1. 创建anchor point

```
make_anchors(feats, self.stride, 0.5)
```

其构建的anchor point是代表着中心点，从0.5开始，步长为1.

```
tensor([[ 0.5000,  0.5000],
> special variables
> function variables
> H = tensor([[ 0.5000,  1.5000,  2.5000,  ..., 17.5000, 18.5000, 19.5000],
> T = tensor([[ 0.5000,  1.5000,  2.5000,  ..., 17.5000, 18.5000, 19.5000],
> data = tensor([[ 0.5000,  0.5000],
> device = device(type='cuda', index=0)
> dtype = torch.float16
  grad = None
  grad_fn = None
  imag = 'Traceback (most recent call last):\n  File "/media/dataStore/ReID
  is_cpu = False
  is_cuda = True
  is_ipu = False
  is_leaf = True
  is_maia = False
  is_meta = False
按住 Alt 键可切换到编辑器语言悬停
```

2. gt值预处理

将其从归一化的xywh转化为原始框的坐标xyxy

```python
def preprocess(self, targets, batch_size, scale_tensor):
    """Preprocesses the target counts and matches with the input batch size to output a tensor."""
    nl, ne = targets.shape
    if nl == 0:
        out = torch.zeros(batch_size, 0, ne - 1, device=self.device)
    else:
        i = targets[:, 0]  # image index
        _, counts = i.unique(return_counts=True)
        counts = counts.to(dtype=torch.int32)
        out = torch.zeros(batch_size, counts.max(), ne - 1, device=self.device)
        for j in range(batch_size):
            matches = i == j
            n = matches.sum()
            if n:
                out[j, :n] = targets[matches, 1:]
        out[..., 1:5] = xywh2xyxy(out[..., 1:5].mul_(scale_tensor))
    return out
```

3. 计算每个anchor预测的框

channel是64，为每个anchor的上下左右预测16个偏移概率，在[0,15]上积分，得到每个anchor的上下左右的偏移值。再在anchor point坐标上对这四个方向进行偏移，返回xyxy形式。

```python
def bbox_decode(self, anchor_points, pred_dist):
    """Decode predicted object bounding box coordinates from anchor points and distribution."""
    if self.use_dfl:
        b, a, c = pred_dist.shape  # batch, anchors, channels
        pred_dist = pred_dist.view(b, a, 4, c // 4).softmax(3).matmul(self.proj.type(pred_dist.dtype))
        # pred_dist = pred_dist.view(b, a, c // 4, 4).transpose(2,3).softmax(3).matmul(self.proj.type(pred_dist.dtype))
        # pred_dist = (pred_dist.view(b, a, c // 4, 4).softmax(2) * self.proj.type(pred_dist.dtype).view(1, 1, -1, 1)).sum(2)
    return dist2bbox(pred_dist, anchor_points, xywh=False)
```

```python
def dist2bbox(distance, anchor_points, xywh=True, dim=-1):
    """Transform distance(ltrb) to box(xywh or xyxy)."""
    lt, rb = distance.chunk(2, dim)
    x1y1 = anchor_points - lt
    x2y2 = anchor_points + rb
    if xywh:
        c_xy = (x1y1 + x2y2) / 2
        wh = x2y2 - x1y1
        return torch.cat((c_xy, wh), dim)  # xywh bbox
    return torch.cat((x1y1, x2y2), dim)  # xyxy bbox
```

4. gt框分配

为每个gt框分配topk个预测框来进行损失计算。

```python
mask_pos, align_metric, overlaps = self.get_pos_mask(
    pd_scores, pd_bboxes, gt_labels, gt_bboxes, anc_points, mask_gt
)

target_gt_idx, fg_mask, mask_pos = self.select_highest_overlaps(mask_pos, overlaps, self.n_max_boxes)

# Assigned target
target_labels, target_bboxes, target_scores = self.get_targets(gt_labels, gt_bboxes, target_gt_idx, fg_mask)

# Normalize
align_metric *= mask_pos
pos_align_metrics = align_metric.amax(dim=-1, keepdim=True)  # b, max_num_obj
pos_overlaps = (overlaps * mask_pos).amax(dim=-1, keepdim=True)  # b, max_num_obj
norm_align_metric = (align_metric * pos_overlaps / (pos_align_metrics + self.eps)).amax(-2).unsqueeze(-1)
target_scores = target_scores * norm_align_metric

return target_labels, target_bboxes, target_scores, fg_mask.bool(), target_gt_idx
```

