KeypointDetection笔记——HRNet

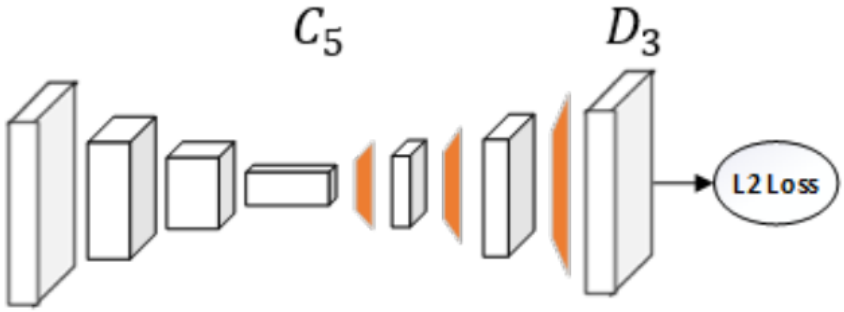
Deep High-Resolution Representation

- Paper: Deep High-Resolution Representation Learning for Human Pose Estimation
- Code: leoxiaobin/deep-high-resolution-net.pytorch

1. Introduction

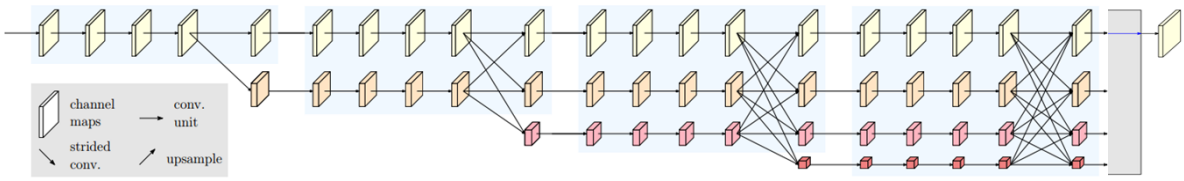
1.1 Why

- 现在存在的大多数方法，都是从图像的低分辨率特征恢复成高分辨率特征，这些低分辨率特征是由高分辨率到低分辨率的网络结构生成的。例如下图的Simple Baselines for Human Pose Estimation and Tracking，**由于下采样导致网络输出表征的有效空间分辨率损失，从而使得定位精度受限**；



1.2 What

- 在本文的网络中，整个过程中都保留着高分辨率的表示。
- 本文的网络设计，首先是一个高分辨率的子网络分支作为第一个stage，后面的stage会添加新的网络分支，同时新分支网络提取的特征的分辨率是逐渐减少的，并且每个分支网络之间是相互平行的。
- 通过重复执行multi-scale fusions（多尺寸融合），让多个不同分辨率的平行网络分支进行信息的交换，这样让输出的高分辨率特征包含更多的信息。因此提取出来的关键点热图，在空间上的分布会更准确（我理解为既有高分辨率低语义的信息，又融合了低分辨率高语义的信息）。
- 下图为HRNet的结构图，它是由多个平行，分辨率逐渐降低的子网络构成，平行的子网路通过**multi-scale fusion**进行信息交流。水平方向代表网络深度，垂直方向表示网络特征图的分辨率。最后在高分辨率特征上生成关键点热图。



1.3 How

- 该网络分辨率的降低，是使用平行的方式，而不是一连串的去降低特征分辨率，因此HRNet能**一直保持高分辨率特征**，而不需要一个分辨率由低到高的处理过程。因此预测出来的关键点热图，在空间上的分布会更加精准。

- 现在大多数特征融合的方式，是让深层特征图和浅层特征图进行融合，相反，本文**通过重复的多尺度融合，借助相同深度和相似级别的低分辨率表征，来提升高分辨率表征**的语义信息。很显然，结果是这种高分辨率表征对于位姿估计而言信息也很丰富。因此，本文预测的热图可能更准确。

2. Related Work

传统的单人位姿估计方法大多采用概率图模型或图形结构模型，最近通过深度学习，自动提取特征方式，相对于传统的算法，提升是比较明显的。现在深度学习提出的解决方式主要分为两类，分别为:

- 关键点位置回归，
- 估算关键点热图，

大多数网络都包含了一个主干网络，类似于分类网络一样，其降低了分辨率。以及另外一个主干网络，其产生与其输入具有相同分辨率的特征图。然后利用该特征去回归关键点或者估算热图。其主要是采用了分辨率 high-to-low 以及 low-to-high 的结构。可能增加多尺度融合和中间(深层)监督。

2.1 High-to-low and low-to-high

这个 high-to-low 的处理过程，主要是获得低分辨率高语义的特征，low-to-high 的处理过程，其主要是生成高分辨率的特征图。为了提高性能，这两个过程可能会重复几次。比较有代表的设计有以下几种：

- 对称的高到低和低到高过程，如Hourglass 把 low-to-high 与 high-to-low 的结构设计成镜像模式。
- 重量级 high-to-low 和 轻量级 low-to-high。其 high-to-low 的结构设计，主要是基于分类网络的主干网络，如 ResNet。low-to-high 的设计主要通过简单的双线性上采样或者转置卷积。
- 与反卷积结合。如ResNet或VGGNet的后两个阶段采用了扩张卷积来消除空间分辨率的损失。这是随后一个轻量级 low-to-high 过程，以进一步提高分辨率。避免频繁使用反卷积昂贵的计算成本。

2.2 Multi-scale fusion

直接的方法是将多分辨率图像分别输入多个网络，并聚合输出一个特征图，通过跳过连接将 high-to-low 流程中的低级特征，通过 low-to-high 流程得到高分辨率的高级特征图。在级联金字塔网络中，globalnet将高high-to-low 过程中的低到高层次特征，逐步组合为低到高过程，再由refinenet将卷积处理的低到高层次特征组合。我们的方法是重复多尺度融合，部分是受深度融合及其扩展的启发（该出翻译比较难，有兴趣的朋友可以多琢磨一下）

2.3 Intermediate supervision

中间监督或深度监督，早期发展为图像分类，帮助深度网络训练，提高热图估计质量。hourglass 方式和卷积方法处理中间热图作为输入，中间热图作为剩余子网的输入或输入的一部分。

2.4 This approach

本文的网络由与high-to-low相互平行的分支网络构成，其主要是使用高分辨率的分支网络进行姿态（heatmap）估算。它通过反复融合high-to-low 子网来产生可靠的高分辨率特征。该方法和现在存在的大多数方法（需要一个 low-to-high 上采样处理方式，聚合低级以及高级特征）不一样，同时我们的方法，不需要 intermediate heatmap supervision，在关键点检测精度和计算复杂度和参数效率方面具有优势。

3. Approach

人类姿态估算、关键点检测，其目的是为了输入从一张 W×H 图片，然后定位图中人类的关键点，如鼻子、眼睛、嘴巴等。目前好的办法是把关键点的预测问题，转化为K个大小为W'× H'的heatmaps={H1, H2, H3}进行计算。每个热图H*k*都表示第k个关键点的位置置信度。

我们也采用这种被广泛使用的方法，去预测人体的关键点，首先使用2个strided的卷积，减少输入图像的分辨率，获得初步特征图，然后把该特征图作为一个主体网络的输入，该主体网络的输出和输入的分辨率一样，其会其估算关键点的heatmaps 。

