# 改进yolo11-SCConv等200+全套创新点大全：空中目标检测系统源码＆数据集全套

## 3.背景

研究背景与意义  
  
随着无人机技术的迅猛发展，空中目标检测在军事、民用和环境监测等领域的应用日益广泛。无人机不仅可以用于监视和侦察，还能在灾害救援、交通监控以及生态保护等方面发挥重要作用。因此，构建一个高效、准确的空中目标检测系统显得尤为重要。现有的目标检测算法在处理复杂环境下的目标识别时，往往面临着准确率低、实时性差等问题，亟需通过改进算法来提升其性能。  
  
本研究旨在基于改进的YOLOv11算法，开发一个针对空中目标的检测系统。该系统将专注于识别三类目标：飞机、鸟类和无人机。这三类目标在空中场景中具有不同的特征和行为模式，针对性地优化检测算法，将有助于提高系统的整体识别能力。为此，我们将利用包含4500张图像的“Drone\_Bird\_Aircraft”数据集进行训练和测试。该数据集不仅涵盖了多种空中目标，还提供了丰富的标注信息，为模型的训练提供了坚实的基础。  
  
在技术层面，YOLO系列算法以其高效的实时检测能力和较高的准确率受到广泛关注。通过对YOLOv11的改进，我们将结合最新的深度学习技术，探索更为精细的特征提取和目标定位方法，以提升模型在复杂空中环境中的表现。此外，针对不同目标的特征差异，我们将设计特定的训练策略，以实现更为精准的目标检测。  
  
本研究的成果不仅将推动空中目标检测技术的发展，还将为相关领域的应用提供理论支持和实践指导。通过构建一个高效的空中目标检测系统，我们希望能够为无人机的智能化应用提供更为可靠的技术保障，从而促进无人机技术的广泛应用与发展。

## 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专注于空中目标检测，旨在为改进YOLOv11算法提供丰富的训练素材。数据集的主题为“Drone\_Bird\_Aircraft”，涵盖了三种主要类别：飞机（aircraft）、鸟类（bird）和无人机（drone）。这些类别的选择不仅反映了现代空中交通的多样性，也为算法在复杂环境中的应用提供了广泛的测试场景。  
  
在数据集的构建过程中，研究团队精心收集了来自不同环境和条件下的图像，以确保数据的多样性和代表性。飞机类图像包括商用飞机、私人飞机及军用飞机等多种类型，涵盖了不同的飞行高度和角度；鸟类图像则选取了多种常见鸟类的飞行状态，旨在模拟自然环境中的空中目标；无人机类图像则包括了多种型号的无人机，反映了当前无人机技术的快速发展。  
  
数据集的设计考虑到了不同光照、天气条件及背景复杂度对目标检测的影响，使得训练出的模型能够在各种实际应用场景中保持高效的检测性能。通过对这三类目标的深入分析与研究，数据集不仅为算法的训练提供了基础，也为后续的性能评估和优化奠定了坚实的基础。  
  
总之，本项目的数据集在类别数量和多样性上均表现出色，旨在为空中目标检测领域的研究提供有力支持。通过使用这一数据集，研究人员希望能够显著提升YOLOv11在空中目标检测任务中的准确性和鲁棒性，为无人机监控、航空安全等应用领域的进一步发展做出贡献。

## 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 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)  
   
 # 反转小波的分解高通和低通滤波器  
 dec\_hi = torch.tensor(w.dec\_hi[::-1], dtype=type)  
 dec\_lo = torch.tensor(w.dec\_lo[::-1], dtype=type)  
   
 # 创建分解滤波器  
 dec\_filters = torch.stack([  
 dec\_lo.unsqueeze(0) \* dec\_lo.unsqueeze(1),  
 dec\_lo.unsqueeze(0) \* dec\_hi.unsqueeze(1),  
 dec\_hi.unsqueeze(0) \* dec\_lo.unsqueeze(1),  
 dec\_hi.unsqueeze(0) \* dec\_hi.unsqueeze(1)  
 ], dim=0)  
  
 # 扩展滤波器以适应输入通道数  
 dec\_filters = dec\_filters[:, None].repeat(in\_size, 1, 1, 1)  
  
 # 反转小波的重构高通和低通滤波器  
 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])  
   
 # 创建重构滤波器  
 rec\_filters = torch.stack([  
 rec\_lo.unsqueeze(0) \* rec\_lo.unsqueeze(1),  
 rec\_lo.unsqueeze(0) \* rec\_hi.unsqueeze(1),  
 rec\_hi.unsqueeze(0) \* rec\_lo.unsqueeze(1),  
 rec\_hi.unsqueeze(0) \* rec\_hi.unsqueeze(1)  
 ], dim=0)  
  
