# 体操动作识别系统源码 # [一条龙教学YOLOV8标注好的数据集一键训练\_70+全套改进创新点发刊\_Web前端展示]

## 1. 研究背景与意义

研究背景与意义  
  
随着科技的不断进步，计算机视觉技术在各个领域的应用日益广泛，尤其是在体育领域，运动动作的识别与分析已成为提升运动员训练效果和比赛表现的重要手段。体操作为一项对运动员身体素质和技术要求极高的运动，其动作的复杂性和多样性使得传统的动作识别方法面临诸多挑战。近年来，深度学习技术的快速发展为体操动作识别提供了新的解决方案，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。YOLOv8作为该系列的最新版本，具备更强的特征提取能力和更快的推理速度，适合用于复杂动作的实时识别。  
  
本研究旨在基于改进的YOLOv8模型，构建一个高效的体操动作识别系统，以提升体操训练的智能化水平。为实现这一目标，我们使用了CaliAI\_v1.0.0数据集，该数据集包含1100张图像，涵盖了8种体操动作类别，包括弯臂平衡（bent\_arm\_planche）、前引体向上（front\_lever）、前引体触碰（front\_lever\_touch）、倒立十字（inverted\_cross）、铁十字（iron\_cross）、马耳他（maltese）、平衡（planche）和环马耳他（rings\_maltese）。这些动作不仅具有代表性，而且在体操比赛中频繁出现，能够为模型的训练和评估提供丰富的样本。  
  
体操动作的复杂性体现在其多样的姿态和变化的动态特征上，传统的基于规则的方法难以有效捕捉这些细微的动作差异。通过采用YOLOv8模型，我们能够利用其深度学习的优势，自动提取图像中的特征，从而实现对不同体操动作的准确识别。此外，改进的YOLOv8模型在处理多类别检测时表现出色，能够在保证识别精度的同时，提高处理速度，满足实时应用的需求。  
  
本研究的意义不仅在于技术层面的创新，更在于推动体操训练的智能化和科学化。通过构建高效的体操动作识别系统，教练员和运动员可以实时获取动作分析反馈，从而及时调整训练策略，优化动作技巧，减少受伤风险。此外，该系统还可以为体操比赛的裁判提供辅助判定，提升比赛的公正性和准确性。  
  
综上所述，基于改进YOLOv8的体操动作识别系统的研究，不仅具有重要的学术价值，也对实际应用具有深远的影响。通过本研究，我们希望能够为体操运动的智能化发展贡献一份力量，同时为相关领域的研究提供新的思路和方法。

## 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  
  
在现代体育科学与人工智能的交叉领域，体操动作识别系统的开发与应用正逐渐成为研究的热点。为此，我们构建了一个名为“CaliAI\_v1.0.0”的数据集，旨在为改进YOLOv8模型在体操动作识别方面的性能提供支持。该数据集包含了8个不同的体操动作类别，涵盖了从基础到高级的多种体操姿势，具有广泛的应用潜力和研究价值。  
  
“CaliAI\_v1.0.0”数据集的类别数量为8，具体类别包括：弯臂平衡（bent\_arm\_planche）、前杠（front\_lever）、前杠触碰（front\_lever\_touch）、倒立十字（inverted\_cross）、铁十字（iron\_cross）、马耳他式（maltese）、平衡（planche）以及环马耳他式（rings\_maltese）。这些类别不仅代表了体操运动中的不同技术动作，还体现了运动员在训练和比赛中所需的不同技能水平。每个类别的动作都具有独特的身体姿态和运动轨迹，这为动作识别系统的训练提供了丰富的样本。  
  
在数据集的构建过程中，我们注重数据的多样性和真实性。每个类别的样本均来源于专业体操运动员的实际表演，确保了数据的高质量和可靠性。此外，为了增强模型的泛化能力，我们还在不同的环境条件下采集了数据，包括不同的光照、背景和摄像角度。这种多样化的数据采集方式使得“CaliAI\_v1.0.0”数据集在体操动作识别的研究中具有较强的适应性和实用性。  
  
数据集中的每个动作类别都配有详细的标注信息，包括动作的起始和结束帧、关键点位置以及运动员的身体姿态。这些标注信息为YOLOv8模型的训练提供了必要的监督信号，使得模型能够在学习过程中准确识别和分类不同的体操动作。同时，我们还对数据集进行了预处理，包括图像的归一化、数据增强等，以提高模型的训练效率和识别准确率。  
  
“CaliAI\_v1.0.0”数据集的推出，不仅为体操动作识别领域的研究提供了宝贵的资源，也为相关技术的应用奠定了基础。通过对该数据集的深入研究，我们期望能够推动体操动作识别技术的发展，为教练员和运动员提供更加科学的训练指导和反馈。同时，该数据集的应用也将为智能体育设备的研发提供数据支持，助力体操运动的智能化和数字化转型。  
  
总之，“CaliAI\_v1.0.0”数据集以其丰富的类别、真实的样本和详尽的标注，为改进YOLOv8的体操动作识别系统提供了坚实的基础。随着研究的深入和技术的不断进步，我们相信这一数据集将在体操动作识别领域发挥重要作用，推动相关研究的不断发展与创新。

## 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.ixigua.com/7404473917358506534?logTag=c807d0cbc21c0ef59de5

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.ixigua.com/7404474678003106304?logTag=1f1041108cd1f708b01a

## 6. 手把手YOLOV8训练视频教程（零基础小白有手就能学会）

https://www.ixigua.com/7404477157818401292?logTag=d31a2dfd1983c9668658

## 7.70+种全套YOLOV8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.ixigua.com/7404478314661806627?logTag=29066f8288e3f4eea3a4

