# 改进yolo11-efficientViT等200+全套创新点大全：遥感图温室实例分割系统源码＆数据集全套

## 3.背景

研究背景与意义  
  
随着全球气候变化和人口增长的加剧，温室农业作为一种高效的农业生产方式，逐渐受到重视。温室能够提供一个可控的生长环境，优化作物的生长条件，提高产量和质量。然而，传统的温室管理依赖于人工监测和管理，效率低下且容易出现误差。因此，利用先进的计算机视觉技术对温室进行智能化管理，成为了当前农业科技研究的重要方向。  
  
在这一背景下，遥感技术的应用为温室的监测和管理提供了新的可能性。通过遥感图像获取温室的实时数据，结合图像处理和深度学习技术，可以实现对温室内作物生长状态的自动化分析和监测。尤其是实例分割技术，可以精确地识别和分离出温室内的作物，进而为后续的生长分析、病虫害监测和产量预测提供数据支持。  
  
本研究基于改进的YOLOv11模型，构建一个高效的遥感图温室实例分割系统。该系统利用3900张标注为“温室”的图像数据集，能够实现对温室内作物的精确分割与识别。通过对YOLOv11模型的改进，提升其在实例分割任务中的表现，能够有效应对温室环境中的复杂背景和光照变化，提高分割精度和实时性。  
  
此外，本研究的成果不仅可以为温室农业的智能化管理提供技术支持，还可以为相关领域的研究提供借鉴。通过对温室实例分割的深入研究，推动计算机视觉技术在农业中的应用，助力实现农业生产的数字化和智能化转型，从而提升农业生产效率，保障粮食安全，促进可持续发展。

## 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  
  
本项目所使用的数据集以“detectron160”为主题，专门针对遥感图像中的温室实例分割任务进行设计和构建。该数据集的主要目标是为改进YOLOv11模型提供高质量的训练数据，确保其在温室检测和分割方面的性能得到显著提升。数据集中包含的类别数量为1，具体类别为“greenhouse”，这意味着所有的标注和数据均围绕温室这一特定对象展开。通过集中于单一类别，数据集能够提供更为精确和深入的特征学习，进而提升模型在特定场景下的识别能力。  
  
在数据集的构建过程中，研究团队采用了多种遥感图像采集技术，确保数据的多样性和代表性。这些图像涵盖了不同的地理区域、气候条件以及不同生长阶段的温室，力求反映出真实世界中温室的多样性和复杂性。此外，数据集中的图像经过精细的标注，确保每个温室实例都被准确地框定和分割，从而为模型的训练提供了可靠的监督信号。  
  
在数据预处理阶段，团队还对图像进行了标准化处理，以适应YOLOv11模型的输入要求。数据集的规模和质量将直接影响模型的训练效果，因此在数据集的选择和构建上，团队注重数据的完整性和准确性，力求为后续的模型训练和评估提供坚实的基础。通过使用“detectron160”数据集，本项目旨在实现对遥感图像中温室的高效检测与分割，为农业监测和管理提供更为智能化的解决方案。

