# 残疾人识别图像分割系统源码＆数据集分享 [yolov8-seg-C2f-DCNV2＆yolov8-seg-AFPN-P345等50+全套改进创新点发刊\_一键训练教程\_Web前端展示]

## 1. 研究背景与意义

研究背景与意义  
  
随着社会的不断发展与进步，残疾人群体的生活质量和社会参与度日益受到重视。残疾人识别技术作为辅助残疾人融入社会的重要手段，近年来得到了广泛的研究与应用。传统的残疾人识别方法多依赖于人工观察和判断，效率低下且易受主观因素影响。随着计算机视觉和深度学习技术的快速发展，基于图像处理的自动识别系统逐渐成为研究热点。其中，YOLO（You Only Look Once）系列模型因其高效的实时目标检测能力而备受关注，尤其是最新的YOLOv8版本，其在精度和速度上的优势使其成为图像分割任务的理想选择。  
  
本研究旨在基于改进的YOLOv8模型，构建一个高效的残疾人识别图像分割系统。该系统将通过对图像进行实例分割，准确识别和分类不同类型的残疾人及其所使用的辅助设备。数据集“Yang Baru”包含6200张图像，涵盖了四个类别：辅助（Assistive）、非辅助（Non Assistive）、正常（Normal）。这一数据集的多样性和丰富性为模型的训练和验证提供了坚实的基础。通过对这些图像的分析，系统不仅能够识别残疾人及其辅助设备，还能为后续的智能辅助技术提供数据支持，推动残疾人辅助技术的进一步发展。  
  
改进YOLOv8模型的应用，将在提高识别精度的同时，降低误识率，确保系统在复杂环境下的稳定性和可靠性。此外，实例分割技术的引入，将使得系统能够更细致地处理图像信息，区分不同类别的对象，进而实现更为精准的识别。这对于残疾人群体的日常生活和社会交往具有重要意义，能够帮助他们更好地融入社会，提升生活质量。  
  
在全球范围内，残疾人群体的比例逐年上升，如何为这一特殊群体提供更好的服务和支持，已成为社会各界关注的焦点。通过构建基于改进YOLOv8的残疾人识别图像分割系统，不仅能够推动计算机视觉技术在社会服务领域的应用，还能够为政策制定者提供数据支持，帮助其制定更为合理的残疾人服务政策。此外，该系统的研究成果还可以为其他相关领域的研究提供借鉴，如智能监控、公共安全等。  
  
综上所述，本研究不仅具有重要的学术价值，还具有广泛的社会意义。通过对残疾人识别技术的深入探索，我们希望能够为残疾人群体的生活带来积极的变化，促进社会的包容性发展。

## 2. 图片演示

##### 注意：由于此博客编辑较早，上面“2.图片演示”和“3.视频演示”展示的系统图片或者视频可能为老版本，新版本在老版本的基础上升级如下：（实际效果以升级的新版本为准）  
  
 （1）适配了YOLOV8的“目标检测”模型和“实例分割”模型，通过加载相应的权重（.pt）文件即可自适应加载模型。  
  
 （2）支持“图片识别”、“视频识别”、“摄像头实时识别”三种识别模式。  
  
 （3）支持“图片识别”、“视频识别”、“摄像头实时识别”三种识别结果保存导出，解决手动导出（容易卡顿出现爆内存）存在的问题，识别完自动保存结果并导出到tempDir中。  
  
 （4）支持Web前端系统中的标题、背景图等自定义修改，后面提供修改教程。  
  
 另外本项目提供训练的数据集和训练教程,暂不提供权重文件（best.pt）,需要您按照教程进行训练后实现图片演示和Web前端界面演示的效果。

## 3.视频演示

## 4. 数据集信息展示

数据集信息展示  
  
在本研究中，我们使用了名为“Yang Baru”的数据集，以训练和改进YOLOv8-seg模型，旨在提升残疾人识别图像分割系统的性能。该数据集专门设计用于支持对不同类型辅助设备及其使用者的准确识别与分类，尤其是在图像分割任务中展现出其独特的价值。数据集包含四个主要类别，分别为“Assistive”、“Non Assistive”、“Non-Assistive”和“Normal”。这些类别的划分不仅有助于我们更好地理解和分析残疾人群体的需求，还能在技术上推动辅助技术的创新与应用。  
  
“Assistive”类别主要包括各种辅助设备的图像，如轮椅、助行器、听力辅助器等。这些设备在帮助残疾人提高生活质量、增强独立性方面发挥着重要作用。通过对这一类别的准确识别与分割，系统能够有效地提供相关的辅助信息，帮助残疾人更好地利用这些设备。此外，准确识别辅助设备的存在与使用情况，也为后续的智能化服务提供了数据支持，例如个性化的健康管理和生活指导。  
  
“Non Assistive”与“Non-Assistive”类别则主要涵盖那些不属于辅助设备的物品或环境元素。这些类别的设定旨在排除干扰因素，以便更清晰地识别出残疾人所处的环境与条件。这种分类不仅有助于提高模型的准确性，还能为后续的图像分析提供更为丰富的背景信息，确保系统在复杂场景下的鲁棒性和可靠性。  
  
最后，“Normal”类别代表的是未受残疾影响的个体或环境。通过对这一类别的识别，系统能够更好地进行对比分析，进而评估辅助设备在不同环境下的有效性和必要性。这种对比不仅有助于提升系统的智能化水平，还能为政策制定者和相关机构提供重要的数据支持，以推动社会对残疾人群体的关注与支持。  
  
“Yang Baru”数据集的构建与使用，体现了对残疾人群体需求的深刻理解与尊重。通过精确的图像分割与分类，我们的目标是开发出一种更加智能化的识别系统，能够在各种环境中快速、准确地识别出残疾人及其使用的辅助设备。这不仅有助于提升残疾人的生活质量，也为社会的包容性发展提供了技术支持。  
  
总之，数据集“Yang Baru”在残疾人识别图像分割系统中的应用，展现了其在实际场景中的重要性与必要性。通过对不同类别的深入分析与理解，我们期待能够推动相关技术的进步，为残疾人群体创造更加友好的生活环境与条件。

