

一、介绍概述

本文主要介绍yolov8在训练过程中的两个阶段：

1. Task-Aligned Assigner 正负样本动态分配策略
2. 损失函数计算

由于个人感觉官方代码读起来比较困难，故按照自己的思路重新写了一遍，下面将按照自己的代码进行讲解。

假设：

网络输入大小为：images = b x 3 x 640 x 640

类别数目 cls_num = 2 (person + car)

超参数 reg_max = 16 16 16

输出通道为： 4 * reg_max + cls_num

则三个分支的预测size分别为：b x 66 x 80 x 80 + b x 66 x 40 x 40 + b x 66 x 20 x 20

tip: 这里的66并不是网络直接输出66个通道的特征图，而是为了方便处理，在cls(输出通道c=2)和reg(c=64),经过concat后得到的66.

二、Task-Aligned Assigner

1. 分配原理

Task-Aligned Assigner，又名对齐分配器，在YOLOv8中是一种**动态的分配策略**。

一言以蔽之：针对所有像素点预测的 **Cls score** 和 **Reg Score**(Box与每个GT box的 **IOU**) ,通过**加权的方式**得到最终的**加权分数**，通过对加权分数进行排序后选择**Topk**个正样本。

$$t = s^{\alpha} \times u^{\beta}$$

其中，s是所有像素点-所有类别的Cls score，U是所有像素点预测box与所有GTbox的Reg score(IOU)， α 和 β 为权重超参数，两者**相乘**就可以**衡量对齐程度**，t作为加权分数。

2. 前期处理

代码如下：

```
1 self.cuda = True if inputs[0].is_cuda else False
```

```

2         self.FloatTensor = torch.cuda.FloatTensor if self.cuda else torch.FloatTensor
3         self.LongTensor = torch.cuda.LongTensor if self.cuda else torch.LongTensor
4
5         # ----- 预测结果预处理 ----- #
6         # 将多尺度输出整合为一个Tensor,便于整体进展矩阵运算
7         pred_scores,pred_regs,strides = self.pred_process(inputs)
8
9         # ----- 生成anchors锚点 -----#
10        # 各尺度特征图每个位置一个锚点Anchors(与yolov5中的anchors不同,此处不是先验框)
11        # 表示每个像素点只有一个预测结果
12        self.anc_points,self.stride_scales = self.make_anchors(strides)
13
14        # ----- 解码 ----- #
15        # 预测回归结果解码到bbox xmin,ymin,xmax,ymax格式
16        pred_bboxes = self.decode(pred_regs)
17
18        # ----- 标注数据预处理 ----- #
19        gt_bboxes,gt_labels,gt_mask = self.ann_process(annotations)

```

A. 预测结果解码

对于网络输出的box信息，实际上表示的是相对于每个像素点上不同anchor的偏移值(左上角或右下角相对于中心点的距离)

(1) 预测数据预处理

```

1 # ----- 预测结果预处理 ----- #
2 # 将多尺度输出整合为一个Tensor,便于整体进展矩阵运算
3 pred_scores,pred_regs,strides = self.pred_process(inputs)

```

```

1 def pred_process(self,inputs):
2     '''
3         L = class_num + 4*self.reg_max = class_num + 64
4         多尺度结果bxLx80x80,bxLx40x40,bxLx20x20,整合到一起为 b x 8400 x L
5         按照cls 与 box 拆分为 b x 8400 x 2 , b x 8400 x 64
6     '''

```

```

7      predictions = [] # 记录每个尺度的转换结果
8      strides = [] # 记录每个尺度的缩放倍数
9      for input in inputs:
10         self.bs,cs,in_h,in_w = input.shape
11         # 计算该尺度特征图相对于网络输入的缩放倍数
12         stride = self.input_h // in_h
13         strides.append(stride)
14         # shape 转换 如 b x 80 x 80 x (64+cls_num) -> b x 6400 x (64+cls_num)
15         prediction = input.view(self.bs,4*self.reg_max+self.class_num,-1).permute(0,2,1)
16         predictions.append(prediction)
17         # b x (6400+1600+400) x (cls_num+64) = b x 8400 x (64+cls_num) = b x 8400 x (cls_num+64)
18         predictions = torch.cat(predictions,dim=1)
19         # 按照cls 与 reg 进行拆分
20         # b x 8400 x cls_num = b x 8400 x 2
21         pred_scores = predictions[...,4*self.reg_max:]
22         # b x 8400 x 64
23         pred_regs = predictions[:4*self.reg_max]
24         return pred_scores,pred_regs,strides
25

