# 改进yolo11-GhostDynamicConv等200+全套创新点大全：血管中红细胞与病毒检测系统源码＆数据集全套

## 3.背景

研究背景与意义  
  
随着生物医学技术的迅速发展，自动化和智能化的医疗检测系统在疾病诊断和治疗中扮演着越来越重要的角色。红细胞和病毒的检测是临床医学中不可或缺的环节，尤其是在传染病暴发和血液疾病诊断中，准确、高效的检测手段能够显著提高医疗响应速度和治疗效果。传统的检测方法往往依赖于人工观察和实验室分析，存在着时间长、效率低和主观性强等缺点。因此，基于计算机视觉的自动检测系统应运而生，成为解决这一问题的有效途径。  
  
本研究旨在基于改进的YOLOv11模型，开发一种高效的血管中红细胞与病毒检测系统。YOLO（You Only Look Once）系列模型以其快速的检测速度和高精度而闻名，适合实时监测和分析。在本项目中，我们将利用一个包含216幅医学图像的数据集，该数据集专注于血液和病毒的识别，类别数量为2，分别为“blood”和“virus”。通过对该数据集的深入分析和处理，我们希望能够提升YOLOv11模型在医学图像分析中的表现，特别是在红细胞与病毒的分离和识别方面。  
  
此外，随着数据集的不断丰富和模型的持续优化，基于深度学习的检测系统将能够适应更复杂的临床场景，为医生提供更为精准的辅助诊断工具。这不仅有助于提高疾病的早期检测率，还能在公共卫生危机中提供快速反应的能力，从而降低疾病传播的风险。综上所述，本研究的开展不仅具有重要的学术价值，也将为实际医疗应用提供有力支持，推动智能医疗的发展。

## 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模型，以实现对血管中红细胞与病毒的高效检测。为此，我们构建了一个专门针对医学领域的多样化数据集，旨在为模型的训练提供丰富而准确的样本。该数据集包含两个主要类别：血液细胞（blood）和病毒（virus），这两个类别在医学图像分析中具有重要的临床意义。通过对这两个类别的深入学习，模型能够更好地识别和区分血管中的红细胞与潜在的病毒感染，从而为临床诊断提供有力支持。  
  
在数据集的构建过程中，我们采用了多种医学影像技术，包括显微镜图像和超声波图像，以确保数据的多样性和代表性。每个类别下的样本均经过精心挑选和标注，确保数据的准确性和可靠性。数据集中的血液细胞样本涵盖了不同的形态和状态，能够反映出各种生理和病理条件下的红细胞特征。而病毒样本则包括了多种类型的病毒图像，旨在帮助模型学习到不同病毒在血管中的表现形式。  
  
此外，为了增强模型的泛化能力，我们还对数据集进行了数据增强处理，包括旋转、缩放、翻转等操作。这些技术不仅提高了数据集的多样性，还帮助模型在面对不同条件下的图像时，能够保持较高的识别准确率。通过这一系列的努力，我们期望构建一个高质量的数据集，为改进YOLOv11模型的训练提供坚实的基础，从而推动血管中红细胞与病毒检测技术的发展，最终实现更为精准的医学诊断和治疗方案。

## 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

```以下是经过简化和注释的核心代码，保留了Swin Transformer的主要结构和功能。  
  
```python  
import torch  
import torch.nn as nn  
import torch.nn.functional as F  
import numpy as np  
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  
 hidden\_features = hidden\_features or in\_features # 隐藏层特征数  
 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)  
 x = self.act(x)  
 x = self.drop(x)  
 x = self.fc2(x)  
 x = self.drop(x)  
 return x  
  
def window\_partition(x, window\_size):  
 """将输入张量分割成窗口"""  
 B, H, W, C = x.shape # 获取输入的维度  
 x = x.view(B, H // window\_size, window\_size, W // window\_size, window\_size, C) # 重新排列维度  
 windows = x.permute(0, 1, 3, 2, 4, 5).contiguous().view(-1, window\_size, window\_size, C) # 生成窗口  
 return windows  
  
