# 改进yolo11-GhostDynamicConv等200+全套创新点大全：建筑物内部物体检测系统源码＆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，建筑物的复杂性和多样性不断增加，建筑内部的物体检测变得愈发重要。传统的人工检测方法不仅效率低下，而且容易受到人为因素的影响，导致检测结果的不准确。因此，基于深度学习的自动化物体检测技术逐渐成为研究的热点，尤其是在建筑物内部环境中。YOLO（You Only Look Once）系列模型因其高效的实时检测能力和良好的准确性，成为了物体检测领域的主流选择。  
  
本研究旨在基于改进的YOLOv11模型，构建一个高效的建筑物内部物体检测系统。该系统将能够识别和定位建筑内部的多种物体，包括但不限于门、窗、管道、灯具、灭火器等，共计47个类别。这些物体在建筑物的安全、维护和管理中扮演着重要角色，准确的检测能够为建筑物的智能管理提供数据支持，提升管理效率。  
  
在数据集方面，本研究使用了包含1400张经过精细标注的图像，涵盖了建筑内部的多种结构和设备。这些图像经过了多种预处理和增强技术，确保了模型训练的多样性和鲁棒性。通过利用先进的计算机视觉技术，系统将能够实时监测建筑内部的物体状态，及时发现潜在的安全隐患，从而为建筑物的安全管理提供有力保障。  
  
此外，随着智能建筑和物联网技术的发展，建筑物内部的物体检测系统将与其他智能系统相结合，实现更高层次的自动化和智能化管理。研究的成果不仅能够推动建筑物管理技术的进步，也将为相关领域的研究提供新的思路和方法，具有重要的理论意义和实际应用价值。

## 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的建筑物内部物体检测系统，所使用的数据集主题为“ML const”，其设计目的是为了增强在复杂建筑环境中对多种物体的识别能力。该数据集包含47个类别，涵盖了建筑内部的各种元素，提供了丰富的样本以支持模型的训练和验证。具体类别包括但不限于：门、窗框、墙壁、天花板、地板、以及各种电气和消防设备，如CCTV摄像头、火灾灭火器和电源面板等。这些类别的选择不仅反映了建筑内部的常见构件，也考虑到了现代建筑设计中的智能化和安全性需求。  
  
数据集中每个类别的样本数量经过精心挑选，以确保模型在训练过程中能够接触到多样化的场景和物体。这种多样性对于提高模型的泛化能力至关重要，尤其是在面对不同建筑风格和布局时。数据集中的“人”类别也为模型提供了动态元素，使其能够在实际应用中更好地适应人流量变化和人机交互场景。  
  
此外，数据集还包含了一些特定的建筑构件，如声学面板、混凝土梁、柱子及其不同的构造形式（如模板和钢筋），这些元素的细致分类有助于模型在复杂环境中进行精确的物体检测。通过使用该数据集进行训练，改进后的YOLOv11模型将能够更准确地识别和定位建筑内部的各种物体，从而为智能建筑管理、安防监控及维护保养提供强有力的技术支持。

## 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  
  
class AGLU(nn.Module):  
 """AGLU（Adaptive Gated Linear Unit）激活函数模块，来源于 https://github.com/kostas1515/AGLU。"""  
  
 def \_\_init\_\_(self, device=None, dtype=None) -> None:  
 """初始化AGLU激活函数模块。  
   
 参数：  
 device: 指定张量的设备（如CPU或GPU）。  
 dtype: 指定张量的数据类型（如float32或float64）。  
 """  
 super().\_\_init\_\_()  
 # 使用Softplus作为基础激活函数，beta设置为-1.0  
 self.act = nn.Softplus(beta=-1.0)  
 # 初始化lambda参数，并将其设为可学习参数  
 self.lambd = nn.Parameter(nn.init.uniform\_(torch.empty(1, device=device, dtype=dtype)))   
 # 初始化kappa参数，并将其设为可学习参数  
 self.kappa = nn.Parameter(nn.init.uniform\_(torch.empty(1, device=device, dtype=dtype)))   
  
