如何将Nanodet修改成为四点模型

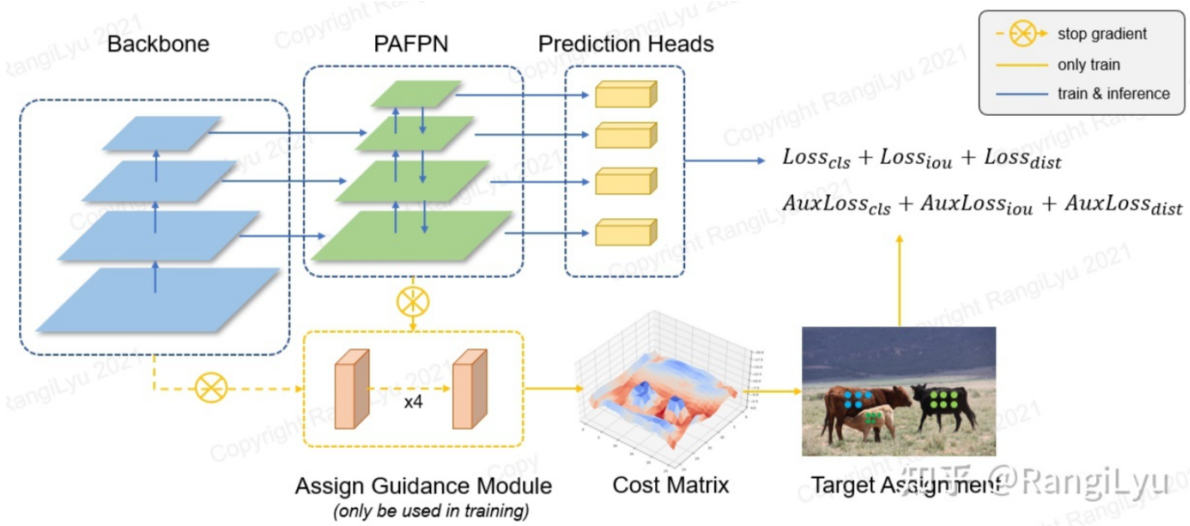
by 

[更新于 2023 . 2 . 4]

> **本文档只包括了训练部分，不包括推理的所有部分。**随后会出推理部分（包括pytorch推理以及openvino推理）。
>
> 不会特别详细的介绍目标检测，对于在RoboMaster赛场上应用，应该是够了。
>
> 我们团队自己的Nanodet四点模型仓库：https://github.com/HUSTLYRM/Nanodet
>
> github上的一个nanodet_keypoint仓库：https://github.com/1248289414/nanodet_keypoint
>
> 跃鹿战队关于原始Nanodet-plus的注释：https://blog.csdn.net/NeoZng/article/details/123299419
>
> 在此，感谢以上开源部分。【致敬】
>
> 准确的说，是修改Nanodet-plus网络，较Nanodet有了一定的提升。
>
> Nanodet-plus作者：NanoDet-Plus总结了上一代模型在标签分配、模型结构以及训练策略上的不足，提出了**AGM**和**DSLA**以及**Ghost-PAN**模块，并全面改进了训练策略，更加易于训练！同时也全面修改了模型部署时的输出方式，简化了结构，并提供了**ncnn、MNN、OpenVINO以及安卓端的Demo**，每个demo下都有**非常详细的教程**指导大家上手。

零、首先看Nanodet-plus结构

> 由于本人的学识有限并且理解不深入，只说明一些关键的部分，说的不会很详细，请见谅。
>
> Nanodet-plus作者的文章 https://zhuanlan.zhihu.com/p/449912627
>
> 建议阅读一下作者自己的介绍的文章，有助于理解网络。
>
> 本身不需要修改Backbone和Neck (PAFPN) 部分，这两部分负责特征提取和特征融合，并且这两部分有大厂用NAS搜出来的效果很好的Backbone和Neck，在训练中直接获取预训练好的权重就可以了（不需要自己做）。
>
> **重点** 就是修改 `head` 部分。



学习一下 标签分配、损失函数Focal Loss，了解作者的AGM模块。

关于标签分配、Focal Loss比较难理解（我自己也没有很理解清除），需要多花一点时间。

> 1. 查找到的资料：
>
> 标签分配划分成正负样本，正样本就是能和GT进行匹配的anchor，负样本就是不能和GT进行匹配的anchor，正样本可以和GT计算分类、回归、置信度损失，负样本只能计算置信度损失
>
> [yolov5的资料，看一下目标检测理论上的流程]
>
> 2. 理解一下：
>
> - 仅mask矩阵中对应位置为true的预测框（个人认为可以理解为正样本），需要计算矩形框损失；
> - 仅mask矩阵中对应位置为true的预测框（个人认为可以理解为正样本），需要计算分类损失；
> - 所有预测框都需要计算置信度损失，但是mask为true的预测框与mask为false的预测框的置信度标签值不一样。
>
> 3. Focal Loss：
>
> $$\mathrm{FL}(p_t) = -\alpha_t(1-p_t)^{\gamma}\log(p_t)$$
>
> [Focal Loss损失函数公式的理解]
>
> [剖析Focal Loss损失函数]
>
> 在那些大量未匹配到目标的负样本中，大部分都是简单易分的负样本，这些简单的负样本对网络训练起不到太大的作用，但是由于数量太多，会淹没掉少量但有助于训练的样本。
>
> 注意：Focal Loss损失函数容易受到噪声的干扰。也就是说训练集中标注的信息尽量不要出现错误的情况，否则Focal Loss损失函数就会针对那些标注错误的样本进行重点学习，使得模型的训练效果越来越差。因为根据Focal Loss损失函数的原理，它会重点关注困难样本，而此时如果我们将某个样本标注错误，那么该样本对于网络来说就是一个"困难样本"，所以Focal Loss损失函数就会重点学习这些"困难样本"，导致模型训练效果越来越差。
>
> 了解有QFL、DFL，其中QFL用来表示分类的损失函数，DFL表示回归的损失函数（用到了积分）。
>
> 4. Anchor-free

> anchor-based基于预设的若干固定尺度和长宽比的anchor生成预测框，anchor-free基于点生成预测框。
>
> anchor-based可以参考yolo系列的前几个版本，现在anchor-free比较流行。
>
> Nanodet就是一个典型的Anchor-free。

一、config文件的修改（宏观上）

`config`文件指明了所有的参数，包括了网络的框架，损失函数，数据集的相关配置，训练超参数等。

所以，通过查看对应的`config`文件（`yml`格式），就可以确定网络、数据集等内容。

config中的参数跟具体类的构造函数init函数中的参数对应，实现时通过解析参数然后调用构造函数从而构造网络的结构，以及数据集类等等。这一点和很多流行的网络结构一致，都采用了配置文件的形式。

