# 改进yolo11-ASF等200+全套创新点大全：大疆无人机热成像人员目标检测系统源码＆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  
  
随着无人机技术的迅猛发展，热成像技术在各类应用场景中的重要性日益凸显，尤其是在安全监控、灾害救援和环境监测等领域。大疆无人机作为行业的领先者，其搭载的热成像设备能够在低光照和复杂环境下有效捕捉目标信息，极大地提升了目标检测的效率和准确性。然而，传统的目标检测算法在处理热成像数据时，往往面临着识别精度不足和实时性差的问题。因此，基于改进YOLOv11的热成像人员目标检测系统的研究具有重要的现实意义。  
  
本项目旨在通过对YOLOv11模型的改进，结合大疆无人机的热成像数据，构建一个高效、精准的人员目标检测系统。我们将使用名为“Thermal\_Overhead”的数据集，该数据集包含1900幅热成像图像，专注于单一类别的目标检测。这一数据集的构建为模型的训练和测试提供了坚实的基础，能够有效支持算法的优化与验证。  
  
通过对YOLOv11的改进，我们将引入更为先进的特征提取技术和优化策略，以提升模型在热成像图像中的目标检测能力。改进后的系统不仅能够提高检测的准确率，还将显著降低误报率，从而为无人机在复杂环境下的应用提供可靠的技术支持。此外，该系统的成功实施将为相关领域的研究提供重要的参考价值，推动热成像技术与无人机应用的深度融合，进而提升公共安全、环境保护等领域的工作效率和响应速度。

## 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“Thermal\_Overhead”，专门为改进YOLOv11的大疆无人机热成像人员目标检测系统而设计。该数据集包含了针对特定目标——人员的热成像图像，旨在提升无人机在复杂环境下的目标检测能力。数据集中仅包含一个类别，即“person”，这使得模型的训练过程更加专注于特定目标的识别与定位。  
  
“Thermal\_Overhead”数据集的构建过程经过精心设计，涵盖了多种环境条件下的热成像数据。这些环境包括城市街道、乡村地区以及各种天气条件，如晴天、阴天和雨天等，确保模型在不同场景下的适应性和鲁棒性。数据集中每张图像均经过标注，确保训练过程中模型能够准确学习到目标的特征和热信号。  
  
为了增强数据集的多样性，数据集还包含了不同距离和角度下拍摄的热成像图像。这种多样性不仅有助于提高模型的泛化能力，还能使其在实际应用中更有效地识别和跟踪目标。此外，数据集中的图像均经过预处理，以保证其质量和一致性，从而为模型训练提供一个高质量的基础。  
  
在训练过程中，YOLOv11模型将利用“Thermal\_Overhead”数据集中的图像数据进行深度学习，学习如何从热成像信号中提取出与人员相关的特征。通过这种方式，模型将不断优化其检测算法，以实现更高的准确率和更快的检测速度。最终，期望该系统能够在实际应用中有效地识别和定位人员，为无人机在安全监控、救援行动等领域的应用提供强有力的支持。