 def forward(self, x: torch.Tensor) -> torch.Tensor:  
 """计算AGLU激活函数的前向传播。  
   
 参数：  
 x: 输入张量，通常是来自前一层的输出。  
   
 返回：  
 返回经过AGLU激活函数处理后的张量。  
 """  
 # 将lambda参数限制在最小值0.0001，避免除零错误  
 lam = torch.clamp(self.lambd, min=0.0001)  
 # 计算AGLU激活函数的输出  
 return torch.exp((1 / lam) \* self.act((self.kappa \* x) - torch.log(lam)))  
```  
  
### 代码核心部分分析：  
1. \*\*AGLU类\*\*：这是一个自定义的激活函数模块，继承自`nn.Module`，用于实现AGLU激活函数。  
2. \*\*初始化方法`\_\_init\_\_`\*\*：  
 - 初始化了两个可学习的参数`lambd`和`kappa`，这两个参数的值在训练过程中会被优化。  
 - 使用`Softplus`作为基础激活函数，这是一种平滑的激活函数，能够避免ReLU的死神经元问题。  
3. \*\*前向传播方法`forward`\*\*：  
 - 接收输入张量`x`，并计算AGLU激活函数的输出。  
 - `torch.clamp`确保`lambd`的值不会过小，以避免在计算中出现数值不稳定。  
 - 最终返回经过AGLU激活函数处理后的结果。  
  
### 总结：  
AGLU激活函数结合了可学习的参数，能够自适应地调整其输出特性，适用于深度学习模型中的非线性变换。```

这个文件名为 `activation.py`，主要定义了一个名为 `AGLU` 的激活函数模块。它是基于 PyTorch 框架构建的，包含了一个统一的激活函数，旨在提高神经网络的性能。  
  
首先，文件导入了 PyTorch 的核心库 `torch` 和神经网络模块 `torch.nn`。接着，定义了一个名为 `AGLU` 的类，该类继承自 `nn.Module`，这是 PyTorch 中所有神经网络模块的基类。  
  
在 `AGLU` 类的初始化方法 `\_\_init\_\_` 中，首先调用了父类的初始化方法。然后，定义了一个激活函数 `self.act`，使用了 `nn.Softplus`，这是一个平滑的激活函数，类似于 ReLU，但在负值区域也有非零输出。接下来，定义了两个可学习的参数 `lambd` 和 `kappa`，它们分别是激活函数的两个关键参数。这两个参数使用 `nn.Parameter` 包装，并通过均匀分布初始化，确保它们在训练过程中可以被优化。  
  
在 `forward` 方法中，定义了前向传播的计算过程。输入参数 `x` 是一个张量，表示神经网络的输入。首先，使用 `torch.clamp` 函数对 `lambd` 参数进行限制，确保其最小值为 0.0001，以避免在后续计算中出现除以零的情况。然后，计算并返回激活函数的输出，公式为 `torch.exp((1 / lam) \* self.act((self.kappa \* x) - torch.log(lam)))`。这个公式结合了 `Softplus` 激活函数和 `lambd`、`kappa` 参数的影响，生成了新的激活值。  
  
总的来说，这个文件实现了一个新的激活函数模块 `AGLU`，它通过引入可学习的参数来增强网络的表达能力，适用于各种深度学习任务。

