Mask R-CNN

简介

💂 详细介绍Mask-RCNN 的模型结构和代码

论文地址 : https://arxiv.org/abs/1703.06870

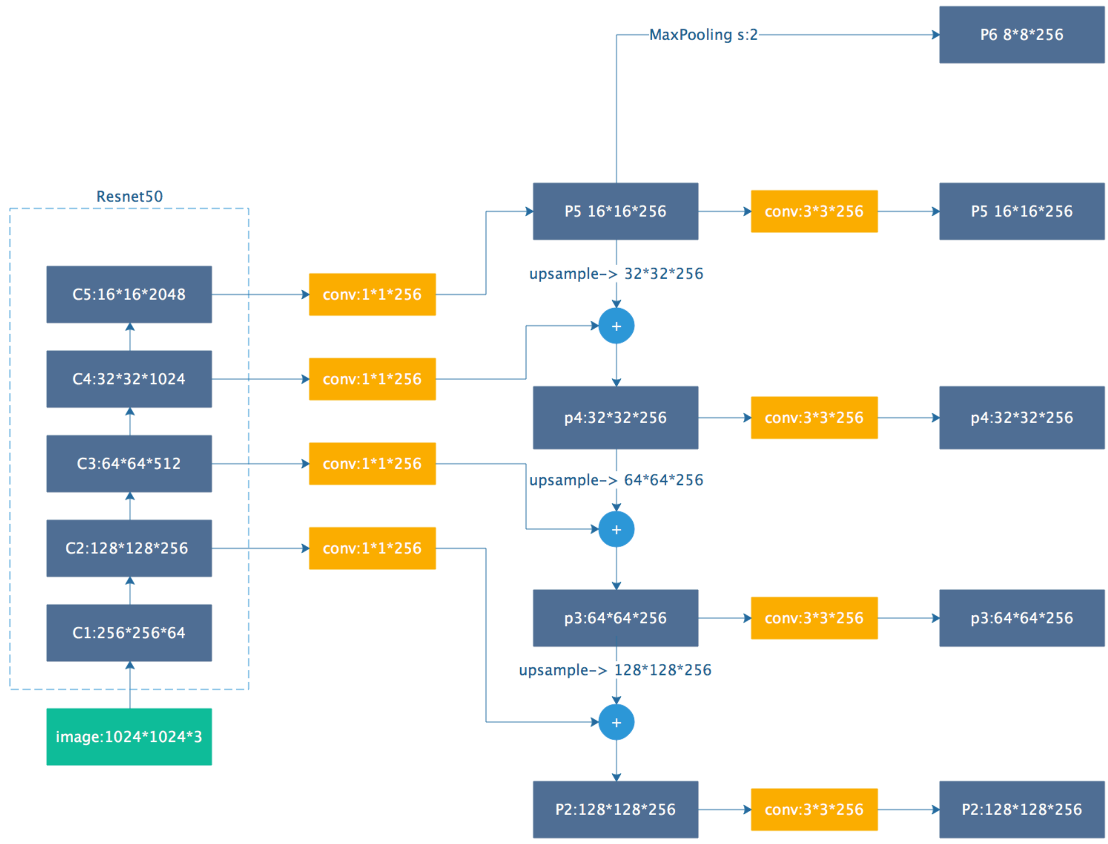
参考Github 代码 : https://github.com/matterport/Mask_RCNN

1 网络结构

1.1 Basebone

用Resnet50和特征金字塔网络 提取图片的特征

结构：



Basebone生成的 [p2,p3,p4,p5,p6] 是后面所有操作的数据基础

```
#加载resnet50模型
_, C2, C3, C4, C5 = resnet_graph(input_image, config.BACKBONE,
                                  stage5=True, train_bn=config.TRAIN_BN)


P5 = KL.Conv2D(config.TOP_DOWN_PYRAMID_SIZE, (1, 1), name='fpn_c5p5')(C5)
P4 = KL.Add(name="fpn_p4add")([
    KL.UpSampling2D(size=(2, 2), name="fpn_p5upsampled")(P5),
    KL.Conv2D(config.TOP_DOWN_PYRAMID_SIZE, (1, 1), name='fpn_c4p4')(C4)])
P3 = KL.Add(name="fpn_p3add")([
    KL.UpSampling2D(size=(2, 2), name="fpn_p4upsampled")(P4),
    KL.Conv2D(config.TOP_DOWN_PYRAMID_SIZE, (1, 1), name='fpn_c3p3')(C3)])
P2 = KL.Add(name="fpn_p2add")([
    KL.UpSampling2D(size=(2, 2), name="fpn_p3upsampled")(P3),
    KL.Conv2D(config.TOP_DOWN_PYRAMID_SIZE, (1, 1), name='fpn_c2p2')(C2)])
```

```
P2 = KL.Conv2D(config.TOP_DOWN_PYRAMID_SIZE, (3, 3), padding="SAME",
name="fpn_p2")(P2)
P3 = KL.Conv2D(config.TOP_DOWN_PYRAMID_SIZE, (3, 3), padding="SAME",
name="fpn_p3")(P3)
P4 = KL.Conv2D(config.TOP_DOWN_PYRAMID_SIZE, (3, 3), padding="SAME",
name="fpn_p4")(P4)
P5 = KL.Conv2D(config.TOP_DOWN_PYRAMID_SIZE, (3, 3), padding="SAME",
name="fpn_p5")(P5)
P6 = KL.MaxPooling2D(pool_size=(1, 1), strides=2, name="fpn_p6")(P5)

rpn_feature_maps = [P2, P3, P4, P5, P6]
mrcnn_feature_maps = [P2, P3, P4, P5]
```

