# 改进yolo11-ASF-DySample等200+全套创新点大全：船舶类型检测系统源码＆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  
  
随着全球航运业的迅猛发展，船舶类型的自动检测与识别成为了海洋监测、海洋安全及智能交通系统中的重要研究课题。传统的船舶识别方法往往依赖于人工观察和经验判断，效率低下且易受人为因素影响，难以满足现代化海洋管理的需求。因此，开发一种高效、准确的船舶类型检测系统显得尤为重要。近年来，深度学习技术的飞速发展为目标检测领域带来了新的机遇，尤其是YOLO（You Only Look Once）系列模型在实时目标检测任务中表现出了优异的性能。  
  
本研究旨在基于改进的YOLOv11模型，构建一个高效的船舶类型检测系统。YOLOv11作为YOLO系列的最新版本，结合了多种先进的网络结构和算法优化，具备更高的检测精度和更快的推理速度。通过对YOLOv11的改进，我们将进一步提升其在复杂海洋环境中对船舶类型的识别能力。为此，我们将利用特定的数据集，该数据集包含了多种类型的船舶图像，涵盖了不同的角度、光照条件和背景环境。这些数据的多样性将为模型的训练提供丰富的样本，有助于提高模型的泛化能力。  
  
本项目的意义不仅在于技术层面的创新，更在于其对海洋安全和环境保护的实际应用价值。通过实现高效的船舶类型检测，可以为海洋监管部门提供实时监测数据，帮助其及时识别和处理海洋环境中的潜在威胁。同时，该系统还可应用于智能交通管理、船舶调度等领域，推动海洋经济的可持续发展。因此，基于改进YOLOv11的船舶类型检测系统的研究具有重要的理论价值和实际意义。

## 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“boat”，旨在为改进YOLOv11的船舶类型检测系统提供丰富的训练素材。该数据集包含六个主要类别，分别为：散货船、集装箱船、渔船、一般货船、矿石运输船和客船。这些类别涵盖了船舶运输行业的多样性，确保了模型在不同类型船舶识别上的全面性和准确性。  
  
数据集中的图像经过精心挑选和标注，确保每个类别的样本都具有代表性。散货船作为一种常见的货物运输方式，其特征包括宽大的船体和开放的货舱，适合运输大宗货物。集装箱船则以其标准化的集装箱为特点，具有明显的结构特征，适合于全球贸易。渔船的设计则更加多样化，通常较小且灵活，适合在近海或内陆水域作业。一般货船的设计相对通用，能够运输多种类型的货物，而矿石运输船则专门用于大宗矿石的运输，通常具有更大的载重能力。最后，客船则以其舒适性和安全性为主要设计目标，适合于人员运输。  
  
通过使用“boat”数据集，改进后的YOLOv11模型将能够更准确地识别和分类不同类型的船舶，从而提高船舶监测和管理的效率。这一数据集不仅为模型的训练提供了坚实的基础，也为未来在船舶类型检测领域的研究和应用奠定了重要的理论和实践基础。通过不断优化模型的识别能力，我们期望能够在实际应用中实现更高的准确率和更快的检测速度，为船舶行业的智能化发展贡献力量。

