# 工厂危险工作区域监测设备图像分割系统源码＆数据集分享 [yolov8-seg-C2f-DAttention＆yolov8-seg-repvit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  
  
随着工业化进程的加快，工厂的生产效率和安全性成为了社会各界关注的焦点。尤其是在危险工作区域，工人面临着多种潜在的安全隐患，如机械设备的操作风险、化学品的泄漏以及高处作业的坠落等。因此，如何有效监测和管理这些危险区域，保障工人的安全，已成为亟待解决的重要课题。近年来，计算机视觉技术的快速发展为这一问题提供了新的解决方案，尤其是基于深度学习的图像分割技术，能够实现对工厂环境中各类物体的精准识别与定位。  
  
在众多图像分割算法中，YOLO（You Only Look Once）系列模型因其高效的实时性和准确性而受到广泛关注。YOLOv8作为该系列的最新版本，进一步提升了目标检测和分割的性能，能够在复杂环境中快速、准确地识别出目标物体。然而，针对工厂危险工作区域的特定需求，YOLOv8的标准模型可能无法完全满足实际应用的要求。因此，基于YOLOv8的改进，开发一套专门针对工厂危险工作区域的图像分割系统，具有重要的理论价值和实际意义。  
  
本研究所采用的数据集包含3400张图像，涵盖了8个类别，包括安全帽、机器、手套、无手套、无背心、人员、安全鞋和背心。这些类别的选择不仅反映了工厂工作环境中的常见物品，也突出了安全防护装备的重要性。通过对这些物体的精确分割与识别，能够有效监测工人在危险区域的安全状态，及时发现未佩戴安全装备的人员，从而降低事故发生的风险。此外，图像分割系统还可以为工厂管理者提供实时数据，帮助其优化安全管理措施，提升整体安全水平。  
  
本研究的意义不仅在于技术上的创新，更在于其对工厂安全管理实践的推动。通过构建基于改进YOLOv8的图像分割系统，能够实现对工厂危险工作区域的智能监测，为工人提供更安全的工作环境。同时，该系统的成功应用也为其他行业的安全监测提供了借鉴，具有广泛的推广价值。  
  
综上所述，基于改进YOLOv8的工厂危险工作区域监测设备图像分割系统的研究，不仅是对现有技术的提升，更是对工厂安全管理实践的有力支持。通过有效识别和监测危险工作区域的各类物体，能够显著降低工伤事故的发生率，保障工人的生命安全，推动工业生产的可持续发展。因此，本研究具有重要的理论意义和实践价值，为未来的安全管理提供了新的思路和方法。

## 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“Instance\_segmention\_msn3”的数据集，以支持改进YOLOv8-seg的工厂危险工作区域监测设备图像分割系统的训练和评估。该数据集专门设计用于识别和分割工厂环境中的各种安全装备和人员，以提高工作场所的安全性和效率。数据集包含8个类别，分别是：头盔（Helmet）、机器（Machines）、手套（gloves）、无手套（no-gloves）、无背心（no-vest）、人员（person）、安全鞋（safety shoes）和背心（vest）。这些类别涵盖了工厂工作环境中常见的安全防护装备及其相关对象，旨在帮助监测系统准确识别潜在的安全隐患。  
  
在数据集的构建过程中，特别关注了工厂环境的多样性和复杂性。每个类别的图像都经过精心挑选，以确保在不同的光照条件、角度和背景下都能有效进行分割和识别。例如，头盔和背心的图像可能在不同的颜色和样式上有所变化，而机器的图像则可能包括多种类型的设备。这种多样性不仅增强了模型的鲁棒性，也使得系统在实际应用中能够更好地适应各种工厂环境。  
  
数据集中的每个图像都附带了详细的标注信息，标注不仅包括对象的类别，还涵盖了对象的精确位置和轮廓。这种高质量的标注为YOLOv8-seg模型的训练提供了丰富的监督信号，使得模型能够学习到不同类别对象的特征和分布，从而在实际应用中实现高效的图像分割。通过对图像中每个对象的精确分割，系统能够实时监测工厂内的安全装备佩戴情况，及时发现未佩戴安全装备的人员，进而降低事故发生的风险。  
  
此外，数据集的构建还考虑到了数据的平衡性。每个类别的样本数量经过精心设计，以避免在训练过程中出现类别不平衡的问题。通过确保每个类别都有足够的样本，模型能够在学习过程中获得更全面的特征，从而提高其在实际应用中的识别准确率。  
  
在进行模型训练时，我们采用了数据增强技术，以进一步提升模型的泛化能力。通过对图像进行旋转、缩放、裁剪和颜色变换等操作，生成多样化的训练样本，增强模型对不同场景和条件的适应能力。这种策略不仅提高了模型的鲁棒性，还能有效防止过拟合现象的发生。  
  
综上所述，“Instance\_segmention\_msn3”数据集为改进YOLOv8-seg的工厂危险工作区域监测设备图像分割系统提供了坚实的基础。通过丰富的类别信息和高质量的标注，结合有效的数据增强策略，期望该系统能够在实际应用中发挥重要作用，为工厂安全管理提供有力支持。

## 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2023年由Ultralytics推出的一款先进的目标检测与分割模型，它在YOLO系列的基础上进行了多项创新与优化，结合了YOLOv5、YOLOv6、YOLOv7以及PP-YOLOE等多种算法的优点，标志着目标检测领域的一次重要进步。YOLOv8-seg不仅继承了YOLO系列一贯的高效性和实时性，还在精度和功能上实现了显著提升，尤其是在图像分割任务中的表现，展现了其在复杂场景下的强大能力。  
  