1. 只保留存在于gt框内的anchor point
2. 计算每个anchor point与gt框的iou,记为overlap。再计算分类得分和iou的加权和作为分配得分align_metric
3. 为每个gt框取topk个iou最大的anchor point

```python
def get_pos_mask(self, pd_scores, pd_bboxes, gt_labels, gt_bboxes, anc_points, mask_gt):
    """Get in_gts mask, (b, max_num_obj, h*w)."""
    mask_in_gts = self.select_candidates_in_gts(anc_points, gt_bboxes)
    # Get anchor_align metric, (b, max_num_obj, h*w)
    align_metric, overlaps = self.get_box_metrics(pd_scores, pd_bboxes, gt_labels, gt_bboxes, mask_in_gts * mask_gt)
    # Get topk_metric mask, (b, max_num_obj, h*w)
    mask_topk = self.select_topk_candidates(align_metric, topk_mask=mask_gt.expand(-1, -1, self.topk).bool())
    # Merge all mask to a final mask, (b, max_num_obj, h*w)
    mask_pos = mask_topk * mask_in_gts * mask_gt

    return mask_pos, align_metric, overlaps
```

```python
@staticmethod
def select_candidates_in_gts(xy_centers, gt_bboxes, eps=1e-9):
    """
    Select positive anchor centers within ground truth bounding boxes.

    Args:
        xy_centers (torch.Tensor): Anchor center coordinates, shape (h*w, 2).
        gt_bboxes (torch.Tensor): Ground truth bounding boxes, shape (b, n_boxes, 4).
        eps (float, optional): Small value for numerical stability. Defaults to 1e-9.

    Returns:
        (torch.Tensor): Boolean mask of positive anchors, shape (b, n_boxes, h*w).

    Note:
        b: batch size, n_boxes: number of ground truth boxes, h: height, w: width.
        Bounding box format: [x_min, y_min, x_max, y_max].
    """
    n_anchors = xy_centers.shape[0]
    bs, n_boxes, _ = gt_bboxes.shape
    lt, rb = gt_bboxes.view(-1, 1, 4).chunk(2, 2)  # left-top, right-bottom
    bbox_deltas = torch.cat((xy_centers[None] - lt, rb - xy_centers[None]), dim=2).view(bs, n_boxes, n_anchors, -1)
    # return (bbox_deltas.min(3)[0] > eps).to(gt_bboxes.dtype)
    return bbox_deltas.amin(3).gt_(eps)
```

```python
def get_box_metrics(self, pd_scores, pd_bboxes, gt_labels, gt_bboxes, mask_gt):
    """Compute alignment metric given predicted and ground truth bounding boxes."""
    na = pd_bboxes.shape[-2]
    mask_gt = mask_gt.bool()  # b, max_num_obj, h*w
    overlaps = torch.zeros([self.bs, self.n_max_boxes, na], dtype=pd_bboxes.dtype, device=pd_bboxes.device)
    bbox_scores = torch.zeros([self.bs, self.n_max_boxes, na], dtype=pd_scores.dtype, device=pd_scores.device)

    ind = torch.zeros([2, self.bs, self.n_max_boxes], dtype=torch.long)  # 2, b, max_num_obj
    ind[0] = torch.arange(end=self.bs).view(-1, 1).expand(-1, self.n_max_boxes)  # b, max_num_obj
    ind[1] = gt_labels.squeeze(-1)  # b, max_num_obj
    # Get the scores of each grid for each gt cls
    bbox_scores[mask_gt] = pd_scores[ind[0], :, ind[1]][mask_gt]  # b, max_num_obj, h*w

    # (b, max_num_obj, 1, 4), (b, 1, h*w, 4)
    pd_boxes = pd_bboxes.unsqueeze(1).expand(-1, self.n_max_boxes, -1, -1)[mask_gt]
    gt_boxes = gt_bboxes.unsqueeze(2).expand(-1, -1, na, -1)[mask_gt]
    overlaps[mask_gt] = self.iou_calculation(gt_boxes, pd_boxes)

    align_metric = bbox_scores.pow(self.alpha) * overlaps.pow(self.beta)
    return align_metric, overlaps
```

```python
def select_topk_candidates(self, metrics, largest=True, topk_mask=None):
    """
    Select the top-k candidates based on the given metrics.

    Args:
        metrics (Tensor): A tensor of shape (b, max_num_obj, h*w), where b is the batch size,
                          max_num_obj is the maximum number of objects, and h*w represents the
                          total number of anchor points.
        largest (bool): If True, select the largest values; otherwise, select the smallest values.
        topk_mask (Tensor): An optional boolean tensor of shape (b, max_num_obj, topk), where
                            topk is the number of top candidates to consider. If not provided,
                            the top-k values are automatically computed based on the given metrics.

    Returns:
        (Tensor): A tensor of shape (b, max_num_obj, h*w) containing the selected top-k candidates.
    """
    # (b, max_num_obj, topk)
    topk_metrics, topk_idxs = torch.topk(metrics, self.topk, dim=-1, largest=largest)
    if topk_mask is None:
        topk_mask = (topk_metrics.max(-1, keepdim=True)[0] > self.eps).expand_as(topk_idxs)
    # (b, max_num_obj, topk)
    topk_idxs.masked_fill_(~topk_mask, 0)

    # (b, max_num_obj, topk, h*w) -> (b, max_num_obj, h*w)
    count_tensor = torch.zeros(metrics.shape, dtype=torch.int8, device=topk_idxs.device)
    ones = torch.ones_like(topk_idxs[:, :, :1], dtype=torch.int8, device=topk_idxs.device)
    for k in range(self.topk):
        # Expand topk_idxs for each value of k and add 1 at the specified positions
        count_tensor.scatter_add_(-1, topk_idxs[:, :, k : k + 1], ones)
    # count_tensor.scatter_add_(-1, topk_idxs, torch.ones_like(topk_idxs, dtype=torch.int8, device=topk_idxs.device))
    # Filter invalid bboxes
    count_tensor.masked_fill_(count_tensor > 1, 0)

    return count_tensor.to(metrics.dtype)
```

