# 改进yolo11-HSFPN等200+全套创新点大全：水稻病害检测系统源码＆数据集全套

## 3.背景

研究背景与意义  
  
水稻是全球范围内重要的粮食作物之一，尤其在亚洲地区，水稻不仅是主要的食物来源，也是农民生计的重要保障。然而，水稻在生长过程中容易受到多种病害的侵袭，其中水稻稻瘟病（blast）和叶斑病（leaf）是最为常见且危害严重的病害。这些病害不仅会导致水稻产量下降，还会影响水稻的品质，进而对农民的经济收入造成负面影响。因此，及时、准确地检测和识别水稻病害，对于保障粮食安全和提高农业生产效率具有重要的现实意义。  
  
随着计算机视觉技术的迅速发展，基于深度学习的图像识别方法在农业病害检测中展现出了良好的应用前景。YOLO（You Only Look Once）系列模型因其高效的实时检测能力，已成为目标检测领域的主流方法之一。特别是YOLOv11的推出，进一步提升了检测精度和速度，为水稻病害的自动识别提供了新的技术手段。本研究旨在基于改进的YOLOv11模型，构建一个高效的水稻病害检测系统，以实现对稻瘟病和叶斑病的快速识别和分类。  
  
本项目将利用一个包含5800张图像的数据集，数据集中涵盖了稻瘟病和叶斑病两个类别，经过精心标注，能够为模型的训练提供丰富的样本。通过对图像进行预处理和数据增强，提升模型的泛化能力，从而提高病害检测的准确性和鲁棒性。最终，该系统的应用将有助于农民及时发现和处理水稻病害，降低损失，提高水稻的产量和质量，为实现可持续农业发展贡献力量。

## 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水稻病害检测，特别是针对水稻的“稻瘟病”及其叶片病害的评估。数据集的主题为“rice\_blast\_grade\_assessment”，旨在为改进YOLOv11模型提供高质量的训练数据，以提高其在水稻病害识别方面的准确性和效率。该数据集包含两类主要的病害类型，分别为“blast”（稻瘟病）和“leaf”（叶片病害），总共涵盖了两种类别。这些类别的选择不仅反映了水稻在生长过程中常见的病害类型，也为模型的训练提供了丰富的样本。  
  
在数据集的构建过程中，数据采集涵盖了不同生长阶段的水稻植株，确保了样本的多样性和代表性。每个类别的样本均经过精心标注，确保在训练过程中模型能够学习到有效的特征。稻瘟病和叶片病害的样本不仅包括不同病害程度的表现，还涵盖了不同环境条件下的表现，从而增强了模型的泛化能力。  
  
此外，为了提高数据集的实用性，样本图像的拍摄角度、光照条件和背景环境均有所变化。这种多样化的样本设计使得模型在实际应用中能够更好地适应不同的农业环境，进而提高病害检测的准确性和可靠性。通过对该数据集的深入分析与应用，期望能够为水稻种植者提供更为高效的病害监测工具，进而推动农业生产的可持续发展。

## 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部分：  
  
```python  
import torch  
import torch.nn as nn  
  
# 自定义的批量归一化层，包含一个可学习的参数 alpha  
class RepBN(nn.Module):  
 def \_\_init\_\_(self, channels):  
 super(RepBN, self).\_\_init\_\_()  
 # 初始化可学习参数 alpha，初始值为 1  
 self.alpha = nn.Parameter(torch.ones(1))  
 # 初始化 1D 批量归一化层  
 self.bn = nn.BatchNorm1d(channels)  
  
 def forward(self, x):  
 # 将输入张量的维度进行转置，以适应 BatchNorm 的输入格式  
 x = x.transpose(1, 2)  
 # 进行批量归一化，并加上 alpha 乘以原始输入  
 x = self.bn(x) + self.alpha \* x  
 # 再次转置回原来的维度  
 x = x.transpose(1, 2)  
 return x  
  
  
# 自定义的线性归一化层，支持两种归一化方式的线性组合  
class LinearNorm(nn.Module):  
 def \_\_init\_\_(self, dim, norm1, norm2, warm=0, step=300000, r0=1.0):  
 super(LinearNorm, self).\_\_init\_\_()  
 # 注册一些缓冲区变量，用于控制训练过程中的参数  
 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  
 # 初始化两种归一化方式  
 self.norm1 = norm1(dim)  
 self.norm2 = norm2(dim)  
  
 def forward(self, x):  
 if self.training: # 如果处于训练模式  
 if self.warm > 0: # 如果仍在预热阶段  
 self.warm.copy\_(self.warm - 1) # 递减预热计数  
 x = self.norm1(x) # 仅使用 norm1 进行归一化  
 else:  
 # 计算当前的比例 lamda  
 lamda = self.r0 \* self.iter / self.total\_step  
 if self.iter > 0:  
 self.iter.copy\_(self.iter - 1) # 递减迭代计数  
 # 分别使用两种归一化方式  
 x1 = self.norm1(x)  
 x2 = self.norm2(x)  
 # 按照 lamda 比例线性组合两种归一化结果  
 x = lamda \* x1 + (1 - lamda) \* x2  
 else:  
 # 如果处于评估模式，直接使用 norm2 进行归一化  
 x = self.norm2(x)  
 return x  
```  
  