YOLOv8-seg的核心思想是将目标检测与实例分割相结合，通过单一的神经网络模型实现对目标的定位、分类以及精确的边界分割。这一创新使得YOLOv8-seg在处理多种目标时，能够同时输出目标的类别、位置和分割掩码，极大地提升了模型的实用性和灵活性。YOLOv8-seg的网络结构主要由主干网络、特征增强网络和检测头组成，采用了现代深度学习中的多种先进技术。  
  
在主干网络方面，YOLOv8-seg依然延续了CSP（Cross Stage Partial）网络的设计理念，旨在通过分阶段的特征提取来提高网络的学习能力和计算效率。CSP结构通过将特征图分为多个部分并进行交叉连接，有效地减少了模型的参数量，同时保持了较高的特征表达能力。这种设计不仅加快了模型的训练速度，还增强了模型对不同特征的提取能力，使得YOLOv8-seg在处理复杂场景时表现得更加出色。  
  
特征增强网络则采用了PAN-FPN（Path Aggregation Network - Feature Pyramid Network）的结合，旨在实现多尺度特征的高效融合。PAN-FPN通过引入路径聚合的思想，增强了不同层次特征之间的信息流动，确保了高层特征与低层特征的有效结合。这一设计使得YOLOv8-seg能够更好地捕捉到不同尺度目标的特征信息，从而在目标检测和分割任务中实现更高的精度。  
  
在检测头部分，YOLOv8-seg采用了解耦头的设计，将目标的分类和回归任务分为两个独立的分支。这一创新的解耦策略使得模型在处理复杂场景时，能够更加专注于各自的任务，显著降低了定位不准和分类错误的风险。同时，YOLOv8-seg引入了Anchor-free目标检测方法，直接通过回归的方式预测目标的位置和大小，省去了传统方法中繁琐的锚点框定义与调整过程。这种简化不仅提高了模型的训练效率，也使得YOLOv8-seg在不同尺度和形状的目标检测中表现得更加灵活和高效。  
  
YOLOv8-seg的损失函数设计也经过了精心优化，结合了多种损失策略以提高模型的训练效果。特别是在分割任务中，YOLOv8-seg采用了交叉熵损失与Dice损失的结合，确保了模型在边界处理和目标分割上的高精度。这种多损失策略使得模型在面对复杂背景和多目标场景时，能够更好地进行目标分割，减少了漏检和误检的情况。  
  
此外，YOLOv8-seg在轻量化设计方面也做出了显著努力。通过对网络结构的优化，YOLOv8-seg的权重文件相比于之前的YOLO系列进一步减小，使得模型能够在各种嵌入式设备上高效运行。这一特性使得YOLOv8-seg不仅适用于传统的计算平台，也能够在移动设备和边缘计算环境中实现实时目标检测与分割，极大地拓宽了其应用场景。  
  
在实际应用中，YOLOv8-seg的表现令人瞩目。以苹果采摘为例，利用YOLOv8-seg的视觉识别能力，自动检测和定位苹果，不仅提高了采摘效率，还降低了人工成本。同时，结合蚁群算法进行路径规划，进一步优化了采摘机器人的工作流程，使得整个过程更加智能化和自动化。这一应用展示了YOLOv8-seg在农业领域的潜力，预示着未来在更多行业中的广泛应用。  
  
综上所述，YOLOv8-seg算法以其高效、精确的特性，结合了现代深度学习的多项创新，成为目标检测与分割领域的一颗新星。它不仅提升了目标检测的实时性和准确性，更为未来的智能应用奠定了坚实的基础。随着技术的不断进步，YOLOv8-seg有望在更多复杂场景中展现出其卓越的性能，推动目标检测与分割技术的进一步发展。

## 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）

```以下是对给定代码的核心部分进行保留和详细注释的版本：  
  
```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，0 表示成功，非0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" # 这里可以直接使用相对路径  
  
 # 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。  
  
2. \*\*`run\_script` 函数\*\*：  
 - \*\*参数\*\*：接收一个字符串参数 `script\_path`，表示要运行的 Python 脚本的路径。  
 - \*\*功能\*\*：构建并执行一个命令来运行指定的脚本。  
  
3. \*\*获取 Python 解释器路径\*\*：  
 - `sys.executable` 返回当前 Python 解释器的完整路径。  
  
4. \*\*构建命令\*\*：  
 - 使用 f-string 构建一个命令字符串，调用 `streamlit` 来运行指定的脚本。  
  
5. \*\*执行命令\*\*：  
 - `subprocess.run` 方法用于执行构建的命令，`shell=True` 表示在 shell 中执行。  
  
6. \*\*检查返回码\*\*：  
 - `result.returncode` 用于检查命令的执行结果，0 表示成功，非0表示出错，若出错则打印错误信息。  
  
7. \*\*主程序块\*\*：  
 - 使用 `if \_\_name\_\_ == "\_\_main\_\_":` 确保该代码块仅在脚本作为主程序运行时执行。  
 - 指定要运行的脚本路径，并调用 `run\_script` 函数执行该脚本。  
  
通过以上注释，代码的核心逻辑和功能得以清晰呈现。```

这个程序文件名为 `ui.py`，其主要功能是运行一个指定的 Python 脚本，具体来说是通过 Streamlit 框架来启动一个 Web 应用。  
  
首先，文件导入了几个必要的模块，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` 模块则用于执行外部命令。  
  
