# 改进yolo11-swintransformer等200+全套创新点大全：垃圾车检测系统源码＆数据集全套

## 1. 图片效果展示

注意：由于项目一直在更新迭代，上面“1.图片效果展示”和“2.视频效果展示”展示的系统图片或者视频可能为老版本，新版本在老版本的基础上升级如下：（实际效果以升级的新版本为准）  
  
 （1）适配了YOLOV11的“目标检测”模型和“实例分割”模型，通过加载相应的权重（.pt）文件即可自适应加载模型。  
  
 （2）支持“图片识别”、“视频识别”、“摄像头实时识别”三种识别模式。  
  
 （3）支持“图片识别”、“视频识别”、“摄像头实时识别”三种识别结果保存导出，解决手动导出（容易卡顿出现爆内存）存在的问题，识别完自动保存结果并导出到tempDir中。  
  
 （4）支持Web前端系统中的标题、背景图等自定义修改。  
  
 另外本项目提供训练的数据集和训练教程,暂不提供权重文件（best.pt）,需要您按照教程进行训练后实现图片演示和Web前端界面演示的效果。

## 2.视频效果展示

## 3.背景

研究背景与意义  
  
随着城市化进程的加快，城市垃圾管理问题日益凸显，垃圾车的高效调度与管理成为城市环境维护的重要环节。传统的垃圾车监控方式多依赖人工巡查，不仅效率低下，而且容易受到人为因素的影响，难以实现实时、准确的监控。因此，基于计算机视觉的自动化垃圾车检测系统应运而生，成为提升城市垃圾管理效率的重要技术手段。  
  
近年来，深度学习技术的迅猛发展为目标检测领域带来了革命性的变化。YOLO（You Only Look Once）系列模型以其高效的实时检测能力和良好的精度，广泛应用于各类目标检测任务中。YOLOv11作为该系列的最新版本，进一步优化了模型结构，提高了检测速度和准确性，尤其适合于动态场景下的目标识别。然而，现有的YOLOv11模型在特定应用场景下的表现仍有提升空间，尤其是在复杂背景和光照变化下的垃圾车检测。  
  
本研究旨在基于改进的YOLOv11模型，构建一个高效的垃圾车检测系统。为此，我们将使用包含1000张图像的GarBage数据集，该数据集专注于垃圾车这一特定类别，具有较高的应用价值。通过对数据集的深入分析与处理，结合改进的YOLOv11模型，我们期望能够显著提升垃圾车的检测精度和实时性，从而为城市垃圾管理提供强有力的技术支持。  
  
此外，本研究不仅有助于推动智能监控技术在城市管理中的应用，还将为相关领域的研究提供参考，促进计算机视觉技术与实际应用场景的深度融合。通过实现高效的垃圾车检测系统，我们希望能够为城市环境的可持续发展贡献一份力量。

## 4. 数据集信息展示

本项目数据集信息介绍  
  
本项目所使用的数据集名为“GarBage”，专门用于训练和改进YOLOv11模型，以实现高效的垃圾车检测系统。该数据集的设计旨在提供一个全面而精确的训练基础，确保模型能够在各种环境和条件下准确识别垃圾车。数据集中包含的类别数量为1，具体类别为“Garbage-Trucks”。这一单一类别的设定使得数据集在专注于垃圾车检测的同时，能够更深入地捕捉该类别的多样性和复杂性。  
  
“GarBage”数据集的构建过程中，特别注重了数据的多样性和代表性。数据集中的图像涵盖了不同类型的垃圾车，包括各种型号、颜色和外观设计，以反映现实世界中垃圾车的广泛变化。此外，数据集还考虑了不同的拍摄角度、光照条件和背景环境，确保模型在面对实际应用时具备良好的适应性和鲁棒性。这种多样化的图像来源有助于提升YOLOv11模型的泛化能力，使其能够在不同场景下保持高准确率。  
  
为了进一步增强数据集的实用性，所有图像均经过精心标注，确保每一张图像中的垃圾车都被准确框定。这种精确的标注不仅为模型的训练提供了高质量的监督信号，也为后续的模型评估和性能分析奠定了基础。通过使用“GarBage”数据集，研究团队期望能够显著提升垃圾车检测系统的性能，从而为城市垃圾管理和环境保护提供更为有效的技术支持。