1.2 Region Proposal Network(RPN)

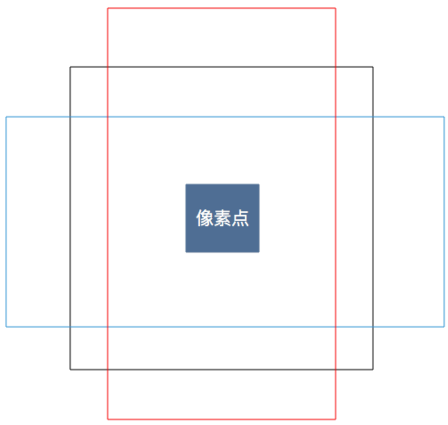
1.2.1 Anchors

在做RPN之前，先要对feature_map的选取锚框anchors，anchor就是选定检测目标存在的范围。

对rpn_feature_maps选取锚框,rpn_feature_maps共有5个特征层[p2,p3,p4,p5,p6], 每个特征层的尺寸不同，选取的锚框大小和规则也不同，所以要分开选择：

1. 每个特征层对应一种锚框选择规则(不同特征层的图像尺寸不同)
2. 所有特征层中每个图片的每个像素点都要用不同的长宽比生成3个锚框

比如:规定长宽比为[1,2,0.5],一个像素点对应的锚框如下




```
def generate_anchors(scales, ratios, shape, feature_stride, anchor_stride):
    """
    scales : 数组，存放不同尺寸feature_map对应的锚框尺寸，如[8,16,32,64,128]
    ratios : 数组，存放每个像素点对应的三个锚框的长宽比，如[0.5,1,2]
    shape  : feature_map的大小，如[height,width]
    feature_stride : 图片的缩放比，用于还原锚框真实大小比如说原图片是1024*1024,
```

```
                 而上述basebone中p2的大小是128*128，则缩放比是8。
    anchor_stride : 像素点的移动步长，真实情况是并不是所有的像素点都要选锚框，可以
                 以一定步长移动，跳着选。
    """

    scales, ratios = np.meshgrid(np.array(scales), np.array(ratios))
    scales = scales.flatten()
    ratios = ratios.flatten()

    #比如scale = 32,ratios=4，这个锚框就是长128，宽16
    heights = scales / np.sqrt(ratios)
    widths = scales * np.sqrt(ratios)

    # 真实像素点分布
    shifts_y = np.arange(0, shape[0], anchor_stride) * feature_stride
    shifts_x = np.arange(0, shape[1], anchor_stride) * feature_stride
    shifts_x, shifts_y = np.meshgrid(shifts_x, shifts_y)

    # 组合上面两个，生成一系列的(x,y,h,w)形式的锚框
    box_widths, box_centers_x = np.meshgrid(widths, shifts_x)
    box_heights, box_centers_y = np.meshgrid(heights, shifts_y)
    box_centers = np.stack(
        [box_centers_y, box_centers_x], axis=2).reshape([-1, 2])

    # 将(x,y,h,w)的形式转换成,左上角和右下角坐标的形式(x1,y1,x2,y2)
    box_sizes = np.stack([box_heights, box_widths], axis=2).reshape([-1,
2])
    boxes = np.concatenate([box_centers - 0.5 * box_sizes,
                            box_centers + 0.5 * box_sizes], axis=1)
    return boxes
```

最后需要生成的锚框数据的shape 为：[N,(x1,y1,x2,y2)]，N为锚框总数

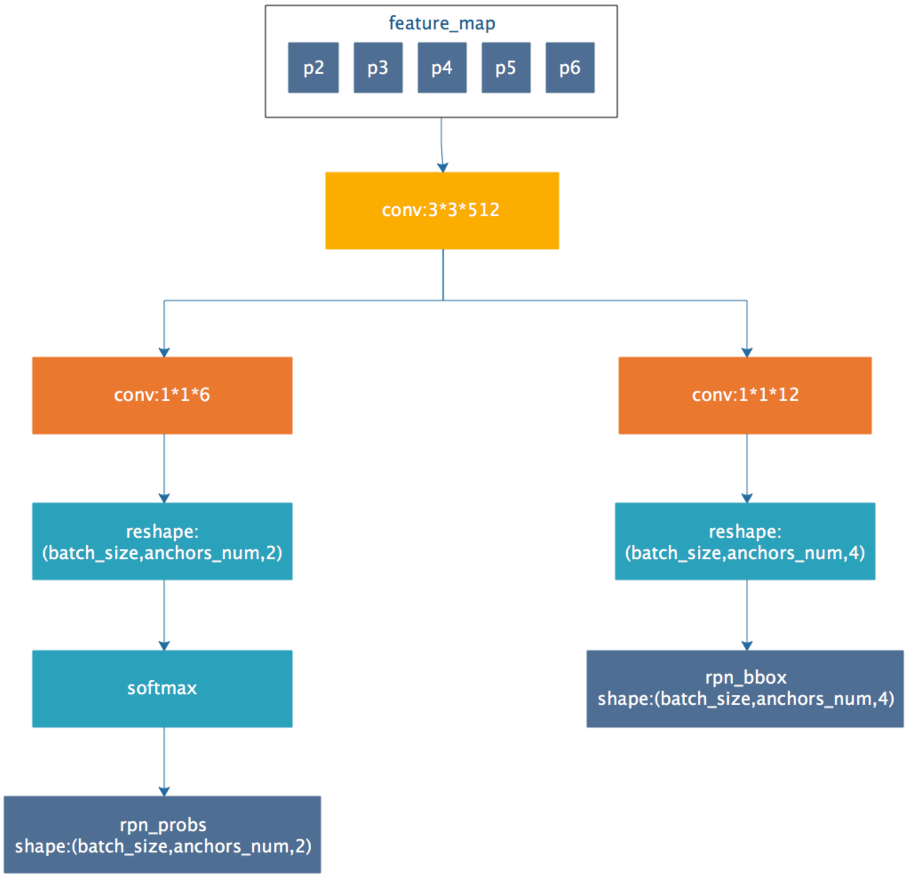
1.2.2 Region Proposal

在生成锚框的过程中可以发现，如果上述生成的所有锚框都用于后续的学习，是十分多余的。因为背景的锚框占了大部分。

假设以上述的feature_map为例，锚点步长为1的情况下，总共会生成(128*128+64*64+32*32+16*16+8*8)*3 = 65472的锚框数

