# 改进yolo11-ASF-P2等200+全套创新点大全：小鼠实例分割系统源码＆数据集全套

## 3.背景

研究背景与意义  
  
在生物医学研究中，小鼠作为一种重要的实验动物，广泛应用于药物开发、疾病模型研究及行为学实验等领域。随着科学技术的进步，传统的观察和记录小鼠行为的方法逐渐显露出其局限性，尤其是在数据收集的效率和准确性方面。因此，开发一种高效、准确的小鼠实例分割系统显得尤为重要。基于深度学习的计算机视觉技术，尤其是YOLO（You Only Look Once）系列模型，因其在实时目标检测中的优越性能，成为了研究者们的关注焦点。  
  
本研究旨在基于改进的YOLOv11模型，构建一个专门针对小鼠行为分析的实例分割系统。该系统将利用一个包含3200张图像的数据集，专注于小鼠这一单一类别的检测与分割。通过对小鼠行为的精准识别与分析，研究者能够更好地理解小鼠在不同实验条件下的反应，从而为药物测试和行为学研究提供更加可靠的数据支持。  
  
此外，实例分割技术的应用，不仅能够提升小鼠行为分析的精度，还能为后续的自动化数据处理和分析提供基础。通过对小鼠行为的细致分割，研究者可以获取更多关于小鼠活动模式的信息，这对于揭示其生理和心理状态具有重要意义。随着数据集的不断扩展和模型的持续优化，基于YOLOv11的小鼠实例分割系统有望在生物医学研究中发挥更大的作用，推动相关领域的进步与发展。通过这一系统的构建与应用，研究者将能够在更大程度上实现对小鼠行为的自动化监测和分析，进而提升实验的效率和结果的可靠性。

## 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名为“Hotplate\_behavior\_annotations\_new”，旨在为改进YOLOv11的小鼠实例分割系统提供必要的训练数据。该数据集专注于小鼠在热板实验中的行为表现，旨在通过高质量的标注数据，帮助研究人员更好地理解小鼠的行为模式以及其在不同实验条件下的反应。数据集中包含了大量小鼠在热板上的行为图像，这些图像经过精确的标注，确保每一只小鼠的实例都能被准确识别和分割。  
  
在类别方面，该数据集设定了单一类别，即“Mouse”，这意味着所有的标注均围绕小鼠这一生物对象展开。通过聚焦于这一特定类别，研究团队能够深入分析小鼠在热板实验中的各种行为表现，包括但不限于小鼠的移动、停留时间及其对热源的反应等。这种细致的标注不仅有助于提升YOLOv11在小鼠实例分割任务中的性能，也为后续的行为分析提供了坚实的数据基础。  
  
数据集的构建过程严格遵循科学研究的标准，确保了数据的真实性和可靠性。每一幅图像都经过仔细的审核和标注，确保在训练过程中能够提供高质量的输入。这一数据集的独特性在于其专注于小鼠行为的细节，使得模型在处理与小鼠相关的生物医学研究时，能够更好地捕捉到行为特征和变化，从而推动相关领域的研究进展。  
  
总之，“Hotplate\_behavior\_annotations\_new”数据集为本项目提供了丰富而高质量的训练数据，助力于改进YOLOv11的小鼠实例分割系统，使其在生物医学研究中发挥更大的作用。通过对小鼠行为的深入分析，研究团队希望能够揭示更多生物学上的重要信息，为未来的实验设计和数据分析提供新的视角。

## 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分析和注释，保留了最重要的功能和结构，同时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  
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  
  
def window\_partition(x, window\_size):  
 """ 将输入张量分割成窗口  
  
 Args:  
 x: 输入张量，形状为 (B, H, W, C)  
 window\_size (int): 窗口大小  
  
 Returns:  
 windows: 分割后的窗口，形状为 (num\_windows\*B, window\_size, window\_size, C)  
 """  
 B, H, W, C = x.shape  
 x = x.view(B, H // window\_size, window\_size, W // window\_size, window\_size, C) # 重塑形状  
 windows = x.permute(0, 1, 3, 2, 4, 5).contiguous().view(-1, window\_size, window\_size, C) # 重新排列并展平  
 return windows  
  
def window\_reverse(windows, window\_size, H, W):  
 """ 将窗口恢复为原始张量  
  
 Args:  
 windows: 窗口张量，形状为 (num\_windows\*B, window\_size, window\_size, C)  
 window\_size (int): 窗口大小  
 H (int): 原始高度  
 W (int): 原始宽度  
  