## 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

```以下是对代码中最核心部分的提取和详细中文注释：  
  
```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))  
  
 # 获取窗口内每个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 # 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  
  
 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)  
 q, k, v = qkv[0], qkv[1], qkv[2] # 获取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)  
 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 = attn.softmax(dim=-1) # 归一化  
 attn = self.attn\_drop(attn) # Dropout  
  
 x = (attn @ v).transpose(1, 2).reshape(B\_, N, C) # 应用注意力  
 x = self.proj(x) # 输出线性变换  
 x = self.proj\_drop(x) # Dropout  
 return x  
  
  
class SwinTransformer(nn.Module):  
 """ Swin Transformer 主体。 """  
  
 def \_\_init\_\_(self, patch\_size=4, embed\_dim=96, depths=[2, 2, 6, 2], num\_heads=[3, 6, 12, 24]):  
 super().\_\_init\_\_()  
 self.patch\_embed = PatchEmbed(patch\_size=patch\_size, in\_chans=3, embed\_dim=embed\_dim) # 图像到补丁的嵌入  
  
 # 构建每一层  
 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=7)  
 self.layers.append(layer)  
  
 def forward(self, x):  
 """ 前向传播函数。 """  
 x = self.patch\_embed(x) # 嵌入补丁  
 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  
  
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]) # 初始化模型  
 if weights:  
 model.load\_state\_dict(torch.load(weights)['model']) # 加载权重  
 return model  
```  
  
### 代码核心部分说明：  
1. \*\*Mlp 类\*\*：实现了一个多层感知机，包括两层线性变换和激活函数，适用于特征转换。  
2. \*\*WindowAttention 类\*\*：实现了窗口自注意力机制，支持相对位置偏置的计算。  
3. \*\*SwinTransformer 类\*\*：构建了Swin Transformer的主体结构，包括补丁嵌入和多个基本层。  
4. \*\*SwinTransformer\_Tiny 函数\*\*：提供了一个创建小型Swin Transformer模型的接口，并可选加载预训练权重。  
  
这些核心部分构成了Swin Transformer的基本功能，适用于图像处理任务。```

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。文件中定义了多个类和函数，主要包括Mlp、WindowAttention、SwinTransformerBlock、PatchMerging、BasicLayer、PatchEmbed和SwinTransformer等。  
  
首先，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（默认为GELU），同时支持dropout。它的前向传播方法依次通过线性层、激活函数和dropout进行处理。  
  
接下来，window\_partition和window\_reverse函数用于将输入张量分割成窗口和将窗口合并回原始形状。window\_partition函数将输入张量按给定窗口大小分割成多个小窗口，而window\_reverse则将这些小窗口合并回原始形状。  
  
WindowAttention类实现了基于窗口的多头自注意力机制。它通过计算查询、键、值的线性变换，并引入相对位置偏置来增强模型的表达能力。前向传播时，它计算注意力权重并应用于值向量，最后通过线性变换和dropout输出结果。  
  
SwinTransformerBlock类则是Swin Transformer的基本构建块，包含一个窗口注意力层和一个多层感知机。它支持窗口的循环移位（shifted window），并在前向传播中进行规范化、窗口分割、注意力计算和特征融合。  
  
PatchMerging类用于将输入特征图的多个patch合并为更大的patch，以降低特征图的分辨率并增加通道数。它通过线性层进行特征降维，并在合并前进行规范化。  
  
BasicLayer类是一个Swin Transformer的基本层，包含多个SwinTransformerBlock，并在层之间进行patch合并。它计算注意力掩码并在每个块中进行前向传播。  
  
PatchEmbed类将输入图像分割成不重叠的patch，并通过卷积层进行线性投影。它还支持在投影后进行规范化。  
  
SwinTransformer类是整个模型的主体，负责构建整个Swin Transformer的层次结构。它将输入图像进行patch嵌入，添加位置嵌入，并依次通过每一层进行处理。最后，它返回指定层的输出。  
  
最后，update\_weight函数用于更新模型的权重，SwinTransformer\_Tiny函数则用于创建一个小型的Swin Transformer模型，并可选择加载预训练权重。  
  
整体来看，该文件实现了Swin Transformer的各个组成部分，提供了一个完整的框架用于图像处理和特征提取。