## 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分析和详细注释的结果：  
  
```python  
import torch  
import torch.nn as nn  
import torch.nn.functional as F  
  
class LayerNorm(nn.Module):  
 """ 自定义的LayerNorm层，支持两种数据格式：channels\_last（默认）和channels\_first。  
 channels\_last对应的输入形状为(batch\_size, height, width, channels)，  
 而channels\_first对应的输入形状为(batch\_size, channels, height, width)。  
 """  
 def \_\_init\_\_(self, normalized\_shape, eps=1e-6, data\_format="channels\_last"):  
 super().\_\_init\_\_()  
 self.weight = nn.Parameter(torch.ones(normalized\_shape)) # 权重参数  
 self.bias = nn.Parameter(torch.zeros(normalized\_shape)) # 偏置参数  
 self.eps = eps # 防止除零的极小值  
 self.data\_format = data\_format # 数据格式  
 if self.data\_format not in ["channels\_last", "channels\_first"]:  
 raise NotImplementedError   
 self.normalized\_shape = (normalized\_shape, )  
   
 def forward(self, x):  
 """ 前向传播函数，根据数据格式进行归一化处理 """  
 if self.data\_format == "channels\_last":  
 return F.layer\_norm(x, self.normalized\_shape, self.weight, self.bias, self.eps)  
 elif self.data\_format == "channels\_first":  
 u = x.mean(1, keepdim=True) # 计算均值  
 s = (x - u).pow(2).mean(1, keepdim=True) # 计算方差  
 x = (x - u) / torch.sqrt(s + self.eps) # 归一化  
 x = self.weight[:, None, None] \* x + self.bias[:, None, None] # 应用权重和偏置  
 return x  
  
class Block(nn.Module):  
 """ ConvNeXtV2的基本块，包含深度可分离卷积、归一化、激活函数等 """  
 def \_\_init\_\_(self, dim, drop\_path=0.):  
 super().\_\_init\_\_()  
 self.dwconv = nn.Conv2d(dim, dim, kernel\_size=7, padding=3, groups=dim) # 深度卷积  
 self.norm = LayerNorm(dim, eps=1e-6) # 归一化层  
 self.pwconv1 = nn.Linear(dim, 4 \* dim) # 1x1卷积（点卷积）  
 self.act = nn.GELU() # 激活函数  
 self.pwconv2 = nn.Linear(4 \* dim, dim) # 1x1卷积（点卷积）  
 self.drop\_path = nn.Identity() if drop\_path <= 0. else DropPath(drop\_path) # 随机深度  
  
 def forward(self, x):  
 """ 前向传播函数 """  
 input = x # 保存输入以便后续残差连接  
 x = self.dwconv(x) # 深度卷积  
 x = self.norm(x.permute(0, 2, 3, 1)) # 归一化  
 x = self.pwconv1(x) # 点卷积  
 x = self.act(x) # 激活  
 x = self.pwconv2(x) # 反向点卷积  
 x = x.permute(0, 3, 1, 2) # 恢复原始形状  
 x = input + self.drop\_path(x) # 残差连接  
 return x  
  
class ConvNeXtV2(nn.Module):  
 """ ConvNeXt V2模型 """  
 def \_\_init\_\_(self, in\_chans=3, num\_classes=1000,   
 depths=[3, 3, 9, 3], dims=[96, 192, 384, 768],   
 drop\_path\_rate=0., head\_init\_scale=1.):  
 super().\_\_init\_\_()  
 self.downsample\_layers = nn.ModuleList() # 下采样层  
 # Stem层  
 stem = nn.Sequential(  
 nn.Conv2d(in\_chans, dims[0], kernel\_size=4, stride=4),  
 LayerNorm(dims[0], eps=1e-6, data\_format="channels\_first")  
 )  
 self.downsample\_layers.append(stem)  
 # 添加下采样层  
 for i in range(3):  
 downsample\_layer = nn.Sequential(  
 LayerNorm(dims[i], eps=1e-6, data\_format="channels\_first"),  
 nn.Conv2d(dims[i], dims[i+1], kernel\_size=2, stride=2),  
 )  
 self.downsample\_layers.append(downsample\_layer)  
  
 self.stages = nn.ModuleList() # 特征分辨率阶段  
 dp\_rates = [x.item() for x in torch.linspace(0, drop\_path\_rate, sum(depths))]   
 cur = 0  
 for i in range(4):  
 stage = nn.Sequential(  
 \*[Block(dim=dims[i], drop\_path=dp\_rates[cur + j]) for j in range(depths[i])]  
 )  
 self.stages.append(stage)  
 cur += depths[i]  
  
 self.norm = nn.LayerNorm(dims[-1], eps=1e-6) # 最终归一化层  
 self.head = nn.Linear(dims[-1], num\_classes) # 分类头  
  
 def forward(self, x):  
 """ 前向传播函数 """  
 res = []  
 for i in range(4):  
 x = self.downsample\_layers[i](x) # 下采样  
 x = self.stages[i](x) # 特征提取  
 res.append(x) # 保存每个阶段的输出  
 return res  
  
def convnextv2\_tiny(weights='', \*\*kwargs):  
 """ 创建ConvNeXtV2 Tiny模型并加载权重 """  
 model = ConvNeXtV2(depths=[3, 3, 9, 3], dims=[96, 192, 384, 768], \*\*kwargs)  
 if weights:  
 model.load\_state\_dict(torch.load(weights)['model']) # 加载权重  
 return model  
```  
  
