# 改进yolo11-DGCST等200+全套创新点大全：海上场景水上交通物体检测图像分割系统源码＆数据集全套

## 3.背景

研究背景与意义  
  
随着全球水上交通的快速发展，海洋运输和水上活动的安全性与效率愈发受到重视。水上交通中涉及的多种物体，如商船、渔船、标志浮标以及其他障碍物，常常对航行安全构成威胁。因此，开发高效的物体检测与图像分割系统，能够实时识别和分类这些物体，成为了提升水上交通安全的重要手段。基于改进YOLOv11的海上场景水上交通物体检测图像分割系统，旨在利用深度学习技术，针对复杂的海洋环境，提供一种高效、准确的解决方案。  
  
本研究所使用的数据集包含3600幅图像，涵盖了13个类别，包括商船、渔船、乘客船、标志浮标等。这些类别的多样性使得模型在训练过程中能够学习到丰富的特征，从而在实际应用中提高识别精度。尤其是在海上场景中，光照变化、波浪干扰以及背景复杂性等因素，都会对物体检测造成挑战。因此，改进YOLOv11模型的设计与训练，将为提升系统的鲁棒性和准确性提供可能。  
  
此外，图像分割技术的引入，使得系统不仅能够识别物体的类别，还能精确地定位物体的边界。这对于后续的水上交通管理、事故预警以及自动驾驶等应用具有重要意义。通过对水上交通环境的深入分析与研究，本项目将为智能水上交通系统的建设提供理论支持和技术保障，推动海洋安全管理的智能化进程。总之，本研究不仅具有重要的学术价值，还有助于提升实际应用中的水上交通安全与效率，具有广泛的社会意义。

## 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“Prototype-Dataset-May”，旨在为改进YOLOv11的海上场景水上交通物体检测图像分割系统提供丰富的训练数据。该数据集涵盖了13个类别，具体包括：Buoy-Marker（浮标标记）、Commercial Vessel（商船）、Fishing Boat（渔船）、Land（陆地）、Other Obstacle（其他障碍物）、POV Vessel（视角船只）、Passenger Vessel（客船）、Person（人）、Powerboat（动力艇）、Sailboat（帆船）、Sky（天空）、Small Craft（小型船只）以及Water（水面）。这些类别的多样性确保了模型在不同海上场景中的适应性和准确性。  
  
数据集中的图像均为高分辨率，涵盖了多种天气和光照条件下的海上环境，确保了训练数据的多样性和真实性。每个类别的样本数量经过精心设计，以确保模型在训练过程中能够充分学习到每种物体的特征。此外，数据集还包含了标注信息，提供了每个物体在图像中的精确位置和形状，这对于图像分割任务至关重要。  
  
通过使用“Prototype-Dataset-May”数据集，研究团队能够有效地训练和优化YOLOv11模型，使其在海上交通物体检测中表现出色。该数据集不仅为模型提供了必要的训练基础，还为后续的测试和验证阶段奠定了坚实的基础。随着模型的不断改进，期望能够在实际应用中实现更高的检测精度和更快的响应速度，从而提升海上交通安全和管理效率。

## 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

```以下是经过简化和注释的核心代码，保留了小波变换（Wavelet Transform）和卷积操作的主要部分：  
  
```python  
import torch  
import torch.nn as nn  
import torch.nn.functional as F  
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)  
  
 # 重复滤波器以匹配输入通道数  
 dec\_filters = dec\_filters[:, None].repeat(in\_size, 1, 1, 1)  
  
 # 反转重构滤波器  
 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)  
  
 # 重复滤波器以匹配输出通道数  
 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 # 获取输入的批次大小、通道数、高度和宽度  
 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，stride=2表示下采样  
 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  
  
# 反小波变换  
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，stride=2表示上采样  
 x = F.conv\_transpose2d(x, filters.to(x.dtype).to(x.device), stride=2, groups=c, padding=pad)  
 return x  
  