```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  
  
# 定义一个自定义的批量归一化类  
class RepBN(nn.Module):  
 def \_\_init\_\_(self, channels):  
 super(RepBN, self).\_\_init\_\_()  
 # alpha是一个可学习的参数，初始化为1  
 self.alpha = nn.Parameter(torch.ones(1))  
 # 使用1D批量归一化  
 self.bn = nn.BatchNorm1d(channels)  
  
 def forward(self, x):  
 # 将输入的维度进行转置，以适应BatchNorm的输入格式  
 x = x.transpose(1, 2)  
 # 进行批量归一化，并加上可学习的参数乘以输入  
 x = self.bn(x) + self.alpha \* x  
 # 再次转置回原来的维度  
 x = x.transpose(1, 2)  
 return x  
  
# 定义一个线性归一化类  
class LinearNorm(nn.Module):  
 def \_\_init\_\_(self, dim, norm1, norm2, warm=0, step=300000, r0=1.0):  
 super(LinearNorm, self).\_\_init\_\_()  
 # 注册一些缓冲区，用于存储训练过程中的状态  
 self.register\_buffer('warm', torch.tensor(warm)) # 预热阶段的计数  
 self.register\_buffer('iter', torch.tensor(step)) # 当前迭代次数  
 self.register\_buffer('total\_step', torch.tensor(step)) # 总迭代次数  
 self.r0 = r0 # 初始比例  
 self.norm1 = norm1(dim) # 第一个归一化层  
 self.norm2 = norm2(dim) # 第二个归一化层  
  
 def forward(self, x):  
 if self.training: # 如果模型在训练模式  
 if self.warm > 0: # 如果还有预热阶段  
 self.warm.copy\_(self.warm - 1) # 递减预热计数  
 x = self.norm1(x) # 仅使用第一个归一化  
 else:  
 # 计算当前的lambda值，用于线性插值  
 lamda = self.r0 \* self.iter / self.total\_step  
 if self.iter > 0:  
 self.iter.copy\_(self.iter - 1) # 递减迭代计数  
 x1 = self.norm1(x) # 使用第一个归一化  
 x2 = self.norm2(x) # 使用第二个归一化  
 # 线性插值  
 x = lamda \* x1 + (1 - lamda) \* x2  
 else:  
 x = self.norm2(x) # 在评估模式下只使用第二个归一化  
 return x  
```  
  
### 代码核心部分解释：  
1. \*\*RepBN 类\*\*：  
 - 这是一个自定义的批量归一化层，增加了一个可学习的参数 `alpha`，用于调整输入的影响。  
 - 在 `forward` 方法中，输入张量的维度被转置，以适应批量归一化的要求，然后进行归一化处理，并结合输入张量的线性组合。  
  
2. \*\*LinearNorm 类\*\*：  
 - 这个类实现了一个线性归一化策略，根据训练的阶段动态选择使用不同的归一化方法。  
 - 在训练过程中，首先会进行预热阶段，使用第一个归一化方法；然后逐渐过渡到第二个归一化方法，通过线性插值的方式结合两者的输出。  
 - 在评估模式下，直接使用第二个归一化方法。```

这个程序文件定义了两个神经网络模块，分别是 `RepBN` 和 `LinearNorm`，它们都是基于 PyTorch 框架构建的。  
  
`RepBN` 类是一个自定义的批量归一化模块。它的构造函数接收一个参数 `channels`，表示输入数据的通道数。在初始化过程中，它定义了一个可学习的参数 `alpha`，并创建了一个标准的批量归一化层 `bn`。在 `forward` 方法中，输入 `x` 首先进行维度转置，以适应批量归一化的要求。接着，经过批量归一化处理后，输出结果与 `alpha` 乘以原始输入相加，最后再进行一次维度转置，返回处理后的结果。这种结构允许网络在训练过程中动态调整输入的归一化程度。  
  
`LinearNorm` 类则实现了一种线性归一化策略。它的构造函数接收多个参数，包括 `dim`（输入的维度）、`norm1` 和 `norm2`（分别是两种归一化方法的构造函数）、`warm`（预热步数）、`step`（总步数）和 `r0`（初始比例因子）。在初始化时，使用 `register\_buffer` 方法注册了一些缓冲区变量，这些变量在模型训练过程中会被更新。`forward` 方法根据模型的训练状态进行不同的处理。如果模型处于训练状态且预热步数大于零，则使用第一种归一化方法 `norm1` 处理输入数据；如果预热步数已用尽，则计算一个动态的比例因子 `lamda`，并根据这个因子将两种归一化方法的输出进行线性组合。若模型处于评估状态，则直接使用第二种归一化方法 `norm2` 处理输入数据。  
  
整体来看，这两个模块的设计旨在提供灵活的归一化策略，以适应不同的训练阶段和需求，从而提升模型的性能和稳定性。