具体修改的地方，可以查看下面的yml文件。

> 对于不清楚含义但是不影响训练、验证的参数，保持默认即可。
>
> config/nanodet-xxx.yml的修改要结合情况更改即可，很多网络都采用了这种配置文件的格式，这样很容易替换backbone、fpn'等。

```yaml
# config/

save_dir: /home/zengyj/nanodet/nanodet-main/workspace/nanodetfp_416 # 存放训练结果的路径(包括了训练的日志以及保存的模型)

model:
  weight_averager:                  # 不太清楚，默认就好了
    name: ExpMovingAverager
    decay: 0.9998
  arch:
    name: NanoDetPlus               # NanoDetPlus
    detach_epoch: 10
    backbone:
      name: ShuffleNetV2            # 默认使用shuffleNetV2
      model_size: 1.0x              # 模型缩放系数，更大的模型就是相应地扩大各层feature map的大小
      out_stages: [2,3,4]           # backbone中输出特征到FPN的stage
      activation: LeakyReLU         # 激活函数
    fpn:
      name: GhostPAN                # 用ghostNet的模块对不同特征层进行融合
      in_channels: [116, 232, 464] # 输入fpn的geature map 尺寸
      out_channels: 96
      kernel_size: 5                # 卷积核大小
      num_extra_level: 1
      use_depthwise: True           # 使用深度可分离卷积
      activation: LeakyReLU         # 激活函数
    head:
      name: NanoDetPlusHead         # 检测头
```

```yaml
      num_classes: 36               # 类别数
      input_channel: 96             # 输入通道数
      feat_channels: 96             # 特征通道数
      stacked_convs: 2              # head的卷积层数
      kernel_size: 5                # 卷积核的大小
      strides: [8, 16, 32, 64]      # 四个头，分别对应了不同尺度特征的检测，不同head检测时的下采样倍数
      activation: LeakyReLU         # 激活函数
      reg_max: 7                    # 用于dfl的参数，head的回归分支会预测框的分布，即用回归reg_max+1个离散的几个值来表示一个分布
      norm_cfg:
        type: BN                    # head选用batch norm 进行归一化操作
      loss:
        loss_qfl:
          name: QualityFocalLoss    # loss继承了nanodet，使用GFL，并且这些loss有不同的权重
          use_sigmoid: True
          beta: 2.0
          loss_weight: 1.0
        loss_dfl:
          name: DistributionFocalLoss
          loss_weight: 0.25
        loss_bbox:                   # bbox的损失函数 giou
          name: GIoULoss
          loss_weight: 2.0
    # Auxiliary head, only use in training time.
    # 新增的辅助模块，（常规检测头，表达能力更强，只在训练的时候用）
    aux_head:
      name: SimpleConvHead
      num_classes: 36             # 类别
      input_channel: 192          # 输入通道数
      feat_channels: 192
      stacked_convs: 4            # 四层卷积
      strides: [8, 16, 32, 64]    # 对应四个头
      activation: LeakyReLU
      reg_max: 7

# VOC格式数据
class_names: &class_names ['B_G', 'B_1', 'B_2', 'B_3', 'B_4', 'B_5', 'B_O', 'B_Bs', 'B_Bb',
                           'R_G', 'R_1', 'R_2', 'R_3', 'R_4', 'R_5', 'R_O', 'R_Bs', 'R_Bb',
                           'N_G', 'N_1', 'N_2', 'N_3', 'N_4', 'N_5', 'N_O', 'N_Bs', 'N_Bb',
                           'P_G', 'P_1', 'P_2', 'P_3', 'P_4', 'P_5', 'P_O', 'P_Bs', 'P_Bb' ]

data:
  train:
    name: XMLDataset
    class_names: *class_names
    img_path: /home/zengyj/nanodet/swu_dataset/train/image
    ann_path: /home/zengyj/nanodet/swu_dataset/train/xml
    input_size: [416,416]      #[w,h]
    keep_ratio: False    # 注意：使用了keep_ratio时，网络单个feat尺寸也会发生一定的变化（例如40x40变成了32x40（图片尺寸1024x1280））
    pipeline:                                # 包含了数据增强的部分
      perspective: 0.0
```

```yaml
      scale: [0.6, 1.4]
      stretch: [[1, 1], [1, 1]]
      rotation: 0
      shear: 0
      translate: 0.2
      flip: 0.5
      brightness: 0.2
      contrast: [0.8, 1.2]
      saturation: [0.8, 1.2]
      normalize: [[103.53, 116.28, 123.675], [57.375, 57.12, 58.395]]
  val:
    name: XMLDataset
    class_names: *class_names
    img_path: /home/zengyj/nanodet/swu_dataset/test/image
    ann_path: /home/zengyj/nanodet/swu_dataset/test/xml
    input_size: [416,416]         #[w,h]
    keep_ratio: False
    pipeline:
      normalize: [[103.53, 116.28, 123.675], [57.375, 57.12, 58.395]]

device:
  gpu_ids: [0]                    # Set like [0, 1, 2, 3] if you have multi-GPUs
  workers_per_gpu: 16             # TODO  命令行中给出的提示，支持16个进程处理，建议num_workers修改成16，根据提示修改的
  batchsize_per_gpu: 32           # TODO 这个根据GPU的显存容量结合实际选择

schedule:
  # resume:                                                         # 恢复训练时需要设置，进行新的训练时，不要设置
  #   load_model: /home/zengyj/nanodet/nanodet-main/workspace/nanodet-plus-m_416/model_last.ckpt
  optimizer:
    name: AdamW                                                     # 优化器，AdamW可以比较快的收敛，而且基本不需要调超参数
    lr: 0.001
    weight_decay: 0.05
  warmup:
    name: linear
    steps: 500
    ratio: 0.0001
  total_epochs: 600                                                 # 总训练轮次
  lr_schedule:                                                      # 学习策略
    name: CosineAnnealingLR
    T_max: 300
    eta_min: 0.00005
  val_intervals: 10                                                 # 每10轮训练，进行一次验证
grad_clip: 35
evaluator:
  name: CocoDetectionEvaluator
  save_key: mAP
log:
  interval: 10
```

二、数据集加载部分修改+warp

XMLDataset继承了CocoDataset，CocoDataset继承了BaseDataset。

> 在Nanodet代码中，将xml格式的数据转换成coco格式，在训练等过程中最终投入的是coco格式，在XMLDataset中对将xml转换成coco格式，在其内容验证部分也是用了coco的验证方式（cocodetection）。
>
> 这部分还包括了数据的预处理部分，即代码中的pipeline，包括将图片缩放成指定的大小。
>
> 以下代码的修改部分在注释中都增加了`TOOD`标签，方便确定修改了哪些地方。
>
> 这部分比较容易，仿照其他部分读取补充就可以完成。

我们采用的是xml格式（也可以根据情况修改成为coco格式），需要修改XMLDataset、CocoDataset。

修改CocoDataset是因为XMLDataset继承了XMLDataset，也就继承了对应的方法，可以在XMLDataset中覆盖相应的方法（了解过面向对象即可），也可以在CocoDataset中直接修改，这里是选择了后者。

新增了从xml中读取points的部分，包括在annotations中。

注释中TODO表明了修改的部分。

`nanodet/data/dataset/xml_dataset.py:`

```
# 数据处理相关内容，XMLDataset，继承了CocoDataset
class XMLDataset(CocoDataset):
    def __init__(self, class_names, **kwargs):
        self.class_names = class_names
        super(XMLDataset, self).__init__(**kwargs)

    # 将xml格式转换成coco格式
    def xml_to_coco(self, ann_path):
        """
        convert xml annotations to coco_api
        :param ann_path:
        :return:
        """
        logging.info("loading annotations into memory...")
        tic = time.time()
        ann_file_names = get_file_list(ann_path, type=".xml")
        logging.info("Found {} annotation
files.".format(len(ann_file_names)))
        image_info = []
        categories = []
        annotations = []
        for idx, supercat in enumerate(self.class_names):
            categories.append(
                {"supercategory": supercat, "id": idx + 1, "name":
supercat}
            )
        ann_id = 1
        for idx, xml_name in enumerate(ann_file_names):     # 将所有数据转换
            tree = ET.parse(os.path.join(ann_path, xml_name))
            root = tree.getroot()
```

```
            file_name = root.find("filename").text
            width = int(root.find("size").find("width").text)
            height = int(root.find("size").find("height").text)
            info = {                                        # 组装图片的info
                "file_name": file_name,
                "height": height,
                "width": width,
                "id": idx + 1,
            }
            image_info.append(info)
            for _object in root.findall("object"):          # 每张图片的所有object加入到数据
                category = _object.find("name").text
                if category not in self.class_names:
                    logging.warning(
                        "WARNING! {} is not in class_names! "
                        "Pass this box annotation.".format(category)
                    )
                    continue
                for cat in categories:
                    if category == cat["name"]:
                        cat_id = cat["id"]
                xmin = int(_object.find("bndbox").find("xmin").text) # voc 格式bbox两个点，左上点和右下点
                ymin = int(_object.find("bndbox").find("ymin").text)
                xmax = int(_object.find("bndbox").find("xmax").text)
                ymax = int(_object.find("bndbox").find("ymax").text)
                w = xmax - xmin
# coco 格式需要的是x, y, w, h, 在这里进行处理
                h = ymax - ymin

                # TODO 新增points部分[x1, y1, x2, y2, x3, y3, x4, y4]
                x1 = int(_object.find("points").find("x1").text)
                y1 = int(_object.find("points").find("y1").text)
                x2 = int(_object.find("points").find("x2").text)
                y2 = int(_object.find("points").find("y2").text)
                x3 = int(_object.find("points").find("x3").text)
                y3 = int(_object.find("points").find("y3").text)
                x4 = int(_object.find("points").find("x4").text)
                y4 = int(_object.find("points").find("y4").text)

                if w < 0 or h < 0:  # 不合适的数据
                    logging.warning(
                        "WARNING! Find error data in file {}! Box w and "
                        "h should > 0. Pass this box annotation.".format(xml_name)
                    )
                    continue

                coco_box = [max(xmin, 0), max(ymin, 0), min(w, width), min(h, height)]  # 组装成coco数据bbox

                points = [x1, y1, x2, y2, x3, y3, x4, y4]       #     TODO 新增points部分数据

                ann = {
                    "image_id": idx + 1,
```

```
                    "bbox": coco_box,        # 重点关注  coco_box
                    "points": points,        # 重点关注                    TODO 新增points部分
                    "category_id": cat_id,  # 重点关注  类别
                    "iscrowd": 0,
                    "id": ann_id,
                    "area": coco_box[2] * coco_box[3],      # w*h   面积
                }

                annotations.append(ann)
                ann_id += 1

        # 组装成一个数据集的coco_dict, 图片info的列表, 类别列表, annotations列表
        coco_dict = {
            "images": image_info,
            "categories": categories,
            "annotations": annotations,         # 这里已经包括了前面的points部分数据
        }
        logging.info(
            "Load {} xml files and {} boxes".format(len(image_info), len(annotations))
        )
        logging.info("Done (t={:0.2f}s)".format(time.time() - tic))

        # 将组装好的coco_dict数据, 返回
        return coco_dict
```

