# 屋顶图像分割系统源码＆数据集分享 [yolov8-seg-C2f-Parc＆yolov8-seg-vanillanet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  
  
随着城市化进程的加快，屋顶作为建筑的重要组成部分，其形态、材料及功能的多样性使得屋顶图像的分析与处理成为了计算机视觉领域的重要研究方向。屋顶图像分割不仅在建筑设计、城市规划中具有重要应用价值，同时也为环境监测、灾害评估等提供了有效的技术支持。近年来，深度学习技术的迅猛发展，尤其是目标检测与图像分割领域的进步，为屋顶图像的自动化分析提供了新的机遇。其中，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，尤其是YOLOv8的推出，更是为图像分割任务提供了更为强大的技术基础。  
  
本研究旨在基于改进的YOLOv8模型，构建一个高效的屋顶图像分割系统。我们所使用的数据集包含2900张屋顶图像，涵盖了四个类别，包括三种屋顶类型和背景类。这一数据集的构建不仅为模型的训练提供了丰富的样本，同时也为后续的性能评估奠定了基础。通过对不同屋顶类型的有效分割，我们能够更好地理解屋顶的结构特征与功能属性，从而为相关领域的研究提供数据支持。  
  
在当前的研究中，屋顶图像分割面临着诸多挑战，如复杂的背景、不同的光照条件以及屋顶形态的多样性等。传统的图像处理方法往往难以应对这些复杂性，而基于深度学习的图像分割技术则展现出了优越的性能。YOLOv8模型在处理实时性和准确性方面的优势，使其成为解决屋顶图像分割问题的理想选择。通过对YOLOv8模型的改进，我们期望在分割精度和处理速度上实现更好的平衡，从而提升屋顶图像分析的效率。  
  
此外，屋顶图像分割的研究不仅具有学术价值，也具有重要的社会意义。随着气候变化和城市热岛效应的加剧，屋顶的设计与材料选择对城市环境的影响日益显著。通过对屋顶图像的精确分割与分析，我们能够为城市规划者提供科学依据，帮助其在屋顶绿化、太阳能利用等方面做出更为合理的决策。同时，屋顶图像分割技术在灾后评估、建筑物安全检测等领域的应用，能够有效提升城市的韧性与安全性。  
  
综上所述，基于改进YOLOv8的屋顶图像分割系统的研究，不仅能够推动计算机视觉技术的发展，也将为城市建设与管理提供切实可行的解决方案。通过本研究，我们希望能够为屋顶图像分割领域的学术研究与实际应用贡献新的思路与方法，从而促进相关领域的进一步发展。

## 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“roof”的数据集，旨在训练和改进YOLOv8-seg模型，以实现高效的屋顶图像分割。该数据集专门设计用于屋顶相关的图像处理任务，具有良好的代表性和多样性，能够为模型的训练提供丰富的样本支持。数据集包含三种主要类别，分别标记为‘1’，‘2’和‘3’，这些类别的选择经过精心考虑，以确保模型能够有效地识别和分割不同类型的屋顶结构。  
  
首先，类别‘1’可能代表一种特定类型的屋顶，比如平屋顶或坡屋顶，这类屋顶在城市建筑中相对常见，具有独特的形状和结构特征。通过对这一类别的图像进行标注和训练，模型能够学习到如何识别这种屋顶的特征，如边缘、角度和表面纹理等。这对于后续的图像分割任务至关重要，因为模型需要能够准确地分离出屋顶与周围环境的界限，从而实现精确的分割效果。  
  
类别‘2’则可能对应于另一种屋顶类型，例如瓦片屋顶或金属屋顶。这类屋顶在建筑设计中也占有重要地位，具有不同的材料和视觉效果。通过引入这一类别的数据，模型可以扩展其识别能力，适应更多样化的屋顶结构。特别是在处理复杂场景时，能够有效区分不同类型的屋顶，将有助于提高分割的准确性和鲁棒性。  
  
最后，类别‘3’可能代表一种特殊的屋顶设计，例如绿色屋顶或太阳能屋顶。这类屋顶不仅在功能上具有特殊性，而且在视觉上也可能与传统屋顶有显著差异。通过对这一类别的学习，模型能够捕捉到更为复杂的屋顶特征，进一步提升其分割性能。这种多样化的类别设置，不仅丰富了数据集的内容，也为模型的训练提供了更为全面的视角。  
  
在数据集的构建过程中，所有图像均经过严格的标注和筛选，以确保数据的质量和准确性。每个类别的样本数量经过精心设计，以保证模型在训练过程中能够获得均衡的学习机会。此外，数据集中的图像涵盖了不同的光照条件、视角和背景，这些因素将进一步增强模型的泛化能力，使其能够在实际应用中表现出色。  
  
通过对“roof”数据集的深入分析和应用，我们期望能够显著提升YOLOv8-seg在屋顶图像分割任务中的表现。这不仅将推动计算机视觉领域的研究进展，也将为城市规划、建筑设计和环境监测等实际应用提供有力的技术支持。最终，我们希望通过这一研究，能够为屋顶图像的自动化处理提供一个高效、准确的解决方案，为相关领域的研究和应用开辟新的方向。

## 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作为YOLO系列中的最新版本，标志着目标检测和分割技术的又一次重大进步。其核心思想是通过高效的网络结构和创新的算法设计，提升目标检测的精度和速度，同时增强对复杂场景中小目标的感知能力。YOLOv8-seg不仅延续了YOLO系列一贯的anchor-free检测方法，还在特征提取和处理流程中引入了多种新技术，以适应日益复杂的应用场景。  
  