## 8. 70+全套YOLOV8创新点原理讲解（非科班也可以轻松写刊发刊，V12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

## 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. 原始YOLOV8算法原理

原始YOLOv8算法原理  
  
YOLOv8是由Ultralytics公司于2023年推出的最新目标检测算法，标志着YOLO系列的又一次重大进步。作为YOLOv5和YOLOv7的继承者，YOLOv8不仅在性能上进行了优化，还在结构设计上引入了多项创新。该算法的核心在于其高效的网络架构，主要由输入层、主干网络、颈部网络和头部网络构成，能够在多种应用场景中实现实时目标检测。  
  
在YOLOv8中，输入图像首先被缩放至特定的尺寸，以满足模型的输入要求。主干网络采用了CSP（Cross Stage Partial）结构，通过卷积操作对图像进行下采样，从而提取出有用的特征。与YOLOv5不同的是，YOLOv8使用了C2f模块替代了YOLOv5中的C3模块。C2f模块的设计灵感来源于YOLOv7的E-ELAN结构，采用了跨层分支连接的方式，增强了模型的梯度流动，改善了特征提取的效果。具体而言，C2f模块由多个CBS（Convolution + Batch Normalization + SiLU）模块和多个Bottleneck网络组成，能够有效地缓解深层网络中的梯度消失问题，提升特征的重用率。  
  
在主干网络的末尾，YOLOv8引入了SPPF（Spatial Pyramid Pooling Fast）模块，通过三个最大池化层对多尺度特征进行处理，进一步增强了网络的特征抽象能力。这一设计使得YOLOv8能够在处理不同尺寸的目标时，保持较高的检测精度。  
  
颈部网络采用了PAN-FPN（Path Aggregation Network - Feature Pyramid Network）结构，这一结构的优势在于能够有效地融合不同尺度的特征图信息。通过自下而上的融合过程，YOLOv8能够将高层特征与中层和浅层特征进行深度整合，从而提高目标检测的准确性。具体来说，PAN-FPN结构在上采样阶段直接将高层特征与中层特征进行连接，确保了细节信息与语义信息的充分融合。  
  
YOLOv8的检测网络采用了Anchor-Free的检测方式，显著简化了目标检测的过程。通过解耦头结构，YOLOv8能够将分类和回归任务分开处理，分别计算目标的类别和边框回归。这一设计不仅提高了模型的灵活性，还在一定程度上降低了计算复杂度。此外，YOLOv8使用了新的损失函数VFLLoss和DFLLoss+CIoULoss，旨在优化模型的训练过程，提升检测性能。  
  
在样本匹配方面，YOLOv8引入了Task-Aligned的Assigner匹配方式，取代了以往的静态匹配策略。这一改进使得模型在训练过程中能够更好地适应不同的任务需求，提高了检测的鲁棒性。  
  
数据预处理环节，YOLOv8依然沿用了YOLOv5的策略，采用了多种数据增强手段，如马赛克增强、混合增强、空间扰动和颜色扰动等。这些增强手段的引入，不仅丰富了训练数据的多样性，还有效提高了模型的泛化能力。  
  
在YOLOv8的具体实现中，C2f模块的设计尤为关键。该模块通过将特征图分为两个分支，分别进行处理，增强了特征提取的灵活性和丰富性。第一个分支直接将特征图送入Concat层，而第二个分支则通过多个Bottleneck网络进行处理，最终将两个分支的特征进行叠加。这一设计使得C2f模块能够在保持输入输出特征图尺寸一致的前提下，提取出更多的隐藏特征。  
  
YOLOv8的头部网络则负责将颈部网络输出的特征图转换为最终的检测结果。通过解耦的分类和回归分支，YOLOv8能够有效地处理不同尺度的特征图，输出目标的类别和边框信息。具体而言，YOLOv8的头部网络输出三个特征图，分别对应于80x80、40x40和20x20的尺度。这些特征图经过处理后，能够生成目标的边框预测和类别预测，确保了模型在不同场景下的适应性。  
  
总的来说，YOLOv8在目标检测领域的创新和改进，使其在速度和精度上都达到了新的高度。通过引入C2f模块、PAN-FPN结构以及Anchor-Free的检测方式，YOLOv8不仅提高了检测的准确性，还显著提升了模型的计算效率。这些改进使得YOLOv8在实际应用中表现出色，能够满足日益增长的目标检测需求。随着YOLOv8的不断发展和完善，未来的目标检测技术将会更加智能和高效。

## 11. 项目核心源码讲解（再也不用担心看不懂代码逻辑）

```以下是经过简化和注释的核心代码部分，保留了主要的结构和功能。  
  
```python  
import torch  
import torch.nn as nn  
import torch.nn.functional as F  
from timm.models.layers import trunc\_normal\_, DropPath  
  
class LayerNorm(nn.Module):  
 """ 自定义的LayerNorm层，支持两种数据格式：channels\_last和channels\_first。 """  
 def \_\_init\_\_(self, normalized\_shape, eps=1e-6, data\_format="channels\_last"):  
 super().\_\_init\_\_()  
 self.weight = nn.Parameter(torch.ones(normalized\_shape)) # 权重参数  
 self.bias = nn.Parameter(torch.zeros(normalized\_shape)) # 偏置参数  
 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)  
 s = (x - u).pow(2).mean(1, keepdim=True)  
 x = (x - u) / torch.sqrt(s + self.eps)  
 x = self.weight[:, None, None] \* x + self.bias[:, None, None]  
 return x  
  
