# 改进yolo11-fasternet等200+全套创新点大全：停车场柱子检测系统源码＆数据集全套

## 3.背景

研究背景与意义  
  
随着城市化进程的加快，停车场的管理与监控变得愈发重要。停车场不仅是车辆停放的场所，更是城市交通管理的关键节点。传统的停车场管理方式往往依赖人工巡查，效率低下且容易出错。为了解决这一问题，基于计算机视觉的自动化检测系统应运而生，能够实时监控停车场内的车辆、行人及其他物体，提升管理效率和安全性。  
  
在众多计算机视觉技术中，YOLO（You Only Look Once）系列模型因其高效的实时检测能力而备受关注。YOLOv11作为该系列的最新版本，结合了深度学习的先进技术，能够在复杂场景中快速、准确地识别和定位目标物体。然而，停车场环境的复杂性，如光照变化、视角不同及遮挡现象，给物体检测带来了挑战。因此，改进YOLOv11以适应停车场柱子及车辆的检测，具有重要的实际意义。  
  
本研究所使用的数据集包含1300幅图像，涵盖了四个类别：车辆、物体、行人和柱子。这些类别的选择不仅反映了停车场环境的多样性，也为模型的训练提供了丰富的样本。通过对这些数据的分析与处理，可以有效提升YOLOv11在停车场场景中的检测精度和鲁棒性。此外，针对停车场柱子的特定检测需求，改进后的模型将能够更好地识别和定位柱子，减少因柱子遮挡导致的误检和漏检现象。  
  
综上所述，基于改进YOLOv11的停车场柱子检测系统，不仅能够提升停车场的管理效率，还能为城市交通管理提供重要的数据支持，具有广泛的应用前景和社会价值。

## 2.视频效果

## 3. 图片效果展示

注意：本项目提供训练的数据集和训练教程,由于版本持续更新,暂不提供权重文件（best.pt）,请按照6.训练教程进行训练后实现上图演示的效果。

## 4. 数据集信息

本项目数据集信息介绍  
  
本项目旨在改进YOLOv11的停车场柱子检测系统，因此构建了一个专门针对“pillar\_and\_car”主题的数据集。该数据集包含四个主要类别，分别为“car”（汽车）、“object”（物体）、“person”（行人）和“pillar”（柱子），这些类别的选择是基于停车场环境中常见的元素，以确保模型能够在实际应用中高效识别和分类。数据集的类别数量为四，涵盖了与停车场相关的关键对象，能够有效支持柱子检测任务的训练和评估。  
  
在数据集的构建过程中，我们收集了大量的图像，确保每个类别的样本数量均衡，覆盖不同的场景和光照条件。这些图像不仅包括停车场的全景视图，还包括特写镜头，以便模型能够学习到柱子与其他对象之间的相对位置关系。此外，为了提高模型的鲁棒性，数据集中还包含了不同天气条件下的图像，如晴天、阴天和雨天等，确保模型在各种环境下都能保持良好的检测性能。  
  
每个图像都经过精确标注，确保柱子、汽车、行人及其他物体的边界框清晰可见。这样的标注方式不仅有助于模型学习目标的特征，还能提高其在复杂场景中的识别能力。通过对这些多样化数据的训练，期望能够显著提升YOLOv11在停车场柱子检测任务中的准确性和效率，为智能停车系统的进一步发展奠定坚实的基础。整体而言，本数据集的设计和构建充分考虑了实际应用需求，力求为模型提供丰富的学习素材，以实现更高的检测性能。

## 5. 全套项目环境部署教程（零基础手把手教学）

5.1 所需软件PyCharm和Anaconda安装教程（第一步）

https://www.bilibili.com/video/BV1BoC1YCEKi/?spm\_id\_from=333.999.0.0&vd\_source=bc9aec86d164b67a7004b996143742dc

5.2 安装Python虚拟环境创建和依赖库安装视频教程（第二步）

https://www.bilibili.com/video/BV1ZoC1YCEBw?spm\_id\_from=333.788.videopod.sections&vd\_source=bc9aec86d164b67a7004b996143742dc

## 6. 改进YOLOv11训练教程和Web\_UI前端加载模型视频教程（零基础手把手教学）

https://www.bilibili.com/video/BV1BoC1YCEhR?spm\_id\_from=333.788.videopod.sections&vd\_source=bc9aec86d164b67a7004b996143742dc

## 7.原始YOLOv11算法讲解

## 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全部展开，具体见下列网址中的改进模块对应项目的技术原理博客网址【Blog】（创新点均为模块化搭建，原理适配YOLOv5~YOLOv11等各种版本）

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. YOLOv11核心改进模块源码讲解

```以下是对给定代码的核心部分进行提炼和详细注释的结果：  
  
```python  
import torch  
import torch.nn as nn  
import math  
import itertools  
  
class Attention4D(nn.Module):  
 """   
 4D注意力机制模块  
 """  
 def \_\_init\_\_(self, dim=384, key\_dim=32, num\_heads=8, attn\_ratio=4, resolution=7, act\_layer=nn.ReLU, stride=None):  
 super().\_\_init\_\_()  
 self.num\_heads = num\_heads # 注意力头的数量  
 self.scale = key\_dim \*\* -0.5 # 缩放因子  
 self.key\_dim = key\_dim # 键的维度  
 self.resolution = resolution if stride is None else math.ceil(resolution / stride) # 解析度  
 self.stride\_conv = nn.Conv2d(dim, dim, kernel\_size=3, stride=stride, padding=1, groups=dim) if stride else None # 步幅卷积  
 self.upsample = nn.Upsample(scale\_factor=stride, mode='bilinear') if stride else None # 上采样  
  