nanodet/data/dataset/coco.py:

```
# 获取图片的annotation，每个图片的重点标注的信息， 重点关注bbox、category_id（points是我自己有其他需求新增的信息）
    def get_img_annotation(self, idx):
        """
        load per image annotation
        :param idx: index in dataloader
        :return: annotation dict
        """
        img_id = self.img_ids[idx]
        ann_ids = self.coco_api.getAnnIds([img_id])
        anns = self.coco_api.loadAnns(ann_ids)
        gt_bboxes = []          # gt就是ground truth，可以理解为就是正确的，就是标注的正样本，方便后续计算损失函数等
        gt_labels = []
        gt_points = []          # TODO 新增读取points部分，存储gt_points

        gt_bboxes_ignore = []
        if self.use_instance_mask:
            gt_masks = []
        if self.use_keypoint:
            gt_keypoints = []

        for ann in anns:            # 对于每一个ann都处理
            x1, y1, w, h = ann["bbox"]                  # 这里说明，coco格式的 ann文件中存放的是 左上角坐标以及宽高
            if ann["area"] <= 0 or w < 1 or h < 1:
                continue
```

```
            if ann["category_id"] not in self.cat_ids:
                continue
            bbox = [x1, y1, x1 + w, y1 + h]              # 将bbox更新成左上角、右下角，  bbox存放的是左上点，右下点
            if ann.get("iscrowd", False) or ann.get("ignore", False):
                gt_bboxes_ignore.append(bbox)
            else:
                gt_bboxes.append(bbox)                           # 将bbox加入到gt_bboxes中

                x1, y1, x2, y2, x3, y3, x4, y4 = ann["points"]  # TODO 新增 points 的 部分，  在annotation中新增points
                points = [x1, y1, x2, y2, x3, y3, x4, y4]
                gt_points.append(points)                         # TODO 将points加入到gt_points中

                gt_labels.append(self.cat2label[ann["category_id"]])    # 将类别id加入到gt_labels中
                if self.use_instance_mask:
                    gt_masks.append(self.coco_api.annToMask(ann))
                if self.use_keypoint:
                    gt_keypoints.append(ann["keypoints"])

        if gt_bboxes:       # 列表  转换成  numpy形式的数组
            gt_bboxes = np.array(gt_bboxes, dtype=np.float32)
            gt_points = np.array(gt_points, dtype=np.float32)             # TODO 新增gt_points
            gt_labels = np.array(gt_labels, dtype=np.int64)
        else:
            gt_bboxes = np.zeros((0, 4), dtype=np.float32)
            gt_points = np.zeros((0, 8), dtype=np.float32)                # TODO 新增gt_points
            gt_labels = np.array([], dtype=np.int64)

        if gt_bboxes_ignore:
            gt_bboxes_ignore = np.array(gt_bboxes_ignore, dtype=np.float32)
        else:
            gt_bboxes_ignore = np.zeros((0, 4), dtype=np.float32)

        # 要重要的ann组装成字典格式
        annotation = dict(
            bboxes=gt_bboxes, labels=gt_labels, bboxes_ignore=gt_bboxes_ignore, points=gt_points    # TODO 新增gt_points
        )

        if self.use_instance_mask:
            annotation["masks"] = gt_masks
        if self.use_keypoint:
            if gt_keypoints:
                annotation["keypoints"] = np.array(gt_keypoints, dtype=np.float32)
            else:
                annotation["keypoints"] = np.zeros((0, 51), dtype=np.float32)

        return annotation   # 将得到的重要信息返回

```

```
    # `训练时` 的关键调用，      根据idx索引来获取训练数据（将重要信息组装成了meta（dict类型））
    def get_train_data(self, idx):
        """
        Load image and annotation
        :param idx:
        :return: meta-data (a dict containing image, annotation and other information)
        """
        img_info = self.get_per_img_info(idx)                # 根据idx图片的信息
        file_name = img_info["file_name"]
        image_path = os.path.join(self.img_path, file_name) # 利用join生成图片的具体路径
        img = cv2.imread(image_path)                         # 读取图片
        if img is None:
            print("image {} read failed.".format(image_path))
            raise FileNotFoundError("Cant load image! Please check image path!")

        ann = self.get_img_annotation(idx)                   # 获取idx索引对应的annotaion
        # meta很重要，基本就是用来计算损失函数等等的 ground truth 部分
        # TODO meta增加gt_points，组装成meta数据
        meta = dict(
            img=img,
            img_info=img_info,
            gt_bboxes=ann["bboxes"],
            gt_labels=ann["labels"],
            gt_points=ann["points"],
            gt_bboxes_ignore=ann["bboxes_ignore"],
        )

        if self.use_instance_mask:
            meta["gt_masks"] = ann["masks"]
        if self.use_keypoint:
            meta["gt_keypoints"] = ann["keypoints"]
        input_size = self.input_size
        if self.multi_scale:
            input_size = self.get_random_size(self.multi_scale, input_size)
        meta = self.pipeline(self, meta, input_size)   # 这里对其进行了处理，最终是调用了warp.py里面的函数，对meta中的数据进行了resize

        # print(meta)
        meta["img"] = torch.from_numpy(meta["img"].transpose(2, 0, 1))
        return meta
```

在get_train_data中，最终调用了pipeline，对读取的数据进行了预处理，在pipeline实现时采取了多种数据增强的方式，利用矩阵变换对数据进行处理（包括在类ShapeTransform中），最终使用warp将图像缩放成输入的大小。

在warp.py文件中就不详细说修改的gt_points部分了，这一部分的修改和数据集的读入类似，也可以看仓库（同样使用TODO标签标注）。

`nanodet/data/dataset/transform/warp.py:`

```python
def warp_boxes(boxes, M, width, height):        # 通过矩阵映射将原图上的点映射到网络输入input_size的图片上
    n = len(boxes)  # 2125:  可以认为是锚点的个数（预测目标的个数）
    if n:
        # warp points
        xy = np.ones((n * 4, 3))
        xy[:, :2] = boxes[:, [0, 1, 2, 3, 0, 3, 2, 1]].reshape(     # x1y1, x2y2, x1y2, x2y1
            n * 4, 2
        )
        xy = xy @ M.T                                               # transform 每个点都映射到原来的图片上
        xy = (xy[:, :2] / xy[:, 2:3]).reshape(n, 8)         # rescale
        # create new boxes
        x = xy[:, [0, 2, 4, 6]]                                     # 所有点的横纵坐标
        y = xy[:, [1, 3, 5, 7]]
        xy = np.concatenate((x.min(1), y.min(1), x.max(1), y.max(1))).reshape(4, n).T       # 再把所有点找出xmin,ymin
        # clip boxes
        xy[:, [0, 2]] = xy[:, [0, 2]].clip(0, width)
        xy[:, [1, 3]] = xy[:, [1, 3]].clip(0, height)
        return xy.astype(np.float32)
    else:
        return boxes

# 仿照warp_boxes编写四点的warp，这里参考了nanodet_keypoints仓库
def warp_points(keypoints, M, width, height):
    n = len(keypoints)
    if n:
        # warp points
        xy = np.ones((n * 4, 3))
        # x1y1, x2y2, x1y2, x2y1
        xy[:, :2] = keypoints.reshape(n * 4, 2)
        xy = xy @ M.T  # transform
        xy = (xy[:, :2] / xy[:, 2:3]).reshape(n, 8)  # rescale
        # clip
        xy[:, [0, 2, 4, 6]] = xy[:, [0, 2, 4, 6]].clip(0, width)
        xy[:, [1, 3, 5 ,7]] = xy[:, [1, 3, 5 ,7]].clip(0, height)
        return xy.astype(np.float32)
    else:
        return keypoints
```

到这里，数据读取的部分就修改完成了，这部分整体来看比较容易，麻烦一点的就是根据warp_boxes修改出warp_points函数，实现四点的映射。

随后就要到达我们修改的大头，也是修改最麻烦的地方，head部分的修改。

三、NanoDetPlusHead修改

> 这一部分在修改时建议生成onnx模型，使用百度的VisualDL或者netron来将网络可视化，这样有助于修改网络，同时建议参考跃鹿战队的nanodet博客，辅助理解原始代码。
>
> Nanodet-plus的head包括了两部分，一部分是输出头head，另一部分是辅助训练模块的head。
>
> 这一部分在前面的config配置文件中也有体现。

> 这一部分的损失函数本来打算采用wing_loss，但是个人使用过程中效果不太好（也可能是哪里使用错误）。参考了nanodet_keypoints仓库，将四点损失函数和bbox两个点（左上和右下）的损失函数融合在一起进行计算，就是仿照bbox借助了DFL来计算四点回归的损失函数，这样修改后效果比wing_loss好很多（就个人而言）。
>
> 一定要注意理解以下内容，重视标签分配，它在目标检测中很重要（本人一开始不够重视，导致理解偏差）
>
> 网络输入 -> 网络输出 -> （解码 -> 标签分配 -> 计算损失函数） -> 其他操作