3.1 Sequential multi-resolution subnetworks

有的位姿估计网络是通过串联高分辨率子网来建立的，每个子网形成一个stage，由一系列卷积组成，并且在相邻的子网之间有一个下样本层来将分辨率减半。设Nsr为某网络在第s个stage层的子网络，r表示分辨率（其分辨率计算方式为1 / 2^{r-1}。如下面表示的是S=4个 stages 的网络结构：

$$\mathcal{N}_{11} \rightarrow \mathcal{N}_{22} \rightarrow \mathcal{N}_{33} \rightarrow \mathcal{N}_{44}. \qquad (1)$$

3.2 Parallel multi-resolution subnetworks

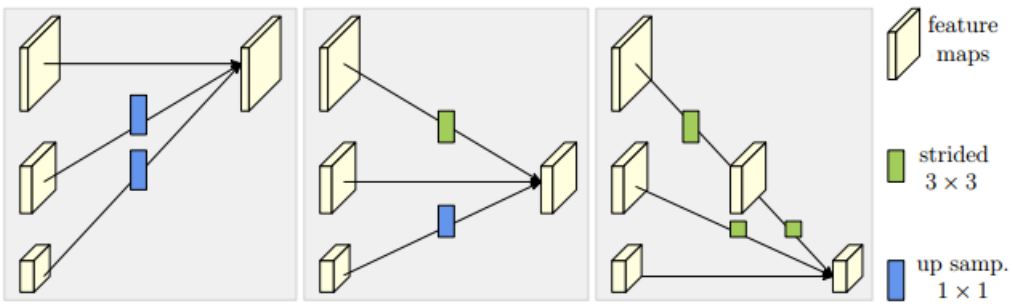
首先我们在第一个 stage 开始了一个高分辨率的网络分支，然后逐步增加高分辨率到低分辨率的子网路，形成一个新的 stages，并将多分辨率子网并行连接。因此，后一阶段并行子网的分辨率由前一阶段的分辨率和一个更低的分辨率组成，一个包含4个并行子网络的网络结构示例如下：

$$\begin{array}{ccccccc} \mathcal{N}_{11} & \rightarrow & \mathcal{N}_{21} & \rightarrow & \mathcal{N}_{31} & \rightarrow & \mathcal{N}_{41} \\ & \searrow & \mathcal{N}_{22} & \rightarrow & \mathcal{N}_{32} & \rightarrow & \mathcal{N}_{42} \\ & & & \searrow & \mathcal{N}_{33} & \rightarrow & \mathcal{N}_{43} \\ & & & & & \searrow & \mathcal{N}_{44} \end{array}$$

3.3 Repeated multi-scale fusion

我们引入了平行网络信息交换单元，比如每个子网络重复接受来自其他平行子网络的信息。下面是一个例子，展示了信息交换的方案。我们将第三 stage 分为几个(例如3个)交换模块，每个模块由3个并行卷积单元和一个跨并行单元的交换单元组成，其结构如下：

$$\begin{array}{ccccccccccc} \mathcal{C}_{31}^{1} & \searrow & & \nearrow & \mathcal{C}_{31}^{2} & \searrow & & \nearrow & \mathcal{C}_{31}^{3} & \searrow & \\ \mathcal{C}_{32}^{1} & \rightarrow & \mathcal{E}_{3}^{1} & \rightarrow & \mathcal{C}_{32}^{2} & \rightarrow & \mathcal{E}_{3}^{2} & \rightarrow & \mathcal{C}_{32}^{3} & \rightarrow & \mathcal{E}_{3}^{3} \\ \mathcal{C}_{33}^{1} & \nearrow & & \searrow & \mathcal{C}_{33}^{2} & \nearrow & & \searrow & \mathcal{C}_{33}^{3} & \nearrow & \end{array}$$



3.4 Heatmap estimation

我们通过最后最后一个交换单元，获得高分辨率的特征回归 heatmaps，经过实验对比，我们使用均方误差作为 loss，用于比较预测的 heatmaps 和 标注的 heatmaps。 groundtruth heatmps 是通过应用，以每个关键点的群真值位置为中心，采用标准差为1像素的二维高斯函数生成。

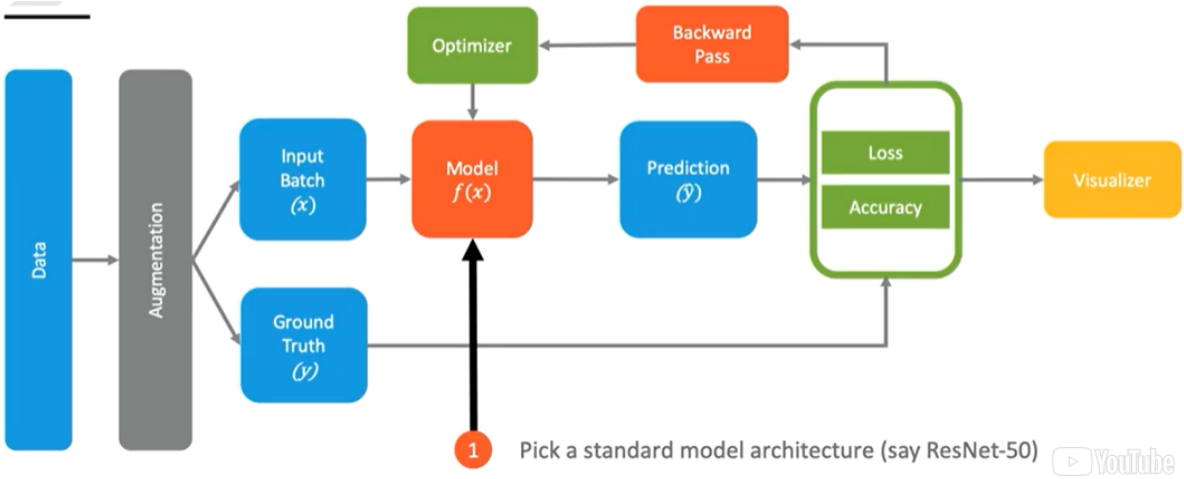
3.5 Network instantiation

我们设计实现了关键点热图估计网络，遵循的设计规则重新分配深度到每个阶段和通道的数量到每个分辨率。

其中的主体，也就是我们的 HRNet，由四个平行子网络共四个 stages 组成，在每个模块，如果其分辨率逐渐降低到一半，相应的宽度(通道数)增加到两倍。第一个 stages 包含4个剩余单元，其中每个单元与ResNet-50相同，是由一个通道为64的 bottleneck 形成。然后进行一个3×3的卷积，将feature maps 的通道数减小到 CCC, 第2、3、4阶段共有1、4、3个信息交换模块，相应的，一个信息交换模块包含4个剩余单元，其中每个单元在每个分辨率中包含两个3×3卷积以及跨分辨率的交换单元。综上所述，共有8个交换单位，即，进行了8次多尺度融合。
在我们的实验中，我们研究了一个小网和一个大网:HRNet-W32和HRNet-W48，其中32和48分别代表了高分辨率子网在最后三个阶段的宽度©。其他三个并行子网的宽度为：HRNet-W32=64、128、256，HRNet-W48=96,192,384。