## 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

```以下是对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 torch.nn.functional as F  
import itertools  
from timm.models.layers import SqueezeExcite  
  
# 定义卷积层和批归一化的组合  
class Conv2d\_BN(torch.nn.Sequential):  
 def \_\_init\_\_(self, in\_channels, out\_channels, kernel\_size=1, stride=1, padding=0, dilation=1, groups=1, bn\_weight\_init=1):  
 super().\_\_init\_\_()  
 # 添加卷积层  
 self.add\_module('conv', torch.nn.Conv2d(in\_channels, out\_channels, kernel\_size, stride, padding, dilation, groups, bias=False))  
 # 添加批归一化层  
 self.add\_module('bn', torch.nn.BatchNorm2d(out\_channels))  
 # 初始化批归一化的权重  
 torch.nn.init.constant\_(self.bn.weight, bn\_weight\_init)  
 torch.nn.init.constant\_(self.bn.bias, 0)  
  
 @torch.no\_grad()  
 def switch\_to\_deploy(self):  
 # 将训练模式的卷积和批归一化合并为一个卷积层  
 conv, bn = self.\_modules.values()  
 w = bn.weight / (bn.running\_var + bn.eps)\*\*0.5  
 w = conv.weight \* w[:, None, None, None]  
 b = bn.bias - bn.running\_mean \* bn.weight / (bn.running\_var + bn.eps)\*\*0.5  
 m = torch.nn.Conv2d(w.size(1) \* conv.groups, w.size(0), w.shape[2:], stride=conv.stride, padding=conv.padding, dilation=conv.dilation, groups=conv.groups)  
 m.weight.data.copy\_(w)  
 m.bias.data.copy\_(b)  
 return m  
  
# 定义有效的ViT块  
class EfficientViTBlock(torch.nn.Module):  
 def \_\_init\_\_(self, type, embed\_dim, key\_dim, num\_heads=8, attn\_ratio=4, resolution=14, window\_resolution=7):  
 super().\_\_init\_\_()  
 # 深度卷积  
 self.dw0 = Residual(Conv2d\_BN(embed\_dim, embed\_dim, 3, 1, 1, groups=embed\_dim))  
 # 前馈网络  
 self.ffn0 = Residual(FFN(embed\_dim, int(embed\_dim \* 2)))  
   
 # 根据类型选择混合器  
 if type == 's':  
 self.mixer = Residual(LocalWindowAttention(embed\_dim, key\_dim, num\_heads, attn\_ratio, resolution, window\_resolution))  
  
 self.dw1 = Residual(Conv2d\_BN(embed\_dim, embed\_dim, 3, 1, 1, groups=embed\_dim))  
 self.ffn1 = Residual(FFN(embed\_dim, int(embed\_dim \* 2)))  
  
 def forward(self, x):  
 # 前向传播  
 return self.ffn1(self.dw1(self.mixer(self.ffn0(self.dw0(x)))))  
  
# 定义EfficientViT模型  
class EfficientViT(torch.nn.Module):  
 def \_\_init\_\_(self, img\_size=400, patch\_size=16, in\_chans=3, stages=['s', 's', 's'], embed\_dim=[64, 128, 192], key\_dim=[16, 16, 16], depth=[1, 2, 3], num\_heads=[4, 4, 4], window\_size=[7, 7, 7]):  
 super().\_\_init\_\_()  
 # 图像嵌入层  
 self.patch\_embed = torch.nn.Sequential(  
 Conv2d\_BN(in\_chans, embed\_dim[0] // 8, 3, 2, 1),  
 torch.nn.ReLU(),  
 Conv2d\_BN(embed\_dim[0] // 8, embed\_dim[0] // 4, 3, 2, 1),  
 torch.nn.ReLU(),  
 Conv2d\_BN(embed\_dim[0] // 4, embed\_dim[0] // 2, 3, 2, 1),  
 torch.nn.ReLU(),  
 Conv2d\_BN(embed\_dim[0] // 2, embed\_dim[0], 3, 1, 1)  
 )  
  
 # 创建多个EfficientViT块  
 self.blocks = []  
 for i, (stg, ed, kd, dpth, nh) in enumerate(zip(stages, embed\_dim, key\_dim, depth, num\_heads)):  
 for d in range(dpth):  
 self.blocks.append(EfficientViTBlock(stg, ed, kd, nh, resolution=img\_size // patch\_size))  
 self.blocks = torch.nn.Sequential(\*self.blocks)  
  
 def forward(self, x):  
 # 前向传播  
 x = self.patch\_embed(x)  
 x = self.blocks(x)  
 return x  
  
# 定义模型的不同配置  
EfficientViT\_m0 = {  
 'img\_size': 224,  
 'patch\_size': 16,  
 'embed\_dim': [64, 128, 192],  
 'depth': [1, 2, 3],  
 'num\_heads': [4, 4, 4],  
 'window\_size': [7, 7, 7],  
}  
  
# 定义模型创建函数  
def EfficientViT\_M0(pretrained='', frozen\_stages=0):  
 model = EfficientViT(frozen\_stages=frozen\_stages, \*\*EfficientViT\_m0)  
 if pretrained:  
 model.load\_state\_dict(torch.load(pretrained)['model'])  
 return model  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 model = EfficientViT\_M0('efficientvit\_m0.pth')  
 inputs = torch.randn((1, 3, 640, 640))  
 res = model(inputs)  
 for i in res:  
 print(i.size())  
```  
  
### 代码注释说明：  
1. \*\*Conv2d\_BN\*\*: 这个类将卷积层和批归一化层组合在一起，提供了一个便捷的方式来初始化和使用卷积层。  
2. \*\*EfficientViTBlock\*\*: 这个类定义了一个有效的ViT块，包含深度卷积、前馈网络和注意力机制。  
3. \*\*EfficientViT\*\*: 这个类是整个模型的实现，负责将输入图像转换为嵌入，并通过多个ViT块进行处理。  
4. \*\*模型配置\*\*: 提供了不同的模型配置，以便于创建不同规模的EfficientViT模型。  
5. \*\*模型创建函数\*\*: 提供了一个函数来创建模型实例，并加载预训练权重（如果提供）。  
  
通过这些核心部分和注释，读者可以更好地理解EfficientViT模型的结构和功能。```

这个文件定义了一个名为 `EfficientViT` 的深度学习模型架构，主要用于图像处理任务。该模型的设计灵感来源于视觉Transformer（ViT），并结合了一些高效的卷积操作和注意力机制，以提高性能和效率。  
  
