# 昆虫实例分割系统源码＆数据集分享 [yolov8-seg-C2f-DCNV2-Dynamic＆yolov8-seg-EfficientHead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  
  
在生态学和生物多样性研究中，昆虫作为地球上最丰富的生物类群之一，扮演着重要的角色。昆虫不仅在生态系统中承担着重要的功能，如授粉、分解和食物链的维持，还在农业、医学和环境监测等领域中具有重要的应用价值。然而，随着全球气候变化和人类活动的加剧，昆虫种群的多样性和分布正面临严峻挑战。因此，开展昆虫的监测与研究，尤其是对其种类和数量的精确识别与分割，显得尤为重要。  
  
近年来，深度学习技术的迅猛发展为昆虫的实例分割提供了新的解决方案。YOLO（You Only Look Once）系列模型因其高效的实时检测能力而受到广泛关注。YOLOv8作为该系列的最新版本，结合了更为先进的特征提取和目标检测算法，展现出在多类目标检测中的优越性能。然而，针对昆虫的实例分割任务，现有的YOLOv8模型仍存在一些局限性，如对小目标的检测精度不足、背景复杂时的分割效果不佳等。因此，基于改进YOLOv8的昆虫实例分割系统的研究，具有重要的理论和实践意义。  
  
本研究将使用SnapHatch\_Segment数据集，该数据集包含1500幅图像，涵盖75个昆虫类别，涉及从成虫到幼虫、蛹等不同发育阶段的多样性。这一丰富的数据集为模型的训练和验证提供了坚实的基础，使得研究者能够在多样化的背景下，针对不同种类的昆虫进行有效的实例分割。通过对YOLOv8模型的改进，我们期望能够提升其在昆虫实例分割任务中的表现，尤其是在小目标检测和复杂背景下的分割精度。  
  
此外，昆虫实例分割系统的构建不仅能够为生态学研究提供高效的工具，还能够为生物多样性保护、生态监测和农业害虫管理等领域提供数据支持。通过精确识别和分割昆虫种类，研究者能够更好地评估生态系统的健康状况，监测物种的变化趋势，从而为生物多样性保护政策的制定提供科学依据。  
  
综上所述，基于改进YOLOv8的昆虫实例分割系统的研究，不仅具有重要的学术价值，也为实际应用提供了可行的解决方案。通过提升昆虫实例分割的精度和效率，我们能够更好地理解和保护这一重要的生物类群，为可持续发展和生态平衡贡献力量。

## 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“SnapHatch\_Segment”的数据集，以训练和改进YOLOv8-seg模型，旨在实现高效的昆虫实例分割。该数据集包含76个不同的昆虫类别，涵盖了广泛的生态群体，适用于多种环境和生态研究。数据集的多样性不仅体现在昆虫的种类上，还包括它们的生命周期阶段，如幼虫、成虫和蛹等，这为模型的训练提供了丰富的样本。  
  
“SnapHatch\_Segment”数据集的类别列表包括了常见的昆虫种类，如蚂蚁（Ant）、甲虫（beetle）、蝉（cicada）、蟋蟀（cricket）等，此外，还包括多种蜉蝣（mayfly）和石蝇（stonefly）的不同发育阶段。这些类别的多样性使得数据集在昆虫生态学、行为研究以及生物多样性监测等领域具有重要的应用价值。每个类别都经过精心标注，确保在训练过程中，模型能够准确识别和分割不同的昆虫实例。  
  
在具体的类别中，我们可以看到一些特定的昆虫种类，例如“caddis\_fly\_american\_grannom”和其不同生命周期阶段的变体，包括幼虫和蛹。这种细致的分类不仅有助于提高模型的分割精度，还为后续的生态学研究提供了重要的数据支持。此外，数据集中还包含了“mosquito”和“midges”等对公共卫生具有重要意义的昆虫，这使得该数据集在公共卫生监测和防治策略的制定中也具备了潜在的应用前景。  
  
为了确保数据集的有效性和代表性，SnapHatch\_Segment在采集过程中遵循了严格的标准，涵盖了多种生态环境和季节变化。这种多样性不仅增强了模型的泛化能力，也使得其在不同环境下的应用更为广泛。例如，数据集中包含的“grasshopper”和“water\_boatman”等昆虫，分别代表了陆地和水域生态系统中的重要组成部分，能够帮助研究人员理解不同生态系统中昆虫的生态角色。  
  
此外，数据集的设计还考虑到了昆虫的生活习性和生态交互。例如，蜉蝣的不同发育阶段反映了其在水域生态系统中的重要性，而石蝇的多样性则显示了其在流动水体中的生态适应性。这些信息不仅对模型训练至关重要，也为生态学研究提供了丰富的背景知识。  
  
综上所述，“SnapHatch\_Segment”数据集凭借其丰富的类别和详细的标注，为改进YOLOv8-seg的昆虫实例分割系统提供了坚实的基础。通过对该数据集的深入分析和应用，我们期望能够推动昆虫识别技术的发展，进而为生态保护和生物多样性研究提供更为精准和高效的工具。

## 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系列中的最新进展，结合了深度学习领域的前沿技术，特别是在目标检测和图像分割方面的应用。该算法在设计上遵循了YOLO系列一贯的高效性和实时性原则，同时通过一系列创新性改进，提升了模型的性能和适用性。YOLOv8-seg不仅在目标检测上表现优异，还能够实现精确的像素级分割，为复杂场景下的物体识别提供了强有力的支持。  
  