class Block(nn.Module):  
 """ ConvNeXtV2的基本块，包含深度可分离卷积和前馈网络。 """  
 def \_\_init\_\_(self, dim, drop\_path=0.):  
 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  
 self.drop\_path = DropPath(drop\_path) if drop\_path > 0. else nn.Identity() # 随机深度  
  
 def forward(self, x):  
 input = x  
 x = self.dwconv(x) # 深度卷积  
 x = x.permute(0, 2, 3, 1) # 转换维度  
 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  
 x = input + self.drop\_path(x) # 残差连接  
 return x  
  
class ConvNeXtV2(nn.Module):  
 """ ConvNeXt V2模型，包含多个特征分辨率阶段和残差块。 """  
 def \_\_init\_\_(self, in\_chans=3, num\_classes=1000,   
 depths=[3, 3, 9, 3], dims=[96, 192, 384, 768],   
 drop\_path\_rate=0.):  
 super().\_\_init\_\_()  
 self.downsample\_layers = nn.ModuleList() # 下采样层  
 # 初始化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)  
 # 添加下采样层  
 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  
 dp\_rates = [x.item() for x in torch.linspace(0, drop\_path\_rate, sum(depths))]   
 cur = 0  
 for i in range(4):  
 stage = nn.Sequential(  
 \*[Block(dim=dims[i], drop\_path=dp\_rates[cur + j]) for j in range(depths[i])]  
 )  
 self.stages.append(stage)  
 cur += depths[i]  
  
 self.norm = nn.LayerNorm(dims[-1], eps=1e-6) # 最终归一化层  
 self.head = nn.Linear(dims[-1], num\_classes) # 分类头  
  
 def forward(self, x):  
 for i in range(4):  
 x = self.downsample\_layers[i](x) # 下采样  
 x = self.stages[i](x) # 通过阶段  
 return x # 返回特征  
  
# 创建不同规模的ConvNeXtV2模型的函数  
def convnextv2\_atto(weights='', \*\*kwargs):  
 model = ConvNeXtV2(depths=[2, 2, 6, 2], dims=[40, 80, 160, 320], \*\*kwargs)  
 if weights:  
 model.load\_state\_dict(torch.load(weights)['model'])  
 return model  
  
# 其他模型创建函数类似...  
```  
  
### 主要改动和注释说明：  
1. \*\*LayerNorm\*\*: 自定义的归一化层，支持不同的数据格式。  
2. \*\*Block\*\*: ConvNeXtV2的基本构建块，包含深度卷积、归一化、激活和残差连接。  
3. \*\*ConvNeXtV2\*\*: 主模型，包含下采样层和多个阶段，每个阶段由多个Block组成。  
4. \*\*模型创建函数\*\*: 提供了不同规模的ConvNeXtV2模型的构造方式。  
  
通过以上的简化和注释，代码的核心结构和功能得以保留，同时增加了可读性。```

这个文件实现了ConvNeXt V2模型的各个组件和结构，主要用于深度学习中的图像分类任务。文件中包含了多个类和函数，分别定义了模型的不同部分以及一些辅助功能。  
  
首先，文件导入了必要的库，包括PyTorch的核心模块和一些功能模块。接着，定义了一个名为`LayerNorm`的类，这个类实现了层归一化（Layer Normalization），支持两种数据格式：`channels\_last`和`channels\_first`。层归一化用于对输入数据进行标准化处理，以提高模型的训练稳定性和收敛速度。  
  
接下来，定义了`GRN`类，它实现了全局响应归一化（Global Response Normalization）层。这个层通过对输入进行归一化处理，帮助模型更好地学习特征。  
  
然后是`Block`类，它代表了ConvNeXt V2的基本构建块。每个块包含一个深度可分离卷积层、层归一化、点卷积、激活函数（GELU）以及全局响应归一化。这个块的设计旨在提高模型的表达能力和计算效率。  
  
`ConvNeXtV2`类是模型的主要结构，包含多个阶段，每个阶段由多个`Block`组成。构造函数中定义了输入通道数、分类头的类别数、每个阶段的块数以及特征维度等参数。模型的下采样层和特征提取阶段通过`nn.ModuleList`进行管理。模型的最后一层是一个线性层，用于将特征映射到类别空间。  
  
文件中还定义了一个`update\_weight`函数，用于更新模型的权重。这个函数会检查权重字典中的每个键是否存在于模型字典中，并且形状是否匹配，匹配的权重会被更新到模型中。  
  
最后，文件提供了一系列函数，如`convnextv2\_atto`、`convnextv2\_femto`等，这些函数用于创建不同规模的ConvNeXt V2模型，并可以选择性地加载预训练权重。每个函数根据指定的深度和维度参数初始化模型，并在提供权重路径时加载相应的权重。  
  
总体而言，这个文件实现了ConvNeXt V2模型的核心组件，提供了灵活的模型构建和权重加载功能，适用于各种图像分类任务。

```以下是代码中最核心的部分，并附上详细的中文注释：  
  
```python  
# 导入必要的模块  
from .predict import DetectionPredictor # 导入目标检测预测器  
from .train import DetectionTrainer # 导入目标检测训练器  
from .val import DetectionValidator # 导入目标检测验证器  
  
# 定义模块的公开接口，指定可以被外部访问的类  
\_\_all\_\_ = 'DetectionPredictor', 'DetectionTrainer', 'DetectionValidator'  
```  
  
### 注释说明：  
1. \*\*导入模块\*\*：  
 - `from .predict import DetectionPredictor`：从当前包的 `predict` 模块中导入 `DetectionPredictor` 类，该类负责进行目标检测的预测。  
 - `from .train import DetectionTrainer`：从当前包的 `train` 模块中导入 `DetectionTrainer` 类，该类用于训练目标检测模型。  
 - `from .val import DetectionValidator`：从当前包的 `val` 模块中导入 `DetectionValidator` 类，该类用于验证目标检测模型的性能。  
  