 # 扩展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  
 # 使用小波滤波器进行卷积  
 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  
 # 使用小波滤波器进行转置卷积  
 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) # 计算梯度  
 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 # 输入和输出通道数必须相同  
  
 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)  
 self.iwt\_filter = nn.Parameter(self.iwt\_filter, requires\_grad=False)  
  
 # 定义基础卷积层  
 self.base\_conv = nn.Conv2d(in\_channels, in\_channels, kernel\_size, padding='same', stride=1, groups=in\_channels, bias=bias)  
  
 def forward(self, x):  
 # 小波变换和逆变换的实现  
 for i in range(self.wt\_levels):  
 x = self.wt\_function(x) # 执行小波变换  
  
 # 进行逆小波变换  
 for i in range(self.wt\_levels-1, -1, -1):  
 x = self.iwt\_function(x) # 执行逆小波变换  
  
 x = self.base\_conv(x) # 通过基础卷积层  
 return x  
```  
  
### 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. \*\*小波变换与逆变换\*\*：`wavelet\_transform` 和 `inverse\_wavelet\_transform` 函数实现了小波变换和逆变换的核心逻辑。  
3. \*\*自定义函数\*\*：`WaveletTransform` 类实现了小波变换的前向和反向传播，支持自动求导。  
4. \*\*小波卷积层\*\*：`WTConv2d` 类是一个卷积层，结合了小波变换的特性，能够在特征提取中利用小波的多分辨率特性。```

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，主要用于图像处理和特征提取。代码中使用了PyTorch框架，结合了小波变换的概念来增强卷积操作的能力。  
  
首先，文件导入了必要的库，包括PyTorch的核心库和小波变换库`pywt`。接着，定义了一个函数`create\_wavelet\_filter`，用于创建小波滤波器。这个函数接受小波类型、输入通道数和输出通道数作为参数，生成相应的小波分解和重构滤波器。这些滤波器通过小波的高通和低通滤波器的组合生成，并根据输入和输出通道数进行扩展。  
  
接下来，定义了两个函数`wavelet\_transform`和`inverse\_wavelet\_transform`，分别用于执行小波变换和逆小波变换。`wavelet\_transform`函数通过卷积操作将输入张量转换为小波域，`inverse\_wavelet\_transform`则将小波域的张量转换回原始域。  
  
随后，定义了两个类`WaveletTransform`和`InverseWaveletTransform`，这两个类继承自`torch.autograd.Function`，实现了小波变换和逆变换的前向和反向传播。它们的`forward`方法调用了之前定义的变换函数，而`backward`方法则计算梯度。  
  
在`WTConv2d`类中，构造函数初始化了小波卷积层的参数，包括输入和输出通道数、卷积核大小、步幅等。它创建了小波滤波器和逆滤波器，并定义了基础卷积层和多个小波卷积层。`forward`方法实现了整个小波卷积的前向传播过程，首先进行小波变换，然后通过一系列卷积操作处理小波系数，最后进行逆小波变换，得到输出。  
  
此外，`\_ScaleModule`类用于实现一个简单的缩放模块，可以对输入进行加权和偏置操作，增强网络的灵活性。  
  
总体来说，这个文件实现了一个创新的卷积层，结合了小波变换的优点，能够更好地捕捉图像中的多尺度特征，适用于图像处理和计算机视觉任务。