1. 首先修改网络的结构：即增加网络的输出

`nanodet/model/head/nanodet_plus_head.py:`

```
    # 生成网络需要的结构：一个head对应，两个卷积cls_convs + 一个gfl_cls部分，nanodet-plus有4个head
    def _init_layers(self):
        self.cls_convs = nn.ModuleList()
        for _ in self.strides:                                  # 为每个stride创建一个head，cls和reg共享这些参数
            cls_convs = self._buid_not_shared_head()
            self.cls_convs.append(cls_convs)    # 四层
        # 为每个头增加gfl卷积 输出（1x1的卷积，改变通道数）， 类似于一个全连接层
        self.gfl_cls = nn.ModuleList(
            [
                nn.Conv2d(
                    self.feat_channels,
                    self.num_classes + 4 * (self.reg_max + 1) + 8 * (self.reg_max + 1), #TODO 增加
                    1,                                          # 使用1x1的卷积更改通道数
                    padding=0,
                )
                for _ in self.strides                           # 每个尺度增加一个gfl卷积(最终的1x1卷积)
            ]
        )
```

这样修改后，增加了网络的输出，在前面介绍过，个人使用四点直接输出（也就是输出通道增加8）效果不太好，而采用8 x (reg_max+1) 通道输出效果比较好，在损失函数计算式采用了DFL，所以后面要注意是增加了8 x (reg_max+1)个输出。

个人认为的head处理的流程大概是： **先调用了forward，生成预测pred，随后便要计算loss**（通过其他部分完成反向传播）。

postprocess（其中调用了get_bboxes进行解码），也就是将预测的结果转换成一个result_list，转换成和标签一致的形式，这一部分主要是用在推理的部分上，也就是pytorch版本的推理。

2.损失函数部分修改

下面说明修改比较困难的地方，损失函数部分的修改，这部分涉及内容较多。

`nanodet/model/head/nanodet_plus_head.py：`

其中包括了计算中心点坐标的部分，这一部分和前面说明的Anchor-free是一致的。

```
    # gt_meta 就是用户标注的数据，可以认为就是标签文件的内容， preds就是[batchsize, w*h, c]
    def loss(self, preds, gt_meta, aux_preds=None):
        device = preds.device
        batch_size = preds.shape[0]   # 得到本次loss计算的batch数，pred是3维的tensor

        # 把gt相关的数据分离出来，这两个数据都是list，长度为batchsize的大小
        # 每个list都包含他们各自对应的图像上的gt和label
        gt_bboxes = gt_meta["gt_bboxes"]    # 检验框              [batchsize, num_gts, 4]
        gt_labels = gt_meta["gt_labels"]    # 类别               [batchsize, num_gts]
        gt_points = gt_meta["gt_points"]    # TODO:  数据集中需要新增的四个点 [batchsize, num_gts, 8]

        gt_bboxes_ignore = gt_meta["gt_bboxes_ignore"]
        if gt_bboxes_ignore is None:
            gt_bboxes_ignore = [None for _ in range(batch_size)]

        # img信息提取长宽，（就是标签中的一张图片的大小）
        # 所有图片都会在前处理中被resize成网络的输入大小，不足则直接加zero padding
        input_height, input_width = gt_meta["img"].shape[2:]

        # 如果修改了输入或者采样率，输入无法被stride整除，所以要用ceil取整
        # 因为稍后要布置priors，这里要计算出feature map的大小   [40, 40] [20, 20] [10, 10] [5, 5]
        featmap_sizes = [
            (math.ceil(input_height / stride), math.ceil(input_width) / stride)
            for stride in self.strides
        ]

        # get grid cells of one image       为了方便计算bbox的损失函数准备
        # 在不同大小的stride上放置一组prior，默认四个检测头也就是四个不同尺寸的stride
        # 最后返回的tensor维度是[batchsize, strideW*strideH, 4]
        # 其中每一个都是[x, y, strideH, strideW]的结构，当featuremap不是正方形的时候两个stride不相等
        # 相当于生成了一堆锚点（图像中横纵一定步长，生成一些列点，anchor-free 会生成一系列锚点）
        mlvl_center_priors = [
            self.get_single_level_center_priors(
                batch_size,
                featmap_sizes[i],
                stride,
                dtype=torch.float32,
                device=device,
            )
            for i, stride in enumerate(self.strides)
        ]
```

```
42
43          # 按照第二个维度拼接后的prior的维度是[batchsize, 40x40+20x20+10x10+5x5,
    4]
44          # 其中四个值为[cx, cy, strideW, strideH]，横纵像素坐标，以及步长
45          center_priors = torch.cat(mlvl_center_priors, dim=1)
46
47          # 预测部分：     把预测值拆分成分类和框回归、四点回归
48          cls_preds, reg_preds = preds.split(                # TODO [更新]新增
    pts_preds部分，split出四个角点回归的部分
49              [self.num_classes, 12 * (self.reg_max + 1)], dim=-1
50          )
51
52          # cls_preds : [1, 2125, 80]
53          # reg_preds : [1, 2125, 32]
54          # 相应的 要求，pts_preds : [1, 2125, 64]
55
56          # 对reg_preds进行    `积分`求和   得到位置预测，reg_preds 表示的是一条边的离
    散分布，每个锚点预测四条边的距离
57          dis_preds = self.distribution_project(reg_preds) *
    center_priors[..., 2, None]
58          dis_bbox_preds, dis_pts_preds = dis_preds.split([4, 8], dim=2)
59          # 根据中心点和距离得到bbox（左上点、右下点），也就是2125个输出结果的bbox以及
    points
60          decoded_bboxes = distance2bbox(center_priors[..., :2],
    dis_bbox_preds)
61          decoded_points = distance2pts(center_priors[..., :2],
    dis_pts_preds)   # TODO [更新] 改进点预测
62
63          # 如果启用了辅助训练模块，将用辅助训练的结果进行`标签分配`，
64          if aux_preds is not None:
65              aux_cls_preds, aux_reg_preds = aux_preds.split(
66                  [self.num_classes, 12 * (self.reg_max + 1)], dim=-1
67              )
68
69              # 对reg_preds积分得到预测位置，reg_preds 表示的是一条边的离散分布
70              aux_dis_preds = (
71                  self.distribution_project(aux_reg_preds) *
    center_priors[..., 2, None]
72              )
73
74              aux_dis_bbox_preds, aux_dis_pts_preds = aux_dis_preds.split([4,
    8], dim=2)
75
76              # 根据中心点和距离得到bbox（左上点、右下点）
77              aux_decoded_bboxes = distance2bbox(center_priors[..., :2],
    aux_dis_bbox_preds)
78              aux_decoded_points = distance2pts(center_priors[..., :2],
    aux_dis_pts_preds)    # TODO [更新] 改进点预测
79
80              # 每次给一张图片进行分配，应该是为了避免显存溢出
81              batch_assign_res = multi_apply(
82                  self.target_assign_single_img,
83                  aux_cls_preds.detach(),          # 类别预测
84                  center_priors,                   # [cx,cy,strideW,strideH]
    中心点
85                  aux_decoded_bboxes.detach(),     # 预测框 的 左上角、右下角坐标
    （一共strideW*strideH个框）
86                  aux_decoded_points.detach(),     # TODO  新增部分
```

```
                gt_bboxes,                          # 真实框
                gt_labels,                          # 真实类别
                gt_points,                          # TODO  对于AGM也增加gt_points
                gt_bboxes_ignore,
            )
        else:
            # multi_apply将参数中的函数作用在后面的每一个可迭代对象上，一次处理批量数据
            # use self prediction to assign
            # target_assign_single_img 一次只能分配一张图片
            batch_assign_res = multi_apply(         # 分配的时候还是按照bbox进行的标签分配
                self.target_assign_single_img,
                cls_preds.detach(),
                center_priors,
                decoded_bboxes.detach(),
                decoded_points.detach(),            # TODO  新增部分
                gt_bboxes,
                gt_labels,
                gt_points,                          # TODO 修改target_assign_single_img函数，新增gt_points部分
                gt_bboxes_ignore,
            )
        # 根据·分配结果·计算loss
        loss, loss_states = self._get_loss_from_assign(
            cls_preds, reg_preds, decoded_bboxes, batch_assign_res     # TODO 新增pts_preds输入
        )

        # 加入 `辅助训练模块的loss` ，这可以让网络在初期收敛的更快
        if aux_preds is not None:
            aux_loss, aux_loss_states = self._get_loss_from_assign(
                aux_cls_preds, aux_reg_preds, aux_decoded_bboxes, batch_assign_res
            )
            loss = loss + aux_loss
            for k, v in aux_loss_states.items():
                loss_states["aux_" + k] = v

        return loss, loss_states
```