接着，文件从 `QtFusion.path` 模块中导入了 `abs\_path` 函数，这个函数的作用是获取文件的绝对路径。  
  
在 `run\_script` 函数中，首先定义了一个参数 `script\_path`，它表示要运行的脚本的路径。函数内部通过 `sys.executable` 获取当前 Python 解释器的路径，然后构建一个命令字符串，该命令用于运行指定的脚本。命令的格式是使用当前 Python 环境下的 Streamlit 模块来运行脚本。  
  
接下来，使用 `subprocess.run` 方法执行构建好的命令。如果命令执行后返回的状态码不为 0，表示脚本运行过程中出现了错误，程序会打印出“脚本运行出错”的提示信息。  
  
在文件的最后部分，使用 `if \_\_name\_\_ == "\_\_main\_\_":` 语句来判断该文件是否是被直接运行的。如果是，则指定要运行的脚本路径，这里是通过 `abs\_path` 函数获取的 `web.py` 的绝对路径。最后，调用 `run\_script` 函数来执行这个脚本。  
  
总体来说，这个程序的主要功能是提供一个简单的接口来运行一个 Streamlit 应用，方便用户通过命令行启动 Web 应用。

```以下是提取后的核心代码部分，并附上详细的中文注释：  
  
```python  
import torch  
import torch.nn as nn  
import torch.nn.functional as F  
  
class LayerNorm(nn.Module):  
 """ 自定义的LayerNorm层，支持两种数据格式：channels\_last（默认）和channels\_first。  
 channels\_last对应的输入形状为(batch\_size, height, width, channels)，  
 而channels\_first对应的输入形状为(batch\_size, channels, height, width)。  
 """  
 def \_\_init\_\_(self, normalized\_shape, eps=1e-6, data\_format="channels\_last"):  
 super().\_\_init\_\_()  
 # 权重和偏置参数  
 self.weight = nn.Parameter(torch.ones(normalized\_shape))  
 self.bias = nn.Parameter(torch.zeros(normalized\_shape))  
 self.eps = eps  
 self.data\_format = data\_format  
 if self.data\_format not in ["channels\_last", "channels\_first"]:  
 raise NotImplementedError   
 self.normalized\_shape = (normalized\_shape, )  
   
 def forward(self, x):  
 # 根据数据格式进行归一化  
 if self.data\_format == "channels\_last":  
 return F.layer\_norm(x, self.normalized\_shape, self.weight, self.bias, self.eps)  
 elif self.data\_format == "channels\_first":  
 u = x.mean(1, keepdim=True) # 计算均值  
 s = (x - u).pow(2).mean(1, keepdim=True) # 计算方差  
 x = (x - u) / torch.sqrt(s + self.eps) # 标准化  
 x = self.weight[:, None, None] \* x + self.bias[:, None, None] # 应用权重和偏置  
 return x  
  
class Block(nn.Module):  
 """ ConvNeXtV2的基本块，包含深度可分离卷积、归一化、激活等操作。  
   
 Args:  
 dim (int): 输入通道数。  
 """  
 def \_\_init\_\_(self, dim):  
 super().\_\_init\_\_()  
 self.dwconv = nn.Conv2d(dim, dim, kernel\_size=7, padding=3, groups=dim) # 深度卷积  
 self.norm = LayerNorm(dim, eps=1e-6) # 归一化层  
 self.pwconv1 = nn.Linear(dim, 4 \* dim) # 1x1卷积（用线性层实现）  
 self.act = nn.GELU() # 激活函数  
 self.pwconv2 = nn.Linear(4 \* dim, dim) # 1x1卷积（用线性层实现）  
  
 def forward(self, x):  
 input = x # 保存输入以便后续残差连接  
 x = self.dwconv(x) # 深度卷积  
 x = x.permute(0, 2, 3, 1) # 转换维度顺序  
 x = self.norm(x) # 归一化  
 x = self.pwconv1(x) # 第一个1x1卷积  
 x = self.act(x) # 激活  
 x = self.pwconv2(x) # 第二个1x1卷积  
 x = x.permute(0, 3, 1, 2) # 恢复维度顺序  
  
 x = input + x # 残差连接  
 return x  
  
class ConvNeXtV2(nn.Module):  
 """ ConvNeXt V2模型定义。  
   
 Args:  
 in\_chans (int): 输入图像的通道数。默认值为3。  
 num\_classes (int): 分类头的类别数。默认值为1000。  
 depths (tuple(int)): 每个阶段的块数。默认值为[3, 3, 9, 3]。  
 dims (int): 每个阶段的特征维度。默认值为[96, 192, 384, 768]。  
 """  
 def \_\_init\_\_(self, in\_chans=3, num\_classes=1000,   
 depths=[3, 3, 9, 3], dims=[96, 192, 384, 768]):  
 super().\_\_init\_\_()  
 self.downsample\_layers = nn.ModuleList() # 下采样层列表  
 # 定义stem层  
 stem = nn.Sequential(  
 nn.Conv2d(in\_chans, dims[0], kernel\_size=4, stride=4),  
 LayerNorm(dims[0], eps=1e-6, data\_format="channels\_first")  
 )  
 self.downsample\_layers.append(stem)  
 # 定义下采样层  
 for i in range(3):  
 downsample\_layer = nn.Sequential(  
 LayerNorm(dims[i], eps=1e-6, data\_format="channels\_first"),  
 nn.Conv2d(dims[i], dims[i+1], kernel\_size=2, stride=2),  
 )  
 self.downsample\_layers.append(downsample\_layer)  
  
 self.stages = nn.ModuleList() # 特征分辨率阶段列表  
 for i in range(4):  
 stage = nn.Sequential(  
 \*[Block(dim=dims[i]) for \_ in range(depths[i])] # 添加多个Block  
 )  
 self.stages.append(stage)  
  