```以下是经过简化和注释的核心代码部分，保留了模型的主要结构和功能：  
  
```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  
  
 # 定义卷积层用于生成查询、键、值  
 self.q = nn.Conv2d(dim, num\_heads \* key\_dim, kernel\_size=1)  
 self.k = nn.Conv2d(dim, num\_heads \* key\_dim, kernel\_size=1)  
 self.v = nn.Conv2d(dim, num\_heads \* attn\_ratio \* key\_dim, kernel\_size=1)  
  
 # 注意力偏置  
 self.attention\_biases = nn.Parameter(torch.zeros(num\_heads, resolution \*\* 2, resolution \*\* 2))  
 self.register\_buffer('attention\_bias\_idxs', self.\_create\_attention\_bias\_idxs(resolution))  
  
 def \_create\_attention\_bias\_idxs(self, resolution):  
 """创建注意力偏置索引"""  
 points = list(itertools.product(range(resolution), range(resolution)))  
 idxs = []  
 for p1 in points:  
 for p2 in points:  
 offset = (abs(p1[0] - p2[0]), abs(p1[1] - p2[1]))  
 idxs.append(offset)  
 return torch.LongTensor(idxs).view(len(points), len(points))  
  
 def forward(self, x):  
 """前向传播"""  
 B, C, H, W = x.shape # 获取输入的形状  
 q = self.q(x).view(B, self.num\_heads, -1, self.resolution \*\* 2).permute(0, 1, 3, 2) # 查询  
 k = self.k(x).view(B, self.num\_heads, -1, self.resolution \*\* 2).permute(0, 1, 2, 3) # 键  
 v = self.v(x).view(B, self.num\_heads, -1, self.resolution \*\* 2).permute(0, 1, 3, 2) # 值  
  
 # 计算注意力  
 attn = (q @ k) \* self.scale + self.attention\_biases[:, self.attention\_bias\_idxs]  
 attn = attn.softmax(dim=-1) # 归一化  
  
 # 计算输出  
 x = (attn @ v).permute(0, 1, 3, 2).view(B, -1, self.resolution, self.resolution)  
 return x  
  
class EfficientFormerV2(nn.Module):  
 """EfficientFormer V2模型"""  
 def \_\_init\_\_(self, layers, embed\_dims, num\_classes=1000):  
 super().\_\_init\_\_()  
 self.num\_classes = num\_classes  
 self.patch\_embed = nn.Conv2d(3, embed\_dims[0], kernel\_size=3, stride=2, padding=1) # 初始嵌入层  
  
 # 构建网络层  
 self.network = nn.ModuleList()  
 for i in range(len(layers)):  
 stage = self.\_create\_stage(embed\_dims[i], layers[i])  
 self.network.append(stage)  
  
 def \_create\_stage(self, dim, layers):  
 """创建网络的每一层"""  
 blocks = []  
 for \_ in range(layers):  
 blocks.append(Attention4D(dim=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) # 通过每一层  
 return x  
  
# 示例用法  
if \_\_name\_\_ == '\_\_main\_\_':  
 inputs = torch.randn((1, 3, 640, 640)) # 创建输入张量  
 model = EfficientFormerV2(layers=[2, 2, 2], embed\_dims=[32, 64, 128]) # 初始化模型  
 res = model(inputs) # 前向传播  
 print(res.shape) # 输出结果的形状  
```  
  
### 代码说明：  
1. \*\*Attention4D类\*\*：实现了一个4D注意力机制，包含查询、键、值的生成以及注意力计算。  
2. \*\*EfficientFormerV2类\*\*：构建了一个EfficientFormer V2模型，包含多个注意力模块。  
3. \*\*前向传播\*\*：在`forward`方法中，输入经过嵌入层和多个注意力模块，最终输出特征图。  
  
这段代码是EfficientFormer V2模型的核心部分，展示了如何使用注意力机制来处理输入数据。```

这个程序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，如图像分类。该模型是对 EfficientFormer 的改进版本，具有更高的效率和性能。文件中包含了模型的各个组件和结构，以及不同规模的模型实例化函数。  
  
首先，文件中定义了一些模型的超参数，包括不同规模的模型的宽度和深度。这些参数以字典的形式存储，便于后续使用。`EfficientFormer\_width` 和 `EfficientFormer\_depth` 分别定义了不同规模（如 S0、S1、S2 和 L）的模型在每一层的通道数和层数。  
  
接下来，文件中定义了多个类，这些类构成了模型的基本单元。`Attention4D` 类实现了一个四维注意力机制，支持对输入特征图进行注意力计算。该类的构造函数中定义了多个卷积层和参数，用于生成查询、键和值的特征图，并计算注意力权重。  
  
`LGQuery` 类用于生成局部和全局的查询特征。`Attention4DDownsample` 类则结合了注意力机制和下采样操作，进一步增强了模型的特征提取能力。  
  