Region Proposal的目的就是为了在**如此庞大的数量的锚框数中寻找有用的，传给后续训练。**

Step1



左侧：

其目的是为了分辨出哪些anchors是背景，哪些anchors是物体。

通过训练卷积层(conv:1/1/6)来实现，(conv:1/1/6)实际上是(conv:1/1/2*3)，即对每一个像素点的三个锚框都生成两个值:1.这个锚框是背景的概率 2.这个锚框是物体的概率。最后通过reshape转换成[batch_size,anchors_num(所有锚框数量),2] 的形式，记做rpn_probs

右侧：

其目的是为了生成每个锚框四个坐标点的修正量，即赋予所有锚框自动修正长宽的能力。

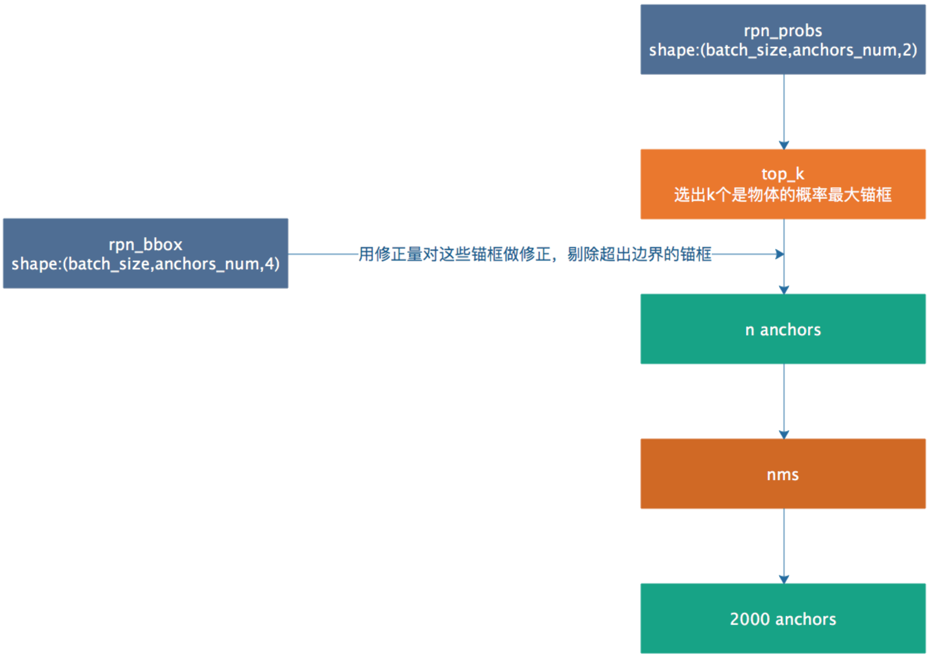
通过训练卷积层(conv:1/1/12)来实现，(conv:1/1/12)实际上是(conv:1/1/4*3)，即每一个像素点三个锚框都要生成(x1,y1,x2,y2)对应的修正量(dx1.dy1,dx2,dy2)，最后reshape成[batch_size,anchors_num(所有锚框数量),4]的形式，记做rpn_probs

```
def rpn_graph(feature_map, anchors_per_location, anchor_stride):
    """
    feature_map : [p2,p3,p4,p5,p6]
    anchors_per_location : 每个像素点对应的锚框数，默认是3个
    anchor_stride : 像素点的移动步长，真实情况是并不是所有的像素点都要选锚框，可以
                    以一定步长移动，跳着选。
```

```
    """
    # 上述的3*3*512的卷积
    shared = KL.Conv2D(512, (3, 3), padding='same', activation='relu',
                       strides=anchor_stride,
                       name='rpn_conv_shared')(feature_map)

    # 左侧第一个1*1*6的卷积
    x = KL.Conv2D(2 * anchors_per_location, (1, 1), padding='valid',
                  activation='linear', name='rpn_class_raw')(shared)

    # 左侧softmax前的reshape
    rpn_class_logits = KL.Lambda(
        lambda t: tf.reshape(t, [tf.shape(t)[0], -1, 2]))(x)

    # softmax
    rpn_probs = KL.Activation(
        "softmax", name="rpn_class_xxx")(rpn_class_logits)

    # 右侧第一个1*1*12的卷积
    x = KL.Conv2D(anchors_per_location * 4, (1, 1), padding="valid",
                  activation='linear', name='rpn_bbox_pred')(shared)

    # 右侧reshape
    rpn_bbox = KL.Lambda(lambda t: tf.reshape(t, [tf.shape(t)[0], -1, 4]))
(x)

    return [rpn_class_logits, rpn_probs, rpn_bbox]
```

Step2

step1中得到了两个输出rpn_probs和rpn_probs，通过这两个输出，来选择我们需要的锚框。



1. 根据rpn_probs，用锚框属于物体的概率，选择概率最大的若干个锚框
2. 通过rpn_bbox修正选出来的锚框
3. 做nms(非极大抑制)，根据IoU选出2000个anchors