```

(2) 生成所有anchor锚点的中心坐标和缩放尺度

```

1 # ----- 生成anchors锚点 -----#
2 # 各尺度特征图每个位置一个锚点Anchors(与yolov5中的anchors不同,此处不是先验框)
3 # 表示每个像素点只有一个预测结果
4 self.anc_points,self.stride_scales = self.make_anchors(strides)

```

```

1 def make_anchors(self,strides,grid_cell_offset=0.5):
2     '''
3         各特征图每个像素点一个锚点即Anchors,即每个像素点只预测一个box
4         故共有 80x80 + 40x40 + 20x20 = 8400个anchors
5     '''
6     # anc_points : 8400 x 2 , 每个像素中心点坐标
7     # strides_tensor: 8400 x 1 , 每个像素的缩放倍数
8     anc_points,strides_tensor = [],[]
9     for i , stride in enumerate(strides):

```

```

10         in_h = self.input_h//stride
11         in_w = self.input_w//stride
12
13         # anchor坐标取特征图每个特征点的中心点
14         sx = torch.arange(0,in_w).type(self.FloatTensor) + grid_cell_offset
15         sy = torch.arange(0,in_h).type(self.FloatTensor) + grid_cell_offset
16         # in_h x in_w
17         grid_y,grid_x = torch.meshgrid(sy,sx)
18         # in_h x in_w x 2 -> N x 2
19         anc_points.append(torch.stack((grid_x,grid_y),-1).view(-1,2).type(self.FloatTensor))
20         strides_tensor.append(torch.full((in_h*in_w,1),stride).type(self.FloatTensor))
21
22         return torch.cat(anc_points,dim=0),torch.cat(strides_tensor,dim=0)

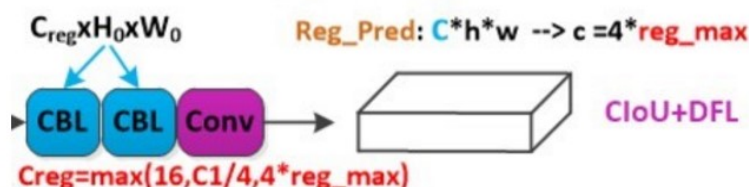
```

(3) 预测结果解码

```

1 # ----- 解码 ----- #
2 # 预测回归结果解码到bbox xmin,ymin,xmax,ymax格式
3 pred_bboxes = self.decode(pred_regs)

```



此处， $reg_max = 16$ ，通过16个值，结合softmax对**box**的**4个预测值**实现离散回归。最后通过**积分的方式**，得到最终结果。

具体代码如下：

```

1     def decode(self,pred_regs):
2         '''
3             预测结果解码
4             1. 对bbox预测回归的分布进行积分
5             2. 结合anc_points，得到所有8400个像素点的预测结果
6         '''
7         if self.use_dfl:
8             b,a,c = pred_regs.shape # b x 8400 x 64

```

```

9          # 分布通过 softmax 进行离散化处理
10         # b x 8400 x 64 -> b x 8400 x 4 x 16 -> softmax处理
11         # l,t,r,b其中每个坐标值对应16个位置(0-15)的概率值
12         # 概率表示每个位置对于最终坐标值的重要程度
13         pred_regs = pred_regs.view(b,a,4,c//4).softmax(3)
14         # 积分，相当于对16个分布值进行加权求和，最终的结果是所有位置的加权求和
15         # b x 8400 x 4
16         pred_regs = pred_regs.matmul(self.proj.type(self.FloatTensor))
17
18         # 此时的regs,shape-> bx8400x4,其中4表示 anc_point中心点分别距离预测box的左
19         lt = pred_regs[..., :2]
20         rb = pred_regs[..., 2:]
21         # xmin ymin
22         x1y1 = self.anc_points - lt
23         # xmax ymax
24         x2y2 = self.anc_points + rb
25         # b x 8400 x 4
26         pred_bboxes = torch.cat([x1y1,x2y2],dim=-1)
27         return pred_bboxes