### 代码分析与注释总结  
1. \*\*LayerNorm\*\*: 自定义的层归一化实现，支持两种数据格式，适用于不同的输入形状。  
2. \*\*Block\*\*: ConvNeXtV2的基本构建块，包含深度卷积、归一化、激活和残差连接。  
3. \*\*ConvNeXtV2\*\*: 主模型类，包含多个下采样层和特征提取阶段，构建了整个网络的结构。  
4. \*\*convnextv2\_tiny\*\*: 用于创建特定配置的ConvNeXtV2模型并加载预训练权重的函数。  
  
通过以上分析，可以看出这些核心部分构成了ConvNeXtV2模型的基础，提供了深度学习中的常用模块和功能。```

这个文件定义了一个名为 `ConvNeXtV2` 的深度学习模型，主要用于图像分类任务。该模型是基于卷积神经网络（CNN）的架构，结合了一些新的设计理念，以提高性能和效率。  
  
首先，文件中导入了必要的库，包括 PyTorch 和一些辅助函数。接着，定义了几个重要的类。`LayerNorm` 类实现了层归一化，支持两种数据格式：`channels\_last` 和 `channels\_first`。这使得模型可以灵活处理不同格式的输入数据。`GRN` 类实现了全局响应归一化（Global Response Normalization），通过计算输入的 L2 范数来进行归一化，增强了模型的表达能力。  
  
`Block` 类是 ConvNeXtV2 的基本构建块，包含了深度可分离卷积、层归一化、点卷积、激活函数（GELU）和 GRN。这个模块的设计使得信息能够在不同层之间有效传递，并且通过随机深度（Drop Path）技术增加了模型的鲁棒性。  
  
`ConvNeXtV2` 类是整个模型的核心，构造函数中定义了输入通道数、分类头的类别数、每个阶段的块数、特征维度等参数。模型的前端包括一个初始卷积层和多个下采样层，后面是多个特征分辨率阶段，每个阶段由多个 `Block` 组成。最后，模型还包括一个层归一化和一个线性分类头。  
  
模型的权重初始化通过 `\_init\_weights` 方法完成，使用了截断正态分布进行卷积和线性层的权重初始化。`forward` 方法定义了模型的前向传播过程，逐层处理输入数据，并在每个阶段保存输出结果。  
  
文件的最后部分定义了一些函数，用于创建不同规模的 ConvNeXtV2 模型（如 atto、femto、pico 等），这些函数允许用户加载预训练的权重，以便在特定任务上进行微调。  
  
