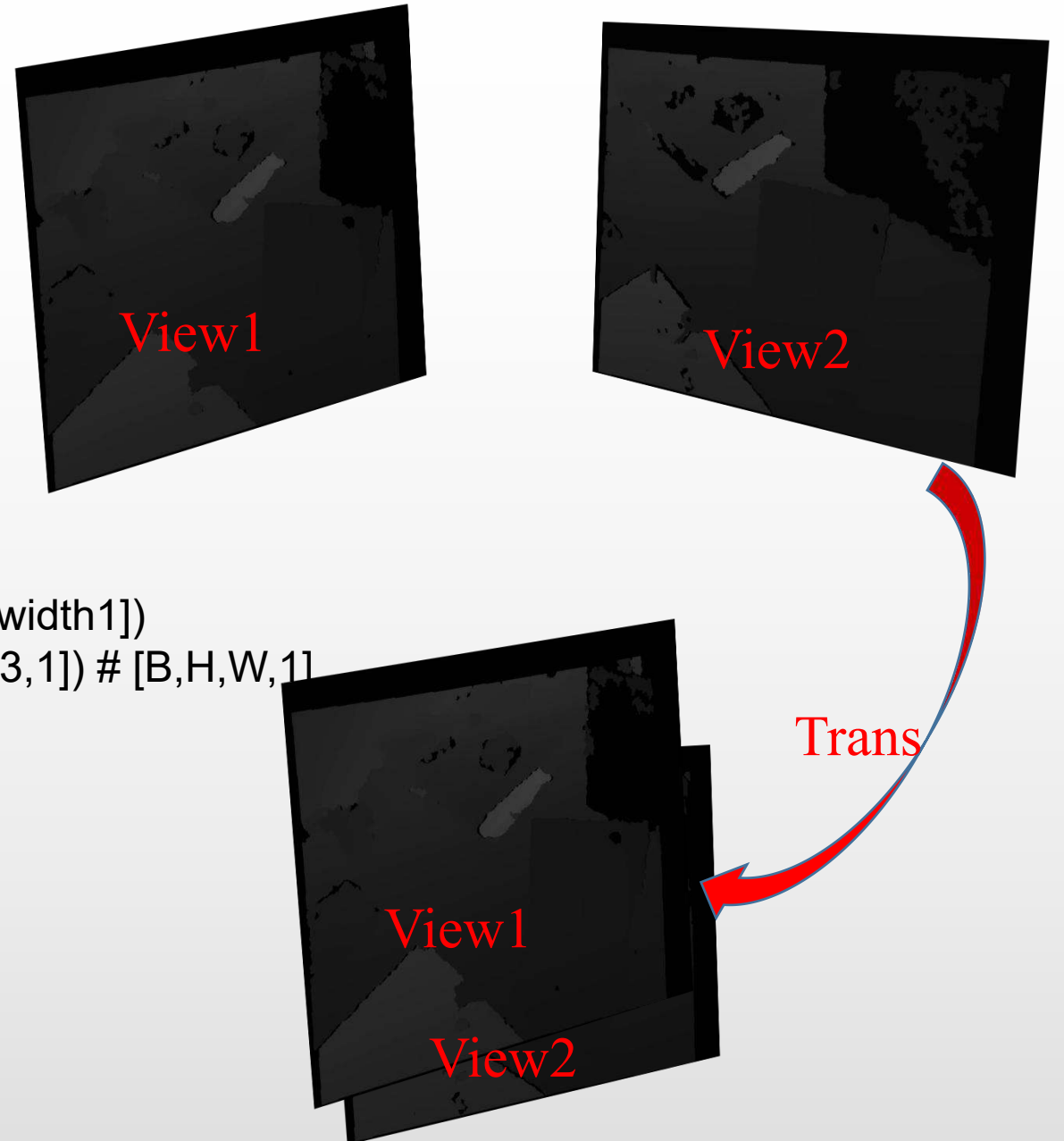
proj\_T = tf.matmul(K2\_4x4, c2Tc1s)

xyz2 = tf.matmul(proj\_T, XYZ1)

z2 = tf.slice(xyz2, [0,2,0], [-1,1,-1])

reproj\_depths = tf.reshape(z2, [batch\_size, 1, height1, width1])

reproj\_depths = tf.transpose(reproj\_depths, perm=[0,2,3,1]) # [B,H,W,1]