```以下是经过简化并注释的核心代码部分，主要保留了 `EMA`, `SimAM`, `SpatialGroupEnhance`, `TopkRouting`, `KVGather`, `QKVLinear`, `BiLevelRoutingAttention` 等类的实现。这些类实现了不同的注意力机制和特征增强模块。  
  
```python  
import torch  
from torch import nn  
import torch.nn.functional as F  
  
class EMA(nn.Module):  
 """ Exponential Moving Average (EMA) Module """  
 def \_\_init\_\_(self, channels, factor=8):  
 super(EMA, self).\_\_init\_\_()  
 self.groups = factor # 将通道分成多个组  
 assert channels // self.groups > 0 # 确保每组有通道  
 self.softmax = nn.Softmax(-1) # Softmax层  
 self.agp = nn.AdaptiveAvgPool2d((1, 1)) # 自适应平均池化  
 self.pool\_h = nn.AdaptiveAvgPool2d((None, 1)) # 针对高度的池化  
 self.pool\_w = nn.AdaptiveAvgPool2d((1, None)) # 针对宽度的池化  
 self.gn = nn.GroupNorm(channels // self.groups, channels // self.groups) # 组归一化  
 self.conv1x1 = nn.Conv2d(channels // self.groups, channels // self.groups, kernel\_size=1) # 1x1卷积  
 self.conv3x3 = nn.Conv2d(channels // self.groups, channels // self.groups, kernel\_size=3, padding=1) # 3x3卷积  
  
 def forward(self, x):  
 b, c, h, w = x.size() # 获取输入的尺寸  
 group\_x = x.reshape(b \* self.groups, -1, h, w) # 重新排列为组  
 x\_h = self.pool\_h(group\_x) # 对高度进行池化  
 x\_w = self.pool\_w(group\_x).permute(0, 1, 3, 2) # 对宽度进行池化并转置  
 hw = self.conv1x1(torch.cat([x\_h, x\_w], dim=2)) # 1x1卷积  
 x\_h, x\_w = torch.split(hw, [h, w], dim=2) # 分割为高度和宽度  
 x1 = self.gn(group\_x \* x\_h.sigmoid() \* x\_w.permute(0, 1, 3, 2).sigmoid()) # 组归一化  
 x2 = self.conv3x3(group\_x) # 3x3卷积  
 # 计算权重  
 x11 = self.softmax(self.agp(x1).reshape(b \* self.groups, -1, 1).permute(0, 2, 1))  
 x12 = x2.reshape(b \* self.groups, c // self.groups, -1)  
 x21 = self.softmax(self.agp(x2).reshape(b \* self.groups, -1, 1).permute(0, 2, 1))  
 x22 = x1.reshape(b \* self.groups, c // self.groups, -1)  
 weights = (torch.matmul(x11, x12) + torch.matmul(x21, x22)).reshape(b \* self.groups, 1, h, w)  
 return (group\_x \* weights.sigmoid()).reshape(b, c, h, w) # 返回加权后的输出  
  
class SimAM(nn.Module):  
 """ Similarity Attention Module (SimAM) """  
 def \_\_init\_\_(self, e\_lambda=1e-4):  
 super(SimAM, self).\_\_init\_\_()  
 self.activaton = nn.Sigmoid() # Sigmoid激活函数  
 self.e\_lambda = e\_lambda # 正则化参数  
  
 def forward(self, x):  
 b, c, h, w = x.size() # 获取输入的尺寸  
 n = w \* h - 1 # 计算样本数  
 x\_minus\_mu\_square = (x - x.mean(dim=[2, 3], keepdim=True)).pow(2) # 计算方差  
 y = x\_minus\_mu\_square / (4 \* (x\_minus\_mu\_square.sum(dim=[2, 3], keepdim=True) / n + self.e\_lambda)) + 0.5 # 计算y  
 return x \* self.activaton(y) # 返回加权后的输出  
  
class SpatialGroupEnhance(nn.Module):  
 """ Spatial Group Enhancement Module """  
 def \_\_init\_\_(self, groups=8):  
 super().\_\_init\_\_()  
 self.groups = groups # 组数  
 self.avg\_pool = nn.AdaptiveAvgPool2d(1) # 自适应平均池化  
 self.weight = nn.Parameter(torch.zeros(1, groups, 1, 1)) # 权重参数  
 self.bias = nn.Parameter(torch.zeros(1, groups, 1, 1)) # 偏置参数  
 self.sig = nn.Sigmoid() # Sigmoid激活函数  
 self.init\_weights() # 初始化权重  
  
 def init\_weights(self):  
 for m in self.modules():  
 if isinstance(m, nn.Conv2d):  
 nn.init.kaiming\_normal\_(m.weight, mode='fan\_out') # Kaiming初始化  
 if m.bias is not None:  
 nn.init.constant\_(m.bias, 0) # 偏置初始化为0  
  
 def forward(self, x):  
 b, c, h, w = x.shape # 获取输入的尺寸  
 x = x.view(b \* self.groups, -1, h, w) # 重新排列为组  
 xn = x \* self.avg\_pool(x) # 计算平均池化  
 xn = xn.sum(dim=1, keepdim=True) # 求和  
 t = xn.view(b \* self.groups, -1) # 重新排列  
 t = t - t.mean(dim=1, keepdim=True) # 减去均值  
 std = t.std(dim=1, keepdim=True) + 1e-5 # 计算标准差  
 t = t / std # 归一化  
 t = t.view(b, self.groups, h, w) # 重新排列  
 t = t \* self.weight + self.bias # 加权  
 t = t.view(b \* self.groups, 1, h, w) # 重新排列  
 x = x \* self.sig(t) # 返回加权后的输出  
 return x.view(b, c, h, w) # 返回原始形状  
  
class TopkRouting(nn.Module):  
 """ Differentiable Top-k Routing Module """  
 def \_\_init\_\_(self, qk\_dim, topk=4):  
 super().\_\_init\_\_()  
 self.topk = topk # Top-k值  
 self.qk\_dim = qk\_dim # 查询和键的维度  
 self.scale = qk\_dim \*\* -0.5 # 缩放因子  
 self.routing\_act = nn.Softmax(dim=-1) # Softmax激活函数  
  
 def forward(self, query: Tensor, key: Tensor) -> Tuple[Tensor]:  
 """ 前向传播 """  
 query\_hat, key\_hat = query, key # 直接使用输入  
 attn\_logit = (query\_hat \* self.scale) @ key\_hat.transpose(-2, -1) # 计算注意力日志  
 topk\_attn\_logit, topk\_index = torch.topk(attn\_logit, k=self.topk, dim=-1) # 获取Top-k  
 r\_weight = self.routing\_act(topk\_attn\_logit) # 计算路由权重  
 return r\_weight, topk\_index # 返回权重和索引  
  
class KVGather(nn.Module):  
 """ Key-Value Gather Module """  
 def \_\_init\_\_(self, mul\_weight='none'):  
 super().\_\_init\_\_()  
 assert mul\_weight in ['none', 'soft', 'hard'] # 确保权重类型有效  
 self.mul\_weight = mul\_weight # 权重类型  
  
 def forward(self, r\_idx: Tensor, r\_weight: Tensor, kv: Tensor):  
 """ 前向传播 """  
 n, p2, w2, c\_kv = kv.size() # 获取输入的尺寸  
 topk = r\_idx.size(-1) # Top-k值  
 topk\_kv = torch.gather(kv.view(n, 1, p2, w2, c\_kv).expand(-1, p2, -1, -1, -1), # 根据路由索引选择kv  
 dim=2,  
 index=r\_idx.view(n, p2, topk, 1, 1).expand(-1, -1, -1, w2, c\_kv)) # 扩展索引  
 if self.mul\_weight == 'soft':  
 topk\_kv = r\_weight.view(n, p2, topk, 1, 1) \* topk\_kv # 软权重  
 return topk\_kv # 返回选择的kv  
  
class QKVLinear(nn.Module):  
 """ QKV Linear Projection Module """  
 def \_\_init\_\_(self, dim, qk\_dim, bias=True):  
 super().\_\_init\_\_()  
 self.qkv = nn.Linear(dim, qk\_dim \* 3, bias=bias) # QKV线性映射  
  
 def forward(self, x):  
 q, kv = self.qkv(x).split([self.qk\_dim, self.qk\_dim \* 2], dim=-1) # 分割为q和kv  
 return q, kv # 返回q和kv  
  
class BiLevelRoutingAttention(nn.Module):  
 """ Bi-Level Routing Attention Module """  
 def \_\_init\_\_(self, dim, num\_heads=8, n\_win=7, qk\_dim=None, topk=4):  
 super().\_\_init\_\_()  
 self.dim = dim # 输入维度  
 self.n\_win = n\_win # 窗口数  
 self.num\_heads = num\_heads # 注意力头数  
 self.qk\_dim = qk\_dim or dim # 查询和键的维度  
 self.router = TopkRouting(qk\_dim=self.qk\_dim, topk=topk) # 路由模块  
 self.qkv = QKVLinear(self.dim, self.qk\_dim) # QKV映射  
  
 def forward(self, x):  
 """ 前向传播 """  
 q, kv = self.qkv(x) # 计算q和kv  
 q\_win = q.mean([2, 3]) # 窗口平均  
 r\_weight, r\_idx = self.router(q\_win, kv) # 路由权重和索引  
 return r\_weight, r\_idx # 返回路由权重和索引  
```  
  