整体来看，这个文件实现了一个灵活且高效的卷积神经网络架构，结合了现代深度学习中的一些先进技术，适用于图像分类等任务。

```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  
from einops import rearrange  
  
class RFAConv(nn.Module):  
 def \_\_init\_\_(self, in\_channel, out\_channel, kernel\_size, stride=1):  
 super().\_\_init\_\_()  
 self.kernel\_size = kernel\_size  
  
 # 用于获取权重的卷积模块  
 self.get\_weight = nn.Sequential(  
 nn.AvgPool2d(kernel\_size=kernel\_size, padding=kernel\_size // 2, stride=stride),  
 nn.Conv2d(in\_channel, in\_channel \* (kernel\_size \*\* 2), kernel\_size=1, groups=in\_channel, bias=False)  
 )  
   
 # 用于生成特征的卷积模块  
 self.generate\_feature = nn.Sequential(  
 nn.Conv2d(in\_channel, in\_channel \* (kernel\_size \*\* 2), kernel\_size=kernel\_size, padding=kernel\_size // 2, stride=stride, groups=in\_channel, bias=False),  
 nn.BatchNorm2d(in\_channel \* (kernel\_size \*\* 2)),  
 nn.ReLU()  
 )  
   
 # 最终的卷积操作  
 self.conv = nn.Conv2d(in\_channel, out\_channel, kernel\_size=kernel\_size, stride=kernel\_size)  
  
 def forward(self, x):  
 b, c = x.shape[0:2] # 获取输入的批次大小和通道数  
 weight = self.get\_weight(x) # 获取权重  
 h, w = weight.shape[2:] # 获取特征图的高和宽  
   
 # 对权重进行softmax处理  
 weighted = weight.view(b, c, self.kernel\_size \*\* 2, h, w).softmax(2) # b c\*kernel\*\*2, h, w  
   
 # 生成特征  
 feature = self.generate\_feature(x).view(b, c, self.kernel\_size \*\* 2, h, w) # b c\*kernel\*\*2, h, w  
   
 # 加权特征  
 weighted\_data = feature \* weighted  
   
 # 重新排列特征数据  
 conv\_data = rearrange(weighted\_data, 'b c (n1 n2) h w -> b c (h n1) (w n2)', n1=self.kernel\_size, n2=self.kernel\_size)  
   
 return self.conv(conv\_data) # 返回卷积结果  
  
  
class SE(nn.Module):  
 def \_\_init\_\_(self, in\_channel, ratio=16):  
 super(SE, self).\_\_init\_\_()  
 self.gap = nn.AdaptiveAvgPool2d((1, 1)) # 全局平均池化  
 self.fc = nn.Sequential(  
 nn.Linear(in\_channel, ratio, bias=False), # 从 c -> c/r  
 nn.ReLU(),  
 nn.Linear(ratio, in\_channel, bias=False), # 从 c/r -> c  
 nn.Sigmoid()  
 )  
  
 def forward(self, x):  
 b, c = x.shape[0:2] # 获取输入的批次大小和通道数  
 y = self.gap(x).view(b, c) # 进行全局平均池化并调整形状  
 y = self.fc(y).view(b, c, 1, 1) # 通过全连接层并调整形状  
 return y # 返回通道注意力权重  
  
  
class RFCBAMConv(nn.Module):  
 def \_\_init\_\_(self, in\_channel, out\_channel, kernel\_size=3, stride=1):  
 super().\_\_init\_\_()  
 assert kernel\_size % 2 == 1, "the kernel\_size must be odd." # 确保卷积核大小为奇数  
 self.kernel\_size = kernel\_size  
   
 # 特征生成模块  
 self.generate = nn.Sequential(  
 nn.Conv2d(in\_channel, in\_channel \* (kernel\_size \*\* 2), kernel\_size, padding=kernel\_size // 2, stride=stride, groups=in\_channel, bias=False),  
 nn.BatchNorm2d(in\_channel \* (kernel\_size \*\* 2)),  
 nn.ReLU()  
 )  
   
 # 权重获取模块  
 self.get\_weight = nn.Sequential(nn.Conv2d(2, 1, kernel\_size=3, padding=1, bias=False), nn.Sigmoid())  
 self.se = SE(in\_channel) # 通道注意力模块  
  
 self.conv = nn.Conv2d(in\_channel, out\_channel, kernel\_size=kernel\_size, stride=kernel\_size)  
  
 def forward(self, x):  
 b, c = x.shape[0:2] # 获取输入的批次大小和通道数  
 channel\_attention = self.se(x) # 获取通道注意力  
 generate\_feature = self.generate(x) # 生成特征  
  
 h, w = generate\_feature.shape[2:] # 获取特征图的高和宽  
 generate\_feature = generate\_feature.view(b, c, self.kernel\_size \*\* 2, h, w) # 调整形状  
   
 # 重新排列特征数据  
 generate\_feature = rearrange(generate\_feature, 'b c (n1 n2) h w -> b c (h n1) (w n2)', n1=self.kernel\_size, n2=self.kernel\_size)  
   
 # 加权特征  
 unfold\_feature = generate\_feature \* channel\_attention  
   
 # 计算最大和平均特征  
 max\_feature, \_ = torch.max(generate\_feature, dim=1, keepdim=True)  
 mean\_feature = torch.mean(generate\_feature, dim=1, keepdim=True)  
   
 # 获取感受野注意力  
 receptive\_field\_attention = self.get\_weight(torch.cat((max\_feature, mean\_feature), dim=1))  
   
 # 返回卷积结果  
 conv\_data = unfold\_feature \* receptive\_field\_attention  
 return self.conv(conv\_data)  
```  
  