首先，YOLOv8-seg的网络结构主要由输入端、主干网络、Neck端和输出端四个模块组成。输入端负责对输入图像进行预处理，包括Mosaic数据增强、自适应图像缩放和灰度填充等，这些步骤旨在提升模型对不同场景和条件下的适应性。通过这些预处理技术，YOLOv8-seg能够有效地扩展训练数据集，从而提高模型的泛化能力。  
  
在主干网络中，YOLOv8-seg采用了更深的卷积神经网络结构，结合了卷积、池化等操作，提取图像的多层次特征。与以往的YOLO版本相比，YOLOv8-seg在特征提取过程中引入了C2f和SPPF结构，这些结构通过更高效的特征提取机制，增强了模型对复杂背景和小目标的感知能力。特别是在复杂水面环境下，YOLOv8-seg能够更好地捕捉小目标漂浮物的特征，减少定位误差。  
  
Neck端的设计是YOLOv8-seg的一大亮点。它基于路径聚合网络（PAN）结构，通过上采样、下采样和特征拼接的方式，融合不同尺度的特征图。这种特征融合策略使得YOLOv8-seg能够在多尺度上进行目标检测，进一步提升了对小目标的检测能力。此外，Neck端还引入了GSConv和Slim-neck技术，这些技术不仅保持了检测精度，还有效降低了计算量，使得YOLOv8-seg在处理复杂场景时依然能够保持高效的实时性能。  
  
在输出端，YOLOv8-seg采用了decoupled head结构，实现了分类和回归过程的解耦。这种设计使得模型在进行目标检测时，能够更好地匹配正负样本，并进行精确的损失计算。具体而言，YOLOv8-seg使用了Task-Aligned Assigner方法，对分类分数和回归分数进行加权，从而提高了样本匹配的准确性。在损失计算方面，YOLOv8-seg采用了BCE（binary cross-entropy）计算分类损失，同时使用DFL（distribution focal loss）和CIoU（complete intersection over union）损失函数计算回归损失，这一组合进一步提升了模型的精度和鲁棒性。  
  
然而，尽管YOLOv8-seg在多个方面进行了优化，但在复杂水面环境下，仍然面临小目标漏检和定位误差的问题。为了解决这些挑战，YOLOv8-seg的改进版本——YOLOv8-WSSOD算法应运而生。该算法在YOLOv8-seg的基础上，首先引入了BiFormer双层路由注意力机制，构建了C2fBF模块，以减轻主干网络下采样过程中的噪声影响。这一机制能够更好地保留特征提取过程中细粒度的上下文信息，从而提升模型对小目标的感知能力。  
  
其次，YOLOv8-WSSOD在网络中添加了一个更小的检测头，专门用于提升对小目标的检测能力。这一改进使得模型在处理小目标时，能够更加敏感，减少漏检现象。此外，在Neck端的GSConv和Slim-neck技术的引入，不仅保持了模型的检测精度，还有效降低了计算量，使得YOLOv8-WSSOD在实时性和准确性之间达到了更好的平衡。  
  
最后，YOLOv8-WSSOD还使用了MPDIoU损失函数替换了CIoU损失函数，这一变化显著提高了模型的泛化能力和精准度。MPDIoU损失函数通过考虑目标框的多个维度信息，使得模型在训练过程中能够更好地适应不同场景下的目标检测任务。  
  
综上所述，YOLOv8-seg算法通过一系列创新的设计和技术，显著提升了目标检测的性能，尤其是在复杂环境下的小目标检测能力。尽管仍存在一些挑战，但通过不断的改进和优化，YOLOv8-seg及其衍生版本正在向更高的目标检测精度和速度迈进，展现出广阔的应用前景。随着技术的不断发展，YOLOv8-seg有望在智能监控、自动驾驶等多个领域发挥重要作用，为相关应用提供更加可靠和高效的解决方案。

## 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）

```以下是对代码的核心部分进行提炼和详细注释的版本：  
  
```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。")  
  
# 程序入口  
if \_\_name\_\_ == "\_\_main\_\_":  
 # 指定要运行的脚本路径  
 script\_path = "web.py" # 这里可以直接指定脚本名称  
  
 # 调用函数运行脚本  
 run\_script(script\_path)  
```  
  
### 代码分析与注释：  
  
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. \*\*`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。  
 - 使用 `sys.executable` 获取当前 Python 解释器的路径，以确保使用正确的 Python 环境来运行脚本。  
 - 构建命令字符串，使用 `streamlit` 模块运行指定的脚本。  
 - 使用 `subprocess.run` 执行命令，并等待其完成。  
 - 检查命令的返回码，如果返回码不为0，表示脚本运行出错，打印错误信息。  
  
3. \*\*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（这里为 `web.py`）。  
 - 调用 `run\_script` 函数，传入脚本路径以执行该脚本。  
  
这个代码的核心功能是通过当前 Python 环境来运行一个指定的脚本，并在执行过程中处理可能出现的错误。```

这个文件名为 `ui.py`，其主要功能是通过当前的 Python 环境来运行一个指定的脚本。文件中首先导入了必要的模块，包括 `sys`、`os` 和 `subprocess`，这些模块分别用于获取系统信息、处理文件路径和执行系统命令。  
  
在 `run\_script` 函数中，首先定义了一个参数 `script\_path`，它是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，存储在 `python\_path` 变量中。接着，构建了一个命令字符串 `command`，这个命令使用 `streamlit` 模块来运行指定的脚本。`streamlit` 是一个用于构建数据应用的库。  
  
然后，使用 `subprocess.run` 方法执行这个命令。`shell=True` 参数表示命令将在一个新的 shell 中执行。执行后，检查返回的结果码，如果不为零，表示脚本运行过程中出现了错误，程序会打印出相应的错误信息。  
  
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后面的代码。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运行一个名为 `web.py` 的脚本，并且在运行过程中能够处理可能出现的错误。