## 5. 全套项目环境部署教程（零基础手把手教学）

5.1 环境部署视频教程（零基础手把手教学）

https://www.bilibili.com/video/BV1jG4Ve4E9t/?vd\_source=bc9aec86d164b67a7004b996143742dc

5.2 安装Python虚拟环境创建和依赖库安装视频教程（零基础手把手教学）

https://www.bilibili.com/video/BV1nA4VeYEze/?vd\_source=bc9aec86d164b67a7004b996143742dc

## 6. 手把手YOLOV8-seg训练视频教程（零基础小白有手就能学会）

https://www.bilibili.com/video/BV1cA4VeYETe/?vd\_source=bc9aec86d164b67a7004b996143742dc

## 7.50+种全套YOLOV8-seg创新点代码加载调参视频教程（一键加载写好的改进模型的配置文件）

https://www.bilibili.com/video/BV1Hw4VePEXv/?vd\_source=bc9aec86d164b67a7004b996143742dc

## 8. YOLOV8-seg图像分割算法原理

原始YOLOv8-seg算法原理  
  
YOLOv8-seg算法是YOLO系列目标检测算法的最新版本，结合了目标检测与图像分割的能力，代表了计算机视觉领域中的一项重要进展。YOLO（You Only Look Once）算法自首次提出以来，就以其高效的实时检测能力和较高的精度而受到广泛关注。YOLOv8-seg在YOLOv8的基础上，进一步扩展了其功能，旨在同时实现目标的检测与精确的像素级分割，这对于许多应用场景，尤其是自动驾驶、医学影像分析和机器人视觉等领域，具有重要的实际意义。  
  
YOLOv8-seg的核心思想是将目标检测任务转化为一个回归问题，利用一个单一的神经网络模型同时进行目标的定位和分类。在YOLOv8中，算法结构被划分为主干网络（backbone）、特征增强网络（neck）和检测头（head）三个主要部分。主干网络负责提取输入图像的特征，特征增强网络则通过多层次的特征融合来提升模型对不同尺度目标的检测能力，而检测头则是最终进行目标定位和分类的部分。  
  
在YOLOv8-seg中，主干网络依然采用了CSP（Cross Stage Partial）结构，旨在提高特征提取的效率和效果。CSP结构通过将特征图分割成多个部分，并在不同的路径中进行处理，从而实现更丰富的特征表达。这种设计不仅提升了网络的学习能力，还有效减少了计算资源的消耗，使得YOLOv8-seg能够在嵌入式设备上运行，满足实时检测的需求。  
  
特征增强网络采用了PAN-FPN（Path Aggregation Network - Feature Pyramid Network）的思想，旨在更好地融合来自不同层次的特征信息。通过特征金字塔的构建，YOLOv8-seg能够在不同尺度上获取更全面的上下文信息，进而提高对小目标和复杂场景的检测能力。PAN-FPN通过路径聚合的方式，将高层语义特征与低层细节特征进行有效结合，使得网络在处理复杂背景时，能够更好地保持目标的边界信息。  
  
检测头部分，YOLOv8-seg采用了解耦头的结构，将目标的分类和回归任务分开处理。这种解耦设计使得模型在进行目标定位时，可以更加专注于位置的准确性，而在进行目标分类时，则能够更好地提高分类的精度。这种分离的方式有效解决了传统目标检测中常见的定位不准和分类错误的问题，尤其是在复杂场景下，解耦头的优势愈加明显。  
  
值得一提的是，YOLOv8-seg引入了Anchor-free的目标检测方法，这一创新使得模型不再依赖于预先定义的锚点框来预测目标的位置。传统的锚点框方法在处理不同尺度和形状的目标时，往往需要进行繁琐的调整，而YOLOv8-seg通过直接回归目标的位置和大小，简化了这一过程。这种方法不仅加快了模型的收敛速度，还使得网络能够更快地聚焦于目标的实际位置，提升了检测的精度。  
  
在训练过程中，YOLOv8-seg利用了一系列先进的损失函数设计，确保模型在优化过程中能够有效地平衡分类和回归任务的权重。通过引入自适应损失函数，YOLOv8-seg能够在不同的训练阶段动态调整损失的计算方式，从而更好地适应数据的分布特征，提高模型的泛化能力。  
  
YOLOv8-seg的应用场景非常广泛，尤其是在需要同时进行目标检测和图像分割的任务中，其优势尤为明显。例如，在自动驾驶领域，YOLOv8-seg可以实时检测路上的行人、车辆等目标，并同时提供精确的分割信息，为后续的决策提供支持。在医学影像分析中，YOLOv8-seg能够帮助医生快速识别和分割病灶区域，提高诊断的效率和准确性。  
  
总之，YOLOv8-seg算法在YOLO系列的基础上，融合了目标检测与图像分割的能力，凭借其高效的网络结构和创新的解耦设计，展现出了卓越的性能。随着计算机视觉技术的不断发展，YOLOv8-seg无疑将为各个领域带来更多的应用可能性和研究价值。通过进一步的优化和改进，YOLOv8-seg有望在未来的视觉任务中发挥更大的作用，推动智能化应用的深入发展。

## 9.系统功能展示（检测对象为举例，实际内容以本项目数据集为准）

图9.1.系统支持检测结果表格显示  
  
 图9.2.系统支持置信度和IOU阈值手动调节  
  
 图9.3.系统支持自定义加载权重文件best.pt(需要你通过步骤5中训练获得)  
  