首先，文件导入了必要的库，包括 PyTorch 和一些特定的模块。接着，定义了一些基本的构建块，例如 `Conv2d\_BN` 类，它是一个包含卷积层和批归一化层的组合。这个类在初始化时会创建一个卷积层和一个批归一化层，并对批归一化的权重进行初始化。  
  
`replace\_batchnorm` 函数用于替换模型中的批归一化层为恒等映射，以便在推理时提高效率。接下来，定义了 `PatchMerging` 类，它用于将输入特征图的多个小块合并为更大的块，以减少特征图的分辨率并增加通道数。  
  
`Residual` 类实现了残差连接的功能，可以在训练过程中随机丢弃部分输入，以增强模型的鲁棒性。`FFN` 类实现了前馈神经网络，包含两个卷积层和一个激活函数。  
  
`CascadedGroupAttention` 和 `LocalWindowAttention` 类实现了不同类型的注意力机制。前者是级联组注意力，后者是局部窗口注意力。它们都通过计算查询、键和值之间的关系来捕捉特征之间的依赖关系。  
  
`EfficientViTBlock` 类是模型的基本构建块，包含多个卷积层、前馈网络和注意力机制。`EfficientViT` 类则是整个模型的核心，负责构建模型的不同阶段，包括特征提取和特征融合。  
  
在模型的初始化过程中，定义了多个超参数，例如图像大小、补丁大小、嵌入维度、深度等，并根据这些参数构建模型的不同层。模型的前向传播方法将输入数据依次通过各个层，并在每个阶段收集输出。  
  
最后，文件中定义了一些不同配置的模型实例（如 `EfficientViT\_m0` 到 `EfficientViT\_m5`），并提供了加载预训练权重和替换批归一化的功能。`update\_weight` 函数用于更新模型的权重字典，确保模型的权重与预训练权重匹配。  
  
在 `\_\_main\_\_` 部分，代码展示了如何实例化模型并进行一次前向传播，打印出每个输出的尺寸。这段代码可以用来验证模型的结构和功能是否正常。

```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，如果未指定则与输入特征数相同  
 hidden\_features = hidden\_features or in\_features # 隐藏层特征数，如果未指定则与输入特征数相同  
 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):  
 """ 前向传播函数。"""  
 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, 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 = 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() # 调整维度  
 relative\_coords[:, :, 0] += self.window\_size[0] - 1 # 位置偏置调整  
 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) # 线性变换用于生成Q, K, V  
 self.attn\_drop = nn.Dropout(attn\_drop) # 注意力权重的Dropout  
 self.proj = nn.Linear(dim, dim) # 输出线性变换  
 self.proj\_drop = nn.Dropout(proj\_drop) # 输出的Dropout  
 self.softmax = nn.Softmax(dim=-1) # Softmax层  
  
 def forward(self, x, mask=None):  
 """ 前向传播函数。"""  
 B\_, N, C = x.shape # B\_: 批量大小, N: token数量, 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) # 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  
  
 attn = self.softmax(attn) # 计算注意力权重  
 attn = self.attn\_drop(attn) # 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) # 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., drop\_rate=0., attn\_drop\_rate=0., drop\_path\_rate=0.2, norm\_layer=nn.LayerNorm):  
 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,  
 drop=drop\_rate,  
 attn\_drop=attn\_drop\_rate,  
 norm\_layer=norm\_layer)  
 self.layers.append(layer)  
  
 def forward(self, x):  
 """ 前向传播函数。"""  
 x = self.patch\_embed(x) # 图像分块嵌入  
 x = self.pos\_drop(x) # Dropout  
  
 for layer in self.layers:  
 x\_out, \_, \_, x, \_, \_ = layer(x) # 逐层前向传播  
  
 return x\_out # 返回最后一层的输出  
```  
  
### 主要部分解释：  
1. \*\*Mlp类\*\*：实现了一个简单的多层感知机，包含两层线性变换和激活函数。  
2. \*\*WindowAttention类\*\*：实现了窗口基础的多头自注意力机制，计算输入特征的注意力分数，并考虑相对位置偏置。  
3. \*\*SwinTransformer类\*\*：构建了Swin Transformer的主体结构，包括图像分块嵌入和多个Transformer层的堆叠。```

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深度学习模型。Swin Transformer的核心思想是通过局部窗口注意力机制来处理图像数据，从而有效地捕捉图像中的上下文信息。  
  
首先，文件导入了必要的库，包括PyTorch及其相关模块。接着定义了一个多层感知机（Mlp）类，用于构建全连接层和激活函数，包含前向传播方法。  
  
接下来，定义了两个函数`window\_partition`和`window\_reverse`，分别用于将输入特征图划分为窗口和将窗口合并回特征图。这种窗口划分的方式使得模型能够在局部区域内进行自注意力计算。  
  