```以下是经过简化和注释的核心代码部分：  
  
```python  
# 导入必要的库  
from ultralytics.utils import SETTINGS, TESTS\_RUNNING  
from ultralytics.utils.torch\_utils import model\_info\_for\_loggers  
  
try:  
 # 确保不是在测试环境中运行  
 assert not TESTS\_RUNNING   
 # 确保WandB集成已启用  
 assert SETTINGS['wandb'] is True   
 import wandb as wb # 导入WandB库  
  
 # 确保WandB库已正确安装  
 assert hasattr(wb, '\_\_version\_\_')   
  
 import numpy as np # 导入NumPy库  
 import pandas as pd # 导入Pandas库  
  
 \_processed\_plots = {} # 用于存储已处理的图表  
  
except (ImportError, AssertionError):  
 wb = None # 如果导入失败或断言失败，则将wb设置为None  
  
  
def \_custom\_table(x, y, classes, title='Precision Recall Curve', x\_title='Recall', y\_title='Precision'):  
 """  
 创建并记录自定义的精确度-召回率曲线可视化。  
  
 参数:  
 x (List): x轴的值，长度为N。  
 y (List): y轴的值，长度为N。  
 classes (List): 每个点的类别标签，长度为N。  
 title (str, optional): 图表标题，默认为'Precision Recall Curve'。  
 x\_title (str, optional): x轴标签，默认为'Recall'。  
 y\_title (str, optional): y轴标签，默认为'Precision'。  
  
 返回:  
 (wandb.Object): 适合记录的WandB对象，展示自定义的可视化。  
 """  
 # 创建数据框  
 df = pd.DataFrame({'class': classes, 'y': y, 'x': x}).round(3)  
 fields = {'x': 'x', 'y': 'y', 'class': 'class'}  
 string\_fields = {'title': title, 'x-axis-title': x\_title, 'y-axis-title': y\_title}  
 # 返回WandB表格对象  
 return wb.plot\_table('wandb/area-under-curve/v0',  
 wb.Table(dataframe=df),  
 fields=fields,  
 string\_fields=string\_fields)  
  
  
def \_plot\_curve(x, y, names=None, id='precision-recall', title='Precision Recall Curve', x\_title='Recall', y\_title='Precision', num\_x=100, only\_mean=False):  
 """  
 记录度量曲线可视化。  
  
 参数:  
 x (np.ndarray): x轴数据点，长度为N。  
 y (np.ndarray): y轴数据点，形状为CxN，C为类别数。  
 names (list, optional): y轴数据对应的类别名称，长度为C。  
 id (str, optional): 记录数据的唯一标识符，默认为'precision-recall'。  
 title (str, optional): 可视化图表的标题，默认为'Precision Recall Curve'。  
 x\_title (str, optional): x轴标签，默认为'Recall'。  
 y\_title (str, optional): y轴标签，默认为'Precision'。  
 num\_x (int, optional): 可视化的插值数据点数量，默认为100。  
 only\_mean (bool, optional): 是否仅绘制均值曲线的标志，默认为True。  
 """  
 # 创建新的x值  
 if names is None:  
 names = []  
 x\_new = np.linspace(x[0], x[-1], num\_x).round(5)  
  
 # 创建记录用的数组  
 x\_log = x\_new.tolist()  
 y\_log = np.interp(x\_new, x, np.mean(y, axis=0)).round(3).tolist()  
  
 if only\_mean:  
 # 仅记录均值曲线  
 table = wb.Table(data=list(zip(x\_log, y\_log)), columns=[x\_title, y\_title])  
 wb.run.log({title: wb.plot.line(table, x\_title, y\_title, title=title)})  
 else:  
 # 记录每个类别的曲线  
 classes = ['mean'] \* len(x\_log)  
 for i, yi in enumerate(y):  
 x\_log.extend(x\_new) # 添加新的x值  
 y\_log.extend(np.interp(x\_new, x, yi)) # 插值y到新的x  
 classes.extend([names[i]] \* len(x\_new)) # 添加类别名称  
 wb.log({id: \_custom\_table(x\_log, y\_log, classes, title, x\_title, y\_title)}, commit=False)  
  
  
def on\_fit\_epoch\_end(trainer):  
 """在每个训练周期结束时记录训练指标和模型信息。"""  
 wb.run.log(trainer.metrics, step=trainer.epoch + 1) # 记录指标  
 # 记录图表  
 \_log\_plots(trainer.plots, step=trainer.epoch + 1)  
 \_log\_plots(trainer.validator.plots, step=trainer.epoch + 1)  
 if trainer.epoch == 0:  
 wb.run.log(model\_info\_for\_loggers(trainer), step=trainer.epoch + 1) # 记录模型信息  
  
  
def on\_train\_end(trainer):  
 """在训练结束时保存最佳模型作为工件。"""  
 \_log\_plots(trainer.validator.plots, step=trainer.epoch + 1)  
 \_log\_plots(trainer.plots, step=trainer.epoch + 1)  
 art = wb.Artifact(type='model', name=f'run\_{wb.run.id}\_model') # 创建模型工件  
 if trainer.best.exists():  
 art.add\_file(trainer.best) # 添加最佳模型文件  
 wb.run.log\_artifact(art, aliases=['best']) # 记录工件  
 # 记录验证指标曲线  
 for curve\_name, curve\_values in zip(trainer.validator.metrics.curves, trainer.validator.metrics.curves\_results):  
 x, y, x\_title, y\_title = curve\_values  
 \_plot\_curve(  
 x,  
 y,  
 names=list(trainer.validator.metrics.names.values()),  
 id=f'curves/{curve\_name}',  
 title=curve\_name,  
 x\_title=x\_title,  
 y\_title=y\_title,  
 )  
 wb.run.finish() # 结束WandB运行  
  
  
# 定义回调函数  
callbacks = {  
 'on\_fit\_epoch\_end': on\_fit\_epoch\_end,  
 'on\_train\_end': on\_train\_end  
} if wb else {}  
```  
  