YOLOv8-seg的架构由三个主要部分组成：输入端、主干网络和检测端。输入端负责接收图像数据并进行预处理，以适应后续的特征提取和检测任务。主干网络则是算法的核心，负责从输入图像中提取多层次的特征信息。YOLOv8-seg采用了CSP结构和C2f模块，这一设计不仅提升了特征提取的效率，还增强了模型的表达能力。C2f模块通过引入更多的残差连接，能够在保持轻量化的同时，获取更加丰富的梯度信息，进而提升模型的训练效果。  
  
在特征融合方面，YOLOv8-seg继续采用PAN-FPN结构，这一结构通过自下而上的特征传递和自上而下的特征融合，有效地整合了不同尺度的特征信息。这种多尺度特征融合的策略，使得模型在处理不同大小物体时，能够保持较高的检测精度。同时，YOLOv8-seg在检测端采用了解耦合头结构，将分类和回归任务分开处理，进一步提升了模型的灵活性和准确性。  
  
YOLOv8-seg的创新之处还体现在其损失函数的设计上。算法采用了BCELoss作为分类损失，DFLLoss和CIoULoss作为回归损失，这种多重损失函数的组合，能够更好地平衡分类和定位的精度，确保模型在复杂场景下的鲁棒性。此外，YOLOv8-seg在训练过程中引入了动态Task-Aligned Assigner样本分配策略，能够根据不同任务的需求，灵活调整样本的分配方式，从而提高训练效率和模型的泛化能力。  
  
值得一提的是，YOLOv8-seg在数据增强方面也进行了创新，尤其是在训练的最后阶段，关闭了马赛克增强，这一策略的调整有助于模型更好地适应真实场景中的数据分布，减少过拟合的风险。通过这些改进，YOLOv8-seg在目标检测和图像分割任务中，展现出了更高的精度和更快的推理速度。  
  
YOLOv8-seg的应用场景非常广泛，尤其在农业、自动驾驶、安防监控等领域，均能发挥出其强大的视觉识别能力。以苹果采摘为例，YOLOv8-seg能够快速、准确地识别和定位苹果，为自动采摘机器人提供实时的决策支持。结合蚁群算法进行路径规划，能够使机器人在复杂的果园环境中，快速找到最佳采摘路径，显著提高采摘效率。  
  
总的来说，YOLOv8-seg算法通过一系列创新性设计和改进，提升了目标检测和图像分割的性能，为相关领域的应用提供了新的可能性。其高效的特征提取、多尺度特征融合、灵活的损失函数设计以及先进的数据增强策略，使得YOLOv8-seg在实际应用中表现出色，成为目标检测领域的重要工具。随着技术的不断进步，YOLOv8-seg有望在更多复杂场景中得到应用，推动视觉识别技术的发展。

## 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：  
  
```python  
# 引入必要的模块和类  
from ultralytics.engine.model import Model  
from ultralytics.models import yolo # 引入YOLO模型相关的模块  
from ultralytics.nn.tasks import ClassificationModel, DetectionModel, PoseModel, SegmentationModel  
  
class YOLO(Model):  
 """YOLO (You Only Look Once) 目标检测模型类。"""  
  
 @property  
 def task\_map(self):  
 """将任务类型映射到相应的模型、训练器、验证器和预测器类。"""  
 return {  
 'classify': { # 分类任务  
 'model': ClassificationModel, # 分类模型  
 'trainer': yolo.classify.ClassificationTrainer, # 分类训练器  
 'validator': yolo.classify.ClassificationValidator, # 分类验证器  
 'predictor': yolo.classify.ClassificationPredictor, # 分类预测器  
 },  
 'detect': { # 检测任务  
 'model': DetectionModel, # 检测模型  
 'trainer': yolo.detect.DetectionTrainer, # 检测训练器  
 'validator': yolo.detect.DetectionValidator, # 检测验证器  
 'predictor': yolo.detect.DetectionPredictor, # 检测预测器  
 },  
 'segment': { # 分割任务  
 'model': SegmentationModel, # 分割模型  
 'trainer': yolo.segment.SegmentationTrainer, # 分割训练器  
 'validator': yolo.segment.SegmentationValidator, # 分割验证器  
 'predictor': yolo.segment.SegmentationPredictor, # 分割预测器  
 },  
 'pose': { # 姿态估计任务  
 'model': PoseModel, # 姿态模型  
 'trainer': yolo.pose.PoseTrainer, # 姿态训练器  
 'validator': yolo.pose.PoseValidator, # 姿态验证器  
 'predictor': yolo.pose.PosePredictor, # 姿态预测器  
 },  
 }  
```  
  
### 代码说明：  
1. \*\*引入模块\*\*：首先引入了YOLO模型相关的模块和类，以便后续使用。  
2. \*\*YOLO类\*\*：定义了一个名为`YOLO`的类，继承自`Model`类，表示YOLO目标检测模型。  
3. \*\*task\_map属性\*\*：这是一个属性方法，返回一个字典，字典的键是任务类型（如分类、检测、分割、姿态），值是一个包含模型、训练器、验证器和预测器的字典。这样做的目的是为了根据不同的任务类型方便地获取相应的类。  
  
### 核心功能：  
- 通过`task\_map`，用户可以轻松地根据任务类型选择合适的模型和相关组件，增强了代码的可扩展性和灵活性。```

这个文件定义了一个名为 `YOLO` 的类，该类继承自 `Model`，主要用于实现 YOLO（You Only Look Once）目标检测模型。YOLO 是一种广泛使用的深度学习模型，主要用于实时目标检测。  
  
在 `YOLO` 类中，有一个名为 `task\_map` 的属性，它返回一个字典，该字典将不同的任务类型（如分类、检测、分割和姿态估计）映射到相应的模型、训练器、验证器和预测器类。这种设计使得 `YOLO` 类能够灵活地处理多种计算机视觉任务。  
  