```

积分之后得到的pred_regs，其中的4个值，分别表示什么？

如果pred_regs的最后维度的4个值用 **left_regs**, **top_regs**, **right_regs**, **bottom_regs** 表示，则它们分别表示在特征图(80x80,40x40,20x20)每个像素点上，anchors points**中心点**距离预测框**左侧边**、**上侧边**、**右侧边**、**下侧边**的距离。

16个距离中心点距固定值

分别预测	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	
左边	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.05	0.05	0.6	0.3	0.0	0.0	0.0	0.0	Softmax 离散 分布 值
上边	0.0	0.0	0.0	0.03	0.2	0.7	0.07	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	
右边	0.0	0.0	0.0	0.0	0.05	0.0	0.0	0.0	0.0	0.05	0.6	0.02	0.1	0.8	0.08	0.0	
下边	0.0	0.0	0.0	0.0	0.4	0.5	0.06	0.04	0.0	0.05	0.6	0.0	0.0	0.0	0.0	0.0	
grid中心 点距离																	

解码加权求和后得到的左边距离中心点的距离值

```

left_regs = 0.05x8 + 0.05x9 + 0.6x10 + 0.3x11 = 10.15
top_reg = 0.03x3 + 0.2x4 + 0.7x5 + 0.07x6 = 4.81
right_reg = 0.02x11 + 0.1x12 + 0.8x13 + 0.08x14 = 12.94
bottom_reg = 0.4x4 + 0.5x5 + 0.06x6 + 0.04x7 = 3.58

```

如当前特征图为40x40,中心点横坐标为20.6,纵坐标为18.3

```

x1 = 20.6 - 10.15 = 10.45
y1 = 18.3 - 4.81 = 13.49
x2 = 20.6 + 12.94 = 33.54
y2 = 18.3 + 3.58 = 21.88

```

代码

```

# l,t,r,b其中每个坐标值对应16个位置(0-15)的概率值
# 概率表示每个位置对于最终坐标值的重要程度
pred_regs = pred_regs.view(b,a,4,c//4).softmax(3)
# 积分，相当于对16个分布值进行加权求和
# b x 8400 x 4
pred_regs = pred_regs.matmul(self.proj.type(self.FloatTensor))

```

↓

代码

```


# anc_point中心点分别距离预测box的左上边与右下边的距离
lt = pred_regs[..., :2]
rb = pred_regs[..., 2:]
# xmin ymin
x1y1 = self.anc_points - lt
# xmax ymax
x2y2 = self.anc_points + rb

```

B. 标注Targets数据预处理

```
1      # ----- 标注数据预处理 ----- #
2      gt_bboxes,gt_labels,gt_mask = self.ann_process(annotations)
```

预处理的目的是：

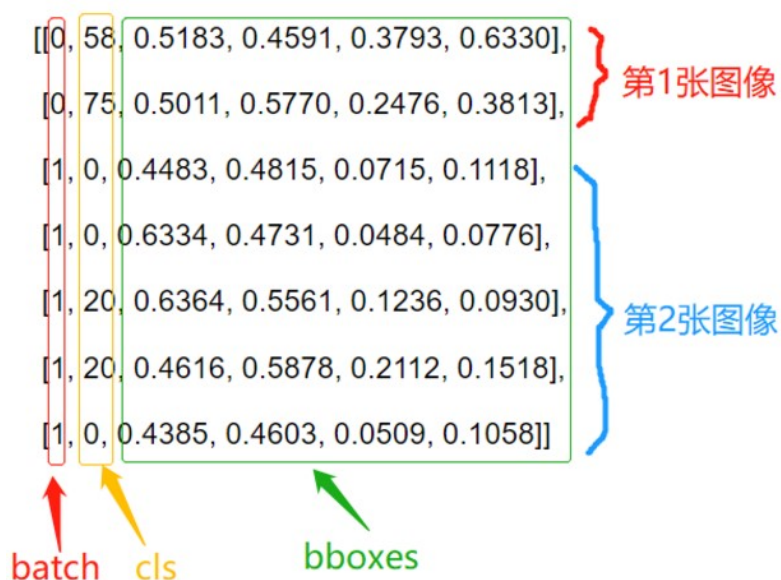
因为batch内不同图像的标注目标个数可能不同，需要进行对齐处理。所谓对齐，如batch_size=2, 其中第二张图像标注5个box,则其shape为 5 x 6, 6表示，第一张图像标注2个box，则其shape为2x6,故需要按标注目标数目最大的进行对齐，即将第1张图像的2x6填充为5x6,空余位置用0补齐。