### 代码说明：  
1. \*\*导入库\*\*：导入必要的库，包括Ultralytics的工具和WandB（Weights and Biases）用于可视化和记录实验。  
2. \*\*异常处理\*\*：在导入WandB时进行异常处理，确保在测试环境中不记录日志，并检查WandB的设置。  
3. \*\*自定义表格函数\*\*：`\_custom\_table`函数用于创建和记录精确度-召回率曲线的可视化，使用Pandas创建数据框并返回WandB表格对象。  
4. \*\*绘制曲线函数\*\*：`\_plot\_curve`函数根据输入数据生成并记录度量曲线的可视化，可以选择记录均值曲线或每个类别的曲线。  
5. \*\*训练结束时的回调\*\*：`on\_fit\_epoch\_end`和`on\_train\_end`函数在训练结束时记录指标、图表和最佳模型。  
  
这些核心部分和注释提供了对代码功能的清晰理解，便于后续的使用和修改。```

这个程序文件是用于集成和管理训练过程中的可视化日志，特别是与WandB（Weights and Biases）平台的交互。文件首先导入了一些必要的模块和库，包括Ultralytics的设置和工具，以及WandB库。如果在导入过程中发生错误（例如WandB未安装或未启用），则会将WandB设置为None。  
  
程序中定义了几个函数。首先是`\_custom\_table`函数，它用于创建和记录一个自定义的精确度-召回曲线可视化。该函数接受x轴和y轴的数据、类别标签以及图表的标题和轴标签，并生成一个WandB对象以便于记录。  
  
接下来是`\_plot\_curve`函数，它生成一个度量曲线的可视化，并将其记录到WandB。这个函数可以选择绘制平均曲线或每个类别的曲线，具体取决于`only\_mean`参数。它通过插值方法生成新的x值，并根据输入数据计算相应的y值，最后将结果记录到WandB。  
  
`\_log\_plots`函数用于记录输入字典中的图表，如果在指定的步骤中尚未记录过。它检查每个图表的时间戳，以避免重复记录。  
  
接下来的几个函数是训练过程中的回调函数。`on\_pretrain\_routine\_start`函数在预训练例程开始时初始化WandB项目。`on\_fit\_epoch\_end`函数在每个训练周期结束时记录训练指标和模型信息。`on\_train\_epoch\_end`函数在每个训练周期结束时记录损失和学习率，并在特定条件下记录图表。`on\_train\_end`函数在训练结束时保存最佳模型，并记录验证指标的曲线。  
  
最后，程序将这些回调函数整理成一个字典，只有在WandB可用的情况下才会定义这个字典。整体来看，这个文件的目的是通过WandB平台对模型训练过程中的性能进行可视化和记录，以便于后续分析和调试。

```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 time  
import cv2  
import pandas as pd  
from PIL import Image  
  
def save\_chinese\_image(file\_path, image\_array):  
 """  
 保存带有中文路径的图片文件  
  
 参数：  
 file\_path (str): 图片的保存路径，应包含中文字符  
 image\_array (numpy.ndarray): 要保存的 OpenCV 图像（即 numpy 数组）  
 """  
 try:  
 # 将 OpenCV 图片转换为 Pillow Image 对象  
 image = Image.fromarray(cv2.cvtColor(image\_array, cv2.COLOR\_BGR2RGB))  
  
 # 使用 Pillow 保存图片文件  
 image.save(file\_path)  
  
 print(f"成功保存图像到: {file\_path}")  
 except Exception as e:  
 print(f"保存图像失败: {str(e)}")  
  
class LogTable:  
 def \_\_init\_\_(self, csv\_file\_path=None):  
 """  
 初始化类实例。  
  
 参数：  
 csv\_file\_path (str): 保存初始数据的CSV文件路径。  
 """  
 self.csv\_file\_path = csv\_file\_path  
 self.data = pd.DataFrame(columns=['文件路径', '识别结果', '位置', '面积', '时间'])  
  
 # 尝试从CSV文件加载数据，如果失败则创建一个空的DataFrame  
 if csv\_file\_path and os.path.exists(csv\_file\_path):  
 self.data = pd.read\_csv(csv\_file\_path, encoding='utf-8')  
  
 def add\_log\_entry(self, file\_path, recognition\_result, position, confidence, time\_spent):  
 """  
 向日志中添加一条新记录。  
  
 参数：  
 file\_path (str): 文件路径  
 recognition\_result (str): 识别结果  
 position (str): 位置  
 confidence (float): 置信度  
 time\_spent (float): 用时（通常是秒或毫秒）  
  
 返回：  
 None  
 """  
 # 创建新的数据行  
 new\_entry = pd.DataFrame([[file\_path, recognition\_result, position, confidence, time\_spent]],  
 columns=['文件路径', '识别结果', '位置', '面积', '时间'])  
  
 # 将新行添加到DataFrame中  
 self.data = pd.concat([new\_entry, self.data]).reset\_index(drop=True)  
  
 def save\_to\_csv(self):  
 """  
 将更新后的DataFrame保存到CSV文件  
 """  
 self.data.to\_csv(self.csv\_file\_path, index=False, encoding='utf-8', mode='a', header=False)  
  
 def update\_table(self, log\_table\_placeholder):  
 """  
 更新表格，显示最新的500条记录。  
  
 参数：  
 log\_table\_placeholder: Streamlit的表格占位符  
  
 返回：  
 None  
 """  
 # 判断DataFrame的长度是否超过500  
 if len(self.data) > 500:  
 # 如果超过500，仅显示最新的500条记录  
 display\_data = self.data.head(500)  
 else:  
 # 如果不超过，显示全部数据  
 display\_data = self.data  
  
 log\_table\_placeholder.table(display\_data)  
```  
  