然后，定义了`WindowAttention`类，它实现了基于窗口的多头自注意力机制。该类的构造函数中初始化了输入通道数、窗口大小、注意力头数等参数，并计算了相对位置偏置。前向传播方法中，输入特征经过线性变换得到查询、键、值，并计算注意力权重，最后通过加权求和得到输出。  
  
`SwinTransformerBlock`类实现了Swin Transformer的基本模块，包含规范化层、窗口注意力层和多层感知机。该模块支持循环移位操作，以实现跨窗口的信息交流。  
  
`PatchMerging`类用于将特征图的补丁合并，以降低特征图的分辨率。它通过线性变换将四个补丁合并为一个新的补丁，并进行规范化。  
  
`BasicLayer`类则表示Swin Transformer中的一个基本层，包含多个Swin Transformer块和一个下采样层。它负责构建一系列的Swin Transformer块，并在必要时进行下采样。  
  
`PatchEmbed`类用于将输入图像划分为补丁并进行嵌入，使用卷积层实现补丁的线性投影。  
  
`SwinTransformer`类是整个模型的主体，负责将图像输入转化为补丁，进行位置编码，并通过多个基本层进行特征提取。它还包含了权重更新的函数`update\_weight`，用于加载预训练模型的权重。  
  
最后，`SwinTransformer\_Tiny`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。  
  
整体来看，该文件实现了Swin Transformer的完整结构，涵盖了从输入处理到特征提取的各个环节，适用于图像分类、目标检测等视觉任务。

```以下是代码中最核心的部分，并附上详细的中文注释：  
  
```python  
import torch  
import torch.nn as nn  
import torch.nn.functional as F  
from torch.autograd import Function  
import pywt  
  
# 创建小波滤波器  
def create\_wavelet\_filter(wave, in\_size, out\_size, type=torch.float):  
 # 使用PyWavelets库创建小波对象  
 w = pywt.Wavelet(wave)  
   
 # 获取小波的分解滤波器  
 dec\_hi = torch.tensor(w.dec\_hi[::-1], dtype=type) # 高频滤波器  
 dec\_lo = torch.tensor(w.dec\_lo[::-1], dtype=type) # 低频滤波器  
   
 # 生成分解滤波器  
 dec\_filters = torch.stack([  
 dec\_lo.unsqueeze(0) \* dec\_lo.unsqueeze(1), # 低频-低频  
 dec\_lo.unsqueeze(0) \* dec\_hi.unsqueeze(1), # 低频-高频  
 dec\_hi.unsqueeze(0) \* dec\_lo.unsqueeze(1), # 高频-低频  
 dec\_hi.unsqueeze(0) \* dec\_hi.unsqueeze(1) # 高频-高频  
 ], dim=0)  
  
 # 重复滤波器以适应输入通道数  
 dec\_filters = dec\_filters[:, None].repeat(in\_size, 1, 1, 1)  
  
 # 获取小波的重构滤波器  
 rec\_hi = torch.tensor(w.rec\_hi[::-1], dtype=type).flip(dims=[0])  
 rec\_lo = torch.tensor(w.rec\_lo[::-1], dtype=type).flip(dims=[0])  
   
 # 生成重构滤波器  
 rec\_filters = torch.stack([  
 rec\_lo.unsqueeze(0) \* rec\_lo.unsqueeze(1), # 低频-低频  
 rec\_lo.unsqueeze(0) \* rec\_hi.unsqueeze(1), # 低频-高频  
 rec\_hi.unsqueeze(0) \* rec\_lo.unsqueeze(1), # 高频-低频  
 rec\_hi.unsqueeze(0) \* rec\_hi.unsqueeze(1) # 高频-高频  
 ], dim=0)  
  
 # 重复滤波器以适应输出通道数  
 rec\_filters = rec\_filters[:, None].repeat(out\_size, 1, 1, 1)  
  
 return dec\_filters, rec\_filters  
  
# 小波变换  
def wavelet\_transform(x, filters):  
 b, c, h, w = x.shape # 获取输入的形状  
 pad = (filters.shape[2] // 2 - 1, filters.shape[3] // 2 - 1) # 计算填充  
 # 进行卷积操作，执行小波变换  
 x = F.conv2d(x, filters.to(x.dtype).to(x.device), stride=2, groups=c, padding=pad)  
 x = x.reshape(b, c, 4, h // 2, w // 2) # 重塑输出形状  
 return x  
  