> 非极大抑制：
>
> 目的是为了去除冗余的锚框，因为不做nms的话，很有可能会出现图片中某块区域是物体,那么这区域内的锚框rpn_probs都很大，就可能选出一堆重叠度很大的锚框。这些是冗余的。
>
> 步骤：
>
> 1. 将锚框有序排列
> 2. 选中一个得分最高的锚框x，且未处理过的锚框
> 3. 将其余锚框中与x的重叠度IoU大于一定阈值的锚框都删除
> 4. 标记x为处理过的锚框，回到2

```
'''
    scores : 所有锚框的是物体的分数，在rpn_probs中获得
    deltas : 所有锚框的修正量，在rpn_bbox中获得
    anchors: 所有锚框数据，由2.1的generate_anchors获得
'''

# top_k步骤，self.config.PRE_NMS_LIMIT为k
pre_nms_limit = tf.minimum(self.config.PRE_NMS_LIMIT, tf.shape(anchors)[1])
ix = tf.nn.top_k(scores, pre_nms_limit, sorted=True,
                 name="top_anchors").indices
```

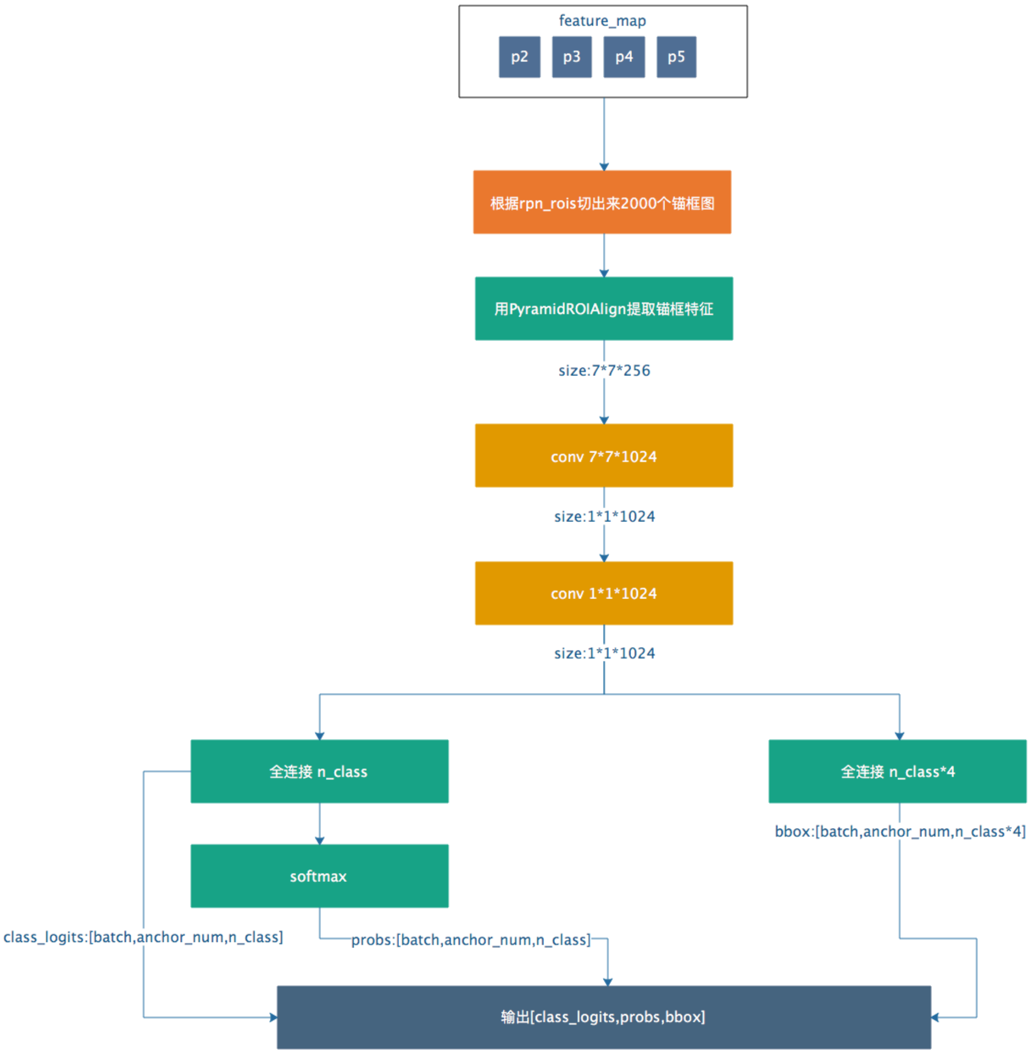
```
scores = utils.batch_slice([scores, ix], lambda x, y: tf.gather(x, y),
                           self.config.IMAGES_PER_GPU)
deltas = utils.batch_slice([deltas, ix], lambda x, y: tf.gather(x, y),
                           self.config.IMAGES_PER_GPU)
pre_nms_anchors = utils.batch_slice([anchors, ix], lambda a, x:
tf.gather(a, x),
                            self.config.IMAGES_PER_GPU,
                            names=["pre_nms_anchors"])

# 通过修正量deltas修正top_k选出来的锚框
boxes = utils.batch_slice([pre_nms_anchors, deltas],
                          lambda x, y: apply_box_deltas_graph(x, y),
                          self.config.IMAGES_PER_GPU,
                          names=["refined_anchors"])

#判断修正后的锚框是否超出边界，超出就剔除
window = np.array([0, 0, 1, 1], dtype=np.float32)
boxes = utils.batch_slice(boxes,
                          lambda x: clip_boxes_graph(x, window),
                          self.config.IMAGES_PER_GPU,
                          names=["refined_anchors_clipped"])

#nms
def nms(boxes, scores):
    indices = tf.image.non_max_suppression(
        boxes, scores, self.proposal_count,
        self.nms_threshold, name="rpn_non_max_suppression")
    proposals = tf.gather(boxes, indices)
    padding = tf.maximum(self.proposal_count - tf.shape(proposals)[0], 0)
    proposals = tf.pad(proposals, [(0, padding), (0, 0)])
    return proposals



proposals = utils.batch_slice([boxes, scores], nms,
                              self.config.IMAGES_PER_GPU)
return proposals
```