经过预处理之后，得到的标注数据，在后期训练正样本的筛选过程中就可以更方便的调用。

下面结合图像对该过程进行叙述：

(1) Dataloader获取信息

如batch_size = 2,cls_num=80, 其中第1张图像有2个obj，第二张图像有5个obj，如下所示：



(2) 对齐操作

将box信息由归一化尺度转换到输入图像尺度，并对batch内每张图像的gt个数进行对齐(目标个数都设置一个统一的值M，方便进行矩阵运算)

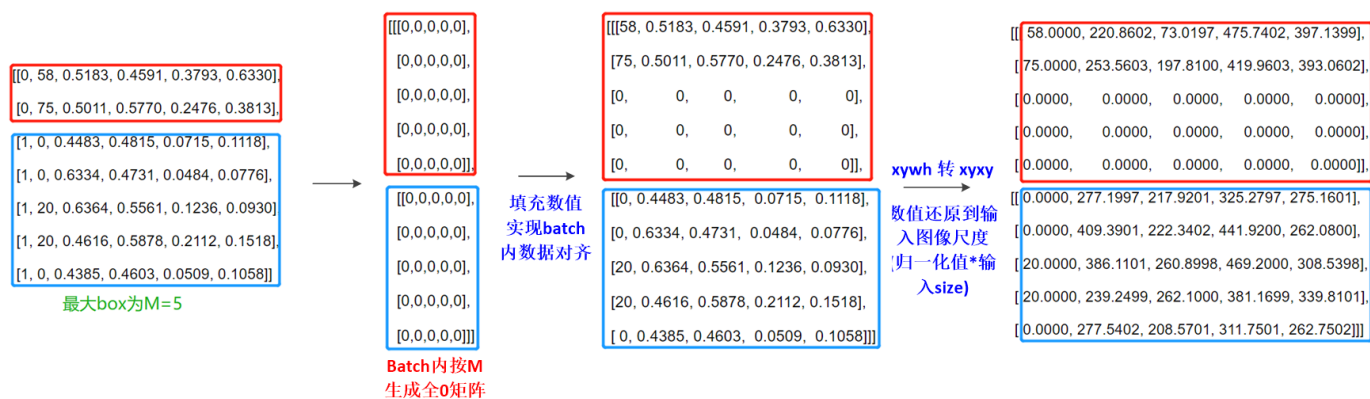
M值的设置规则为，选取batch内最大的gt num作为M

```

1  def ann_process(self, annotations):
2      '''
3          batch内不同图像标注box个数可能不同，故进行对齐处理
4          1. 按照batch内的最大box数目M,新建全0tensor
5          2. 然后将实际标注数据填充与前面，如后面为0，则说明不足M，用0补齐
6      '''
7      # 获取batch内每张图像标注box的batch_idx
8      batch_idx = annotations[:,0]
9      # 计算每张图像中标注框的个数
10     # 原理对tensor内相同值进行汇总
11     _,counts = batch_idx.unique(return_counts=True)
12     counts = counts.type(torch.int32)
13     # 按照batch内最大M个GT创新全0的tensor b x M x 5 ,其中5 = cx,cy,width,height
14     res = torch.zeros(self.bs,counts.max(),5).type(self.FloatTensor)
15     for j in range(self.bs):
16         matches = batch_idx == j
17         n = matches.sum()
18         if n:
19             res[j,:n] = annotations[matches,1:]
20     # res 为归一化之后的结果,需通过scales映射回输入尺度
21     scales = [self.input_w,self.input_h,self.input_w,self.input_h]
22     scales = torch.tensor(scales).type(self.FloatTensor)
23     res[..., :4] = xywh2xyxy(res[..., :4]).mul_(scales)
24     # gt_bboxes b x M x 4
25     # gt_labels b x M x 1
26     gt_bboxes,gt_labels = res[..., :4],res[..., 4:]
27     # gt_mask b x M
28     # 通过对四个坐标值相加，如果为0，则说明该gt信息为填充信息，在mask中为False，
29     # 后期计算过程中会进行过滤
30     gt_mask = gt_bboxes.sum(2,keepdim=True).gt_(0)
31     return gt_bboxes,gt_labels,gt_mask