以上代码展示了几个核心模块的实现，包含了详细的中文注释，帮助理解每个模块的功能和工作原理。```

这个程序文件 `attention.py` 定义了一系列用于深度学习模型的注意力机制模块，主要用于图像处理和计算机视觉任务。文件中使用了 PyTorch 框架，包含多个类和函数，每个类实现了一种特定的注意力机制或相关操作。  
  
首先，文件导入了必要的库，包括 `torch`、`torch.nn` 和一些其他模块。接着，定义了一些全局变量和可用的注意力模块名称。  
  
接下来，定义了多个类，每个类实现了不同的注意力机制：  
  
1. \*\*EMA (Exponential Moving Average)\*\*：该类实现了一种基于通道的注意力机制，通过对输入特征图进行分组和池化来增强特征表示。  
  
2. \*\*SimAM (Similarity Attention Module)\*\*：该模块通过计算输入特征的均值和方差来生成注意力权重，并通过 Sigmoid 激活函数进行处理。  
  
3. \*\*SpatialGroupEnhance\*\*：该模块通过自适应平均池化和通道归一化来增强特征图的空间信息。  
  
4. \*\*TopkRouting\*\*：实现了一种可微分的 Top-k 路由机制，用于选择最相关的特征。  
  
5. \*\*KVGather\*\*：该模块根据路由索引和权重从键值对中选择特征。  
  
6. \*\*QKVLinear\*\*：用于生成查询、键和值的线性映射。  
  
7. \*\*BiLevelRoutingAttention\*\*：实现了一种双层路由注意力机制，结合了全局和局部注意力。  
  
8. \*\*BiLevelRoutingAttention\_nchw\*\*：与前一个类类似，但支持 NCHW 格式的输入。  
  
9. \*\*CoordAtt\*\*、\*\*TripletAttention\*\*、\*\*BAMBlock\*\*、\*\*EfficientAttention\*\* 等其他类实现了不同的注意力机制，利用卷积、池化和激活函数等操作来增强特征表示。  
  