 Returns:  
 x: 恢复后的张量，形状为 (B, H, W, C)  
 """  
 B = int(windows.shape[0] / (H \* W / window\_size / window\_size)) # 计算批量大小  
 x = windows.view(B, H // window\_size, W // window\_size, window\_size, window\_size, -1) # 重塑形状  
 x = x.permute(0, 1, 3, 2, 4, 5).contiguous().view(B, H, W, -1) # 重新排列并展平  
 return x  
  
class WindowAttention(nn.Module):  
 """ 窗口注意力机制模块 """  
  
 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  
 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  
 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) # 初始化位置偏置  
 self.softmax = nn.Softmax(dim=-1) # Softmax层  
  
 def forward(self, x, mask=None):  
 """ 前向传播  
  
 Args:  
 x: 输入特征，形状为 (num\_windows\*B, N, C)  
 mask: 注意力掩码  
 """  
 B\_, N, C = x.shape  
 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) # 形状调整  
 relative\_position\_bias = relative\_position\_bias.permute(2, 0, 1).contiguous() # 重新排列  
 attn = attn + relative\_position\_bias.unsqueeze(0) # 加入位置偏置  
  
 if mask is not None:  
 nW = mask.shape[0]  
 attn = attn.view(B\_ // nW, nW, self.num\_heads, N, N) + mask.unsqueeze(1).unsqueeze(0) # 加入掩码  
 attn = attn.view(-1, self.num\_heads, N, N)  
 attn = self.softmax(attn) # 计算Softmax  
 else:  
 attn = self.softmax(attn)  
  
 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主干网络 """  
  
 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 = self.pos\_drop(x) # Dropout  
  
 outs = []  
 for layer in self.layers:  
 x\_out, \_, \_, x, \_, \_ = layer(x, x.size(1), x.size(2)) # 逐层前向传播  
 outs.append(x\_out) # 收集输出  
  
 return outs # 返回所有层的输出  
  
def SwinTransformer\_Tiny(weights=''):  
 """ 创建Swin Transformer Tiny模型 """  
 model = SwinTransformer(depths=[2, 2, 6, 2], num\_heads=[3, 6, 12, 24]) # 初始化模型  
 if weights:  
 model.load\_state\_dict(torch.load(weights)['model']) # 加载权重  
 return model  
```  
  
### 主要改动和注释  
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. \*\*window\_partition和window\_reverse\*\*: 这两个函数用于将输入张量分割成窗口以及将窗口恢复为原始张量。  
3. \*\*WindowAttention类\*\*: 实现了窗口注意力机制，包含相对位置偏置的计算。  
4. \*\*SwinTransformer类\*\*: 这是Swin Transformer的核心类，负责构建模型的各个层，并在前向传播中处理输入数据。  
5. \*\*SwinTransformer\_Tiny函数\*\*: 用于创建一个小型的Swin Transformer模型，并可选择加载预训练权重。  
  
以上代码展示了Swin Transformer的基本结构和功能，注释详细解释了每个部分的作用和流程。```

该文件实现了Swin Transformer模型，这是一个用于计算机视觉任务的层次化视觉Transformer架构。Swin Transformer通过使用移动窗口机制来有效地处理图像数据，具有较好的性能和效率。  
  
首先，文件导入了必要的库，包括PyTorch的基本模块和一些用于模型构建的工具函数。接着，定义了一个多层感知机（Mlp）类，该类由两个线性层和一个激活函数（默认为GELU）组成，适用于特征的非线性变换。  
  
接下来，定义了窗口划分和窗口反转的函数。这两个函数用于将输入特征图划分为多个小窗口，以便在局部区域内进行自注意力计算。窗口划分函数将输入的特征图分割成指定大小的窗口，而窗口反转函数则将经过处理的窗口重新组合成原始特征图的形状。  
  
随后，定义了窗口注意力（WindowAttention）类，这是Swin Transformer的核心组件之一。该类实现了基于窗口的多头自注意力机制，并引入了相对位置偏置，以增强模型对位置关系的理解。窗口注意力模块能够处理输入特征并计算注意力权重，从而生成加权后的输出特征。  
  
接着，定义了Swin Transformer块（SwinTransformerBlock），该块包含了归一化层、窗口注意力模块和前馈网络（FFN）。在前向传播中，输入特征经过归一化、窗口划分、注意力计算、窗口反转等步骤，最终输出经过残差连接的特征。  
  
接下来，定义了PatchMerging类，用于将输入特征图的不同区域合并成更大的特征块，从而降低特征图的分辨率。这个过程在Swin Transformer的不同阶段中使用，以逐步减少特征图的空间维度。  
  
BasicLayer类则表示Swin Transformer的一个基本层，其中包含多个Swin Transformer块，并可选择性地在层的末尾进行下采样。该类负责构建每个阶段的深度学习模块。  
  
PatchEmbed类用于将输入图像划分为小块，并将每个小块嵌入到一个高维空间中。它使用卷积操作来实现这一过程，并可选择性地进行归一化。  
  
SwinTransformer类是整个模型的主体，负责将所有的层组合在一起。它接收输入图像并通过多个阶段进行处理，输出特征图。该类还支持绝对位置嵌入、随机深度、归一化等功能。  
  
最后，文件中还定义了一个用于更新模型权重的函数，以及一个用于创建Swin Transformer Tiny模型的函数。这个Tiny版本的模型相较于标准版本具有更少的层数和头数，适合在资源有限的环境中使用。  
  
总体而言，该文件实现了Swin Transformer的各个组成部分，展示了如何构建一个高效的视觉Transformer模型，适用于各种计算机视觉任务。