# Dataset

## inverse\_warp\_view\_2\_to\_1

```
(heatmaps2, depths2, depths1, c2Tc1s, K1, K2, inv_thetas1,  
thetas2, depth_thresh=0.5, get_warped_depth=False)
```

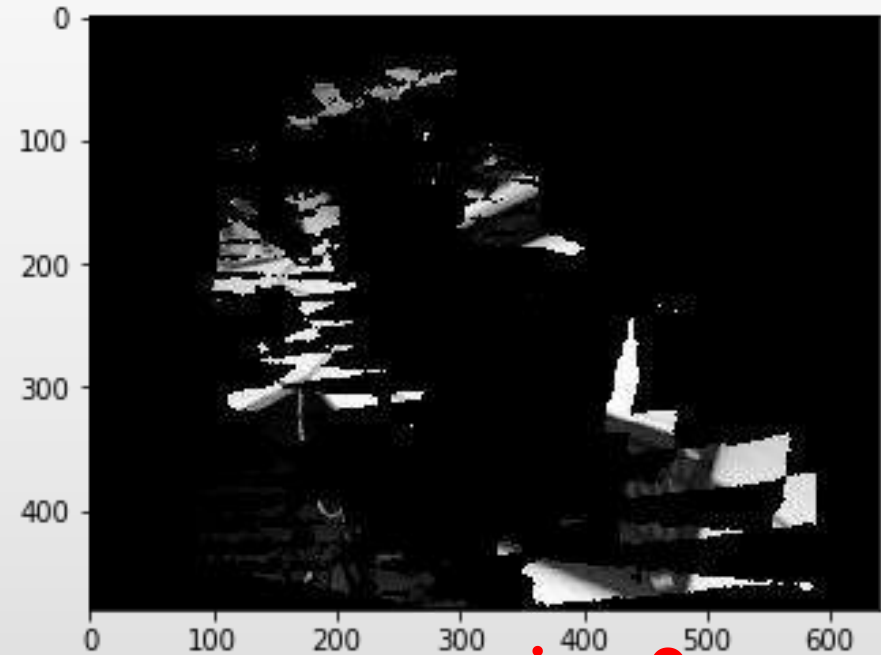
```
visible_masks = get_visibility_mask(xy_u2)
```

```
camfront_masks = tf.cast(tf.greater(reproj_depths, tf.zeros(), reproj_depths.dtype)), tf.float32)
```

```
nonocc_masks = tf.cast(tf.less(tf.squared_difference(depths1w, depths1), depth_thresh**2), tf.float32)
```

```
visible_masks = visible_masks * camfront_masks * nonocc_masks # take logical_and
```

```
heatmaps1w = heatmaps1w * visible_masks
```