### 代码核心部分说明：  
  
1. \*\*`save\_chinese\_image` 函数\*\*：  
 - 该函数用于保存带有中文路径的图片。它将OpenCV格式的图像转换为Pillow格式，并使用Pillow库保存图像文件。  
  
2. \*\*`LogTable` 类\*\*：  
 - 该类用于管理日志记录，包括初始化、添加日志条目、保存到CSV文件和更新显示表格等功能。  
 - `\_\_init\_\_` 方法用于初始化类实例，并尝试从指定的CSV文件加载数据。  
 - `add\_log\_entry` 方法用于向日志中添加一条新记录。  
 - `save\_to\_csv` 方法将当前的DataFrame保存到CSV文件中。  
 - `update\_table` 方法用于更新表格显示，限制显示的记录数为500条。   
  
通过这些核心部分，代码实现了图像保存和日志记录的基本功能。```

这个程序文件 `log.py` 是一个用于处理图像和记录识别结果的工具，主要功能包括保存图像、记录识别结果、生成视频和保存数据到CSV文件等。  
  
首先，程序导入了一些必要的库，包括 `os`、`time`、`cv2`（OpenCV库）、`pandas`（用于数据处理）、`PIL`（用于图像处理）、`numpy`（用于数组操作）和 `datetime`（用于时间处理）。这些库为后续的图像处理和数据管理提供了支持。  
  
接下来，定义了一个名为 `save\_chinese\_image` 的函数，该函数用于保存带有中文路径的图像文件。函数接收两个参数：文件路径和图像数组。它首先将OpenCV格式的图像转换为Pillow格式，然后尝试保存图像，并在控制台输出保存结果。  
  
然后，定义了一个 `ResultLogger` 类，该类用于记录识别结果。初始化时，它创建一个空的DataFrame，用于存储识别结果、位置、面积和时间等信息。`concat\_results` 方法用于将新的检测结果添加到DataFrame中。  
  
接着，定义了 `LogTable` 类，该类用于管理图像和识别结果的记录。初始化时，它尝试从指定的CSV文件加载数据，如果文件不存在，则创建一个空的DataFrame。该类提供了多个方法，包括添加图像帧、清空帧、保存图像或视频、添加日志条目、清空数据、保存到CSV文件以及更新显示表格等。  
  
在 `add\_frames` 方法中，程序将图像和检测信息存储到相应的列表中，并在有检测信息时将图像添加到目标图像列表中。`clear\_frames` 方法则用于清空这些列表。  
  
`save\_frames\_file` 方法根据保存的图像数量决定是保存为单张图片还是视频。如果只有一张图像，则保存为PNG格式；如果有多张图像，则保存为AVI格式的视频文件。  
  
`add\_log\_entry` 方法用于向日志中添加新记录，包括文件路径、识别结果、位置、置信度和用时等信息。`clear\_data` 方法用于清空DataFrame中的数据，而 `save\_to\_csv` 方法则将更新后的DataFrame保存到CSV文件中。  
  
最后，`update\_table` 方法用于更新表格显示，限制最多显示最新的500条记录，确保用户能够快速查看最新的识别结果。  
  
整体来看，这个程序文件为图像处理和结果记录提供了一个完整的框架，适合用于需要记录和管理图像识别结果的应用场景。

```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  
from ultralytics.utils import ops  
  
class NASValidator:  
 """  
 Ultralytics YOLO NAS 验证器，用于目标检测。  
  
 该类用于对 YOLO NAS 模型生成的原始预测结果进行后处理。它执行非极大值抑制（NMS），以去除重叠和低置信度的框，最终生成最终检测结果。  
 """  
  
 def postprocess(self, preds\_in):  
 """对预测输出应用非极大值抑制（NMS）。"""  
 # 将预测结果中的边界框从 xyxy 格式转换为 xywh 格式  
 boxes = ops.xyxy2xywh(preds\_in[0][0])  
   
 # 将边界框和置信度合并，并调整维度顺序  
 preds = torch.cat((boxes, preds\_in[0][1]), -1).permute(0, 2, 1)  
   
 # 应用非极大值抑制，去除重叠的框  
 return ops.non\_max\_suppression(preds,  
 self.args.conf, # 置信度阈值  
 self.args.iou, # IoU 阈值  
 labels=self.lb, # 可选的多标签 NMS  
 multi\_label=False, # 是否使用多标签  
 agnostic=self.args.single\_cls, # 是否类别无关  
 max\_det=self.args.max\_det, # 最大检测框数量  
 max\_time\_img=0.5) # 每张图像的最大处理时间  
```  
  