2. \*\*定义公开接口\*\*：  
 - `\_\_all\_\_` 是一个特殊变量，用于定义模块的公共接口。当使用 `from module import \*` 语句时，只有在 `\_\_all\_\_` 中列出的名称会被导入。这里指定了 `DetectionPredictor`、`DetectionTrainer` 和 `DetectionValidator` 这三个类为模块的公共接口，方便用户使用。```

这个程序文件是Ultralytics YOLO项目的一部分，主要用于目标检测的实现。文件名指向的是YOLOv8算法的改进源码和相关的调试、加载、训练教程。文件的内容非常简洁，主要包括了几个重要的模块导入和一个`\_\_all\_\_`声明。  
  
首先，文件的开头有一个注释，表明这是Ultralytics YOLO项目，并且该项目遵循AGPL-3.0许可证。这意味着用户在使用该代码时需要遵循该许可证的条款。  
  
接下来，文件从当前目录中导入了三个类：`DetectionPredictor`、`DetectionTrainer`和`DetectionValidator`。这些类分别负责目标检测的不同方面：`DetectionPredictor`用于进行预测，`DetectionTrainer`用于训练模型，而`DetectionValidator`则用于验证模型的性能。这种模块化的设计使得代码结构清晰，便于维护和扩展。  
  
最后，`\_\_all\_\_`变量的定义指定了当使用`from module import \*`语句时，哪些名称会被导入。在这里，它包含了之前导入的三个类。这意味着如果其他模块想要使用这个模块中的功能，只需导入这三个类即可。  
  
总的来说，这个文件是YOLOv8目标检测框架的核心组成部分之一，负责组织和提供模型训练、预测和验证的功能。

``````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  
 command = f'"{python\_path}" -m streamlit run "{script\_path}"'  
  
 # 执行命令  
 result = subprocess.run(command, shell=True)  
 if result.returncode != 0:  
 print("脚本运行出错。")  
  
  
# 实例化并运行应用  
if \_\_name\_\_ == "\_\_main\_\_":  
 # 指定您的脚本路径  
 script\_path = "web.py" # 这里可以直接指定脚本路径  
  
 # 运行脚本  
 run\_script(script\_path)  
```  
  
### 代码核心部分及注释  
  
1. \*\*导入模块\*\*：  
 ```python  
 import sys  
 import subprocess  
 ```  
 - `sys`模块用于访问与Python解释器相关的变量和函数，例如获取当前Python解释器的路径。  
 - `subprocess`模块用于创建新进程、连接到它们的输入/输出/错误管道，并获取返回码。  
  
2. \*\*定义函数 `run\_script`\*\*：  
 ```python  
 def run\_script(script\_path):  
 ```  
 - 该函数接收一个参数 `script\_path`，表示要运行的Python脚本的路径。  
  
3. \*\*获取当前Python解释器路径\*\*：  
 ```python  
 python\_path = sys.executable  
 ```  
 - 使用 `sys.executable` 获取当前Python解释器的完整路径，以便在后续命令中使用。  
  
4. \*\*构建运行命令\*\*：  
 ```python  
 command = f'"{python\_path}" -m streamlit run "{script\_path}"'  
 ```  
 - 使用格式化字符串构建命令，命令的形式为：`python -m streamlit run script\_path`，用于运行指定的Streamlit应用。  
  
5. \*\*执行命令\*\*：  
 ```python  
 result = subprocess.run(command, shell=True)  
 ```  
 - 使用 `subprocess.run` 执行构建的命令。`shell=True` 表示在shell中执行命令。  
  
6. \*\*检查命令执行结果\*\*：  
 ```python  
 if result.returncode != 0:  
 print("脚本运行出错。")  
 ```  
 - 检查命令的返回码，如果不为0，表示脚本运行出错，打印错误信息。  
  
7. \*\*主程序入口\*\*：  
 ```python  
 if \_\_name\_\_ == "\_\_main\_\_":  
 ```  
 - 该部分代码在脚本作为主程序运行时执行。  
  
8. \*\*指定脚本路径并运行\*\*：  
 ```python  
 script\_path = "web.py" # 这里可以直接指定脚本路径  
 run\_script(script\_path)  
 ```  
 - 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。代码首先导入了必要的模块，包括 `sys`、`os` 和 `subprocess`，以及一个自定义模块 `QtFusion.path` 中的 `abs\_path` 函数。  
  
在 `run\_script` 函数中，首先获取当前 Python 解释器的路径，存储在 `python\_path` 变量中。接着，构建一个命令字符串，该命令用于运行指定的脚本 `script\_path`，并使用 `streamlit` 进行展示。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。  
  
然后，使用 `subprocess.run` 方法执行构建好的命令。该方法的 `shell=True` 参数允许在 shell 中执行命令。执行完命令后，检查返回的结果码，如果结果码不为零，表示脚本运行出错，程序会输出相应的错误信息。  
  
在文件的最后部分，使用 `if \_\_name\_\_ == "\_\_main\_\_":` 判断当前模块是否为主程序。如果是，则指定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。这里的 `abs\_path` 函数用于获取 `web.py` 的绝对路径，确保脚本能够被正确找到并运行。整体上，这个程序提供了一种简便的方式来启动一个基于 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  
class DetectionTrainer(BaseTrainer):  
 """  
 DetectionTrainer类扩展了BaseTrainer类，用于基于检测模型的训练。  
 """  
  
 def build\_dataset(self, img\_path, mode="train", batch=None):  
 """  
 构建YOLO数据集。  
  