10. \*\*LSKA (Large-Separable-Kernel-Attention)\*\* 和 \*\*SegNext\_Attention\*\*：这些类实现了更复杂的注意力机制，适用于更高效的特征提取和图像分割任务。  
  
11. \*\*MLCA (MultiPath Coordinate Attention)\*\* 和 \*\*AFGCAttention\*\*：这些模块结合了多路径和自适应机制，以提高模型的表现。  
  
文件中还定义了一些辅助函数，如 `img2windows` 和 `windows2img`，用于在图像和窗口之间进行转换，以便于处理。  
  
总的来说，这个文件实现了多种先进的注意力机制，旨在提高深度学习模型在视觉任务中的性能，尤其是在图像分类、分割和特征提取等方面。每个模块都可以根据具体的需求进行组合和使用，以实现更复杂的网络结构。

```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  
from timm.models.layers import SqueezeExcite  
  
def \_make\_divisible(v, divisor, min\_value=None):  
 """  
 确保所有层的通道数是可被8整除的。  
 :param v: 输入的通道数  
 :param divisor: 需要整除的数  
 :param min\_value: 最小值  
 :return: 处理后的通道数  
 """  
 if min\_value is None:  
 min\_value = divisor  
 new\_v = max(min\_value, int(v + divisor / 2) // divisor \* divisor)  
 if new\_v < 0.9 \* v: # 确保向下取整不会超过10%  
 new\_v += divisor  
 return new\_v  
  
class Conv2d\_BN(nn.Sequential):  
 """  
 带有BatchNorm的卷积层  
 """  
 def \_\_init\_\_(self, in\_channels, out\_channels, kernel\_size=1, stride=1, padding=0, dilation=1, groups=1):  
 super().\_\_init\_\_()  
 # 添加卷积层  
 self.add\_module('conv', nn.Conv2d(in\_channels, out\_channels, kernel\_size, stride, padding, dilation, groups, bias=False))  
 # 添加BatchNorm层  
 self.add\_module('bn', nn.BatchNorm2d(out\_channels))  
  
 @torch.no\_grad()  
 def fuse\_self(self):  
 """  
 融合卷积层和BatchNorm层为一个卷积层  
 """  
 conv, bn = self.\_modules.values()  
 # 计算融合后的权重和偏置  
 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  
 # 创建新的卷积层  
 fused\_conv = 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)  
 fused\_conv.weight.data.copy\_(w)  
 fused\_conv.bias.data.copy\_(b)  
 return fused\_conv  
  
class RepViTBlock(nn.Module):  
 """  
 RepViT的基本块  
 """  
 def \_\_init\_\_(self, inp, hidden\_dim, oup, kernel\_size, stride, use\_se, use\_hs):  
 super(RepViTBlock, self).\_\_init\_\_()  
 self.identity = stride == 1 and inp == oup # 判断是否为身份映射  
 assert(hidden\_dim == 2 \* inp) # 隐藏层维度是输入的两倍  
  
 if stride == 2:  
 # 当步幅为2时，使用卷积和SqueezeExcite  
 self.token\_mixer = nn.Sequential(  
 Conv2d\_BN(inp, inp, kernel\_size, stride, (kernel\_size - 1) // 2, groups=inp),  
 SqueezeExcite(inp, 0.25) if use\_se else nn.Identity(),  
 Conv2d\_BN(inp, oup, ks=1, stride=1, pad=0)  
 )  
 else:  
 assert(self.identity)  
 # 当步幅为1时，使用RepVGGDW  
 self.token\_mixer = nn.Sequential(  
 RepVGGDW(inp),  
 SqueezeExcite(inp, 0.25) if use\_se else nn.Identity(),  
 )  
  
 # 通道混合器  
 self.channel\_mixer = Residual(nn.Sequential(  
 Conv2d\_BN(oup, hidden\_dim, 1, 1, 0),  
 nn.GELU() if use\_hs else nn.GELU(),  
 Conv2d\_BN(hidden\_dim, oup, 1, 1, 0)  
 ))  
  
 def forward(self, x):  
 return self.channel\_mixer(self.token\_mixer(x))  
  
class RepViT(nn.Module):  
 """  
 RepViT模型  
 """  
 def \_\_init\_\_(self, cfgs):  
 super(RepViT, self).\_\_init\_\_()  
 self.cfgs = cfgs # 配置参数  
 layers = []  
  
 # 构建初始层  
 input\_channel = self.cfgs[0][2]  
 patch\_embed = nn.Sequential(Conv2d\_BN(3, input\_channel // 2, 3, 2, 1), nn.GELU(),  
 Conv2d\_BN(input\_channel // 2, input\_channel, 3, 2, 1))  
 layers.append(patch\_embed)  
  
 # 构建RepViT块  
 for k, t, c, use\_se, use\_hs, s in self.cfgs:  
 output\_channel = \_make\_divisible(c, 8)  
 exp\_size = \_make\_divisible(input\_channel \* t, 8)  
 layers.append(RepViTBlock(input\_channel, exp\_size, output\_channel, k, s, use\_se, use\_hs))  
 input\_channel = output\_channel  
  
 self.features = nn.ModuleList(layers)  
  
 def forward(self, x):  
 for f in self.features:  
 x = f(x)  
 return x  
  
def repvit\_m2\_3(weights=''):  
 """  
 构建RepViT模型  
 """  
 cfgs = [  
 # k, t, c, SE, HS, s   
 [3, 2, 80, 1, 0, 1],  
 [3, 2, 80, 0, 0, 1],  
 # 省略其他配置...  
 ]  
 model = RepViT(cfgs)  
 if weights:  
 model.load\_state\_dict(torch.load(weights)['model'])  
 return model  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 model = repvit\_m2\_3('repvit\_m2\_3\_distill\_450e.pth') # 加载模型  
 inputs = torch.randn((1, 3, 640, 640)) # 随机输入  
 res = model(inputs) # 前向传播  
 for i in res:  
 print(i.size()) # 输出每层的尺寸  
```  
  