```

过程如下：



然后，**gt_bboxes**、**gt_labels**分开，并得到相应的**gt_mask**(用于区分正负样本)



3. 正负样本分配

大体流程：

(1) 网络输出的**pred_scores**[bx8400xcls_num],进行sigmoid处理(每个类别按照2分类处理)。

(2) 经过解码的**pred_bboxes**[bx8400x4], 与 **stride_tensor**[8400x1]相乘,将bboxes转换到网络输入尺度[bx3x640x640];

(3) 预处理得到的**anchors_points**[8400x2],与**stride_tensor**[8400x1]相乘, 将anchors的中心点坐标转换到输入尺度

(4) 然后, 将上述**pred_scores**、**pred_bboxes**、**anchors_points**,还有**标注数据预处理**之后的**gt_labels**、**gt_bboxes**、**gt_mask**, 相结合进行正样本的筛选工作。

```
1 self.assigner = TaskAlignedAssigner(topk=10, num_classes=self.nc, alpha=0.5, be
2 # 每个gt box 最多选择topk个候选框作为正样本
```


$$t = s^{\alpha} \times u^{\beta}$$

其中，S是预测类别分值，U是预测框和GT Box的ciou值， α 和 β 为权重超参数，两者相乘就可以衡量匹配程度，当Cls的分值越高且CIOU越高时，t的值就越接近于1,此时预测box就与GTbox越匹配，就越符合正样本的标准。

通过训练t可以引导网络动态的关注于高质量的正样本。

a. 对于每个GT，对所有预测框基于该GT的类别pred score 结合 与该GT box的 CIOU，加权得到一个关联Cls及Box Reg的对齐分数alignment_metrics。

b. 对于每个GT，直接基于alignment_metrics对齐分数，通过排序后，选取topK个预测框作为正样本。

```

1      # ----- 正负样本筛选 ----- #
2      target_bboxes,target_scores,fg_mask= self.assigner(pred_scores.detach()
3                                                         pred_bboxes.detach()*se
4                                                         self.anc_points*self.st
5                                                         gt_labels,
6                                                         gt_bboxes,
7                                                         gt_mask)

```

对流程进行梳理：

<1> 初步筛选

原理： anchor_points即落在gt_boxes内部，作为初步筛选的正样本。

得到gt_boxes的左上角lt，以及右下角rb，分别令anchor_points减去lt，rb减去anchor_points,如果结果均为正，则说明该anchor_point在gt_box内部。对于anchor_points: 8400 x 2，得到对应的mask -> in_gts_mask, 便于后面的过滤操作。

代码如下：

```

1 # ----- 初筛正样本 ----- #
2 # ----- 判断anchor锚点是否在gtbox内部 ----- #
3 # M x 8400
4 in_gts_mask = self.__get_in_gts_mask(gt_bboxes,anc_points)

```

```

1  def __get_in_gts_mask(self, gt_bboxes, anc_points):
2      # 找到M个GTBox的左上与右下坐标 M x 1 x 4 -> M x 8400 x 4
3      gt_bboxes = gt_bboxes.view(-1, 1, 4).repeat(1, self.nc, 1)
4      lt, rb = gt_bboxes[..., :2], gt_bboxes[..., 2:] # M x 8400 x 2
5      # anc_points 增加一个维度 1 x 8400 x 2 -> M x 8400 x 2
6      anc_points = anc_points.view(1, -1, 2).repeat(self.n_max_boxes, 1, 1)
7      # 差值结果 M x 8400 x 4
8      bbox_details = torch.cat([anc_points - lt, rb - anc_points], dim=-1)
9      # 第三个维度均大于0才说明在gt内部
10     # M x 8400
11     in_gts_mask = bbox_details.amin(2).gt_(self.eps)
12     return in_gts_mask