```以下是经过简化和注释的核心代码部分，主要包括基本模块和网络结构的定义。  
  
```python  
import torch  
import torch.nn as nn  
import torch.nn.functional as F  
from ..modules.conv import Conv # 导入卷积模块  
  
# 定义基本块  
class BasicBlock(nn.Module):  
 def \_\_init\_\_(self, filter\_in, filter\_out):  
 super(BasicBlock, self).\_\_init\_\_()  
 # 定义两个卷积层  
 self.conv1 = Conv(filter\_in, filter\_out, 3) # 第一个卷积层  
 self.conv2 = Conv(filter\_out, filter\_out, 3, act=False) # 第二个卷积层，不使用激活函数  
  
 def forward(self, x):  
 residual = x # 保存输入以便后续加回  
 out = self.conv1(x) # 通过第一个卷积层  
 out = self.conv2(out) # 通过第二个卷积层  
 out += residual # 残差连接  
 return self.conv1.act(out) # 返回经过激活函数的输出  
  
# 定义上采样模块  
class Upsample(nn.Module):  
 def \_\_init\_\_(self, in\_channels, out\_channels, scale\_factor=2):  
 super(Upsample, self).\_\_init\_\_()  
 # 定义上采样序列  
 self.upsample = nn.Sequential(  
 Conv(in\_channels, out\_channels, 1), # 1x1卷积  
 nn.Upsample(scale\_factor=scale\_factor, mode='bilinear') # 双线性插值上采样  
 )  
  
 def forward(self, x):  
 return self.upsample(x) # 返回上采样后的结果  
  
# 定义下采样模块  
class Downsample\_x2(nn.Module):  
 def \_\_init\_\_(self, in\_channels, out\_channels):  
 super(Downsample\_x2, self).\_\_init\_\_()  
 # 定义2倍下采样的卷积  
 self.downsample = Conv(in\_channels, out\_channels, 2, 2, 0)  
  
 def forward(self, x):  
 return self.downsample(x) # 返回下采样后的结果  
  
# 定义自适应特征融合模块（ASFF）  
class ASFF\_2(nn.Module):  
 def \_\_init\_\_(self, inter\_dim=512):  
 super(ASFF\_2, self).\_\_init\_\_()  
 self.inter\_dim = inter\_dim  
 compress\_c = 8 # 压缩通道数  
  
 # 定义权重卷积层  
 self.weight\_level\_1 = Conv(self.inter\_dim, compress\_c, 1)  
 self.weight\_level\_2 = Conv(self.inter\_dim, compress\_c, 1)  
 self.weight\_levels = nn.Conv2d(compress\_c \* 2, 2, kernel\_size=1, stride=1, padding=0)  
 self.conv = Conv(self.inter\_dim, self.inter\_dim, 3) # 最后的卷积层  
  
 def forward(self, input1, input2):  
 # 计算权重  
 level\_1\_weight\_v = self.weight\_level\_1(input1)  
 level\_2\_weight\_v = self.weight\_level\_2(input2)  
 levels\_weight\_v = torch.cat((level\_1\_weight\_v, level\_2\_weight\_v), 1) # 拼接权重  
 levels\_weight = self.weight\_levels(levels\_weight\_v) # 计算最终权重  
 levels\_weight = F.softmax(levels\_weight, dim=1) # 归一化权重  
  
 # 融合输入特征  
 fused\_out\_reduced = input1 \* levels\_weight[:, 0:1, :, :] + input2 \* levels\_weight[:, 1:2, :, :]  
 out = self.conv(fused\_out\_reduced) # 通过卷积层  
 return out # 返回融合后的输出  
  
# 定义网络主体  
class BlockBody\_P345(nn.Module):  