4. Code

4.1 模型训练测试



4.1.1 tools/train.py

```
1.解析参数
2.构建网络模型
3.加载训练测试数据集迭代器
4.迭代训练
5.模型评估保存
```

- 模型

```
    # 根据配置文件构建网络
    # 两个模型：models.pose_hrnet和models.pose_resnet，用get_pose_net这个函数可以获得网络结构
    print('models.' + cfg.MODEL.NAME + '.get_pose_net')
    model = eval('models.'+cfg.MODEL.NAME+'.get_pose_net')(
        cfg, is_train=True
    )
```

print输出`models.pose_hrnet.get_pose_net`，此处就是构建网络的代码，会调用/lib/models/pose_hrnet.py的get_pose_net函数：

```
def get_pose_net(cfg, is_train, **kwargs):
    model = PoseHighResolutionNet(cfg, **kwargs)

    if is_train and cfg['MODEL']['INIT_WEIGHTS']:
        model.init_weights(cfg['MODEL']['PRETRAINED'])

    return model
```

- 数据

```
    # Data loading code
    # 对输入图像数据进行正则化处理
    normalize = transforms.Normalize(
        mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225]
    )

    # 创建训练以及测试数据的迭代器
    train_dataset = eval('dataset.'+cfg.DATASET.DATASET)(
        cfg, cfg.DATASET.ROOT, cfg.DATASET.TRAIN_SET, True,
        transforms.Compose([
            transforms.ToTensor(),
            normalize,
        ])
    )
    valid_dataset = eval('dataset.'+cfg.DATASET.DATASET)(
        cfg, cfg.DATASET.ROOT, cfg.DATASET.TEST_SET, False,
        transforms.Compose([
            transforms.ToTensor(),
            normalize,
        ])
    )

    train_loader = torch.utils.data.DataLoader(
        train_dataset,
        batch_size=cfg.TRAIN.BATCH_SIZE_PER_GPU*len(cfg.GPUS),
        shuffle=cfg.TRAIN.SHUFFLE,
        num_workers=cfg.WORKERS,
        pin_memory=cfg.PIN_MEMORY
    )
    valid_loader = torch.utils.data.DataLoader(
        valid_dataset,
        batch_size=cfg.TEST.BATCH_SIZE_PER_GPU*len(cfg.GPUS),
        shuffle=False,
        num_workers=cfg.WORKERS,
        pin_memory=cfg.PIN_MEMORY
```

```
    )
```

- 损失函数

```
    # 计算loss
    criterion = JointsMSELoss(
        use_target_weight=cfg.LOSS.USE_TARGET_WEIGHT
    ).cuda()
```

- 优化器

```
    # 优化器
    optimizer = get_optimizer(cfg, model)
```

```
    def get_optimizer(cfg, model):
        optimizer = None
        if cfg.TRAIN.OPTIMIZER == 'sgd':
            optimizer = optim.SGD(
                model.parameters(),
                lr=cfg.TRAIN.LR,
                momentum=cfg.TRAIN.MOMENTUM,
                weight_decay=cfg.TRAIN.WD,
                nesterov=cfg.TRAIN.NESTEROV
            )
        elif cfg.TRAIN.OPTIMIZER == 'adam':
            optimizer = optim.Adam(
                model.parameters(),
                lr=cfg.TRAIN.LR
            )

        return optimizer
```

- 迭代训练

```
    # 循环迭代进行训练
    for epoch in range(begin_epoch, cfg.TRAIN.END_EPOCH):
        lr_scheduler.step()

        # train for one epoch
        train(cfg, train_loader, model, criterion, optimizer, epoch,
              final_output_dir, tb_log_dir, writer_dict)

        # evaluate on validation set
        perf_indicator = validate(
            cfg, valid_loader, valid_dataset, model, criterion,
            final_output_dir, tb_log_dir, writer_dict
        )

        if perf_indicator >= best_perf:
            best_perf = perf_indicator
            best_model = True
        else:
            best_model = False
```

4.1.2 lib/dataset/coco.py

通过COCODataset的初始化函数，利用`_load_coco_keypoint_annotation_kernal`获得一个rec的数据，它包含了coco中所有人体以及对应关键点的信息。同时附带图片路径，以及标准化缩放比例等信息。

4.1.3 lib/dataset/JointsDataset.py

在计算loss的时候，需要的是heatmap，要根据rec中的信息，读取图片像素(用于训练)，同时把标签信息(人体关键点位置)转化为heatmap。

```
    def generate_target(self, joints, joints_vis):
        '''
        :param joints:  [num_joints, 3]
        :param joints_vis: [num_joints, 3]
        :return: target, target_weight(1: visible, 0: invisible)
        '''
        # target_weight形状为[17, 1]
        target_weight = np.ones((self.num_joints, 1), dtype=np.float32)
        target_weight[:, 0] = joints_vis[:, 0]

        # 检测制作热图的方式是否为gaussian，如果不是则报错
        assert self.target_type == 'gaussian', \
            'Only support gaussian map now!'

        # 如果使用高斯模糊的方法制作热图
        if self.target_type == 'gaussian':
            # 形状为[17, 64, 48]
            target = np.zeros((self.num_joints,
                               self.heatmap_size[1],
                               self.heatmap_size[0]),
                              dtype=np.float32)

            # self.sigma 默认为2, tmp_size=6
            tmp_size = self.sigma * 3

            # 为每个关键点生成热图target以及对应的热图权重target_weight
            for joint_id in range(self.num_joints):
                # 先计算出原图到输出热图的缩小倍数
                feat_stride = self.image_size / self.heatmap_size

                # 计算出输入原图的关键点，转换到热图的位置
                mu_x = int(joints[joint_id][0] / feat_stride[0] + 0.5)
                mu_y = int(joints[joint_id][1] / feat_stride[1] + 0.5)

                # Check that any part of the gaussian is in-bounds
                # 根据tmp_size参数，计算出关键点范围左上角和右下角坐标
                ul = [int(mu_x - tmp_size), int(mu_y - tmp_size)]
                br = [int(mu_x + tmp_size + 1), int(mu_y + tmp_size + 1)]

                # 判断该关键点是否处于热图之外，如果处于热图之外，则把该热图对应的
target_weight设置为0，然后continue
                if ul[0] >= self.heatmap_size[0] or ul[1] >= self.heatmap_size[1]
\
                        or br[0] < 0 or br[1] < 0:
```

```
                    # If not, just return the image as is
                    target_weight[joint_id] = 0
                    continue

                # # Generate gaussian
                # 产生高斯分布的大小
                size = 2 * tmp_size + 1
                # x[ 0.  1.  2.  3.  4.  5.  6.  7.  8.  9. 10. 11. 12.]
                x = np.arange(0, size, 1, np.float32)
                # y[[ 0.][ 1.][ 2.][ 3.][ 4.][ 5.][ 6.][ 7.][ 8.][ 9.][10.][11.]
[12.]]
                y = x[:, np.newaxis]
                # x0 = y0 = 6
                x0 = y0 = size // 2
                # The gaussian is not normalized, we want the center value to
equal 1
                # g形状[13, 13]，该数组中间的[7, 7]=1，离开该中心点越远数值越小
                g = np.exp(- ((x - x0) ** 2 + (y - y0) ** 2) / (2 * self.sigma **
2))

                # Usable gaussian range,
                # 判断边界,获得有效高斯分布的范围
                g_x = max(0, -ul[0]), min(br[0], self.heatmap_size[0]) - ul[0]
                g_y = max(0, -ul[1]), min(br[1], self.heatmap_size[1]) - ul[1]

                # Image range
                # 判断边界，获得有有效的图片像素边界
                img_x = max(0, ul[0]), min(br[0], self.heatmap_size[0])
                img_y = max(0, ul[1]), min(br[1], self.heatmap_size[1])

                # 如果该关键点对应的target_weight>0.5(即表示该关键点可见)，则把关键点附近
的特征点赋值成gaussian
                v = target_weight[joint_id]
                if v > 0.5:
                    target[joint_id][img_y[0]:img_y[1], img_x[0]:img_x[1]] =
g[g_y[0]:g_y[1], g_x[0]:g_x[1]]

        # 如果各个关键点训练权重不一样
        if self.use_different_joints_weight:
            target_weight = np.multiply(target_weight, self.joints_weight)

        # img = np.transpose(target.copy(), [1,2,0])*255
        # img = img[:,:,0].astype(np.uint8)
        # img = np.expand_dims(img,axis=-1)
        # cv2.imwrite('./test.jpg', img) # 关键点的热图
        return target, target_weight
```