具体来说，`task\_map` 字典的结构如下：  
  
- 对于分类任务（`classify`），它映射到 `ClassificationModel` 作为模型，`ClassificationTrainer` 作为训练器，`ClassificationValidator` 作为验证器，以及 `ClassificationPredictor` 作为预测器。  
- 对于检测任务（`detect`），它映射到 `DetectionModel`、`DetectionTrainer`、`DetectionValidator` 和 `DetectionPredictor`。  
- 对于分割任务（`segment`），它映射到 `SegmentationModel`、`SegmentationTrainer`、`SegmentationValidator` 和 `SegmentationPredictor`。  
- 对于姿态估计任务（`pose`），它映射到 `PoseModel`、`PoseTrainer`、`PoseValidator` 和 `PosePredictor`。  
  
这种结构化的映射关系使得在使用 YOLO 模型时，可以方便地选择适合特定任务的组件，提升了代码的可维护性和扩展性。总的来说，这个文件为 YOLO 模型的多任务处理提供了基础框架，便于后续的模型训练、验证和预测操作。

```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，并等待其完成  
 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\*\*：  
 - 该函数接受一个参数 `script\_path`，表示要运行的 Python 脚本的路径。  
 - 函数内部首先获取当前 Python 解释器的路径，然后构建一个命令字符串，使用 `streamlit` 来运行指定的脚本。  
  
3. \*\*执行命令\*\*：  
 - 使用 `subprocess.run` 方法执行构建的命令，并等待命令执行完成。  
 - 通过检查 `result.returncode` 来判断命令是否成功执行，如果返回码不为0，则打印错误信息。  
  
4. \*\*主程序入口\*\*：  
 - 在 `if \_\_name\_\_ == "\_\_main\_\_":` 块中，指定要运行的脚本路径，并调用 `run\_script` 函数来执行该脚本。```

这个程序文件的主要功能是通过当前的 Python 环境来运行一个指定的脚本，具体是一个名为 `web.py` 的文件。程序首先导入了必要的模块，包括 `sys`、`os` 和 `subprocess`，这些模块分别用于访问系统特性、操作系统功能和执行外部命令。  
  
在文件中定义了一个名为 `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。函数内部首先获取当前 Python 解释器的路径，这通过 `sys.executable` 实现。接着，构建一个命令字符串，使用 `streamlit` 模块来运行指定的脚本。这个命令的格式是将 Python 解释器的路径和要运行的脚本路径组合在一起，形成一个完整的命令。  
  
然后，使用 `subprocess.run` 方法来执行这个命令，`shell=True` 参数表示在一个新的 shell 中执行命令。执行后，程序会检查返回的结果码，如果结果码不为零，说明脚本运行过程中出现了错误，此时会打印出“脚本运行出错”的提示信息。  
  
在文件的最后部分，使用 `if \_\_name\_\_ == "\_\_main\_\_":` 语句来确保只有在直接运行该文件时才会执行下面的代码。这里指定了要运行的脚本路径为 `web.py`，并调用 `run\_script` 函数来执行这个脚本。  
  
整体来看，这个程序的目的是为了方便地通过 Python 环境来启动一个基于 Streamlit 的 Web 应用。

```以下是代码中最核心的部分，并附上详细的中文注释：  
  
```python  
import os  
import re  
import shutil  
import socket  
import sys  
import tempfile  
from pathlib import Path  
  
def find\_free\_network\_port() -> int:  
 """  
 查找本地主机上可用的端口。  
  
 在单节点训练时，如果不想连接到真实的主节点，但必须设置  
 `MASTER\_PORT` 环境变量时，这个函数非常有用。  
 """  
 with socket.socket(socket.AF\_INET, socket.SOCK\_STREAM) as s:  
 s.bind(('127.0.0.1', 0)) # 绑定到本地地址和随机端口  
 return s.getsockname()[1] # 返回绑定的端口号  
  
  
def generate\_ddp\_file(trainer):  
 """生成 DDP 文件并返回其文件名。"""  
 # 获取训练器的模块和类名  
 module, name = f'{trainer.\_\_class\_\_.\_\_module\_\_}.{trainer.\_\_class\_\_.\_\_name\_\_}'.rsplit('.', 1)  
  
 # 创建 DDP 文件的内容  
 content = f'''overrides = {vars(trainer.args)} \nif \_\_name\_\_ == "\_\_main\_\_":  
 from {module} import {name}  
 from ultralytics.utils import DEFAULT\_CFG\_DICT  
  
 cfg = DEFAULT\_CFG\_DICT.copy()  
 cfg.update(save\_dir='') # 处理额外的键 'save\_dir'  
 trainer = {name}(cfg=cfg, overrides=overrides)  
 trainer.train()'''  
   
 # 创建 DDP 目录  
 (USER\_CONFIG\_DIR / 'DDP').mkdir(exist\_ok=True)  
   
 # 创建临时文件并写入内容  
 with tempfile.NamedTemporaryFile(prefix='\_temp\_',  
 suffix=f'{id(trainer)}.py',  
 mode='w+',  
 encoding='utf-8',  
 dir=USER\_CONFIG\_DIR / 'DDP',  
 delete=False) as file:  
 file.write(content) # 写入生成的内容  
 return file.name # 返回临时文件的名称  
  
  
def generate\_ddp\_command(world\_size, trainer):  
 """生成并返回用于分布式训练的命令。"""  
 import \_\_main\_\_ # 本地导入以避免某些问题  
 if not trainer.resume:  
 shutil.rmtree(trainer.save\_dir) # 如果不恢复训练，删除保存目录  
  
 file = str(Path(sys.argv[0]).resolve()) # 获取当前脚本的绝对路径  
 safe\_pattern = re.compile(r'^[a-zA-Z0-9\_. /\\-]{1,128}$') # 允许的字符和最大长度  
 # 检查文件名是否安全且存在  
 if not (safe\_pattern.match(file) and Path(file).exists() and file.endswith('.py')):  
 file = generate\_ddp\_file(trainer) # 生成 DDP 文件  
  
 dist\_cmd = 'torch.distributed.run' if TORCH\_1\_9 else 'torch.distributed.launch' # 根据 PyTorch 版本选择命令  
 port = find\_free\_network\_port() # 查找可用端口  
 # 构建命令列表  
 cmd = [sys.executable, '-m', dist\_cmd, '--nproc\_per\_node', f'{world\_size}', '--master\_port', f'{port}', file]  
 return cmd, file # 返回命令和文件名  
  
  
def ddp\_cleanup(trainer, file):  
 """如果创建了临时文件，则删除它。"""  
 if f'{id(trainer)}.py' in file: # 检查文件名是否包含临时文件的后缀  
 os.remove(file) # 删除临时文件  
```  
  