### 代码核心部分解释：  
1. \*\*导入必要的库\*\*：  
 - `torch`：用于张量操作和深度学习模型的计算。  
 - `ops`：包含一些操作函数，主要用于处理边界框和执行非极大值抑制。  
  
2. \*\*NASValidator 类\*\*：  
 - 该类用于处理 YOLO NAS 模型的预测结果，主要功能是进行后处理以获得最终的检测结果。  
  
3. \*\*postprocess 方法\*\*：  
 - 该方法接收原始预测结果 `preds\_in`，并执行以下步骤：  
 - \*\*边界框转换\*\*：将边界框的坐标格式从 `xyxy`（左上角和右下角坐标）转换为 `xywh`（中心点坐标和宽高）。  
 - \*\*合并预测结果\*\*：将转换后的边界框和置信度合并，并调整张量的维度顺序，以便后续处理。  
 - \*\*非极大值抑制\*\*：调用 `non\_max\_suppression` 函数，去除重叠的边界框，返回最终的检测结果。该函数使用置信度和 IoU 阈值来决定保留哪些框。```

这个程序文件 `val.py` 是 Ultralytics YOLO（You Only Look Once）模型的一部分，主要用于对象检测中的验证过程。文件中定义了一个名为 `NASValidator` 的类，它继承自 `DetectionValidator`，专门用于处理 YOLO NAS 模型生成的原始预测结果。  
  
在这个类中，主要的功能是对模型的输出进行后处理，具体包括应用非极大值抑制（Non-Maximum Suppression, NMS）来去除重叠和低置信度的边界框，从而生成最终的检测结果。类的属性包括 `args`，这是一个命名空间，包含了后处理所需的各种配置，比如置信度和交并比（IoU）阈值；还有 `lb`，这是一个可选的张量，用于多标签 NMS。  
  
在使用示例中，首先从 `ultralytics` 导入 `NAS` 类，然后实例化一个 YOLO NAS 模型，并获取其验证器。假设已经获得了原始预测结果 `raw\_preds`，可以通过调用 `validator.postprocess(raw\_preds)` 来获取最终的预测结果。  
  
值得注意的是，这个类通常不会被直接实例化，而是在 `NAS` 类内部使用。类中的 `postprocess` 方法是其核心功能，负责将输入的预测结果进行处理。具体来说，它首先将预测框的坐标从 xyxy 格式转换为 xywh 格式，然后将边界框和置信度合并，并通过非极大值抑制来过滤结果，最终返回处理后的检测结果。  
  
总的来说，这个文件的主要目的是提供一个有效的方式来处理 YOLO NAS 模型的输出，以便在对象检测任务中获得更准确的结果。

```以下是简化后的核心代码，并附上详细的中文注释：  
  
```python  
import torch  
import torch.nn as nn  
import torch.nn.functional as F  
from timm.layers import weight\_init  
  
# 定义激活函数类  
class Activation(nn.ReLU):  
 def \_\_init\_\_(self, dim, act\_num=3, deploy=False):  
 super(Activation, self).\_\_init\_\_()  
 self.deploy = deploy  
 # 初始化权重和偏置  
 self.weight = torch.nn.Parameter(torch.randn(dim, 1, act\_num \* 2 + 1, act\_num \* 2 + 1))  
 self.bias = None  
 self.bn = nn.BatchNorm2d(dim, eps=1e-6) # 批归一化  
 self.dim = dim  
 self.act\_num = act\_num  
 weight\_init.trunc\_normal\_(self.weight, std=.02) # 权重初始化  
  
 def forward(self, x):  
 # 前向传播  
 if self.deploy:  
 return F.conv2d(  
 super(Activation, self).forward(x),   
 self.weight, self.bias, padding=(self.act\_num \* 2 + 1) // 2, groups=self.dim)  
 else:  
 return self.bn(F.conv2d(  
 super(Activation, self).forward(x),  
 self.weight, padding=self.act\_num, groups=self.dim))  
  
 def switch\_to\_deploy(self):  
 # 切换到部署模式，融合批归一化  
 if not self.deploy:  
 kernel, bias = self.\_fuse\_bn\_tensor(self.weight, self.bn)  
 self.weight.data = kernel  
 self.bias = torch.nn.Parameter(torch.zeros(self.dim))  
 self.bias.data = bias  
 self.\_\_delattr\_\_('bn') # 删除bn属性  
 self.deploy = True  
  
 def \_fuse\_bn\_tensor(self, weight, bn):  
 # 融合批归一化的权重和偏置  
 kernel = weight  
 running\_mean = bn.running\_mean  
 running\_var = bn.running\_var  
 gamma = bn.weight  
 beta = bn.bias  
 eps = bn.eps  
 std = (running\_var + eps).sqrt()  
 t = (gamma / std).reshape(-1, 1, 1, 1)  
 return kernel \* t, beta + (0 - running\_mean) \* gamma / std  
  