### 代码说明：  
1. \*\*\_make\_divisible\*\*: 确保通道数是8的倍数。  
2. \*\*Conv2d\_BN\*\*: 定义一个带BatchNorm的卷积层，并提供融合功能。  
3. \*\*RepViTBlock\*\*: 代表ViT的基本块，包含token混合和通道混合的逻辑。  
4. \*\*RepViT\*\*: 构建整个RepViT模型，包含多个RepViTBlock。  
5. \*\*repvit\_m2\_3\*\*: 构建特定配置的RepViT模型，并可加载预训练权重。  
  
这段代码实现了一个基于RepViT架构的深度学习模型，适用于图像处理任务。```

这个程序文件 `repvit.py` 实现了一个基于 RepVGG 结构的视觉模型，主要用于图像分类等任务。代码中包含了多个类和函数，下面是对其主要部分的讲解。  
  
首先，文件导入了必要的库，包括 PyTorch 的神经网络模块 `torch.nn`、NumPy、以及 `timm` 库中的 SqueezeExcite 层。然后定义了一个 `\_\_all\_\_` 列表，列出了可以从该模块导入的模型名称。  
  
接下来，定义了一个 `replace\_batchnorm` 函数，该函数用于遍历网络的所有子模块，并将 BatchNorm2d 层替换为 Identity 层，以便在推理时提高效率。同时，如果子模块具有 `fuse\_self` 方法，则会调用该方法进行融合。  
  
`\_make\_divisible` 函数用于确保每一层的通道数是 8 的倍数，这对于某些模型架构（如 MobileNet）是必要的。它通过向上取整和下取整的方式来实现这一点。  
  
`Conv2d\_BN` 类是一个自定义的卷积层，包含卷积操作和 BatchNorm。它在初始化时会对 BatchNorm 的权重进行常数初始化，并提供了一个 `fuse\_self` 方法来融合卷积和 BatchNorm 层，以提高推理速度。  
  
`Residual` 类实现了残差连接，允许在训练时使用随机丢弃（dropout）来增强模型的鲁棒性。它同样提供了 `fuse\_self` 方法，用于融合卷积层和残差连接。  
  
`RepVGGDW` 类实现了一个特定的卷积块，结合了深度可分离卷积和残差连接。它的 `forward` 方法定义了前向传播的计算过程。  
  
`RepViTBlock` 类是 RepViT 的基本构建块，包含了 token mixer 和 channel mixer 的结构。根据步幅的不同，它会选择不同的结构来处理输入。  
  
`RepViT` 类是整个模型的主体，负责构建模型的各个层。它根据配置参数（如卷积核大小、扩展因子、输出通道数等）构建多个 RepViTBlock，并在 `forward` 方法中定义了前向传播过程。  
  
此外，`switch\_to\_deploy` 方法用于将模型切换到推理模式，调用 `replace\_batchnorm` 函数以提高推理效率。  
  
`update\_weight` 函数用于更新模型的权重，确保新权重与模型结构相匹配。  
  
最后，定义了一系列函数（如 `repvit\_m0\_9`、`repvit\_m1\_0` 等）用于构建不同配置的 RepViT 模型，并可以选择加载预训练权重。  
  