```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  
  
# 定义构建卷积层的函数  
def conv\_2d(inp, oup, kernel\_size=3, stride=1, groups=1, bias=False, norm=True, act=True):  
 conv = nn.Sequential()  
 padding = (kernel\_size - 1) // 2 # 计算填充  
 # 添加卷积层  
 conv.add\_module('conv', nn.Conv2d(inp, oup, kernel\_size, stride, padding, bias=bias, groups=groups))  
 if norm:  
 # 添加批归一化层  
 conv.add\_module('BatchNorm2d', nn.BatchNorm2d(oup))  
 if act:  
 # 添加激活函数层（ReLU6）  
 conv.add\_module('Activation', nn.ReLU6())  
 return conv  
  
# 定义反向残差块  
class InvertedResidual(nn.Module):  
 def \_\_init\_\_(self, inp, oup, stride, expand\_ratio, act=False):  
 super(InvertedResidual, self).\_\_init\_\_()  
 self.stride = stride  
 assert stride in [1, 2] # 确保步幅为1或2  
 hidden\_dim = int(round(inp \* expand\_ratio)) # 计算扩展后的维度  
 self.block = nn.Sequential()  
 if expand\_ratio != 1:  
 # 添加扩展卷积层  
 self.block.add\_module('exp\_1x1', conv\_2d(inp, hidden\_dim, kernel\_size=1, stride=1))  
 # 添加深度卷积层  
 self.block.add\_module('conv\_3x3', conv\_2d(hidden\_dim, hidden\_dim, kernel\_size=3, stride=stride, groups=hidden\_dim))  
 # 添加投影卷积层  
 self.block.add\_module('red\_1x1', conv\_2d(hidden\_dim, oup, kernel\_size=1, stride=1, act=act))  
 # 判断是否使用残差连接  
 self.use\_res\_connect = self.stride == 1 and inp == oup  
  
 def forward(self, x):  
 if self.use\_res\_connect:  
 # 如果使用残差连接，则返回输入与块的输出相加  
 return x + self.block(x)  
 else:  
 return self.block(x)  
  
# 定义MobileNetV4模型  
class MobileNetV4(nn.Module):  
 def \_\_init\_\_(self, model):  
 super().\_\_init\_\_()  
 self.model = model  
 # 根据模型名称构建相应的层  
 self.conv0 = build\_blocks(self.spec['conv0'])  
 self.layer1 = build\_blocks(self.spec['layer1'])  
 self.layer2 = build\_blocks(self.spec['layer2'])  
 self.layer3 = build\_blocks(self.spec['layer3'])  
 self.layer4 = build\_blocks(self.spec['layer4'])  
 self.layer5 = build\_blocks(self.spec['layer5'])  
 self.features = nn.ModuleList([self.conv0, self.layer1, self.layer2, self.layer3, self.layer4, self.layer5])   
  
 def forward(self, x):  
 features = [None, None, None, None]  
 for f in self.features:  
 x = f(x) # 通过每一层进行前向传播  
 # 根据输入大小选择特征图  
 if input\_size // x.size(2) in scale:  
 features[scale.index(input\_size // x.size(2))] = x  
 return features  
  