# 定义小波变换类  
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 # 输入通道数和输出通道数必须相等  
  
 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):  
 # 执行小波变换和卷积操作  
 x\_tag = wavelet\_transform(x, self.wt\_filter) # 小波变换  
 x = self.base\_conv(x) # 基础卷积  
 x = x + x\_tag # 合并卷积结果和小波变换结果  
 return x  
```  
  
### 代码注释说明：  
1. \*\*创建小波滤波器\*\*：使用PyWavelets库生成小波的分解和重构滤波器，并将其转换为PyTorch张量。  
2. \*\*小波变换和反小波变换\*\*：定义了两个函数，分别用于执行小波变换和反小波变换，利用卷积和转置卷积实现。  
3. \*\*WaveletTransform类\*\*：实现了小波变换的前向和反向传播，便于在神经网络中使用。  
4. \*\*WTConv2d类\*\*：定义了一个小波卷积层，包含基础卷积和小波变换的结合。```

这个文件 `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。文件中使用了 PyTorch 框架，并结合了小波变换的数学原理来构建一个新的卷积层 `WTConv2d`。  
  
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、功能模块，以及用于小波变换的 `pywt` 库。接着，定义了一个函数 `create\_wavelet\_filter`，该函数根据给定的小波类型和输入输出通道的数量，创建小波变换和逆小波变换所需的滤波器。这个函数生成了两个滤波器：一个用于小波变换（降采样），另一个用于逆小波变换（升采样）。  
  
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。它们通过使用 PyTorch 的卷积操作实现，支持多通道输入，并且在变换过程中考虑了适当的填充。  
  
然后，定义了两个类 `WaveletTransform` 和 `InverseWaveletTransform`，它们继承自 `torch.autograd.Function`。这两个类分别实现了小波变换和逆小波变换的前向和反向传播方法，以便在训练过程中能够计算梯度。  
  
接下来，定义了两个初始化函数 `wavelet\_transform\_init` 和 `inverse\_wavelet\_transform\_init`，用于生成小波变换和逆小波变换的应用函数。  
  
`WTConv2d` 类是这个文件的核心部分，继承自 `nn.Module`。在初始化方法中，首先验证输入和输出通道数相同。然后，创建小波变换和逆小波变换的滤波器，并将其设置为不可训练的参数。接着，定义了基本的卷积层和小波卷积层，后者用于处理小波变换后的特征。  
  
在 `forward` 方法中，输入数据首先经过小波变换，得到低频和高频特征。然后，逐层处理这些特征，使用小波卷积层进行特征提取。最后，进行逆小波变换，将处理后的特征合并回去，并与基本卷积层的输出相加，形成最终的输出。  
  
最后，定义了一个辅助类 `\_ScaleModule`，用于对输入进行缩放操作，便于在网络中调整特征的尺度。  
  
整体而言，这个文件实现了一个结合小波变换的卷积神经网络层，能够有效地提取图像特征，同时保留了小波变换的多分辨率特性，适用于图像处理和计算机视觉任务。

```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  
import torch.nn.functional as F  
  
class SS2D(nn.Module):  
 def \_\_init\_\_(self, d\_model, d\_state=16, d\_conv=3, expand=2, dropout=0.):  
 super().\_\_init\_\_()  
 self.d\_model = d\_model # 输入特征维度  
 self.d\_state = d\_state # 状态维度  
 self.d\_conv = d\_conv # 卷积核大小  
 self.expand = expand # 扩展因子  
 self.d\_inner = int(self.expand \* self.d\_model) # 内部特征维度  
  
 # 输入投影层，将输入特征维度映射到内部特征维度  
 self.in\_proj = nn.Linear(self.d\_model, self.d\_inner \* 2)  
 # 卷积层，进行特征提取  
 self.conv2d = nn.Conv2d(in\_channels=self.d\_inner, out\_channels=self.d\_inner, groups=self.d\_inner, kernel\_size=d\_conv, padding=(d\_conv - 1) // 2)  
 self.act = nn.SiLU() # 激活函数  
  