在loss函数中调用了很多函数，这些函数也需要进行一定的修改，接下来就会对这些函数进行一个比较详细的说明，主要修改了以下部分。

- **计算loss时提到了distance2pts，这一部分仿照distance2bbox修改就可以。**

`nanodet/util/box_transform.py:`

这里一定要**注意加减符号**，本人第一次没注意就搞错了，导致输出结果的部分位置框的位置很完美，但是四点的位置缺歪七扭八，这一点要注意。

```
# TODO 仿照distance2bbox新增distance2pts
def distance2pts(points, distance, max_shape=None):
    x1 = points[..., 0] - distance[..., 0]
    y1 = points[..., 1] - distance[..., 1]
    x2 = points[..., 0] - distance[..., 2]
    y2 = points[..., 1] + distance[..., 3]
```

```
    x3 = points[..., 0] + distance[..., 4]
    y3 = points[..., 1] + distance[..., 5]
    x4 = points[..., 0] + distance[..., 6]
    y4 = points[..., 1] - distance[..., 7]
    if max_shape is not None:
        x1 = x1.clamp(min=0, max=max_shape[1])
        y1 = y1.clamp(min=0, max=max_shape[0])
        x2 = x2.clamp(min=0, max=max_shape[1])
        y2 = y2.clamp(min=0, max=max_shape[0])
        x3 = x3.clamp(min=0, max=max_shape[1])
        y3 = y3.clamp(min=0, max=max_shape[0])
        x4 = x4.clamp(min=0, max=max_shape[1])
        y4 = y4.clamp(min=0, max=max_shape[0])
    return torch.stack([x1, y1, x2, y2, x3, y3, x4, y4], -1)
```

- **target_assign_single_img，这个函数主要是对一张图片的处理结果进行标签的分配**

标签分配部分（这一部分比较复杂并且修改方式不唯一）

> 如果这一部分解释的不是很详细，请见谅，个人对这一部分的理解也不够深入。
>
> 个人感觉这一部分写的比较差，见谅。
>
> 实际上并没有动标签分配，而是利用bbox的标签分配的结果，直接对四点进行索引，类似实现了标签分配。

`nanodet/model/head/nanodet_plus_head.py:`

标签分配，就如同前面所说的，将所有的输出结果进行打标签，正样本还是负样本，或者ignore。

计算损失函数时，正样本计算回归损失以及类别损失，负样本只计算类别损失。

```
    # 标签分配时的运算不会被记录，只是在计算cost并进行匹配，需要特别注意，这个函数只为一张图片，即一个样本进行标签分配
    @torch.no_grad()
    def target_assign_single_img(
        self,
        cls_preds,                          # [2125, 36]
        center_priors,
        decoded_bboxes,
        decoded_points,                     # TODO: 新增部分 decoded_points
        gt_bboxes,                          # [num_gts, 4]

        gt_labels,                          # [num_gts]
        gt_points,                          # TODO: 新增gt_points部分
        gt_bboxes_ignore=None,
    ):
        device = center_priors.device
        gt_bboxes = torch.from_numpy(gt_bboxes).to(device)  # [num_gts, 4]
        gt_points = torch.from_numpy(gt_points).to(device)  # TODO :新增部分“仿照gt_points" [num_gts, 8]
        gt_labels = torch.from_numpy(gt_labels).to(device)  # [num_gts]
        gt_bboxes = gt_bboxes.to(decoded_bboxes.dtype)
        gt_points = gt_points.to(decoded_points.dtype)      # TODO 仿照to(decoded_bboxes.dtype)进行处理

```

```
        if gt_bboxes_ignore is not None:
            gt_bboxes_ignore = torch.from_numpy(gt_bboxes_ignore).to(device)
            gt_bboxes_ignore = gt_bboxes_ignore.to(decoded_bboxes.dtype)

        # class的输出要映射到0-1之间，head构建conv layer 可以发现最后的分类没有激活函数
        assign_result = self.assigner.assign(
            cls_preds.sigmoid(),
            center_priors,
            decoded_bboxes,
            decoded_points,                  # TODO：新增的内容
            gt_bboxes,
            gt_points,                       # TODO：新增的内容
            gt_labels,
            gt_bboxes_ignore,
        )

        # 调用采样函数，获得正负样本，pos正，neg负
        pos_inds, neg_inds, pos_gt_bboxes, pos_assigned_gt_inds = self.sample(
            assign_result, gt_bboxes
        )

        num_priors = center_priors.size(0)                          # prior的个数
        bbox_targets = torch.zeros_like(center_priors)

        dist_bbox_targets = torch.zeros(num_priors, 4).to(device)         # TODO 修改
        dist_pts_targets = torch.zeros(num_priors, 8).to(device)          # TODO 修改

        # 把label扩充成 one - hot 向量
        labels = center_priors.new_full(
            (num_priors,), self.num_classes, dtype=torch.long
        )

        # No target
        label_weights = center_priors.new_zeros(num_priors, dtype=torch.float)
        label_scores = center_priors.new_zeros(labels.shape, dtype=torch.float)

        # 当前分配到这个图片上的正样本数
        num_pos_per_img = pos_inds.size(0)

        # 把分配到了gt的那些prior预测的检验框和gt的iou算出来，用于QFL计算
        pos_ious = assign_result.max_overlaps[pos_inds]

        if len(pos_inds) > 0:

            # bbox_targets就是最终用来和gt计算回归损失的东西，维度为[2125, 4]
            # bbox_targets是检测框的四条边和它对应的prior的偏移量，要换成原图上的框，和gt进行回归损失计算
            bbox_targets[pos_inds, :] = pos_gt_bboxes

            # TODO 训练时，这里报了一个错，需要解决： Expected all tensors to be on the same device, but found at least two devices, cuda:0 and cpu!
```

```
            # 怀疑前面的zeros是在cpu里的，zeros_like 仿照定义，在gpu里了
            dist_bbox_targets[pos_inds, :] = (
                bbox2distance(center_priors[pos_inds, :2], pos_gt_bboxes)
                / center_priors[pos_inds, None, 2]
            )

            # 参考sample函数中的这一句  pos_gt_bboxes = gt_bboxes[pos_assigned_gt_inds, :]
            dist_pts_targets[pos_inds, :] = (

                pts2distance(center_priors[pos_inds, :2], gt_points[pos_assigned_gt_inds])  # 重要，选出分配了正样本的对应的四点
                / center_priors[pos_inds, None, 2]
            )

            dist_bbox_targets = dist_bbox_targets.clamp(min=0, max=self.reg_max - 0.1)
            dist_pts_targets = dist_pts_targets.clamp(min=0, max=self.reg_max - 0.1)    # TODO 仿照新增

            # 上面计算回归，这里就是得到用于计算类别损失的，把那些匹配到的prior利用pos_inds索引筛出来
            labels[pos_inds] = gt_labels[pos_assigned_gt_inds]
            label_scores[pos_inds] = pos_ious
            label_weights[pos_inds] = 1.0
        if len(neg_inds) > 0:
            label_weights[neg_inds] = 1.0
        return (
            labels,
            label_scores,
            label_weights,      # label的这些，是为了方便计算qfl损失
            bbox_targets,       # bbox（用来计算iou损失）
            dist_bbox_targets,  # 距离形式：bbox  这两个dist都是为了方便计算dfl，得到对应的损失函数。【重要】
            dist_pts_targets,   # 距离形式：四点
            num_pos_per_img,    # 正样本的个数
        )
```

> **以下内容感觉可以不必修改**（没有测试不修改）**仁者见仁智者见智**！
>
> 主要就是在计算cost矩阵时，新增了四点的cost，这个cost的计算参考了nanodet_keypoints仓库。