## 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提炼和详细注释的结果。保留了模型的主要结构和功能，删除了冗余的部分。  
  
```python  
import torch  
import torch.nn as nn  
from typing import List  
from torch import Tensor  
  
class PartialConv3(nn.Module):  
 """实现部分卷积的模块，用于处理输入的不同部分"""  
   
 def \_\_init\_\_(self, dim, n\_div, forward):  
 super().\_\_init\_\_()  
 self.dim\_conv3 = dim // n\_div # 计算卷积通道数  
 self.dim\_untouched = dim - self.dim\_conv3 # 计算未改变的通道数  
 self.partial\_conv3 = nn.Conv2d(self.dim\_conv3, self.dim\_conv3, 3, 1, 1, bias=False) # 定义卷积层  
  
 # 根据前向传播方式选择方法  
 if forward == 'slicing':  
 self.forward = self.forward\_slicing  
 elif forward == 'split\_cat':  
 self.forward = self.forward\_split\_cat  
 else:  
 raise NotImplementedError  
  
 def forward\_slicing(self, x: Tensor) -> Tensor:  
 """仅用于推理阶段的前向传播"""  
 x = x.clone() # 保持原始输入不变，以便后续的残差连接  
 x[:, :self.dim\_conv3, :, :] = self.partial\_conv3(x[:, :self.dim\_conv3, :, :]) # 处理部分通道  
 return x  
  
 def forward\_split\_cat(self, x: Tensor) -> Tensor:  
 """用于训练和推理阶段的前向传播"""  
 x1, x2 = torch.split(x, [self.dim\_conv3, self.dim\_untouched], dim=1) # 分割输入  
 x1 = self.partial\_conv3(x1) # 处理分割的部分  
 x = torch.cat((x1, x2), 1) # 合并处理后的部分和未处理的部分  
 return x  
  
  
class MLPBlock(nn.Module):  
 """多层感知机模块，包含卷积、归一化和激活函数"""  
   
 def \_\_init\_\_(self, dim, n\_div, mlp\_ratio, drop\_path, act\_layer, norm\_layer, pconv\_fw\_type):  
 super().\_\_init\_\_()  
 self.dim = dim  
 self.mlp\_ratio = mlp\_ratio  
 self.drop\_path = nn.Identity() if drop\_path <= 0 else nn.Dropout(drop\_path) # 随机丢弃层  
 self.n\_div = n\_div  
  
 mlp\_hidden\_dim = int(dim \* mlp\_ratio) # 计算隐藏层维度  
  
 # 定义多层感知机结构  
 self.mlp = nn.Sequential(  
 nn.Conv2d(dim, mlp\_hidden\_dim, 1, bias=False),  
 norm\_layer(mlp\_hidden\_dim),  
 act\_layer(),  
 nn.Conv2d(mlp\_hidden\_dim, dim, 1, bias=False)  
 )  
  
 self.spatial\_mixing = PartialConv3(dim, n\_div, pconv\_fw\_type) # 空间混合模块  
  
 def forward(self, x: Tensor) -> Tensor:  
 """前向传播"""  
 shortcut = x # 保存输入以便后续残差连接  
 x = self.spatial\_mixing(x) # 进行空间混合  
 x = shortcut + self.drop\_path(self.mlp(x)) # 残差连接  
 return x  
  
  
class PatchEmbed(nn.Module):  
 """将输入图像分割成补丁并进行嵌入"""  
   
 def \_\_init\_\_(self, patch\_size, patch\_stride, in\_chans, embed\_dim, norm\_layer):  
 super().\_\_init\_\_()  
 self.proj = nn.Conv2d(in\_chans, embed\_dim, kernel\_size=patch\_size, stride=patch\_stride, bias=False) # 卷积层用于补丁嵌入  
 self.norm = norm\_layer(embed\_dim) if norm\_layer is not None else nn.Identity() # 归一化层  
  
 def forward(self, x: Tensor) -> Tensor:  
 """前向传播"""  
 x = self.norm(self.proj(x)) # 进行卷积和归一化  
 return x  
  
  
class FasterNet(nn.Module):  
 """FasterNet模型定义"""  
   
 def \_\_init\_\_(self, in\_chans=3, num\_classes=1000, embed\_dim=96, depths=(1, 2, 8, 2), mlp\_ratio=2., n\_div=4, patch\_size=4, patch\_stride=4, drop\_path\_rate=0.1, norm\_layer=nn.BatchNorm2d, act\_layer=nn.ReLU):  
 super().\_\_init\_\_()  
  
 self.num\_stages = len(depths) # 模型阶段数  
 self.embed\_dim = embed\_dim # 嵌入维度  
  
 # 初始化补丁嵌入层  
 self.patch\_embed = PatchEmbed(patch\_size=patch\_size, patch\_stride=patch\_stride, in\_chans=in\_chans, embed\_dim=embed\_dim, norm\_layer=norm\_layer)  
  
 # 构建各个阶段的层  
 stages\_list = []  
 for i\_stage in range(self.num\_stages):  
 stage = nn.Sequential(\*[MLPBlock(dim=int(embed\_dim \* 2 \*\* i\_stage), n\_div=n\_div, mlp\_ratio=mlp\_ratio, drop\_path=0.1, act\_layer=act\_layer, norm\_layer=norm\_layer, pconv\_fw\_type='split\_cat') for \_ in range(depths[i\_stage])])  
 stages\_list.append(stage)  
  
 self.stages = nn.Sequential(\*stages\_list) # 将所有阶段组合成一个序列  
  
 def forward(self, x: Tensor) -> List[Tensor]:  
 """前向传播，输出每个阶段的特征"""  
 x = self.patch\_embed(x) # 进行补丁嵌入  
 outs = []  
 for stage in self.stages:  
 x = stage(x) # 通过每个阶段  
 outs.append(x) # 保存每个阶段的输出  
 return outs  
  
  
# 用于创建FasterNet模型的函数  
def fasternet\_t0(weights=None, cfg='path/to/config.yaml'):  
 with open(cfg) as f:  
 cfg = yaml.load(f, Loader=yaml.SafeLoader) # 读取配置文件  
 model = FasterNet(\*\*cfg) # 创建模型  
 if weights is not None:  
 pretrain\_weight = torch.load(weights, map\_location='cpu') # 加载预训练权重  
 model.load\_state\_dict(pretrain\_weight) # 更新模型权重  
 return model  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 model = fasternet\_t0(weights='path/to/weights.pth', cfg='path/to/config.yaml') # 实例化模型  
 inputs = torch.randn((1, 3, 640, 640)) # 创建输入张量  
 outputs = model(inputs) # 进行前向传播  
 for i in outputs:  
 print(i.size()) # 打印每个阶段的输出尺寸  
```  
  