在文件的最后部分，使用 `if \_\_name\_\_ == '\_\_main\_\_':` 块来测试模型的构建和前向传播，生成随机输入并输出各层的尺寸。  
  
总体而言，这个文件实现了一个灵活且高效的视觉模型架构，适用于各种图像处理任务。

```以下是经过简化并添加详细中文注释的核心代码部分：  
  
```python  
import torch  
import torch.nn as nn  
from timm.models.layers import DropPath, trunc\_normal\_  
  
# 定义一个卷积层和批归一化层的组合  
class ConvBN(torch.nn.Sequential):  
 def \_\_init\_\_(self, in\_planes, out\_planes, kernel\_size=1, stride=1, padding=0, dilation=1, groups=1, with\_bn=True):  
 super().\_\_init\_\_()  
 # 添加卷积层  
 self.add\_module('conv', torch.nn.Conv2d(in\_planes, out\_planes, kernel\_size, stride, padding, dilation, groups))  
 # 如果需要，添加批归一化层  
 if with\_bn:  
 self.add\_module('bn', torch.nn.BatchNorm2d(out\_planes))  
 # 初始化批归一化层的权重和偏置  
 torch.nn.init.constant\_(self.bn.weight, 1)  
 torch.nn.init.constant\_(self.bn.bias, 0)  
  
# 定义网络中的基本模块  
class Block(nn.Module):  
 def \_\_init\_\_(self, dim, mlp\_ratio=3, drop\_path=0.):  
 super().\_\_init\_\_()  
 # 深度可分离卷积  
 self.dwconv = ConvBN(dim, dim, 7, 1, (7 - 1) // 2, groups=dim, with\_bn=True)  
 # 线性变换  
 self.f1 = ConvBN(dim, mlp\_ratio \* dim, 1, with\_bn=False)  
 self.f2 = ConvBN(dim, mlp\_ratio \* dim, 1, with\_bn=False)  
 self.g = ConvBN(mlp\_ratio \* dim, dim, 1, with\_bn=True)  
 self.dwconv2 = ConvBN(dim, dim, 7, 1, (7 - 1) // 2, groups=dim, with\_bn=False)  
 self.act = nn.ReLU6() # 激活函数  
 self.drop\_path = DropPath(drop\_path) if drop\_path > 0. else nn.Identity() # 随机深度  
  
 def forward(self, x):  
 input = x # 保存输入  
 x = self.dwconv(x) # 深度可分离卷积  
 x1, x2 = self.f1(x), self.f2(x) # 线性变换  
 x = self.act(x1) \* x2 # 元素级乘法  
 x = self.dwconv2(self.g(x)) # 再次卷积  
 x = input + self.drop\_path(x) # 残差连接  
 return x  
  
# 定义StarNet网络结构  
class StarNet(nn.Module):  
 def \_\_init\_\_(self, base\_dim=32, depths=[3, 3, 12, 5], mlp\_ratio=4, drop\_path\_rate=0.0, num\_classes=1000, \*\*kwargs):  
 super().\_\_init\_\_()  
 self.num\_classes = num\_classes  
 self.in\_channel = 32  
 # stem层  
 self.stem = nn.Sequential(ConvBN(3, self.in\_channel, kernel\_size=3, stride=2, padding=1), nn.ReLU6())  
 dpr = [x.item() for x in torch.linspace(0, drop\_path\_rate, sum(depths))] # 随机深度  
 # 构建各个阶段  
 self.stages = nn.ModuleList()  
 cur = 0  
 for i\_layer in range(len(depths)):  
 embed\_dim = base\_dim \* 2 \*\* i\_layer # 当前层的嵌入维度  
 down\_sampler = ConvBN(self.in\_channel, embed\_dim, 3, 2, 1) # 下采样  
 self.in\_channel = embed\_dim  
 blocks = [Block(self.in\_channel, mlp\_ratio, dpr[cur + i]) for i in range(depths[i\_layer])] # 添加Block  
 cur += depths[i\_layer]  
 self.stages.append(nn.Sequential(down\_sampler, \*blocks)) # 将下采样和Block组合  
  
 def forward(self, x):  
 features = []  
 x = self.stem(x) # 通过stem层  
 features.append(x)  
 for stage in self.stages:  
 x = stage(x) # 通过每个阶段  
 features.append(x)  
 return features # 返回特征  
  