4. 如果有预测的anchor框匹配了多个gt框，则只保留iou最大的那个匹配结果。

```python
@staticmethod
def select_highest_overlaps(mask_pos, overlaps, n_max_boxes):
    """
    Select anchor boxes with highest IoU when assigned to multiple ground truths.

    Args:
        mask_pos (torch.Tensor): Positive mask, shape (b, n_max_boxes, h*w).
        overlaps (torch.Tensor): IoU overlaps, shape (b, n_max_boxes, h*w).
        n_max_boxes (int): Maximum number of ground truth boxes.

    Returns:
        target_gt_idx (torch.Tensor): Indices of assigned ground truths, shape (b, h*w).
        fg_mask (torch.Tensor): Foreground mask, shape (b, h*w).
        mask_pos (torch.Tensor): Updated positive mask, shape (b, n_max_boxes, h*w).

    Note:
        b: batch size, h: height, w: width.
    """
    # Convert (b, n_max_boxes, h*w) -> (b, h*w)
    fg_mask = mask_pos.sum(-2)
    if fg_mask.max() > 1:  # one anchor is assigned to multiple gt_bboxes
        mask_multi_gts = (fg_mask.unsqueeze(1) > 1).expand(-1, n_max_boxes, -1)  # (b, n_max_boxes, h*w)
        max_overlaps_idx = overlaps.argmax(1)  # (b, h*w)

        is_max_overlaps = torch.zeros(mask_pos.shape, dtype=mask_pos.dtype, device=mask_pos.device)
        is_max_overlaps.scatter_(1, max_overlaps_idx.unsqueeze(1), 1)

        mask_pos = torch.where(mask_multi_gts, is_max_overlaps, mask_pos).float()  # (b, n_max_boxes, h*w)
        fg_mask = mask_pos.sum(-2)
    # Find each grid serve which gt(index)
    target_gt_idx = mask_pos.argmax(-2)  # (b, h*w)
    return target_gt_idx, fg_mask, mask_pos
```