 参数:  
 img\_path (str): 包含图像的文件夹路径。  
 mode (str): 模式，`train`表示训练模式，`val`表示验证模式，用户可以为每种模式自定义不同的增强。  
 batch (int, optional): 批次大小，仅用于`rect`模式。默认为None。  
 """  
 gs = max(int(de\_parallel(self.model).stride.max() if self.model else 0), 32) # 获取模型的最大步幅  
 return build\_yolo\_dataset(self.args, img\_path, batch, self.data, mode=mode, rect=mode == "val", stride=gs)  
  
 def get\_dataloader(self, dataset\_path, batch\_size=16, rank=0, mode="train"):  
 """构造并返回数据加载器。"""  
 assert mode in ["train", "val"] # 确保模式是训练或验证  
 with torch\_distributed\_zero\_first(rank): # 在分布式环境中，确保数据集只初始化一次  
 dataset = self.build\_dataset(dataset\_path, mode, batch\_size) # 构建数据集  
 shuffle = mode == "train" # 训练模式下打乱数据  
 if getattr(dataset, "rect", False) and shuffle:  
 LOGGER.warning("WARNING ⚠️ 'rect=True'与DataLoader的shuffle不兼容，设置shuffle=False")  
 shuffle = False # 如果是rect模式，禁用shuffle  
 workers = self.args.workers if mode == "train" else self.args.workers \* 2 # 设置工作线程数  
 return build\_dataloader(dataset, batch\_size, workers, shuffle, rank) # 返回数据加载器  
  
 def preprocess\_batch(self, batch):  
 """对一批图像进行预处理，包括缩放和转换为浮点数。"""  
 batch["img"] = batch["img"].to(self.device, non\_blocking=True).float() / 255 # 将图像转换为浮点数并归一化  
 if self.args.multi\_scale: # 如果启用多尺度训练  
 imgs = batch["img"]  
 sz = (  
 random.randrange(self.args.imgsz \* 0.5, self.args.imgsz \* 1.5 + self.stride)  
 // self.stride  
 \* self.stride  
 ) # 随机选择新的尺寸  
 sf = sz / max(imgs.shape[2:]) # 计算缩放因子  
 if sf != 1:  
 ns = [  
 math.ceil(x \* sf / self.stride) \* self.stride for x in imgs.shape[2:]  
 ] # 计算新的形状  
 imgs = nn.functional.interpolate(imgs, size=ns, mode="bilinear", align\_corners=False) # 进行插值缩放  
 batch["img"] = imgs # 更新批次中的图像  
 return batch  
  
 def set\_model\_attributes(self):  
 """设置模型的属性，包括类别数量和名称。"""  
 self.model.nc = self.data["nc"] # 将类别数量附加到模型  
 self.model.names = self.data["names"] # 将类别名称附加到模型  
 self.model.args = self.args # 将超参数附加到模型  
  
 def get\_model(self, cfg=None, weights=None, verbose=True):  
 """返回YOLO检测模型。"""  
 model = DetectionModel(cfg, nc=self.data["nc"], verbose=verbose and RANK == -1) # 创建检测模型  
 if weights:  
 model.load(weights) # 加载权重  
 return model  
  
 def get\_validator(self):  
 """返回用于YOLO模型验证的DetectionValidator。"""  
 self.loss\_names = "box\_loss", "cls\_loss", "dfl\_loss" # 定义损失名称  
 return yolo.detect.DetectionValidator(  
 self.test\_loader, save\_dir=self.save\_dir, args=copy(self.args), \_callbacks=self.callbacks  
 )  
  
 def label\_loss\_items(self, loss\_items=None, prefix="train"):  
 """  
 返回带标签的训练损失项字典。  
  
 对于分类不需要，但对于分割和检测是必要的。  
 """  
 keys = [f"{prefix}/{x}" for x in self.loss\_names] # 创建损失项的键  
 if loss\_items is not None:  
 loss\_items = [round(float(x), 5) for x in loss\_items] # 将张量转换为5位小数的浮点数  
 return dict(zip(keys, loss\_items)) # 返回损失项字典  
 else:  
 return keys  
  
 def progress\_string(self):  
 """返回格式化的训练进度字符串，包括轮次、GPU内存、损失、实例和大小。"""  
 return ("\n" + "%11s" \* (4 + len(self.loss\_names))) % (  
 "Epoch",  
 "GPU\_mem",  
 \*self.loss\_names,  
 "Instances",  
 "Size",  
 )  
  
 def plot\_training\_samples(self, batch, ni):  
 """绘制带有注释的训练样本。"""  
 plot\_images(  
 images=batch["img"],  
 batch\_idx=batch["batch\_idx"],  
 cls=batch["cls"].squeeze(-1),  
 bboxes=batch["bboxes"],  
 paths=batch["im\_file"],  
 fname=self.save\_dir / f"train\_batch{ni}.jpg",  
 on\_plot=self.on\_plot,  
 )  
  
 def plot\_metrics(self):  
 """从CSV文件中绘制指标。"""  
 plot\_results(file=self.csv, on\_plot=self.on\_plot) # 保存结果图像  
  