### 代码说明：  
1. \*\*RFAConv\*\*: 该类实现了一种基于卷积的特征加权机制。通过平均池化和卷积生成权重，并利用这些权重对特征进行加权处理，最后通过卷积层输出结果。  
   
2. \*\*SE (Squeeze-and-Excitation)\*\*: 该类实现了通道注意力机制，通过全局平均池化和全连接层来生成通道权重，从而增强重要特征的表达。  
  
3. \*\*RFCBAMConv\*\*: 该类结合了特征生成和通道注意力机制，进一步通过最大池化和平均池化来计算感受野注意力，并对生成的特征进行加权，最后通过卷积层输出结果。  
  
以上代码展示了如何通过不同的模块和机制来增强卷积神经网络的特征提取能力。```

这个程序文件 `RFAConv.py` 定义了一些基于卷积神经网络的模块，主要包括 RFAConv、RFCBAMConv 和 RFCAConv。这些模块使用了不同的卷积和注意力机制，以增强特征提取的能力。  
  
首先，文件中导入了必要的库，包括 PyTorch 和 einops。接着，定义了两个激活函数类：`h\_sigmoid` 和 `h\_swish`。`h\_sigmoid` 是一种带有 ReLU6 的激活函数，而 `h\_swish` 则是将输入乘以 `h\_sigmoid` 的结果。  
  
`RFAConv` 类是一个自定义的卷积模块，包含了一个生成特征的卷积层和一个加权卷积层。其构造函数中定义了获取权重的操作，通过平均池化和卷积来实现。前向传播中，首先计算输入特征的权重，然后生成特征，接着将加权后的特征进行重排，最后通过卷积层输出结果。  
  
`SE` 类实现了 Squeeze-and-Excitation（SE）机制，通过全局平均池化和全连接层来计算通道注意力。前向传播中，首先对输入进行全局平均池化，然后通过全连接层得到通道权重。  
  
`RFCBAMConv` 类结合了 SE 机制和 RFAConv 的特性。它在构造函数中定义了生成特征的卷积层和用于计算通道注意力的 SE 模块。在前向传播中，首先计算通道注意力，然后生成特征，接着计算最大值和均值特征，利用这些特征计算接收场注意力，最后将结果通过卷积层输出。  
  
`RFCAConv` 类是一个更复杂的模块，结合了 RFAConv 和通道注意力机制。它在构造函数中定义了生成特征的卷积层，并通过自适应平均池化计算特征的高度和宽度信息。在前向传播中，生成特征后，分别对高度和宽度进行池化，接着通过全连接层计算通道注意力，并将生成的特征与注意力相乘，最后通过卷积层输出结果。  
  