`Embedding` 类用于对输入图像进行嵌入处理，将其转换为适合后续处理的特征表示。`Mlp` 类实现了多层感知机（MLP），用于特征的非线性变换。  
  
`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。它们的主要功能是对输入特征进行进一步处理，提取更高层次的特征。  
  
`eformer\_block` 函数用于构建模型的基本块，结合了不同的网络结构和参数设置。`EfficientFormerV2` 类则是整个模型的核心，负责将各个组件组合在一起，形成完整的网络结构。  
  
在模型的初始化过程中，用户可以选择不同的层数、嵌入维度、下采样策略等参数。模型的前向传播函数 `forward` 接收输入图像，经过嵌入层和多个网络块的处理，最终输出特征。  
  
此外，文件中还定义了多个函数用于创建不同规模的 EfficientFormerV2 模型实例（如 `efficientformerv2\_s0`、`efficientformerv2\_s1` 等），并支持加载预训练权重。  
  
最后，在 `\_\_main\_\_` 部分，程序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，输出每个模型的特征图尺寸。这部分代码可以用于快速验证模型的构建和运行是否正常。  
  
总体而言，这个程序文件提供了一个灵活且高效的深度学习模型框架，适用于图像分类等任务，具有良好的可扩展性和可配置性。

```以下是代码中最核心的部分，并附上详细的中文注释：  
  
```python  
import torch  
import torch.nn as nn  
  
class ReverseFunction(torch.autograd.Function):  
 @staticmethod  
 def forward(ctx, run\_functions, alpha, \*args):  
 # 保存运行的函数和alpha参数  
 ctx.run\_functions = run\_functions  
 ctx.alpha = alpha  
 ctx.preserve\_rng\_state = True # 保持随机数生成器的状态  
  
 # 获取输入参数  
 assert len(args) == 5  
 [x, c0, c1, c2, c3] = args  
   
 # 计算每一层的输出  
 with torch.no\_grad():  
 c0 = run\_functions[0](x, c1) + c0 \* alpha[0]  
 c1 = run\_functions[1](c0, c2) + c1 \* alpha[1]  
 c2 = run\_functions[2](c1, c3) + c2 \* alpha[2]  
 c3 = run\_functions[3](c2, None) + c3 \* alpha[3]  
   
 # 保存输入以便在反向传播时使用  
 ctx.save\_for\_backward(x, c0, c1, c2, c3)  
 return x, c0, c1, c2, c3  
  
 @staticmethod  
 def backward(ctx, \*grad\_outputs):  
 # 获取保存的张量  
 x, c0, c1, c2, c3 = ctx.saved\_tensors  
 run\_functions = ctx.run\_functions  
 alpha = ctx.alpha  
   
 # 反向传播计算梯度  
 g3\_up = grad\_outputs[4]  
 g3\_left = g3\_up \* alpha[3] # 使用alpha进行缩放  
   
 # 计算c3的梯度  
 oup3 = run\_functions[3](c2, None)  
 torch.autograd.backward(oup3, g3\_up, retain\_graph=True)  
   
 # 计算c2的梯度  
 g2\_up = grad\_outputs[3] + c2.grad  
 g2\_left = g2\_up \* alpha[2]  
   
 # 继续反向传播  
 oup2 = run\_functions[2](c1, c3)  
 torch.autograd.backward(oup2, g2\_up, retain\_graph=True)  
   
 # 依此类推，计算c1和c0的梯度  
 g1\_up = grad\_outputs[2] + c1.grad  
 g1\_left = g1\_up \* alpha[1]  
   
 oup1 = run\_functions[1](c0, c2)  
 torch.autograd.backward(oup1, g1\_up, retain\_graph=True)  
   
 g0\_up = grad\_outputs[0] + c0.grad  
 g0\_left = g0\_up \* alpha[0]  
   