 # 初始化查询、键、值的卷积层  
 self.q = nn.Conv2d(dim, num\_heads \* key\_dim, 1)  
 self.k = nn.Conv2d(dim, num\_heads \* key\_dim, 1)  
 self.v = nn.Conv2d(dim, num\_heads \* attn\_ratio \* key\_dim, 1)  
  
 # 注意力偏置参数  
 self.attention\_biases = nn.Parameter(torch.zeros(num\_heads, len(self.\_get\_attention\_offsets(resolution))))  
  
 def forward(self, x):  
 """ 前向传播 """  
 B, C, H, W = x.shape # B: 批量大小, C: 通道数, H: 高度, W: 宽度  
 if self.stride\_conv is not None:  
 x = self.stride\_conv(x) # 应用步幅卷积  
  
 # 计算查询、键、值  
 q = self.q(x).flatten(2).permute(0, 1, 3, 2) # (B, num\_heads, N, key\_dim)  
 k = self.k(x).flatten(2).permute(0, 1, 2, 3) # (B, num\_heads, N, key\_dim)  
 v = self.v(x).flatten(2).permute(0, 1, 3, 2) # (B, num\_heads, N, attn\_dim)  
  
 # 计算注意力  
 attn = (q @ k) \* self.scale + self.attention\_biases  
 attn = attn.softmax(dim=-1) # softmax归一化  
  
 # 计算输出  
 x = (attn @ v).transpose(2, 3).reshape(B, -1, self.resolution, self.resolution)  
 if self.upsample is not None:  
 x = self.upsample(x) # 上采样  
 return x  
  
 def \_get\_attention\_offsets(self, resolution):  
 """ 计算注意力偏置的偏移量 """  
 points = list(itertools.product(range(resolution), repeat=2))  
 attention\_offsets = {}  
 idxs = []  
 for p1 in points:  
 for p2 in points:  
 offset = (abs(p1[0] - p2[0]), abs(p1[1] - p2[1]))  
 if offset not in attention\_offsets:  
 attention\_offsets[offset] = len(attention\_offsets)  
 idxs.append(attention\_offsets[offset])  
 return idxs  
  
class EfficientFormerV2(nn.Module):  
 """  
 EfficientFormer V2 模型  
 """  
 def \_\_init\_\_(self, layers, embed\_dims, num\_classes=1000):  
 super().\_\_init\_\_()  
 self.patch\_embed = nn.Conv2d(3, embed\_dims[0], kernel\_size=3, stride=2, padding=1) # 初始嵌入层  
 self.network = nn.ModuleList() # 存储网络的各个层  
  
 for i in range(len(layers)):  
 # 构建每个阶段的块  
 stage = self.\_build\_stage(embed\_dims[i], layers[i])  
 self.network.append(stage)  
  
 def \_build\_stage(self, dim, layers):  
 """ 构建网络的每个阶段 """  
 blocks = []  
 for \_ in range(layers):  
 blocks.append(Attention4D(dim)) # 添加注意力模块  
 return nn.Sequential(\*blocks)  
  
 def forward(self, x):  
 """ 前向传播 """  
 x = self.patch\_embed(x) # 通过嵌入层  
 for block in self.network:  
 x = block(x) # 通过每个块  
 return x  
  