最后返回2000个锚框，记做rpn_rois

1.3 Network Heads

Network Heads分成两个主要的部分，一个是计算传统目标检测的结果，包括锚框和类别，另一部是mask r-cnn的特色，计算用于分割图像的mask

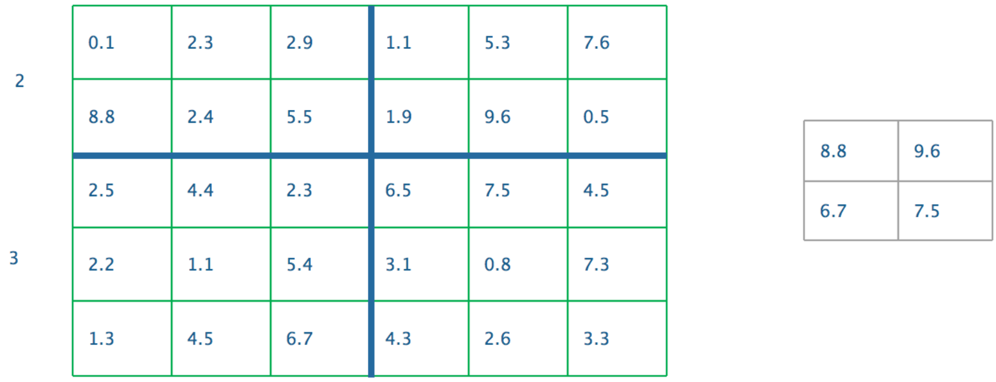
1.3.1 目标检测

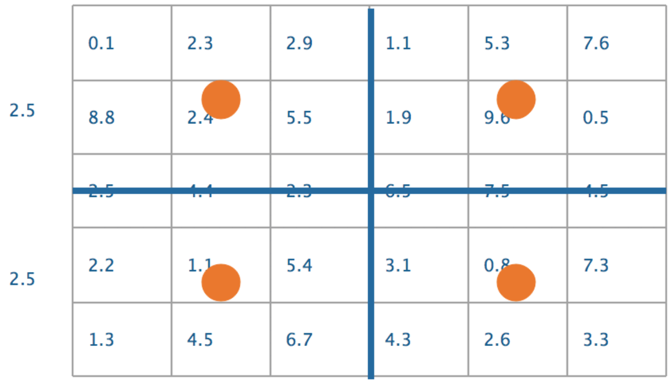


在RPN网络中,已经选出了2000个锚框，即rpn_rois, 将rpn_rois运用到feature_map中(剔除P6)，切出2000个锚框图。

但是存在的问题是，2000个锚框的图片大小不一，无法进行后续的训练，需要提取出统一大小的特征，才能进行后续的操作。一般目标检测的网络中，用pooling的方法来实现这一步的特征提取，Mask R-CNN 提出了一种新的特征提取的办法ROIAlign。

> ROIAlign





> 上图部分是ROIpooling的缩放方式
>
> 下图部分是ROIAlign的缩放方式，最后均分的每个局域内，根据距离中心点最近几个点，用双向线性插值求出中心点的数值

[p2,p3,p4,p5]分别对应的shape为[(128,128,256),(64,64,256),(32,32,256),(16,16,256)]

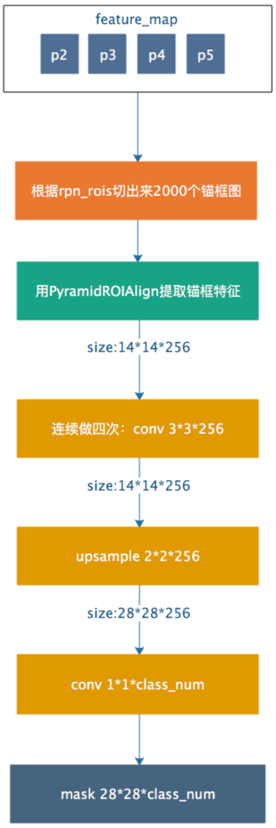
默认的ROIAlign是对图片统一缩至7*7的大小，所以经过ROIAlign后的，shape为(7,7,256)，完整的shape为(batch,anchors_num,7,7,256)，后续操作如图所示，最后输出一个带有类别概率和锚框修正量的结果。

```
def fpn_classifier_graph(rois, feature_maps, image_meta,
                         pool_size, num_classes, train_bn=True,
                         fc_layers_size=1024):
    """
    rois: [batch, anchors_num, (y1, x1, y2, x2)]

    feature_maps: [P2, P3, P4, P5]

    image_meta:原图片的一些原始参数，如原图片大小
```

```
    pool_size: 缩放后的图片宽度 默认7
    num_classes: 类别数
    train_bn: 是否要用Batch Norm

    fc_layers_size: 两个全连接层的大小

    返回值
        logits: [batch, anchors_num, NUM_CLASSES]
        probs: [batch, anchors_num, NUM_CLASSES]
        bbox_deltas: [batch, anchors_num, NUM_CLASSES, (dy, dx, log(dh),
log(dw))]
    """
    # ROIAlign
    x = PyramidROIAlign([pool_size, pool_size],
                        name="roi_align_classifier")([rois, image_meta] +
feature_maps)

    # 7*7*1024 卷积
    x = KL.TimeDistributed(KL.Conv2D(fc_layers_size, (pool_size,
pool_size), padding="valid"),
                           name="mrcnn_class_conv1")(x)
    #Batch Norm
    x = KL.TimeDistributed(BatchNorm(), name='mrcnn_class_bn1')(x,
training=train_bn)
    x = KL.Activation('relu')(x)


    # 1*1*1024卷积
    x = KL.TimeDistributed(KL.Conv2D(fc_layers_size, (1, 1)),
                           name="mrcnn_class_conv2")(x)
    x = KL.TimeDistributed(BatchNorm(), name='mrcnn_class_bn2')(x,
training=train_bn)
    x = KL.Activation('relu')(x)

    # 铺平
    shared = KL.Lambda(lambda x: K.squeeze(K.squeeze(x, 3), 2),
                       name="pool_squeeze")(x)


    # 左侧全连接和卷积
    mrcnn_class_logits = KL.TimeDistributed(KL.Dense(num_classes),
                                            name='mrcnn_class_logits')
(shared)
    mrcnn_probs = KL.TimeDistributed(KL.Activation("softmax"),
                                     name="mrcnn_class")
(mrcnn_class_logits)

    # 右侧全连接和卷积
```

```
    x = KL.TimeDistributed(KL.Dense(num_classes * 4, activation='linear'),
                           name='mrcnn_bbox_fc')(shared)
    s = K.int_shape(x)
    mrcnn_bbox = KL.Reshape((s[1], num_classes, 4), name="mrcnn_bbox")(x)



    return mrcnn_class_logits, mrcnn_probs, mrcnn_bbox
```