# 构建不同版本的MobileNetV4模型  
def MobileNetV4ConvSmall():  
 return MobileNetV4('MobileNetV4ConvSmall')  
  
def MobileNetV4ConvMedium():  
 return MobileNetV4('MobileNetV4ConvMedium')  
  
def MobileNetV4ConvLarge():  
 return MobileNetV4('MobileNetV4ConvLarge')  
  
def MobileNetV4HybridMedium():  
 return MobileNetV4('MobileNetV4HybridMedium')  
  
def MobileNetV4HybridLarge():  
 return MobileNetV4('MobileNetV4HybridLarge')  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 model = MobileNetV4ConvSmall() # 创建MobileNetV4ConvSmall模型  
 inputs = torch.randn((1, 3, 640, 640)) # 创建随机输入  
 res = model(inputs) # 前向传播  
 for i in res:  
 print(i.size()) # 输出每个特征图的大小  
```  
  
### 代码核心部分说明：  
1. \*\*卷积层构建函数 `conv\_2d`\*\*：用于构建包含卷积、批归一化和激活函数的卷积层。  
2. \*\*反向残差块 `InvertedResidual`\*\*：实现了MobileNetV4中的反向残差结构，包含扩展卷积、深度卷积和投影卷积。  
3. \*\*MobileNetV4模型类 `MobileNetV4`\*\*：根据指定的模型构建各个层，并实现前向传播，提取特征图。  
4. \*\*模型构建函数\*\*：提供了不同版本的MobileNetV4模型的构建函数，方便用户调用。```

这个程序文件定义了一个名为 `MobileNetV4` 的深度学习模型，主要用于图像分类等计算机视觉任务。文件中使用了 PyTorch 框架，包含了多个类和函数，构建了不同版本的 MobileNetV4 模型。  
  
首先，文件中定义了一些常量，描述了不同 MobileNetV4 变体的结构，包括 `MobileNetV4ConvSmall`、`MobileNetV4ConvMedium`、`MobileNetV4ConvLarge`、`MobileNetV4HybridMedium` 和 `MobileNetV4HybridLarge`。这些结构以字典的形式存储，包含了每一层的类型、数量和参数设置。  
  
接下来，定义了一个 `make\_divisible` 函数，用于确保所有层的通道数都是 8 的倍数，这对于某些硬件加速是必要的。该函数接受一个值、一个除数和其他可选参数，并返回一个调整后的整数值。  
  
然后，定义了一个 `conv\_2d` 函数，用于创建一个包含卷积层、批归一化层和激活函数的序列模块。这个函数简化了后续层的构建过程。  
  
接下来是 `InvertedResidual` 类，它实现了倒残差块的结构。该类的构造函数接受输入和输出通道数、步幅和扩展比率等参数，并根据这些参数构建卷积层。`forward` 方法实现了前向传播逻辑，支持残差连接。  
  
`UniversalInvertedBottleneckBlock` 类类似于 `InvertedResidual`，但它提供了更灵活的卷积结构，包括起始深度卷积和中间深度卷积的可选性。该类的构造函数也根据传入的参数构建相应的卷积层。  
  
`build\_blocks` 函数根据给定的层规格构建相应的层模块。它会根据层的类型（如 `convbn`、`uib` 或 `fused\_ib`）调用相应的构造函数，生成层并添加到序列中。  
  
`MobileNetV4` 类是模型的主要实现。它的构造函数接受一个模型名称，并根据该名称构建相应的层。模型的每一层都通过 `build\_blocks` 函数构建，并存储在 `features` 列表中。`forward` 方法实现了模型的前向传播，返回不同尺度的特征图。  
  
最后，文件提供了几个函数，用于实例化不同版本的 MobileNetV4 模型。这些函数返回相应的模型实例，方便用户根据需求选择合适的模型。  
  
在文件的最后部分，提供了一个简单的测试代码，创建了一个 `MobileNetV4ConvSmall` 模型，并对随机输入进行前向传播，输出各层的特征图尺寸。这段代码可以用于验证模型的构建是否正确。