4.2 模型总体结构

```
/lib/models/pose_hrnet.py：完整构建HRNet网络架构
包含：
1.基础框架：
    1）resnet网络的两种形式：
        class BasicBlock(nn.Module):
            def __init__(self,
```

```
                        inplanes,
                        planes,
                        stride=1,
                        downsample=None
                        ):
        class Bottleneck(nn.Module):
            def __init__(self,
                        inplanes,
                        planes,
                        stride=1,
                        downsample=None
                        ):
```

2）关键部分：

平行子网络信息多尺度融合模块：

```
            class HighResolutionModule(nn.Module)
                def __init__(self,
                             num_branches,
                             blocks,
                             num_blocks,
                             num_inchannels,
                             num_channels,
                             fuse_method,
                             multi_scale_output=True
                             ):
```

该模块的重要函数：

```
                # 搭建1个分支，单个分支内部分辨率相等，1个分支由num_blocks[branch_index]
个block组成，
                # 生成一个分支
                def _make_one_branch(self,
                                     branch_index,
                                     block, # block可以是两种ResNet模块中的一种；
                                     num_blocks,
                                     num_channels,
                                     stride=1
                                     ):
                # 循环调用_make_one_branch函数创建多个分支；
                def _make_branches(self,
                                   num_branches,
                                   block,
                                   num_blocks,
                                   num_channels
                                   ):
                # 融合模块
                def _make_fuse_layers(self):
```

2.关键点预测模块的完整网络：

```
    class PoseHighResolutionNet(nn.Module):
        def __init__(self,
                     cfg,
                     **kwargs
                     ):
```

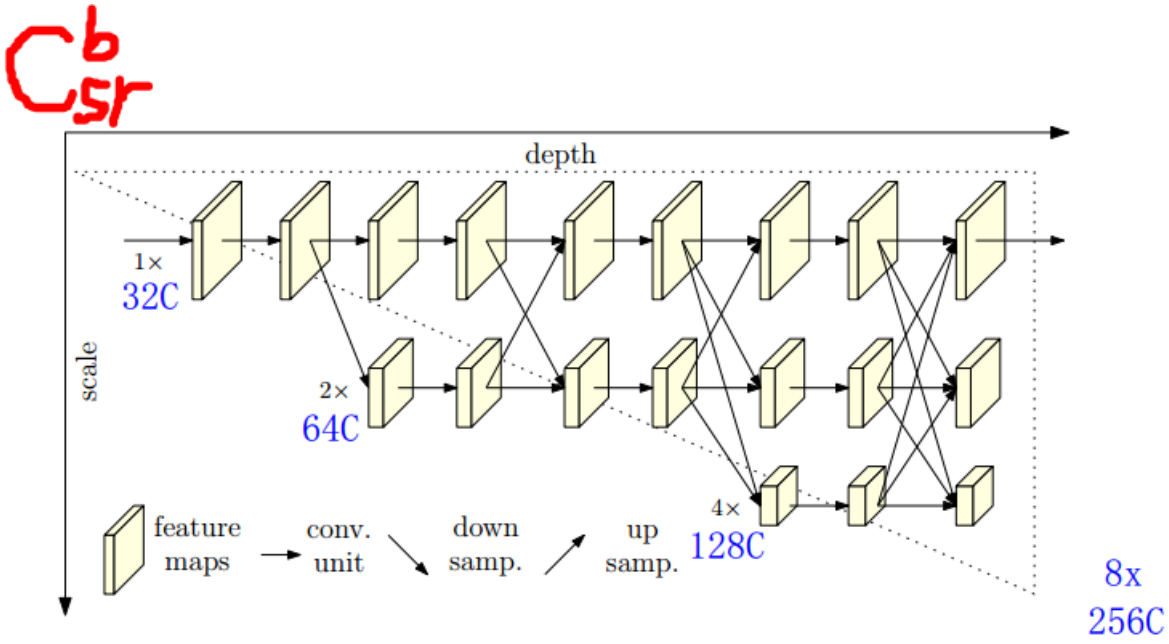
该模块的重要函数：

```
        # 创建新的平行子分支网络
        def _make_transition_layer(self,
                                   num_channels_pre_layer,
                                   num_channels_cur_layer
                                   ):
        # 创建BasicBlock或Bottleneck模块
```

```
        def _make_layer(self,
                        block,
                        planes,
                        blocks,
                        stride=1
                        ):
        # 构建论文中平行子网络信息交流的模块
        def _make_stage(self,
                        layer_config,
                        num_inchannels,
                        multi_scale_output=True
                        ):
```

4.2.1 PoseHighResolutionNet

- **网络结构**



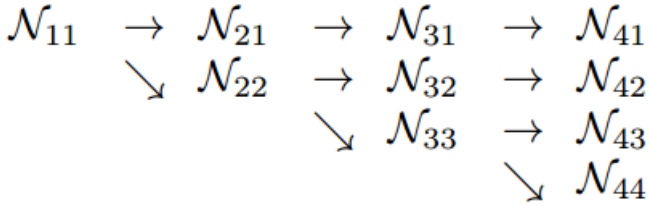

```
# 以w32_256x192_adam_lr1e-3.yaml为例
  EXTRA:
    PRETRAINED_LAYERS:
    - 'conv1'
    - 'bn1'
    - 'conv2'
    - 'bn2'
    - 'layer1'
    - 'transition1'
    - 'stage2'
    - 'transition2'
    - 'stage3'
    - 'transition3'
    - 'stage4'
```

```
STAGE2:
  NUM_MODULES: 1
  NUM_BRANCHES: 2
  BLOCK: BASIC
  NUM_BLOCKS:
  - 4
  - 4
  NUM_CHANNELS:
  - 32
  - 64
  FUSE_METHOD: SUM
```

```
STAGE3:
  NUM_MODULES: 4
  NUM_BRANCHES: 3
  BLOCK: BASIC
  NUM_BLOCKS:
  - 4
  - 4
  - 4
  NUM_CHANNELS:
  - 32
  - 64
  - 128
  FUSE_METHOD: SUM
```

```
STAGE4:
  NUM_MODULES: 3
  NUM_BRANCHES: 4
  BLOCK: BASIC
  NUM_BLOCKS:
  - 4
  - 4
  - 4
  - 4
  NUM_CHANNELS:
  - 32
  - 64
  - 128
  - 256
  FUSE_METHOD: SUM
```