### 代码说明  
1. \*\*PartialConv3\*\*: 实现部分卷积，支持两种前向传播方式，分别用于推理和训练。  
2. \*\*MLPBlock\*\*: 多层感知机模块，包含卷积、归一化和激活函数，支持残差连接。  
3. \*\*PatchEmbed\*\*: 将输入图像分割成补丁并进行嵌入。  
4. \*\*FasterNet\*\*: 主模型类，包含多个阶段的MLPBlock，负责输入的补丁嵌入和特征提取。  
5. \*\*fasternet\_t0\*\*: 用于创建FasterNet模型的函数，支持加载预训练权重。  
  
通过这种方式，保留了模型的核心结构和功能，同时去除了不必要的部分，使得代码更加简洁易懂。```

这个文件 `fasternet.py` 是一个实现了 FasterNet 模型的 PyTorch 代码。FasterNet 是一种用于图像处理的深度学习模型，主要用于目标检测和图像分割等任务。代码中包含了模型的定义、各个组件的实现以及模型的加载和初始化方法。  
  
首先，代码引入了一些必要的库，包括 PyTorch、YAML 解析库、以及一些用于构建神经网络的模块。接着，定义了一些类，这些类是构建 FasterNet 模型的基础。  
  
`Partial\_conv3` 类实现了一个部分卷积层，支持两种前向传播方式：切片（slicing）和分割拼接（split\_cat）。在切片模式下，输入的张量会被切分，部分通道会经过卷积操作，而在分割拼接模式下，输入会被分为两部分，第一部分经过卷积后与第二部分拼接。这种设计可以在训练和推理阶段灵活使用。  
  
`MLPBlock` 类实现了一个多层感知机（MLP）模块，包含了卷积层、归一化层和激活函数。这个模块还支持层级缩放（layer scale），可以在训练过程中调整输出的尺度。  
  
`BasicStage` 类则是由多个 `MLPBlock` 组成的基本阶段，负责处理输入数据并输出特征。  
  
`PatchEmbed` 和 `PatchMerging` 类分别实现了图像的嵌入和合并操作。`PatchEmbed` 将输入图像划分为不重叠的补丁并进行卷积处理，而 `PatchMerging` 则在不同阶段对特征图进行合并，以减少特征图的尺寸并增加通道数。  
  
`FasterNet` 类是整个模型的核心，包含了多个阶段（stages），每个阶段由 `BasicStage` 组成。模型的初始化方法允许用户自定义输入通道数、类别数、嵌入维度、深度等参数。模型还支持不同的归一化和激活函数。通过 `forward` 方法，模型能够输出四个阶段的特征图，适用于密集预测任务。  
  
文件中还定义了一些函数，用于加载模型的权重。这些函数读取配置文件（YAML 格式），根据配置初始化相应的 FasterNet 模型，并在需要时加载预训练权重。  
  
最后，在 `\_\_main\_\_` 部分，代码展示了如何使用 `fasternet\_t0` 函数来创建模型实例，并打印模型的输出通道和输入张量的尺寸。这部分代码可以作为测试和验证模型的基本功能。  
  