# 定义不同规模的StarNet模型  
def starnet\_s1(pretrained=False, \*\*kwargs):  
 model = StarNet(24, [2, 2, 8, 3], \*\*kwargs)  
 return model  
  
def starnet\_s2(pretrained=False, \*\*kwargs):  
 model = StarNet(32, [1, 2, 6, 2], \*\*kwargs)  
 return model  
  
def starnet\_s3(pretrained=False, \*\*kwargs):  
 model = StarNet(32, [2, 2, 8, 4], \*\*kwargs)  
 return model  
  
def starnet\_s4(pretrained=False, \*\*kwargs):  
 model = StarNet(32, [3, 3, 12, 5], \*\*kwargs)  
 return model  
```  
  
### 代码说明：  
1. \*\*ConvBN类\*\*：封装了卷积层和批归一化层的组合，方便创建卷积模块。  
2. \*\*Block类\*\*：实现了StarNet的基本构建块，包含深度可分离卷积、线性变换和元素级乘法的操作。  
3. \*\*StarNet类\*\*：构建了整个网络结构，包括stem层和多个阶段，每个阶段由下采样和多个Block组成。  
4. \*\*starnet\_sX函数\*\*：提供了不同规模的StarNet模型的构造函数，方便用户根据需求创建模型。```

这个程序文件实现了一个名为StarNet的深度学习网络，主要用于图像处理任务。StarNet的设计旨在展示元素级乘法的关键贡献，因此在网络设计中尽量简化，比如没有使用层级缩放（layer-scale）和训练过程中的指数移动平均（EMA），这些简化可能会进一步提升性能。  
  
文件首先导入了必要的库，包括PyTorch和一些用于构建网络层的模块。接着定义了一个包含不同StarNet模型的列表和对应的模型权重下载链接。  
  
接下来，定义了一个名为`ConvBN`的类，它是一个顺序容器，包含卷积层和可选的批归一化层。这个类的构造函数接受多个参数来设置卷积层的属性，并初始化批归一化层的权重和偏置。  
  
然后定义了`Block`类，表示StarNet中的基本构建块。这个类中包含了深度卷积、两个全连接层（通过`ConvBN`实现）和一个ReLU6激活函数。它的前向传播方法实现了输入的处理流程，包括对输入进行卷积、激活和元素级乘法操作，最后将结果与输入相加并通过DropPath进行随机深度的处理。  
  
`StarNet`类是整个网络的核心，包含了多个阶段（stages），每个阶段由一个下采样层和多个`Block`组成。构造函数中设置了网络的基础维度、每个阶段的深度、MLP比率和随机丢弃率等参数。网络的前向传播方法会依次通过各个阶段，并收集特征图。  
  
文件中还定义了多个函数（如`starnet\_s1`、`starnet\_s2`等），用于创建不同配置的StarNet模型。这些函数允许用户选择是否加载预训练权重，并根据需要传递其他参数。  
  
最后，文件中还定义了一些非常小的网络版本（如`starnet\_s050`、`starnet\_s100`和`starnet\_s150`），这些网络具有更少的层数和参数，适合于资源受限的环境或快速实验。  
  
总体来说，这个文件提供了一个灵活且易于扩展的深度学习模型实现，适合用于图像分类等任务，同时展示了元素级乘法在网络设计中的重要性。

### 整体功能和构架概括  
  
该工程包含多个深度学习模型和模块，主要用于图像处理和计算机视觉任务。每个文件实现了不同的网络结构或注意力机制，提供了灵活的构建块和功能，以便于用户根据需求进行组合和扩展。以下是每个文件的主要功能概述：  
  
1. \*\*wtconv2d.py\*\*：实现了基于小波变换的二维卷积层，结合了小波变换的优点来增强卷积操作，适用于图像处理和特征提取。  
  
2. \*\*attention.py\*\*：定义了多种注意力机制模块，包括通道注意力、空间注意力等，旨在提高深度学习模型在视觉任务中的性能。  
  
3. \*\*repvit.py\*\*：实现了RepVGG结构的视觉模型，结合了深度可分离卷积和残差连接，适用于图像分类等任务，并提供了多种配置选项。  
  
4. \*\*starnet.py\*\*：实现了StarNet网络，展示了元素级乘法的关键贡献，设计上简化了许多复杂的操作，适合用于图像处理任务。  
  
### 文件功能整理表  
  
| 文件名 | 功能描述 |  
|------------------|--------------------------------------------------------------------------------------------|  
| wtconv2d.py | 实现基于小波变换的二维卷积层，增强卷积操作的能力，适用于图像处理和特征提取。 |  
| attention.py | 定义多种注意力机制模块，提高深度学习模型在视觉任务中的性能，支持多种注意力策略。 |  
| repvit.py | 实现RepVGG结构的视觉模型，结合深度可分离卷积和残差连接，适用于图像分类，提供多种配置选项。 |  
| starnet.py | 实现StarNet网络，展示元素级乘法的关键贡献，设计上简化了复杂操作，适合图像处理任务。 |  
  
这个表格总结了每个文件的主要功能，便于快速了解工程的整体结构和各个模块的作用。