def efficientformerv2\_s0(weights='', \*\*kwargs):  
 """ 创建 S0 版本的 EfficientFormerV2 """  
 model = EfficientFormerV2(layers=[2, 2, 6, 4], embed\_dims=[32, 48, 96, 176], \*\*kwargs)  
 if weights:  
 model.load\_state\_dict(torch.load(weights))  
 return model  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 inputs = torch.randn((1, 3, 640, 640)) # 随机输入  
 model = efficientformerv2\_s0() # 创建模型  
 res = model(inputs) # 前向传播  
 print(res.size()) # 输出结果的尺寸  
```  
  
### 代码分析和注释说明：  
  
1. \*\*Attention4D 类\*\*：实现了一个4D注意力机制模块。该模块包括查询、键、值的卷积层，并通过计算注意力权重来生成输出。注意力权重是通过对查询和键的点积进行缩放和softmax计算得到的。  
  
2. \*\*EfficientFormerV2 类\*\*：这是整个模型的主类，负责构建网络结构。它通过嵌入层将输入图像转换为特征表示，并通过多个注意力模块进行处理。  
  
3. \*\*efficientformerv2\_s0 函数\*\*：这是一个工厂函数，用于创建特定配置的EfficientFormerV2模型（S0版本），并可选择加载预训练权重。  
  
4. \*\*主程序\*\*：在主程序中，创建了一个随机输入并通过模型进行前向传播，输出结果的尺寸。  
  
这些核心部分构成了EfficientFormerV2模型的基础结构，注意力机制是其关键特性之一。```

这个文件实现了一个名为EfficientFormerV2的深度学习模型，主要用于图像处理任务。该模型的设计灵感来源于高效的Transformer架构，结合了卷积神经网络（CNN）的优点，以提高计算效率和性能。  
  
首先，文件中定义了一些超参数和结构体，这些参数包括不同模型版本的宽度和深度配置。EfficientFormer的不同版本（如S0、S1、S2和L）具有不同的层数和通道数，适应不同的计算资源和应用需求。  
  
接下来，文件中定义了多个类来构建模型的不同组件。Attention4D类实现了一个四维注意力机制，支持多头注意力，并可以通过卷积操作进行下采样。这个类的构造函数中定义了多个卷积层和归一化层，用于计算查询、键和值，并生成注意力权重。  
  
LGQuery类用于生成局部查询，通过平均池化和卷积操作来处理输入特征图。Attention4DDownsample类则结合了注意力机制和下采样操作，以减少特征图的空间维度，同时保留重要信息。  
  
Embedding类负责将输入图像转换为嵌入表示，可以选择不同的实现方式，包括轻量级卷积和注意力机制。Mlp类实现了一个多层感知机，使用1x1卷积来处理特征。  
  
AttnFFN和FFN类分别实现了带有注意力机制的前馈网络和普通前馈网络，后者在输入上应用多层感知机。eformer\_block函数用于构建EfficientFormer的基本模块，结合了注意力和前馈网络的功能。  
  
EfficientFormerV2类是模型的核心，负责构建整个网络结构。它接受多个参数，包括层数、嵌入维度、下采样策略等。该类还实现了前向传播方法，处理输入数据并返回特征图。  
  
最后，文件提供了几个函数（如efficientformerv2\_s0、efficientformerv2\_s1等）用于创建不同版本的EfficientFormerV2模型，并加载预训练权重。文件的最后部分包含了一个示例，展示了如何使用这些模型进行推理，并打印输出特征图的尺寸。  
  
总体而言，这个文件实现了一个灵活且高效的图像处理模型，适用于各种计算资源和任务需求，充分利用了现代深度学习中的注意力机制和卷积操作。

```以下是代码中最核心的部分，并附上详细的中文注释：  
  
```python  
import torch  
import torch.nn as nn  
import torch.nn.functional as F  
import numpy as np  
  
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 # 输出特征数默认为输入特征数  
 hidden\_features = hidden\_features or in\_features # 隐藏层特征数默认为输入特征数  
 self.fc1 = nn.Linear(in\_features, hidden\_features) # 第一层线性变换  
 self.act = act\_layer() # 激活函数  
 self.fc2 = nn.Linear(hidden\_features, out\_features) # 第二层线性变换  
 self.drop = nn.Dropout(drop) # Dropout层  
  
 def forward(self, x):  
 """前向传播"""  
 x = self.fc1(x) # 线性变换  
 x = self.act(x) # 激活  
 x = self.drop(x) # Dropout  
 x = self.fc2(x) # 线性变换  
 x = self.drop(x) # Dropout  
 return x  
  
class LePEAttention(nn.Module):  
 """带有局部增强位置编码的注意力机制"""  
 def \_\_init\_\_(self, dim, resolution, idx, split\_size=7, num\_heads=8, attn\_drop=0.):  
 super().\_\_init\_\_()  
 self.dim = dim # 输入特征维度  
 self.resolution = resolution # 输入分辨率  
 self.split\_size = split\_size # 切分窗口的大小  
 self.num\_heads = num\_heads # 注意力头的数量  
 head\_dim = dim // num\_heads # 每个头的维度  
 self.scale = head\_dim \*\* -0.5 # 缩放因子  
 self.get\_v = nn.Conv2d(dim, dim, kernel\_size=3, stride=1, padding=1, groups=dim) # 卷积层用于获取v  
  
 self.attn\_drop = nn.Dropout(attn\_drop) # Dropout层  
  
 def im2cswin(self, x):  
 """将输入图像转换为窗口格式"""  
 B, N, C = x.shape # B: 批量大小, N: 序列长度, C: 特征维度  
 H = W = int(np.sqrt(N)) # 计算图像的高度和宽度  
 x = x.transpose(-2, -1).contiguous().view(B, C, H, W) # 转换为 (B, C, H, W) 格式  
 x = img2windows(x, self.split\_size, self.split\_size) # 切分图像为窗口  
 return x  
  
 def forward(self, qkv):  
 """前向传播"""  
 q, k, v = qkv[0], qkv[1], qkv[2] # 分别获取查询、键、值  
 q = self.im2cswin(q) # 将查询转换为窗口格式  
 k = self.im2cswin(k) # 将键转换为窗口格式  
 v = self.get\_v(v) # 通过卷积获取值  
  
 q = q \* self.scale # 应用缩放因子  
 attn = (q @ k.transpose(-2, -1)) # 计算注意力权重  
 attn = nn.functional.softmax(attn, dim=-1) # 归一化  
 attn = self.attn\_drop(attn) # 应用Dropout  
  
 x = (attn @ v) # 计算输出  
 return x  
  
class CSWinBlock(nn.Module):  
 """CSWin Transformer的基本块"""  
 def \_\_init\_\_(self, dim, num\_heads, split\_size=7, mlp\_ratio=4.):  
 super().\_\_init\_\_()  
 self.dim = dim # 输入特征维度  
 self.num\_heads = num\_heads # 注意力头的数量  
 self.qkv = nn.Linear(dim, dim \* 3) # 线性层用于生成q, k, v  
 self.attn = LePEAttention(dim, resolution=split\_size, idx=0, num\_heads=num\_heads) # 注意力层  
 self.mlp = Mlp(in\_features=dim, hidden\_features=int(dim \* mlp\_ratio), out\_features=dim) # MLP层  
  
 def forward(self, x):  
 """前向传播"""  
 qkv = self.qkv(x).reshape(x.shape[0], -1, 3, self.dim).permute(2, 0, 1, 3) # 生成q, k, v  
 x = self.attn(qkv) # 通过注意力层  
 x = x + self.mlp(x) # 加上MLP的输出  
 return x  
  
class CSWinTransformer(nn.Module):  
 """CSWin Transformer模型"""  
 def \_\_init\_\_(self, img\_size=640, in\_chans=3, num\_classes=1000, embed\_dim=96, depth=[2, 2, 6, 2]):  
 super().\_\_init\_\_()  
 self.num\_classes = num\_classes  
 self.embed\_dim = embed\_dim # 嵌入维度  
  
 # 输入层：卷积嵌入  
 self.stage1\_conv\_embed = nn.Sequential(  
 nn.Conv2d(in\_chans, embed\_dim, kernel\_size=7, stride=4, padding=2),  
 nn.LayerNorm(embed\_dim)  
 )  
  
 # Transformer的多个阶段  
 self.stage1 = nn.ModuleList([CSWinBlock(embed\_dim, num\_heads=8) for \_ in range(depth[0])])  
 self.stage2 = nn.ModuleList([CSWinBlock(embed\_dim \* 2, num\_heads=16) for \_ in range(depth[1])])  
 self.stage3 = nn.ModuleList([CSWinBlock(embed\_dim \* 4, num\_heads=32) for \_ in range(depth[2])])  
 self.stage4 = nn.ModuleList([CSWinBlock(embed\_dim \* 8, num\_heads=64) for \_ in range(depth[3])])  
  
 def forward(self, x):  
 """前向传播"""  
 x = self.stage1\_conv\_embed(x) # 通过卷积嵌入  
 for blk in self.stage1:  
 x = blk(x) # 通过每个Transformer块  
 for blk in self.stage2:  
 x = blk(x)  
 for blk in self.stage3:  
 x = blk(x)  
 for blk in self.stage4:  
 x = blk(x)  
 return x  
  
# 示例代码，创建模型并进行前向传播  
if \_\_name\_\_ == '\_\_main\_\_':  
 inputs = torch.randn((1, 3, 640, 640)) # 随机输入  
 model = CSWinTransformer() # 创建模型  
 res = model(inputs) # 前向传播  
 print(res.size()) # 输出结果的尺寸  
```  
  