 self.norm = nn.LayerNorm(dims[-1], eps=1e-6) # 最后的归一化层  
 self.head = nn.Linear(dims[-1], num\_classes) # 分类头  
  
 def forward(self, x):  
 res = [] # 存储每个阶段的输出  
 for i in range(4):  
 x = self.downsample\_layers[i](x) # 下采样  
 x = self.stages[i](x) # 通过当前阶段  
 res.append(x) # 保存输出  
 return res # 返回所有阶段的输出  
```  
  
### 代码核心部分说明：  
1. \*\*LayerNorm\*\*: 自定义的层归一化实现，支持不同的数据格式，确保模型在训练时的稳定性。  
2. \*\*Block\*\*: ConvNeXtV2的基本构建块，使用深度卷积和1x1卷积进行特征提取，并实现残差连接。  
3. \*\*ConvNeXtV2\*\*: 主模型类，定义了模型的整体结构，包括下采样层和多个特征提取阶段，最后通过分类头输出结果。```

这个程序文件定义了一个名为 `ConvNeXtV2` 的深度学习模型，主要用于图像分类任务。它的结构和功能可以分为几个主要部分。  
  
首先，文件中引入了一些必要的库，包括 PyTorch 和 NumPy。接着，定义了几个类，分别实现了不同的功能。  
  
`LayerNorm` 类实现了层归一化，支持两种数据格式：`channels\_last` 和 `channels\_first`。在前向传播中，根据输入数据的格式，选择合适的归一化方法。  
  
`GRN` 类实现了全局响应归一化层，通过计算输入的 L2 范数来进行归一化，并引入了可学习的参数 `gamma` 和 `beta`，以便在训练过程中调整输出。  
  
`Block` 类是 ConvNeXtV2 的基本构建块，包含深度可分离卷积、层归一化、点卷积、激活函数（GELU）和 GRN。它的前向传播过程包括对输入进行卷积、归一化、激活和再次卷积，并通过残差连接将输入与输出相加。  
  
`ConvNeXtV2` 类是整个模型的核心部分，构造了模型的整体结构。它包括一个初始的卷积层和多个下采样层，接着是多个特征分辨率阶段，每个阶段由多个 `Block` 组成。模型的最后部分是一个层归一化和一个线性分类头。模型的初始化过程中，还调用了 `\_init\_weights` 方法来初始化卷积和线性层的权重。  
  
此外，文件中还定义了一个 `update\_weight` 函数，用于更新模型的权重。它会将加载的权重与模型当前的权重进行匹配，并更新相应的参数。  
  
最后，文件提供了一系列函数（如 `convnextv2\_atto`、`convnextv2\_femto` 等），用于创建不同规模的 ConvNeXtV2 模型。这些函数允许用户根据需要加载预训练的权重，并返回相应的模型实例。  
  
总体来说，这个文件实现了一个灵活且可扩展的图像分类模型，具有多种配置选项和预训练权重加载功能，适合用于各种计算机视觉任务。