def window\_reverse(windows, window\_size, H, W):  
 """将窗口合并回原始张量"""  
 B = int(windows.shape[0] / (H \* W / window\_size / window\_size)) # 计算批次大小  
 x = windows.view(B, H // window\_size, W // window\_size, window\_size, window\_size, -1) # 重新排列维度  
 x = x.permute(0, 1, 3, 2, 4, 5).contiguous().view(B, H, W, -1) # 合并窗口  
 return x  
  
class WindowAttention(nn.Module):  
 """窗口基础的多头自注意力模块"""  
 def \_\_init\_\_(self, dim, window\_size, num\_heads, qkv\_bias=True, attn\_drop=0., proj\_drop=0.):  
 super().\_\_init\_\_()  
 self.dim = dim  
 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 = head\_dim \*\* -0.5 # 缩放因子  
  
 # 定义相对位置偏置参数  
 self.relative\_position\_bias\_table = nn.Parameter(  
 torch.zeros((2 \* window\_size[0] - 1) \* (2 \* window\_size[1] - 1), num\_heads))  
  
 # 计算相对位置索引  
 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])) # 生成坐标网格  
 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() # 重新排列  
 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  
 self.relative\_position\_index = relative\_coords.sum(-1) # 计算相对位置索引  
 self.qkv = nn.Linear(dim, dim \* 3, bias=qkv\_bias) # 线性变换  
 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 # 获取输入维度  
 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)) # 计算注意力分数  
  
 # 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)  
 relative\_position\_bias = relative\_position\_bias.permute(2, 0, 1).contiguous() # 重新排列  
 attn = attn + relative\_position\_bias.unsqueeze(0) # 加入相对位置偏置  
  
 attn = self.softmax(attn) # 计算注意力权重  
 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主类"""  
 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., drop\_rate=0., attn\_drop\_rate=0.,  
 drop\_path\_rate=0.2, norm\_layer=nn.LayerNorm, patch\_norm=True, out\_indices=(0, 1, 2, 3)):  
 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.layers = nn.ModuleList() # 存储各层  
  
 # 构建各层  
 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,  
 drop=drop\_rate,  
 attn\_drop=attn\_drop\_rate,  
 norm\_layer=norm\_layer,  
 downsample=PatchMerging if (i\_layer < len(depths) - 1) else None)  
 )  
 self.layers.append(layer)  
  
 def forward(self, x):  
 """前向传播"""  
 x = self.patch\_embed(x) # 嵌入图像  
 outs = []  
 for layer in self.layers:  
 x\_out, \_, \_, \_, \_, \_ = layer(x) # 通过每一层  
 outs.append(x\_out) # 保存输出  
 return outs # 返回所有层的输出  
  