```

<2> 精细筛选

上面只是进行了粗略的初步筛选，所得到的结果中**仍然存在一部分负样本**(虽然 anchor_point 在 gtbox 内部，但 IOU 过低或 scores 过低)，需要进一步进行筛选。

- 计算 每个预测 box 与所有 GT box 的 CIOW 值，即上面公式中的 u 。
- 预测的 scores 是上面公式中的 s ，将 scores 和 ciows 带入是公式得到 t
- 对 t 进行排序后得到前 topk 个作为正样本，其余的作为负样本

思考：此处 yolov8 与 yolov5 的区别？

a. yolov5 采用的是静态分配策略，通过观察 anchor_box 与 gt_box 的 iou 值，如满足一定阈值 0.5，则认为是正样本。

b. yolov8 采用的是动态分配策略，在分配过程中综合考虑了 iou 与 scores 值，对预测的两个分支综合考虑。

静态分配策略是训练开始之前就确定的，这种分配策略通常基于经验得出，可以根据数据集的特点进行调整，但是不够灵活，可能无法充分利用样本的信息，导致训练结果不佳。

动态分配策略，可以根据训练的进展和样本的特点动态的调整权重(在损失函数中会添加对应的权重，下面会详细叙述)，**在训练初期，模型可能会很难区分正负样本，此时权重惩罚值很大，会更加关注那些容易被错分的样本。**随着训练的进行，模型组件变得更加强大，可以更好地区分样本，因此权重的惩罚值就会动态的变低。**动态分配策略可以根据训练损失或者其他指标来进行调整，可以更好地适应不同的数据集和模型。**

代码如下：

```

1 # ----- 精细筛选 ----- #
2 # 按照公式获取计算结果
3 align_metrics, overlaps = self.__refine_select(pb_scores, pb_bboxes, gt_labels, gt_
4 # 根据计算结果, 排序并选择top10
5 # M x 8400
6 topk_mask = self.__select_topk_candidates(align_metrics, gt_mask.repeat(1, self.n

```

```

1 def __refine_select(self, pb_scores, pb_bboxes, gt_labels, gt_bboxes, gt_mask):
2     # 根据论文公式进行计算得到对应的计算结果
3     # reshape M x 4 -> M x 1 x 4 -> M x 8400 x 4
4     gt_bboxes = gt_bboxes.unsqueeze(1).repeat(1, self.nc, 1)
5     # reshape 8400 x 4 -> 1 x 8400 x 4 -> M x 8400 x 4
6     pb_bboxes = pb_bboxes.unsqueeze(0).repeat(self.n_max_boxes, 1, 1)
7     # 计算所有预测box与所有gtbox的ciou, 相当于公式中的U. M x 8400
8     gt_pb_cious = bbox_iou(gt_bboxes, pb_bboxes, xywh=False, CIoU=True).squeeze(
9     # 过滤填充的GT以及不在GTbox范围内的部分
10    # M x 8400
11    gt_pb_cious = gt_pb_cious * gt_mask
12
13    # 获取与GT同类别的预测结果的scores
14    # 8400 x cls_num -> 1 x 8400 x cls_num -> M x 8400 x cls_num
15    pb_scores = pb_scores.unsqueeze(0).repeat(self.n_max_boxes, 1, 1)
16    # M x 1 -> M
17    gt_labels = gt_labels.long().squeeze(-1)
18    # 针对每个GTBOX从预测值(Mx8400xcls_num)中筛选出对应自己类别Cls的结果, 每个结
19    # M x 8400
20    scores = pb_scores[torch.arange(self.n_max_boxes), :, gt_labels]
21
22    # 根据公式进行计算 M x 8400
23    align_metric = scores.pow(self.alpha) * gt_pb_cious.pow(self.beta)
24    # 过滤填充的GT以及不在GTbox范围内的部分
25    align_metric = align_metric * gt_mask
26    return align_metric, gt_pb_cious