 def plot\_training\_labels(self):  
 """创建YOLO模型的标记训练图。"""  
 boxes = np.concatenate([lb["bboxes"] for lb in self.train\_loader.dataset.labels], 0) # 合并所有边界框  
 cls = np.concatenate([lb["cls"] for lb in self.train\_loader.dataset.labels], 0) # 合并所有类别  
 plot\_labels(boxes, cls.squeeze(), names=self.data["names"], save\_dir=self.save\_dir, on\_plot=self.on\_plot) # 绘制标签  
```  
  
以上代码是YOLO检测模型训练的核心部分，包含了数据集构建、数据加载、图像预处理、模型设置和训练过程中的一些可视化功能。每个方法都有详细的中文注释，帮助理解其功能和作用。```

这个程序文件 `train.py` 是一个用于训练 YOLO（You Only Look Once）目标检测模型的实现，继承自 `BaseTrainer` 类。文件中包含了多个方法，主要用于构建数据集、获取数据加载器、预处理图像、设置模型属性、获取模型、验证模型、记录损失、显示训练进度、绘制训练样本和绘制训练指标等功能。  
  
首先，`DetectionTrainer` 类通过 `build\_dataset` 方法构建 YOLO 数据集。该方法接收图像路径、模式（训练或验证）和批次大小作为参数，并使用 `build\_yolo\_dataset` 函数生成数据集。模式参数允许用户为训练和验证设置不同的数据增强方式。  
  
接下来，`get\_dataloader` 方法用于构建并返回数据加载器。它会根据传入的模式（训练或验证）初始化数据集，并根据模式设置是否打乱数据顺序。该方法还会调整工作线程的数量，以优化数据加载的效率。  
  
在 `preprocess\_batch` 方法中，对每个批次的图像进行预处理，包括将图像缩放到适当的大小并转换为浮点数。该方法支持多尺度训练，随机选择图像的大小，以提高模型的鲁棒性。  
  
`set\_model\_attributes` 方法用于设置模型的属性，包括类别数量和类别名称等，以确保模型能够正确处理特定的数据集。  
  
`get\_model` 方法用于返回一个 YOLO 检测模型，支持加载预训练权重。该方法创建一个 `DetectionModel` 实例，并根据需要加载权重。  
  
`get\_validator` 方法返回一个用于验证 YOLO 模型的验证器。它会初始化损失名称，并创建一个 `DetectionValidator` 实例，用于在验证阶段评估模型性能。  
  
`label\_loss\_items` 方法用于返回带有标签的训练损失项字典，便于记录和分析训练过程中的损失情况。  
  
`progress\_string` 方法生成一个格式化的字符串，显示训练进度，包括当前的 epoch、GPU 内存使用情况、损失值、实例数量和图像大小等信息。  
  
`plot\_training\_samples` 方法用于绘制训练样本及其注释，帮助可视化训练数据的效果。它会将图像、类别和边界框等信息绘制到图像上，并保存为文件。  
  
最后，`plot\_metrics` 和 `plot\_training\_labels` 方法分别用于绘制训练过程中的指标和标签，帮助用户分析模型的训练效果和性能。  
  
整体而言，这个文件提供了一个完整的训练框架，涵盖了从数据加载、预处理到模型训练和验证的各个环节，为用户提供了一个高效的目标检测模型训练工具。

```以下是经过简化并注释的核心代码部分：  
  
```python  
import torch  
  
def adjust\_bboxes\_to\_image\_border(boxes, image\_shape, threshold=20):  
 """  
 调整边界框，使其在距离图像边界一定阈值内时，贴合图像边界。  
  
 参数:  
 boxes (torch.Tensor): 边界框坐标，形状为 (n, 4)  
 image\_shape (tuple): 图像的高度和宽度，形状为 (height, width)  
 threshold (int): 像素阈值  
  
 返回:  
 adjusted\_boxes (torch.Tensor): 调整后的边界框  
 """  
 h, w = image\_shape # 获取图像的高度和宽度  
  
 # 调整边界框坐标  
 boxes[boxes[:, 0] < threshold, 0] = 0 # 如果左上角x坐标小于阈值，则设置为0  
 boxes[boxes[:, 1] < threshold, 1] = 0 # 如果左上角y坐标小于阈值，则设置为0  
 boxes[boxes[:, 2] > w - threshold, 2] = w # 如果右下角x坐标大于图像宽度减去阈值，则设置为图像宽度  
 boxes[boxes[:, 3] > h - threshold, 3] = h # 如果右下角y坐标大于图像高度减去阈值，则设置为图像高度  
 return boxes # 返回调整后的边界框  
  
def bbox\_iou(box1, boxes, iou\_thres=0.9, image\_shape=(640, 640), raw\_output=False):  
 """  
 计算一个边界框与其他边界框的交并比（IoU）。  
  
 参数:  
 box1 (torch.Tensor): 单个边界框坐标，形状为 (4, )  
 boxes (torch.Tensor): 一组边界框坐标，形状为 (n, 4)  
 iou\_thres (float): IoU阈值  
 image\_shape (tuple): 图像的高度和宽度，形状为 (height, width)  
 raw\_output (bool): 如果为True，则返回原始IoU值而不是索引  
  
 返回:  
 high\_iou\_indices (torch.Tensor): IoU大于阈值的边界框索引  
 """  
 boxes = adjust\_bboxes\_to\_image\_border(boxes, image\_shape) # 调整边界框以贴合图像边界  
  
 # 计算交集的坐标  
 x1 = torch.max(box1[0], boxes[:, 0]) # 交集左上角x坐标  
 y1 = torch.max(box1[1], boxes[:, 1]) # 交集左上角y坐标  
 x2 = torch.min(box1[2], boxes[:, 2]) # 交集右下角x坐标  
 y2 = torch.min(box1[3], boxes[:, 3]) # 交集右下角y坐标  
  
 # 计算交集面积  
 intersection = (x2 - x1).clamp(0) \* (y2 - y1).clamp(0) # 使用clamp确保面积不为负  
  
 # 计算各个边界框的面积  
 box1\_area = (box1[2] - box1[0]) \* (box1[3] - box1[1]) # box1的面积  
 box2\_area = (boxes[:, 2] - boxes[:, 0]) \* (boxes[:, 3] - boxes[:, 1]) # 其他边界框的面积  
  
 # 计算并集面积  
 union = box1\_area + box2\_area - intersection # 并集面积  
  
 # 计算IoU  
 iou = intersection / union # 交并比  
 if raw\_output:  
 return 0 if iou.numel() == 0 else iou # 如果需要原始IoU值，直接返回  
  
 # 返回IoU大于阈值的边界框索引  
 return torch.nonzero(iou > iou\_thres).flatten()  
```  
  