```以下是经过简化和注释的核心代码部分：  
  
```python  
import torch  
import torch.nn as nn  
from functools import partial  
  
# 导入自定义的归一化模块  
from .prepbn import RepBN, LinearNorm  
from ..modules.transformer import TransformerEncoderLayer  
  
# 定义可用的模块  
\_\_all\_\_ = ['AIFI\_RepBN']  
  
# 定义线性归一化的部分应用  
ln = nn.LayerNorm  
linearnorm = partial(LinearNorm, norm1=ln, norm2=RepBN, step=60000)  
  
class TransformerEncoderLayer\_RepBN(TransformerEncoderLayer):  
 def \_\_init\_\_(self, c1, cm=2048, num\_heads=8, dropout=0, act=..., normalize\_before=False):  
 # 初始化父类  
 super().\_\_init\_\_(c1, cm, num\_heads, dropout, act, normalize\_before)  
   
 # 使用自定义的归一化方法  
 self.norm1 = linearnorm(c1)  
 self.norm2 = linearnorm(c1)  
  
class AIFI\_RepBN(TransformerEncoderLayer\_RepBN):  
 """定义AIFI变换器层。"""  
  
 def \_\_init\_\_(self, c1, cm=2048, num\_heads=8, dropout=0, act=nn.GELU(), normalize\_before=False):  
 """使用指定参数初始化AIFI实例。"""  
 super().\_\_init\_\_(c1, cm, num\_heads, dropout, act, normalize\_before)  
  
 def forward(self, x):  
 """AIFI变换器层的前向传播。"""  
 c, h, w = x.shape[1:] # 获取输入的通道数、高度和宽度  
 pos\_embed = self.build\_2d\_sincos\_position\_embedding(w, h, c) # 构建位置嵌入  
   
 # 将输入张量从形状[B, C, H, W]展平为[B, HxW, C]  
 x = super().forward(x.flatten(2).permute(0, 2, 1), pos=pos\_embed.to(device=x.device, dtype=x.dtype))  
   
 # 将输出张量恢复为形状[B, C, H, W]  
 return x.permute(0, 2, 1).view([-1, c, h, w]).contiguous()  
  
 @staticmethod  
 def build\_2d\_sincos\_position\_embedding(w, h, embed\_dim=256, temperature=10000.0):  
 """构建2D正弦-余弦位置嵌入。"""  
 assert embed\_dim % 4 == 0, "嵌入维度必须是4的倍数以便进行2D正弦-余弦位置嵌入"  
   
 # 创建宽度和高度的网格  
 grid\_w = torch.arange(w, dtype=torch.float32)  
 grid\_h = torch.arange(h, dtype=torch.float32)  
 grid\_w, grid\_h = torch.meshgrid(grid\_w, grid\_h, indexing="ij")  
   
 pos\_dim = embed\_dim // 4 # 计算位置维度  
 omega = torch.arange(pos\_dim, dtype=torch.float32) / pos\_dim  
 omega = 1.0 / (temperature\*\*omega) # 计算频率  
  
 # 计算宽度和高度的正弦和余弦值  
 out\_w = grid\_w.flatten()[..., None] @ omega[None]  
 out\_h = grid\_h.flatten()[..., None] @ omega[None]  
  
 # 返回组合的正弦和余弦位置嵌入  
 return torch.cat([torch.sin(out\_w), torch.cos(out\_w), torch.sin(out\_h), torch.cos(out\_h)], 1)[None]  
```  
  
### 代码说明：  
1. \*\*导入模块\*\*：引入了PyTorch及其相关模块，以及自定义的归一化和变换器模块。  
2. \*\*线性归一化\*\*：使用`partial`函数创建一个自定义的线性归一化函数。  
3. \*\*TransformerEncoderLayer\_RepBN\*\*：这是一个继承自`TransformerEncoderLayer`的类，增加了自定义的归一化层。  
4. \*\*AIFI\_RepBN\*\*：这是AIFI变换器层的实现，包含了前向传播和位置嵌入的构建。  
5. \*\*前向传播\*\*：在`forward`方法中，输入张量被展平并传递给父类的前向方法，同时生成位置嵌入。  
6. \*\*位置嵌入构建\*\*：`build\_2d\_sincos\_position\_embedding`方法生成2D正弦-余弦位置嵌入，用于为输入数据提供位置信息。```