 # 输出层，最终将内部特征映射回输入特征维度  
 self.out\_proj = nn.Linear(self.d\_inner, self.d\_model)  
 self.dropout = nn.Dropout(dropout) if dropout > 0. else None # 可选的dropout层  
  
 def forward(self, x: torch.Tensor):  
 # 输入 x 的形状为 (B, H, W, C)，需要调整为 (B, C, H, W)  
 x = x.permute(0, 3, 1, 2).contiguous() # 变换维度顺序  
 x = self.in\_proj(x) # 输入投影  
 x, z = x.chunk(2, dim=-1) # 将投影结果分为两个部分  
  
 x = self.act(self.conv2d(x)) # 经过卷积和激活函数  
 y = self.forward\_core(x) # 通过核心前向传播  
 y = y \* F.silu(z) # 结合 z 的信息  
 out = self.out\_proj(y) # 输出层映射回原始维度  
 if self.dropout is not None:  
 out = self.dropout(out) # 应用dropout  
 return out  
  
 def forward\_core(self, x: torch.Tensor):  
 # 核心前向传播逻辑  
 B, C, H, W = x.shape  
 L = H \* W # 计算总的空间维度  
 K = 4 # 定义 K 的值  
  
 # 处理输入，准备进行选择性扫描  
 x\_hwwh = torch.stack([x.view(B, -1, L), torch.transpose(x, dim0=2, dim1=3).contiguous().view(B, -1, L)], dim=1).view(B, 2, -1, L)  
 xs = torch.cat([x\_hwwh, torch.flip(x\_hwwh, dims=[-1])], dim=1) # 生成对称输入  
  
 # 进行选择性扫描的计算  
 # 省略具体实现细节，假设有一个选择性扫描函数  
 out\_y = self.selective\_scan(xs) # 调用选择性扫描函数  
  
 # 处理输出  
 y = out\_y.view(B, H, W, -1) # 变换输出维度  
 return y  
  
class VSSBlock(nn.Module):  
 def \_\_init\_\_(self, hidden\_dim: int = 0, drop\_path: float = 0.2):  
 super().\_\_init\_\_()  
 self.ln\_1 = nn.LayerNorm(hidden\_dim) # 归一化层  
 self.self\_attention = SS2D(d\_model=hidden\_dim) # 自注意力层  
 self.drop\_path = nn.Dropout(drop\_path) # 随机丢弃路径  
  
 def forward(self, input: torch.Tensor):  
 input = input.permute((0, 2, 3, 1)) # 调整输入维度  
 x = input + self.drop\_path(self.self\_attention(self.ln\_1(input))) # 残差连接  
 return x.permute((0, 3, 1, 2)) # 恢复维度顺序  
  