def SwinTransformer\_Tiny(weights=''):  
 """构建Swin Transformer Tiny模型"""  
 model = SwinTransformer(depths=[2, 2, 6, 2], num\_heads=[3, 6, 12, 24]) # 创建模型  
 if weights:  
 model.load\_state\_dict(torch.load(weights)['model']) # 加载权重  
 return model  
```  
  
### 主要功能概述：  
1. \*\*Mlp\*\*：实现了一个简单的多层感知机，包含两个线性层和激活函数。  
2. \*\*window\_partition\*\* 和 \*\*window\_reverse\*\*：实现了将输入张量分割成窗口和将窗口合并回原始张量的功能。  
3. \*\*WindowAttention\*\*：实现了窗口基础的多头自注意力机制，包含相对位置偏置的计算。  
4. \*\*SwinTransformer\*\*：实现了Swin Transformer的整体结构，包含图像嵌入、多个层的堆叠等功能。  
5. \*\*SwinTransformer\_Tiny\*\*：构建了一个小型的Swin Transformer模型，并可选择加载预训练权重。  
  
以上代码是Swin Transformer的核心部分，提供了基本的功能和结构。```

这个文件实现了Swin Transformer模型，主要用于计算机视觉任务。Swin Transformer是一种分层的视觉Transformer架构，采用了窗口注意力机制，通过局部窗口内的自注意力来提高计算效率，同时通过分层结构来处理不同尺度的特征。  
  
首先，文件中定义了一个多层感知机（Mlp）类，它包含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。这个模块通常用于Transformer中的前馈网络部分。  
  
接下来，定义了两个函数`window\_partition`和`window\_reverse`，用于将输入特征分割成窗口和将窗口合并回原始特征。窗口分割有助于在局部区域内计算自注意力，从而减少计算复杂度。  
  
`WindowAttention`类实现了窗口基础的多头自注意力机制（W-MSA），它支持相对位置偏置，并能够处理平移窗口（SW-MSA）。该类的构造函数中定义了相对位置偏置表，并计算了每个窗口内的相对位置索引。在前向传播中，输入特征被分为查询、键和值，并计算注意力权重，最后将注意力应用于值。  
  
`SwinTransformerBlock`类实现了Swin Transformer的基本模块，包含了窗口注意力和前馈网络。它支持窗口的平移，通过对输入特征进行规范化、分割、计算注意力、合并窗口等步骤，完成了Transformer块的前向传播。  
  
`PatchMerging`类用于将特征图的patch进行合并，减少特征图的分辨率，并增加通道数。这一过程通常在每个阶段的末尾进行，以实现下采样。  
  
`BasicLayer`类代表Swin Transformer的一个基本层，它由多个Swin Transformer块组成，并在必要时包含下采样层。它还计算了用于SW-MSA的注意力掩码。  
  
`PatchEmbed`类将输入图像分割成patch，并通过卷积层进行线性投影，生成嵌入特征。它还可以选择性地应用归一化。  
  
`SwinTransformer`类是整个模型的核心，构造函数中定义了模型的各个参数，包括输入图像的大小、patch大小、嵌入维度、各层的深度和头数等。它还实现了前向传播过程，将输入图像经过patch嵌入、位置嵌入、多个层的处理，最终输出各层的特征。  
  
最后，`update\_weight`函数用于更新模型的权重，确保模型字典中的权重与加载的权重字典匹配。`SwinTransformer\_Tiny`函数则是一个便捷的构造函数，用于创建一个小型的Swin Transformer模型，并可选择性地加载预训练权重。  
  
总体来说，这个文件实现了Swin Transformer的各个组成部分，提供了一个灵活的框架用于视觉任务的特征提取和表示学习。

```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  
  
class OmniAttention(nn.Module):  
 def \_\_init\_\_(self, in\_planes, out\_planes, kernel\_size, groups=1, reduction=0.0625, kernel\_num=4, min\_channel=16):  
 super(OmniAttention, self).\_\_init\_\_()  
 # 计算注意力通道数  
 attention\_channel = max(int(in\_planes \* reduction), min\_channel)  
 self.kernel\_size = kernel\_size  
 self.kernel\_num = kernel\_num  
 self.temperature = 1.0 # 温度参数，用于调整注意力的平滑程度  
  
 # 定义层  
 self.avgpool = nn.AdaptiveAvgPool2d(1) # 自适应平均池化  
 self.fc = nn.Conv2d(in\_planes, attention\_channel, 1, bias=False) # 全连接层  
 self.bn = nn.BatchNorm2d(attention\_channel) # 批归一化  
 self.relu = nn.ReLU(inplace=True) # ReLU激活函数  
  
 # 定义通道注意力  
 self.channel\_fc = nn.Conv2d(attention\_channel, in\_planes, 1, bias=True)  
 self.func\_channel = self.get\_channel\_attention  
  
 # 定义滤波器注意力  
 if in\_planes == groups and in\_planes == out\_planes: # 深度卷积  
 self.func\_filter = self.skip  
 else:  
 self.filter\_fc = nn.Conv2d(attention\_channel, out\_planes, 1, bias=True)  
 self.func\_filter = self.get\_filter\_attention  
  
 # 定义空间注意力  
 if kernel\_size == 1: # 点卷积  
 self.func\_spatial = self.skip  
 else:  
 self.spatial\_fc = nn.Conv2d(attention\_channel, kernel\_size \* kernel\_size, 1, bias=True)  
 self.func\_spatial = self.get\_spatial\_attention  
  