### 代码核心功能概述：  
1. \*\*查找可用端口\*\*：`find\_free\_network\_port` 函数用于查找本地主机上可用的网络端口，方便进行分布式训练时的通信。  
2. \*\*生成 DDP 文件\*\*：`generate\_ddp\_file` 函数根据训练器的参数生成一个临时的 DDP 文件，以便在分布式训练中使用。  
3. \*\*生成训练命令\*\*：`generate\_ddp\_command` 函数生成用于启动分布式训练的命令，包括必要的参数和配置。  
4. \*\*清理临时文件\*\*：`ddp\_cleanup` 函数用于在训练结束后删除生成的临时文件，保持文件系统的整洁。```

这个程序文件是用于处理分布式训练的工具，主要涉及到在使用Ultralytics YOLO框架时的分布式数据并行（DDP）设置。代码的主要功能包括查找可用的网络端口、生成用于分布式训练的Python文件、生成分布式训练的命令以及清理临时文件。  
  
首先，程序导入了一些必要的模块，包括操作系统、正则表达式、文件处理、网络通信、系统信息、临时文件和路径处理等。它还导入了用户配置目录和Torch的版本信息。  
  
`find\_free\_network\_port`函数用于查找本地主机上可用的网络端口。这在单节点训练时非常有用，因为我们不想连接到真实的主节点，但仍需要设置`MASTER\_PORT`环境变量。该函数通过创建一个TCP套接字并绑定到本地地址和随机端口来实现，返回找到的可用端口号。  
  
`generate\_ddp\_file`函数用于生成一个DDP文件并返回其文件名。它根据传入的训练器对象生成相应的Python代码，并将其写入一个临时文件中。生成的代码包括训练器的参数和训练过程的调用。临时文件存储在用户配置目录下的`DDP`文件夹中，确保文件夹存在。  
  
`generate\_ddp\_command`函数用于生成分布式训练的命令。它首先检查训练器是否需要恢复训练，如果不需要，则删除保存目录。接着，它获取当前脚本的路径，并使用正则表达式检查文件名是否合法。如果文件名不符合要求，则调用`generate\_ddp\_file`生成一个新的文件。然后，构建分布式训练的命令，使用合适的Torch分布式运行模块，并查找可用的网络端口，最终返回命令和文件名。  
  