 def \_\_init\_\_(self, channels=[64, 128, 256, 512]):  
 super(BlockBody\_P345, self).\_\_init\_\_()  
 # 定义不同尺度的卷积块和下采样、上采样模块  
 self.blocks\_scalezero1 = nn.Sequential(Conv(channels[0], channels[0], 1))  
 self.blocks\_scaleone1 = nn.Sequential(Conv(channels[1], channels[1], 1))  
 self.blocks\_scaletwo1 = nn.Sequential(Conv(channels[2], channels[2], 1))  
   
 self.downsample\_scalezero1\_2 = Downsample\_x2(channels[0], channels[1])  
 self.upsample\_scaleone1\_2 = Upsample(channels[1], channels[0], scale\_factor=2)  
  
 self.asff\_scalezero1 = ASFF\_2(inter\_dim=channels[0])  
 self.asff\_scaleone1 = ASFF\_2(inter\_dim=channels[1])  
  
 # 定义多个BasicBlock  
 self.blocks\_scalezero2 = nn.Sequential(\*[BasicBlock(channels[0], channels[0]) for \_ in range(4)])  
 self.blocks\_scaleone2 = nn.Sequential(\*[BasicBlock(channels[1], channels[1]) for \_ in range(4)])  
  
 def forward(self, x):  
 x0, x1, x2 = x # 分别获取不同尺度的输入  
 x0 = self.blocks\_scalezero1(x0) # 处理尺度0  
 x1 = self.blocks\_scaleone1(x1) # 处理尺度1  
 x2 = self.blocks\_scaletwo1(x2) # 处理尺度2  
  
 # 进行特征融合  
 scalezero = self.asff\_scalezero1(x0, self.upsample\_scaleone1\_2(x1))  
 scaleone = self.asff\_scaleone1(self.downsample\_scalezero1\_2(x0), x1)  
  
 x0 = self.blocks\_scalezero2(scalezero) # 再次处理尺度0  
 x1 = self.blocks\_scaleone2(scaleone) # 再次处理尺度1  
  
 return x0, x1, x2 # 返回处理后的特征  
  
# 定义主网络AFPN\_P345  
class AFPN\_P345(nn.Module):  
 def \_\_init\_\_(self, in\_channels=[256, 512, 1024], out\_channels=256, factor=4):  
 super(AFPN\_P345, self).\_\_init\_\_()  
 # 定义输入卷积层  
 self.conv0 = Conv(in\_channels[0], in\_channels[0] // factor, 1)  
 self.conv1 = Conv(in\_channels[1], in\_channels[1] // factor, 1)  
 self.conv2 = Conv(in\_channels[2], in\_channels[2] // factor, 1)  
  
 # 定义网络主体  
 self.body = BlockBody\_P345([in\_channels[0] // factor, in\_channels[1] // factor, in\_channels[2] // factor])  
  
 # 定义输出卷积层  
 self.conv00 = Conv(in\_channels[0] // factor, out\_channels, 1)  
 self.conv11 = Conv(in\_channels[1] // factor, out\_channels, 1)  
 self.conv22 = Conv(in\_channels[2] // factor, out\_channels, 1)  
  
 def forward(self, x):  
 x0, x1, x2 = x # 获取输入  
 x0 = self.conv0(x0) # 处理输入0  
 x1 = self.conv1(x1) # 处理输入1  
 x2 = self.conv2(x2) # 处理输入2  
  
 out0, out1, out2 = self.body([x0, x1, x2]) # 通过主体网络处理  
  
 out0 = self.conv00(out0) # 输出处理  
 out1 = self.conv11(out1)  
 out2 = self.conv22(out2)  
  
 return [out0, out1, out2] # 返回输出  
```  
  