`nanodet/model/head/assigner/dsl_assigner.py`

```
    def assign(                                                 # TODO 修改标签分配策略
        self,
        pred_scores,
        priors,
        decoded_bboxes,
        decoded_points,     # TODO 增加
        gt_bboxes,
        gt_points,          # TODO 增加
```

```
        gt_labels,
        gt_bboxes_ignore=None,
    ):
        """Assign gt to priors with dynamic soft label assignment.
        Args:
            pred_scores (Tensor): Classification scores of one image,
        分类得分2Dtensor
                a 2D-Tensor with shape [num_priors, num_classes]
            priors (Tensor): All priors of one image, a 2D-Tensor with
shape         一个图片的锚点
                [num_priors, 4] in [cx, xy, stride_w, stride_y] format.
            decoded_bboxes (Tensor): Predicted bboxes, a 2D-Tensor with
shape        预测的检测框 和 以上的锚点是对应的
                [num_priors, 4] in [tl_x, tl_y, br_x, br_y] format.
            gt_bboxes (Tensor): Ground truth bboxes of one image, a 2D-
Tensor        一个图片各个检测框的实际标注
                with shape [num_gts, 4] in [tl_x, tl_y, br_x, br_y] format.
            gt_bboxes_ignore (Tensor, optional): Ground truth bboxes that
are
                labelled as `ignored`, e.g., crowd boxes in COCO.
            gt_labels (Tensor): Ground truth labels of one image, a Tensor
         一个图片中各个识别框实际标注类别
                with shape [num_gts].

        Returns:
            :obj:`AssignResult`: The assigned result.
        """
        INF = 100000000
        num_gt = gt_bboxes.size(0)                                      #
实际标注的数量
        num_bboxes = decoded_bboxes.size(0)                             #
检验框的数量

        # assign 0 by default
        assigned_gt_inds = decoded_bboxes.new_full((num_bboxes,), 0,
dtype=torch.long)

        # 切片得到prior的为位置（类似anchor point的中心点）
        prior_center = priors[:, :2]        # [num_priors, 2]

        # 计算prior center到GT左上角和右下角的距离，从而判断prior是否在GT框内，得到
的是  每个prior center和每个 gt的 距离
        lt_ = prior_center[:, None] - gt_bboxes[:, :2]                  #
[:, None]在不改变数据的情况下，追加一个维度
        rb_ = gt_bboxes[:, 2:] - prior_center[:, None]

        deltas = torch.cat([lt_, rb_], dim=-1)
        # is_in_gts 通过判断deltas全部大于0，筛选出处在gt中的prior 锚点
        # [dxlt,dylt,dxrb,dyrb] 四个值都需要大于零，则他们中的最小值也要大于0
        is_in_gts = deltas.min(dim=-1).values > 0
        # 这一步生成有效的prior的索引，这里注意之所以使用sum是因为一个prior可能落在多
个gt中
        # 因此上一步生成的is_in_gts确定的是某个prior是否落在每一个gt中，只要落在一个
gt范围内，便是有效的
        valid_mask = is_in_gts.sum(dim=1) > 0

        # 利用得到的mask确定由哪些prior生成的pred_box和它们对应的scores是有效的
        # valid_decoded_bbox和valid_pred_scores的长度是落在gt中prior的个数
```

```
        valid_decoded_bbox = decoded_bboxes[valid_mask]
        valid_decoded_points = decoded_points[valid_mask]    # TODO: 仿照新增
        valid_pred_scores = pred_scores[valid_mask]

        num_valid = valid_decoded_bbox.size(0)

        # 如果没有预测框或者训练样本中没有gt的情况
        if num_gt == 0 or num_bboxes == 0 or num_valid == 0:
            # No ground truth or boxes, return empty assignment
            max_overlaps = decoded_bboxes.new_zeros((num_bboxes,))
            if num_gt == 0:
                # No truth, assign everything to background
                assigned_gt_inds[:] = 0
            if gt_labels is None:
                assigned_labels = None
            else:
                assigned_labels = decoded_bboxes.new_full(
                    (num_bboxes,), -1, dtype=torch.long
                )
            return AssignResult(
                num_gt, assigned_gt_inds, max_overlaps, labels=assigned_labels
            )

        # 计算bbox和gt的iou损失
        pairwise_ious = bbox_overlaps(valid_decoded_bbox, gt_bboxes)
        # 加上一个很小的数防止出现NaN
        iou_cost = -torch.log(pairwise_ious + 1e-7)

        pts_cost = points_loss(valid_decoded_points, gt_points)  # TODO：新增pts_cost

        # 根据num_valid的数量（有效bbox）生成对应长度的one-hot label之后用于计算soft lable
        # 每个匹配到gt的prior都会有一个tensor, label位置的元素为1，其余为0 （one-hot编码）
        gt_onehot_label = (
            F.one_hot(gt_labels.to(torch.int64), pred_scores.shape[-1])
            .float()
            .unsqueeze(0)
            .repeat(num_valid, 1, 1)
        )
        valid_pred_scores = valid_pred_scores.unsqueeze(1).repeat(1, num_gt, 1)

        # one-hot * IOU得到软标签， 预测框和gt越接近，预测越好
        soft_label = gt_onehot_label * pairwise_ious[..., None]
        # 差距越大，说明当前的预测效果越差，稍后的cost计算应该给一个更大的惩罚
        scale_factor = soft_label - valid_pred_scores

        # 计算分类交叉熵损失
        cls_cost = F.binary_cross_entropy(
            valid_pred_scores, soft_label, reduction="none"
        ) * scale_factor.abs().pow(2.0)

        cls_cost = cls_cost.sum(dim=-1)

```

```
        # 得到匹配开销矩阵，数值为分类损失，iou损失，这里利用iou_factor作为调制系数m和n的一个匹配矩阵
        cost_matrix = cls_cost + iou_cost * self.iou_factor + pts_cost #TODO新增加pts_cost（个人感觉pts_cost这一部分或许影响不大）

        # 返回值为分配到标签的prior与它们对应的gt的iou和这些prior匹配到的gt的索引
        matched_pred_ious, matched_gt_inds = self.dynamic_k_matching(
            cost_matrix, pairwise_ious, num_gt, valid_mask
        )

        # convert to AssignResult format
        # 把结果还原为priors的长度
        assigned_gt_inds[valid_mask] = matched_gt_inds + 1
        assigned_labels = assigned_gt_inds.new_full((num_bboxes,), -1)
        assigned_labels[valid_mask] = gt_labels[matched_gt_inds].long()
        max_overlaps = assigned_gt_inds.new_full(
            (num_bboxes,), -INF, dtype=torch.float32
        )
        max_overlaps[valid_mask] = matched_pred_ious

        if (
            self.ignore_iof_thr > 0
            and gt_bboxes_ignore is not None
            and gt_bboxes_ignore.numel() > 0
            and num_bboxes > 0
        ):
            ignore_overlaps = bbox_overlaps(
                valid_decoded_bbox, gt_bboxes_ignore, mode="iof"
            )
            ignore_max_overlaps, _ = ignore_overlaps.max(dim=1)
            ignore_idxs = ignore_max_overlaps > self.ignore_iof_thr
            assigned_gt_inds[ignore_idxs] = -1

        return AssignResult(
            num_gt, assigned_gt_inds, max_overlaps, labels=assigned_labels
        )

# points_loss这部分来自：https://github.com/1248289414/nanodet_keypoint/blob/rmdet/nanodet/model/loss/wing_loss.py
# 感谢nanodet_keypoints的仓库
def points_loss(pre_x, gt_x, alpha=1.03, beta=100):
    cost = torch.sub(pre_x[...,None,:],gt_x[...,None,:,:]).abs().sum(-1) / gt_x.size(-1)
    cost = torch.pow(alpha, cost - beta)
    return cost
```

- **_get_loss_from_assign真正进行计算损失函数的部分（重点）**

> 这一部分就是拿着标签分配的结果来计算损失函数。
>
> **计算loss时，bbox要计算iou损失，四点的损失归入了dfl，所以 没有decoded_points.** 这一点要谨记。

`nanodet/model/head/nanodet_plus_head.py：`

```python
    # prior就是框分布的回归起点，将以prior的位置作为目标中心，预测四个值形成检验框，assign target_assign_single_img的结果
    def _get_loss_from_assign(self, cls_preds, reg_preds, decoded_bboxes, assign):
        device = cls_preds.device
        (
            labels,                         # label
            label_scores,
            label_weights,
            bbox_targets,                   # bbox
            dist_bbox_targets,              # 距离
            dist_pts_targets,
            num_pos,
        ) = assign

        # 因为要对整个batch进行平均，因此，在这里计算出这次分配的总正样本数，用于稍后的weight_loss
        num_total_samples = max(
            reduce_mean(torch.tensor(sum(num_pos)).to(device)).item(), 1.0
        )

        # 为了一次性处理一个batch的数据，把每个结果都拼接起来
        # labels 和 label_score 都是[batchsize*2125], bbox_targets 是[batchsize*2125, 4*(reg_max+1)]
        labels = torch.cat(labels, dim=0)
        label_scores = torch.cat(label_scores, dim=0)
        label_weights = torch.cat(label_weights, dim=0)
        bbox_targets = torch.cat(bbox_targets, dim=0)

        # 把预测结果和检验框都reshape成和batch对应的形状
        cls_preds = cls_preds.reshape(-1, self.num_classes)
        reg_preds = reg_preds.reshape(-1, 12 * (self.reg_max + 1))

        decoded_bboxes = decoded_bboxes.reshape(-1, 4)
        # 计算loss时，bbox要计算iou损失，四点的损失归入了dfl，所以 没有decoded_points.

        # 计算quality focal loss
        # 利用iou联合了质量估计和分类表示，和软标签计算相似
        loss_qfl = self.loss_qfl(
            cls_preds,                                                  # cls_preds部分的损失函数
            (labels, label_scores),
            weight=label_weights,
            avg_factor=num_total_samples,
        )

        # 获取对应的label标签（把当前batch的所有index交给pos_inds）
        # tensor 中常用逻辑判断语句生成mask掩膜，元素中符合者编程True，反之为Flase
        pos_inds = torch.nonzero(
            (labels >= 0) & (labels < self.num_classes), as_tuple=False
        ).squeeze(1)

        # 结果label为空，说明没有分配任何标签，则不需要计算iou loss
        if len(pos_inds) > 0:
            # 计算用于weight_reduce的参数，weight_target的长度和被分配了gt的prior的数量相同
```

```
            # sigmoid后取最大值得到的就是该prior输出的类别分数
            weight_targets = cls_preds[pos_inds].detach().sigmoid().max(dim=1)[0]
            # 同步GPU上的其他worker，获得此参数
            bbox_avg_factor = max(reduce_mean(weight_targets.sum()).item(), 1.0)

            # 计算GIoU损失，加入了weighted_loss
            loss_bbox = self.loss_bbox(
                decoded_bboxes[pos_inds],
                bbox_targets[pos_inds],
                weight=weight_targets,
                avg_factor=bbox_avg_factor,
            )

            dist_bbox_targets = torch.cat(dist_bbox_targets, dim=0)    # TODO
            dist_pts_targets = torch.cat(dist_pts_targets, dim=0)    # TODO

            dist_targets = torch.cat([dist_bbox_targets, dist_pts_targets], dim=-1) # TODO

            loss_dfl = self.loss_dfl(                                # 计算dfl时，新增了四点的计算
                reg_preds[pos_inds].reshape(-1, self.reg_max + 1),
                dist_targets[pos_inds].reshape(-1),
                weight=weight_targets[:, None].expand(-1, 12).reshape(-1), # TODO：修改成12，这里就成功计算了loss
                avg_factor=4.0 * bbox_avg_factor,
            )

        else:
            loss_bbox = reg_preds.sum() * 0
            loss_dfl = reg_preds.sum() * 0

        # 计算损失函数
        loss = loss_qfl + loss_bbox + loss_dfl
        loss_states = dict(loss_qfl=loss_qfl, loss_bbox=loss_bbox, loss_dfl=loss_dfl, loss=loss)
        return loss, loss_states
```