总体来说，这个文件实现了一个灵活且高效的图像处理模型，适用于各种计算机视觉任务。

```以下是经过简化和注释的核心代码部分：  
  
```python  
import math  
import torch  
from torch import nn  
from einops.layers.torch import Rearrange  
  
# 定义一个自定义的二维卷积层，带有特定的权重处理  
class Conv2d\_cd(nn.Module):  
 def \_\_init\_\_(self, in\_channels, out\_channels, kernel\_size=3, stride=1,  
 padding=1, dilation=1, groups=1, bias=False, theta=1.0):  
 super(Conv2d\_cd, self).\_\_init\_\_()  
 # 初始化标准卷积层  
 self.conv = nn.Conv2d(in\_channels, out\_channels, kernel\_size=kernel\_size, stride=stride, padding=padding, dilation=dilation, groups=groups, bias=bias)  
 self.theta = theta # 用于权重调整的参数  
  
 def get\_weight(self):  
 # 获取卷积层的权重并进行处理  
 conv\_weight = self.conv.weight # 获取卷积权重  
 conv\_shape = conv\_weight.shape # 获取权重的形状  
 # 将权重重排为 (输入通道数, 输出通道数, 卷积核大小)  
 conv\_weight = Rearrange('c\_in c\_out k1 k2 -> c\_in c\_out (k1 k2)')(conv\_weight)  
   
 # 创建一个新的权重张量并初始化为0  
 conv\_weight\_cd = torch.zeros(conv\_shape[0], conv\_shape[1], 3 \* 3, device=conv\_weight.device, dtype=conv\_weight.dtype)  
 # 将原始权重复制到新的权重张量  
 conv\_weight\_cd[:, :, :] = conv\_weight[:, :, :]  
 # 调整权重，使得中心点的权重为周围权重的负和  
 conv\_weight\_cd[:, :, 4] = conv\_weight[:, :, 4] - conv\_weight[:, :, :].sum(2)  
 # 将权重重排回原来的形状  
 conv\_weight\_cd = Rearrange('c\_in c\_out (k1 k2) -> c\_in c\_out k1 k2', k1=conv\_shape[2], k2=conv\_shape[3])(conv\_weight\_cd)  
 return conv\_weight\_cd, self.conv.bias # 返回处理后的权重和偏置  
  
  
# 定义一个组合卷积层，使用多个自定义卷积层  
class DEConv(nn.Module):  
 def \_\_init\_\_(self, dim):  
 super(DEConv, self).\_\_init\_\_()  
 # 初始化多个自定义卷积层  
 self.conv1\_1 = Conv2d\_cd(dim, dim, 3, bias=True)  
 self.conv1\_2 = Conv2d\_cd(dim, dim, 3, bias=True) # 这里可以替换为其他卷积类  
 self.conv1\_3 = Conv2d\_cd(dim, dim, 3, bias=True) # 这里可以替换为其他卷积类  
 self.conv1\_4 = Conv2d\_cd(dim, dim, 3, bias=True) # 这里可以替换为其他卷积类  
 self.conv1\_5 = nn.Conv2d(dim, dim, 3, padding=1, bias=True) # 最后一个标准卷积层  
   
 self.bn = nn.BatchNorm2d(dim) # 批归一化层  
 self.act = nn.ReLU() # 激活函数（可以替换为其他激活函数）  
  
 def forward(self, x):  
 # 前向传播  
 # 获取所有卷积层的权重和偏置  
 w1, b1 = self.conv1\_1.get\_weight()  
 w2, b2 = self.conv1\_2.get\_weight()  
 w3, b3 = self.conv1\_3.get\_weight()  
 w4, b4 = self.conv1\_4.get\_weight()  
 w5, b5 = self.conv1\_5.weight, self.conv1\_5.bias  
  
 # 将所有权重和偏置相加  
 w = w1 + w2 + w3 + w4 + w5  
 b = b1 + b2 + b3 + b4 + b5  
 # 使用合并后的权重和偏置进行卷积操作  
 res = nn.functional.conv2d(input=x, weight=w, bias=b, stride=1, padding=1, groups=1)  
   