### 代码说明：  
1. \*\*RepBN 类\*\*：  
 - 该类实现了一个自定义的批量归一化层，除了标准的批量归一化操作外，还引入了一个可学习的参数 `alpha`，用于调整输入特征的影响。  
  
2. \*\*LinearNorm 类\*\*：  
 - 该类实现了一个线性归一化层，支持在训练过程中动态调整使用的归一化方式。它根据预热阶段和迭代次数来决定使用哪种归一化方式的组合。  
 - 在训练初期（预热阶段），只使用 `norm1` 进行归一化；在预热结束后，根据当前迭代次数和总迭代次数计算一个比例 `lamda`，用于线性组合 `norm1` 和 `norm2` 的输出。```

这个程序文件`prepbn.py`定义了两个神经网络模块，分别是`RepBN`和`LinearNorm`，它们都继承自PyTorch的`nn.Module`类。  
  
首先，`RepBN`类是一个自定义的批量归一化模块。它的构造函数接受一个参数`channels`，表示输入数据的通道数。在构造函数中，首先调用父类的构造函数，然后定义了一个可学习的参数`alpha`，初始值为1，并创建了一个标准的1维批量归一化层`bn`。在前向传播方法`forward`中，输入张量`x`的维度被转置，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，最后再将结果转置回原来的维度并返回。这种设计使得网络在进行批量归一化的同时，还能保留原始输入的特征，通过`alpha`参数的学习来调节两者的权重。  
  
接下来是`LinearNorm`类，它实现了一种线性归一化的机制。构造函数接受多个参数，包括`dim`（维度）、`norm1`和`norm2`（分别是两种归一化方法的类），以及`warm`（预热步数）、`step`（迭代步数）和`r0`（初始比例因子）。在构造函数中，使用`register\_buffer`方法注册了一些常量，这些常量在模型训练时不会被视为模型参数。`forward`方法中包含了训练和推理两种模式的处理逻辑。在训练模式下，如果还有预热步数，则使用第一种归一化方法`norm1`处理输入；否则，计算一个动态的比例因子`lamda`，并根据当前的迭代步数决定使用哪种归一化方法。最终的输出是这两种归一化结果的加权和。在推理模式下，直接使用第二种归一化方法`norm2`处理输入。  
  