总之，计算loss时，重点关注 target_assign_single_img以及_get_loss_from_assign，这两个函数互相联系。

3. 推理涉及的一部分修改

> 下面说明一下nanodet_plus_head.py中其他比较关键的部分，这一部分在推理时具有很大的作用。
>
> 即运行demo.py检验效果要使用的部分。

`nanodet/model/head/nanodet_plus_head.py：`

Post_process这一部分就是将输出部分进行解码，将网络原始输出转换成方便计算loss、可以准确直接表示推理结果的数据。

其中preds是网络的输出，没有处理过的原始输出，meta是读入的数据，可以看数据读取部分的修改，meta可以认为就是图片的信息，包括宽高、标签等等。

```
 def post_process(self, preds, meta): # 解码bbox 并且 rescale到原本的图像大小
        # print(preds.shape)  keep_ratio为True时，为[4, 1700, 76]
        # 根据preds得到对应的部分
        cls_scores, bbox_preds, pts_preds = preds.split( # TODO [更新]新增pts_preds，位于其他输出的后边，输出占用8*（reg_max+1）个通道
            [self.num_classes, 4 * (self.reg_max + 1), 8 * (self.reg_max + 1)], dim=-1
        )

        # print(cls_scores.shape)   [batchsize, 2125, 36]
        # print(bbox_preds.shape)   [batchsize, 2125, 32]
        # print(pts_preds.shape)    [batchsize, 2125, 8]

        # 注意一下顺序，先输出的是class（长度为num_classes），然后是bbox( 长度为(reg_max+1)*4 )，最后是points( 长度是8*(reg_max+1) )
        #                                  类别   +    检验框   +    关键点
        # 获取结果result_list[tuple] （bbox是左上点、右下点）

        # TODO 对get_bboxes进行处理，让其同时能够输出关键点， get_bboxes不仅能够获得bbox，还能够返回四点
        result_list = self.get_bboxes(cls_scores, bbox_preds, pts_preds, meta)  # TODO 新增pts_preds，通过get_bboxes将

        # 最终的处理部分
        det_results = {}
        warp_matrixes = (                                       # 这一部分在数据读取部分已经写明了，warp_matrix就是映射的矩阵
            meta["warp_matrix"]
            if isinstance(meta["warp_matrix"], list)
            else meta["warp_matrix"]
        )

        img_heights = (
            meta["img_info"]["height"].cpu().numpy()
            if isinstance(meta["img_info"]["height"], torch.Tensor)
            else meta["img_info"]["height"]
        )
        img_widths = (
            meta["img_info"]["width"].cpu().numpy()
            if isinstance(meta["img_info"]["width"], torch.Tensor)
            else meta["img_info"]["width"]
        )
        img_ids = (
            meta["img_info"]["id"].cpu().numpy()
            if isinstance(meta["img_info"]["id"], torch.Tensor)
            else meta["img_info"]["id"]
        )

        # 遍历所有的图片
        for result, img_width, img_height, img_id, warp_matrix in zip(
            result_list, img_widths, img_heights, img_ids, warp_matrixes
```

```
        ):
            det_result = {}

            det_bboxes, det_labels, det_pts = result   # TODO 新增det_pts, 注意顺序，bboxes，labels, points

            det_bboxes = det_bboxes.detach().cpu().numpy()
            det_pts = det_pts.detach().cpu().numpy()      # TODO 仿照det_bboxes添加

            det_bboxes[:, :4] = warp_boxes(
                det_bboxes[:, :4], np.linalg.inv(warp_matrix), img_width, img_height # inv: 矩阵求逆
            )

            det_pts[:, :8] = warp_points(
                det_pts[:, :8], np.linalg.inv(warp_matrix), img_width, img_height # TODO：warp_boxes应该是对两个点进行处理，仿照修改
            )

            classes = det_labels.detach().cpu().numpy()  # 类别

            for i in range(self.num_classes):
                inds = classes == i
                det_result[i] = np.concatenate(                             # 检验框、得分、四点
                    [
                        det_bboxes[inds, :4].astype(np.float32),            # 框 bbox
                        det_bboxes[inds, 4:5].astype(np.float32),           # score
                        det_pts[inds, 0:8].astype(np.float32),              # TODO 增加四个角点
                    ],
                    axis=1,                                                 # 将det按照第二个维度进行合并
                ).tolist()

            det_results[img_id] = det_result                                # 读进来的顺序时cls_scores, bbox_preds, pts_preds
        return det_results
```

post_process中调用了get_bboxes，只是说明了作用，接下来就要修改get_bboxes，增加解码四点作用。

为了方便修改，将四点的解码融入到了检验框的解码。

其中使用了mlvl_center_priors，生成一张图片的所有中心点，这一点就很明显，和之前说过的Anchor-free是一致的。

详细内容还是看注释。

`nanodet/model/head/nanodet_plus_head.py:`

```
    # 将结果解码成bboxes，这里同时
    def get_bboxes(self, cls_preds, reg_preds, pts_preds, img_metas):    # TODO 新增加了pts_preds
```

```
        device = cls_preds.device                                   # 获取设备，判断是在gpu还是cpu上
        b = cls_preds.shape[0]                                      # 获得batchsize
        input_height, input_width = img_metas["img"].shape[2:]  # 图片的高度、宽度
        input_shape = (input_height, input_width)

        featmap_sizes = [                   # [40, 20, 10, 5]
            (math.ceil(input_height / stride), math.ceil(input_width / stride))
            for stride in self.strides
        ]                                   # [(40, 40), (20, 20), (10, 10), (5, 5)]

        # 生成一张图片的所有中心点的坐标
        # get grid cells of one image
        mlvl_center_priors = [
            self.get_single_level_center_priors(
                b,
                featmap_sizes[i],
                stride,
                dtype=torch.float32,
                device=device,
            )
            for i, stride in enumerate(self.strides)
        ]

        # 所有中心点 [batchsize, 2125, 4]  其中4 是 x,y,stride,stride, x,y是输入图（320x320）上的坐标
        center_priors = torch.cat(mlvl_center_priors, dim=1)
        # print(reg_preds.shape)       # [batchsize, 2125, 32]

        # 得到距离，dis_preds 中心点、 `积分求和`
        # dis_preds = self.distribution_project(reg_preds) * center_priors[..., 2, None]
        dis_preds = self.distribution_project(torch.cat((reg_preds, pts_preds), dim=2)) * center_priors[..., 2, None]
        # [更新，将两个输出部分合并后积分再输出，不影响，只是借用bbox的积分同时对pts也进行积分]
        # print(dis_preds.shape)   [batchsize, 2125, 4]， 修改后会变成 [b, 2125, 12] , 包括了4点

        dis_bbox_preds, dis_pts_preds = dis_preds.split([4, 8], dim=2)
 # 积分后的结果分成 bbox 和 pts

        # 根据中心点和距离得到bbox（左上点、右下点）     #
        bboxes = distance2bbox(center_priors[..., :2], dis_bbox_preds, max_shape=input_shape)
        # print(bboxes.shape)           # torch.Size([b, 2125, 4])

        points = distance2pts(center_priors[..., :2], dis_pts_preds, max_shape=input_shape)  # TODO [更新] 改进点预测
        # print(points.shape)          # torch.Size([b, 2125, 8])

        # 类别得分
        scores = cls_preds.sigmoid()    # torch.Size([b, 2125, 36])
        # print(scores.shape)
```

```
48
49          # 存放最终的结果集
50          result_list = []
51
52          # 一个batch
53          for i in range(b):                          # batch中的每一张图片
54              # step1：首先，取出来一张图片，包括它的各个检验框的位置，四点位置，以及各个得分类别
55              score, bbox, pts = scores[i], bboxes[i], points[i]      # TODO 增加pts = points[i]
56              padding = score.new_zeros(score.shape[0], 1)            # padding
57              score = torch.cat([score, padding], dim=1)              # [2125, 37]，说是增加了一个背景的类别， 其实不起作用
58
59              # step2: 然后，将这张图片中的所有检验框，进行nms处理
60              results = multiclass_nms(       # 内部仍然按照bbox进行nms处理，只是输出时，同时输出points而已
61                  bbox,                       # 一张图片的预测的bbox      [2125, 4]
62                  score,                      # 预测的得分     [2125, 36+1]
63                  pts,                        # 预测的四点     [2125, 8]
64                  score_thr=0.05,             # 得分阈值
65                  nms_cfg=dict(type="nms", iou_threshold=0.6),    # nms处理
66                  max_num=20,                 # 最多的个数
67              )
68              # step3: 最后，将这张图片经过nms处理后的结果
69              result_list.append(results)     # 注意顺序bbox， label， point
70          return result_list
```