 # 应用批归一化和激活函数  
 res = self.bn(res)  
 return self.act(res)  
  
 def switch\_to\_deploy(self):  
 # 切换到部署模式，合并权重和偏置  
 w1, b1 = self.conv1\_1.get\_weight()  
 w2, b2 = self.conv1\_2.get\_weight()  
 w3, b3 = self.conv1\_3.get\_weight()  
 w4, b4 = self.conv1\_4.get\_weight()  
 w5, b5 = self.conv1\_5.weight, self.conv1\_5.bias  
  
 # 合并权重和偏置  
 self.conv1\_5.weight = torch.nn.Parameter(w1 + w2 + w3 + w4 + w5)  
 self.conv1\_5.bias = torch.nn.Parameter(b1 + b2 + b3 + b4 + b5)  
   
 # 删除不再需要的卷积层  
 del self.conv1\_1  
 del self.conv1\_2  
 del self.conv1\_3  
 del self.conv1\_4  
  
# 示例代码（可选）  
if \_\_name\_\_ == '\_\_main\_\_':  
 data = torch.randn((1, 128, 64, 64)).cuda() # 随机输入数据  
 model = DEConv(128).cuda() # 初始化模型  
 output1 = model(data) # 前向传播  
 model.switch\_to\_deploy() # 切换到部署模式  
 output2 = model(data) # 再次前向传播  
 print(torch.allclose(output1, output2)) # 检查输出是否一致  
```  
  
### 代码注释说明：  
1. \*\*Conv2d\_cd 类\*\*：定义了一个自定义的二维卷积层，包含了权重的特殊处理逻辑。  
2. \*\*DEConv 类\*\*：组合多个卷积层，提供了前向传播和切换到部署模式的功能。  
3. \*\*get\_weight 方法\*\*：获取卷积层的权重并进行调整，确保卷积操作的特定性质。  
4. \*\*switch\_to\_deploy 方法\*\*：合并多个卷积层的权重和偏置，以便在推理时提高效率。```

这个程序文件 `deconv.py` 定义了一系列自定义的卷积层，主要用于深度学习中的卷积神经网络（CNN）。文件中包含多个类，每个类实现了不同类型的卷积操作，具体如下：  
  
首先，程序导入了必要的库，包括 `math`、`torch` 和 `torch.nn`，以及 `einops` 库中的 `Rearrange`，用于重排张量的维度。还导入了一个自定义的 `Conv` 模块和一个用于融合卷积和批归一化的工具。  
  
接下来，定义了多个卷积类，每个类都继承自 `nn.Module`。这些类分别是 `Conv2d\_cd`、`Conv2d\_ad`、`Conv2d\_rd`、`Conv2d\_hd` 和 `Conv2d\_vd`。每个类的构造函数都初始化了一个标准的二维卷积层 `nn.Conv2d`，并且可以接受多个参数，如输入通道数、输出通道数、卷积核大小、步幅、填充、扩张、分组和偏置。  
  
在每个卷积类中，`get\_weight` 方法用于获取卷积层的权重，并对其进行重排和处理，以便适应特定的卷积操作。例如，`Conv2d\_cd` 类在计算权重时，会对权重进行重排，并计算出新的权重形式；而 `Conv2d\_ad` 类则通过对权重进行特定的变换来生成新的权重。  
  
`Conv2d\_rd` 类的 `forward` 方法实现了前向传播逻辑，根据 `theta` 的值选择不同的卷积方式。如果 `theta` 接近于零，则使用标准卷积；否则，计算新的卷积权重并执行卷积操作。  
  
`Conv2d\_hd` 和 `Conv2d\_vd` 类的 `get\_weight` 方法也实现了类似的权重处理，目的是生成适合于不同操作的卷积权重。  
  
最后，定义了一个 `DEConv` 类，它将之前定义的多个卷积层组合在一起，形成一个更复杂的卷积模块。在 `forward` 方法中，依次调用各个卷积层的 `get\_weight` 方法，获取权重并进行卷积操作，最后通过批归一化和激活函数处理输出。  
  
`switch\_to\_deploy` 方法用于在模型部署时优化性能。它将多个卷积层的权重合并为一个卷积层的权重，并删除不再需要的卷积层，从而减少计算量和内存占用。  
  
在文件的最后部分，提供了一个简单的测试代码，创建一个随机输入张量并通过 `DEConv` 模型进行前向传播，验证模型在切换到部署模式前后的输出是否一致。  
  
总体来说，这个文件实现了一种灵活的卷积操作方式，适用于深度学习中的特定需求，尤其是在需要自定义卷积行为的场景中。

