# 改进yolo11-attention等200+全套创新点大全：作物病害检测系统源码＆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  
  
随着全球气候变化和农业生产方式的转变，作物病害的发生频率和范围逐渐增加，严重影响了农业的可持续发展和粮食安全。作物病害不仅导致农作物减产，还可能引发生态失衡，影响农民的经济收入。因此，及时、准确地检测和识别作物病害，成为现代农业管理中亟待解决的重要问题。传统的人工检测方法不仅耗时耗力，而且容易受到主观因素的影响，导致漏检和误检现象频发。基于计算机视觉和深度学习的自动化检测系统，因其高效性和准确性，逐渐成为研究的热点。  
  
本研究旨在基于改进的YOLOv11模型，构建一个高效的作物病害检测系统。YOLO（You Only Look Once）系列模型以其实时性和高精度的特性，在目标检测领域取得了显著的成果。通过对YOLOv11进行改进，结合Mask RCNN的实例分割能力，我们希望在检测作物病害的同时，提供更为细致的分割信息，从而提高病害识别的准确性和可靠性。研究中使用的数据集包含30张图像，涵盖了早疫病、晚疫病和叶片等类别，为模型的训练和测试提供了基础。  
  
通过对这些数据的深入分析与处理，我们期望能够构建一个具有较强泛化能力的作物病害检测系统，能够在不同环境和条件下有效识别和分类作物病害。这不仅将为农民提供及时的病害预警，帮助其采取有效的防治措施，还将为农业生产的智能化和精准化管理提供重要的技术支持，推动农业科技的进步与发展。

## 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“Mask RCNN”，其主要目的是为了训练和改进YOLOv11模型在作物病害检测系统中的表现。该数据集包含三种主要类别，分别为“Early\_blight”（早期枯萎病）、“Late\_blight”（晚期枯萎病）和“Leaf”（叶片健康状态）。通过对这些类别的细致标注和分类，我们能够为模型提供丰富的训练样本，以提高其在实际应用中的准确性和鲁棒性。  
  
在农业生产中，作物病害的早期识别至关重要，能够有效地减少农作物损失，提高产量和质量。为此，Mask RCNN数据集不仅包含了不同病害的图像样本，还涵盖了健康叶片的图像，以便模型能够更好地区分病害与健康状态。这种多样化的样本选择使得模型在训练过程中能够学习到更为复杂的特征，从而在面对实际环境中的多变情况时，依然能够保持较高的识别率。  
  
数据集中的图像经过精心挑选和标注，确保了每个类别的样本在数量和质量上的均衡。这种平衡不仅有助于避免模型在训练过程中出现偏差，还能提高其在不同病害识别上的灵活性。此外，数据集还考虑到了不同生长阶段和环境条件下作物的表现，进一步增强了模型的适应性。  
  
综上所述，Mask RCNN数据集为本项目提供了一个坚实的基础，使得改进后的YOLOv11模型能够在作物病害检测领域发挥更大的作用，助力农业智能化的发展。通过对该数据集的深入分析和应用，我们期望能够实现更高效的病害监测和管理，推动现代农业的可持续发展。

## 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

```以下是对代码中最核心部分的分析和详细注释。代码主要实现了一些自定义的激活函数、动态ReLU模块、动态卷积模块以及一个结合了这些模块的DyHeadBlock。  
  
```python  
import torch  
import torch.nn as nn  
import torch.nn.functional as F  
  
# 定义一个函数用于将输入值调整为可被指定的除数整除的值  
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)  
 # 确保向下取整不会超过原值的90%  
 if new\_v < 0.9 \* v:  
 new\_v += divisor  
 return new\_v  
  
# Swish激活函数  
class swish(nn.Module):  
 def forward(self, x):  
 return x \* torch.sigmoid(x)  
  
# Hard Swish激活函数  
class h\_swish(nn.Module):  
 def \_\_init\_\_(self, inplace=False):  
 super(h\_swish, self).\_\_init\_\_()  
 self.inplace = inplace  
  
 def forward(self, x):  
 return x \* F.relu6(x + 3.0, inplace=self.inplace) / 6.0  
  