其中distribution_project、distance2pts、nulticlass_nms比较重要，distribution_project就是积分，distance2pts是把四点的距离表示转成坐标表示,nulticlass_nms就是对预测的结果进行nms处理。

接下来就要对着及部分分别进行修改。

1. distribution_project(也就是Integral)

在构造时，有这么一句，这个函数的功能就是完成积分，也就是将网络的输出转换成真实的输出： 可以认为是4*(reg_max+1)到4的转换，由于增加了四点，同时也要做积分，干脆直接编程12*(reg_max+1)到12的转变，其中前4个是bbox角点，后8个是poly四点。

```
1 self.distribution_project = Integral(self.reg_max)
```

`nanodet/model/head/gfl_head.py:`

所以也要对Integral做简单的修改，所做修改如下

```
1  class Integral(nn.Module):
2      def __init__(self, reg_max=16):
3          super(Integral, self).__init__()
4          self.reg_max = reg_max
5          self.register_buffer(
6              "project", torch.linspace(0, self.reg_max, self.reg_max + 1)
7          )
8
9      def forward(self, x):
10         shape = x.size()
```

```
        # x = F.softmax(x.reshape(*shape[:-1], 4, self.reg_max + 1), dim=-1)
        # x = F.linear(x, self.project.type_as(x)).reshape(*shape[:-1], 4)
        x = F.softmax(x.reshape(*shape[:-1], 12, self.reg_max + 1), dim=-1)
 # TODO 修改积分，使得在完成bbox积分时，同时完成pts的积分
        x = F.linear(x, self.project.type_as(x)).reshape(*shape[:-1], 12)
        return x
```

2. distance2pts，仿照distance2bbox修改就可以。

`nanodet/util/box_transform.py:`

这里一定要**注意加减符号**，本人第一次没注意就搞错了，导致输出结果的部分位置框的位置很完美，但是四点的位置缺歪七扭八，这一点要注意。

```
# TODO 仿照distance2bbox新增distance2pts
def distance2pts(points, distance, max_shape=None):
    x1 = points[..., 0] - distance[..., 0]
    y1 = points[..., 1] - distance[..., 1]
    x2 = points[..., 0] - distance[..., 2]
    y2 = points[..., 1] + distance[..., 3]
    x3 = points[..., 0] + distance[..., 4]
    y3 = points[..., 1] + distance[..., 5]
    x4 = points[..., 0] + distance[..., 6]
    y4 = points[..., 1] - distance[..., 7]
    if max_shape is not None:
        x1 = x1.clamp(min=0, max=max_shape[1])
        y1 = y1.clamp(min=0, max=max_shape[0])
        x2 = x2.clamp(min=0, max=max_shape[1])
        y2 = y2.clamp(min=0, max=max_shape[0])
        x3 = x3.clamp(min=0, max=max_shape[1])
        y3 = y3.clamp(min=0, max=max_shape[0])
        x4 = x4.clamp(min=0, max=max_shape[1])
        y4 = y4.clamp(min=0, max=max_shape[0])
    return torch.stack([x1, y1, x2, y2, x3, y3, x4, y4], -1)
```

3. multiclass_nms

`nanodet/model/module/nms.py:`

这一部分就是对输出的结果进行nms处理，得到预期的输出（就是哪些可以画在image图像上的信息）。

nms处理的时候，只是将bbox进行了nms处理，并没有计算四点的iou等等，bbox和四点是绑定的，bbox一定程度上代表了四点所表示的四边形，points只是利用了bbox进行nms处理的索引值（近似进行了nms处理）。

```
def multiclass_nms(
                        # TODO: 新增multi_pts
    multi_bboxes, multi_scores, multi_pts, score_thr, nms_cfg, max_num=-1,
score_factors=None
):
    num_classes = multi_scores.size(1) - 1  # 36， 减去背景的类别
    # print(multi_scores.shape)       # [2125, 36+1] num_classes + padding
    # print(multi_bboxes.shape)       # [2125, 4]
    # print(multi_pts.shape)          # [2125, 8]

    # exclude background category
```

```
    if multi_bboxes.shape[1] > 4:
        bboxes = multi_bboxes.view(multi_scores.size(0), -1, 4)
        points = multi_pts.view(multi_scores.size(0), -1, 8)
           # TODO 仿照bboxes增加points
    else:
        bboxes = multi_bboxes[:, None].expand(multi_scores.size(0), num_classes, 4)     # [:None], 在数据不变的情况下，增加一个维度    expand进行扩展，并且expand只能对维度值等于1的那个维度进行扩展
        points = multi_pts[:, None].expand(multi_scores.size(0), num_classes, 8)        # TODO 同上
    scores = multi_scores[:, :-1]
    # print(scores.shape)       # [2125, 36]
    # print(bboxes.shape)       # [2125, 36, 4]          增加成为了36类别
    # print(points.shape)       # [2125, 36, 8]

    # filter out boxes with low scores  过滤掉得分很低的部分，筛选出大于阈值的那些目标
    valid_mask = scores > score_thr     # 相当于一个bool的列表  [2125, 36] 可以认为是 一个bool张量

    # We use masked_select for ONNX exporting purpose,
    # which is equivalent to bboxes = bboxes[valid_mask]
    # we have to use this ugly code  找出2125xnum_classes个目标中，得分比较高的一些结果，并将结果分割成[num, 4]大小
    bboxes = torch.masked_select(
        bboxes, torch.stack((valid_mask, valid_mask, valid_mask, valid_mask), -1)
    ).view(-1, 4)
    # print(bboxes.shape)       # [num, 4]

    points = torch.masked_select( # 找出2125xnum_classes个目标中，得分比较高的一些结果，并将结果分割成[num, 8]大小
        points, torch.stack((valid_mask, valid_mask, valid_mask, valid_mask, valid_mask, valid_mask, valid_mask, valid_mask), -1)
    ).view(-1, 8)
    # print(points.shape)       # [num, 8]

    if score_factors is not None:
        scores = scores * score_factors[:, None]
    scores = torch.masked_select(scores, valid_mask)        # torch.Size([0])
    labels = valid_mask.nonzero(as_tuple=False)[:, 1]       # torch.Size([0])

    if bboxes.numel() == 0:    # numel()函数是统计张量的个数，总共有多少个数据 eg [1,2,3] ,则返回 1x2x3=6， 6个数据
        # 也就是没有结果
        bboxes_pts = multi_bboxes.new_zeros((0, 5))  # torch.Size([0, 5]) 检验框，得分
        labels = multi_bboxes.new_zeros((0,), dtype=torch.long)    # torch.Size([0]) 类别
        points = multi_pts.new_zeros((0, 8))         # 四点

        if torch.onnx.is_in_onnx_export():
            raise RuntimeError(
                "[ONNX Error] Can not record NMS "
                "as it has not been executed this time"
            )
```

```
        return bboxes_pts, labels, points  # TODO 增加返回points，注意返回的顺序： bboxes, points, labels

    dets, keep = batched_nms(bboxes, scores, labels, nms_cfg)
    # 只取出max_num个结果，只选出最高的几个
    if max_num > 0:
        dets = dets[:max_num]
        keep = keep[:max_num]

    return dets, labels[keep], points[keep]     # TODO 增加返回pooints
```