### 代码核心部分解释：  
1. \*\*adjust\_bboxes\_to\_image\_border\*\*：该函数用于调整边界框的位置，使其在距离图像边界一定阈值内时，贴合图像边界，确保边界框不会超出图像范围。  
   
2. \*\*bbox\_iou\*\*：该函数计算一个边界框与一组其他边界框的交并比（IoU），用于评估两个边界框的重叠程度。函数首先调用 `adjust\_bboxes\_to\_image\_border` 来确保所有边界框都在图像内，然后计算交集和并集，最后返回满足IoU阈值的边界框索引。```

该程序文件是一个用于YOLO（You Only Look Once）目标检测算法的辅助工具，主要实现了两个功能：调整边界框（bounding boxes）和计算交并比（Intersection over Union, IoU）。  
  
首先，`adjust\_bboxes\_to\_image\_border`函数的作用是将边界框调整到图像边界内，确保边界框不会超出图像的边界。函数接受三个参数：`boxes`是一个包含多个边界框的张量，每个边界框由四个坐标（x1, y1, x2, y2）表示；`image\_shape`是图像的尺寸，包含高度和宽度；`threshold`是一个阈值，用于确定边界框距离图像边缘的最小距离。如果边界框的某个边距图像边缘的距离小于该阈值，则将该边界框的坐标调整为图像边缘的坐标。  
  
接下来，`bbox\_iou`函数用于计算一个边界框与一组其他边界框之间的交并比。该函数接受多个参数：`box1`是待计算的边界框，`boxes`是其他边界框的集合，`iou\_thres`是IoU的阈值，`image\_shape`是图像的尺寸，`raw\_output`是一个布尔值，指示是否返回原始的IoU值。函数首先调用`adjust\_bboxes\_to\_image\_border`来确保所有边界框都在图像内。然后，它计算交集的坐标，并根据这些坐标计算交集的面积、各个边界框的面积以及并集的面积。最后，通过交集面积与并集面积的比值计算IoU。如果`raw\_output`为真，则返回IoU值；否则，返回与IoU大于阈值的边界框的索引。  
  
总的来说，这段代码为YOLO算法提供了重要的边界框处理和评估功能，确保在目标检测任务中，边界框的有效性和准确性。

```以下是代码中最核心的部分，并附上详细的中文注释：  
  
```python  
import os  
import cv2  
import pandas as pd  
import streamlit as st  
from PIL import Image  
from QtFusion.path import abs\_path  
  
def save\_uploaded\_file(uploaded\_file):  
 """  
 保存上传的文件到服务器上。  
  
 Args:  
 uploaded\_file (UploadedFile): 通过Streamlit上传的文件。  
  
 Returns:  
 str: 保存文件的完整路径，如果没有文件上传则返回 None。  
 """  
 # 检查是否有文件上传  
 if uploaded\_file is not None:  
 base\_path = "tempDir" # 定义文件保存的基本路径  
  
 # 如果路径不存在，创建这个路径  
 if not os.path.exists(base\_path):  
 os.makedirs(base\_path)  
 # 获取文件的完整路径  
 file\_path = os.path.join(base\_path, uploaded\_file.name)  
  
 # 以二进制写模式打开文件  
 with open(file\_path, "wb") as f:  
 f.write(uploaded\_file.getbuffer()) # 写入文件  
  
 return file\_path # 返回文件路径  
  
 return None # 如果没有文件上传，返回 None  
  
  
def concat\_results(result, location, confidence, time):  
 """  
 显示检测结果。  
  
 Args:  
 result (str): 检测结果。  
 location (str): 检测位置。  
 confidence (str): 置信度。  
 time (str): 检测用时。  
  
 Returns:  
 DataFrame: 包含检测结果的 DataFrame。  
 """  
 # 创建一个包含这些信息的 DataFrame  
 result\_data = {  
 "识别结果": [result],  
 "位置": [location],  
 "置信度": [confidence],  
 "用时": [time]  
 }  
  
 results\_df = pd.DataFrame(result\_data) # 将结果数据转换为 DataFrame  
 return results\_df # 返回结果 DataFrame  
  
  
def get\_camera\_names():  
 """  
 获取可用摄像头名称列表。  
  
 Returns:  
 list: 返回包含“未启用摄像头”和可用摄像头索引号的列表。  
 """  
 camera\_names = ["摄像头检测关闭", "0"] # 初始化摄像头名称列表  
 max\_test\_cameras = 10 # 定义要测试的最大摄像头数量  
  