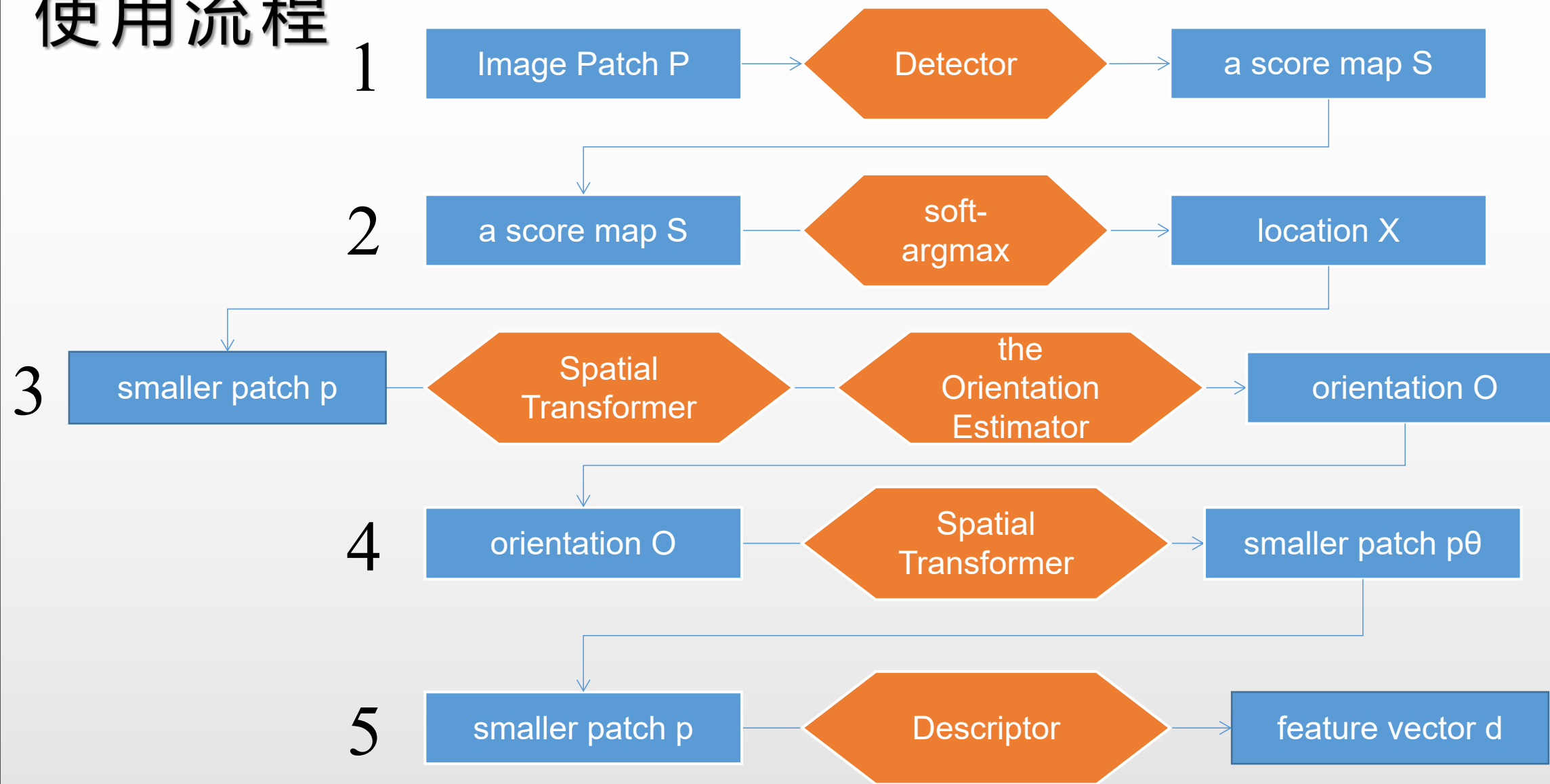


wrap\_view2

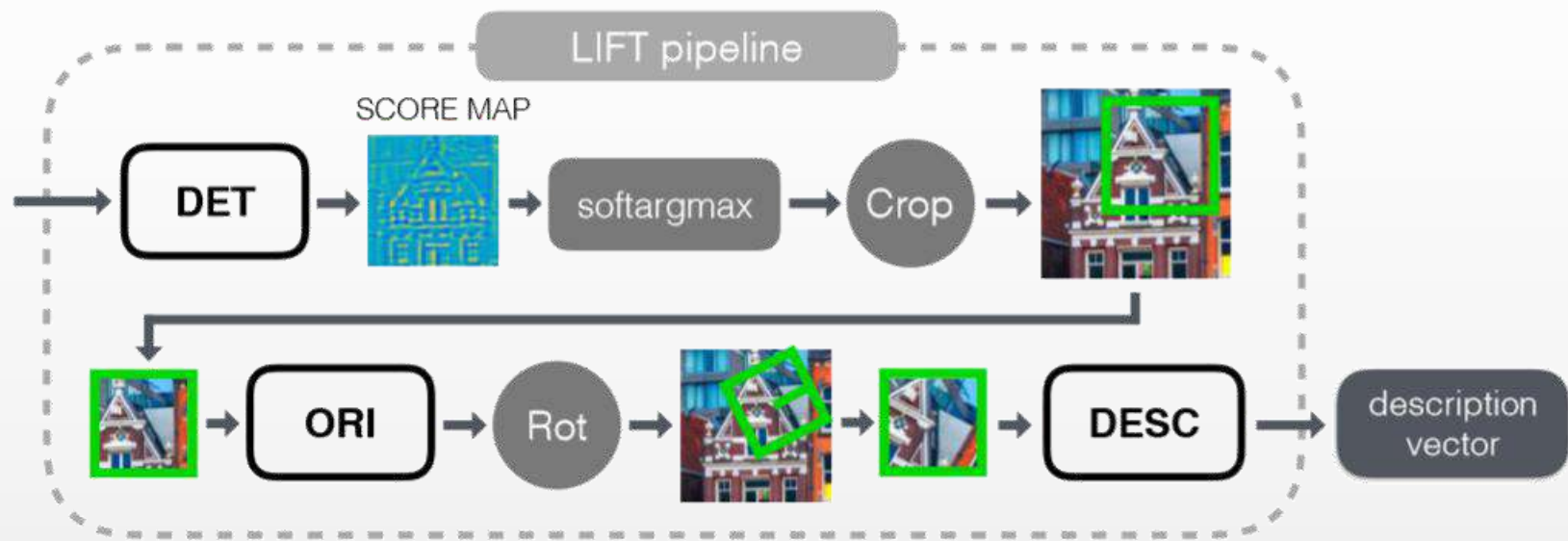
# LIFT

学习不变特征变换