# 逆小波变换  
def inverse\_wavelet\_transform(x, filters):  
 b, c, \_, h\_half, w\_half = x.shape # 获取输入的形状  
 pad = (filters.shape[2] // 2 - 1, filters.shape[3] // 2 - 1) # 计算填充  
 x = x.reshape(b, c \* 4, h\_half, w\_half) # 重塑输入形状  
 # 进行转置卷积操作，执行逆小波变换  
 x = F.conv\_transpose2d(x, filters.to(x.dtype).to(x.device), stride=2, groups=c, padding=pad)  
 return x  
  
# 定义小波变换的自定义函数  
class WaveletTransform(Function):  
 @staticmethod  
 def forward(ctx, input, filters):  
 ctx.filters = filters # 保存滤波器  
 with torch.no\_grad():  
 x = wavelet\_transform(input, filters) # 执行小波变换  
 return x  
  
 @staticmethod  
 def backward(ctx, grad\_output):  
 grad = inverse\_wavelet\_transform(grad\_output, ctx.filters) # 执行逆小波变换  
 return grad, None  
  
# 定义小波卷积层  
class WTConv2d(nn.Module):  
 def \_\_init\_\_(self, in\_channels, out\_channels, kernel\_size=5, stride=1, bias=True, wt\_levels=1, wt\_type='db1'):  
 super(WTConv2d, self).\_\_init\_\_()  
  
 assert in\_channels == out\_channels # 输入通道数必须等于输出通道数  
  
 self.in\_channels = in\_channels  
 self.wt\_levels = wt\_levels  
 self.stride = stride  
  
 # 创建小波滤波器  
 self.wt\_filter, self.iwt\_filter = create\_wavelet\_filter(wt\_type, in\_channels, in\_channels, torch.float)  
 self.wt\_filter = nn.Parameter(self.wt\_filter, requires\_grad=False) # 不需要训练的小波滤波器  
 self.iwt\_filter = nn.Parameter(self.iwt\_filter, requires\_grad=False) # 不需要训练的逆小波滤波器  
   
 # 定义小波变换和逆小波变换的应用函数  
 self.wt\_function = wavelet\_transform\_init(self.wt\_filter)  
 self.iwt\_function = inverse\_wavelet\_transform\_init(self.iwt\_filter)  
  
 # 基础卷积层  
 self.base\_conv = nn.Conv2d(in\_channels, in\_channels, kernel\_size, padding='same', stride=1, groups=in\_channels, bias=bias)  
 self.base\_scale = \_ScaleModule([1, in\_channels, 1, 1]) # 缩放模块  
  
 # 定义小波卷积层  
 self.wavelet\_convs = nn.ModuleList(  
 [nn.Conv2d(in\_channels \* 4, in\_channels \* 4, kernel\_size, padding='same', stride=1, groups=in\_channels \* 4, bias=False) for \_ in range(self.wt\_levels)]  
 )  
 self.wavelet\_scale = nn.ModuleList(  
 [\_ScaleModule([1, in\_channels \* 4, 1, 1], init\_scale=0.1) for \_ in range(self.wt\_levels)]  
 )  
  
 def forward(self, x):  
 # 前向传播过程  
 x\_ll\_in\_levels = [] # 存储低频信息  
 x\_h\_in\_levels = [] # 存储高频信息  
 shapes\_in\_levels = [] # 存储形状信息  
  
 curr\_x\_ll = x # 当前低频信号  
  
 # 小波变换过程  
 for i in range(self.wt\_levels):  
 curr\_shape = curr\_x\_ll.shape  
 shapes\_in\_levels.append(curr\_shape)  
 if (curr\_shape[2] % 2 > 0) or (curr\_shape[3] % 2 > 0):  
 curr\_pads = (0, curr\_shape[3] % 2, 0, curr\_shape[2] % 2) # 处理边界情况  
 curr\_x\_ll = F.pad(curr\_x\_ll, curr\_pads)  
  
 curr\_x = self.wt\_function(curr\_x\_ll) # 执行小波变换  
 curr\_x\_ll = curr\_x[:, :, 0, :, :] # 获取低频部分  
   
 # 处理高频部分  
 shape\_x = curr\_x.shape  
 curr\_x\_tag = curr\_x.reshape(shape\_x[0], shape\_x[1] \* 4, shape\_x[3], shape\_x[4])  
 curr\_x\_tag = self.wavelet\_scale[i](self.wavelet\_convs[i](curr\_x\_tag))  
 curr\_x\_tag = curr\_x\_tag.reshape(shape\_x)  
  
 x\_ll\_in\_levels.append(curr\_x\_tag[:, :, 0, :, :]) # 存储低频信息  
 x\_h\_in\_levels.append(curr\_x\_tag[:, :, 1:4, :, :]) # 存储高频信息  
  