最后，`ddp\_cleanup`函数用于在训练结束后删除临时文件。如果生成的临时文件的后缀与训练器的ID匹配，则执行删除操作，以清理不再需要的文件。  
  
整体来看，这个文件为Ultralytics YOLO框架提供了分布式训练的支持，确保在多GPU环境下能够有效地管理训练过程。

```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  
  
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量  
 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  
  
 # 获取数据集的yaml配置文件的绝对路径  
 data\_path = abs\_path(f'datasets/data/data.yaml', path\_type='current')  
  
 # 将路径转换为Unix风格  
 unix\_style\_path = data\_path.replace(os.sep, '/')  
 # 获取目录路径  
 directory\_path = os.path.dirname(unix\_style\_path)  
  
 # 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YAML文件中的路径项  
 if 'train' in data and 'val' in data and 'test' in data:  
 data['train'] = directory\_path + '/train' # 更新训练集路径  
 data['val'] = directory\_path + '/val' # 更新验证集路径  
 data['test'] = 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"C:\codeseg\codenew\50+种YOLOv8算法改进源码大全和调试加载训练教程（非必要）\改进YOLOv8模型配置文件\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, # 使用的设备（GPU或CPU）  
 workers=workers, # 数据加载的工作进程数量  
 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. \*\*导入库\*\*：导入必要的库，包括操作系统、PyTorch、YAML解析器和YOLO模型。  
2. \*\*主程序入口\*\*：通过`if \_\_name\_\_ == '\_\_main\_\_':`确保代码只在直接运行时执行。  
3. \*\*训练参数设置\*\*：  
 - `workers`：设置数据加载的工作进程数量。  
 - `batch`：设置每个批次的样本数量。  
 - `device`：检查是否有可用的GPU，并设置为相应的设备。  
4. \*\*数据路径处理\*\*：  
 - 获取数据集配置文件的绝对路径，并转换为Unix风格的路径。  
 - 读取YAML文件，更新训练、验证和测试集的路径。  
5. \*\*模型加载\*\*：加载YOLO模型的配置文件和预训练权重。  
6. \*\*模型训练\*\*：调用`model.train()`方法开始训练，传入必要的参数，如数据路径、设备、工作进程数量、图像大小、训练轮数和批次大小。```

这个程序文件 `train.py` 是用于训练 YOLO（You Only Look Once）模型的脚本，主要使用了 PyTorch 框架和 Ultraytics 提供的 YOLO 实现。以下是对代码的逐行讲解。  
  
首先，程序导入了必要的库，包括 `os`、`torch`、`yaml`、`ultralytics` 中的 YOLO 模型、`QtFusion` 中的路径处理工具以及 `matplotlib`。其中，`matplotlib` 被设置为使用 `TkAgg` 后端，这通常用于图形界面的显示。  
  
接下来，程序在 `\_\_main\_\_` 块中开始执行。首先定义了一些训练参数，包括 `workers`（工作进程数）、`batch`（每个批次的样本数）和 `device`（计算设备，优先使用 GPU，如果不可用则使用 CPU）。`batch` 的值可以根据计算机的显存和内存进行调整，以避免显存溢出。  
  
然后，程序获取数据集配置文件的绝对路径，该配置文件是一个 YAML 文件，包含了训练、验证和测试数据的路径。使用 `abs\_path` 函数将相对路径转换为绝对路径，并将路径中的分隔符统一为 Unix 风格的斜杠。  
  
接下来，程序读取 YAML 文件并将其内容加载到 `data` 变量中。它检查 YAML 文件中是否包含 `train`、`val` 和 `test` 三个字段，如果存在，则将这些字段的值修改为当前目录下的 `train`、`val` 和 `test` 子目录的路径。修改后的数据会被写回到原 YAML 文件中，以确保后续训练能够正确找到数据集。  
  
程序中有一段注释，提醒用户不同的模型对设备的要求不同，如果当前模型在训练时出现错误，可以尝试使用其他模型配置文件。  
  
接下来，程序加载了一个 YOLO 模型的配置文件，并使用预训练的权重文件进行初始化。模型配置文件的路径和权重文件的路径都是硬编码的，用户需要根据自己的文件结构进行调整。  
  
最后，程序调用 `model.train()` 方法开始训练模型。训练过程中指定了数据配置文件的路径、计算设备、工作进程数、输入图像的大小（640x640）、训练的轮数（100个 epoch）以及每个批次的大小（8）。训练结果会被存储在 `results` 变量中。  
  
总体而言，这个脚本的功能是设置并启动 YOLO 模型的训练过程，确保数据路径正确，模型配置合适，并且可以根据硬件条件进行调整。