# 示例代码，用于测试模型  
if \_\_name\_\_ == '\_\_main\_\_':  
 inputs = torch.randn((1, 64, 32, 32)).cuda() # 生成随机输入  
 model = VSSBlock(64).cuda() # 实例化模型  
 pred = model(inputs) # 前向传播  
 print(pred.size()) # 输出预测结果的尺寸  
```  
  
### 代码说明：  
1. \*\*SS2D类\*\*：实现了一个自注意力机制，包含输入投影、卷积层和输出投影。核心前向传播逻辑在`forward\_core`方法中实现。  
2. \*\*VSSBlock类\*\*：包含一个归一化层和一个自注意力层，使用残差连接来增强模型的学习能力。  
3. \*\*前向传播\*\*：通过`forward`方法实现，输入数据的维度需要调整以适应模型的要求。  
4. \*\*示例代码\*\*：用于测试模型的输入和输出尺寸。```

这个程序文件 `mamba\_vss.py` 实现了两个神经网络模块：`VSSBlock` 和 `Mamba2Block`，它们都基于自定义的 `SS2D` 类。该文件使用了 PyTorch 库，主要用于构建深度学习模型，特别是在处理图像数据时。  
  
首先，文件导入了一些必要的库，包括 PyTorch 的核心库、神经网络模块、功能性操作、以及用于处理张量的 `einops` 库。还尝试导入了一些自定义的模块和函数，如果这些模块不存在则会被忽略。  
  
`SS2D` 类是实现的核心部分，它继承自 `nn.Module`。在初始化方法中，定义了一系列参数，包括模型的维度、状态维度、卷积核大小、扩展因子等。接着，创建了一些线性层和卷积层，使用了 SiLU 激活函数，并定义了一些参数的初始化方法。特别地，`dt\_init`、`A\_log\_init` 和 `D\_init` 方法用于初始化不同的权重和偏置，以确保模型在训练开始时的稳定性。  
  
`forward\_corev0` 方法是 `SS2D` 类的核心前向传播逻辑，处理输入张量并进行一系列的线性变换、卷积操作和选择性扫描。这个方法的输出经过一系列的变换后，返回最终的结果。  
  
`forward` 方法则是 `SS2D` 类的前向传播接口，接收输入数据，经过线性投影、卷积、核心前向传播以及最终的输出投影，返回模型的输出。  
  
`VSSBlock` 类同样继承自 `nn.Module`，并在初始化时创建了一个层归一化层和一个 `SS2D` 自注意力层。它的前向传播方法对输入进行变换，并将自注意力的输出与原始输入相加，应用了 DropPath 技术以增强模型的鲁棒性。  
  
`Mamba2Block` 类继承自 `VSSBlock`，并在初始化时替换了自注意力层为 `Mamba2Simple`，这是另一个自定义的模块。它的前向传播方法与 `VSSBlock` 类似，但在处理输入时进行了额外的维度变换。  
  
在文件的最后部分，包含了一个测试代码块，用于创建随机输入并通过 `VSSBlock` 和 `Mamba2Block` 进行前向传播，打印输出的尺寸。这部分代码可以用于验证模型的基本功能和形状是否正确。  
  
总体来说，这个文件实现了一个复杂的自注意力机制，适用于图像处理任务，具有灵活的参数设置和初始化方法，旨在提高模型的性能和稳定性。

```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.nn as nn  
import torch  
  
# 定义一个用于替换BatchNorm层的函数  
def replace\_batchnorm(net):  
 for child\_name, child in net.named\_children():  
 # 如果子模块有fuse\_self方法，进行融合  
 if hasattr(child, 'fuse\_self'):  
 fused = child.fuse\_self()  
 setattr(net, child\_name, fused)  
 replace\_batchnorm(fused)  
 # 如果子模块是BatchNorm2d，替换为Identity层  
 elif isinstance(child, torch.nn.BatchNorm2d):  
 setattr(net, child\_name, torch.nn.Identity())  
 else:  
 replace\_batchnorm(child)  
  
# 定义一个函数，用于确保通道数是可被8整除的  
def \_make\_divisible(v, divisor, min\_value=None):  
 if min\_value is None:  
 min\_value = divisor  
 new\_v = max(min\_value, int(v + divisor / 2) // divisor \* divisor)  
 # 确保向下取整不会减少超过10%  
 if new\_v < 0.9 \* v:  
 new\_v += divisor  
 return new\_v  
  
# 定义一个包含卷积和BatchNorm的模块  
class Conv2d\_BN(torch.nn.Sequential):  
 def \_\_init\_\_(self, a, b, ks=1, stride=1, pad=0, dilation=1, groups=1, bn\_weight\_init=1):  
 super().\_\_init\_\_()  
 # 添加卷积层  
 self.add\_module('c', torch.nn.Conv2d(a, b, ks, stride, pad, dilation, groups, bias=False))  
 # 添加BatchNorm层  
 self.add\_module('bn', torch.nn.BatchNorm2d(b))  
 # 初始化BatchNorm的权重  
 torch.nn.init.constant\_(self.bn.weight, bn\_weight\_init)  
 torch.nn.init.constant\_(self.bn.bias, 0)  
  
 @torch.no\_grad()  
 def fuse\_self(self):  
 # 融合卷积层和BatchNorm层  
 c, bn = self.\_modules.values()  
 w = bn.weight / (bn.running\_var + bn.eps)\*\*0.5  
 w = c.weight \* w[:, None, None, None]  
 b = bn.bias - bn.running\_mean \* bn.weight / (bn.running\_var + bn.eps)\*\*0.5  
 m = torch.nn.Conv2d(w.size(1) \* self.c.groups, w.size(0), w.shape[2:], stride=self.c.stride, padding=self.c.padding, dilation=self.c.dilation, groups=self.c.groups)  
 m.weight.data.copy\_(w)  
 m.bias.data.copy\_(b)  
 return m  
  