 图9.4.系统支持摄像头实时识别  
  
 图9.5.系统支持图片识别  
  
 图9.6.系统支持视频识别  
  
 图9.7.系统支持识别结果文件自动保存  
  
 图9.8.系统支持Excel导出检测结果数据

## 10. 50+全套YOLOV8-seg创新点原理讲解（非科班也可以轻松写刊发刊，V11版本正在科研待更新）

由于篇幅限制，每个创新点的具体原理讲解就不一一展开，具体见下列网址中的创新点对应子项目的技术原理博客网址【Blog】：

https://gitee.com/qunmasj/good

## 11. 项目核心源码讲解（再也不用担心看不懂代码逻辑）

```以下是经过精简和注释的核心代码部分，主要集中在对象检测的任务对齐分配器（TaskAlignedAssigner）和相关的辅助函数上。  
  
```python  
import torch  
import torch.nn as nn  
  
def select\_candidates\_in\_gts(xy\_centers, gt\_bboxes, eps=1e-9):  
 """  
 选择在真实框(gt)中的正锚点中心。  
  
 参数:  
 xy\_centers (Tensor): 形状为(h\*w, 2)的张量，表示锚点中心坐标。  
 gt\_bboxes (Tensor): 形状为(b, n\_boxes, 4)的张量，表示真实框的坐标。  
  
 返回:  
 (Tensor): 形状为(b, n\_boxes, h\*w)的张量，表示每个真实框中是否包含锚点中心。  
 """  
 n\_anchors = xy\_centers.shape[0]  
 bs, n\_boxes, \_ = gt\_bboxes.shape  
 lt, rb = gt\_bboxes.view(-1, 1, 4).chunk(2, 2) # 左上角和右下角  
 bbox\_deltas = torch.cat((xy\_centers[None] - lt, rb - xy\_centers[None]), dim=2).view(bs, n\_boxes, n\_anchors, -1)  
 return bbox\_deltas.amin(3).gt\_(eps) # 返回是否大于eps的布尔值  
  
class TaskAlignedAssigner(nn.Module):  
 """  
 任务对齐分配器，用于对象检测。  
  
 属性:  
 topk (int): 考虑的前k个候选框数量。  
 num\_classes (int): 对象类别数量。  
 alpha (float): 分类组件的alpha参数。  
 beta (float): 定位组件的beta参数。  
 eps (float): 防止除以零的小值。  
 """  
  
 def \_\_init\_\_(self, topk=13, num\_classes=80, alpha=1.0, beta=6.0, eps=1e-9):  
 """初始化任务对齐分配器对象，设置超参数。"""  
 super().\_\_init\_\_()  
 self.topk = topk  
 self.num\_classes = num\_classes  
 self.bg\_idx = num\_classes # 背景类别索引  
 self.alpha = alpha  
 self.beta = beta  
 self.eps = eps  
  
 @torch.no\_grad()  
 def forward(self, pd\_scores, pd\_bboxes, anc\_points, gt\_labels, gt\_bboxes, mask\_gt):  
 """  
 计算任务对齐分配。  
  
 参数:  
 pd\_scores (Tensor): 形状为(bs, num\_total\_anchors, num\_classes)的张量，表示预测的分数。  
 pd\_bboxes (Tensor): 形状为(bs, num\_total\_anchors, 4)的张量，表示预测的边界框。  
 anc\_points (Tensor): 形状为(num\_total\_anchors, 2)的张量，表示锚点坐标。  
 gt\_labels (Tensor): 形状为(bs, n\_max\_boxes, 1)的张量，表示真实框的标签。  
 gt\_bboxes (Tensor): 形状为(bs, n\_max\_boxes, 4)的张量，表示真实框的坐标。  
 mask\_gt (Tensor): 形状为(bs, n\_max\_boxes, 1)的张量，表示真实框的掩码。  
  
 返回:  
 target\_labels (Tensor): 形状为(bs, num\_total\_anchors)的张量，表示目标标签。  
 target\_bboxes (Tensor): 形状为(bs, num\_total\_anchors, 4)的张量，表示目标边界框。  
 target\_scores (Tensor): 形状为(bs, num\_total\_anchors, num\_classes)的张量，表示目标分数。  
 fg\_mask (Tensor): 形状为(bs, num\_total\_anchors)的布尔张量，表示前景锚点。  
 target\_gt\_idx (Tensor): 形状为(bs, num\_total\_anchors)的张量，表示目标真实框索引。  
 """  
 self.bs = pd\_scores.size(0)  
 self.n\_max\_boxes = gt\_bboxes.size(1)  
  
 if self.n\_max\_boxes == 0:  
 device = gt\_bboxes.device  
 return (torch.full\_like(pd\_scores[..., 0], self.bg\_idx).to(device),   
 torch.zeros\_like(pd\_bboxes).to(device),  
 torch.zeros\_like(pd\_scores).to(device),   
 torch.zeros\_like(pd\_scores[..., 0]).to(device),  
 torch.zeros\_like(pd\_scores[..., 0]).to(device))  
  
 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 = select\_highest\_overlaps(mask\_pos, overlaps, self.n\_max\_boxes)  
  
 # 计算目标标签、边界框和分数  
 target\_labels, target\_bboxes, target\_scores = self.get\_targets(gt\_labels, gt\_bboxes, target\_gt\_idx, fg\_mask)  
  
 # 归一化  
 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  
  
 def get\_pos\_mask(self, pd\_scores, pd\_bboxes, gt\_labels, gt\_bboxes, anc\_points, mask\_gt):  
 """获取在真实框中的锚点掩码，返回正锚点掩码、对齐度量和重叠度量。"""  
 mask\_in\_gts = select\_candidates\_in\_gts(anc\_points, gt\_bboxes)  
 align\_metric, overlaps = self.get\_box\_metrics(pd\_scores, pd\_bboxes, gt\_labels, gt\_bboxes, mask\_in\_gts \* mask\_gt)  
 mask\_topk = self.select\_topk\_candidates(align\_metric, topk\_mask=mask\_gt.expand(-1, -1, self.topk).bool())  
 mask\_pos = mask\_topk \* mask\_in\_gts \* mask\_gt  
 return mask\_pos, align\_metric, overlaps  
  
 def get\_box\_metrics(self, pd\_scores, pd\_bboxes, gt\_labels, gt\_bboxes, mask\_gt):  
 """计算给定预测和真实边界框的对齐度量。"""  
 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  
 bbox\_scores[mask\_gt] = pd\_scores[ind[0], :, ind[1]][mask\_gt] # b, max\_num\_obj, h\*w  
  