 next\_x\_ll = 0 # 初始化下一个低频信号  
  
 # 逆小波变换过程  
 for i in range(self.wt\_levels - 1, -1, -1):  
 curr\_x\_ll = x\_ll\_in\_levels.pop() # 获取当前低频信息  
 curr\_x\_h = x\_h\_in\_levels.pop() # 获取当前高频信息  
 curr\_shape = shapes\_in\_levels.pop() # 获取当前形状信息  
  
 curr\_x\_ll = curr\_x\_ll + next\_x\_ll # 组合低频信息  
  
 curr\_x = torch.cat([curr\_x\_ll.unsqueeze(2), curr\_x\_h], dim=2) # 合并低频和高频信息  
 next\_x\_ll = self.iwt\_function(curr\_x) # 执行逆小波变换  
  
 next\_x\_ll = next\_x\_ll[:, :, :curr\_shape[2], :curr\_shape[3]] # 修剪输出  
  
 x\_tag = next\_x\_ll # 获取最终输出  
 assert len(x\_ll\_in\_levels) == 0 # 确保所有低频信息都已处理  
   
 x = self.base\_scale(self.base\_conv(x)) # 通过基础卷积层和缩放模块处理输入  
 x = x + x\_tag # 添加小波变换的输出  
   
 return x  
  
# 定义缩放模块  
class \_ScaleModule(nn.Module):  
 def \_\_init\_\_(self, dims, init\_scale=1.0):  
 super(\_ScaleModule, self).\_\_init\_\_()  
 self.dims = dims  
 self.weight = nn.Parameter(torch.ones(\*dims) \* init\_scale) # 初始化权重  
   
 def forward(self, x):  
 return torch.mul(self.weight, x) # 执行缩放操作  
```  
  
### 代码核心部分说明：  
1. \*\*小波滤波器创建\*\*：`create\_wavelet\_filter` 函数生成小波变换和逆变换所需的滤波器。  
2. \*\*小波变换和逆变换\*\*：`wavelet\_transform` 和 `inverse\_wavelet\_transform` 函数分别执行小波变换和逆变换。  
3. \*\*自定义函数\*\*：`WaveletTransform` 和 `InverseWaveletTransform` 类实现了小波变换的前向和反向传播。  
4. \*\*小波卷积层\*\*：`WTConv2d` 类实现了一个结合小波变换的卷积层，能够在卷积过程中同时处理低频和高频信息。  
5. \*\*缩放模块\*\*：`\_ScaleModule` 类用于对卷积输出进行缩放，提供了可训练的缩放因子。```

这个程序文件`wtconv2d.py`实现了一个基于小波变换的二维卷积层，主要用于深度学习中的图像处理任务。程序中使用了PyTorch库，结合了小波变换的数学原理，提供了对输入图像的多层次分析和重构。  
  
首先，文件导入了必要的库，包括PyTorch的核心模块和功能模块，以及用于小波变换的`pywt`库。程序定义了几个重要的函数和类。  
  
`create\_wavelet\_filter`函数用于创建小波变换的滤波器。它接受小波类型、输入和输出通道数，并返回用于小波分解和重构的滤波器。函数中通过`pywt.Wavelet`获取小波的高通和低通滤波器，并将其转换为PyTorch张量格式。  
  
`wavelet\_transform`和`inverse\_wavelet\_transform`函数分别实现了小波变换和逆小波变换。它们使用了PyTorch的卷积操作，将输入张量进行分解或重构。具体来说，小波变换将输入图像分解为低频和高频部分，而逆小波变换则将这些部分重构回原始图像。  
  
接下来，定义了`WaveletTransform`和`InverseWaveletTransform`两个类，分别实现了小波变换和逆小波变换的前向和反向传播功能。这些类继承自`torch.autograd.Function`，允许在计算图中进行自定义的前向和反向传播操作。  
  
`wavelet\_transform\_init`和`inverse\_wavelet\_transform\_init`函数用于初始化小波变换和逆小波变换的应用函数，返回一个可以直接调用的函数。  
  
`WTConv2d`类是这个文件的核心部分，继承自`nn.Module`，实现了一个自定义的卷积层。构造函数中，首先检查输入和输出通道数是否相等，然后初始化小波滤波器和基本卷积层。该类支持多层次的小波变换，通过`wavelet\_convs`和`wavelet\_scale`来处理不同层次的特征。  
  
在`forward`方法中，输入数据经过小波变换和卷积操作，逐层处理并最终重构。该方法还处理了输入的填充和步幅，以确保输出的尺寸符合预期。最后，经过基本卷积层和缩放模块的处理，返回最终的输出。  
  
`\_ScaleModule`类是一个简单的缩放模块，用于对输入张量进行缩放操作。它通过一个可学习的参数来调整输入的权重。  
  
整体来看，这个程序实现了一个结合小波变换的卷积层，能够有效地提取图像特征，并在深度学习模型中应用。通过小波变换，可以在不同的频率和空间尺度上分析图像，从而提高模型的表现。