以下的正向传播过程，主要对应论文中的如下过程：



- **正向传播**

```
def forward(self, x):
    # 对应论文中的stage1
```

```
        # # 经过一系列的卷积，获得初步特征图，总体过程为x[b, 3, 256, 192]-->x[b, 256,
64, 48]
        x = self.conv1(x)  # x[b,   3, 256, 192] --> x[b,  64, 128,  96]
        x = self.bn1(x)
        x = self.relu(x)
        x = self.conv2(x)  # x[b,  64, 128,  96] --> x[b,  64,  64,  48]
        x = self.bn2(x)
        x = self.relu(x)
        x = self.layer1(x) # x[b,  64,  64,  48] --> x[b, 256,  64,  48]


        # 对应论文中的stage2
        # 其中包含了创建分支的过程，即 N11-->N21,N22 这个过程
        # N22的分辨率为N21的二分之一，总体过程为:
        # x[b,256,64,48] --> y[b, 32, 64, 48]  因为通道数不一致，通过卷积进行通道数变换
        #                    y[b, 64, 32, 24]  通过新建平行分支生成
        x_list = []
        for i in range(self.stage2_cfg['NUM_BRANCHES']):
            if self.transition1[i] is not None:
                x_list.append(self.transition1[i](x))
            else:
                x_list.append(x)

        # 总体过程如下(经过一些卷积操作，但是特征图的分辨率和通道数都没有改变)：
        # x[b, 32, 64, 48] -->  y[b, 32, 64, 48]
        # x[b, 64, 32, 24] -->  y[b, 64, 32, 24]
        y_list = self.stage2(x_list)


        # 对应论文中的stage3
        # 其中包含了创建分支的过程，即 N22-->N32,N33 这个过程
        # N32的分辨率为N31的二分之一，
        # N33的分辨率为N32的二分之一，
        # y[b, 32, 64, 48] --> x[b, 32,  64, 48]   因为通道数一致，没有做任何操作
        # y[b, 64, 32, 24] --> x[b, 64,  32, 24]   因为通道数一致，没有做任何操作
        #                      x[b, 128, 16, 12]   通过新建平行分支生成
        x_list = []
        for i in range(self.stage3_cfg['NUM_BRANCHES']):
            if self.transition2[i] is not None:
                x_list.append(self.transition2[i](y_list[-1]))
            else:
                x_list.append(y_list[i])

        # 总体过程如下(经过一些卷积操作,但是特征图的分辨率和通道数都没有改变)：
        # x[b, 32, 64, 48] --> x[b, 32, 64, 48]
        # x[b, 32, 32, 24] --> x[b, 32, 32, 24]
        # x[b, 64, 16, 12] --> x[b, 64, 16, 12]
        y_list = self.stage3(x_list)


        # 对应论文中的stage4
        # 其中包含了创建分支的过程，即 N33-->N43,N44 这个过程
        # N42的分辨率为N41的二分之一
        # N43的分辨率为N42的二分之一
        # N44的分辨率为N43的二分之一
        # y[b,  32,  64,  48] --> x[b,  32,  64,  48]  因为通道数一致，没有做任何操作
        # y[b,  64,  32,  24] --> x[b,  64,  32,  24]  因为通道数一致，没有做任何操作
        # y[b, 128,  16,  12] --> x[b, 128,  16,  12]  因为通道数一致，没有做任何操作
```

```
        #                           x[b, 256,   8,   6]  通过新建平行分支生成
        x_list = []
        for i in range(self.stage4_cfg['NUM_BRANCHES']):
            if self.transition3[i] is not None:
                x_list.append(self.transition3[i](y_list[-1]))
            else:
                x_list.append(y_list[i])

        # 进行多尺度特征融合
        # x[b,  32,  64,  48] -->
        # x[b,  64,  32,  24] -->
        # x[b, 128,  16,  12] -->
        # x[b, 256,   8,   6] --> y[b,  32,  64,  48]
        y_list = self.stage4(x_list)

        # y[b,  32,  64,  48] --> x[b,  17,  64,  48]
        x = self.final_layer(y_list[0])

        return x
```

通过代码可以很明显地看到，最终获得了一个大小为[b, 17, 64, 48]的**heatmap**，这就是最终想要的结果，上图中的每个Nsr可以分成两个重要的模块，分别为`self.transition`以及`self.stage`，它们在初始化函数`def __init__(self, cfg, **kwargs):`中构建。