``````python  
import torch  
import torch.nn as nn  
  
class AGLU(nn.Module):  
 """统一激活函数模块，来自 https://github.com/kostas1515/AGLU。"""  
  
 def \_\_init\_\_(self, device=None, dtype=None) -> None:  
 """初始化统一激活函数模块。"""  
 super().\_\_init\_\_()  
 # 使用Softplus作为激活函数的一部分，beta设为-1.0  
 self.act = nn.Softplus(beta=-1.0)  
 # 初始化lambda参数，并将其定义为可学习的参数  
 self.lambd = nn.Parameter(nn.init.uniform\_(torch.empty(1, device=device, dtype=dtype))) # lambda参数  
 # 初始化kappa参数，并将其定义为可学习的参数  
 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:  
 """计算统一激活函数的前向传播。"""  
 # 将lambda参数限制在最小值0.0001，以避免除零错误  
 lam = torch.clamp(self.lambd, min=0.0001)  
 # 计算激活函数的输出  
 return torch.exp((1 / lam) \* self.act((self.kappa \* x) - torch.log(lam)))  
```  
  
### 代码注释说明：  
1. \*\*导入必要的库\*\*：导入`torch`和`torch.nn`模块，后者提供了构建神经网络所需的基本组件。  
2. \*\*AGLU类\*\*：定义了一个名为`AGLU`的类，继承自`nn.Module`，表示一个自定义的激活函数模块。  
3. \*\*初始化方法\*\*：  
 - `\_\_init\_\_`方法中，首先调用父类的初始化方法。  
 - 使用`Softplus`作为激活函数的一部分，`beta`参数设为-1.0。  
 - 初始化两个可学习的参数`lambd`和`kappa`，并将它们定义为`nn.Parameter`，以便在训练过程中更新。  
4. \*\*前向传播方法\*\*：  
 - `forward`方法接收一个张量`x`作为输入。  
 - 使用`torch.clamp`将`lambd`限制在最小值0.0001，以避免在后续计算中出现除零错误。  
 - 计算并返回激活函数的输出，公式中结合了`Softplus`激活和可学习参数的影响。```

这个程序文件名为 `activation.py`，主要定义了一个名为 `AGLU` 的激活函数模块。该模块是基于一个统一的激活函数，来源于一个开源项目。文件中使用了 PyTorch 库来构建神经网络模块。  
  
首先，文件导入了必要的库，包括 `torch` 和 `torch.nn`。`torch` 是 PyTorch 的核心库，而 `torch.nn` 提供了构建神经网络所需的各种功能和模块。  
  
在 `AGLU` 类的定义中，继承自 `nn.Module`，这使得 `AGLU` 成为一个可用于构建神经网络的模块。在类的初始化方法 `\_\_init\_\_` 中，首先调用了父类的初始化方法 `super().\_\_init\_\_()`。接着，定义了一个激活函数 `self.act`，使用了 `nn.Softplus`，其参数 `beta` 设置为 -1.0。`Softplus` 是一种平滑的激活函数，类似于 ReLU，但在零点附近更加平滑。  
  
此外，类中还定义了两个可学习的参数 `lambd` 和 `kappa`，它们都是通过 `nn.Parameter` 创建的，并且在初始化时使用均匀分布进行随机初始化。`lambd` 和 `kappa` 的初始值是根据指定的设备和数据类型生成的。  
  
在 `forward` 方法中，定义了前向传播的计算过程。输入 `x` 是一个张量，首先对 `lambd` 进行限制，确保其最小值为 0.0001，以避免在后续计算中出现除以零的情况。然后，计算并返回激活函数的输出，使用了指数函数和 `Softplus` 的组合。具体的计算过程是将输入 `x` 乘以 `kappa`，再减去 `lambd` 的对数，最后通过 `Softplus` 处理后，再乘以 `1/lambd` 的指数。  
  
总体而言，这个模块实现了一种新的激活函数，结合了可学习的参数和非线性变换，可能在深度学习模型中用于提高模型的表现。