```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 MF\_Attention(nn.Module):  
 """  
 自注意力机制（Self-Attention）实现，参考Transformer论文。  
 """  
 def \_\_init\_\_(self, dim, head\_dim=32, num\_heads=None, qkv\_bias=False,  
 attn\_drop=0., proj\_drop=0., proj\_bias=False):  
 super().\_\_init\_\_()  
  
 # 设置每个头的维度  
 self.head\_dim = head\_dim  
 # 缩放因子  
 self.scale = head\_dim \*\* -0.5  
  
 # 计算头的数量  
 self.num\_heads = num\_heads if num\_heads else dim // head\_dim  
 if self.num\_heads == 0:  
 self.num\_heads = 1  
   
 # 总的注意力维度  
 self.attention\_dim = self.num\_heads \* self.head\_dim  
  
 # 定义线性层用于计算Q、K、V  
 self.qkv = nn.Linear(dim, self.attention\_dim \* 3, bias=qkv\_bias)  
 # 定义注意力丢弃层  
 self.attn\_drop = nn.Dropout(attn\_drop)  
 # 定义输出的线性层  
 self.proj = nn.Linear(self.attention\_dim, dim, bias=proj\_bias)  
 # 定义输出丢弃层  
 self.proj\_drop = nn.Dropout(proj\_drop)  
  
 def forward(self, x):  
 # 获取输入的批量大小、高度、宽度和通道数  
 B, H, W, C = x.shape  
 N = H \* W # 计算总的序列长度  
  
 # 计算Q、K、V  
 qkv = self.qkv(x).reshape(B, N, 3, self.num\_heads, self.head\_dim).permute(2, 0, 3, 1, 4)  
 q, k, v = qkv.unbind(0) # 将Q、K、V分开  
  
 # 计算注意力得分  
 attn = (q @ k.transpose(-2, -1)) \* self.scale  
 attn = attn.softmax(dim=-1) # 应用softmax  
 attn = self.attn\_drop(attn) # 应用丢弃  
  
 # 计算输出  
 x = (attn @ v).transpose(1, 2).reshape(B, H, W, self.attention\_dim)  
 x = self.proj(x) # 投影到原始维度  
 x = self.proj\_drop(x) # 应用丢弃  
 return x # 返回输出  
  
class MetaFormerBlock(nn.Module):  
 """  
 MetaFormer块的实现。  
 """  
 def \_\_init\_\_(self, dim,  
 token\_mixer=nn.Identity, mlp=Mlp,  
 norm\_layer=partial(LayerNormWithoutBias, eps=1e-6),  
 drop=0., drop\_path=0.,  
 layer\_scale\_init\_value=None, res\_scale\_init\_value=None  
 ):  
 super().\_\_init\_\_()  
  
 # 第一层归一化  
 self.norm1 = norm\_layer(dim)  
 # 令牌混合器  
 self.token\_mixer = token\_mixer(dim=dim, drop=drop)  
 # 路径丢弃  
 self.drop\_path1 = DropPath(drop\_path) if drop\_path > 0. else nn.Identity()  
 # 层缩放  
 self.layer\_scale1 = Scale(dim=dim, init\_value=layer\_scale\_init\_value) \  
 if layer\_scale\_init\_value else nn.Identity()  
 self.res\_scale1 = Scale(dim=dim, init\_value=res\_scale\_init\_value) \  
 if res\_scale\_init\_value else nn.Identity()  
  
 # 第二层归一化  
 self.norm2 = norm\_layer(dim)  
 # MLP  
 self.mlp = mlp(dim=dim, drop=drop)  
 # 路径丢弃  
 self.drop\_path2 = DropPath(drop\_path) if drop\_path > 0. else nn.Identity()  
 # 层缩放  
 self.layer\_scale2 = Scale(dim=dim, init\_value=layer\_scale\_init\_value) \  
 if layer\_scale\_init\_value else nn.Identity()  
 self.res\_scale2 = Scale(dim=dim, init\_value=res\_scale\_init\_value) \  
 if res\_scale\_init\_value else nn.Identity()  
   