# 使用流程

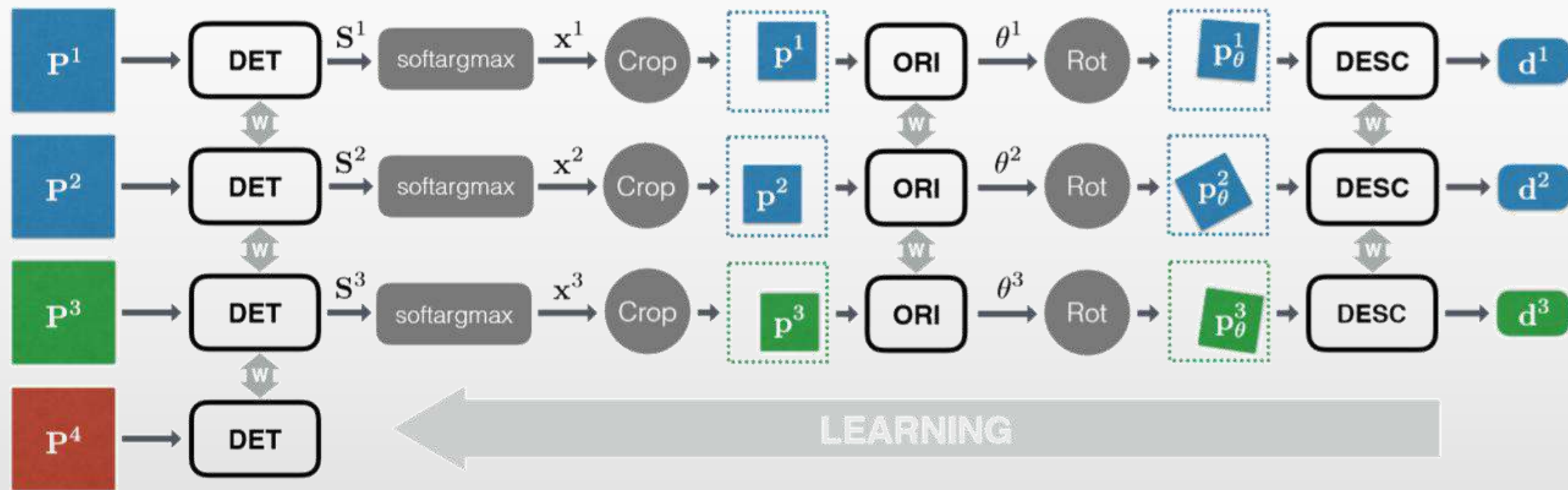


# 使用流程



# 训练流程

1. 建立Siamese网络;
2. 训练描述符;
3. 训练方向估计;
4. 训练特征点检测;