### 代码说明：  
1. \*\*Mlp类\*\*：实现了一个多层感知机，包含两个线性层和一个激活函数，支持Dropout。  
2. \*\*LePEAttention类\*\*：实现了带有局部增强位置编码的注意力机制，包含输入转换、注意力计算等功能。  
3. \*\*CSWinBlock类\*\*：定义了CSWin Transformer的基本块，包含了注意力层和MLP层。  
4. \*\*CSWinTransformer类\*\*：构建了整个CSWin Transformer模型，包含多个阶段的Transformer块，并进行前向传播。  
  
这个代码的核心部分实现了CSWin Transformer的基本结构和功能，适用于图像分类等任务。```

这个程序文件实现了一个名为CSWin Transformer的视觉变换器模型，主要用于图像分类等计算机视觉任务。该模型的设计灵感来源于Transformer架构，结合了窗口注意力机制和卷积操作，以提高模型的性能和效率。  
  
首先，文件中导入了必要的库，包括PyTorch、timm库和一些功能性模块。接着，定义了一些基本的类和函数。  
  
Mlp类实现了一个多层感知机（MLP），包含两个线性层和一个激活函数（默认为GELU），用于在特征提取阶段进行非线性变换。LePEAttention类实现了一个局部增强的注意力机制，支持对输入进行分块处理，并通过卷积操作提取局部特征。该类中的方法包括将输入图像转换为窗口格式、计算注意力权重、以及将注意力结果与局部增强特征结合。  
  
CSWinBlock类是CSWin Transformer的基本构建块，包含了注意力机制和MLP。该类在初始化时会根据输入的参数设置注意力头的数量、分块大小等。forward方法实现了前向传播，首先对输入进行归一化，然后计算注意力，并通过MLP进行特征变换。  
  
接下来，定义了img2windows和windows2img两个辅助函数，用于将图像从空间域转换为窗口域，反之亦然。Merge\_Block类用于在不同阶段之间进行特征合并，通常是通过卷积操作来减小特征图的尺寸。  
  
CSWinTransformer类是整个模型的核心，包含多个阶段，每个阶段由多个CSWinBlock和Merge\_Block组成。该类的初始化方法设置了输入图像的大小、嵌入维度、深度、注意力头的数量等超参数。forward\_features方法负责通过各个阶段进行特征提取，并返回提取的特征。  
  