```

```

1 def __select_topk_candidates(self, align_metric, gt_mask):
2     # 从大到小排序, 每个GT的从8400个结果中取前 topk个值, 以及其中的对应索引

```

```

3         # top_metrics : M x topk
4         # top_idx : M x topk
5         topk_metrics, topk_idx = torch.topk(align_metric, self.topk, dim=-1, largest=True)
6         # 生成一个全0矩阵用于记录每个GT的topk的mask
7         topk_mask = torch.zeros_like(align_metric, dtype=gt_mask.dtype, device=align_metric.device)
8         for i in range(self.topk):
9             top_i = topk_idx[:, i]
10            # 对应的top_i位置值为1
11            topk_mask[torch.arange(self.n_max_boxes), top_i] = 1
12        topk_mask = topk_mask * gt_mask
13        # M x 8400
14        return topk_mask

```

<3> 排除一个锚点被分配给多个GT box的情况

a. 通过对mask矩阵，每个anchor对于所有GT求和，查看值是否大于1，如大于1，这说明被分配给多个GT

b. 筛除多余分配的情况，原则：通过观察该anchor与被多分配的每个GT的CIoU值，选择值最大者。

代码如下：

```

1 # ----- 排除某个anchor被重复分配的问题 ----- #
2 # target_gt_idx : 8400
3 # fg_mask : 8400
4 # pos_mask: M x 8400
5 target_gt_idx, fg_mask, pos_mask = self.__filter_repeat_assign_candidates(pos_mask, target_gt_idx)

```

```

1 def __filter_repeat_assign_candidates(self, pos_mask, overlaps):
2     '''
3         pos_mask : M x 8400
4         overlaps: M x 8400
5         过滤原则: 如某anchor被重复分配, 则保留与anchor的ciou值最大的GT
6     '''
7     # 对列求和, 即每个anchor对应的M个GT的mask值求和, 如果大于1, 则说明该anchor被多个GT分配
8     # 8400
9     fg_mask = pos_mask.sum(0)

```

```

10         if fg_mask.max() > 1:#某个anchor被重复分配
11             # 找到被重复分配的anchor, mask位置设为True,复制M个, 为了后面与overlaps
12             # 8400 -> 1 x 8400 -> M x 8400
13             mask_multi_gts = (fg_mask.unsqueeze(0) > 1).repeat([self.n_max_boxe
14             # 每个anchor找到CIOU值最大的GT 索引
15             # 8400
16             max_overlaps_idx = overlaps.argmax(0)
17             # 用于记录重复分配的anchor的与所有GTbox的CIOU最大的位置mask
18             # M x 8400
19             is_max_overlaps = torch.zeros(overlaps.shape, dtype=pos_mask.dtype,
20             # 每个anchor只保留ciou值最大的GT, 对应位置设置为1
21             is_max_overlaps.scatter_(0,max_overlaps_idx.unsqueeze(0),1)
22             # 过滤掉重复匹配的情况
23             pos_mask = torch.where(mask_multi_gts, is_max_overlaps, pos_mask).f
24             # 得到更新后的每个anchor的mask 8400
25             fg_mask = pos_mask.sum(0)
26             # 找到每个anchor最匹配的GT 8400
27             target_gt_idx = pos_mask.argmax(0)
28             '''
29             target_gt_idx: 8400 为每个anchor最匹配的GT索引(包含了正负样本)
30             fg_mask: 8400 为每个anchor设置mask,用于区分正负样本
31             pos_mask: M x 8400 每张图像中每个GT设置正负样本的mask
32             '''
33             return target_gt_idx,fg_mask,pos_mask

```

<4>获得筛选样本的训练标签

前面<1><2><3>步的目的是为了获得正负样本的mask,即**fg_mask**、**pos_mask**、以及**target_gt_idx**,其中:

fg_mask: shape为 8400,其作用是服务于8400个anchors的, 对8400个anchors设置True和False, 代表该样本为正还是为负。

pos_mask: shape为M x 8400,其作用是服务于M个gtbox的, 表示了每个gtbox的正负样本, 其作用是对M个gtbox的负样本进行过滤。

target_gt_index: shape为 8400,其作用服务于8400个anchors,表示与每个anchor, M个gtbox中最匹配的gtbox的索引index。