 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] = bbox\_iou(gt\_boxes, pd\_boxes, xywh=False, CIoU=True).squeeze(-1).clamp\_(0)  
  
 align\_metric = bbox\_scores.pow(self.alpha) \* overlaps.pow(self.beta)  
 return align\_metric, overlaps  
  
 def get\_targets(self, gt\_labels, gt\_bboxes, target\_gt\_idx, fg\_mask):  
 """  
 计算正锚点的目标标签、目标边界框和目标分数。  
  
 参数:  
 gt\_labels (Tensor): 真实框标签，形状为(b, max\_num\_obj, 1)。  
 gt\_bboxes (Tensor): 真实框边界框，形状为(b, max\_num\_obj, 4)。  
 target\_gt\_idx (Tensor): 正锚点分配的真实框索引，形状为(b, h\*w)。  
 fg\_mask (Tensor): 布尔张量，形状为(b, h\*w)，表示前景锚点。  
  
 返回:  
 (Tuple[Tensor, Tensor, Tensor]): 包含目标标签、目标边界框和目标分数的元组。  
 """  
 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)  
  
 target\_bboxes = gt\_bboxes.view(-1, 4)[target\_gt\_idx] # (b, h\*w, 4)  
  
 target\_labels.clamp\_(0)  
  
 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  
```  
  
### 代码说明：  
1. \*\*选择锚点\*\*：`select\_candidates\_in\_gts` 函数用于选择在真实框中的锚点中心。  
2. \*\*任务对齐分配器\*\*：`TaskAlignedAssigner` 类负责将真实框分配给锚点，计算目标标签、边界框和分数。  
3. \*\*对齐度量计算\*\*：通过 `get\_box\_metrics` 方法计算预测分数和真实框之间的对齐度量。  
4. \*\*目标计算\*\*：`get\_targets` 方法根据分配的真实框索引计算目标标签、边界框和分数。  
  
这些核心部分是对象检测中锚点分配和目标计算的关键逻辑。```

这个文件是一个用于目标检测的任务对齐分配器（TaskAlignedAssigner）的实现，主要用于将真实目标（ground truth, gt）与锚框（anchors）进行匹配。该实现基于PyTorch框架，包含了一些重要的功能和方法，以下是对文件内容的详细讲解。  
  
首先，文件导入了必要的库，包括PyTorch的核心模块和一些自定义的检查和度量模块。`check\_version`用于检查PyTorch的版本，`bbox\_iou`和`wasserstein\_loss`用于计算边界框的交并比和Wasserstein损失。  
  
文件中定义了几个函数，首先是`select\_candidates\_in\_gts`，该函数用于选择在真实目标框内的锚框中心。它接受锚框中心和真实目标框作为输入，返回一个布尔张量，指示哪些锚框中心位于真实目标框内。  
  
接下来是`select\_highest\_overlaps`函数，该函数用于处理锚框与多个真实目标框的重叠情况。如果一个锚框被分配给多个真实目标框，它将选择与真实目标框重叠度最高的那个。  
  
`TaskAlignedAssigner`类是文件的核心部分。它继承自`nn.Module`，用于实现任务对齐的分配器。类的构造函数初始化了一些超参数，包括考虑的最大候选框数量、类别数量、分类和定位的权重参数等。  
  
在`forward`方法中，主要逻辑是计算任务对齐的分配。它接受预测的分数、边界框、锚框点、真实标签和边界框等输入，返回目标标签、目标边界框、目标分数、前景掩码和目标真实索引。该方法首先检查真实目标框的数量，如果没有真实目标框，则返回背景标签和零值张量。接着，调用`get\_pos\_mask`方法获取正样本掩码和对齐度量，随后调用`select\_highest\_overlaps`选择具有最高重叠的真实目标框。  
  
`get\_pos\_mask`方法用于获取在真实目标框内的锚框掩码，并计算对齐度量和重叠度。`get\_box\_metrics`方法计算预测边界框与真实边界框之间的对齐度量。`select\_topk\_candidates`方法根据给定的度量选择前k个候选框。  
  
`get\_targets`方法用于计算正样本的目标标签、目标边界框和目标分数。它根据分配的真实目标索引提取相应的标签和边界框，并生成目标分数。  
  
此外，文件还定义了一些辅助函数，例如`make\_anchors`用于生成锚框，`dist2bbox`用于将距离转换为边界框格式，`bbox2dist`用于将边界框转换为距离格式。  
  
整体而言，这个文件实现了一个复杂的目标检测分配机制，结合了分类和定位信息，以提高模型在训练过程中的性能。通过精确地将真实目标与锚框进行匹配，能够有效地提升目标检测的准确性。

```以下是代码中最核心的部分，并附上详细的中文注释：  
  
```python  
import sys  
import subprocess  
  
def run\_script(script\_path):  
 """  
 使用当前 Python 环境运行指定的脚本。  
  
 Args:  
 script\_path (str): 要运行的脚本路径  
  
 Returns:  
 None  
 """  
 # 获取当前 Python 解释器的路径  
 python\_path = sys.executable  
  
 # 构建运行命令，使用 streamlit 运行指定的脚本  
 command = f'"{python\_path}" -m streamlit run "{script\_path}"'  
  
 # 执行命令  
 result = subprocess.run(command, shell=True)  
 # 检查命令执行的返回码，如果不为0则表示出错  
 if result.returncode != 0:  
 print("脚本运行出错。")  
  
# 实例化并运行应用  
if \_\_name\_\_ == "\_\_main\_\_":  
 # 指定要运行的脚本路径  
 script\_path = "web.py" # 这里可以替换为实际的脚本路径  
  
 # 调用函数运行脚本  
 run\_script(script\_path)  
```  
  
### 代码注释说明：  
1. \*\*导入模块\*\*：  
 - `sys`：用于访问与 Python 解释器紧密相关的变量和函数。  
 - `subprocess`：用于执行外部命令和与其交互。  
  
2. \*\*定义 `run\_script` 函数\*\*：  
 - 该函数接收一个脚本路径作为参数，并使用当前 Python 环境运行该脚本。  
  
3. \*\*获取 Python 解释器路径\*\*：  
 - `sys.executable` 返回当前 Python 解释器的完整路径，确保在正确的环境中运行脚本。  
  