 # 定义核注意力  
 if kernel\_num == 1:  
 self.func\_kernel = self.skip  
 else:  
 self.kernel\_fc = nn.Conv2d(attention\_channel, kernel\_num, 1, bias=True)  
 self.func\_kernel = self.get\_kernel\_attention  
  
 self.\_initialize\_weights() # 初始化权重  
  
 def \_initialize\_weights(self):  
 # 权重初始化  
 for m in self.modules():  
 if isinstance(m, nn.Conv2d):  
 nn.init.kaiming\_normal\_(m.weight, mode='fan\_out', nonlinearity='relu')  
 if m.bias is not None:  
 nn.init.constant\_(m.bias, 0)  
 if isinstance(m, nn.BatchNorm2d):  
 nn.init.constant\_(m.weight, 1)  
 nn.init.constant\_(m.bias, 0)  
  
 def get\_channel\_attention(self, x):  
 # 计算通道注意力  
 channel\_attention = torch.sigmoid(self.channel\_fc(x).view(x.size(0), -1, 1, 1) / self.temperature)  
 return channel\_attention  
  
 def get\_filter\_attention(self, x):  
 # 计算滤波器注意力  
 filter\_attention = torch.sigmoid(self.filter\_fc(x).view(x.size(0), -1, 1, 1) / self.temperature)  
 return filter\_attention  
  
 def get\_spatial\_attention(self, x):  
 # 计算空间注意力  
 spatial\_attention = self.spatial\_fc(x).view(x.size(0), 1, 1, 1, self.kernel\_size, self.kernel\_size)  
 spatial\_attention = torch.sigmoid(spatial\_attention / self.temperature)  
 return spatial\_attention  
  
 def get\_kernel\_attention(self, x):  
 # 计算核注意力  
 kernel\_attention = self.kernel\_fc(x).view(x.size(0), -1, 1, 1, 1, 1)  
 kernel\_attention = F.softmax(kernel\_attention / self.temperature, dim=1)  
 return kernel\_attention  
  
 def forward(self, x):  
 # 前向传播  
 x = self.avgpool(x) # 自适应平均池化  
 x = self.fc(x) # 全连接层  
 x = self.bn(x) # 批归一化  
 x = self.relu(x) # ReLU激活  
 return self.func\_channel(x), self.func\_filter(x), self.func\_spatial(x), self.func\_kernel(x) # 返回四种注意力  
  
class AdaptiveDilatedConv(nn.Module):  
 """自适应膨胀卷积类"""  
 def \_\_init\_\_(self, in\_channels, out\_channels, kernel\_size, stride=1, padding=0, dilation=1, groups=1, bias=True):  
 super(AdaptiveDilatedConv, self).\_\_init\_\_()  
 self.conv = nn.Conv2d(in\_channels, out\_channels, kernel\_size, stride=stride, padding=padding, dilation=dilation, groups=groups, bias=bias)  
  