### 代码说明：  
1. \*\*BasicBlock\*\*: 实现了一个基本的残差块，包含两个卷积层和残差连接。  
2. \*\*Upsample/Downsample\*\*: 定义了上采样和下采样模块，分别用于调整特征图的尺寸。  
3. \*\*ASFF\*\*: 自适应特征融合模块，通过计算输入特征的权重进行融合。  
4. \*\*BlockBody\_P345\*\*: 组合多个卷积块和ASFF模块，处理不同尺度的特征。  
5. \*\*AFPN\_P345\*\*: 主网络结构，包含输入卷积层、主体处理和输出卷积层。  
  
以上是代码的核心部分和详细注释，旨在帮助理解网络结构和功能。```

这个程序文件 `afpn.py` 实现了一个基于自适应特征金字塔网络（AFPN）的深度学习模型，主要用于计算机视觉任务，如目标检测和图像分割。代码中定义了多个类，每个类代表网络的不同组件或模块，以下是对代码的详细说明。  
  
首先，导入了一些必要的库，包括 `torch` 和 `torch.nn`，以及一些自定义的模块如 `Conv` 和不同的块（`C2f`, `C3`, `C3Ghost`, `C3k2`）。这些模块提供了构建卷积层和其他操作的基础。  
  
接下来，定义了多个类。`BasicBlock` 类是一个基本的残差块，包含两个卷积层，使用了残差连接。`Upsample` 和 `Downsample\_x2/x4/x8` 类分别实现了上采样和不同倍数的下采样操作。这些类在网络中用于调整特征图的尺寸，以便进行特征融合。  
  
`ASFF\_2`, `ASFF\_3`, 和 `ASFF\_4` 类实现了自适应特征融合模块（ASFF），它们通过计算不同输入特征图的权重并进行加权融合来增强特征表示。这些模块的设计使得网络能够自适应地选择不同尺度的特征进行融合，从而提高模型的性能。  
  
`BlockBody\_P345` 和 `BlockBody\_P2345` 类是网络的主体部分，分别处理三个和四个尺度的特征图。它们通过堆叠多个卷积块和自适应特征融合模块来构建深层网络结构。这些类的 `forward` 方法定义了数据在网络中的前向传播过程，包括特征的下采样、上采样和融合。  
  
`AFPN\_P345` 和 `AFPN\_P2345` 类是最终的网络结构，它们负责接收输入特征图并通过前面定义的模块进行处理，输出融合后的特征图。这些类的构造函数中还包括对卷积层权重的初始化，以确保网络在训练开始时具有良好的性能。  
  
此外，`BlockBody\_P345\_Custom` 和 `BlockBody\_P2345\_Custom` 类允许用户自定义块的类型，这使得网络具有更大的灵活性，可以根据具体任务需求选择不同的卷积块。  
  
总体而言，这个程序文件实现了一个复杂的深度学习模型，利用自适应特征融合和多尺度特征处理来提升计算机视觉任务的性能。通过模块化的设计，代码结构清晰，便于扩展和修改。

```以下是经过简化和注释的核心代码部分：  
  
```python  
import torch  
import torch.nn.functional as F  
  
def selective\_scan\_easy(us, dts, As, Bs, Cs, Ds, delta\_bias=None, delta\_softplus=False, return\_last\_state=False, chunksize=64):  
 """  
 选择性扫描函数，执行状态更新和输出计算。  
   
 参数：  
 us: 输入张量，形状为 (B, G \* D, L)  
 dts: 时间增量张量，形状为 (B, G \* D, L)  
 As: 系数矩阵，形状为 (G \* D, N)  
 Bs: 权重张量，形状为 (B, G, N, L)  
 Cs: 输出权重张量，形状为 (B, G, N, L)  
 Ds: 偏置张量，形状为 (G \* D)  
 delta\_bias: 可选的偏置调整，形状为 (G \* D)  
 delta\_softplus: 是否应用softplus激活  
 return\_last\_state: 是否返回最后的状态  
 chunksize: 处理的块大小  
 """  
   