```以下是保留的核心代码部分，并添加了详细的中文注释：  
  
```python  
import torch  
import torch.nn as nn  
from torch.nn.modules.utils import \_pair as to\_2tuple  
from functools import partial  
  
class Mlp(nn.Module):  
 """多层感知机（MLP）模块"""  
 def \_\_init\_\_(self, in\_features, hidden\_features=None, out\_features=None, act\_layer=nn.GELU, drop=0.):  
 super().\_\_init\_\_()  
 out\_features = out\_features or in\_features # 输出特征数  
 hidden\_features = hidden\_features or in\_features # 隐藏层特征数  
 self.fc1 = nn.Conv2d(in\_features, hidden\_features, 1) # 第一个卷积层  
 self.dwconv = DWConv(hidden\_features) # 深度卷积层  
 self.act = act\_layer() # 激活函数  
 self.fc2 = nn.Conv2d(hidden\_features, out\_features, 1) # 第二个卷积层  
 self.drop = nn.Dropout(drop) # Dropout层  
  
 def forward(self, x):  
 x = self.fc1(x) # 前向传播  
 x = self.dwconv(x)  
 x = self.act(x)  
 x = self.drop(x)  
 x = self.fc2(x)  
 x = self.drop(x)  
 return x  
  
class Attention(nn.Module):  
 """注意力模块"""  
 def \_\_init\_\_(self, d\_model):  
 super().\_\_init\_\_()  
 self.proj\_1 = nn.Conv2d(d\_model, d\_model, 1) # 线性投影  
 self.activation = nn.GELU() # 激活函数  
 self.spatial\_gating\_unit = LSKblock(d\_model) # 空间门控单元  
 self.proj\_2 = nn.Conv2d(d\_model, d\_model, 1) # 线性投影  
  
 def forward(self, x):  
 shortcut = x.clone() # 残差连接  
 x = self.proj\_1(x)  
 x = self.activation(x)  
 x = self.spatial\_gating\_unit(x)  
 x = self.proj\_2(x)  
 x = x + shortcut # 残差连接  
 return x  
  
class Block(nn.Module):  
 """网络的基本块"""  
 def \_\_init\_\_(self, dim, mlp\_ratio=4., drop=0., drop\_path=0., act\_layer=nn.GELU):  
 super().\_\_init\_\_()  
 self.norm1 = nn.BatchNorm2d(dim) # 归一化层  
 self.norm2 = nn.BatchNorm2d(dim)  
 self.attn = Attention(dim) # 注意力模块  
 self.drop\_path = nn.Identity() if drop\_path <= 0. else DropPath(drop\_path) # 随机深度  
 mlp\_hidden\_dim = int(dim \* mlp\_ratio) # MLP隐藏层维度  
 self.mlp = Mlp(in\_features=dim, hidden\_features=mlp\_hidden\_dim, act\_layer=act\_layer, drop=drop) # MLP模块  
  
 def forward(self, x):  
 x = x + self.drop\_path(self.attn(self.norm1(x))) # 添加注意力模块的输出  
 x = x + self.drop\_path(self.mlp(self.norm2(x))) # 添加MLP模块的输出  
 return x  
  
class LSKNet(nn.Module):  
 """LSKNet模型"""  
 def \_\_init\_\_(self, img\_size=224, in\_chans=3, embed\_dims=[64, 128, 256, 512], depths=[3, 4, 6, 3]):  
 super().\_\_init\_\_()  
 self.num\_stages = len(depths) # 网络阶段数  
 for i in range(self.num\_stages):  
 # 每个阶段的嵌入层  
 patch\_embed = OverlapPatchEmbed(img\_size=img\_size // (2 \*\* i), in\_chans=in\_chans if i == 0 else embed\_dims[i - 1], embed\_dim=embed\_dims[i])  
 # 每个阶段的块  
 block = nn.ModuleList([Block(dim=embed\_dims[i]) for \_ in range(depths[i])])  
 setattr(self, f"patch\_embed{i + 1}", patch\_embed)  
 setattr(self, f"block{i + 1}", block)  
  
 def forward(self, x):  
 outs = []  
 for i in range(self.num\_stages):  
 patch\_embed = getattr(self, f"patch\_embed{i + 1}")  
 block = getattr(self, f"block{i + 1}")  
 x, \_, \_ = patch\_embed(x) # 嵌入  
 for blk in block:  
 x = blk(x) # 通过块  
 outs.append(x) # 保存输出  
 return outs  
  
class DWConv(nn.Module):  
 """深度卷积模块"""  
 def \_\_init\_\_(self, dim=768):  
 super(DWConv, self).\_\_init\_\_()  
 self.dwconv = nn.Conv2d(dim, dim, 3, 1, 1, bias=True, groups=dim) # 深度卷积  
  
 def forward(self, x):  
 return self.dwconv(x) # 前向传播  
  
def lsknet\_t(weights=''):  
 """构建LSKNet-t模型"""  
 model = LSKNet(embed\_dims=[32, 64, 160, 256], depths=[3, 3, 5, 2])  
 if weights:  
 model.load\_state\_dict(torch.load(weights)['state\_dict']) # 加载权重  
 return model  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 model = lsknet\_t('lsk\_t\_backbone-2ef8a593.pth') # 创建模型实例  
 inputs = torch.randn((1, 3, 640, 640)) # 随机输入  
 for i in model(inputs):  
 print(i.size()) # 输出每个阶段的输出尺寸  
```  
  
### 代码核心部分说明：  
1. \*\*Mlp类\*\*：实现了一个多层感知机模块，包含两个卷积层和一个深度卷积层，使用激活函数和Dropout。  
2. \*\*Attention类\*\*：实现了一个注意力机制模块，包含两个线性投影和一个空间门控单元。  
3. \*\*Block类\*\*：网络的基本构建块，包含注意力模块和MLP模块，使用残差连接。  
4. \*\*LSKNet类\*\*：整个网络的结构，包含多个阶段，每个阶段由嵌入层和多个Block组成。  
5. \*\*DWConv类\*\*：实现了深度卷积操作，用于特征提取。  
6. \*\*lsknet\_t函数\*\*：用于构建LSKNet-t模型，并可选择加载预训练权重。```

这个程序文件定义了一个名为 `lsknet.py` 的深度学习模型，主要用于图像处理任务。该模型基于 PyTorch 框架构建，包含多个模块和类，主要包括 Mlp、LSKblock、Attention、Block、OverlapPatchEmbed、LSKNet、DWConv 等。  
  
首先，程序导入了必要的库，包括 PyTorch 和一些其他的工具类。接着，定义了一个多层感知机（Mlp）类，该类包含两个卷积层和一个深度卷积层（DWConv），用于对输入特征进行处理，并在每个卷积层后应用激活函数和 dropout。  
  
然后，定义了 `LSKblock` 类，它实现了一种特殊的注意力机制，结合了空间卷积和通道注意力。该类首先通过不同的卷积操作提取特征，然后通过平均和最大池化操作生成注意力权重，最后将这些权重应用于输入特征，以增强模型的表达能力。  
  
接下来，定义了 `Attention` 类，它使用 `LSKblock` 作为其核心组件，包含两个线性投影层和一个激活函数。该类的前向传播过程包括对输入进行投影、激活、注意力计算和最终的残差连接。  
  
`Block` 类则将上述的注意力机制和多层感知机结合在一起，形成一个完整的模块。它包含了批归一化和 dropout 机制，并通过残差连接来增强信息流动。  
  
`OverlapPatchEmbed` 类用于将输入图像分割成重叠的补丁，并通过卷积层进行嵌入，输出嵌入后的特征和补丁的高宽信息。  
  
`LSKNet` 类是整个模型的核心，包含多个阶段（stages），每个阶段由补丁嵌入、多个 `Block` 和归一化层组成。该类在初始化时根据给定的参数构建各个阶段，并在前向传播时依次处理输入数据，最终输出多个阶段的特征。  
  