总体来看，这个文件中的模块设计灵活，能够在不同的训练阶段动态调整归一化策略，旨在提高模型的训练效果和性能。

```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  
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) # 重新调整形状  
 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) # 重新调整形状  
 # 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  
  
# 定义逆小波变换的类  
class InverseWaveletTransform(Function):  
 @staticmethod  
 def forward(ctx, input, filters):  
 ctx.filters = filters # 保存滤波器  
 with torch.no\_grad():  
 x = inverse\_wavelet\_transform(input, filters) # 执行逆小波变换  
 return x  
  
 @staticmethod  
 def backward(ctx, grad\_output):  
 grad = 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) # 不需要训练的小波滤波器  
 self.iwt\_filter = nn.Parameter(self.iwt\_filter, requires\_grad=False) # 不需要训练的逆小波滤波器  
   
 # 初始化小波变换和逆小波变换的函数  
 self.wt\_function = wavelet\_transform\_init(self.wt\_filter)  
 self.iwt\_function = inverse\_wavelet\_transform\_init(self.iwt\_filter)  
  
 # 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)  
  
 # 小波卷积层  
 self.wavelet\_convs = nn.ModuleList(  
 [nn.Conv2d(in\_channels\*4, in\_channels\*4, kernel\_size, padding='same', stride=1, groups=in\_channels\*4, bias=False) for \_ in range(self.wt\_levels)]  
 )  
  
 def forward(self, x):  
 # 前向传播过程  
 x\_ll\_in\_levels = [] # 存储低频分量  
 x\_h\_in\_levels = [] # 存储高频分量  
  
 curr\_x\_ll = x # 当前低频分量  
  
 for i in range(self.wt\_levels):  
 curr\_x = self.wt\_function(curr\_x\_ll) # 执行小波变换  
 curr\_x\_ll = curr\_x[:,:,0,:,:] # 更新低频分量  
   
 # 处理高频分量  
 curr\_x\_h = curr\_x[:,:,1:4,:,:]  
 x\_ll\_in\_levels.append(curr\_x\_ll)  
 x\_h\_in\_levels.append(curr\_x\_h)  
  
 # 逆小波变换  
 next\_x\_ll = 0  
 for i in range(self.wt\_levels-1, -1, -1):  
 curr\_x\_ll = x\_ll\_in\_levels.pop()  
 curr\_x\_h = x\_h\_in\_levels.pop()  
 curr\_x = torch.cat([curr\_x\_ll.unsqueeze(2), curr\_x\_h], dim=2) # 合并低频和高频分量  
 next\_x\_ll = self.iwt\_function(curr\_x) # 执行逆小波变换  
  
 x = self.base\_conv(x) + next\_x\_ll # 添加基础卷积的输出  
 return x  
```  
  
### 代码核心部分说明：  
1. \*\*小波滤波器的创建\*\*：`create\_wavelet\_filter` 函数用于生成小波变换和逆变换所需的滤波器。  
2. \*\*小波变换和逆变换\*\*：`wavelet\_transform` 和 `inverse\_wavelet\_transform` 函数分别实现小波变换和逆变换的具体操作。  
3. \*\*自定义的自动求导功能\*\*：`WaveletTransform` 和 `InverseWaveletTransform` 类继承自 `Function`，实现了前向和反向传播的逻辑。  
4. \*\*小波卷积层\*\*：`WTConv2d` 类是一个卷积层，结合了小波变换的特性，能够在卷积过程中同时处理低频和高频信息。```

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。文件中使用了 PyTorch 框架，并结合了小波变换的数学原理，提供了对输入图像的多层次分解和重构功能。  
  
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 `create\_wavelet\_filter` 函数，该函数根据指定的小波类型生成小波变换和逆小波变换所需的滤波器。滤波器的生成过程包括反转小波的分解和重构系数，并通过张量操作构建出滤波器。  
  
接下来，定义了 `wavelet\_transform` 和 `inverse\_wavelet\_transform` 函数，分别用于执行小波变换和逆小波变换。这两个函数利用卷积操作实现对输入数据的变换，支持对多个通道的处理。  
  
在小波变换的实现中，使用了 PyTorch 的 `Function` 类来定义前向和反向传播的操作。`WaveletTransform` 和 `InverseWaveletTransform` 类分别实现了小波变换和逆变换的具体逻辑。在前向传播中，调用相应的变换函数，而在反向传播中，则计算梯度并返回。  
  
随后，定义了 `WTConv2d` 类，继承自 `nn.Module`，这是实现小波卷积的核心类。在初始化方法中，设置了输入输出通道数、卷积核大小、步幅等参数，并生成小波滤波器。该类还定义了多个卷积层和缩放模块，用于处理不同层次的小波特征。  
  
在 `forward` 方法中，首先对输入进行小波变换，并将结果分为低频和高频部分。然后，逐层处理这些特征，最后通过逆小波变换将特征重构回原始空间。重构后的特征与基础卷积层的输出相加，形成最终的输出。  
  