 def forward(self, x):  
 # 前向传播  
 return self.conv(x) # 返回卷积结果  
```  
  
### 代码说明  
1. \*\*OmniAttention 类\*\*：实现了多种注意力机制，包括通道注意力、滤波器注意力、空间注意力和核注意力。通过对输入特征进行处理，生成不同类型的注意力权重，用于增强模型的表达能力。  
  
2. \*\*AdaptiveDilatedConv 类\*\*：自适应膨胀卷积的实现，允许在卷积操作中使用不同的膨胀率，从而捕捉不同尺度的特征。此类封装了常规卷积操作，便于在模型中使用。  
  
以上是核心部分的代码和详细注释，涵盖了主要的功能和结构。```

这个程序文件 `fadc.py` 是一个用于实现自适应膨胀卷积（Adaptive Dilated Convolution）和频率选择的深度学习模块，主要基于 PyTorch 框架。文件中定义了多个类和函数，以下是对其主要内容的讲解。  
  
首先，文件导入了必要的库，包括 PyTorch 的核心库和一些功能模块。接着，定义了一个名为 `OmniAttention` 的类，该类实现了一种全局注意力机制。该类的构造函数接受多个参数，如输入通道数、输出通道数、卷积核大小等，并根据这些参数初始化不同的卷积层和激活函数。`OmniAttention` 类中定义了多个方法来计算通道注意力、过滤器注意力、空间注意力和卷积核注意力，最后在 `forward` 方法中将这些注意力机制应用于输入数据。  
  
接下来，定义了一个名为 `generate\_laplacian\_pyramid` 的函数，用于生成拉普拉斯金字塔。该函数通过逐层下采样输入张量，计算高频和低频部分，并将其存储在金字塔中。该函数的参数包括输入张量、金字塔层数以及插值模式等。  
  
然后，定义了 `FrequencySelection` 类，该类用于在频域中选择特定的频率成分。该类的构造函数接受多个参数，包括输入通道数、频率列表、空间卷积的参数等。该类中实现了频率选择的逻辑，并在 `forward` 方法中将输入数据与选择的频率成分进行处理。  
  
最后，定义了两个自适应卷积类 `AdaptiveDilatedConv` 和 `AdaptiveDilatedDWConv`，它们继承自 `ModulatedDeformConv2d`。这两个类的构造函数中初始化了卷积层、偏移量卷积层和掩码卷积层，并根据不同的参数配置相应的注意力机制和频率选择。它们的 `forward` 方法实现了自适应卷积的具体计算过程，包括对输入数据的偏移、掩码和权重的计算。  
  
总体来说，这个程序文件实现了一个复杂的卷积神经网络模块，结合了自适应卷积、注意力机制和频率选择等先进的技术，适用于图像处理和计算机视觉等领域。

```以下是提取出的核心代码部分，并添加了详细的中文注释：  
  
```python  
import torch  
import torch.nn as nn  
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  
 hidden\_features = hidden\_features or in\_features # 隐藏层特征数  
 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 CSWinBlock(nn.Module):  
 """CSWin Transformer中的一个块"""  
 def \_\_init\_\_(self, dim, num\_heads, mlp\_ratio=4., drop=0., attn\_drop=0.):  
 super().\_\_init\_\_()  
 self.dim = dim # 输入特征维度  
 self.num\_heads = num\_heads # 注意力头数  
 self.mlp\_ratio = mlp\_ratio # MLP的隐藏层比率  
 self.qkv = nn.Linear(dim, dim \* 3) # 线性变换生成Q、K、V  
 self.norm1 = nn.LayerNorm(dim) # 第一层归一化  
 self.attn = LePEAttention(dim, num\_heads=num\_heads, attn\_drop=attn\_drop) # 注意力层  
 self.mlp = Mlp(in\_features=dim, hidden\_features=int(dim \* mlp\_ratio), out\_features=dim) # MLP层  
 self.norm2 = nn.LayerNorm(dim) # 第二层归一化  
  
 def forward(self, x):  
 """前向传播"""  
 img = self.norm1(x) # 归一化  
 qkv = self.qkv(img).reshape(-1, 3, self.dim) # 生成Q、K、V  
 attn\_output = self.attn(qkv) # 注意力计算  
 x = x + attn\_output # 残差连接  
 x = x + self.mlp(self.norm2(x)) # MLP层和残差连接  
 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 # 类别数  
 self.embed\_dim = embed\_dim # 嵌入维度  
  
 # 第一阶段的卷积嵌入  
 self.stage1\_conv\_embed = nn.Sequential(  
 nn.Conv2d(in\_chans, embed\_dim, kernel\_size=7, stride=4, padding=2),  
 nn.LayerNorm(embed\_dim)  
 )  
  
 # 定义多个CSWinBlock  
 self.stage1 = nn.ModuleList([  
 CSWinBlock(dim=embed\_dim, num\_heads=12) for \_ in range(depth[0])  
 ])  
 # 其他阶段的定义略去...  
  
 def forward(self, x):  
 """前向传播"""  
 x = self.stage1\_conv\_embed(x) # 卷积嵌入  
 for blk in self.stage1:  
 x = blk(x) # 通过每个块  
 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)) # 输入张量  
 model = CSWinTransformer() # 创建模型  
 res = model(inputs) # 前向传播  
 print(res.size()) # 输出结果的尺寸  
```  
  