- **模型初始化**

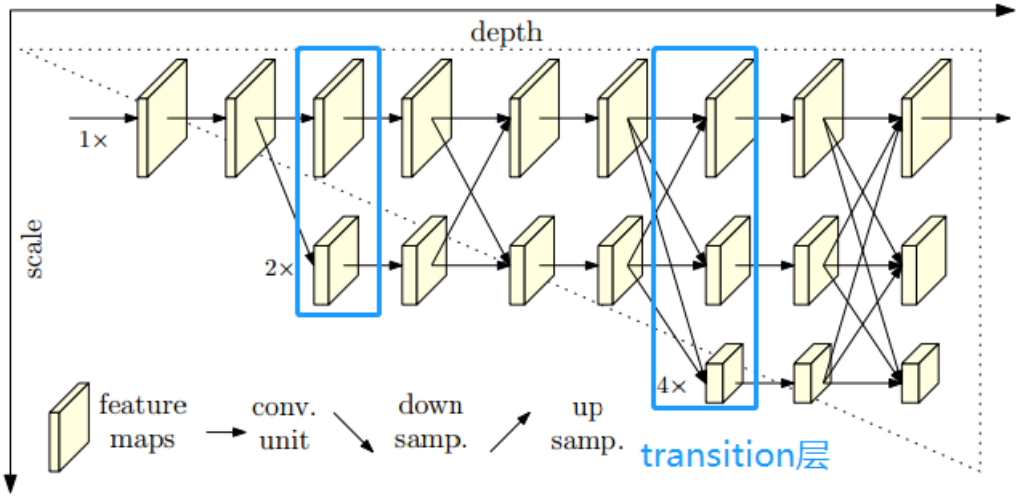
```
class PoseHighResolutionNet(nn.Module):
    def __init__(self, cfg, **kwargs):
        self.inplanes = 64
        extra = cfg['MODEL']['EXTRA'] # _C.MODEL.EXTRA = CN(new_allowed=True), 
whether adding new key is allowed when merging with other configs.
        super(PoseHighResolutionNet, self).__init__()


        # stem net
        # 进行一系列的卷积操作，获得最初始的特征图N11
        self.conv1 = nn.Conv2d(3, 64, kernel_size=3, stride=2, padding=1,
                               bias=False)
        self.bn1 = nn.BatchNorm2d(64, momentum=BN_MOMENTUM)
        self.conv2 = nn.Conv2d(64, 64, kernel_size=3, stride=2, padding=1,
                               bias=False)
        self.bn2 = nn.BatchNorm2d(64, momentum=BN_MOMENTUM)
        self.relu = nn.ReLU(inplace=True)
        self.layer1 = self._make_layer(Bottleneck, 64, 4)


        # 获取stage2的相关配置信息
        self.stage2_cfg = extra['STAGE2']
        # num_channels=[32, 64]，表示输出通道；
        # 32是高分辨率平行分支N21的输出通道数；
        # 64是新建平行分支N22的输出通道数；
        num_channels = self.stage2_cfg['NUM_CHANNELS']
        # 这里的block为BASIC
        block = blocks_dict[self.stage2_cfg['BLOCK']]
        # block.expansion默认为1，num_channels表示输出通道[32, 64]
        num_channels = [
            num_channels[i] * block.expansion for i in range(len(num_channels))
```

```
        ]
        # 这里会生成新的平行分支N2网络，即N11-->N21,N22这个过程
        # 同时会对输入的特征图x进行通道变换(如果输入输出通道数不一致)
        self.transition1 = self._make_transition_layer([256], num_channels)
        # 对平行子网络进行加工，让其输出的y，可以当作下一个stage的输入x，
        # 这里的pre_stage_channels为当前stage的输出通道数，也就是下一个stage的输入通道数
        # 同时平行子网络信息交换模块也包含再其中
        self.stage2, pre_stage_channels = self._make_stage(
            self.stage2_cfg, num_channels)


        # 获取stage3的相关配置信息
        self.stage3_cfg = extra['STAGE3']
        # num_channels=[32, 64, 128]，表示输出通道；
        # 32是高分辨率平行分支N31的输出通道数；
        # 64是平行分支N32的输出通道数；
        # 128是新建平行分支N33的输出通道数
        num_channels = self.stage3_cfg['NUM_CHANNELS']
        # 这里的block为BasicBlock
        block = blocks_dict[self.stage3_cfg['BLOCK']]
        # block.expansion默认为1，num_channels表示输出通道[32, 64, 128]
        num_channels = [
            num_channels[i] * block.expansion for i in range(len(num_channels))
        ]
        # 这里会生成新的平行分支N3网络，即N22-->N32,N33这个过程
        # 同时会对输入的特征图x进行通道变换(如果输入输出通道数不一致)
        self.transition2 = self._make_transition_layer(
            pre_stage_channels, num_channels)
        # 对平行子网络进行加工，让其输出的y，可以当作下一个stage的输入x，
        # 这里的pre_stage_channels为当前stage的输出通道数，也就是下一个stage的输入通道数
        # 同时平行子网络信息交换模块，也包含再其中
        self.stage3, pre_stage_channels = self._make_stage(
            self.stage3_cfg, num_channels)


        # 获取stage4的相关配置信息
        self.stage4_cfg = extra['STAGE4']
        # num_channels=[32, 64, 128, 256]，表示输出通道；
        # 32是高分辨率平行分支N41的输出通道数；
        # 64是平行分支N42的输出通道数；
        # 128是平行分支N43的输出通道数；
        # 256是新建平行分支N44的输出通道数；
        num_channels = self.stage4_cfg['NUM_CHANNELS']
        # 这里的block为BasicBlock；
        block = blocks_dict[self.stage4_cfg['BLOCK']]
        # block.expansion默认为1，num_channels表示输出通道[32, 64, 128, 256]
        num_channels = [
            num_channels[i] * block.expansion for i in range(len(num_channels))
        ]
        # 这里会生成新的平行分支N4网络，即N33-->N43,N44这个过程
        # 同时会对输入的特征图x进行通道变换(如果输入输出通道数不一致)
        self.transition3 = self._make_transition_layer(
            pre_stage_channels, num_channels)
        # 对平行子网络进行加工，让其输出的y，可以当作下一个stage的输入x，
        # 这里的pre_stage_channels为当前stage的输出通道数，也就是下一个stage的输入通道数
        # 同时平行子网络信息交换模块，也包含再其中
        self.stage4, pre_stage_channels = self._make_stage(
            self.stage4_cfg, num_channels, multi_scale_output=False)
```

```
        # 对最终的特征图混合之后进行一次卷积，预测人体关键点的heatmap
        self.final_layer = nn.Conv2d(
            in_channels=pre_stage_channels[0],
            out_channels=cfg['MODEL']['NUM_JOINTS'], # 17个关键点，通道数为17层，每一层对应1个关键点的heatmap
            kernel_size=extra['FINAL_CONV_KERNEL'],
            stride=1,
            padding=1 if extra['FINAL_CONV_KERNEL'] == 3 else 0
        )

        self.pretrained_layers = extra['PRETRAINED_LAYERS']
```

初始化函数`def __init__(self, cfg, **kwargs):`的构建过程看起来较复杂，其实主要调用了如下两个函数：

- `_make_transition_layer`主要是创建新的平行子分支网络；



- `_make_stage`是为了构建论文中平行子网络信息交流的模块；

```
    def _make_transition_layer(self, num_channels_pre_layer, num_channels_cur_layer):
        """
        :param num_channels_pre_layer: 上一个stage平行网络的输出通道数目，为一个list,
            stage=2时, num_channels_pre_layer=[256]
            stage=3时, num_channels_pre_layer=[32,64]
            stage=4时, num_channels_pre_layer=[32,64,128]
        :param num_channels_cur_layer: 当前stage平行网络的输出通道数目，为一个list,
            stage=2时, num_channels_cur_layer = [32,64]
            stage=3时, num_channels_cur_layer = [32,64,128]
            stage=4时, num_channels_cur_layer = [32,64,128,256]
        """
        num_branches_cur = len(num_channels_cur_layer)
        num_branches_pre = len(num_channels_pre_layer)

        transition_layers = []
        # 对stage的每个分支进行处理
        for i in range(num_branches_cur):
            # 如果不是最后一个分支
            if i < num_branches_pre:
                # 如果当前层的输入通道和输出通道数不相等，则通过卷积对通道数进行变换
                if num_channels_cur_layer[i] != num_channels_pre_layer[i]:
```