最后，定义了一个 `\_ScaleModule` 类，用于对输入进行缩放操作。该模块通过学习得到的参数对输入进行加权，增强模型的灵活性。  
  
整体来看，这个程序文件实现了一个复杂的卷积层，结合了小波变换的优势，能够有效地提取图像中的多尺度特征，并提供了灵活的前向和反向传播机制，适用于深度学习中的图像处理任务。

```以下是经过简化并添加详细中文注释的核心代码部分：  
  
```python  
try:  
 # 尝试导入swattention模块和TransNext\_cuda中的所有内容  
 import swattention  
 from ultralytics.nn.backbone.TransNeXt.TransNext\_cuda import \*  
except ImportError as e:  
 # 如果导入失败（例如swattention模块不存在），则导入TransNext\_native中的所有内容  
 from ultralytics.nn.backbone.TransNeXt.TransNext\_native import \*  
 pass # 继续执行后续代码，不做任何处理  
```  
  
### 代码说明：  
1. \*\*导入模块\*\*：  
 - 首先尝试导入`swattention`模块，这可能是一个用于实现注意力机制的库。  
 - 然后从`ultralytics.nn.backbone.TransNeXt`路径下导入`TransNext\_cuda`中的所有内容，这通常是一个基于CUDA的实现，适合在支持GPU的环境中运行。  
  
2. \*\*异常处理\*\*：  
 - 如果在导入过程中发生`ImportError`（例如，`swattention`模块未安装），则进入`except`块。  
 - 在`except`块中，导入`TransNext\_native`中的所有内容，这通常是一个不依赖CUDA的实现，适合在不支持GPU的环境中运行。  
  
3. \*\*`pass`语句\*\*：  
 - `pass`语句表示在捕获到异常后不执行任何操作，继续执行后续代码。这是一个占位符，表示在这里不需要做任何处理。  
  
这种结构确保了代码在不同环境下的灵活性，能够根据可用的库和硬件条件选择合适的实现。```

这个程序文件名为 `TransNext.py`，其主要功能是导入与 `TransNeXt` 相关的模块。代码的结构采用了 `try-except` 语句，目的是为了处理可能出现的导入错误。  
  
首先，程序尝试从 `swattention` 模块中导入 `TransNext\_cuda`。这个模块可能是针对 CUDA 加速的实现，适用于支持 GPU 的环境，以提高计算性能。如果导入成功，程序将继续执行后续的代码（虽然在这段代码中没有显示后续内容）。  
  
如果在导入 `TransNext\_cuda` 时发生 `ImportError`，程序会进入 `except` 块。在这个块中，程序会尝试导入 `TransNext\_native` 模块。这个模块可能是一个不依赖于 CUDA 的实现，适用于不支持 GPU 的环境。通过这种方式，程序能够在不同的运行环境中灵活选择合适的模块，从而提高了代码的兼容性和可移植性。  
  
总的来说，这段代码的设计考虑到了不同硬件环境的需求，确保在无法使用 CUDA 的情况下，仍然能够正常运行程序。

```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 DWConv2d(nn.Module):  
 """ 深度可分离卷积类 """  
 def \_\_init\_\_(self, dim, kernel\_size, stride, padding):  
 super().\_\_init\_\_()  
 # 使用 groups=dim 实现深度可分离卷积  
 self.conv = nn.Conv2d(dim, dim, kernel\_size, stride, padding, groups=dim)  
  
 def forward(self, x: torch.Tensor):  
 """ 前向传播函数  
 x: 输入张量，形状为 (b, h, w, c)  
 """  
 x = x.permute(0, 3, 1, 2) # 转换为 (b, c, h, w)  
 x = self.conv(x) # 卷积操作  
 x = x.permute(0, 2, 3, 1) # 转换回 (b, h, w, c)  
 return x  
  
class MaSA(nn.Module):  
 """ 多头自注意力机制类 """  
 def \_\_init\_\_(self, embed\_dim, num\_heads, value\_factor=1):  
 super().\_\_init\_\_()  
 self.factor = value\_factor  
 self.embed\_dim = embed\_dim  
 self.num\_heads = num\_heads  
 self.head\_dim = self.embed\_dim \* self.factor // num\_heads  
 self.key\_dim = self.embed\_dim // num\_heads  
 self.scaling = self.key\_dim \*\* -0.5  
   