`DWConv` 类实现了深度卷积操作，用于在 `Mlp` 类中进行特征处理。  
  
此外，程序还定义了 `update\_weight` 函数，用于更新模型的权重，确保模型字典和权重字典中的键匹配，并且形状一致。  
  
最后，提供了两个函数 `lsknet\_t` 和 `lsknet\_s`，分别用于创建不同配置的 LSKNet 模型，并可以选择加载预训练权重。在主程序中，创建了一个 `lsknet\_t` 模型实例，并对随机生成的输入进行前向传播，打印输出特征的尺寸。  
  
整体来看，这个文件实现了一个灵活且强大的图像处理模型，利用了现代深度学习中的多种技术，如注意力机制、残差连接和深度卷积等。

```以下是经过简化并添加详细中文注释的核心代码部分：  
  
```python  
import torch  
from ultralytics.data import ClassificationDataset, build\_dataloader  
from ultralytics.engine.validator import BaseValidator  
from ultralytics.utils.metrics import ClassifyMetrics, ConfusionMatrix  
from ultralytics.utils.plotting import plot\_images  
  
class ClassificationValidator(BaseValidator):  
 """  
 分类模型验证器类，继承自BaseValidator类。  
 该类用于基于分类模型的验证。  
 """  
  
 def \_\_init\_\_(self, dataloader=None, save\_dir=None, pbar=None, args=None, \_callbacks=None):  
 """初始化ClassificationValidator实例，设置数据加载器、保存目录、进度条和参数。"""  
 super().\_\_init\_\_(dataloader, save\_dir, pbar, args, \_callbacks)  
 self.targets = None # 存储真实标签  
 self.pred = None # 存储预测结果  
 self.args.task = 'classify' # 设置任务类型为分类  
 self.metrics = ClassifyMetrics() # 初始化分类指标  
  
 def init\_metrics(self, model):  
 """初始化混淆矩阵、类名及准确率指标。"""  
 self.names = model.names # 获取类别名称  
 self.nc = len(model.names) # 获取类别数量  
 self.confusion\_matrix = ConfusionMatrix(nc=self.nc, conf=self.args.conf, task='classify') # 初始化混淆矩阵  
 self.pred = [] # 初始化预测结果列表  
 self.targets = [] # 初始化真实标签列表  
  
 def preprocess(self, batch):  
 """预处理输入批次数据并返回。"""  
 batch['img'] = batch['img'].to(self.device, non\_blocking=True) # 将图像数据移动到设备上  
 batch['img'] = batch['img'].half() if self.args.half else batch['img'].float() # 根据参数选择数据类型  
 batch['cls'] = batch['cls'].to(self.device) # 将标签数据移动到设备上  
 return batch  
  
 def update\_metrics(self, preds, batch):  
 """使用模型预测和批次目标更新运行指标。"""  
 n5 = min(len(self.names), 5) # 获取前5个预测类别  
 self.pred.append(preds.argsort(1, descending=True)[:, :n5]) # 记录预测结果  
 self.targets.append(batch['cls']) # 记录真实标签  
  
 def finalize\_metrics(self, \*args, \*\*kwargs):  
 """最终化模型的指标，如混淆矩阵和速度。"""  
 self.confusion\_matrix.process\_cls\_preds(self.pred, self.targets) # 处理预测结果和真实标签  
 self.metrics.speed = self.speed # 记录速度  
 self.metrics.confusion\_matrix = self.confusion\_matrix # 记录混淆矩阵  
  
 def get\_stats(self):  
 """返回通过处理目标和预测获得的指标字典。"""  
 self.metrics.process(self.targets, self.pred) # 处理真实标签和预测结果  
 return self.metrics.results\_dict # 返回结果字典  
  
 def build\_dataset(self, img\_path):  
 """创建并返回ClassificationDataset实例，使用给定的图像路径和预处理参数。"""  
 return ClassificationDataset(root=img\_path, args=self.args, augment=False, prefix=self.args.split)  
  
 def get\_dataloader(self, dataset\_path, batch\_size):  
 """构建并返回分类任务的数据加载器。"""  
 dataset = self.build\_dataset(dataset\_path) # 构建数据集  
 return build\_dataloader(dataset, batch\_size, self.args.workers, rank=-1) # 返回数据加载器  
  
 def print\_results(self):  
 """打印YOLO模型的评估指标。"""  
 pf = '%22s' + '%11.3g' \* len(self.metrics.keys) # 打印格式  
 LOGGER.info(pf % ('all', self.metrics.top1, self.metrics.top5)) # 打印总体准确率  
  
 def plot\_val\_samples(self, batch, ni):  
 """绘制验证图像样本。"""  
 plot\_images(  
 images=batch['img'],  
 batch\_idx=torch.arange(len(batch['img'])),  
 cls=batch['cls'].view(-1), # 使用.view()而不是.squeeze()以适应分类模型  
 fname=self.save\_dir / f'val\_batch{ni}\_labels.jpg',  
 names=self.names  
 )  
  