总体来说，这个文件实现了一些先进的卷积模块，利用不同的特征生成和注意力机制来提高网络的性能，适用于图像处理和计算机视觉任务。

```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  
import torch.nn.functional as F  
  
# 定义一个相对位置的函数，用于生成卷积核的坐标  
def rel\_pos(kernel\_size):  
 # 生成从-1到1的线性空间，作为卷积核的坐标  
 tensors = [torch.linspace(-1, 1, steps=kernel\_size) for \_ in range(2)]  
 kernel\_coord = torch.stack(torch.meshgrid(\*tensors), dim=-0) # 生成网格坐标  
 kernel\_coord = kernel\_coord.unsqueeze(0) # 增加一个维度  
 return kernel\_coord  
  
# 定义自定义卷积层  
class SMPConv(nn.Module):  
 def \_\_init\_\_(self, planes, kernel\_size, n\_points, stride, padding, groups):  
 super().\_\_init\_\_()  
  
 self.planes = planes # 输出通道数  
 self.kernel\_size = kernel\_size # 卷积核大小  
 self.n\_points = n\_points # 卷积核中的点数  
 self.init\_radius = 2 \* (2/kernel\_size) # 初始化半径  
  
 # 生成卷积核坐标  
 kernel\_coord = rel\_pos(kernel\_size)  
 self.register\_buffer('kernel\_coord', kernel\_coord) # 注册为buffer，不会被优化  
  
 # 初始化权重坐标  
 weight\_coord = torch.empty(1, n\_points, 2)  
 nn.init.trunc\_normal\_(weight\_coord, std=0.2, a=-1., b=1.) # 截断正态分布初始化  
 self.weight\_coord = nn.Parameter(weight\_coord) # 权重坐标为可学习参数  
  
 # 初始化半径  
 self.radius = nn.Parameter(torch.empty(1, n\_points).unsqueeze(-1).unsqueeze(-1))  
 self.radius.data.fill\_(value=self.init\_radius) # 填充初始值  
  
 # 初始化卷积权重  
 weights = torch.empty(1, planes, n\_points)  
 nn.init.trunc\_normal\_(weights, std=.02) # 权重初始化  
 self.weights = nn.Parameter(weights) # 权重为可学习参数  
  
 def forward(self, x):  
 # 生成卷积核并进行卷积操作  
 kernels = self.make\_kernels().unsqueeze(1) # 生成卷积核  
 x = x.contiguous() # 确保输入是连续的  
 kernels = kernels.contiguous() # 确保卷积核是连续的  
  
 # 根据输入数据类型选择合适的卷积实现  
 if x.dtype == torch.float32:  
 x = \_DepthWiseConv2dImplicitGEMMFP32.apply(x, kernels) # FP32卷积  
 elif x.dtype == torch.float16:  
 x = \_DepthWiseConv2dImplicitGEMMFP16.apply(x, kernels) # FP16卷积  
 else:  
 raise TypeError("Only support fp32 and fp16, get {}".format(x.dtype)) # 抛出异常  
 return x   
  
 def make\_kernels(self):  
 # 计算卷积核  
 diff = self.weight\_coord.unsqueeze(-2) - self.kernel\_coord.reshape(1, 2, -1).transpose(1, 2) # 计算差值  
 diff = diff.transpose(2, 3).reshape(1, self.n\_points, 2, self.kernel\_size, self.kernel\_size) # 变形  
 diff = F.relu(1 - torch.sum(torch.abs(diff), dim=2) / self.radius) # 计算ReLU激活后的差值  
   
 # 计算最终的卷积核  
 kernels = torch.matmul(self.weights, diff.reshape(1, self.n\_points, -1)) # 加权求和  
 kernels = kernels.reshape(1, self.planes, \*self.kernel\_coord.shape[2:]) # 变形为卷积核形状  
 kernels = kernels.squeeze(0) # 去掉多余的维度  
 kernels = torch.flip(kernels.permute(0, 2, 1), dims=(1,)) # 反转维度  
 return kernels  
  
# 定义SMPBlock模块  
class SMPBlock(nn.Module):  
 def \_\_init\_\_(self, in\_channels, dw\_channels, lk\_size, drop\_path, n\_points=None, n\_points\_divide=4):  
 super().\_\_init\_\_()  
 # 定义逐点卷积和SMP卷积  
 self.pw1 = conv\_bn\_relu(in\_channels, dw\_channels, 1, 1, 0, groups=1)  
 self.pw2 = conv\_bn(dw\_channels, in\_channels, 1, 1, 0, groups=1)  
 self.large\_kernel = SMPCNN(in\_channels=dw\_channels, out\_channels=dw\_channels, kernel\_size=lk\_size,  
 stride=1, groups=dw\_channels, n\_points=n\_points, n\_points\_divide=n\_points\_divide)  
 self.lk\_nonlinear = nn.ReLU() # 激活函数  
 self.drop\_path = DropPath(drop\_path) if drop\_path > 0. else nn.Identity() # DropPath  
  
 def forward(self, x):  
 # 前向传播  
 out = self.pw1(x) # 逐点卷积  
 out = self.large\_kernel(out) # 大卷积  
 out = self.lk\_nonlinear(out) # 激活  
 out = self.pw2(out) # 逐点卷积  
 return x + self.drop\_path(out) # 残差连接  
```  
  