### 代码注释说明：  
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. \*\*CSWinBlock类\*\*：实现了CSWin Transformer中的一个基本块，包含注意力机制和MLP，使用残差连接。  
3. \*\*CSWinTransformer类\*\*：定义了整个CSWin Transformer模型的结构，包括输入嵌入和多个CSWinBlock的堆叠。  
4. \*\*前向传播\*\*：展示了如何通过模型进行前向传播并输出结果的尺寸。  
  
以上代码为CSWin Transformer的核心部分，其他辅助函数和类被省略以保持简洁。```

这个程序文件 `CSWinTransformer.py` 实现了 CSWin Transformer 模型，这是一个用于计算机视觉任务的深度学习模型。文件中包含了多个类和函数，主要功能是构建和训练 CSWin Transformer 网络。  
  
首先，文件导入了必要的库，包括 PyTorch 和一些用于图像处理和模型构建的辅助函数。接着，定义了一些全局变量和常量，便于后续使用。  
  
文件中定义了多个类。`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），并在每个线性层后添加了 dropout 层以防止过拟合。`LePEAttention` 类实现了一个局部增强的自注意力机制，主要通过将输入图像划分为窗口来计算注意力。它支持多头注意力机制，并可以处理不同的分辨率和窗口大小。  
  
`CSWinBlock` 类是 CSWin Transformer 的基本构建块，包含了多头自注意力层和 MLP 层。它通过归一化层和残差连接来增强模型的学习能力。该类根据输入的分辨率和分支数量，动态地构建不同的注意力层。  
  
`Merge\_Block` 类用于在不同阶段之间合并特征图，通常是通过卷积层来减少特征图的尺寸，同时增加通道数。  
  
`CSWinTransformer` 类是整个模型的核心，负责将输入图像转换为特征表示。它包含多个阶段，每个阶段由多个 CSWinBlock 组成。每个阶段后面都有一个 Merge\_Block，用于处理特征图的合并和下采样。模型的初始化函数中设置了输入参数，包括图像大小、补丁大小、嵌入维度、深度等，并使用这些参数构建各个阶段的模块。  
  
此外，文件中还定义了一些辅助函数，例如 `img2windows` 和 `windows2img`，用于在图像和窗口之间进行转换，方便在自注意力计算中使用。  
  
最后，文件提供了四个不同规模的模型构造函数（`CSWin\_tiny`、`CSWin\_small`、`CSWin\_base` 和 `CSWin\_large`），允许用户根据需求选择不同的模型结构。每个函数都可以选择加载预训练权重，以便在特定任务上进行微调。  
  