 def plot\_predictions(self, batch, preds, ni):  
 """在输入图像上绘制预测结果并保存结果。"""  
 plot\_images(batch['img'],  
 batch\_idx=torch.arange(len(batch['img'])),  
 cls=torch.argmax(preds, dim=1), # 获取预测类别  
 fname=self.save\_dir / f'val\_batch{ni}\_pred.jpg',  
 names=self.names)  
```  
  
### 代码核心部分解释：  
1. \*\*初始化方法\*\*：设置验证器的基本参数，包括数据加载器、保存目录、进度条和其他参数。  
2. \*\*指标初始化\*\*：为混淆矩阵和准确率指标做准备。  
3. \*\*数据预处理\*\*：将输入数据转换为适合模型处理的格式。  
4. \*\*更新指标\*\*：在每个批次后更新预测结果和真实标签。  
5. \*\*最终化指标\*\*：计算最终的混淆矩阵和速度等指标。  
6. \*\*数据集和数据加载器构建\*\*：创建数据集和相应的数据加载器。  
7. \*\*结果打印和绘图\*\*：输出评估结果并可视化验证样本和预测结果。```

这个程序文件 `val.py` 是用于对基于分类模型的验证过程进行实现的，主要用于 YOLO（You Only Look Once）系列模型的分类任务。文件中定义了一个名为 `ClassificationValidator` 的类，继承自 `BaseValidator`，其主要功能是评估分类模型的性能。  
  
在类的初始化方法中，接受了一些参数，包括数据加载器、保存目录、进度条、参数字典等。初始化后，设置了一些属性，如目标值和预测值，并将任务类型设置为“分类”。同时，创建了一个 `ClassifyMetrics` 实例，用于后续的性能指标计算。  
  
`get\_desc` 方法返回一个格式化的字符串，概述了分类指标，包括类别、Top-1 准确率和 Top-5 准确率。`init\_metrics` 方法则用于初始化混淆矩阵、类别名称和准确率的计算。  
  
在 `preprocess` 方法中，对输入的批次数据进行预处理，包括将图像数据转移到指定设备（如 GPU），并根据参数决定数据类型（半精度或单精度）。`update\_metrics` 方法则在每个批次中更新模型的预测结果和真实目标，以便后续计算性能指标。  
  
`finalize\_metrics` 方法用于最终化模型的性能指标，包括处理混淆矩阵和计算速度。如果启用了绘图功能，还会生成混淆矩阵的可视化图像。`get\_stats` 方法返回一个字典，包含通过处理目标和预测得到的指标结果。  
  
`build\_dataset` 方法用于创建分类数据集实例，`get\_dataloader` 方法则根据给定的参数构建数据加载器，以便于批量处理数据。  
  
`print\_results` 方法用于打印评估指标，展示模型在验证集上的表现。`plot\_val\_samples` 和 `plot\_predictions` 方法分别用于绘制验证样本和模型预测结果，生成可视化图像并保存到指定目录。  
  
整体来看，这个文件实现了一个完整的分类模型验证流程，包括数据预处理、性能指标计算、结果输出和可视化，适用于使用 YOLO 模型进行图像分类任务的场景。

```以下是代码中最核心的部分，并附上详细的中文注释：  
  
```python  
import 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，如果没有则使用CPU  
  
 # 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')  
  
 # 将路径转换为Unix风格（使用'/'作为分隔符）  
 unix\_style\_path = data\_path.replace(os.sep, '/')  
 # 获取数据集目录路径  
 directory\_path = os.path.dirname(unix\_style\_path)  
  
 # 读取YAML配置文件  
 with open(data\_path, 'r') as file:  
 data = yaml.load(file, Loader=yaml.FullLoader)  
  
 # 修改YAML文件中的训练、验证和测试数据路径  
 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, # 指定训练所用的设备（GPU或CPU）  
 workers=workers, # 指定用于数据加载的工作进程数  
 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, # 指定每个批次的样本数量  
 )  
```  
  
### 代码说明：  
1. \*\*导入必要的库\*\*：导入`os`、`torch`、`yaml`和YOLO模型库，以便进行文件操作、深度学习和模型训练。  
2. \*\*设置训练参数\*\*：定义了工作进程数、批次大小和设备类型（GPU或CPU）。  
3. \*\*读取和修改YAML配置文件\*\*：读取数据集的配置文件，修改其中的训练、验证和测试数据路径，并将修改后的内容写回文件。  
4. \*\*加载YOLO模型\*\*：使用指定的配置文件和预训练权重加载YOLO模型。  
5. \*\*开始训练模型\*\*：调用`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、图像大小、训练轮数和批次大小。```

这个程序文件 `train.py` 是一个用于训练 YOLO（You Only Look Once）模型的脚本，主要用于目标检测或分割任务。程序首先导入了一些必要的库，包括 `os`、`torch`、`yaml` 和 `ultralytics` 中的 YOLO 模型。它还使用 `matplotlib` 来设置图形显示的后端。  
  
在主程序中，首先定义了一些参数，包括工作进程数 `workers` 和批次大小 `batch`。批次大小可以根据计算机的显存和内存进行调整，如果出现显存不足的情况，可以适当降低这个值。接着，程序会检查是否有可用的 GPU，如果有则将设备设置为 GPU（"0"），否则使用 CPU。  
  
接下来，程序构建了数据集的 YAML 文件的绝对路径，并将路径格式转换为 Unix 风格，以便于后续处理。然后，程序读取这个 YAML 文件，并提取其中的数据。特别地，它会检查 YAML 文件中是否包含训练、验证和测试数据的路径，如果包含，则将这些路径修改为当前目录下的相应子目录（train、val、test），并将修改后的数据写回到 YAML 文件中。  
  
程序还提到，不同的 YOLO 模型对设备的要求不同，如果当前模型在训练时出现错误，可以尝试使用其他的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，传入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。这一系列设置确保了模型能够在指定的环境下进行有效的训练。

```以下是代码中最核心的部分，并附上详细的中文注释：  
  
```python  
# 导入所需的跟踪器类  
from .bot\_sort import BOTSORT # 导入BOTSORT类，用于目标跟踪  
from .byte\_tracker import BYTETracker # 导入BYTETracker类，用于目标跟踪  
from .track import register\_tracker # 导入注册跟踪器的函数  
  