### 代码核心部分解释：  
1. \*\*SMPConv\*\*：自定义卷积层，使用相对位置的卷积核。通过生成卷积核坐标和权重坐标来计算卷积操作。  
2. \*\*make\_kernels\*\*：生成卷积核的核心函数，计算卷积核的差值并进行加权求和。  
3. \*\*SMPBlock\*\*：构建一个包含逐点卷积和SMP卷积的模块，使用残差连接和DropPath技术。```

这个程序文件 `SMPConv.py` 定义了一些用于深度学习的卷积神经网络模块，主要是基于 PyTorch 框架实现的。文件中包含了多个类和函数，主要包括 `SMPConv`、`SMPCNN`、`SMPCNN\_ConvFFN` 和 `SMPBlock`，这些模块结合了传统卷积和一些新的卷积策略，以提高模型的表现。  
  
首先，文件导入了必要的库，包括 PyTorch 的核心模块、神经网络模块和一些实用功能。接着，定义了一个辅助函数 `rel\_pos`，用于生成相对位置的坐标，这在卷积操作中可能用于计算卷积核的位置。  
  
`SMPConv` 类是这个文件的核心部分，构造函数中初始化了一些参数，包括输出通道数、卷积核大小、点数、步幅、填充和分组数。它还计算了卷积核的相对位置，并初始化了权重和半径。`forward` 方法定义了前向传播的过程，根据输入数据的类型（32位或16位浮点数）选择合适的深度可分离卷积实现。`make\_kernels` 方法则用于生成卷积核，结合了权重和相对位置的差异，使用 ReLU 激活函数来处理这些差异。  
  
`get\_conv2d` 函数根据输入参数决定返回 `SMPConv` 或标准的 `nn.Conv2d`，这使得用户可以根据需求灵活选择卷积实现。接下来的 `get\_bn` 和 `conv\_bn` 函数用于创建批归一化层和组合卷积与批归一化的模块。  
  
`SMPCNN` 类是一个组合卷积模块，它结合了 `SMPConv` 和一个小卷积核的卷积操作，目的是通过不同大小的卷积核提取特征。`SMPCNN\_ConvFFN` 类则实现了一个前馈神经网络，包含两个逐点卷积层和一个非线性激活函数（GELU），并通过残差连接将输入与输出相加。  
  
最后，`SMPBlock` 类实现了一个完整的块结构，包含了逐点卷积、SMP卷积和残差连接，适用于构建更复杂的网络架构。整个文件展示了如何通过组合不同的卷积策略和结构来设计高效的神经网络模块，以适应不同的任务需求。

```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  
  
# 引入自定义的归一化模块  
from .prepbn import RepBN, LinearNorm  
from ..modules.transformer import TransformerEncoderLayer  
  
# 定义一个包含自定义归一化的线性归一化  
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)  
   
 # 使用自定义的线性归一化  
 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))  
 # 将输出张量恢复为原始形状[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的倍数以适应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-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-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]  
```  
  
### 代码注释说明：  
1. \*\*导入模块\*\*：导入必要的PyTorch模块和自定义的归一化模块。  
2. \*\*自定义归一化\*\*：使用`functools.partial`创建一个包含层归一化和自定义归一化的线性归一化。  
3. \*\*TransformerEncoderLayer\_RepBN类\*\*：继承自`TransformerEncoderLayer`，并在初始化中添加自定义的归一化层。  
4. \*\*AIFI\_RepBN类\*\*：继承自`TransformerEncoderLayer\_RepBN`，定义了AIFI变换器层的结构和前向传播逻辑。  
5. \*\*前向传播\*\*：在`forward`方法中，输入张量被展平并与位置嵌入结合，最后恢复为原始形状。  
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 `transformer.py` 定义了一个基于 Transformer 的编码层，主要用于处理图像数据，采用了特定的归一化方法（RepBN）和位置编码方式。代码中引入了 PyTorch 库，利用其深度学习功能构建神经网络模型。  
  