# 定义残差模块  
class Residual(torch.nn.Module):  
 def \_\_init\_\_(self, m, drop=0.):  
 super().\_\_init\_\_()  
 self.m = m # 模块  
 self.drop = drop # 丢弃率  
  
 def forward(self, x):  
 # 在训练时根据丢弃率决定是否添加残差  
 if self.training and self.drop > 0:  
 return x + self.m(x) \* torch.rand(x.size(0), 1, 1, 1, device=x.device).ge\_(self.drop).div(1 - self.drop).detach()  
 else:  
 return x + self.m(x)  
  
 @torch.no\_grad()  
 def fuse\_self(self):  
 # 融合残差模块  
 if isinstance(self.m, Conv2d\_BN):  
 m = self.m.fuse\_self()  
 identity = torch.ones(m.weight.shape[0], m.weight.shape[1], 1, 1)  
 identity = torch.nn.functional.pad(identity, [1, 1, 1, 1])  
 m.weight += identity.to(m.weight.device)  
 return m  
 else:  
 return self  
  
# 定义RepViTBlock模块  
class RepViTBlock(nn.Module):  
 def \_\_init\_\_(self, inp, hidden\_dim, oup, kernel\_size, stride, use\_se, use\_hs):  
 super(RepViTBlock, self).\_\_init\_\_()  
 assert stride in [1, 2]  
 self.identity = stride == 1 and inp == oup # 判断是否为身份映射  
 assert(hidden\_dim == 2 \* inp) # 隐藏层维度为输入维度的两倍  
  
 if stride == 2:  
 # 如果步幅为2，构建token混合和通道混合  
 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)  
 )  
 self.channel\_mixer = Residual(nn.Sequential(  
 Conv2d\_BN(oup, 2 \* oup, 1, 1, 0),  
 nn.GELU() if use\_hs else nn.GELU(),  
 Conv2d\_BN(2 \* oup, oup, 1, 1, 0, bn\_weight\_init=0),  
 ))  
 else:  
 assert(self.identity)  
 self.token\_mixer = nn.Sequential(  
 RepVGGDW(inp),  
 SqueezeExcite(inp, 0.25) if use\_se else nn.Identity(),  
 )  
 self.channel\_mixer = Residual(nn.Sequential(  
 Conv2d\_BN(inp, hidden\_dim, 1, 1, 0),  
 nn.GELU() if use\_hs else nn.GELU(),  
 Conv2d\_BN(hidden\_dim, oup, 1, 1, 0, bn\_weight\_init=0),  
 ))  
  
 def forward(self, x):  
 return self.channel\_mixer(self.token\_mixer(x))  
  
# 定义RepViT模型  
class RepViT(nn.Module):  
 def \_\_init\_\_(self, cfgs):  
 super(RepViT, self).\_\_init\_\_()  
 self.cfgs = cfgs # 配置参数  
 input\_channel = self.cfgs[0][2] # 输入通道数  
 # 构建第一个层  
 patch\_embed = torch.nn.Sequential(Conv2d\_BN(3, input\_channel // 2, 3, 2, 1), torch.nn.GELU(),  
 Conv2d\_BN(input\_channel // 2, input\_channel, 3, 2, 1))  
 layers = [patch\_embed]  
 # 构建反向残差块  
 block = RepViTBlock  
 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(block(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  
  