在文件的最后部分，提供了一个简单的测试代码，创建了不同规模的模型并对随机输入进行前向传播，输出每个模型的特征图大小。这部分代码用于验证模型的构建是否正确。

```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。  
 """  
  
 def \_\_init\_\_(self, dataloader=None, save\_dir=None, pbar=None, args=None, \_callbacks=None):  
 """初始化检测模型所需的变量和设置。"""  
 super().\_\_init\_\_(dataloader, save\_dir, pbar, args, \_callbacks)  
 self.nt\_per\_class = None # 每个类别的目标数量  
 self.is\_coco = False # 是否为 COCO 数据集  
 self.class\_map = None # 类别映射  
 self.args.task = "detect" # 设置任务为检测  
 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) # mAP@0.5:0.95 的 IoU 向量  
 self.niou = self.iouv.numel() # IoU 的数量  
 self.lb = [] # 用于自动标注  
  
 def preprocess(self, batch):  
 """对 YOLO 训练的图像批次进行预处理。"""  
 # 将图像移动到设备上并进行归一化处理  
 batch["img"] = batch["img"].to(self.device, non\_blocking=True)  
 batch["img"] = (batch["img"].half() if self.args.half else batch["img"].float()) / 255  
 for k in ["batch\_idx", "cls", "bboxes"]:  
 batch[k] = batch[k].to(self.device)  
  
 # 如果需要保存混合数据，则进行处理  
 if self.args.save\_hybrid:  
 height, width = batch["img"].shape[2:]  
 nb = len(batch["img"])  
 bboxes = batch["bboxes"] \* torch.tensor((width, height, width, height), device=self.device)  
 self.lb = (  
 [  
 torch.cat([batch["cls"][batch["batch\_idx"] == i], bboxes[batch["batch\_idx"] == i]], dim=-1)  
 for i in range(nb)  
 ]  
 if self.args.save\_hybrid  
 else []  
 ) # 用于自动标注  
  
 return batch  
  
 def postprocess(self, preds):  
 """对预测输出应用非极大值抑制。"""  
 return ops.non\_max\_suppression(  
 preds,  
 self.args.conf,  
 self.args.iou,  
 labels=self.lb,  
 multi\_label=True,  
 agnostic=self.args.single\_cls,  
 max\_det=self.args.max\_det,  
 )  
  
 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  
 npr = len(pred) # 当前预测的数量  
 stat = dict(  
 conf=torch.zeros(0, device=self.device),  
 pred\_cls=torch.zeros(0, device=self.device),  
 tp=torch.zeros(npr, self.niou, dtype=torch.bool, device=self.device),  
 )  
 pbatch = self.\_prepare\_batch(si, batch) # 准备当前批次的真实标签  
 cls, bbox = pbatch.pop("cls"), pbatch.pop("bbox") # 提取类别和边界框  
 nl = len(cls) # 真实标签的数量  
 stat["target\_cls"] = cls # 记录真实类别  
  
 if npr == 0: # 如果没有预测  
 if nl:  
 for k in self.stats.keys():  
 self.stats[k].append(stat[k])  
 continue  
  
 # 处理预测  
 if self.args.single\_cls:  
 pred[:, 5] = 0 # 如果是单类检测，设置类别为0  
 predn = self.\_prepare\_pred(pred, pbatch) # 准备预测数据  
 stat["conf"] = predn[:, 4] # 置信度  
 stat["pred\_cls"] = predn[:, 5] # 预测类别  
  
 # 评估  
 if nl:  
 stat["tp"] = self.\_process\_batch(predn, bbox, cls) # 计算真阳性  
 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  
 self.nt\_per\_class = np.bincount(  
 stats["target\_cls"].astype(int), minlength=self.nc  
 ) # 计算每个类别的目标数量  
 return self.metrics.results\_dict # 返回结果字典  
```  
  
### 代码核心部分解释：  
1. \*\*DetectionValidator 类\*\*：该类负责对 YOLO 模型进行验证，继承自 `BaseValidator`。  
2. \*\*\_\_init\_\_ 方法\*\*：初始化验证器的基本参数，包括数据加载器、保存目录、指标等。  
3. \*\*preprocess 方法\*\*：对输入的图像批次进行预处理，包括归一化和将数据移动到指定设备。  
4. \*\*postprocess 方法\*\*：对模型的预测结果应用非极大值抑制，以减少重复检测。  
5. \*\*update\_metrics 方法\*\*：更新模型的评估指标，包括计算真阳性、置信度等。  
6. \*\*get\_stats 方法\*\*：计算并返回模型的评估结果，包含每个类别的目标数量和相关指标。   
  
这些方法共同构成了检测模型验证的核心逻辑，确保模型在验证集上的性能能够被准确评估。```

这个程序文件 `val.py` 是一个用于验证目标检测模型（如 YOLO）的类和方法的实现。它主要依赖于 Ultralytics YOLO 库，并扩展了一个基类 `BaseValidator`。该文件的核心功能是对模型在验证集上的性能进行评估，计算各种指标，并可视化结果。  
  