4. \*\*构建命令\*\*：  
 - 使用 f-string 格式化字符串构建运行命令，命令中包含了 Python 解释器的路径和要运行的脚本。  
  
5. \*\*执行命令\*\*：  
 - `subprocess.run` 方法用于执行构建的命令，并将 `shell=True` 作为参数以在 shell 中执行命令。  
  
6. \*\*检查返回码\*\*：  
 - 通过 `result.returncode` 检查命令的执行结果，如果返回码不为0，表示脚本运行过程中出现了错误。  
  
7. \*\*主程序入口\*\*：  
 - 使用 `if \_\_name\_\_ == "\_\_main\_\_":` 确保只有在直接运行该脚本时才会执行以下代码。  
 - 指定要运行的脚本路径，并调用 `run\_script` 函数执行该脚本。```

这个程序文件名为 `ui.py`，主要功能是通过当前的 Python 环境运行一个指定的脚本，具体是使用 Streamlit 框架来启动一个 Web 应用。  
  
首先，程序导入了必要的模块，包括 `sys`、`os` 和 `subprocess`。其中，`sys` 模块用于访问与 Python 解释器相关的变量和函数，`os` 模块提供了与操作系统交互的功能，而 `subprocess` 模块则用于创建新进程、连接到它们的输入/输出/错误管道，并获取它们的返回码。此外，还导入了 `abs\_path` 函数，该函数来自 `QtFusion.path` 模块，用于获取文件的绝对路径。  
  
接下来，定义了一个名为 `run\_script` 的函数，该函数接受一个参数 `script\_path`，表示要运行的脚本的路径。在函数内部，首先获取当前 Python 解释器的路径，并将其存储在 `python\_path` 变量中。然后，构建一个命令字符串，该命令使用 Streamlit 运行指定的脚本。命令的格式为 `"{python\_path}" -m streamlit run "{script\_path}"`，其中 `{python\_path}` 和 `{script\_path}` 会被实际的路径替换。  
  
随后，使用 `subprocess.run` 方法执行构建好的命令。这个方法会在 shell 中运行命令，并返回一个结果对象。通过检查 `result.returncode`，可以判断脚本是否成功运行。如果返回码不为 0，程序会输出一条错误信息，提示脚本运行出错。  
  
最后，在 `if \_\_name\_\_ == "\_\_main\_\_":` 这部分代码中，程序指定了要运行的脚本路径，这里使用 `abs\_path("web.py")` 来获取 `web.py` 文件的绝对路径。然后调用 `run\_script` 函数，传入这个路径，从而启动指定的 Web 应用。  
  
总体来说，这个程序的核心功能是通过当前 Python 环境启动一个 Streamlit 应用，提供了一种简单的方式来运行 Web 应用脚本。

```以下是代码中最核心的部分，并附上详细的中文注释：  
  
```python  
import numpy as np  
import torch  
from ultralytics.utils import LOGGER, colorstr  
from ultralytics.utils.torch\_utils import profile  
  
def check\_train\_batch\_size(model, imgsz=640, amp=True):  
 """  
 检查YOLO训练的最佳批量大小。  
  
 参数:  
 model (torch.nn.Module): 要检查批量大小的YOLO模型。  
 imgsz (int): 用于训练的图像大小。  
 amp (bool): 如果为True，则在训练中使用自动混合精度（AMP）。  
  
 返回:  
 (int): 使用autobatch()函数计算的最佳批量大小。  
 """  
 with torch.cuda.amp.autocast(amp):  
 return autobatch(deepcopy(model).train(), imgsz) # 计算最佳批量大小  
  
def autobatch(model, imgsz=640, fraction=0.60, batch\_size=16):  
 """  
 自动估计最佳YOLO批量大小，以使用可用CUDA内存的一部分。  
  
 参数:  
 model (torch.nn.Module): 要计算批量大小的YOLO模型。  
 imgsz (int, optional): 用作YOLO模型输入的图像大小。默认为640。  
 fraction (float, optional): 要使用的可用CUDA内存的比例。默认为0.60。  
 batch\_size (int, optional): 如果检测到错误，则使用的默认批量大小。默认为16。  
  
 返回:  
 (int): 最佳批量大小。  
 """  
   
 # 检查设备  
 prefix = colorstr('AutoBatch: ')  
 LOGGER.info(f'{prefix}计算imgsz={imgsz}的最佳批量大小')  
 device = next(model.parameters()).device # 获取模型设备  
 if device.type == 'cpu':  
 LOGGER.info(f'{prefix}未检测到CUDA，使用默认CPU批量大小 {batch\_size}')  
 return batch\_size  
 if torch.backends.cudnn.benchmark:  
 LOGGER.info(f'{prefix} ⚠️ 需要torch.backends.cudnn.benchmark=False，使用默认批量大小 {batch\_size}')  
 return batch\_size  
  
 # 检查CUDA内存  
 gb = 1 << 30 # 字节转GiB (1024 \*\* 3)  
 properties = torch.cuda.get\_device\_properties(device) # 获取设备属性  
 t = properties.total\_memory / gb # GiB总内存  
 r = torch.cuda.memory\_reserved(device) / gb # GiB保留内存  
 a = torch.cuda.memory\_allocated(device) / gb # GiB已分配内存  
 f = t - (r + a) # GiB可用内存  
 LOGGER.info(f'{prefix}{device} ({properties.name}) {t:.2f}G总内存, {r:.2f}G保留, {a:.2f}G已分配, {f:.2f}G可用')  
  
 # 配置批量大小  
 batch\_sizes = [1, 2, 4, 8, 16]  
 try:  
 img = [torch.empty(b, 3, imgsz, imgsz) for b in batch\_sizes] # 创建空图像张量  
 results = profile(img, model, n=3, device=device) # 评估不同批量大小的性能  
  