```以下是经过简化和注释的核心代码部分，主要包含模型的构建和前向传播逻辑：  
  
```python  
import torch  
import torch.nn as nn  
from timm.models.layers import SqueezeExcite  
  
# 定义一个函数，用于替换模型中的 BatchNorm 层为 Identity 层  
def replace\_batchnorm(net):  
 for child\_name, child in net.named\_children():  
 if hasattr(child, 'fuse\_self'):  
 fused = child.fuse\_self() # 融合卷积和 BatchNorm  
 setattr(net, child\_name, fused)  
 replace\_batchnorm(fused)  
 elif isinstance(child, torch.nn.BatchNorm2d):  
 setattr(net, child\_name, torch.nn.Identity()) # 替换为 Identity  
 else:  
 replace\_batchnorm(child)  
  
# 定义一个可分配的函数，确保通道数是8的倍数  
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)  
 if new\_v < 0.9 \* v:  
 new\_v += divisor  
 return new\_v  
  
# 定义一个卷积层和 BatchNorm 的组合  
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\_\_()  
 self.add\_module('c', torch.nn.Conv2d(a, b, ks, stride, pad, dilation, groups, bias=False))  
 self.add\_module('bn', torch.nn.BatchNorm2d(b))  
 torch.nn.init.constant\_(self.bn.weight, bn\_weight\_init) # 初始化 BatchNorm 权重  
 torch.nn.init.constant\_(self.bn.bias, 0) # 初始化 BatchNorm 偏置  
  
 @torch.no\_grad()  
 def fuse\_self(self):  
 c, bn = self.\_modules.values()  
 w = bn.weight / (bn.running\_var + bn.eps)\*\*0.5 # 计算融合后的权重  
 w = c.weight \* w[:, None, None, None]  
 b = bn.bias - bn.running\_mean \* bn.weight / (bn.running\_var + bn.eps)\*\*0.5 # 计算融合后的偏置  
 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  
  
# 定义一个残差块  
class Residual(torch.nn.Module):  
 def \_\_init\_\_(self, m, drop=0.):  
 super().\_\_init\_\_()  
 self.m = m # 残差模块  
 self.drop = drop # 随机丢弃率  
  
 def forward(self, x):  
 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):  
 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]) # 添加边界  
 m.weight += identity.to(m.weight.device) # 融合残差  
 return m  
 else:  
 return self  
  
# 定义 RepViT 模块  
class RepViT(nn.Module):  
 def \_\_init\_\_(self, cfgs):  
 super().\_\_init\_\_()  
 self.cfgs = cfgs # 配置参数  
 input\_channel = self.cfgs[0][2] # 输入通道数  
 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] # 初始化层  
 block = RepViTBlock # 定义块  
 for k, t, c, use\_se, use\_hs, s in self.cfgs:  
 output\_channel = \_make\_divisible(c, 8) # 计算输出通道数  
 exp\_size = \_make\_divisible(input\_channel \* t, 8) # 计算扩展通道数  
 layers.append(block(input\_channel, exp\_size, output\_channel, k, s, use\_se, use\_hs)) # 添加块  
 input\_channel = output\_channel # 更新输入通道数  
 self.features = nn.ModuleList(layers) # 将所有层组合成一个模块列表  
  
 def forward(self, x):  
 for f in self.features:  
 x = f(x) # 前向传播  
 return x # 返回输出  
  
# 具体模型构建函数  
def repvit\_m2\_3(weights=''):  
 cfgs = [  
 # k, t, c, SE, HS, s   
 [3, 2, 80, 1, 0, 1],  