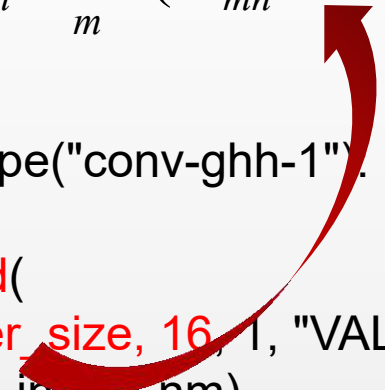
# 构建训练网络——Detector

网络取自《TILDE: A Temporally Invariant Learned DEtector》  
是一组滤波器，代码中使用的是窗口大小为25\*25的16组滤波器

$$S = f_{\mu}(P) = \sum_n^N \delta_n \max_m^M (W_{mn} * P + b_{mn})$$

with tf.variable\_scope("conv-ghh-1").

```
cur_in = conv_2d(  
    cur_in, kp_filter_size, 16, 1, "VALID")  
cur_in = ghh(cur_in, ns, nm)
```

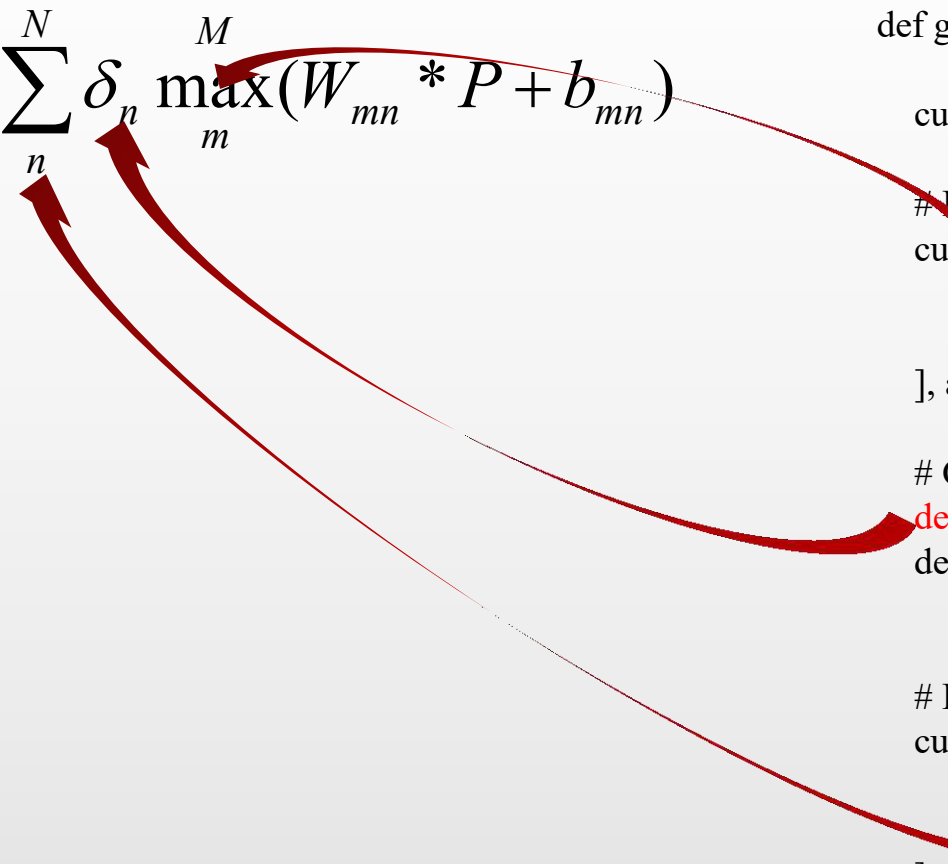




# 构建训练网络——Detector

网络取自《TILDE: A Temporally Invariant Learned DEtector》

卷积后加持一个分段线性激活函数

$$S = f_{\mu}(P) = \sum_n^N \delta_n \max_m^M (W_{mn} * P + b_{mn})$$


```
def ghg(inputs, num_in_sum, num_in_max, data_format="NHWC"):
```

```
    cur_in = inputs
```

```
    ... ..
```

```
    # Do max and concat them back
```

```
    cur_in = tf.concat([
        tf.reduce_max(cur_ins, axis=pool_axis, keep_dims=True) for