 # 返回所有梯度  
 return None, None, g0\_left, g1\_left, g2\_left, g3\_left  
  
class SubNet(nn.Module):  
 def \_\_init\_\_(self, channels, layers, kernel, first\_col, save\_memory) -> None:  
 super().\_\_init\_\_()  
 # 初始化子网络的参数  
 self.alpha0 = nn.Parameter(torch.ones((1, channels[0], 1, 1)), requires\_grad=True)  
 self.alpha1 = nn.Parameter(torch.ones((1, channels[1], 1, 1)), requires\_grad=True)  
 self.alpha2 = nn.Parameter(torch.ones((1, channels[2], 1, 1)), requires\_grad=True)  
 self.alpha3 = nn.Parameter(torch.ones((1, channels[3], 1, 1)), requires\_grad=True)  
  
 # 创建每一层  
 self.level0 = Level(0, channels, layers, kernel, first\_col)  
 self.level1 = Level(1, channels, layers, kernel, first\_col)  
 self.level2 = Level(2, channels, layers, kernel, first\_col)  
 self.level3 = Level(3, channels, layers, kernel, first\_col)  
  
 def forward(self, \*args):  
 # 根据内存管理选择前向传播方式  
 if self.save\_memory:  
 return self.\_forward\_reverse(\*args)  
 else:  
 return self.\_forward\_nonreverse(\*args)  
  
class RevCol(nn.Module):  
 def \_\_init\_\_(self, kernel='C2f', channels=[32, 64, 96, 128], layers=[2, 3, 6, 3], num\_subnet=5, save\_memory=True) -> None:  
 super().\_\_init\_\_()  
 self.num\_subnet = num\_subnet  
 self.channels = channels  
 self.layers = layers  
  
 # 初始化网络的stem部分  
 self.stem = Conv(3, channels[0], k=4, s=4, p=0)  
  
 # 创建多个子网络  
 for i in range(num\_subnet):  
 first\_col = (i == 0)  
 self.add\_module(f'subnet{str(i)}', SubNet(channels, layers, kernel, first\_col, save\_memory=save\_memory))  
  
 def forward(self, x):  
 # 前向传播  
 c0, c1, c2, c3 = 0, 0, 0, 0  
 x = self.stem(x)   
 for i in range(self.num\_subnet):  
 c0, c1, c2, c3 = getattr(self, f'subnet{str(i)}')(x, c0, c1, c2, c3)   
 return [c0, c1, c2, c3]  
```  
  
### 代码注释说明  
1. \*\*ReverseFunction\*\*: 这个类实现了自定义的反向传播逻辑。`forward`方法计算前向传播的输出并保存必要的状态，而`backward`方法则计算梯度。  
2. \*\*SubNet\*\*: 这个类代表一个子网络，包含多个层级。根据是否需要节省内存，选择不同的前向传播方法。  
3. \*\*RevCol\*\*: 这个类是整个网络的主要结构，负责初始化网络的stem部分和多个子网络，并实现前向传播。  
  
以上是代码的核心部分和详细注释，帮助理解其功能和实现逻辑。```

这个程序文件 `revcol.py` 是一个使用 PyTorch 框架实现的深度学习模型，主要用于图像处理任务。它包含了多个类和函数，主要功能是构建一个具有反向传播功能的神经网络结构。以下是对代码的详细讲解。  
  
首先，文件导入了必要的 PyTorch 模块和自定义模块。`torch` 是 PyTorch 的核心库，`torch.nn` 提供了构建神经网络的基础组件，`torch.distributed` 可能用于分布式训练。自定义模块 `Conv`、`C2f`、`C3`、`C3Ghost` 和 `C3k2` 可能是实现特定卷积操作或网络块的类。  
  
接下来，定义了一些辅助函数。`get\_gpu\_states` 用于获取指定 GPU 设备的随机数生成器状态，`get\_gpu\_device` 用于提取输入张量所在的 GPU 设备，`set\_device\_states` 用于设置 CPU 和 GPU 的随机数状态。`detach\_and\_grad` 函数用于从输入中分离出张量并设置其需要梯度的属性。`get\_cpu\_and\_gpu\_states` 则是获取当前 CPU 和 GPU 的随机数状态。  
  
核心部分是 `ReverseFunction` 类，它继承自 `torch.autograd.Function`，实现了自定义的前向和反向传播逻辑。在 `forward` 方法中，模型的各个层（由 `run\_functions` 参数传入）依次进行前向计算，并根据给定的 `alpha` 值进行加权。这个方法还保存了输入和中间计算结果，以便在反向传播时使用。  
  
在 `backward` 方法中，使用保存的中间结果和梯度信息，依次计算每一层的梯度。这里涉及到对 GPU 状态的管理和自动混合精度的使用，以提高计算效率。  
  
接下来是 `Fusion`、`Level` 和 `SubNet` 类的定义。`Fusion` 类用于实现不同层之间的融合操作，`Level` 类则代表网络中的一个层级，包含了融合和卷积操作。`SubNet` 类则是一个子网络，包含多个层级的组合，并根据是否保存内存的需求选择不同的前向传播策略。  
  
最后，`RevCol` 类是整个模型的封装，初始化时构建了多个子网络，并定义了输入的处理流程。在 `forward` 方法中，输入经过 `stem` 层处理后，依次通过每个子网络进行计算，最终返回多个输出。  
  
整体来看，这个文件实现了一个复杂的神经网络结构，利用反向传播和混合精度训练的技术，旨在提高训练效率和模型性能。