 # 线性变换层  
 self.q\_proj = nn.Linear(embed\_dim, embed\_dim, bias=True)  
 self.k\_proj = nn.Linear(embed\_dim, embed\_dim, bias=True)  
 self.v\_proj = nn.Linear(embed\_dim, embed\_dim \* self.factor, bias=True)  
 self.out\_proj = nn.Linear(embed\_dim \* self.factor, embed\_dim, bias=True)  
  
 def forward(self, x: torch.Tensor, rel\_pos):  
 """ 前向传播函数  
 x: 输入张量，形状为 (b, h, w, c)  
 rel\_pos: 位置关系矩阵  
 """  
 bsz, h, w, \_ = x.size()  
 q = self.q\_proj(x) # 计算查询  
 k = self.k\_proj(x) # 计算键  
 v = self.v\_proj(x) # 计算值  
  
 # 计算注意力权重  
 qk\_mat = (q @ k.transpose(-1, -2)) \* self.scaling + rel\_pos  
 qk\_mat = torch.softmax(qk\_mat, dim=-1) # 归一化  
  
 # 计算输出  
 output = (qk\_mat @ v)  
 output = self.out\_proj(output) # 最终线性变换  
 return output  
  
class FeedForwardNetwork(nn.Module):  
 """ 前馈神经网络类 """  
 def \_\_init\_\_(self, embed\_dim, ffn\_dim, activation\_fn=F.gelu, dropout=0.0):  
 super().\_\_init\_\_()  
 self.fc1 = nn.Linear(embed\_dim, ffn\_dim) # 第一层线性变换  
 self.fc2 = nn.Linear(ffn\_dim, embed\_dim) # 第二层线性变换  
 self.dropout = nn.Dropout(dropout) # Dropout层  
 self.activation\_fn = activation\_fn # 激活函数  
  
 def forward(self, x: torch.Tensor):  
 """ 前向传播函数  
 x: 输入张量，形状为 (b, h, w, c)  
 """  
 x = self.fc1(x) # 线性变换  
 x = self.activation\_fn(x) # 激活  
 x = self.dropout(x) # Dropout  
 x = self.fc2(x) # 线性变换  
 return x  
  
class VisRetNet(nn.Module):  
 """ 视觉恢复网络类 """  
 def \_\_init\_\_(self, in\_chans=3, num\_classes=1000, embed\_dims=[96, 192, 384, 768], depths=[2, 2, 6, 2], num\_heads=[3, 6, 12, 24]):  
 super().\_\_init\_\_()  
 self.patch\_embed = PatchEmbed(in\_chans=in\_chans, embed\_dim=embed\_dims[0]) # 图像分块嵌入  
 self.layers = nn.ModuleList() # 存储各层  
  
 # 构建各层  
 for i\_layer in range(len(depths)):  
 layer = BasicLayer(embed\_dim=embed\_dims[i\_layer], depth=depths[i\_layer], num\_heads=num\_heads[i\_layer])  
 self.layers.append(layer)  
  
 def forward(self, x):  
 """ 前向传播函数  
 x: 输入张量，形状为 (b, c, h, w)  
 """  
 x = self.patch\_embed(x) # 图像嵌入  
 for layer in self.layers:  
 x = layer(x) # 逐层处理  
 return x  
  