首先，程序导入了一些必要的库，包括 `os`、`numpy` 和 `torch`，以及 Ultralytics YOLO 中的一些模块。接着定义了 `DetectionValidator` 类，该类专门用于基于检测模型的验证。  
  
在初始化方法 `\_\_init\_\_` 中，类会设置一些必要的变量和参数，例如验证数据加载器、保存目录、进度条、参数字典等。它还初始化了一些指标，如 `DetMetrics` 和混淆矩阵，并设置了一些用于计算 mAP（平均精度）的参数。  
  
`preprocess` 方法用于对输入的图像批次进行预处理，包括将图像数据转移到指定设备（如 GPU），并进行归一化处理。该方法还会根据需要生成用于自动标注的标签。  
  
`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，并设置相应的类别映射和指标名称。  
  
`postprocess` 方法应用非极大值抑制（NMS）来处理模型的预测输出，以减少冗余的检测框。  
  
`\_prepare\_batch` 和 `\_prepare\_pred` 方法分别用于准备验证批次和预测结果，以便进行后续的评估。  
  
`update\_metrics` 方法则是计算和更新各种指标的核心，遍历每个预测结果并与真实标签进行比较，更新统计信息。  
  
`finalize\_metrics` 方法用于设置最终的指标值，包括速度和混淆矩阵。  
  
`get\_stats` 方法返回计算得到的指标统计信息，使用 NumPy 将统计信息转换为数组格式。  
  
`print\_results` 方法负责打印训练或验证集的每个类别的指标结果，并根据需要绘制混淆矩阵。  
  
`\_process\_batch` 方法用于计算正确的预测矩阵，比较预测框和真实框之间的 IoU（交并比）。  
  
`build\_dataset` 和 `get\_dataloader` 方法用于构建数据集和数据加载器，以便于模型验证。  
  
`plot\_val\_samples` 和 `plot\_predictions` 方法用于可视化验证样本和模型预测结果，并将其保存为图像文件。  
  
`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，以便后续分析。  
  
最后，`eval\_json` 方法用于评估 YOLO 输出的 JSON 格式结果，并返回性能统计信息，特别是计算 COCO 数据集的 mAP 值。  
  
整体来看，这个文件实现了一个完整的目标检测模型验证流程，涵盖了数据预处理、指标计算、结果可视化等多个方面，适用于使用 YOLO 模型进行目标检测任务的场景。

### 程序整体功能和构架概括  
  
该程序包含多个文件，每个文件实现了不同的深度学习模型或模块，主要用于计算机视觉任务。整体上，这些文件构成了一个灵活的框架，支持多种模型的构建、训练和验证。具体来说：  
  
1. \*\*SwinTransformer.py\*\*：实现了 Swin Transformer 模型，采用分层结构和窗口注意力机制，适用于图像分类和目标检测等任务。  
2. \*\*fadc.py\*\*：实现了自适应膨胀卷积和频率选择的模块，结合了注意力机制，增强了卷积神经网络的特征提取能力。  
3. \*\*CSwomTransformer.py\*\*：实现了 CSWin Transformer 模型，利用局部增强的自注意力机制和分层结构，进一步提升了模型在视觉任务中的表现。  
4. \*\*val.py\*\*：提供了目标检测模型的验证功能，计算各种性能指标，并可视化结果，支持与 YOLO 等模型的集成。  
  
### 文件功能整理表  
  
| 文件名 | 功能描述 |  
|---------------------------|------------------------------------------------------------------|  
| \*\*SwinTransformer.py\*\* | 实现 Swin Transformer 模型，包含多层感知机、窗口注意力机制等。 |  
| \*\*fadc.py\*\* | 实现自适应膨胀卷积和频率选择模块，结合注意力机制增强特征提取能力。 |  
| \*\*CSwomTransformer.py\*\* | 实现 CSWin Transformer 模型，采用局部增强自注意力机制和分层结构。 |  
| \*\*val.py\*\* | 提供目标检测模型的验证功能，计算性能指标并可视化结果。 |  
  
这些文件共同构成了一个强大的计算机视觉框架，支持多种模型的构建、训练和评估，适用于不同的视觉任务。