 # ... 省略其他配置  
 ]  
 model = RepViT(cfgs) # 创建模型  
 if weights:  
 model.load\_state\_dict(torch.load(weights)['model']) # 加载权重  
 return model  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 model = repvit\_m2\_3('repvit\_m2\_3\_distill\_450e.pth') # 实例化模型  
 inputs = torch.randn((1, 3, 640, 640)) # 创建输入张量  
 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. \*\*replace\_batchnorm\*\*: 该函数遍历模型的所有子模块，将 `BatchNorm2d` 层替换为 `Identity` 层，以便在推理时加速计算。  
2. \*\*\_make\_divisible\*\*: 确保给定的通道数是8的倍数，通常用于深度学习模型中以提高计算效率。  
3. \*\*Conv2d\_BN\*\*: 自定义的卷积层，包含卷积和 BatchNorm 的组合，并提供了融合方法以便在推理时减少计算量。  
4. \*\*Residual\*\*: 残差模块，支持随机丢弃，以提高模型的鲁棒性。  
5. \*\*RepViT\*\*: 主要的模型类，构建了整个网络结构，并实现了前向传播逻辑。  
6. \*\*repvit\_m2\_3\*\*: 构建特定配置的 RepViT 模型，并可选择加载预训练权重。```

这个程序文件 `repvit.py` 实现了一个基于深度学习的模型，主要用于图像处理任务。该模型的结构是 RepViT（Residual Vision Transformer），结合了卷积神经网络（CNN）和视觉变换器（ViT）的特性。代码中包含了多个类和函数，下面对其进行逐一说明。  
  
首先，文件导入了必要的库，包括 PyTorch 的神经网络模块、NumPy 以及 `timm` 库中的 SqueezeExcite 层。接着，定义了一个 `replace\_batchnorm` 函数，用于在模型中替换 BatchNorm 层为 Identity 层，这在模型推理时可以提高效率。  
  
`\_make\_divisible` 函数确保模型中所有层的通道数都是 8 的倍数，以满足特定的硬件要求。接下来，定义了 `Conv2d\_BN` 类，它是一个组合了卷积层和 BatchNorm 层的顺序容器，并在初始化时设置了 BatchNorm 的权重和偏置。  
  
`Residual` 类实现了残差连接，允许在训练时通过随机丢弃部分输入来增强模型的鲁棒性。`RepVGGDW` 类则是一个特定的卷积块，结合了深度可分离卷积和残差连接。  
  
`RepViTBlock` 类实现了 RepViT 的基本构建块，包含了通道混合和令牌混合的操作。根据步幅的不同，该类会选择不同的结构来处理输入。  
  
`RepViT` 类是整个模型的核心，负责构建网络的层次结构。它接收一个配置列表，逐层构建网络。`forward` 方法定义了前向传播的过程，并在特定的尺度下提取特征。  
  
`switch\_to\_deploy` 方法用于在推理阶段替换 BatchNorm 层，以提高推理速度。`update\_weight` 函数则用于更新模型的权重，确保加载的权重与模型结构匹配。  
  
接下来，定义了多个函数（如 `repvit\_m0\_9`、`repvit\_m1\_0` 等），每个函数都构建了一个特定配置的 RepViT 模型，并可以选择加载预训练的权重。  
  
最后，在 `\_\_main\_\_` 块中，程序实例化了一个 RepViT 模型，并生成了一个随机输入以测试模型的前向传播。程序输出了每个特征图的尺寸，便于检查模型的结构是否正确。  
  
整体来看，这个程序文件展示了如何构建一个复杂的深度学习模型，结合了卷积神经网络和变换器的优点，适用于图像分类、目标检测等任务。

### 整体功能和构架概括  
  
该工程包含多个模块，主要用于构建和训练深度学习模型，特别是在计算机视觉任务中。每个文件实现了不同的功能模块，涵盖了模型的构建、激活函数的定义、卷积操作的实现以及特定网络结构的设计。整体架构可以分为以下几个部分：  
  
1. \*\*模型定义\*\*：包括 FasterNet 和 RepViT 模型的实现，支持不同的网络结构和特性。  
2. \*\*激活函数\*\*：定义了自定义的激活函数 AGLU，增强了模型的非线性表达能力。  
3. \*\*卷积操作\*\*：提供了多种卷积层的实现，包括深度可分离卷积和残差连接，灵活地处理输入数据。  
4. \*\*工具函数\*\*：包含了一些辅助函数，用于模型的初始化、权重更新和推理阶段的优化。  
  
### 文件功能整理表  
  
| 文件名 | 功能描述 |  
|-------------------|------------------------------------------------------------------|  
| `fasternet.py` | 实现了 FasterNet 模型，定义了多个卷积层和特征提取模块，支持目标检测和图像分割任务。 |  
| `deconv.py` | 定义了自定义的卷积层，包括部分卷积和多层感知机模块，支持灵活的卷积操作。 |  
| `activation.py` | 实现了 AGLU 激活函数，结合可学习参数和非线性变换，增强模型的表现。 |  
| `repvit.py` | 实现了 RepViT 模型，结合了卷积神经网络和视觉变换器的特性，适用于图像处理任务。 |  
  
这些模块共同构成了一个强大的深度学习框架，能够处理多种计算机视觉任务，并提供了灵活的模型设计和训练机制。

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。