 def selective\_scan\_chunk(us, dts, As, Bs, Cs, hprefix):  
 """  
 处理单个块的选择性扫描。  
   
 参数：  
 us: 输入张量块  
 dts: 时间增量块  
 As, Bs, Cs: 系数和权重张量  
 hprefix: 前一个状态  
 """  
 ts = dts.cumsum(dim=0) # 计算时间增量的累积和  
 Ats = torch.einsum("gdn,lbgd->lbgdn", As, ts).exp() # 计算A的指数  
 rAts = Ats # 归一化处理  
 duts = dts \* us # 计算增量  
 dtBus = torch.einsum("lbgd,lbgn->lbgdn", duts, Bs) # 计算中间结果  
 hs\_tmp = rAts \* (dtBus / rAts).cumsum(dim=0) # 计算状态  
 hs = hs\_tmp + Ats \* hprefix.unsqueeze(0) # 更新状态  
 ys = torch.einsum("lbgn,lbgdn->lbgd", Cs, hs) # 计算输出  
 return ys, hs  
  
 # 数据类型处理  
 dtype = torch.float32  
 dts = dts.to(dtype)  
 if delta\_bias is not None:  
 dts += delta\_bias.view(1, -1, 1).to(dtype) # 应用偏置  
 if delta\_softplus:  
 dts = F.softplus(dts) # 应用softplus激活  
  
 # 维度调整  
 Bs = Bs.unsqueeze(1) if len(Bs.shape) == 3 else Bs  
 Cs = Cs.unsqueeze(1) if len(Cs.shape) == 3 else Cs  
 B, G, N, L = Bs.shape  
 us = us.view(B, G, -1, L).permute(3, 0, 1, 2).to(dtype)  
 dts = dts.view(B, G, -1, L).permute(3, 0, 1, 2).to(dtype)  
 As = As.view(G, -1, N).to(dtype)  
 Bs = Bs.permute(3, 0, 1, 2).to(dtype)  
 Cs = Cs.permute(3, 0, 1, 2).to(dtype)  
  
 oys = [] # 存储输出  
 hprefix = us.new\_zeros((B, G, -1, N), dtype=dtype) # 初始化状态  
 for i in range(0, L, chunksize):  
 ys, hs = selective\_scan\_chunk(  
 us[i:i + chunksize], dts[i:i + chunksize],   
 As, Bs[i:i + chunksize], Cs[i:i + chunksize], hprefix  
 )  
 oys.append(ys) # 收集输出  
 hprefix = hs[-1] # 更新前一个状态  
  
 oys = torch.cat(oys, dim=0) # 合并输出  
 oys = oys.permute(1, 2, 3, 0).view(B, -1, L) # 调整输出形状  
  