1.3.2 图像分割



最后生成mask的大小为 28*28, 并且为每个锚框都生成class_num个mask，确保一个锚框内如果有多个物体重叠，都可以被标记到。

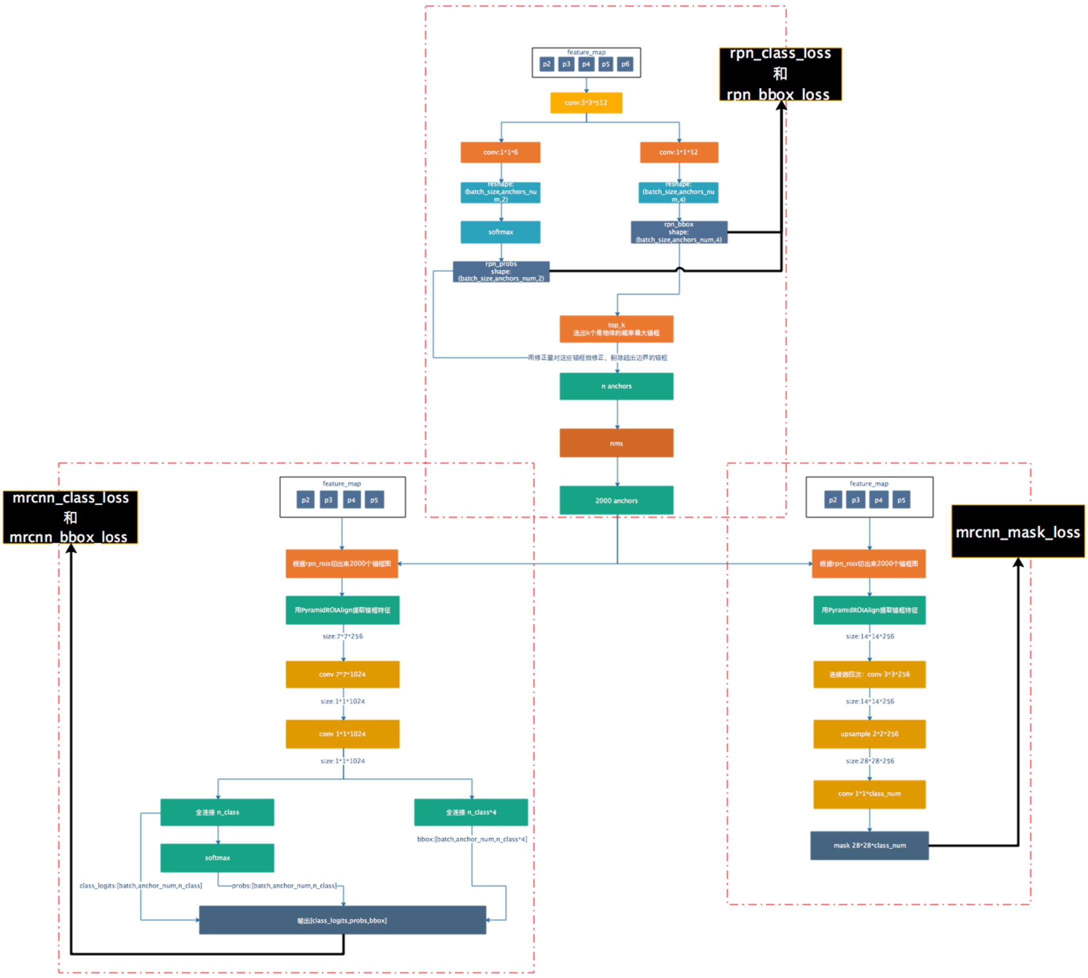
> 因为这里生成的mask是固定大小的，28*28，那么后续生成真实mask的时候要做缩放处理，根据锚框形状和图片的大小对mask进行缩放，来生成真实的mask，在源码的util包中的unmold_mask函数实现

```
def build_fpn_mask_graph(rois, feature_maps, image_meta,
                         pool_size, num_classes, train_bn=True):
    """
```

```
    rois: [batch, anchors_num, (y1, x1, y2, x2)]
    feature_maps:[P2, P3, P4, P5].
    pool_size: ROIAlign缩放的大小，这里默认14
    num_classes: 类别数量
    train_bn:是否batch norm

    返回值: Masks [batch, anchors_num, 28, 28, num_classes]
    """

    #ROIAlign 输出14*14*256
    x = PyramidROIAlign([pool_size, pool_size],
                        name="roi_align_mask")([rois, image_meta] +
feature_maps)

    # 卷积 3*3*256
    x = KL.TimeDistributed(KL.Conv2D(256, (3, 3), padding="same"),
                           name="mrcnn_mask_conv1")(x)
    x = KL.TimeDistributed(BatchNorm(),
                           name='mrcnn_mask_bn1')(x, training=train_bn)
    x = KL.Activation('relu')(x)

    # 卷积 3*3*256
    x = KL.TimeDistributed(KL.Conv2D(256, (3, 3), padding="same"),
                           name="mrcnn_mask_conv2")(x)
    x = KL.TimeDistributed(BatchNorm(),
                           name='mrcnn_mask_bn2')(x, training=train_bn)
    x = KL.Activation('relu')(x)

    # 卷积 3*3*256
    x = KL.TimeDistributed(KL.Conv2D(256, (3, 3), padding="same"),
                           name="mrcnn_mask_conv3")(x)
    x = KL.TimeDistributed(BatchNorm(),
                           name='mrcnn_mask_bn3')(x, training=train_bn)
    x = KL.Activation('relu')(x)

    # 卷积 3*3*256
    x = KL.TimeDistributed(KL.Conv2D(256, (3, 3), padding="same"),
                           name="mrcnn_mask_conv4")(x)
    x = KL.TimeDistributed(BatchNorm(),
                           name='mrcnn_mask_bn4')(x, training=train_bn)
    x = KL.Activation('relu')(x)

    # 卷积 2*2*256
    x = KL.TimeDistributed(KL.Conv2DTranspose(256, (2, 2), strides=2,
activation="relu"),
                           name="mrcnn_mask_deconv")(x)


    # 卷积 1*1*class_num
```

```
    x = KL.TimeDistributed(KL.Conv2D(num_classes, (1, 1), strides=1,
activation="sigmoid"),
                           name="mrcnn_mask")(x)
    return x
```