```以下是代码中最核心的部分，并附上详细的中文注释：  
  
```python  
import torch  
import torch.nn as nn  
from typing import List  
from torch import Tensor  
  
class Partial\_conv3(nn.Module):  
 """  
 实现部分卷积的类，用于在特定通道上进行卷积操作。  
 """  
  
 def \_\_init\_\_(self, dim, n\_div, forward):  
 super().\_\_init\_\_()  
 self.dim\_conv3 = dim // n\_div # 计算需要进行卷积的通道数  
 self.dim\_untouched = dim - self.dim\_conv3 # 计算未被卷积的通道数  
 self.partial\_conv3 = nn.Conv2d(self.dim\_conv3, self.dim\_conv3, 3, 1, 1, bias=False) # 定义卷积层  
  
 # 根据前向传播方式选择不同的前向传播函数  
 if forward == 'slicing':  
 self.forward = self.forward\_slicing  
 elif forward == 'split\_cat':  
 self.forward = self.forward\_split\_cat  
 else:  
 raise NotImplementedError  
  
 def forward\_slicing(self, x: Tensor) -> Tensor:  
 # 仅用于推理阶段  
 x = x.clone() # 克隆输入以保持原始输入不变  
 x[:, :self.dim\_conv3, :, :] = self.partial\_conv3(x[:, :self.dim\_conv3, :, :]) # 对前部分通道进行卷积  
 return x  
  
 def forward\_split\_cat(self, x: Tensor) -> Tensor:  
 # 用于训练和推理阶段  
 x1, x2 = torch.split(x, [self.dim\_conv3, self.dim\_untouched], dim=1) # 将输入分为两部分  
 x1 = self.partial\_conv3(x1) # 对第一部分进行卷积  
 x = torch.cat((x1, x2), 1) # 将卷积后的部分与未卷积的部分拼接  
 return x  
  
  
class MLPBlock(nn.Module):  
 """  
 实现多层感知机（MLP）块的类，包含卷积、归一化和激活函数。  
 """  
  
 def \_\_init\_\_(self, dim, n\_div, mlp\_ratio, drop\_path, layer\_scale\_init\_value, act\_layer, norm\_layer, pconv\_fw\_type):  
 super().\_\_init\_\_()  
 self.dim = dim  
 self.mlp\_ratio = mlp\_ratio  
 self.drop\_path = nn.Identity() if drop\_path <= 0 else DropPath(drop\_path) # 根据drop\_path值选择是否使用DropPath  
 self.n\_div = n\_div  
  
 mlp\_hidden\_dim = int(dim \* mlp\_ratio) # 计算MLP隐藏层的维度  
  
 # 定义MLP的层  
 mlp\_layer: List[nn.Module] = [  
 nn.Conv2d(dim, mlp\_hidden\_dim, 1, bias=False),  
 norm\_layer(mlp\_hidden\_dim),  
 act\_layer(),  
 nn.Conv2d(mlp\_hidden\_dim, dim, 1, bias=False)  
 ]  
  
 self.mlp = nn.Sequential(\*mlp\_layer) # 将MLP层组合成一个顺序模块  
 self.spatial\_mixing = Partial\_conv3(dim, n\_div, pconv\_fw\_type) # 定义空间混合层  
  
 def forward(self, x: Tensor) -> Tensor:  
 shortcut = x # 保存输入以便后续残差连接  
 x = self.spatial\_mixing(x) # 进行空间混合  
 x = shortcut + self.drop\_path(self.mlp(x)) # 添加残差连接  
 return x  
  
  
class FasterNet(nn.Module):  
 """  
 FasterNet模型的实现，包含多个阶段和嵌入层。  
 """  
  
 def \_\_init\_\_(self, in\_chans=3, num\_classes=1000, embed\_dim=96, depths=(1, 2, 8, 2), mlp\_ratio=2., n\_div=4,  
 patch\_size=4, patch\_stride=4, patch\_size2=2, patch\_stride2=2, patch\_norm=True, drop\_path\_rate=0.1,  
 layer\_scale\_init\_value=0, norm\_layer='BN', act\_layer='RELU', pconv\_fw\_type='split\_cat'):  
 super().\_\_init\_\_()  
  
 # 选择归一化层和激活函数  
 norm\_layer = nn.BatchNorm2d if norm\_layer == 'BN' else NotImplementedError  
 act\_layer = nn.GELU if act\_layer == 'GELU' else partial(nn.ReLU, inplace=True)  
  
 self.num\_stages = len(depths) # 计算阶段数量  
 self.embed\_dim = embed\_dim # 嵌入维度  
 self.patch\_embed = PatchEmbed(patch\_size, patch\_stride, in\_chans, embed\_dim, norm\_layer if patch\_norm else None) # 定义Patch嵌入层  
  
 # 构建每个阶段的层  
 stages\_list = []  
 for i\_stage in range(self.num\_stages):  
 stage = BasicStage(dim=int(embed\_dim \* 2 \*\* i\_stage), n\_div=n\_div, depth=depths[i\_stage],  
 mlp\_ratio=mlp\_ratio, drop\_path=drop\_path\_rate, layer\_scale\_init\_value=layer\_scale\_init\_value,  
 norm\_layer=norm\_layer, act\_layer=act\_layer, pconv\_fw\_type=pconv\_fw\_type)  
 stages\_list.append(stage)  
  
 # 添加Patch合并层  
 if i\_stage < self.num\_stages - 1:  
 stages\_list.append(PatchMerging(patch\_size2=patch\_size2, patch\_stride2=patch\_stride2,  
 dim=int(embed\_dim \* 2 \*\* i\_stage), norm\_layer=norm\_layer))  
  
 self.stages = nn.Sequential(\*stages\_list) # 将所有阶段组合成一个顺序模块  
  
 def forward(self, x: Tensor) -> Tensor:  
 # 输出四个阶段的特征用于密集预测  
 x = self.patch\_embed(x) # 进行Patch嵌入  
 outs = []  
 for idx, stage in enumerate(self.stages):  
 x = stage(x) # 通过每个阶段  
 outs.append(x) # 保存输出  
 return outs  
```  
  
### 代码核心部分解释：  
1. \*\*Partial\_conv3\*\*: 该类实现了部分卷积，允许在输入的特定通道上进行卷积操作，支持两种前向传播方式（切片和拼接）。  
2. \*\*MLPBlock\*\*: 该类实现了多层感知机块，包含卷积、归一化和激活函数，支持残差连接。  
3. \*\*FasterNet\*\*: 该类是整个网络的主结构，包含多个阶段和嵌入层，负责处理输入并输出特征。通过组合不同的模块，构建出一个深度学习模型。  
  
以上是对代码的核心部分及其功能的详细注释。```

这个程序文件实现了一个名为FasterNet的深度学习模型，主要用于图像处理任务。文件中包含了多个类和函数，构成了FasterNet的结构和功能。  
  
首先，文件引入了一些必要的库，包括PyTorch和一些模型层的构建模块。接着，定义了一些基础的组件类，例如Partial\_conv3、MLPBlock、BasicStage、PatchEmbed和PatchMerging等。  
  
Partial\_conv3类实现了一个部分卷积层，可以根据不同的前向传播方式（如切片或拼接）来处理输入张量。MLPBlock类则是一个多层感知机模块，包含了卷积层、归一化层和激活函数，并支持残差连接。BasicStage类由多个MLPBlock组成，形成了网络的一个阶段。PatchEmbed和PatchMerging类用于将输入图像分割成补丁并进行合并，以便在不同的网络阶段中处理。  
  
FasterNet类是整个模型的核心，初始化时接受多个参数，如输入通道数、类别数、嵌入维度、深度、激活函数类型等。它通过调用前面定义的类构建了模型的各个部分，包括补丁嵌入、多个阶段和归一化层。模型的前向传播方法会依次通过各个阶段，并在指定的输出层进行归一化处理，最终返回特征图。  
  
此外，文件中还定义了一些函数，用于加载不同配置的FasterNet模型并更新权重。update\_weight函数用于将预训练权重加载到模型中，确保权重的形状匹配。fasternet\_t0、fasternet\_t1等函数则是针对不同模型配置的构造函数，加载相应的配置文件和权重。  
  
最后，在主程序部分，示例代码展示了如何创建一个FasterNet模型实例，并对输入进行处理，输出每个阶段的特征图的尺寸。这部分代码可以作为模型使用的基本示例。  
  
整体来看，这个文件实现了一个灵活且高效的深度学习模型结构，适用于图像分类或其他视觉任务，并提供了方便的模型构建和权重加载功能。

```以下是代码中最核心的部分，并附上详细的中文注释：  
  