 # 拟合解决方案  
 y = [x[2] for x in results if x] # 提取内存使用情况  
 p = np.polyfit(batch\_sizes[:len(y)], y, deg=1) # 一次多项式拟合  
 b = int((f \* fraction - p[1]) / p[0]) # 计算最佳批量大小  
 if None in results: # 如果某些大小失败  
 i = results.index(None) # 找到第一个失败的索引  
 if b >= batch\_sizes[i]: # 如果最佳批量大小在失败点之上  
 b = batch\_sizes[max(i - 1, 0)] # 选择安全的前一个点  
 if b < 1 or b > 1024: # 如果最佳批量大小超出安全范围  
 b = batch\_size  
 LOGGER.info(f'{prefix}警告 ⚠️ 检测到CUDA异常，使用默认批量大小 {batch\_size}.')  
  
 fraction = (np.polyval(p, b) + r + a) / t # 预测的实际内存使用比例  
 LOGGER.info(f'{prefix}使用批量大小 {b}，在 {device} 上 {t \* fraction:.2f}G/{t:.2f}G ({fraction \* 100:.0f}%) ✅')  
 return b  
 except Exception as e:  
 LOGGER.warning(f'{prefix}警告 ⚠️ 检测到错误: {e}, 使用默认批量大小 {batch\_size}.')  
 return batch\_size  
```  
  
### 代码核心部分说明：  
1. \*\*check\_train\_batch\_size\*\*: 该函数用于检查给定YOLO模型的最佳训练批量大小。它会使用自动混合精度（AMP）来提高计算效率。  
   
2. \*\*autobatch\*\*: 该函数的主要功能是自动估计最佳批量大小，以便在训练时合理利用CUDA内存。它会检查设备类型、CUDA内存情况，并根据不同的批量大小进行性能评估，最终返回一个最佳的批量大小。  
  
3. \*\*CUDA内存检查\*\*: 代码中通过获取设备的总内存、已保留内存和已分配内存来计算可用内存，并输出相关信息。  
  
4. \*\*批量大小拟合\*\*: 通过对不同批量大小的内存使用情况进行线性拟合，来预测最佳的批量大小，并确保其在合理范围内。  
  
5. \*\*异常处理\*\*: 在执行过程中，如果发生任何异常，将记录警告并返回默认的批量大小。```

这个程序文件的主要功能是估算在使用PyTorch进行YOLO模型训练时，最佳的批处理大小（batch size），以便有效利用可用的CUDA内存。文件中定义了几个函数，主要包括`check\_train\_batch\_size`和`autobatch`。  
  
首先，`check\_train\_batch\_size`函数接收一个YOLO模型、图像大小和一个布尔值（表示是否使用自动混合精度）作为参数。它通过调用`autobatch`函数来计算最佳的批处理大小。在执行计算时，如果启用了自动混合精度（AMP），则会使用`torch.cuda.amp.autocast`上下文管理器来优化内存使用。  
  
接下来是`autobatch`函数，它是这个文件的核心。该函数接收YOLO模型、图像大小、可用CUDA内存的使用比例和默认批处理大小作为参数。函数首先检查模型所在的设备，如果设备是CPU，则返回默认的批处理大小。如果CUDA环境不符合要求（例如，`torch.backends.cudnn.benchmark`为True），也会返回默认的批处理大小。  
  
函数会检查CUDA内存的使用情况，包括总内存、已保留内存、已分配内存和可用内存，并将这些信息记录到日志中。接着，函数会定义一个批处理大小的列表（1, 2, 4, 8, 16），并为每个批处理大小创建一个空的图像张量。通过调用`profile`函数来评估不同批处理大小下的内存使用情况。  
  
在获取到内存使用数据后，函数会使用一阶多项式拟合来计算最佳的批处理大小。通过计算得到的y截距，函数可以估算出在给定的可用内存下，能够使用的最佳批处理大小。如果在某些批处理大小下出现了错误，函数会选择上一个安全的批处理大小作为备选。此外，函数还会检查计算出的批处理大小是否在安全范围内（1到1024之间），如果不在，则使用默认的批处理大小。  
  
最后，函数会记录使用的批处理大小和实际使用的内存比例，并返回计算出的最佳批处理大小。如果在计算过程中发生异常，函数会记录警告并返回默认的批处理大小。  
  
总体来说，这个文件提供了一种动态计算YOLO模型训练时最佳批处理大小的方法，以便在训练过程中高效利用GPU资源。

``````python  
# Ultralytics YOLO 🚀, AGPL-3.0 license  
  
from pathlib import Path # 导入Path类，用于处理文件路径  
from ultralytics.engine.model import Model # 从ultralytics库中导入Model类  
  
from .predict import FastSAMPredictor # 导入FastSAMPredictor类，用于预测  
from .val import FastSAMValidator # 导入FastSAMValidator类，用于验证  
  
  
class FastSAM(Model):  
 """  
 FastSAM模型接口。  
  
 示例：  
 ```python  
 from ultralytics import FastSAM  
  
 model = FastSAM('last.pt') # 创建FastSAM模型实例  
 results = model.predict('ultralytics/assets/bus.jpg') # 使用模型进行预测  
 ```  
 """  
  
 def \_\_init\_\_(self, model='FastSAM-x.pt'):  
 """初始化FastSAM类，调用父类Model的\_\_init\_\_方法，并设置默认模型。"""  
 # 如果传入的模型名称是'FastSAM.pt'，则将其更改为'FastSAM-x.pt'  
 if str(model) == 'FastSAM.pt':  
 model = 'FastSAM-x.pt'  
 # 确保模型文件后缀不是.yaml或.yml，FastSAM模型只支持预训练模型  
 assert Path(model).suffix not in ('.yaml', '.yml'), 'FastSAM models only support pre-trained models.'  
 # 调用父类的初始化方法，设置模型和任务类型为'segment'  
 super().\_\_init\_\_(model=model, task='segment')  
  