# 定义基本模块  
class Block(nn.Module):  
 def \_\_init\_\_(self, dim, dim\_out, act\_num=3, stride=2, deploy=False):  
 super().\_\_init\_\_()  
 self.deploy = deploy  
 if self.deploy:  
 self.conv = nn.Conv2d(dim, dim\_out, kernel\_size=1)  
 else:  
 self.conv1 = nn.Sequential(  
 nn.Conv2d(dim, dim, kernel\_size=1),  
 nn.BatchNorm2d(dim, eps=1e-6),  
 )  
 self.conv2 = nn.Sequential(  
 nn.Conv2d(dim, dim\_out, kernel\_size=1),  
 nn.BatchNorm2d(dim\_out, eps=1e-6)  
 )  
 self.pool = nn.MaxPool2d(stride) if stride != 1 else nn.Identity() # 池化层  
 self.act = Activation(dim\_out, act\_num) # 激活函数  
  
 def forward(self, x):  
 # 前向传播  
 if self.deploy:  
 x = self.conv(x)  
 else:  
 x = self.conv1(x)  
 x = F.leaky\_relu(x, negative\_slope=1) # 使用Leaky ReLU  
 x = self.conv2(x)  
  
 x = self.pool(x) # 池化  
 x = self.act(x) # 激活  
 return x  
  
# 定义主网络结构  
class VanillaNet(nn.Module):  
 def \_\_init\_\_(self, in\_chans=3, num\_classes=1000, dims=[96, 192, 384, 768], strides=[2, 2, 2, 1], deploy=False):  
 super().\_\_init\_\_()  
 self.deploy = deploy  
 if self.deploy:  
 self.stem = nn.Sequential(  
 nn.Conv2d(in\_chans, dims[0], kernel\_size=4, stride=4),  
 Activation(dims[0])  
 )  
 else:  
 self.stem1 = nn.Sequential(  
 nn.Conv2d(in\_chans, dims[0], kernel\_size=4, stride=4),  
 nn.BatchNorm2d(dims[0], eps=1e-6),  
 )  
 self.stem2 = nn.Sequential(  
 nn.Conv2d(dims[0], dims[0], kernel\_size=1, stride=1),  
 nn.BatchNorm2d(dims[0], eps=1e-6),  
 Activation(dims[0])  
 )  
  
 self.stages = nn.ModuleList()  
 for i in range(len(strides)):  
 stage = Block(dim=dims[i], dim\_out=dims[i + 1], stride=strides[i], deploy=deploy)  
 self.stages.append(stage)  
  
 def forward(self, x):  
 # 前向传播  
 if self.deploy:  
 x = self.stem(x)  
 else:  
 x = self.stem1(x)  
 x = F.leaky\_relu(x, negative\_slope=1)  
 x = self.stem2(x)  
  
 for stage in self.stages:  
 x = stage(x) # 通过每个Block  
 return x  
  
 def switch\_to\_deploy(self):  
 # 切换到部署模式  
 if not self.deploy:  
 self.stem2[2].switch\_to\_deploy()  
 self.deploy = True  
  
# 示例：创建模型并进行前向传播  
if \_\_name\_\_ == '\_\_main\_\_':  
 inputs = torch.randn((1, 3, 640, 640)) # 输入张量  
 model = VanillaNet() # 创建模型  
 pred = model(inputs) # 前向传播  
 print(pred.size()) # 输出预测结果的尺寸  
```  
  
### 代码注释说明：  
1. \*\*Activation 类\*\*：定义了一个自定义的激活函数类，继承自 `nn.ReLU`，并在前向传播中实现了可选的批归一化。  
2. \*\*Block 类\*\*：定义了网络的基本模块，包含卷积层、池化层和激活函数。根据是否处于部署模式选择不同的结构。  
3. \*\*VanillaNet 类\*\*：构建了整个网络结构，包含多个 `Block`，并实现了前向传播和切换到部署模式的功能。  
4. \*\*示例代码\*\*：在主程序中创建一个输入张量，实例化模型并进行前向传播，最后输出预测结果的尺寸。```

这个程序文件实现了一个名为VanillaNet的神经网络模型，主要用于图像处理任务。文件开头包含版权信息和许可证声明，表示该程序是开源的，并遵循MIT许可证。  
  
程序中首先导入了必要的库，包括PyTorch及其神经网络模块、功能模块，以及一些辅助函数和类。接着，定义了一个名为`activation`的类，继承自`nn.ReLU`，用于实现自定义的激活函数。该类在初始化时会创建一个权重参数和一个批归一化层，并在前向传播中根据是否处于部署模式（deploy）选择不同的计算方式。  
  
接下来定义了一个`Block`类，表示网络中的一个基本模块。该模块包含多个卷积层和激活函数，并可以选择是否使用自适应池化。它的前向传播方法根据当前模式处理输入，并应用池化和激活函数。  
  
`VanillaNet`类是整个网络的核心，包含多个`Block`模块。初始化时，它根据输入通道数、类别数、特征维度、丢弃率等参数构建网络结构。在前向传播中，输入数据会经过多个阶段的处理，并根据输入尺寸记录特征图。  
  
此外，程序还定义了一些辅助函数，如`update\_weight`用于更新模型权重，`vanillanet\_x`系列函数用于创建不同配置的VanillaNet模型，并支持加载预训练权重。  
  
最后，程序在主模块中创建了一个输入张量，并实例化了一个VanillaNet模型进行前向推理，输出每个特征图的尺寸。这部分代码可以用于测试模型的基本功能。  
  
总体而言，这个文件实现了一个灵活且可扩展的神经网络结构，适用于多种图像处理任务，并提供了模型训练和推理的基本框架。