```以下是经过简化和注释的代码，保留了核心部分，并对每个重要部分进行了详细的中文注释：  
  
```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  
  
 # 定义查询、键、值的线性变换  
 self.qkv = nn.Linear(dim, dim \* 3, bias=qkv\_bias)  
 self.attn\_drop = nn.Dropout(attn\_drop) # 注意力的Dropout  
 self.proj = nn.Linear(dim, dim) # 输出的线性变换  
 self.proj\_drop = nn.Dropout(proj\_drop) # 输出的Dropout  
  
 def forward(self, x, mask=None):  
 """ 前向传播函数。 """  
 B\_, N, C = x.shape # 获取输入的形状  
 qkv = self.qkv(x).reshape(B\_, N, 3, self.num\_heads, C // self.num\_heads).permute(2, 0, 3, 1, 4)  
 q, k, v = qkv[0], qkv[1], qkv[2] # 分离查询、键、值  
  
 attn = (q @ k.transpose(-2, -1)) # 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 SwinTransformerBlock(nn.Module):  
 """ Swin Transformer块。 """  
  
 def \_\_init\_\_(self, dim, num\_heads, window\_size=7, shift\_size=0, mlp\_ratio=4., drop=0., attn\_drop=0.):  
 super().\_\_init\_\_()  
 self.attn = WindowAttention(dim, window\_size, num\_heads, attn\_drop=attn\_drop) # 注意力模块  
 self.mlp = Mlp(in\_features=dim, hidden\_features=int(dim \* mlp\_ratio), drop=drop) # MLP模块  
  
 def forward(self, x):  
 """ 前向传播函数。 """  
 x = self.attn(x) # 通过注意力模块  
 x = x + self.mlp(x) # 加上MLP的输出  
 return x  
  
  
class SwinTransformer(nn.Module):  
 """ Swin Transformer主模型。 """  
  
 def \_\_init\_\_(self, depths=[2, 2, 6, 2], num\_heads=[3, 6, 12, 24], embed\_dim=96):  
 super().\_\_init\_\_()  
 self.layers = nn.ModuleList() # 存储每一层  
 for i in range(len(depths)):  
 layer = SwinTransformerBlock(  
 dim=int(embed\_dim \* 2 \*\* i),  
 num\_heads=num\_heads[i],  
 window\_size=7  
 )  
 self.layers.append(layer) # 添加层到模型中  
  
 def forward(self, x):  
 """ 前向传播函数。 """  
 for layer in self.layers:  
 x = layer(x) # 逐层传递输入  
 return x  
  
  
def SwinTransformer\_Tiny(weights=''):  
 """ 创建一个小型的Swin Transformer模型。 """  
 model = SwinTransformer(depths=[2, 2, 6, 2], num\_heads=[3, 6, 12, 24])  
 if weights:  
 model.load\_state\_dict(torch.load(weights)) # 加载权重  
 return model  
```  
  