 @property  
 def task\_map(self):  
 """返回一个字典，将分割任务映射到相应的预测器和验证器类。"""  
 return {'segment': {'predictor': FastSAMPredictor, 'validator': FastSAMValidator}}  
```  
  
### 代码核心部分注释说明：  
1. \*\*导入模块\*\*：导入了处理文件路径的`Path`类，以及`Model`类和相关的预测、验证类。  
2. \*\*FastSAM类\*\*：继承自`Model`类，构建了一个FastSAM模型接口。  
3. \*\*初始化方法\*\*：在初始化时，检查模型名称并确保其后缀符合要求，调用父类的初始化方法以设置模型和任务类型。  
4. \*\*任务映射\*\*：定义了一个属性`task\_map`，返回一个字典，映射分割任务到相应的预测器和验证器类。```

这个程序文件定义了一个名为 `FastSAM` 的类，它是 Ultralytics YOLO 模型的一部分，主要用于图像分割任务。文件首先导入了一些必要的模块，包括 `Path` 类用于处理文件路径，以及从 `ultralytics.engine.model` 导入的 `Model` 类，后者是 `FastSAM` 类的父类。此外，还导入了 `FastSAMPredictor` 和 `FastSAMValidator`，这两个类分别用于预测和验证。  
  
`FastSAM` 类的文档字符串中给出了一个简单的使用示例，展示了如何加载模型并对一张图片进行预测。用户可以通过创建 `FastSAM` 类的实例并传入模型文件的路径来使用该模型，随后可以调用 `predict` 方法对指定的图像进行处理。  
  
在 `\_\_init\_\_` 方法中，类首先检查传入的模型名称。如果模型名称是 `'FastSAM.pt'`，则将其更改为 `'FastSAM-x.pt'`。接着，使用 `assert` 语句确保传入的模型文件不是 YAML 格式，因为 `FastSAM` 模型只支持预训练模型。最后，调用父类的 `\_\_init\_\_` 方法，传入模型路径和任务类型（这里是 `'segment'`，表示分割任务）。  
  
类中还有一个名为 `task\_map` 的属性，它返回一个字典，该字典将任务类型（在这里是 `'segment'`）映射到相应的预测器和验证器类。这使得在处理不同任务时，可以方便地找到对应的处理类。  
  
总体而言，这个文件的主要功能是定义一个接口，使得用户能够方便地使用 FastSAM 模型进行图像分割，并提供了必要的类和方法以支持这一功能。

```以下是保留的核心代码部分，并附上详细的中文注释：  
  
```python  
# 导入RTDETR模型、预测器和验证器  
from .model import RTDETR # 导入RTDETR模型类  
from .predict import RTDETRPredictor # 导入RTDETR预测器类  
from .val import RTDETRValidator # 导入RTDETR验证器类  
  
# 定义模块的公开接口，允许外部访问这些类  
\_\_all\_\_ = 'RTDETRPredictor', 'RTDETRValidator', 'RTDETR' # 指定可被外部导入的类  
```  
  
### 注释说明：  
1. \*\*导入部分\*\*：  
 - `from .model import RTDETR`：从当前模块的`model`文件中导入`RTDETR`类，这个类通常用于定义模型的结构和参数。  
 - `from .predict import RTDETRPredictor`：从当前模块的`predict`文件中导入`RTDETRPredictor`类，这个类用于对输入数据进行预测。  
 - `from .val import RTDETRValidator`：从当前模块的`val`文件中导入`RTDETRValidator`类，这个类用于验证模型的性能。  
  
2. \*\*公开接口\*\*：  
 - `\_\_all\_\_`：这是一个特殊的变量，用于定义当使用`from module import \*`时，哪些类或函数是可以被导入的。在这里，`RTDETRPredictor`、`RTDETRValidator`和`RTDETR`都是被允许导出的类。```

这个程序文件是一个Python模块的初始化文件，位于Ultralytics YOLO项目的RTDETR子目录下。文件的开头包含了一条注释，说明该项目使用的是AGPL-3.0许可证，并且标志着这是Ultralytics YOLO的一部分。  
  
接下来，文件通过相对导入的方式引入了三个类：`RTDETR`、`RTDETRPredictor`和`RTDETRValidator`。这些类分别定义在同一目录下的不同文件中，具体来说，`RTDETR`类在`model.py`文件中，`RTDETRPredictor`类在`predict.py`文件中，而`RTDETRValidator`类则在`val.py`文件中。  
  
最后，`\_\_all\_\_`变量被定义为一个包含字符串的元组，列出了该模块公开的接口。这个变量的作用是控制从该模块中使用`from module import \*`语句时，哪些名称会被导入。在这里，只有`RTDETRPredictor`、`RTDETRValidator`和`RTDETR`这三个类会被导入，其他未列出的名称则不会被导入。这种做法有助于维护模块的封装性和清晰性，确保用户只接触到模块的核心功能。

```以下是经过简化和注释的核心代码部分：  
  
```python  
import os  
import torch  
import yaml  
from ultralytics import YOLO # 导入YOLO模型库  
  
if \_\_name\_\_ == '\_\_main\_\_': # 确保该模块被直接运行时才执行以下代码  
 # 设置训练参数  
 workers = 1 # 数据加载的工作进程数  
 batch = 8 # 每个批次的样本数量  
 device = "0" if torch.cuda.is\_available() else "cpu" # 检查是否有可用的GPU，选择设备  
  
 # 获取数据集配置文件的绝对路径  
 data\_path = abs\_path(f'datasets/data/data.yaml', path\_type='current')  
  
 # 读取YAML文件，保持原有顺序  
 with open(data\_path, 'r') as file:  
 data = yaml.load(file, Loader=yaml.FullLoader)  
  
 # 修改数据集中训练、验证和测试的路径  
 if 'train' in data and 'val' in data and 'test' in data:  
 directory\_path = os.path.dirname(data\_path) # 获取目录路径  
 data['train'] = os.path.join(directory\_path, 'train') # 更新训练集路径  
 data['val'] = os.path.join(directory\_path, 'val') # 更新验证集路径  
 data['test'] = os.path.join(directory\_path, 'test') # 更新测试集路径  
  