 # 循环测试摄像头  
 for i in range(max\_test\_cameras):  
 cap = cv2.VideoCapture(i, cv2.CAP\_DSHOW) # 尝试打开摄像头  
 if cap.isOpened() and str(i) not in camera\_names: # 如果摄像头打开成功且未在列表中  
 camera\_names.append(str(i)) # 添加摄像头索引到列表  
 cap.release() # 释放摄像头资源  
 if len(camera\_names) == 1: # 如果没有找到可用摄像头  
 st.write("未找到可用的摄像头") # 提示信息  
 return camera\_names # 返回摄像头名称列表  
```  
  
### 代码核心部分说明：  
1. \*\*`save\_uploaded\_file` 函数\*\*：负责将用户上传的文件保存到服务器的指定目录，并返回文件的完整路径。  
2. \*\*`concat\_results` 函数\*\*：用于创建一个包含检测结果的 DataFrame，便于后续展示和处理。  
3. \*\*`get\_camera\_names` 函数\*\*：检测系统中可用的摄像头，并返回一个包含摄像头索引的列表，以供用户选择。```

这个程序文件 `code/utils.py` 包含了一些实用的函数，主要用于处理文件上传、显示检测结果、加载默认图片以及获取可用摄像头名称。  
  
首先，`save\_uploaded\_file` 函数用于保存用户通过 Streamlit 上传的文件。它首先检查是否有文件被上传，如果有，则定义一个基本路径 `tempDir` 来保存文件。如果该路径不存在，函数会创建这个路径。接着，它获取上传文件的完整路径，并以二进制写模式打开文件，将文件内容写入到指定路径中。最后，函数返回保存文件的完整路径，如果没有文件上传，则返回 `None`。  
  
接下来，`concat\_results` 函数用于创建一个包含检测结果的 DataFrame。它接受四个参数：检测结果、检测位置、置信度和检测用时，并将这些信息存储在一个字典中，随后使用 Pandas 创建一个 DataFrame 并返回。  
  
`load\_default\_image` 函数的作用是加载一个默认的图片。它通过调用 `abs\_path` 函数获取默认图片的绝对路径，然后使用 PIL 库打开并返回这个图片对象。  
  
最后，`get\_camera\_names` 函数用于获取可用摄像头的名称列表。它初始化一个包含“摄像头检测关闭”和第一个摄像头索引（0）的列表，并定义一个最大测试摄像头数量。函数通过循环尝试打开多个摄像头，如果成功打开，则将摄像头的索引添加到列表中。最后，如果没有找到可用的摄像头，函数会在 Streamlit 界面上显示一条消息，并返回摄像头名称列表。  
  
总体而言，这个文件提供了一些基本的工具函数，方便在 Streamlit 应用中处理文件和摄像头相关的操作。

### 整体功能和构架概括  
  
该项目是一个基于YOLOv8算法的目标检测框架，包含多个模块和工具，旨在提供一个完整的训练、验证和推理流程。项目的主要功能包括模型的构建、训练、验证、结果可视化、文件处理以及用户界面交互。通过整合不同的功能模块，用户可以方便地进行目标检测任务的开发和调试。  
  
项目的架构分为以下几个主要部分：  
  
1. \*\*模型构建与训练\*\*：包括YOLO模型的定义、训练过程的实现以及数据集的处理。  
2. \*\*工具函数\*\*：提供了一些实用的工具函数，用于文件处理、边界框调整、IoU计算等。  
3. \*\*用户界面\*\*：使用Streamlit库构建用户界面，方便用户上传文件、查看检测结果等。  
4. \*\*验证与评估\*\*：提供模型验证和性能评估的功能，帮助用户了解模型的表现。  
  
### 文件功能整理表  
  
| 文件路径 | 功能描述 |  
|--------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------|  
| `C:\shangjia\code\70+种YOLOv8算法改进源码大全和调试加载训练教程（非必要）\ultralytics\nn\backbone\convnextv2.py` | 实现ConvNeXt V2模型的各个组件和结构，主要用于图像分类任务。 |  
| `C:\shangjia\code\70+种YOLOv8算法改进源码大全和调试加载训练教程（非必要）\ultralytics\models\yolo\detect\\_\_init\_\_.py` | 组织YOLO模型的训练、预测和验证功能，包含重要的类和方法。 |  
| `C:\shangjia\code\ui.py` | 通过Streamlit运行指定的脚本，提供用户界面交互功能。 |  
| `C:\shangjia\code\70+种YOLOv8算法改进源码大全和调试加载训练教程（非必要）\ultralytics\data\utils.py` | 提供数据处理和增强的工具函数，支持数据集的构建和加载。 |  
| `C:\shangjia\code\train.py` | 实现YOLO模型的训练过程，包括数据加载、预处理、模型训练和验证等功能。 |  
| `C:\shangjia\code\70+种YOLOv8算法改进源码大全和调试加载训练教程（非必要）\ultralytics\models\fastsam\utils.py` | 提供边界框调整和IoU计算的辅助函数，确保目标检测的准确性。 |  
| `C:\shangjia\code\code\utils.py` | 提供文件上传、检测结果显示、默认图片加载和摄像头名称获取等实用函数。 |  
| `C:\shangjia\code\ultralytics\solutions\distance\_calculation.py` | 实现距离计算的功能，可能用于目标检测中的距离估计。 |  
| `C:\shangjia\code\70+种YOLOv8算法改进源码大全和调试加载训练教程（非必要）\ultralytics\utils\benchmarks.py` | 提供基准测试功能，用于评估模型的性能和效率。 |  
| `C:\shangjia\code\code\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表现。 |  
| `C:\shangjia\code\code\ui.py` | 通过Streamlit提供用户界面，支持文件上传和结果展示。 |  
| `C:\shangjia\code\code\ultralytics\models\yolo\\_\_init\_\_.py` | 初始化YOLO模型模块，组织相关的类和函数。 |  
| `C:\shangjia\code\code\ultralytics\utils\plotting.py` | 提供可视化功能，用于绘制训练过程中的指标和样本，帮助分析模型性能。 |  
  
这个表格概述了项目中各个文件的主要功能，帮助用户快速理解每个模块的作用和相互关系。

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“70+种创新点源码”以“13.完整训练+Web前端界面+70+种创新点源码、数据集获取”的内容为准。