 return oys.to(us.dtype) if not return\_last\_state else (oys.to(us.dtype), hprefix.view(B, -1, N).float())  
```  
  
### 代码注释说明：  
1. \*\*函数定义\*\*：`selective\_scan\_easy`是一个选择性扫描函数，主要用于处理时间序列数据的状态更新和输出计算。  
2. \*\*参数说明\*\*：函数接受多个参数，包括输入张量、时间增量、系数矩阵、权重张量等，并且支持可选的偏置和激活函数。  
3. \*\*内部函数\*\*：`selective\_scan\_chunk`负责处理每个数据块的计算，包括状态更新和输出生成。  
4. \*\*数据处理\*\*：在函数开始时，进行数据类型转换和偏置应用，确保输入数据的正确性。  
5. \*\*维度调整\*\*：对输入张量的维度进行调整，以适应后续的计算。  
6. \*\*输出收集\*\*：在循环中收集每个块的输出，并在最后合并为最终结果。  
  
以上是对代码的简化和详细注释，保留了核心逻辑和功能。```

这个程序文件 `test\_selective\_scan\_easy.py` 实现了一个名为 `selective\_scan\_easy` 的函数，该函数主要用于处理时间序列数据的选择性扫描。该函数使用了 PyTorch 框架，结合了自动微分的功能，适用于深度学习中的一些应用场景，如序列建模和状态空间模型。  
  
程序的主要部分包括以下几个方面：  
  
首先，程序导入了必要的库，包括 `torch` 和 `pytest`，以及一些用于张量操作的函数，如 `einsum` 和 `rearrange`。这些库和函数为后续的计算提供了基础。  
  
接下来，定义了 `selective\_scan\_easy` 函数，该函数接受多个参数，包括输入张量 `us`、时间增量 `dts`、矩阵 `As`、`Bs`、`Cs` 和 `Ds`，以及一些可选参数，如 `delta\_bias` 和 `delta\_softplus`。函数内部首先定义了一个嵌套函数 `selective\_scan\_chunk`，用于处理输入数据的一个块，执行选择性扫描的核心计算。  
  
在 `selective\_scan\_chunk` 函数中，使用了张量的逐元素操作和累积和计算，以实现状态的更新和输出的生成。通过 `torch.einsum` 进行高效的张量运算，避免了显式的循环，从而提高了计算效率。  
  
在 `selective\_scan\_easy` 函数的主体部分，首先对输入数据进行格式化和类型转换，然后通过分块处理的方式逐步计算输出。每次处理一个块的数据，并将结果累积到最终输出中。最后，函数返回处理后的输出结果，支持返回最后的状态。  
  
程序还定义了一个 `SelectiveScanEasy` 类，继承自 `torch.autograd.Function`，实现了自定义的前向和反向传播逻辑。这个类的 `forward` 方法调用了 `selective\_scan\_easy` 函数，并保存必要的中间结果以供反向传播使用。`backward` 方法则实现了对输入张量的梯度计算，确保在训练过程中能够正确更新模型参数。  
  
此外，程序中还定义了一些测试函数，使用 `pytest` 框架进行单元测试。这些测试验证了 `selective\_scan\_easy` 函数的正确性和性能，确保其在不同输入条件下能够产生预期的输出。  
  
最后，程序的入口部分包含了一个示例测试的调用，展示了如何使用定义的测试函数进行功能验证。  
  
总的来说，这个程序文件实现了一个高效的选择性扫描算法，结合了深度学习中的自动微分功能，适用于处理复杂的序列数据，并通过测试确保了其正确性和可靠性。

### 程序整体功能和构架概括  
  
该项目包含多个模块，每个模块实现了特定的功能，主要集中在计算机视觉和深度学习领域。整体架构由以下几个部分组成：  
  
1. \*\*Swin Transformer\*\*：实现了Swin Transformer模型，主要用于图像处理和特征提取，具有层次化的特征表示能力，适用于各种视觉任务。  
  
2. \*\*自适应归一化模块\*\*：`prepbn.py` 文件中的模块提供了灵活的归一化策略，旨在提高模型的训练稳定性和性能。  
  
3. \*\*自适应特征金字塔网络\*\*：`afpn.py` 文件实现了自适应特征金字塔网络，增强了多尺度特征的融合能力，适用于目标检测和图像分割等任务。  
  
4. \*\*选择性扫描算法\*\*：`test\_selective\_scan\_easy.py` 文件实现了选择性扫描算法，结合了自动微分功能，适用于时间序列数据的处理。  
  
这些模块通过清晰的接口和结构化的设计，便于扩展和修改，支持多种计算机视觉任务的实现。  
  
### 文件功能整理表  
  
| 文件名 | 功能描述 |  
|---------------------------------|----------------------------------------------------------------------------------------------|  
| `SwinTransformer.py` | 实现Swin Transformer模型，包含多个类和函数，用于图像处理和特征提取，支持多层次特征表示。 |  
| `prepbn.py` | 定义自适应归一化模块（RepBN和LinearNorm），提供灵活的归一化策略以提升模型性能和稳定性。 |  
| `afpn.py` | 实现自适应特征金字塔网络（AFPN），增强多尺度特征的融合能力，适用于目标检测和图像分割。 |  
| `test\_selective\_scan\_easy.py` | 实现选择性扫描算法，结合自动微分功能，处理时间序列数据，包含自定义的前向和反向传播逻辑。 |  
  
这个表格总结了每个文件的主要功能，便于理解项目的整体结构和各个模块的作用。

注意：由于此博客编辑较早，上面“10.YOLOv11核心改进源码讲解”中部分代码可能会优化升级，仅供参考学习，以“11.完整训练+Web前端界面+200+种全套创新点源码、数据集获取”的内容为准。