这个程序文件定义了一个名为 `transformer.py` 的模块，主要实现了基于 Transformer 的编码层，特别是使用了 RepBN（重参数化批归一化）和 AIFI（自适应输入特征融合）技术的变体。  
  
首先，文件导入了必要的 PyTorch 库和模块，包括 `torch`、`torch.nn` 和 `torch.nn.functional`，以及一些自定义的模块，如 `RepBN` 和 `TransformerEncoderLayer`。这些导入为后续的模型构建和操作提供了基础。  
  
接下来，定义了一个名为 `linearnorm` 的部分函数（partial function），它结合了 `LayerNorm` 和 `RepBN`，并设置了一个步骤参数（step=60000）。这个函数用于后续的层归一化操作。  
  
然后，定义了一个 `TransformerEncoderLayer\_RepBN` 类，它继承自 `TransformerEncoderLayer`。在初始化方法中，调用了父类的构造函数，并定义了两个归一化层 `norm1` 和 `norm2`，这两个层使用了之前定义的 `linearnorm` 函数。  
  
接着，定义了 `AIFI\_RepBN` 类，它继承自 `TransformerEncoderLayer\_RepBN`，并在其构造函数中调用了父类的构造函数，初始化了一些参数，如输入通道数、隐藏层大小、头数、丢弃率、激活函数等。  
  
在 `AIFI\_RepBN` 类中，重写了 `forward` 方法，该方法实现了前向传播的逻辑。首先获取输入张量的形状信息（通道数、高度和宽度），然后调用 `build\_2d\_sincos\_position\_embedding` 方法生成二维的正弦余弦位置嵌入。接着，将输入张量从形状 `[B, C, H, W]` 转换为 `[B, HxW, C]`，并将位置嵌入传递给父类的前向方法进行处理。最后，将输出张量重新排列回原来的形状。  
  
最后，定义了一个静态方法 `build\_2d\_sincos\_position\_embedding`，用于构建二维的正弦余弦位置嵌入。该方法首先检查嵌入维度是否可以被4整除，然后生成网格坐标，并计算相应的正弦和余弦值，最终返回一个包含这些位置嵌入的张量。  
  
总体而言，这个文件实现了一个自定义的 Transformer 编码层，结合了现代的归一化技术和位置编码方法，适用于处理图像等二维数据。

```以下是经过简化和注释的核心代码部分：  
  
```python  
# 导入必要的模块  
from ultralytics.engine.predictor import BasePredictor  
from ultralytics.engine.results import Results  
from ultralytics.utils import ops  
  
class DetectionPredictor(BasePredictor):  
 """  
 DetectionPredictor类用于基于检测模型进行预测，继承自BasePredictor类。  
 """  
  
 def postprocess(self, preds, img, orig\_imgs):  
 """  
 对预测结果进行后处理，并返回Results对象的列表。  
  
 参数:  
 preds: 模型的预测结果  
 img: 输入图像  
 orig\_imgs: 原始图像（可能是torch.Tensor或numpy数组）  
  
 返回:  
 results: 包含处理后结果的Results对象列表  
 """  
 # 应用非极大值抑制（NMS）来过滤预测框  
 preds = ops.non\_max\_suppression(  
 preds,  
 self.args.conf, # 置信度阈值  
 self.args.iou, # IOU阈值  
 agnostic=self.args.agnostic\_nms, # 是否使用类别无关的NMS  
 max\_det=self.args.max\_det, # 最大检测框数量  
 classes=self.args.classes, # 过滤的类别  
 )  
  
 # 如果输入的原始图像不是列表，则将其转换为numpy数组  
 if not isinstance(orig\_imgs, list):  
 orig\_imgs = ops.convert\_torch2numpy\_batch(orig\_imgs)  
  
 results = [] # 初始化结果列表  
 for i, pred in enumerate(preds):  
 orig\_img = orig\_imgs[i] # 获取对应的原始图像  
 # 将预测框的坐标缩放到原始图像的尺寸  
 pred[:, :4] = ops.scale\_boxes(img.shape[2:], pred[:, :4], orig\_img.shape)  
 img\_path = self.batch[0][i] # 获取图像路径  
 # 创建Results对象并添加到结果列表  
 results.append(Results(orig\_img, path=img\_path, names=self.model.names, boxes=pred))  
   