2 训练



如上图，我们需要训练的一共有三部分：

1. rpn阶段的rpn_class和rpn_bbox的loss
2. head部分物体检测的mrcnn_class和mrcnn_bbox的loss
3. head部分生成mask的mrcnn_mask的loss

2.1 训练时的特殊处理

对于RPN层，生成了2000个锚框，做预测的时候可以这么做，但是对于训练模型来说，还是太多了，训练时，会对RPN层生成的2000个锚框再次进行筛选，筛选出200个利于训练的锚框，再传给head层计算损失。

筛选方法：

> 计算真实标记的锚框和生成的2000个锚框的IoU
>
> 按IoU降序排序
>
> 顺序选择IoU>=0.5的锚框，和Iou<0.5的锚框， 两种锚框的数量比为1:3，且总数为200

具体代码见model文件detection_targets_graph函数overlaps_graph以下部分。

2.2 损失函数

rpn_class_loss : rpn_class_loss_graph函数

```
def rpn_class_loss_graph(rpn_match, rpn_class_logits):
    """RPN anchor classifier loss.

    rpn_match: [batch, anchors, 1]. Anchor match type. 1=positive,
               -1=negative, 0=neutral anchor.
    rpn_class_logits: [batch, anchors, 2]. RPN classifier logits for FG/BG.
    """
    # Squeeze last dim to simplify
    rpn_match = tf.squeeze(rpn_match, -1)
    # Get anchor classes. Convert the -1/+1 match to 0/1 values.
    anchor_class = K.cast(K.equal(rpn_match, 1), tf.int32)
    # Positive and Negative anchors contribute to the loss,
    # but neutral anchors (match value = 0) don't.
    indices = tf.where(K.not_equal(rpn_match, 0))
    # Pick rows that contribute to the loss and filter out the rest.
    rpn_class_logits = tf.gather_nd(rpn_class_logits, indices)
    anchor_class = tf.gather_nd(anchor_class, indices)
    # Cross entropy loss
    loss = K.sparse_categorical_crossentropy(target=anchor_class,
                                             output=rpn_class_logits,
                                             from_logits=True)
    loss = K.switch(tf.size(loss) > 0, K.mean(loss), tf.constant(0.0))
    return loss
```

rpn_bbox_loss : rpn_bbox_loss_graph函数

```
def rpn_bbox_loss_graph(config, target_bbox, rpn_match, rpn_bbox):
    """Return the RPN bounding box loss graph.

    config: the model config object.
    target_bbox: [batch, max positive anchors, (dy, dx, log(dh), log(dw))].
        Uses 0 padding to fill in unsed bbox deltas.
```

```
    rpn_match: [batch, anchors, 1]. Anchor match type. 1=positive,
               -1=negative, 0=neutral anchor.
    rpn_bbox: [batch, anchors, (dy, dx, log(dh), log(dw))]
    """
    # Positive anchors contribute to the loss, but negative and
    # neutral anchors (match value of 0 or -1) don't.
    rpn_match = K.squeeze(rpn_match, -1)
    indices = tf.where(K.equal(rpn_match, 1))

    # Pick bbox deltas that contribute to the loss
    rpn_bbox = tf.gather_nd(rpn_bbox, indices)

    # Trim target bounding box deltas to the same length as rpn_bbox.
    batch_counts = K.sum(K.cast(K.equal(rpn_match, 1), tf.int32), axis=1)
    target_bbox = batch_pack_graph(target_bbox, batch_counts,
                                   config.IMAGES_PER_GPU)

    loss = smooth_l1_loss(target_bbox, rpn_bbox)

    loss = K.switch(tf.size(loss) > 0, K.mean(loss), tf.constant(0.0))
    return loss
```