文件还定义了一些用于加载预训练权重的函数，如update\_weight和\_conv\_filter。最后，提供了几个函数（CSWin\_tiny、CSWin\_small、CSWin\_base、CSWin\_large）用于创建不同规模的CSWin Transformer模型，并可选择加载预训练权重。  
  
在文件的最后部分，提供了一个示例，展示了如何创建不同规模的模型并对随机生成的输入进行前向传播，输出各个阶段的特征图的尺寸。这部分代码可以用于测试模型的构建和前向传播是否正常工作。  
  
总体而言，这个程序文件实现了一个高效的视觉变换器模型，结合了多种先进的技术，适用于图像分类等任务。

```以下是代码中最核心的部分，并附上详细的中文注释：  
  
```python  
class DetectionValidator(BaseValidator):  
 """  
 扩展自 BaseValidator 类的检测模型验证器类。  
 """  
  
 def \_\_init\_\_(self, dataloader=None, save\_dir=None, pbar=None, args=None, \_callbacks=None):  
 """初始化检测模型，设置必要的变量和参数。"""  
 super().\_\_init\_\_(dataloader, save\_dir, pbar, args, \_callbacks)  
 self.metrics = DetMetrics(save\_dir=self.save\_dir, on\_plot=self.on\_plot) # 初始化检测指标  
 self.iouv = torch.linspace(0.5, 0.95, 10) # 定义 IoU 向量，用于计算 mAP@0.5:0.95  
 self.niou = self.iouv.numel() # IoU 的数量  
  
 def preprocess(self, batch):  
 """对 YOLO 训练的图像批次进行预处理。"""  
 batch["img"] = batch["img"].to(self.device, non\_blocking=True) # 将图像转移到设备上  
 batch["img"] = (batch["img"].half() if self.args.half else batch["img"].float()) / 255 # 归一化图像  
 for k in ["batch\_idx", "cls", "bboxes"]:  
 batch[k] = batch[k].to(self.device) # 将其他数据转移到设备上  
  
 return batch # 返回处理后的批次  
  
 def postprocess(self, preds):  
 """对预测输出应用非极大值抑制（NMS）。"""  
 return ops.non\_max\_suppression(  
 preds,  
 self.args.conf,  
 self.args.iou,  
 multi\_label=True,  
 agnostic=self.args.single\_cls,  
 max\_det=self.args.max\_det,  
 ) # 返回经过 NMS 处理的预测结果  
  
 def update\_metrics(self, preds, batch):  
 """更新指标统计信息。"""  
 for si, pred in enumerate(preds):  
 self.seen += 1 # 记录已处理的样本数量  
 pbatch = self.\_prepare\_batch(si, batch) # 准备当前批次的数据  
 cls, bbox = pbatch.pop("cls"), pbatch.pop("bbox") # 获取当前批次的类别和边界框  
  
 if len(pred) == 0: # 如果没有预测结果  
 continue  
  
 predn = self.\_prepare\_pred(pred, pbatch) # 准备预测结果  
 stat = {  
 "conf": predn[:, 4], # 置信度  
 "pred\_cls": predn[:, 5], # 预测类别  
 "tp": self.\_process\_batch(predn, bbox, cls) # 计算真正例  
 }  
 for k in self.stats.keys():  
 self.stats[k].append(stat[k]) # 更新统计信息  
  
 def get\_stats(self):  
 """返回指标统计信息和结果字典。"""  
 stats = {k: torch.cat(v, 0).cpu().numpy() for k, v in self.stats.items()} # 转换为 numpy 数组  
 if len(stats) and stats["tp"].any():  
 self.metrics.process(\*\*stats) # 处理指标  
 return self.metrics.results\_dict # 返回结果字典  
  
 def print\_results(self):  
 """打印每个类别的训练/验证集指标。"""  
 pf = "%22s" + "%11i" \* 2 + "%11.3g" \* len(self.metrics.keys) # 打印格式  
 LOGGER.info(pf % ("all", self.seen, self.nt\_per\_class.sum(), \*self.metrics.mean\_results())) # 打印总体结果  
  
 # 打印每个类别的结果  
 if self.args.verbose and not self.training and self.nc > 1 and len(self.stats):  
 for i, c in enumerate(self.metrics.ap\_class\_index):  
 LOGGER.info(pf % (self.names[c], self.seen, self.nt\_per\_class[c], \*self.metrics.class\_result(i)))  
  
 def build\_dataset(self, img\_path, mode="val", batch=None):  
 """  
 构建 YOLO 数据集。  
  