### 代码注释说明：  
1. \*\*Mlp类\*\*：实现了一个简单的多层感知机，包含两个线性层和一个激活函数，支持Dropout。  
2. \*\*WindowAttention类\*\*：实现了窗口自注意力机制，计算输入的注意力权重并生成输出。  
3. \*\*SwinTransformerBlock类\*\*：实现了Swin Transformer的基本块，包含一个注意力层和一个MLP层。  
4. \*\*SwinTransformer类\*\*：构建了整个Swin Transformer模型，包含多个Swin Transformer块。  
5. \*\*SwinTransformer\_Tiny函数\*\*：创建一个小型的Swin Transformer模型，并可选择加载预训练权重。  
  
以上是对代码的简化和注释，保留了核心结构和功能。```

这个程序文件实现了Swin Transformer模型的结构，Swin Transformer是一种用于计算机视觉任务的变换器架构。代码主要由多个类组成，每个类负责模型的不同部分，包括多层感知机、窗口注意力机制、Swin Transformer块、补丁合并层、基本层、补丁嵌入层以及最终的Swin Transformer模型。  
  
首先，`Mlp`类定义了一个多层感知机，包含两个线性层和一个激活函数（默认为GELU），同时支持dropout。`window\_partition`和`window\_reverse`函数用于将输入特征图划分为窗口和将窗口合并回特征图。  
  
`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。它通过线性变换计算查询、键和值，并通过softmax计算注意力权重，最后将注意力应用于值。  
  
`SwinTransformerBlock`类是Swin Transformer的基本构建块，包含窗口注意力和前馈网络。它支持循环移位，以实现跨窗口的注意力计算。  
  
`PatchMerging`类用于将特征图的补丁合并，减少空间维度并增加通道数。`BasicLayer`类表示Swin Transformer的一个基本层，包含多个Swin Transformer块，并在必要时进行下采样。  
  
`PatchEmbed`类负责将输入图像划分为补丁并进行嵌入。`SwinTransformer`类是整个模型的核心，负责构建各个层并处理输入数据。它还支持绝对位置嵌入、dropout和随机深度等功能。  
  
最后，`update\_weight`函数用于更新模型权重，`SwinTransformer\_Tiny`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，这个文件提供了Swin Transformer的完整实现，适用于各种计算机视觉任务，如图像分类、目标检测等。

### 整体功能和构架概括  
  
该项目包含多个深度学习模型的实现，主要用于计算机视觉任务。每个文件实现了不同的模型结构和功能，旨在提高图像处理的效率和性能。具体来说：  
  
- \*\*activation.py\*\*：实现了一个自定义的激活函数模块 `AGLU`，通过引入可学习的参数来增强神经网络的表达能力。  
- \*\*EfficientFormerV2.py\*\*：构建了一个高效的图像分类模型 `EfficientFormerV2`，采用了多种卷积和注意力机制，优化了模型的计算效率和性能。  
- \*\*revcol.py\*\*：实现了一个具有反向传播功能的神经网络结构，利用自定义的前向和反向传播逻辑，支持高效的训练过程。  
- \*\*SwinTransformer.py\*\*：实现了Swin Transformer模型，采用窗口注意力机制，适用于各种计算机视觉任务，具有较强的特征提取能力。  
  
### 文件功能整理表  
  
| 文件名 | 功能描述 |  
|--------------------------|--------------------------------------------------------------------------|  
| activation.py | 实现自定义激活函数模块 `AGLU`，增强神经网络的表达能力。 |  
| EfficientFormerV2.py | 构建高效的图像分类模型 `EfficientFormerV2`，结合卷积和注意力机制。 |  
| revcol.py | 实现具有反向传播功能的神经网络结构，支持高效的训练过程。 |  
| SwinTransformer.py | 实现Swin Transformer模型，采用窗口注意力机制，适用于计算机视觉任务。 |  
  
这些文件共同构成了一个多样化的深度学习框架，适用于不同的图像处理和计算机视觉任务，具有良好的可扩展性和灵活性。