5. 根据上述计算结果，进行gt和anchor point预测框的分配。

```python
def get_targets(self, gt_labels, gt_bboxes, target_gt_idx, fg_mask):
    """
    Compute target labels, target bounding boxes, and target scores for the positive anchor points.

    Args:
        gt_labels (Tensor): Ground truth labels of shape (b, max_num_obj, 1), where b is the
                            batch size and max_num_obj is the maximum number of objects.
        gt_bboxes (Tensor): Ground truth bounding boxes of shape (b, max_num_obj, 4).
        target_gt_idx (Tensor): Indices of the assigned ground truth objects for positive
                                anchor points, with shape (b, h*w), where h*w is the total
                                number of anchor points.
        fg_mask (Tensor): A boolean tensor of shape (b, h*w) indicating the positive
                          (foreground) anchor points.

    Returns:
        (Tuple[Tensor, Tensor, Tensor]): A tuple containing the following tensors:
            - target_labels (Tensor): Shape (b, h*w), containing the target labels for
                                      positive anchor points.
            - target_bboxes (Tensor): Shape (b, h*w, 4), containing the target bounding boxes
                                      for positive anchor points.
            - target_scores (Tensor): Shape (b, h*w, num_classes), containing the target scores
                                      for positive anchor points, where num_classes is the number
                                      of object classes.
    """
    # Assigned target labels, (b, 1)
    batch_ind = torch.arange(end=self.bs, dtype=torch.int64, device=gt_labels.device)[..., None]
    target_gt_idx = target_gt_idx + batch_ind * self.n_max_boxes  # (b, h*w)
    target_labels = gt_labels.long().flatten()[target_gt_idx]  # (b, h*w)

    # Assigned target boxes, (b, max_num_obj, 4) -> (b, h*w, 4)
    target_bboxes = gt_bboxes.view(-1, gt_bboxes.shape[-1])[target_gt_idx]

    # Assigned target scores
    target_labels.clamp_(0)

    # 10x faster than F.one_hot()
    target_scores = torch.zeros(
        (target_labels.shape[0], target_labels.shape[1], self.num_classes),
        dtype=torch.int64,
        device=target_labels.device,
    )  # (b, h*w, 80)
    target_scores.scatter_(2, target_labels.unsqueeze(-1), 1)

    fg_scores_mask = fg_mask[:, :, None].repeat(1, 1, self.num_classes)  # (b, h*w, 80)
    target_scores = torch.where(fg_scores_mask > 0, target_scores, 0)

    return target_labels, target_bboxes, target_scores
```

5. 计算损失

分类损失使用bce损失 回归损失使用CIoU损失和DFL损失

CIoU损失

$$L_{CIoU} = 1 - IoU + \frac{\rho^2(b, b^g)}{c^2} + \alpha v$$

1. IoU 是交并比（Intersection over Union）。

2. ρ(b, b^g) 是两个边界框中心点之间的欧几里得距离。

3. c 是最小包围框（convex hull）的对角线长度。

4. v 是衡量长宽比一致性的度量：计算的是预测框和gt框的宽高比率，如果二者宽高比相差较大，则v值大。

$$v = \frac{4}{\pi^2}\left(\tan^{-1}\frac{w^g}{h^g} - \tan^{-1}\frac{w}{h}\right)^2$$

$$\alpha = \frac{v}{(1-IoU)+v}$$

DFL损失 先将gt框转换为对应anchor朝上下左右四个方向的偏移。将偏移量限制在设定的reg_max范围内。 然后对偏移量左右取整。比如说第一个gt框的上偏移量计算出来是5.3，那么tl=5,tr=6,wl=0.7,wr=0.3。

```python
class DFLoss(nn.Module):
    """Criterion class for computing DFL losses during training."""

    def __init__(self, reg_max=16) -> None:
        """Initialize the DFL module."""
        super().__init__()
        self.reg_max = reg_max

    def __call__(self, pred_dist, target): self = DFLoss(), pred_dist = tensor([[ 6.5156,  6.
        """
        Return sum of left and right DFL losses.

        Distribution Focal Loss (DFL) proposed in Generalized Focal Loss
        https://ieeexplore.ieee.org/document/9792391
        """
        target = target.clamp_(0, self.reg_max - 1 - 0.01) target = tensor([[1.0292, 0.5915,
        tl = target.long()  # target left
        tr = tl + 1  # target right
        wl = tr - target  # weight left
        wr = 1 - wl  # weight right
        return (
            F.cross_entropy(pred_dist, tl.view(-1), reduction="none").view(tl.shape) * wl
            + F.cross_entropy(pred_dist, tr.view(-1), reduction="none").view(tl.shape) * wr
        ).mean(-1, keepdim=True)
```