# 示例：构建RepViT模型  
if \_\_name\_\_ == '\_\_main\_\_':  
 cfgs = [  
 # k, t, c, SE, HS, s   
 [3, 2, 64, 1, 0, 1],  
 [3, 2, 64, 0, 0, 1],  
 # 更多配置...  
 ]  
 model = RepViT(cfgs) # 创建模型实例  
 inputs = torch.randn((1, 3, 640, 640)) # 创建输入张量  
 res = model(inputs) # 前向传播  
 print(res.size()) # 输出结果的尺寸  
```  
  
### 代码核心部分解释：  
1. \*\*BatchNorm替换\*\*：`replace\_batchnorm`函数用于将模型中的BatchNorm层替换为Identity层，以便在推理时加速计算。  
2. \*\*可被8整除的通道数\*\*：`\_make\_divisible`函数确保网络中所有层的通道数都是8的倍数，以符合特定的硬件要求。  
3. \*\*卷积和BatchNorm组合\*\*：`Conv2d\_BN`类封装了卷积层和BatchNorm层，并提供了融合的方法。  
4. \*\*残差连接\*\*：`Residual`类实现了残差连接的逻辑，允许在训练时使用随机丢弃。  
5. \*\*RepViTBlock\*\*：这是RepViT模型的基本构建块，负责处理输入特征的混合和通道的转换。  
6. \*\*RepViT模型\*\*：`RepViT`类构建了整个模型，接受配置参数并按顺序添加各个块。  
  
通过这些核心部分的组合，构建了一个高效的视觉Transformer模型。```

这个程序文件 `repvit.py` 实现了一个基于 RepVGG 架构的视觉模型，结合了深度学习中的卷积神经网络（CNN）和注意力机制。代码中使用了 PyTorch 框架，并且包含了一些特定的模块和功能，以下是对代码的详细讲解。  
  
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` 模块。`\_\_all\_\_` 列表定义了可以被外部导入的模型名称。  
  
接下来，定义了一个 `replace\_batchnorm` 函数，用于在模型中替换掉所有的 Batch Normalization 层。这是为了在推理阶段提高模型的效率，通过将卷积层和 Batch Normalization 层融合成一个卷积层来减少计算量。  
  
`\_make\_divisible` 函数用于确保模型中所有层的通道数都是可被 8 整除的，这对于某些硬件加速器的优化是必要的。  
  
`Conv2d\_BN` 类继承自 `torch.nn.Sequential`，它将卷积层和 Batch Normalization 层组合在一起，并提供了一个 `fuse\_self` 方法，用于融合这两个层以提高推理速度。  
  
`Residual` 类实现了残差连接，允许输入通过一个模块后与原始输入相加，支持在训练时随机丢弃部分输出以增强模型的鲁棒性。  
  
`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，允许通过融合方法优化推理性能。  
  
`RepViTBlock` 类是模型的基本构建块，包含了通道混合和令牌混合的功能，使用了 `SqueezeExcite` 模块来增强特征表达能力。  
  
`RepViT` 类是整个模型的主体，负责构建网络结构。它根据配置列表构建多个 `RepViTBlock`，并在前向传播中提取特征。  
  
`switch\_to\_deploy` 方法用于将模型切换到推理模式，主要是调用 `replace\_batchnorm` 函数来优化模型。  
  
`update\_weight` 函数用于更新模型的权重，确保新权重与模型的结构匹配。  
  
最后，定义了多个函数（如 `repvit\_m0\_9`, `repvit\_m1\_0`, 等），这些函数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 `\_\_main\_\_` 部分，示例代码展示了如何实例化一个模型并进行前向传播，输入为一个随机生成的张量，输出则是模型在不同层的特征图的尺寸。  
  
整体来看，这个文件实现了一个灵活且高效的视觉模型，适用于各种计算机视觉任务，并且通过合理的设计和模块化的实现，提高了模型的可维护性和可扩展性。

``````python  
import torch  
import torch.nn as nn  
  
class AGLU(nn.Module):  
 """AGLU激活函数模块，来自https://github.com/kostas1515/AGLU。"""  
  
 def \_\_init\_\_(self, device=None, dtype=None) -> None:  
 """初始化AGLU激活函数模块。"""  
 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，并将其设置为可学习的参数  
 self.lambd = nn.Parameter(nn.init.uniform\_(torch.empty(1, device=device, dtype=dtype))) # lambda参数  
 # 初始化kappa参数，并将其设置为可学习的参数  
 self.kappa = nn.Parameter(nn.init.uniform\_(torch.empty(1, device=device, dtype=dtype))) # kappa参数  
  