 # 将修改后的数据写回YAML文件  
 with open(data\_path, 'w') as file:  
 yaml.safe\_dump(data, file, sort\_keys=False)  
  
 # 加载YOLO模型配置和预训练权重  
 model = YOLO(r"path/to/yolov8-seg-C2f-Faster.yaml").load("./weights/yolov8s-seg.pt")  
  
 # 开始训练模型  
 results = model.train(  
 data=data\_path, # 指定训练数据的配置文件路径  
 device=device, # 指定训练设备  
 workers=workers, # 数据加载的工作进程数  
 imgsz=640, # 输入图像的大小  
 epochs=100, # 训练的轮数  
 batch=batch, # 每个批次的样本数量  
 )  
```  
  
### 代码注释说明：  
1. \*\*导入必要的库\*\*：导入操作系统、PyTorch、YAML解析库和YOLO模型库。  
2. \*\*主程序入口\*\*：`if \_\_name\_\_ == '\_\_main\_\_':` 确保只有在直接运行该脚本时才执行后续代码。  
3. \*\*设置训练参数\*\*：  
 - `workers`：设置数据加载的工作进程数。  
 - `batch`：设置每个批次的样本数量。  
 - `device`：根据是否有可用的GPU选择训练设备。  
4. \*\*读取和修改数据集配置\*\*：  
 - 读取YAML文件并保持原有顺序。  
 - 更新训练、验证和测试数据集的路径。  
 - 将修改后的数据写回YAML文件。  
5. \*\*加载YOLO模型\*\*：指定模型配置文件和预训练权重。  
6. \*\*开始训练模型\*\*：调用`model.train()`方法进行模型训练，传入相关参数。```

该程序文件 `train.py` 是一个用于训练 YOLO（You Only Look Once）模型的脚本。首先，程序导入了必要的库，包括操作系统库 `os`、深度学习框架 `torch`、YAML 处理库 `yaml` 以及 YOLO 模型库 `ultralytics`。此外，还导入了 `QtFusion.path` 中的 `abs\_path` 函数，用于获取文件的绝对路径，并设置了 Matplotlib 的后端为 `TkAgg`。  
  
在 `\_\_main\_\_` 代码块中，程序首先定义了一些训练参数，包括工作进程数 `workers`、批次大小 `batch` 和设备选择 `device`。设备选择会根据是否有可用的 GPU 来决定，如果有则使用 GPU（设备编号为 "0"），否则使用 CPU。  
  
接下来，程序通过 `abs\_path` 函数获取数据集配置文件 `data.yaml` 的绝对路径，并将路径格式转换为 Unix 风格，以便于后续处理。然后，程序读取 YAML 文件，解析出数据集的相关信息。若 YAML 文件中包含 `train`、`val` 和 `test` 字段，程序会将这些字段的路径修改为相对于当前目录的路径，并将修改后的数据重新写回 YAML 文件。  
  
程序中还提到，不同的 YOLO 模型对设备的要求不同，因此如果遇到显存不足的错误，可以尝试使用其他模型配置文件进行测试。接着，程序加载了一个特定的 YOLO 模型配置文件，并加载了预训练的权重文件。  
  
最后，程序调用 `model.train` 方法开始训练模型，指定了训练数据的配置文件路径、设备、工作进程数、输入图像大小、训练的 epoch 数量以及批次大小等参数。训练过程会根据这些设置进行模型的训练。

### 整体功能和构架概括  
  
该项目是一个基于Ultralytics YOLO的目标检测和图像分割框架，提供了多种模型和工具，旨在实现高效的训练和推理。项目的架构包含了模型定义、数据处理、训练过程、自动批处理、用户界面等多个模块，支持不同的深度学习任务，如目标检测和图像分割。整体上，项目通过模块化设计，使得各个功能可以独立开发和维护，同时又能通过接口相互协作。  
  
### 文件功能整理表  
  
| 文件路径 | 功能描述 |  
|-----------------------------------------------|----------------------------------------------------------|  
| `ultralytics/utils/tal.py` | 实现目标检测的任务对齐分配器，计算最佳的锚框与真实目标的匹配。 |  
| `ui.py` | 启动一个Streamlit Web应用，用于可视化和交互操作。 |  
| `ultralytics/utils/autobatch.py` | 动态计算YOLO模型训练时的最佳批处理大小，以优化GPU内存使用。 |  
| `ultralytics/models/fastsam/model.py` | 定义FastSAM类，用于图像分割任务，提供模型加载和预测功能。 |  
| `ultralytics/models/rtdetr/\_\_init\_\_.py` | 初始化RTDETR模块，导入相关的模型、预测器和验证器类。 |  
| `train.py` | 训练YOLO模型的主脚本，处理数据集配置和训练参数设置。 |  
| `ultralytics/nn/extra\_modules/ops\_dcnv3/setup.py` | 配置和安装DCNv3模块的设置脚本。 |  
| `ultralytics/nn/backbone/VanillaNet.py` | 定义VanillaNet模型，作为YOLO模型的骨干网络之一。 |  
| `ultralytics/data/base.py` | 提供数据集的基本类和接口，用于数据加载和预处理。 |  
| `ultralytics/nn/backbone/EfficientFormerV2.py` | 定义EfficientFormerV2模型，作为YOLO模型的另一种骨干网络。 |  
| `ultralytics/data/augment.py` | 实现数据增强功能，提升模型的泛化能力。 |  
| `ultralytics/models/utils/\_\_init\_\_.py` | 初始化模型工具模块，导入相关的工具函数和类。 |  
| `ultralytics/data/annotator.py` | 提供图像标注功能，用于可视化检测结果和数据标注。 |  
  
这个表格总结了项目中各个文件的主要功能，帮助理解项目的整体结构和各个模块之间的关系。

注意：由于此博客编辑较早，上面“11.项目核心源码讲解（再也不用担心看不懂代码逻辑）”中部分代码可能会优化升级，仅供参考学习，完整“训练源码”、“Web前端界面”和“50+种创新点源码”以“14.完整训练+Web前端界面+50+种创新点源码、数据集获取”的内容为准。