因为网络输出的**预测值**分别为: **pred_scores(8400xcls_num)** + **pred_bboxes(8400x4)**

然后根据预处理阶段得到的**gt_labels(5x1)**,**gt_bboxes(5x4)** 结合

target_gt_index、fg_mask，得到最终同shape的训练标

签：**target_labels**(8400xcls_num)、**target_bboxes**(8400x4)

```
1 # ----- 根据Mask设置训练标签 ----- #
2 # target_labels : 8400 x cls_num
3 # target_bboxes : 8400 x 4
4 target_labels,target_bboxes = self.__get_train_targets(gt_labels,gt_bboxes,targ
```

```
1 def __get_train_targets(self,gt_labels,gt_bboxes,target_gt_idx,fg_mask):
2     '''
3         gt_labels: M x 1
4         gt_bboxes: M x 4
5         fg_mask   : 8400 每个anchor为正负样本0或1
6         target_gt_idx: 8400 每个anchor最匹配的GT索引(0~M)
7     '''
8     # gt_labels 拉直 M
9     gt_labels = gt_labels.long().flatten()
10    # 根据索引矩阵,获得cls 8400
11    target_labels = gt_labels[target_gt_idx]
12    # 同理bbox同样操作,
13    # 根据索引矩阵, 获得bbox 8400 x 4
14    target_bboxes = gt_bboxes[target_gt_idx]
15
16    # 类别转换为one-hot形式, 8400xcls_num
17    target_one_hot_labels = torch.zeros((target_labels.shape[0],self.num_cl
18                                         dtype=torch.int64,
19                                         device=target_labels.device)
20    # 赋值, 对应的类别位置置为1, 即one-hot形式 8400 x cls_num
21    target_one_hot_labels.scatter_(1,target_labels.unsqueeze(-1),1)
22
23    # 生成对应的mask, 用于过滤负样本 8400 -> 8400x1 -> 8400 x cls_num
24    fg_labels_mask = fg_mask.unsqueeze(-1).repeat(1,self.num_classes)
25
26    # 正负样本过滤
27    target_one_hot_labels = torch.where(fg_labels_mask>0,target_one_hot_lab
28
29    return target_one_hot_labels,target_bboxes
```


上面提到，动态分配策略，可以根据训练情况动态的调整权重值，所以在训练过程中，需要设置一个动态的权重，实现在训练过程中对欠佳的预测结果(困难样本)惩罚的目的。

提问：那上面提到的这个动态的权重该如何设置，才能有这样的效果呢？

灵感：既然上面在选取正样本过程中进行了scores与overlaps的计算，那是否可以巧妙得利用这个结果，添加上一定的转换进而作为动态的权重呢？

同时，通过添加动态权重，也就更加深了cls与box的关联性，避免出现cls预测准确度很高，iou很低的情况。

$$t = s^{\alpha} \times u^{\beta}$$

答案是可以的，代码如下：

```
1  # align_metric, overlaps均需要进行过滤
2  align_metrics *= pos_mask # M x 8400
3  overlaps *= pos_mask # M x 8400
4
5  # 找出每个GT的最大匹配值 M x 1
6  gt_max_metrics = align_metrics.amax(axis=-1, keepdim=True)
7  # 找到每个GT的最大CIOU值 M x 1
8  gt_max_overlaps = overlaps.amax(axis=-1, keepdim=True)
9  # 为类别one-hot标签添加惩罚项 M x 8400 -> 8400 -> 8400 x 1
10 # 通过M个GT与所有anchor的匹配值 x 每个GT与所有anchor最大IOU / 每个类别与所有anch
11 norm_align_metric = (align_metrics*gt_max_overlaps/(gt_max_metrics+self.eps)
12 # 8400 x cls_num，为类别添加惩罚项
13 target_labels = target_labels * norm_align_metric
14 b_target_labels[i] = target_labels
```