        cur_ins in cur_ins_to_max
    ], axis=pool_axis)
```

```
    # Create delta
```

```
    delta = (1.0 - 2.0 * (np.arange(num_in_sum) % 2)).astype("float32")
    delta = tf.reshape(delta, [1] * (len(inshp) - 1) + [num_in_sum])
```

```
    ... ..
```

```
    # Do delta multiplication, sum, and concat them back
```

```
    cur_in = tf.concat([
        tf.reduce_sum(cur_ins * delta, axis=pool_axis, keep_dims=True) for
        cur_ins in cur_ins_to_sum
    ], axis=pool_axis)
```

```
    return cur_in
```

# 构建训练网络——Detector

网络取自《TILDE: A Temporally Invariant Learned DEtector》

$$S = f_{\mu}(P) = \sum_n^N \delta_n \max_m^M (W_{mn} * P + b_{mn})$$

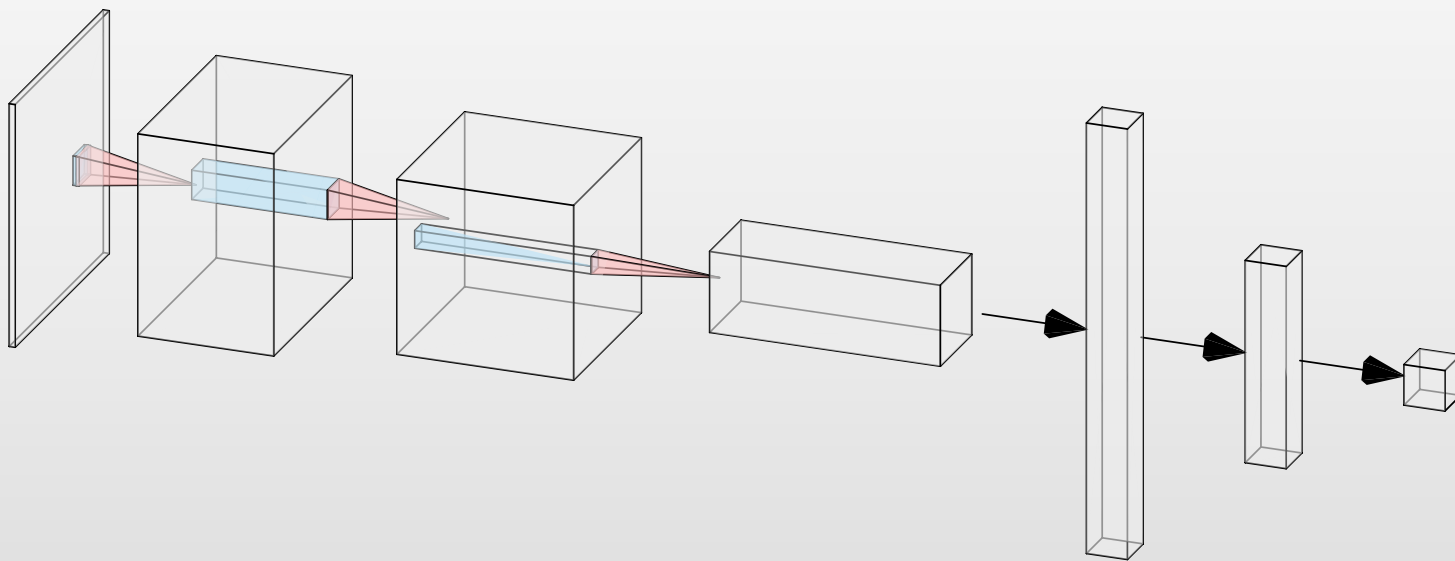
文中发觉使用softargmax去隐式地用scoremap中最大值表示关键点的位置比《LTILDE》中直接回归SFM中关键点位置好得多。