 def forward(self, x):  
 # 进行维度转换  
 x = x.permute(0, 2, 3, 1)  
 # 第一部分的前向传播  
 x = self.res\_scale1(x) + \  
 self.layer\_scale1(  
 self.drop\_path1(  
 self.token\_mixer(self.norm1(x))  
 )  
 )  
 # 第二部分的前向传播  
 x = self.res\_scale2(x) + \  
 self.layer\_scale2(  
 self.drop\_path2(  
 self.mlp(self.norm2(x))  
 )  
 )  
 return x.permute(0, 3, 1, 2) # 返回到原始维度  
```  
  
### 代码核心部分说明：  
1. \*\*MF\_Attention类\*\*：实现了自注意力机制，包含了计算Q、K、V的线性层，注意力得分的计算，以及输出的投影。  
2. \*\*MetaFormerBlock类\*\*：实现了MetaFormer的基本构建块，包含了归一化、令牌混合、MLP等组件，支持残差连接和路径丢弃。   
  
这些类是构建MetaFormer模型的基础，提供了注意力机制和基本的块结构。```

这个程序文件 `metaformer.py` 实现了一些用于构建 MetaFormer 模型的基础组件，主要包括各种层和模块的定义。以下是对文件中各个部分的详细说明。  
  
首先，文件导入了一些必要的库，包括 `torch` 和 `torch.nn`，以及一些来自 `timm` 库的功能。这些库为构建深度学习模型提供了基础。  
  
接下来，定义了一些基本的模块：  
  
1. \*\*Scale\*\* 类用于按元素乘法缩放输入向量。它接受一个维度和一个初始值，创建一个可训练的参数用于缩放。  
  
2. \*\*SquaredReLU\*\* 和 \*\*StarReLU\*\* 是两种激活函数的实现。SquaredReLU 计算 ReLU 的平方，而 StarReLU 则结合了缩放和偏置的 ReLU 变体。  
  
3. \*\*MF\_Attention\*\* 类实现了基本的自注意力机制，类似于 Transformer 中的自注意力。它通过线性变换生成查询、键和值，然后计算注意力权重并应用于值。  
  
4. \*\*RandomMixing\*\* 类实现了一种随机混合机制，通过一个随机矩阵对输入进行变换。  
  
5. \*\*LayerNormGeneral\*\* 和 \*\*LayerNormWithoutBias\*\* 提供了不同形式的层归一化，支持多种输入形状和可选的缩放与偏置。  
  
6. \*\*SepConv\*\* 类实现了分离卷积，这是一种高效的卷积方式，常用于轻量级网络中。  
  
7. \*\*Pooling\*\* 类实现了一种池化操作，特别适用于 PoolFormer 模型。  
  
8. \*\*Mlp\*\* 类实现了多层感知机（MLP），包括线性层、激活函数和 dropout。  
  
9. \*\*ConvolutionalGLU\*\* 类实现了一种卷积门控线性单元（GLU），结合了卷积和激活函数。  
  
接下来，定义了两个主要的 MetaFormer 块：  
  
1. \*\*MetaFormerBlock\*\* 类实现了一个标准的 MetaFormer 块，包含归一化、令牌混合、MLP 和残差连接。它使用了前面定义的组件来构建块的结构。  
  
2. \*\*MetaFormerCGLUBlock\*\* 类是另一个 MetaFormer 块的实现，使用了卷积门控线性单元作为 MLP 部分，其他结构与 MetaFormerBlock 类似。  
  
整体来看，这个文件实现了构建 MetaFormer 模型所需的多个基础组件，提供了灵活的层和模块，可以用于构建不同的网络架构。每个模块都具有良好的可扩展性和可重用性，适合用于各种深度学习任务。

### 整体功能和构架概括  
  
该项目包含多个深度学习模型和组件，主要用于计算机视觉任务。整体架构设计灵活，支持多种模型结构，包括 Efficient ViT、Swin Transformer、基于小波变换的卷积层以及 MetaFormer 模型。每个文件实现了特定的功能模块，能够相互配合以构建复杂的视觉模型。以下是各个文件的功能概述：  
  
- \*\*efficientViT.py\*\*：实现了 Efficient ViT 模型，结合了卷积操作和注意力机制，适用于图像处理任务。  
- \*\*SwinTransformer.py\*\*：实现了 Swin Transformer 模型，采用局部窗口注意力机制，适合处理高分辨率图像。  
- \*\*wtconv2d.py\*\*：实现了基于小波变换的二维卷积层，能够在不同频率和空间尺度上分析图像特征。  
- \*\*metaformer.py\*\*：实现了 MetaFormer 模型的基础组件，包括自注意力机制、激活函数、归一化层等，支持灵活的网络构建。  
  
### 文件功能整理表  
  
| 文件名 | 功能描述 |  
|-----------------------|--------------------------------------------------------|  
| efficientViT.py | 实现 Efficient ViT 模型，结合卷积和注意力机制，用于图像处理任务。 |  
| SwinTransformer.py | 实现 Swin Transformer 模型，采用局部窗口注意力机制，适合高分辨率图像。 |  
| wtconv2d.py | 实现基于小波变换的二维卷积层，分析图像特征的多层次方法。 |  
| metaformer.py | 实现 MetaFormer 模型的基础组件，支持自注意力、激活函数和归一化层等。 |  
  
这些文件共同构成了一个灵活且高效的深度学习框架，适用于各种计算机视觉任务。