首先，文件导入了必要的库，包括 PyTorch 的核心模块和一些自定义的模块。`RepBN` 和 `LinearNorm` 是自定义的归一化层，`TransformerEncoderLayer` 是基础的 Transformer 编码层，而 `AIFI` 可能是某种特定的模块。  
  
接下来，定义了一个名为 `TransformerEncoderLayer\_RepBN` 的类，它继承自 `TransformerEncoderLayer`。在初始化方法中，调用了父类的构造函数，并使用 `linearnorm` 创建了两个归一化层 `norm1` 和 `norm2`，这些归一化层使用了自定义的归一化方法。  
  
然后，定义了 `AIFI\_RepBN` 类，它继承自 `TransformerEncoderLayer\_RepBN`，并且在其构造函数中初始化了一些参数，包括通道数、隐藏层大小、头数、丢弃率、激活函数和归一化方式。这里的激活函数默认使用的是 GELU。  
  
在 `AIFI\_RepBN` 类中，重写了 `forward` 方法，这是模型前向传播的核心部分。首先获取输入张量的形状，然后调用 `build\_2d\_sincos\_position\_embedding` 方法生成二维的正弦余弦位置编码。接着，将输入张量的形状从 `[B, C, H, W]` 转换为 `[B, HxW, C]`，并将位置编码传递给父类的 `forward` 方法进行处理。最后，将输出结果的形状恢复为 `[B, C, H, W]`。  
  
`build\_2d\_sincos\_position\_embedding` 是一个静态方法，用于生成二维的正弦余弦位置编码。该方法首先检查嵌入维度是否能被4整除，然后生成宽度和高度的网格，并计算相应的正弦和余弦值，最终返回一个包含位置编码的张量。  
  
总体来说，这个文件实现了一个增强的 Transformer 编码层，结合了自定义的归一化和位置编码方法，适用于处理图像数据的深度学习任务。

### 整体功能和架构概括  
  
该工程包含多个模块，主要围绕深度学习中的卷积神经网络（CNN）和 Transformer 架构，旨在提升图像处理和分类任务的性能。整体架构可以分为以下几个部分：  
  
1. \*\*卷积模块\*\*：通过不同的卷积策略（如 `ConvNeXtV2`、`RFAConv` 和 `SMPConv`）来提取图像特征。这些模块结合了深度可分离卷积、注意力机制和归一化方法，以提高特征提取的效率和效果。  
  
2. \*\*Transformer 编码层\*\*：实现了一个增强的 Transformer 编码层，专门用于处理图像数据，结合了自定义的归一化方法和位置编码，以便在图像分类等任务中更好地捕捉空间信息。  
  
3. \*\*灵活性和可扩展性\*\*：每个模块都设计得相对独立，便于组合和扩展，用户可以根据具体任务需求选择不同的卷积或编码模块。  
  
### 文件功能整理表  
  
| 文件名 | 功能描述 |  
|----------------------|------------------------------------------------------------------------------------------------|  
| `convnextv2.py` | 定义了 ConvNeXtV2 模型，包含卷积层、特征提取块和分类头，适用于图像分类任务。 |  
| `RFAConv.py` | 实现了 RFAConv 模块，结合了特征生成和注意力机制，增强了卷积操作的特征提取能力。 |  
| `SMPConv.py` | 定义了 SMPConv 模块及其相关结构，结合了不同卷积策略和激活函数，用于高效的特征提取。 |  
| `transformer.py` | 实现了基于 Transformer 的编码层，结合了自定义的归一化和位置编码，适用于图像数据处理。 |  
  
这个工程通过整合多种先进的深度学习技术，提供了一个灵活且高效的框架，适用于多种计算机视觉任务。

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。