 Args:  
 img\_path (str): 图像文件夹路径。  
 mode (str): 模式（`train` 或 `val`）。  
 batch (int, optional): 批次大小，默认为 None。  
 """  
 return build\_yolo\_dataset(self.args, img\_path, batch, self.data, mode=mode, stride=self.stride) # 返回构建的数据集  
  
 def get\_dataloader(self, dataset\_path, batch\_size):  
 """构建并返回数据加载器。"""  
 dataset = self.build\_dataset(dataset\_path, batch=batch\_size, mode="val") # 构建数据集  
 return build\_dataloader(dataset, batch\_size, self.args.workers, shuffle=False, rank=-1) # 返回数据加载器  
```  
  
### 主要功能概述：  
1. \*\*初始化和预处理\*\*：`\_\_init\_\_` 和 `preprocess` 方法负责初始化模型参数和对输入数据进行预处理。  
2. \*\*后处理和指标更新\*\*：`postprocess` 和 `update\_metrics` 方法用于对模型的预测结果进行后处理（如 NMS）并更新评估指标。  
3. \*\*统计和结果打印\*\*：`get\_stats` 和 `print\_results` 方法用于获取统计信息并打印每个类别的结果。  
4. \*\*数据集和数据加载器构建\*\*：`build\_dataset` 和 `get\_dataloader` 方法用于构建数据集和数据加载器，以便进行验证。  
  
这些核心部分是 YOLO 检测模型验证的基础，涵盖了数据处理、模型评估和结果输出等关键环节。```

这个程序文件 `val.py` 是一个用于YOLO（You Only Look Once）目标检测模型验证的实现，继承自 `BaseValidator` 类。该文件主要用于对目标检测模型的性能进行评估，包括计算各种指标和生成可视化结果。  
  
首先，程序导入了一些必要的库和模块，包括 `os`、`numpy`、`torch` 以及 `ultralytics` 包中的多个组件。这些组件用于数据处理、模型验证、日志记录和绘图等功能。  
  
`DetectionValidator` 类是该文件的核心，负责处理验证过程。它的构造函数初始化了一些必要的变量和设置，例如数据加载器、保存目录、进度条、参数和回调函数。它还定义了一些用于计算检测指标的变量，比如 `DetMetrics` 用于存储和计算检测指标，`iou` 用于计算不同阈值下的交并比（IoU）。  
  
在 `preprocess` 方法中，程序对输入的图像批次进行预处理，包括将图像转换为合适的格式和范围，并将标签数据移动到指定的设备上（如GPU）。如果设置了保存混合数据的选项，程序还会准备用于自动标注的数据。  
  
`init\_metrics` 方法用于初始化评估指标，包括检查数据集是否为COCO格式，并根据模型的类别名称设置相应的参数。`get\_desc` 方法返回一个格式化的字符串，用于总结每个类别的指标。  
  
在 `postprocess` 方法中，程序对模型的预测结果应用非极大值抑制（NMS），以去除冗余的检测框。`\_prepare\_batch` 和 `\_prepare\_pred` 方法分别用于准备输入批次和预测结果，以便进行后续的评估。  
  
`update\_metrics` 方法负责更新检测指标，处理每个批次的预测结果，并与真实标签进行比较，计算真阳性、假阳性等统计信息。`finalize\_metrics` 方法用于设置最终的指标值。  
  
`get\_stats` 方法返回计算后的指标统计信息，而 `print\_results` 方法则负责打印训练或验证集的每个类别的指标结果。如果启用了绘图选项，程序还会绘制混淆矩阵。  
  
`\_process\_batch` 方法用于返回正确的预测矩阵，通过计算IoU来匹配预测框和真实框。`build\_dataset` 和 `get\_dataloader` 方法用于构建YOLO数据集和返回数据加载器。  
  
程序还包含了一些可视化功能，例如 `plot\_val\_samples` 和 `plot\_predictions` 方法，用于绘制验证图像样本和预测结果。`save\_one\_txt` 和 `pred\_to\_json` 方法用于将检测结果保存为文本文件或COCO格式的JSON文件。  
  
最后，`eval\_json` 方法用于评估YOLO输出的JSON格式，并返回性能统计信息，利用pycocotools库计算mAP（平均精度均值）等指标。  
  
总体而言，这个文件实现了YOLO模型验证的完整流程，包括数据预处理、指标计算、结果保存和可视化等功能，适用于目标检测任务的评估。

```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  
from timm.models.layers import DropPath, to\_2tuple, trunc\_normal\_  
  
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 # 输出特征数默认为输入特征数  
 hidden\_features = hidden\_features or in\_features # 隐藏层特征数默认为输入特征数  
 self.fc1 = nn.Linear(in\_features, hidden\_features) # 第一层线性变换  
 self.act = act\_layer() # 激活函数  
 self.fc2 = nn.Linear(hidden\_features, out\_features) # 第二层线性变换  
 self.drop = nn.Dropout(drop) # Dropout层  
  
 def forward(self, x):  
 """ 前向传播 """  
 x = self.fc1(x) # 线性变换  
 x = self.act(x) # 激活  
 x = self.drop(x) # Dropout  
 x = self.fc2(x) # 线性变换  
 x = self.drop(x) # Dropout  
 return x  
  
  
class WindowAttention(nn.Module):  
 """ 基于窗口的多头自注意力 (W-MSA) 模块 """  
  
 def \_\_init\_\_(self, dim, window\_size, num\_heads, qkv\_bias=True, qk\_scale=None, attn\_drop=0., proj\_drop=0.):  
 super().\_\_init\_\_()  
 self.dim = dim # 输入通道数  
 self.window\_size = window\_size # 窗口大小  
 self.num\_heads = num\_heads # 注意力头数  
 head\_dim = dim // num\_heads # 每个头的维度  
 self.scale = qk\_scale or head\_dim \*\* -0.5 # 缩放因子  
  