 return results # 返回处理后的结果列表  
```  
  
### 代码说明：  
1. \*\*导入模块\*\*：导入了进行预测和处理结果所需的基础模块。  
2. \*\*DetectionPredictor类\*\*：这是一个用于处理检测模型预测的类，继承自`BasePredictor`。  
3. \*\*postprocess方法\*\*：该方法负责对模型的预测结果进行后处理，主要包括：  
 - 使用非极大值抑制（NMS）来过滤掉重叠的预测框。  
 - 将输入的原始图像转换为numpy数组（如果需要）。  
 - 遍历每个预测框，缩放其坐标到原始图像的尺寸，并创建一个`Results`对象来存储结果。  
4. \*\*返回结果\*\*：最终返回一个包含所有处理后结果的列表。```

这个程序文件 `predict.py` 是一个用于目标检测的预测模块，继承自 `BasePredictor` 类，主要用于处理基于检测模型的预测任务。代码中包含了一个名为 `DetectionPredictor` 的类，该类的主要功能是对输入的图像进行目标检测，并对检测结果进行后处理。  
  
在类的文档字符串中，提供了一个使用示例，展示了如何导入必要的模块并创建 `DetectionPredictor` 的实例。用户可以通过传入模型文件路径和数据源来初始化预测器，并调用 `predict\_cli()` 方法进行预测。  
  
`postprocess` 方法是该类的核心功能之一，负责对模型的预测结果进行后处理。首先，它调用 `ops.non\_max\_suppression` 函数来执行非极大值抑制（NMS），以过滤掉重叠的检测框。该函数的参数包括置信度阈值、IOU阈值、是否使用类别无关的NMS、最大检测框数量以及需要检测的类别。  
  
接下来，代码检查输入的原始图像是否为列表形式。如果不是，说明输入的是一个 PyTorch 张量，则调用 `ops.convert\_torch2numpy\_batch` 函数将其转换为 NumPy 数组，以便后续处理。  
  
然后，代码通过循环遍历每个预测结果，对每个预测框进行坐标缩放，以适应原始图像的尺寸。最后，将原始图像、图像路径、模型名称和预测框信息封装成 `Results` 对象，并将其添加到结果列表中。  
  
最终，`postprocess` 方法返回一个包含所有结果的列表，供后续使用或展示。这段代码展示了如何在目标检测任务中处理和优化预测结果，使其更符合实际应用需求。

### 程序整体功能和构架概括  
  
该程序是一个计算机视觉项目，主要实现了多种深度学习模型（如 Swin Transformer 和 MobileNetV4）以及目标检测的预测功能。程序的整体架构分为以下几个部分：  
  
1. \*\*模型定义\*\*：包含不同类型的深度学习模型（如 Swin Transformer 和 MobileNetV4），这些模型可以用于图像分类和特征提取等任务。  
2. \*\*Transformer 编码层\*\*：实现了基于 Transformer 的编码层，结合了现代的归一化技术和位置编码方法，适用于处理图像等二维数据。  
3. \*\*预测模块\*\*：实现了目标检测的预测功能，包括对输入图像的处理、模型预测结果的后处理（如非极大值抑制）等。  
  
整体而言，该程序提供了一个灵活的框架，用户可以根据需求选择不同的模型进行训练和推理，同时支持对目标检测任务的处理。  
  
### 文件功能整理表  
  
| 文件名 | 功能描述 |  
|----------------------------|-----------------------------------------------------------------------------------------------|  
| `SwinTransformer.py` | 实现了 Swin Transformer 模型，包括各个组成部分（如窗口注意力、残差块、特征合并等），适用于计算机视觉任务。 |  
| `mobilenetv4.py` | 定义了 MobileNetV4 模型的不同变体，构建了倒残差块和特征提取层，适合在资源有限的环境中使用。 |  
| `transformer.py` | 实现了基于 Transformer 的编码层，结合了 RepBN 和 AIFI 技术，适用于处理图像等二维数据。 |  
| `predict.py` | 提供了目标检测的预测功能，包括对输入图像的处理和模型预测结果的后处理（如非极大值抑制）。 |  
  
这个表格总结了每个文件的主要功能，便于快速了解程序的结构和各个模块的作用。