# 构建训练网络——Orientation Estimator

网络取自《Learning to Assign Orientations to Feature Points》

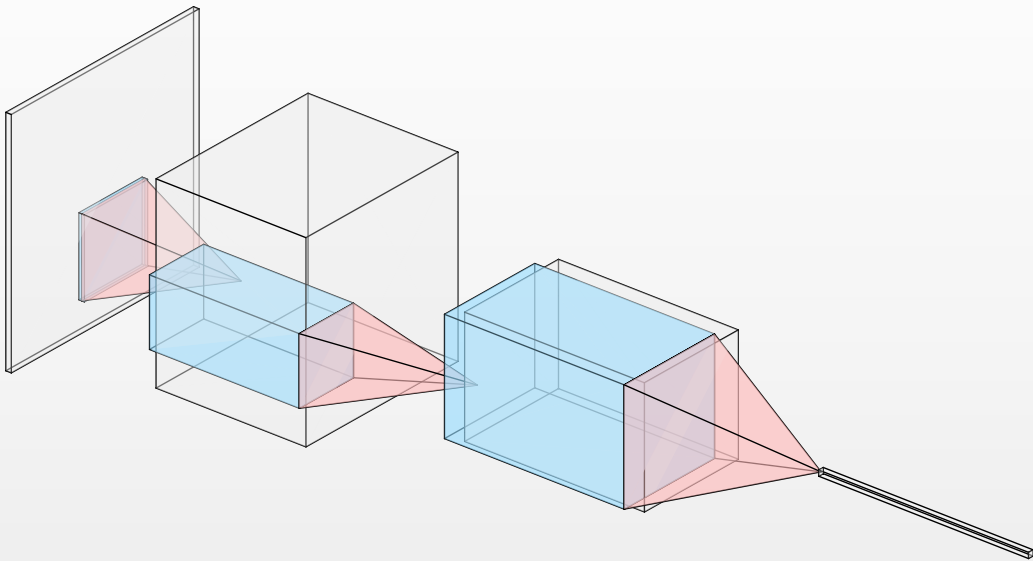
主要问题在于这篇文章 中 每隔5度旋转角就计算一个des然后去那这种数值上的近似去计算loss  
这个方法在本文这种pipeline中必然不可行，所以他们用了Spatial Transformer去近似，也是种学习的方法。

$$\zeta_{orientation}(P^1, x^1, P^2, x^2) = \| h_p(G(P^1, x^1)) - h_p(G(P^2, x^2)) \|_2$$



# 构建训练网络——Descriptor

网络取自《Discriminative Learning of Deep Convolutional Feature Point Descriptors》



Layer	1	2	3
Input size	$64 \times 64$	$29 \times 29$	$8 \times 8$
Filter size	$7 \times 7$	$6 \times 6$	$5 \times 5$
Output channels	32	64	128
Pooling & Norm.tion	$2 \times 2$	$3 \times 3$	$4 \times 4$
Nonlinearity	Tanh	Tanh	Tanh
Stride	2	3	4

$$\zeta_{desc}(p_{\theta}^k, p_{\theta}^l) = \begin{cases} \|h_{\rho}(p_{\theta}^k) - h_{\rho}(p_{\theta}^l)\|_2 & \text{Positive} \\ \max(0, \|h_{\rho}(p_{\theta}^k) - h_{\rho}(p_{\theta}^l)\|_2) & \text{Negative} \end{cases}$$