# 示例模型创建  
def RMT\_T():  
 model = VisRetNet(  
 embed\_dims=[64, 128, 256, 512],  
 depths=[2, 2, 8, 2],  
 num\_heads=[4, 4, 8, 16]  
 )  
 return model  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 model = RMT\_T() # 创建模型  
 inputs = torch.randn((1, 3, 640, 640)) # 随机输入  
 res = model(inputs) # 前向传播  
 print(res.size()) # 输出结果尺寸  
```  
  
### 代码说明：  
1. \*\*DWConv2d\*\*: 实现深度可分离卷积，适用于处理特征图。  
2. \*\*MaSA\*\*: 实现多头自注意力机制，计算输入的注意力权重并生成输出。  
3. \*\*FeedForwardNetwork\*\*: 实现前馈神经网络，包含两个线性层和激活函数。  
4. \*\*VisRetNet\*\*: 视觉恢复网络，包含图像分块嵌入和多个基本层。  
5. \*\*RMT\_T\*\*: 创建一个小型的视觉恢复网络模型示例。  
  
这段代码展示了一个视觉模型的核心组件，使用了深度学习中的自注意力机制和前馈网络结构。```

这个程序文件 `rmt.py` 实现了一个基于视觉变换器（Vision Transformer）的深度学习模型，名为 VisRetNet。该模型通过多种模块和层来处理输入图像，进行特征提取和分类。以下是对代码的详细说明。  
  
首先，文件导入了必要的库，包括 PyTorch 和一些辅助模块。接着，定义了一些基本的神经网络组件，如 `DWConv2d`、`RelPos2d`、`MaSAd`、`MaSA`、`FeedForwardNetwork`、`RetBlock`、`PatchMerging`、`BasicLayer`、`LayerNorm2d` 和 `PatchEmbed`。  
  
`DWConv2d` 是一个深度可分离卷积层，用于处理输入特征图的空间信息。`RelPos2d` 用于生成二维相对位置编码，帮助模型理解输入数据的空间关系。`MaSAd` 和 `MaSA` 是多头自注意力机制的实现，分别支持不同的注意力计算方式。`FeedForwardNetwork` 实现了前馈神经网络，用于进一步处理特征。  
  
`RetBlock` 是一个残差块，结合了注意力机制和前馈网络，并可以选择性地应用层归一化和层缩放。`PatchMerging` 用于将输入特征图的补丁合并，减少特征图的尺寸。`BasicLayer` 代表了模型的基本层，包含多个残差块和可能的下采样操作。  
  
`LayerNorm2d` 是一个二维层归一化层，适用于处理图像数据。`PatchEmbed` 则将输入图像分割成多个补丁，并将其嵌入到高维空间中。  
  
`VisRetNet` 是模型的主类，负责将各个层组合在一起。它的构造函数接收多个参数，定义了嵌入维度、层数、头数等超参数。模型通过 `patch\_embed` 将输入图像转换为补丁，然后通过多个层进行处理，最终输出特征。  
  
在文件的最后部分，定义了几个函数 `RMT\_T`、`RMT\_S`、`RMT\_B` 和 `RMT\_L`，这些函数分别创建不同规模的 VisRetNet 模型。每个函数都指定了不同的嵌入维度、深度、头数等参数，以适应不同的应用场景。  
  
最后，程序的主入口部分创建了一个小型模型 `RMT\_T`，并对随机生成的输入数据进行前向传播，输出特征图的尺寸。这部分代码用于测试模型的构建和功能。  
  
总体来说，这个程序实现了一个复杂的视觉变换器模型，结合了多种先进的技术，如自注意力机制、深度可分离卷积和残差连接，适用于图像分类等任务。

### 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，特别是在图像处理和特征提取方面。整体架构由多个自定义神经网络组件组成，结合了先进的技术，如小波变换、视觉变换器和自注意力机制。各个模块相互协作，以实现高效的特征提取和分类任务。具体来说：  
  
- \*\*prepbn.py\*\*：实现了自定义的批量归一化和线性归一化模块，旨在提高模型的训练效果。  
- \*\*wtconv2d.py\*\*：实现了基于小波变换的二维卷积层，能够有效提取图像中的多尺度特征。  
- \*\*TransNext.py\*\*：处理与视觉变换器相关的模块导入，确保在不同硬件环境下的兼容性。  
- \*\*rmt.py\*\*：实现了一个基于视觉变换器的深度学习模型，结合了多种神经网络组件，适用于图像分类等任务。  
  
### 文件功能整理表  
  
| 文件名 | 功能描述 |  
|------------------|----------------------------------------------------------------------------------------------|  
| `prepbn.py` | 实现自定义的批量归一化和线性归一化模块，通过学习参数调整归一化策略，提高模型训练效果。 |  
| `wtconv2d.py` | 实现基于小波变换的二维卷积层，支持多层次特征提取，适用于图像处理任务。 |  
| `TransNext.py` | 处理视觉变换器相关模块的导入，确保在支持和不支持 CUDA 的环境中均能正常运行。 |  
| `rmt.py` | 实现一个基于视觉变换器的深度学习模型，结合多种神经网络组件，适用于图像分类和特征提取任务。 |  
  
这个表格总结了每个文件的主要功能，帮助理解整个工程的结构和目的。