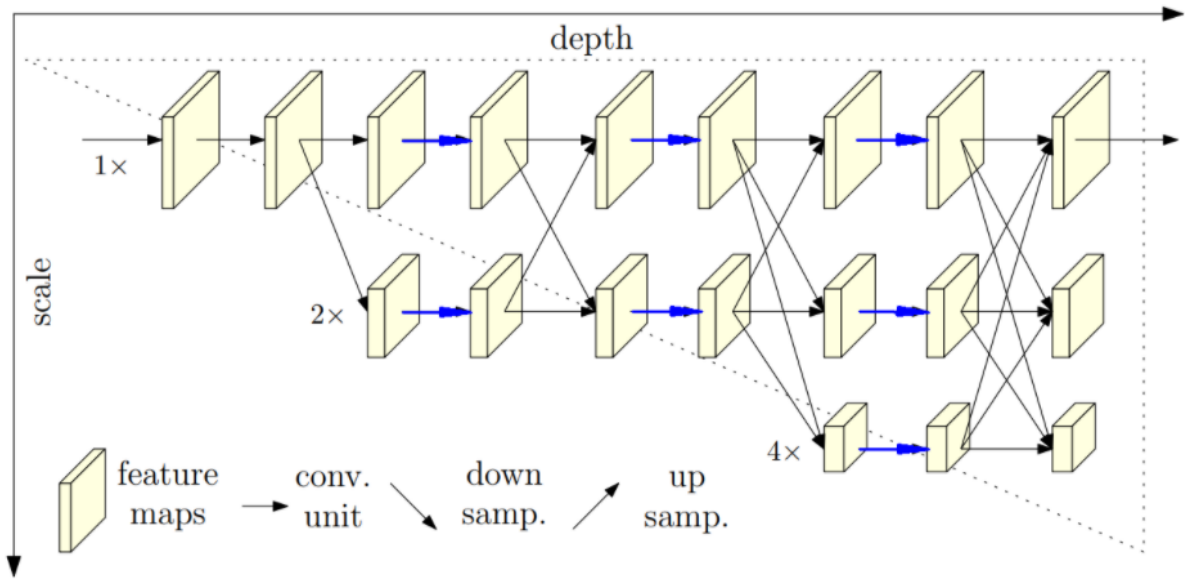
```
                    transition_layers.append(
                        nn.Sequential(
                            nn.Conv2d(
                                num_channels_pre_layer[i],
                                num_channels_cur_layer[i],
                                3, 1, 1, bias=False
                            ),
                            nn.BatchNorm2d(num_channels_cur_layer[i]),
                            nn.ReLU(inplace=True)
                        )
                    )
                # 如果当前层的输入通道和输出通道数相等，则什么都不做
                else:
                    transition_layers.append(None)

            # 如果是最后一个分支，则再新建一个分支（该分支分辨率会减少一半）
            else:
                conv3x3s = []
                for j in range(i+1-num_branches_pre):
                    inchannels = num_channels_pre_layer[-1]
                    outchannels = num_channels_cur_layer[i] \
                        if j == i-num_branches_pre else inchannels
                    conv3x3s.append(
                        nn.Sequential(
                            nn.Conv2d(
                                inchannels, outchannels, 3, 2, 1, bias=False
                            ),
                            nn.BatchNorm2d(outchannels),
                            nn.ReLU(inplace=True)
                        )
                    )
                transition_layers.append(nn.Sequential(*conv3x3s))

        return nn.ModuleList(transition_layers)
```

```
    def _make_stage(self, layer_config, num_inchannels,
multi_scale_output=True):
        """
        当stage=2时：  num_inchannels=[32,64]           multi_scale_output=Ture
        当stage=3时：  num_inchannels=[32,64,128]       multi_scale_output=Ture
        当stage=4时：  num_inchannels=[32,64,128,256]   multi_scale_output=False
        """
        # 当stage=2,3,4时，num_modules分别为：1,4,3
        # 表示HighResolutionModule（平行之网络交换信息模块）模块的数目
        num_modules = layer_config['NUM_MODULES']
        # 当stage=2,3,4时，num_branches分别为：2,3,4
        # 表示每个stage平行网络的数目
        num_branches = layer_config['NUM_BRANCHES']
        # 当stage=2,3,4时，num_blocks分别为：[4,4], [4,4,4], [4,4,4,4]
        # 表示每个stage blocks(BasicBlock或者Bottleneck)的数目
        num_blocks = layer_config['NUM_BLOCKS']
        # 当stage=2,3,4时，num_channels分别为：[32,64], [32,64,128],
[32,64,128,256]
        num_channels = layer_config['NUM_CHANNELS']
        # 当stage=2,3,4时，block分别为：BasicBlock,BasicBlock,BasicBlock
        block = blocks_dict[layer_config['BLOCK']]
        # 当stage=2,3,4时，都为SUM，表示特征融合的方式
```

```
        fuse_method = layer_config['FUSE_METHOD']

        modules = []
        # 根据num_modules的数目创建HighResolutionModule
        for i in range(num_modules):
            # multi_scale_output is only used last module
            # multi_scale_output 只被用在最后一个HighResolutionModule
            if not multi_scale_output and i == num_modules - 1:
                reset_multi_scale_output = False
            else:
                reset_multi_scale_output = True

            # 根据参数，添加HighResolutionModule
            modules.append(
                HighResolutionModule(
                    num_branches,
                    block,
                    num_blocks,
                    num_inchannels,
                    num_channels,
                    fuse_method,
                    reset_multi_scale_output
                )
            )
            # 获得最后一个HighResolutionModule的输出通道数
            num_inchannels = modules[-1].get_num_inchannels()

        return nn.Sequential(*modules), num_inchannels
```

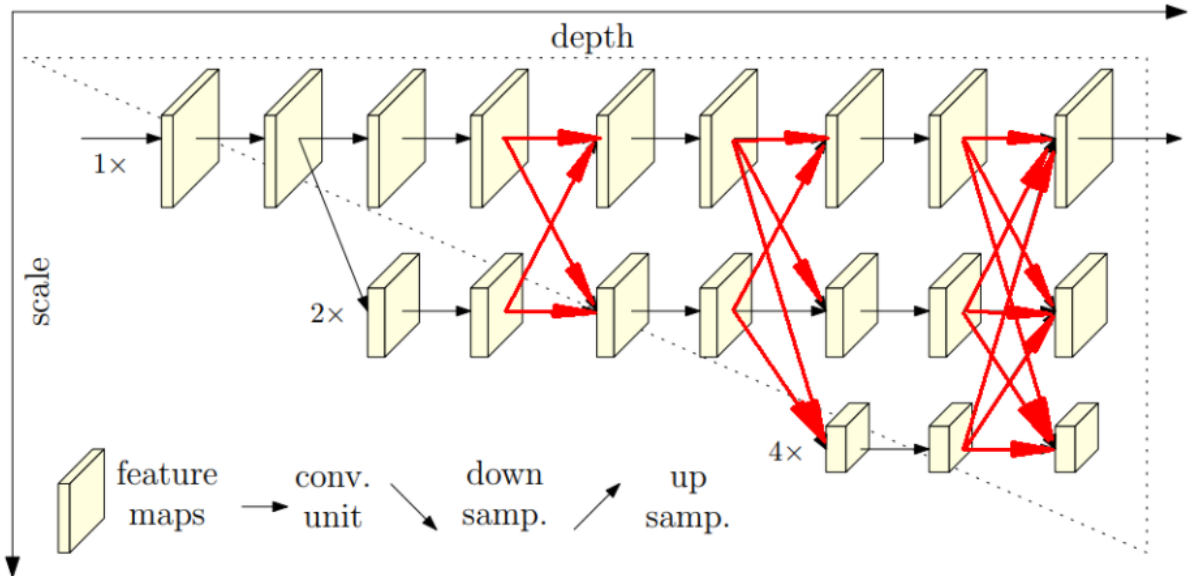
4.2.2 HighResolutionModule

- 前向传播forward中的沿depth方向的直线传播部分（下图蓝色箭头）：



```
        # 当前有多少个网络分支，则有多少个x当作输入
        # 当stage=2：x=[b,32,64,48],[b,64,32,24]
        #           -->[b,32,64,48],[b,64,32,24]
        # 当stage=3：x=[b,32,64,48],[b,64,32,24],[b,128,16,12]
        #           -->[b,32,64,48],[b,64,32,24],[b,128,16,12]
        # 当stage=4：x=[b,32,64,48],[b,64,32,24],[b,128,16,12],[b,256,8,6]
        #           -->[b,32,64,48],[b,64,32,24],[b,128,16,12],[b,256,8,6]
        # 简单的说，该处就是对每个分支进行了BasicBlock或者Bottleneck操作
        for i in range(self.num_branches):
            x[i] = self.branches[i](x[i])
```