 def forward(self, x: torch.Tensor) -> torch.Tensor:  
 """计算AGLU激活函数的前向传播。"""  
 # 将lambda参数限制在最小值0.0001，以避免数值不稳定  
 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)))  
```  
  
### 代码注释说明：  
1. \*\*AGLU类\*\*：定义了一个自定义的激活函数模块，继承自`nn.Module`。  
2. \*\*初始化方法`\_\_init\_\_`\*\*：  
 - `self.act`：使用`Softplus`作为基础激活函数。  
 - `self.lambd`和`self.kappa`：这两个参数是可学习的，初始化为均匀分布的随机值，并使用`nn.Parameter`使其成为模型的一部分。  
3. \*\*前向传播方法`forward`\*\*：  
 - `lam`：对`lambd`参数进行限制，确保其不小于0.0001，以避免在后续计算中出现数值不稳定。  
 - 返回值：计算并返回AGLU激活函数的输出，公式中结合了`Softplus`激活和参数`lambda`与`kappa`的影响。```

这个程序文件定义了一个名为 `activation.py` 的模块，主要用于实现一种统一的激活函数，称为 AGLU（Adaptive Gated Linear Unit）。该模块使用了 PyTorch 框架，包含了一个名为 `AGLU` 的类，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，首先调用了父类的构造函数 `super().\_\_init\_\_()`，以确保正确初始化基类。接着，定义了一个激活函数 `self.act`，使用了 `nn.Softplus`，其参数 `beta` 被设置为 -1.0。Softplus 是一种平滑的激活函数，类似于 ReLU，但在零附近更加平滑。  
  
此外，类中还定义了两个可学习的参数 `self.lambd` 和 `self.kappa`，它们分别用于控制激活函数的行为。这两个参数通过 `nn.Parameter` 声明，并使用 `torch.init.uniform\_` 方法进行初始化，确保它们在给定的设备和数据类型上随机生成。  
  
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。在方法内部，首先对 `self.lambd` 进行限制，确保其值不小于 0.0001，以避免在后续计算中出现除以零的情况。然后，使用公式计算激活值：首先计算 `self.kappa \* x`，再减去 `torch.log(lam)`，最后将结果传入 `self.act`，并根据 `lam` 的值进行指数运算，得到最终的激活输出。  
  
总体来说，这个模块实现了一种新的激活函数，结合了可学习的参数，旨在提高神经网络的表现。

### 程序整体功能和构架概括  
  
该程序包含多个模块，主要用于构建和实现深度学习模型，特别是在计算机视觉任务中的应用。整体架构由以下几个部分组成：  
  
1. \*\*卷积层和激活函数\*\*：`wtconv2d.py` 实现了基于小波变换的卷积层，能够提取图像特征并保持多分辨率特性。`activation.py` 提供了一种新的激活函数 AGLU，具有可学习的参数，旨在增强模型的表现。  
  
2. \*\*视觉模型构建\*\*：`mamba\_vss.py` 实现了基于自注意力机制的视觉模型模块，结合了残差连接和自注意力层，适用于图像特征提取和处理。  
  
3. \*\*RepVGG架构\*\*：`repvit.py` 实现了 RepVGG 架构，结合了深度可分离卷积和注意力机制，优化了模型的推理性能，并提供了多种配置的模型构建方法。  
  
整体上，这些模块通过灵活的设计和组合，构建了一个高效且可扩展的深度学习框架，适用于各种计算机视觉任务。  
  
### 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，结合小波变换和逆小波变换进行特征提取，适用于图像处理任务。 |  
| `mamba\_vss.py` | 实现视觉模型模块，结合自注意力机制和残差连接，增强特征提取能力，适用于计算机视觉任务。 |  
| `repvit.py` | 实现 RepVGG 架构，结合深度可分离卷积和注意力机制，优化推理性能，提供多种配置的模型构建方法。 |  
| `activation.py` | 实现 AGLU（Adaptive Gated Linear Unit）激活函数，具有可学习的参数，旨在提高神经网络的表现。 |  
  
这个表格清晰地总结了每个文件的主要功能，便于理解程序的整体结构和各个模块的作用。