mrcnn_class_loss : mrcnn_class_loss_graph函数

```
def mrcnn_class_loss_graph(target_class_ids, pred_class_logits,
                           active_class_ids):
    """Loss for the classifier head of Mask RCNN.

    target_class_ids: [batch, num_rois]. Integer class IDs. Uses zero
        padding to fill in the array.
    pred_class_logits: [batch, num_rois, num_classes]
    active_class_ids: [batch, num_classes]. Has a value of 1 for
        classes that are in the dataset of the image, and 0
        for classes that are not in the dataset.
    """
    # During model building, Keras calls this function with
    # target_class_ids of type float32. Unclear why. Cast it
    # to int to get around it.
    target_class_ids = tf.cast(target_class_ids, 'int64')

    # Find predictions of classes that are not in the dataset.
    pred_class_ids = tf.argmax(pred_class_logits, axis=2)
    # TODO: Update this line to work with batch > 1. Right now it assumes
all
    #       images in a batch have the same active_class_ids
    pred_active = tf.gather(active_class_ids[0], pred_class_ids)

    # Loss
```

```
    loss = tf.nn.sparse_softmax_cross_entropy_with_logits(
        labels=target_class_ids, logits=pred_class_logits)

    # Erase losses of predictions of classes that are not in the active
    # classes of the image.
    loss = loss * pred_active

    # Computer loss mean. Use only predictions that contribute
    # to the loss to get a correct mean.
    loss = tf.reduce_sum(loss) / tf.reduce_sum(pred_active)
    return loss
```

mrcnn_bbox_loss : mrcnn_bbox_loss_graph函数

```
def mrcnn_bbox_loss_graph(target_bbox, target_class_ids, pred_bbox):
    """Loss for Mask R-CNN bounding box refinement.

    target_bbox: [batch, num_rois, (dy, dx, log(dh), log(dw))]
    target_class_ids: [batch, num_rois]. Integer class IDs.
    pred_bbox: [batch, num_rois, num_classes, (dy, dx, log(dh), log(dw))]
    """
    # Reshape to merge batch and roi dimensions for simplicity.
    target_class_ids = K.reshape(target_class_ids, (-1,))
    target_bbox = K.reshape(target_bbox, (-1, 4))
    pred_bbox = K.reshape(pred_bbox, (-1, K.int_shape(pred_bbox)[2], 4))

    # Only positive ROIs contribute to the loss. And only
    # the right class_id of each ROI. Get their indices.
    positive_roi_ix = tf.where(target_class_ids > 0)[:, 0]
    positive_roi_class_ids = tf.cast(
        tf.gather(target_class_ids, positive_roi_ix), tf.int64)
    indices = tf.stack([positive_roi_ix, positive_roi_class_ids], axis=1)

    # Gather the deltas (predicted and true) that contribute to loss
    target_bbox = tf.gather(target_bbox, positive_roi_ix)
    pred_bbox = tf.gather_nd(pred_bbox, indices)

    # Smooth-L1 Loss
    loss = K.switch(tf.size(target_bbox) > 0,
                    smooth_l1_loss(y_true=target_bbox, y_pred=pred_bbox),
                    tf.constant(0.0))
    loss = K.mean(loss)
    return loss
```

mrcnn_mask_loss : mrcnn_mask_loss_graph函数

```
def mrcnn_mask_loss_graph(target_masks, target_class_ids, pred_masks):
```

```
    """Mask binary cross-entropy loss for the masks head.

    target_masks: [batch, num_rois, height, width].
        A float32 tensor of values 0 or 1. Uses zero padding to fill array.
    target_class_ids: [batch, num_rois]. Integer class IDs. Zero padded.
    pred_masks: [batch, proposals, height, width, num_classes] float32
tensor
                with values from 0 to 1.
    """
    # Reshape for simplicity. Merge first two dimensions into one.
    target_class_ids = K.reshape(target_class_ids, (-1,))
    mask_shape = tf.shape(target_masks)
    target_masks = K.reshape(target_masks, (-1, mask_shape[2],
mask_shape[3]))
    pred_shape = tf.shape(pred_masks)
    pred_masks = K.reshape(pred_masks,
                           (-1, pred_shape[2], pred_shape[3],
pred_shape[4]))
    # Permute predicted masks to [N, num_classes, height, width]
    pred_masks = tf.transpose(pred_masks, [0, 3, 1, 2])

    # Only positive ROIs contribute to the loss. And only
    # the class specific mask of each ROI.
    positive_ix = tf.where(target_class_ids > 0)[:, 0]
    positive_class_ids = tf.cast(
        tf.gather(target_class_ids, positive_ix), tf.int64)
    indices = tf.stack([positive_ix, positive_class_ids], axis=1)

    # Gather the masks (predicted and true) that contribute to loss
    y_true = tf.gather(target_masks, positive_ix)
    y_pred = tf.gather_nd(pred_masks, indices)

    # Compute binary cross entropy. If no positive ROIs, then return 0.
    # shape: [batch, roi, num_classes]
    loss = K.switch(tf.size(y_true) > 0,
                    K.binary_crossentropy(target=y_true, output=y_pred),
                    tf.constant(0.0))
    loss = K.mean(loss)
    return loss
```