 # 定义相对位置偏置参数表  
 self.relative\_position\_bias\_table = nn.Parameter(  
 torch.zeros((2 \* window\_size[0] - 1) \* (2 \* window\_size[1] - 1), num\_heads)) # 位置偏置表  
  
 # 计算每个token的相对位置索引  
 coords\_h = torch.arange(self.window\_size[0])  
 coords\_w = torch.arange(self.window\_size[1])  
 coords = torch.stack(torch.meshgrid([coords\_h, coords\_w])) # 生成网格坐标  
 coords\_flatten = torch.flatten(coords, 1) # 展平坐标  
 relative\_coords = coords\_flatten[:, :, None] - coords\_flatten[:, None, :] # 计算相对坐标  
 relative\_coords = relative\_coords.permute(1, 2, 0).contiguous() # 调整维度  
 relative\_coords[:, :, 0] += self.window\_size[0] - 1 # 偏移  
 relative\_coords[:, :, 1] += self.window\_size[1] - 1  
 relative\_coords[:, :, 0] \*= 2 \* self.window\_size[1] - 1  
 relative\_position\_index = relative\_coords.sum(-1) # 计算相对位置索引  
 self.register\_buffer("relative\_position\_index", relative\_position\_index) # 注册为缓冲区  
  
 self.qkv = nn.Linear(dim, dim \* 3, bias=qkv\_bias) # 线性变换生成Q, K, V  
 self.attn\_drop = nn.Dropout(attn\_drop) # 注意力的Dropout  
 self.proj = nn.Linear(dim, dim) # 输出线性变换  
 self.proj\_drop = nn.Dropout(proj\_drop) # 输出的Dropout  
 trunc\_normal\_(self.relative\_position\_bias\_table, std=.02) # 初始化相对位置偏置  
 self.softmax = nn.Softmax(dim=-1) # Softmax层  
  
 def forward(self, x, mask=None):  
 """ 前向传播 """  
 B\_, N, C = x.shape # B: 批量大小, N: 序列长度, C: 通道数  
 qkv = self.qkv(x).reshape(B\_, N, 3, self.num\_heads, C // self.num\_heads).permute(2, 0, 3, 1, 4) # 计算Q, K, V  
 q, k, v = qkv[0], qkv[1], qkv[2] # 分离Q, K, V  
  
 q = q \* self.scale # 缩放Q  
 attn = (q @ k.transpose(-2, -1)) # 计算注意力权重  
  
 # 添加相对位置偏置  
 relative\_position\_bias = self.relative\_position\_bias\_table[self.relative\_position\_index.view(-1)].view(  
 self.window\_size[0] \* self.window\_size[1], self.window\_size[0] \* self.window\_size[1], -1) # 计算相对位置偏置  
 relative\_position\_bias = relative\_position\_bias.permute(2, 0, 1).contiguous() # 调整维度  
 attn = attn + relative\_position\_bias.unsqueeze(0) # 添加偏置  
  
 if mask is not None: # 如果有mask  
 nW = mask.shape[0]  
 attn = attn.view(B\_ // nW, nW, self.num\_heads, N, N) + mask.unsqueeze(1).unsqueeze(0) # 应用mask  
 attn = attn.view(-1, self.num\_heads, N, N)  
 attn = self.softmax(attn) # 计算softmax  
 else:  
 attn = self.softmax(attn) # 计算softmax  
  
 attn = self.attn\_drop(attn) # 应用Dropout  
  
 x = (attn @ v).transpose(1, 2).reshape(B\_, N, C) # 计算输出  
 x = self.proj(x) # 线性变换  
 x = self.proj\_drop(x) # 应用Dropout  
 return x  
  
  
class SwinTransformer(nn.Module):  
 """ Swin Transformer 主体 """  
  
 def \_\_init\_\_(self, pretrain\_img\_size=224, patch\_size=4, in\_chans=3, embed\_dim=96, depths=[2, 2, 6, 2],  
 num\_heads=[3, 6, 12, 24], window\_size=7, mlp\_ratio=4., qkv\_bias=True, drop\_rate=0.,  
 attn\_drop\_rate=0., drop\_path\_rate=0.2, norm\_layer=nn.LayerNorm, ape=False, patch\_norm=True,  
 out\_indices=(0, 1, 2, 3), frozen\_stages=-1, use\_checkpoint=False):  
 super().\_\_init\_\_()  
  
 self.patch\_embed = PatchEmbed(patch\_size=patch\_size, in\_chans=in\_chans, embed\_dim=embed\_dim) # 图像分块嵌入  
 self.pos\_drop = nn.Dropout(p=drop\_rate) # Dropout层  
  