- 前向传播forward中的多分辨率分支的多尺度融合部分（下图红色箭头）：



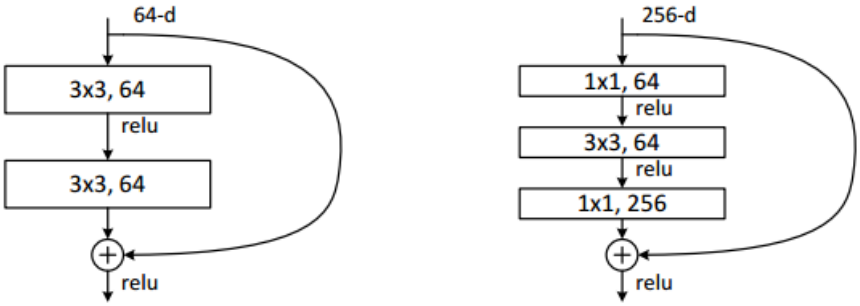

```
        x_fuse = []
        # 对每个分支进行融合(信息交流)
        for i in range(len(self.fuse_layers)):
            # 循环融合多个分支的输出信息，当作输入，进行下一轮融合
            y = x[0] if i == 0 else self.fuse_layers[i][0](x[0])
            for j in range(1, self.num_branches):
                if i == j:
                    y = y + x[j]
                else:
                    y = y + self.fuse_layers[i][j](x[j])
            x_fuse.append(self.relu(y))
```

- 平行子网络信息多尺度融合模块的初始化

```
class HighResolutionModule(nn.Module):
    def __init__(self, num_branches, blocks, num_blocks, num_inchannels,
                 num_channels, fuse_method, multi_scale_output=True):
        """
        :param num_branches: 当前 stage 分支平行子网络的数目
        :param blocks: BasicBlock或者Bottleneck
        :param num_blocks: BasicBlock或者Bottleneck的数目

        :param num_inchannels: 输入通道数目
                    当stage = 2时：  num_inchannels = [32, 64]
                    当stage = 3时：  num_inchannels = [32, 64, 128]
                    当stage = 4时：  num_inchannels = [32, 64, 128, 256]
```

```
        :param num_channels: 输出通道数目
                    当stage = 2时：  num_inchannels = [32, 64]
                    当stage = 3时：  num_inchannels = [32, 64, 128]
                    当stage = 4时：  num_inchannels = [32, 64, 128, 256]

        :param fuse_method: 默认SUM
        :param multi_scale_output:
                    当stage = 2时：  multi_scale_output=True
                    当stage = 3时：  multi_scale_output=True
                    当stage = 4时：  multi_scale_output=False
        """
        super(HighResolutionModule, self).__init__()

        # 对输入的一些参数进行检测
        self._check_branches(
            num_branches, blocks, num_blocks, num_inchannels, num_channels)

        # 上面有详细介绍
        self.num_inchannels = num_inchannels
        self.fuse_method = fuse_method
        self.num_branches = num_branches
        self.multi_scale_output = multi_scale_output

        # 为每个分支构建分支网络
        # 当stage=2,3,4时，num_branches分别为：2,3,4，表示每个stage平行网络的数目
        # 当stage=2,3,4时，num_blocks分别为：[4,4], [4,4,4], [4,4,4,4]，表示每个stage
的每个平行分支BasicBlock或者Bottleneck的数目
        self.branches = self._make_branches(
            num_branches, blocks, num_blocks, num_channels)
        # 创建一个多尺度融合层，当stage=2,3,4时，len(self.fuse_layers)分别为2,3,4；
        # 其与num_branches在每个stage的数目是一致的
        self.fuse_layers = self._make_fuse_layers()
        self.relu = nn.ReLU(True)
```

4.2.3 ResNet



layer name	output size	18-layer	34-layer	50-layer	101-layer	152-layer
conv1	112×112	7×7, 64, stride 2				
conv2_x	56×56	3×3 max pool, stride 2				
		$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$
conv3_x	28×28	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times8$
conv4_x	14×14	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times23$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times36$
conv5_x	7×7	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$
	1×1	average pool, 1000-d fc, softmax				
FLOPs		1.8×10^9	3.6×10^9	3.8×10^9	7.6×10^9	11.3×10^9

```
resnet_spec = {
    18: (BasicBlock, [2, 2, 2, 2]),
    34: (BasicBlock, [3, 4, 6, 3]),
    50: (Bottleneck, [3, 4, 6, 3]),
    101: (Bottleneck, [3, 4, 23, 3]),
    152: (Bottleneck, [3, 8, 36, 3])
}
```

根据Block类型，可以将这五种ResNet分为两类：

- (1) 一种基于BasicBlock，浅层网络ResNet18、34，都由BasicBlock搭成；
- (2) 另一种基于Bottleneck，深层网络ResNet50、101、152乃至更深的网络，都由Bottleneck搭成。

Block相当于积木，每个layer都由若干Block搭建而成，再由layer组成整个网络。每种ResNet都是4个layer（不算一开始的7×7卷积层和3×3maxpooling层），如上图，conv2_x对应layer1，conv3_x对应layer2，conv4_x对应layer3，conv5_x对应layer4。

5. 心得

5.1 Loss函数

- 如果给不可见关键点在loss中的权重设得太高，那么FP会减少，同时TP可能也会减少，因此AP可能会减少、AR可能会减少；
- 如果给不可见关键点在loss中的权重设得很低，甚至为0，那么TP会增多，同时FP也会增多，因此AP可能会减少、AR会升高；

参考资料

1. Overview of Human Pose Estimation Neural Networks — HRNet + HigherHRNet, Architectures and FAQ — 2d3d.ai : https://towardsdatascience.com/overview-of-human-pose-estimation-neural-networks-hrnet-higherhrnet-architectures-and-faq-1954b2f8b249
2. 姿态估计1-00：HR-Net(人体姿态估算)-目录-史上最新无死角讲解: https://blog.csdn.net/weixin_43013761/article/details/106621525
3. 【论文阅读笔记】HRNet--从代码来看论文: https://blog.csdn.net/weixin_38715903/article/details/101629781
4. HRNet阅读笔记及代码理解: https://segmentfault.com/a/1190000019167646

5. TensorBoard Graph with objects other than torch.nn.Module can not be visualized: https://github.com/pytorch/pytorch/issues/30459
6. Numerical Coordinate Regression=高斯热图 VS 坐标回归：https://zhuanlan.zhihu.com/p/53057942

问雪更新于2020-11-13