# 定义模块的公开接口，允许外部简单导入  
\_\_all\_\_ = 'register\_tracker', 'BOTSORT', 'BYTETracker'   
```  
  
### 注释说明：  
1. \*\*导入部分\*\*：  
 - `from .bot\_sort import BOTSORT`：从当前模块的 `bot\_sort` 文件中导入 `BOTSORT` 类。这个类通常用于实现一种特定的目标跟踪算法。  
 - `from .byte\_tracker import BYTETracker`：从当前模块的 `byte\_tracker` 文件中导入 `BYTETracker` 类。这个类也是用于目标跟踪，可能采用不同的算法或策略。  
 - `from .track import register\_tracker`：从当前模块的 `track` 文件中导入 `register\_tracker` 函数。这个函数可能用于注册不同的跟踪器，以便在系统中使用。  
  
2. \*\*公开接口\*\*：  
 - `\_\_all\_\_` 变量定义了模块的公开接口，包含了可以被外部导入的名称。这样做的目的是为了简化外部用户的导入过程，用户只需导入 `register\_tracker`、`BOTSORT` 和 `BYTETracker`，而不需要了解模块内部的具体实现细节。```

这个程序文件是一个 Python 模块的初始化文件，通常用于定义模块的公共接口。在这个文件中，首先有一个注释，提到这是与 Ultralytics YOLO 相关的代码，并且该代码遵循 AGPL-3.0 许可证。  
  
接下来，文件导入了三个组件：`BOTSORT`、`BYTETracker` 和 `register\_tracker`。这些组件分别来自于同一目录下的其他模块文件。`BOTSORT` 和 `BYTETracker` 可能是两种不同的跟踪算法或类，而 `register\_tracker` 则可能是一个用于注册跟踪器的函数。  
  
最后，`\_\_all\_\_` 变量被定义为一个包含字符串的元组，列出了可以通过 `from module import \*` 语句导入的公共接口。这意味着，当其他模块使用这种方式导入时，只会导入 `register\_tracker`、`BOTSORT` 和 `BYTETracker` 这三个名称，从而控制了模块的可见性和使用方式。  
  
总的来说，这个文件的主要作用是组织和暴露跟踪相关的功能，使得其他模块可以方便地使用这些功能。

### 整体功能和构架概括  
  
该项目的整体功能主要集中在深度学习模型的训练、验证和推理，特别是针对目标检测、图像分类和分割任务。项目使用了 YOLO（You Only Look Once）系列模型以及其他一些深度学习架构（如 ConvNeXt 和 LSKNet），并提供了相应的工具和模块来支持这些功能。项目的结构包括模型定义、训练脚本、验证脚本、用户界面、数据处理工具和跟踪器模块。  
  
以下是项目中各个文件的功能整理表：  
  
| 文件路径 | 功能描述 |  
|----------------------------------------------------|------------------------------------------------------------------------------------------|  
| `C:\codeseg\codenew\code\ui.py` | 提供一个用户界面，通过 Streamlit 启动 Web 应用以运行指定的 Python 脚本。 |  
| `C:\codeseg\codenew\code\ultralytics\nn\backbone\convnextv2.py` | 定义 ConvNeXtV2 模型，包含模型的构建块、权重初始化和不同规模模型的创建函数。 |  
| `C:\codeseg\codenew\code\ultralytics\nn\backbone\lsknet.py` | 实现 LSKNet 模型，包含多个模块和类，结合注意力机制和多层感知机，适用于图像处理任务。 |  
| `C:\codeseg\codenew\code\ultralytics\models\yolo\classify\val.py` | 实现分类模型的验证流程，包括数据预处理、性能指标计算和结果可视化。 |  
| `C:\codeseg\codenew\code\train.py` | 训练 YOLO 模型的主脚本，处理数据集配置、模型加载和训练过程的启动。 |  
| `C:\codeseg\codenew\code\ultralytics\trackers\\_\_init\_\_.py` | 初始化跟踪器模块，导入并暴露 BOTSORT 和 BYTETracker 等跟踪算法。 |  
| `C:\codeseg\codenew\code\ultralytics\nn\extra\_modules\ops\_dcnv3\modules\\_\_init\_\_.py` | 初始化 DCNv3 模块，可能用于实现特定的卷积操作或功能。 |  
| `C:\codeseg\codenew\code\ultralytics\models\rtdetr\model.py` | 定义 RT-DETR 模型，可能用于实时目标检测任务，包含模型结构和前向传播方法。 |  
| `C:\codeseg\codenew\code\ultralytics\models\yolo\detect\val.py` | 实现 YOLO 模型的检测验证流程，计算检测性能指标并进行结果可视化。 |  
| `C:\codeseg\codenew\code\ultralytics\models\yolo\segment\val.py` | 实现 YOLO 模型的分割验证流程，计算分割性能指标并进行结果可视化。 |  
| `C:\codeseg\codenew\code\ultralytics\data\utils.py` | 提供数据处理和转换的工具函数，支持数据集的构建和预处理。 |  
| `C:\codeseg\codenew\code\ultralytics\data\build.py` | 构建数据集的逻辑，可能包括数据加载器的创建和数据增强等功能。 |  
| `C:\codeseg\codenew\code\ultralytics\models\yolo\detect\\_\_init\_\_.py` | 初始化 YOLO 检测模块，可能用于导入和组织检测相关的功能和类。 |  
  
这个表格总结了项目中各个文件的主要功能，展示了项目的模块化设计和各个部分之间的关系。整体上，该项目提供了一个完整的框架，支持从数据处理到模型训练、验证和推理的全流程。

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。