 # 构建层  
 self.layers = nn.ModuleList()  
 for i\_layer in range(len(depths)):  
 layer = BasicLayer(  
 dim=int(embed\_dim \* 2 \*\* i\_layer),  
 depth=depths[i\_layer],  
 num\_heads=num\_heads[i\_layer],  
 window\_size=window\_size,  
 mlp\_ratio=mlp\_ratio,  
 qkv\_bias=qkv\_bias,  
 drop=drop\_rate,  
 attn\_drop=attn\_drop\_rate,  
 drop\_path=drop\_path\_rate,  
 norm\_layer=norm\_layer,  
 downsample=PatchMerging if (i\_layer < len(depths) - 1) else None,  
 use\_checkpoint=use\_checkpoint)  
 self.layers.append(layer)  
  
 def forward(self, x):  
 """ 前向传播 """  
 x = self.patch\_embed(x) # 图像分块嵌入  
 x = x.flatten(2).transpose(1, 2) # 展平并转置  
 x = self.pos\_drop(x) # 应用Dropout  
  
 outs = []  
 for i in range(len(self.layers)):  
 layer = self.layers[i]  
 x\_out, H, W, x, Wh, Ww = layer(x, Wh, Ww) # 通过每一层  
 outs.append(x\_out) # 保存输出  
  
 return outs # 返回所有层的输出  
```  
  
### 代码说明：  
1. \*\*Mlp类\*\*：实现了一个简单的多层感知机，包括两层线性变换和激活函数。  
2. \*\*WindowAttention类\*\*：实现了窗口自注意力机制，计算输入特征的注意力权重，并添加相对位置偏置。  
3. \*\*SwinTransformer类\*\*：构建了Swin Transformer的整体结构，包括图像分块嵌入、多个基本层和前向传播逻辑。  
  
这些核心部分是Swin Transformer模型的基础，负责处理输入数据并生成输出特征。```

该程序文件实现了Swin Transformer模型，这是一个用于计算机视觉任务的变换器架构。Swin Transformer通过引入局部窗口注意力机制和层次化特征表示，提升了对图像的处理能力。  
  
文件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。接着定义了多个类和函数，构成了Swin Transformer的各个组件。  
  
`Mlp`类实现了一个多层感知机（MLP），包括两个线性层和一个激活函数（默认为GELU），并在每个线性层后应用了Dropout以防止过拟合。  
  
`window\_partition`和`window\_reverse`函数用于将输入特征分割成窗口以及将窗口合并回原始形状。这种窗口机制是Swin Transformer的核心，允许模型在局部区域内进行自注意力计算。  
  
`WindowAttention`类实现了基于窗口的多头自注意力机制，支持相对位置偏置。它通过计算查询、键、值的线性变换，应用注意力机制，并结合相对位置偏置来增强模型的空间感知能力。  
  
`SwinTransformerBlock`类是Swin Transformer的基本构建块，包含了窗口注意力层和前馈网络（MLP）。它通过规范化、残差连接和DropPath技术来增强模型的稳定性和性能。  
  
`PatchMerging`类用于将特征图进行下采样，将4个相邻的patch合并为一个新的patch，从而减少特征图的尺寸。  
  
`BasicLayer`类构建了Swin Transformer的一个层次，包括多个Swin Transformer块，并在必要时应用下采样。  
  
`PatchEmbed`类负责将输入图像分割成patch并进行嵌入，使用卷积层进行线性投影。  
  
`SwinTransformer`类是整个模型的主体，负责构建不同层次的Swin Transformer，并处理输入图像的嵌入、位置编码和输出。  
  
最后，`update\_weight`函数用于更新模型的权重，而`SwinTransformer\_Tiny`函数则是一个工厂函数，用于创建一个小型的Swin Transformer模型实例，并可选择加载预训练权重。  
  
整体而言，该文件实现了Swin Transformer的完整架构，适用于各种视觉任务，如图像分类、目标检测等。通过模块化设计，便于理解和扩展。

### 程序整体功能和构架概括  
  
该程序包含多个深度学习模型的实现，主要用于计算机视觉任务，如图像分类和目标检测。程序通过模块化设计，提供了不同的变换器架构（如EfficientFormer、CSWin Transformer和Swin Transformer），每个模型都具有独特的特性和优势，以适应不同的应用需求。  
  
- \*\*EfficientFormerV2.py\*\*：实现了EfficientFormer V2模型，结合了卷积和注意力机制，旨在提高计算效率和性能，适用于图像处理任务。  
- \*\*CSwomTransformer.py\*\*：实现了CSWin Transformer模型，采用局部增强的注意力机制，能够有效处理图像特征，适用于图像分类等任务。  
- \*\*val.py\*\*：提供了YOLO目标检测模型的验证功能，包括数据预处理、指标计算、结果保存和可视化，旨在评估模型在目标检测任务中的性能。  
- \*\*SwinTransformer.py\*\*：实现了Swin Transformer模型，采用层次化特征表示和窗口注意力机制，适用于各种视觉任务，提升了对图像的处理能力。  
  
### 文件功能整理表  
  
| 文件名 | 功能描述 |  
|---------------------------|---------------------------------------------------------------------------------------------|  
| EfficientFormerV2.py | 实现EfficientFormer V2模型，结合卷积和注意力机制，适用于图像处理任务。 |  
| CSwomTransformer.py | 实现CSWin Transformer模型，采用局部增强的注意力机制，适用于图像分类等任务。 |  
| val.py | 提供YOLO目标检测模型的验证功能，包括数据预处理、指标计算、结果保存和可视化。 |  
| SwinTransformer.py | 实现Swin Transformer模型，采用层次化特征表示和窗口注意力机制，适用于各种视觉任务。 |  
  
以上表格清晰地总结了每个文件的主要功能，便于理解整个程序的结构和用途。