```python  
import torch  
import torch.nn as nn  
import torch.nn.functional as F  
import itertools  
  
class Conv2d\_BN(torch.nn.Sequential):  
 """   
 结合卷积层和批归一化层的模块  
 """  
 def \_\_init\_\_(self, in\_channels, out\_channels, kernel\_size=1, stride=1, padding=0, dilation=1,  
 groups=1, bn\_weight\_init=1):  
 super().\_\_init\_\_()  
 # 添加卷积层  
 self.add\_module('c', torch.nn.Conv2d(  
 in\_channels, out\_channels, kernel\_size, stride, padding, dilation, groups, bias=False))  
 # 添加批归一化层  
 self.add\_module('bn', torch.nn.BatchNorm2d(out\_channels))  
 # 初始化批归一化层的权重和偏置  
 torch.nn.init.constant\_(self.bn.weight, bn\_weight\_init)  
 torch.nn.init.constant\_(self.bn.bias, 0)  
  
 @torch.no\_grad()  
 def switch\_to\_deploy(self):  
 """  
 将模型切换到推理模式，合并卷积和批归一化层  
 """  
 c, bn = self.\_modules.values()  
 # 计算新的卷积权重和偏置  
 w = bn.weight / (bn.running\_var + bn.eps)\*\*0.5  
 w = c.weight \* w[:, None, None, None]  
 b = bn.bias - bn.running\_mean \* bn.weight / (bn.running\_var + bn.eps)\*\*0.5  
 # 创建新的卷积层  
 m = torch.nn.Conv2d(w.size(1) \* self.c.groups, w.size(0), w.shape[2:], stride=self.c.stride, padding=self.c.padding, dilation=self.c.dilation, groups=self.c.groups)  
 m.weight.data.copy\_(w)  
 m.bias.data.copy\_(b)  
 return m  
  
class EfficientViTBlock(torch.nn.Module):  
 """   
 EfficientViT的基本构建块  
 """  
 def \_\_init\_\_(self, type, ed, kd, nh=8, ar=4, resolution=14, window\_resolution=7):  
 super().\_\_init\_\_()  
 # 深度卷积和前馈网络  
 self.dw0 = Residual(Conv2d\_BN(ed, ed, 3, 1, 1, groups=ed, bn\_weight\_init=0.))  
 self.ffn0 = Residual(FFN(ed, int(ed \* 2)))  
  
 if type == 's':  
 # 使用局部窗口注意力机制  
 self.mixer = Residual(LocalWindowAttention(ed, kd, nh, attn\_ratio=ar, resolution=resolution, window\_resolution=window\_resolution))  
  
 self.dw1 = Residual(Conv2d\_BN(ed, ed, 3, 1, 1, groups=ed, bn\_weight\_init=0.))  
 self.ffn1 = Residual(FFN(ed, int(ed \* 2)))  
  
 def forward(self, x):  
 # 前向传播  
 return self.ffn1(self.dw1(self.mixer(self.ffn0(self.dw0(x)))))  
  
class EfficientViT(torch.nn.Module):  
 """   
 EfficientViT模型  
 """  
 def \_\_init\_\_(self, img\_size=400, patch\_size=16, in\_chans=3, stages=['s', 's', 's'],  
 embed\_dim=[64, 128, 192], key\_dim=[16, 16, 16], depth=[1, 2, 3], num\_heads=[4, 4, 4]):  
 super().\_\_init\_\_()  
  
 # 图像嵌入层  
 self.patch\_embed = torch.nn.Sequential(  
 Conv2d\_BN(in\_chans, embed\_dim[0] // 8, 3, 2, 1),  
 torch.nn.ReLU(),  
 Conv2d\_BN(embed\_dim[0] // 8, embed\_dim[0] // 4, 3, 2, 1),  
 torch.nn.ReLU(),  
 Conv2d\_BN(embed\_dim[0] // 4, embed\_dim[0] // 2, 3, 2, 1),  
 torch.nn.ReLU(),  
 Conv2d\_BN(embed\_dim[0] // 2, embed\_dim[0], 3, 1, 1)  
 )  
  
 # 构建各个阶段的块  
 self.blocks = []  
 for i, (stg, ed, kd, dpth, nh) in enumerate(zip(stages, embed\_dim, key\_dim, depth, num\_heads)):  
 for d in range(dpth):  
 self.blocks.append(EfficientViTBlock(stg, ed, kd, nh))  
 self.blocks = torch.nn.Sequential(\*self.blocks)  
  
 def forward(self, x):  
 # 前向传播  
 x = self.patch\_embed(x)  
 x = self.blocks(x)  
 return x  
  