```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 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  
 batch = 8 # 每个批次的大小，适当调整以适应显存/内存  
 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')   
  
 # 将路径格式转换为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:  
 data['train'] = directory\_path + '/train' # 设置训练数据路径  
 data['val'] = directory\_path + '/val' # 设置验证数据路径  
 data['test'] = directory\_path + '/test' # 设置测试数据路径  
  
 # 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模型配置文件和预训练权重  
 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, # 指定训练设备（GPU或CPU）  
 workers=workers, # 指定使用的工作进程数  
 imgsz=640, # 指定输入图像的大小为640x640  
 epochs=100, # 指定训练的轮数为100  
 batch=batch, # 指定每个批次的大小  
 )  
```  
  
### 代码核心部分说明：  
1. \*\*导入必要的库\*\*：导入了`os`、`torch`、`yaml`和`YOLO`模型，后者是用于目标检测的深度学习模型。  
2. \*\*参数设置\*\*：设置了数据加载的工作进程数、批次大小和设备（GPU或CPU）。  
3. \*\*数据路径处理\*\*：获取数据集的配置文件路径，并将路径格式转换为Unix风格，便于后续处理。  
4. \*\*读取和修改YAML文件\*\*：读取YAML文件，修改训练、验证和测试数据的路径，并将修改后的内容写回文件。  
5. \*\*模型加载\*\*：加载YOLO模型的配置文件和预训练权重，以便进行训练。  
6. \*\*模型训练\*\*：调用`train`方法开始训练模型，指定训练数据、设备、工作进程数、图像大小、训练轮数和批次大小等参数。```

这个程序文件`train.py`的主要功能是使用YOLO（You Only Look Once）模型进行图像分割任务的训练。程序首先导入了必要的库，包括`os`、`torch`、`yaml`和`ultralytics`中的YOLO模型。接着，它使用`matplotlib`库来设置图形界面。  
  
在`\_\_main\_\_`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`、以及设备`device`（如果有可用的GPU则使用GPU，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的绝对路径，这个配置文件是一个YAML格式的文件，包含了训练、验证和测试数据的路径。  
  
程序将路径中的分隔符转换为Unix风格，并提取出目录路径。然后，它打开YAML文件并读取内容。若YAML文件中包含`train`、`val`和`test`的键，程序会将这些键的值修改为对应的绝对路径，并将修改后的内容写回到YAML文件中。  
  
接下来，程序加载YOLO模型的配置文件，并使用预训练的权重文件进行初始化。需要注意的是，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。  
  
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、输入图像的大小（640x640）、训练的轮数（100个epoch）以及每个批次的大小（8）。通过这些设置，程序将开始进行模型的训练过程。

### 整体功能和构架概括  
  
该项目是一个基于Ultralytics YOLO框架的计算机视觉应用，主要用于对象检测和图像分割任务。项目的结构包括多个模块和文件，每个文件负责特定的功能，形成一个完整的训练、验证和推理流程。整体上，项目的功能可以分为以下几个部分：  
  
1. \*\*用户界面\*\*：通过`ui.py`提供简单的界面来运行模型。  
2. \*\*训练和验证\*\*：`train.py`负责模型的训练过程，而`val.py`则用于验证模型的性能。  
3. \*\*日志记录\*\*：`log.py`用于记录训练过程中的数据和结果，便于后续分析。  
4. \*\*模型定义\*\*：`VanillaNet.py`和其他模型文件定义了网络架构和模块。  
5. \*\*回调和可视化\*\*：`wb.py`用于与WandB集成，记录训练过程中的可视化数据。  
6. \*\*辅助功能\*\*：其他文件提供了额外的功能，如数据处理、模型的构建和设置等。  
  
### 文件功能整理表  
  
| 文件路径 | 功能描述 |  
|------------------------------------------------------|-----------------------------------------------------------------|  
| `ui.py` | 提供用户界面，方便运行指定的脚本（如`web.py`）。 |  
| `ultralytics/utils/callbacks/wb.py` | 集成WandB进行训练过程的可视化和日志记录。 |  
| `log.py` | 处理图像和记录识别结果，支持保存图像、生成视频和记录到CSV文件。 |  
| `ultralytics/models/nas/val.py` | 定义验证器，用于处理YOLO NAS模型的输出，执行后处理和非极大值抑制。 |  
| `ultralytics/nn/backbone/VanillaNet.py` | 实现VanillaNet模型的结构，定义网络模块和前向传播逻辑。 |  
| `train.py` | 负责YOLO模型的训练过程，加载数据集配置并启动训练。 |  
| `ultralytics/nn/extra\_modules/ops\_dcnv3/setup.py` | 设置DCNv3模块的安装和配置。 |  
| `demo\_test\_image.py` | 进行图像测试，展示模型在给定图像上的推理结果。 |  
| `ultralytics/nn/extra\_modules/afpn.py` | 实现自适应特征金字塔网络（AFPN），用于改进特征提取。 |  
| `ultralytics/hub/auth.py` | 处理模型的认证和访问控制。 |  
| `ultralytics/nn/extra\_modules/ops\_dcnv3/modules/dcnv3.py` | 实现DCNv3模块的核心功能，支持深度可分离卷积。 |  
| `ultralytics/engine/model.py` | 定义模型的基本结构和训练、推理的核心逻辑。 |  
| `ultralytics/data/build.py` | 处理数据集的构建和预处理，准备训练和验证所需的数据。 |  
  
以上表格总结了每个文件的主要功能，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。