# 创建模型实例  
if \_\_name\_\_ == '\_\_main\_\_':  
 model = EfficientViT(img\_size=224)  
 inputs = torch.randn((1, 3, 640, 640))  
 res = model(inputs)  
 for i in res:  
 print(i.size())  
```  
  
### 代码核心部分说明：  
1. \*\*Conv2d\_BN\*\*: 这个类将卷积层和批归一化层结合在一起，方便构建卷积神经网络。`switch\_to\_deploy`方法用于将训练模式下的模型转换为推理模式，合并卷积和批归一化层以提高推理效率。  
  
2. \*\*EfficientViTBlock\*\*: 这是EfficientViT的基本构建块，包含深度卷积、前馈网络和局部窗口注意力机制。通过组合这些模块来实现特征提取和增强。  
  
3. \*\*EfficientViT\*\*: 这是整个模型的主类，负责构建整个网络结构，包括图像嵌入层和多个EfficientViT块。前向传播方法将输入数据通过嵌入层和各个块进行处理。  
  
4. \*\*主程序\*\*: 在主程序中创建了一个EfficientViT模型实例，并对随机输入进行前向传播，输出每个阶段的特征图大小。```

这个程序文件实现了一个名为 EfficientViT 的深度学习模型架构，主要用于图像处理任务。该模型的设计灵感来源于视觉变换器（Vision Transformer），并结合了高效的卷积操作和注意力机制，以提高计算效率和性能。  
  
文件开头部分包含了一些版权信息和必要的库导入，包括 PyTorch 和一些辅助模块。接下来，定义了一些模型的基础组件，例如 Conv2d\_BN 类，它将卷积层和批归一化层结合在一起，并提供了一个方法用于在推理时切换到部署模式，优化模型的计算。  
  
在 PatchMerging 类中，模型通过合并图像的补丁来减少空间维度，同时增加通道数。Residual 类实现了残差连接，允许信息在网络中流动，防止梯度消失。FFN 类实现了前馈神经网络模块，包含两个卷积层和一个激活函数。  
  
CascadedGroupAttention 和 LocalWindowAttention 类实现了不同类型的注意力机制，用于在图像的局部窗口内进行信息交互。EfficientViTBlock 类则是模型的基本构建块，结合了卷积、前馈网络和注意力机制，形成了一个完整的模块。  
  
EfficientViT 类是整个模型的核心，包含了多个构建块和补丁嵌入层。模型的构造函数接受多个参数，例如图像大小、补丁大小、嵌入维度、深度、注意力头数等。通过这些参数，模型可以根据需求进行调整。  
  
在模型的前向传播中，输入图像经过补丁嵌入和多个构建块的处理，最终输出多个特征图。文件中还定义了多个不同配置的 EfficientViT 模型（如 EfficientViT\_m0 到 EfficientViT\_m5），每个模型具有不同的参数设置。  
  
最后，程序提供了一些辅助函数，如 update\_weight，用于加载预训练模型的权重，并确保权重的形状与当前模型匹配。主函数部分展示了如何实例化一个 EfficientViT 模型，并对随机生成的输入进行前向传播，输出各层的特征图尺寸。  
  
整体来看，这个文件实现了一个灵活且高效的视觉变换器模型，适用于各种下游任务，具有良好的可扩展性和适应性。

### 整体功能和构架概括  
  
该项目是一个基于 Ultralytics YOLO 框架的深度学习目标检测和图像处理系统。它提供了多种模型架构（如 YOLO、FasterNet 和 EfficientViT），支持分布式训练、模型训练、验证和推理等功能。项目结构清晰，模块化设计使得不同功能的实现相对独立，便于维护和扩展。  
  
- \*\*模型定义\*\*：提供了 YOLO、FasterNet 和 EfficientViT 等模型的实现，支持多种计算机视觉任务。  
- \*\*训练与验证\*\*：提供了训练和验证的脚本，支持用户自定义数据集和训练参数。  
- \*\*分布式训练\*\*：支持分布式训练，优化了多 GPU 环境下的训练效率。  
- \*\*用户界面\*\*：提供了一个简单的用户界面，用于启动和管理训练过程。  
- \*\*工具函数\*\*：包含了一些实用的工具函数，支持模型权重加载、图像处理等功能。  
  
### 文件功能整理表  
  
| 文件路径 | 功能描述 |  
|----------------------------------------------------|------------------------------------------------------------------------------------------|  
| `ultralytics/models/yolo/model.py` | 定义 YOLO 模型类及其任务映射，支持多种计算机视觉任务（分类、检测、分割、姿态估计）。 |  
| `ui.py` | 提供一个用户界面，通过命令行启动指定的 Streamlit 脚本（如 web.py）。 |  
| `ultralytics/utils/dist.py` | 实现分布式训练的工具，包括查找可用网络端口、生成 DDP 文件和命令、清理临时文件等功能。 |  
| `train.py` | 启动 YOLO 模型的训练过程，处理数据集路径和训练参数，调用模型进行训练。 |  
| `ultralytics/nn/backbone/fasternet.py` | 实现 FasterNet 模型结构，包含多个基础组件和前向传播方法，适用于图像处理任务。 |  
| `ultralytics/nn/backbone/efficientViT.py` | 实现 EfficientViT 模型结构，结合卷积和注意力机制，支持图像处理任务。 |  
| `ultralytics/trackers/bot\_sort.py` | 实现 BotSort 跟踪算法，用于目标跟踪任务，结合检测结果进行目标的持续跟踪。 |  
| `ultralytics/models/yolo/detect/val.py` | 实现 YOLO 模型的验证过程，评估模型在验证集上的性能，计算指标如 mAP（平均精度均值）。 |  
| `ultralytics/models/yolo/pose/val.py` | 实现 YOLO 模型的姿态估计验证过程，评估模型在姿态估计任务上的性能。 |  
| `ultralytics/hub/utils.py` | 提供一些实用的工具函数，支持模型的加载、保存和配置等功能。 |  
| `ultralytics/utils/ops.py` | 包含一些操作相关的工具函数，可能涉及图像处理、数据增强等功能。 |  
| `ultralytics/utils/torch\_utils.py` | 提供与 PyTorch 相关的工具函数，如设备管理、模型权重加载等。 |  
| `ultralytics/engine/exporter.py` | 实现模型导出功能，将训练好的模型导出为不同格式（如 ONNX、TorchScript），以便于部署和推理。 |  
  
这个表格概述了项目中各个文件的主要功能，帮助理解整个系统的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。