# Hard Sigmoid激活函数  
class h\_sigmoid(nn.Module):  
 def \_\_init\_\_(self, inplace=True, h\_max=1):  
 super(h\_sigmoid, self).\_\_init\_\_()  
 self.relu = nn.ReLU6(inplace=inplace)  
 self.h\_max = h\_max  
  
 def forward(self, x):  
 return self.relu(x + 3) \* self.h\_max / 6  
  
# 动态ReLU模块  
class DyReLU(nn.Module):  
 def \_\_init\_\_(self, inp, reduction=4, lambda\_a=1.0, K2=True, use\_bias=True, use\_spatial=False,  
 init\_a=[1.0, 0.0], init\_b=[0.0, 0.0]):  
 super(DyReLU, self).\_\_init\_\_()  
 self.oup = inp # 输出通道数  
 self.lambda\_a = lambda\_a \* 2 # 动态调整因子  
 self.K2 = K2 # 控制参数  
 self.avg\_pool = nn.AdaptiveAvgPool2d(1) # 自适应平均池化  
  
 # 确定压缩比  
 squeeze = inp // reduction if reduction == 4 else \_make\_divisible(inp // reduction, 4)  
  
 # 定义全连接层  
 self.fc = nn.Sequential(  
 nn.Linear(inp, squeeze),  
 nn.ReLU(inplace=True),  
 nn.Linear(squeeze, self.oup \* (4 if K2 else 2)),  
 h\_sigmoid()  
 )  
 self.spa = nn.Sequential(  
 nn.Conv2d(inp, 1, kernel\_size=1),  
 nn.BatchNorm2d(1),  
 ) if use\_spatial else None  
  
 def forward(self, x):  
 # 处理输入  
 x\_in = x[0] if isinstance(x, list) else x  
 x\_out = x[1] if isinstance(x, list) else x  
 b, c, h, w = x\_in.size() # 获取输入的尺寸  
 y = self.avg\_pool(x\_in).view(b, c) # 平均池化  
 y = self.fc(y).view(b, self.oup \* (4 if self.K2 else 2), 1, 1) # 全连接层输出  
  
 # 计算动态ReLU的输出  
 # 省略了具体的计算细节以简化代码  
 out = ... # 计算逻辑  
  
 if self.spa:  
 ys = self.spa(x\_in).view(b, -1)  
 ys = F.softmax(ys, dim=1).view(b, 1, h, w) \* h \* w  
 ys = F.hardtanh(ys, 0, 3, inplace=True) / 3  
 out = out \* ys # 加入空间注意力  
  
 return out  
  
# 动态卷积模块  
class DyDCNv2(nn.Module):  
 def \_\_init\_\_(self, in\_channels, out\_channels, stride=1, norm\_cfg=dict(type='GN', num\_groups=16, requires\_grad=True)):  
 super().\_\_init\_\_()  
 self.conv = ModulatedDeformConv2d(in\_channels, out\_channels, 3, stride=stride, padding=1)  
 self.norm = build\_norm\_layer(norm\_cfg, out\_channels)[1] if norm\_cfg else None  
  
 def forward(self, x, offset, mask):  
 x = self.conv(x.contiguous(), offset, mask) # 进行动态卷积  
 if self.norm:  
 x = self.norm(x) # 进行归一化  
 return x  
  
# DyHead模块  
class DyHeadBlock\_Prune(nn.Module):  
 def \_\_init\_\_(self, in\_channels, norm\_type='GN', zero\_init\_offset=True, act\_cfg=dict(type='HSigmoid', bias=3.0, divisor=6.0)):  
 super().\_\_init\_\_()  
 self.spatial\_conv\_high = DyDCNv2(in\_channels, in\_channels)  
 self.spatial\_conv\_mid = DyDCNv2(in\_channels, in\_channels)  
 self.spatial\_conv\_low = DyDCNv2(in\_channels, in\_channels, stride=2)  
 self.spatial\_conv\_offset = nn.Conv2d(in\_channels, 27, 3, padding=1) # 计算偏移和掩码  
 self.scale\_attn\_module = nn.Sequential(  
 nn.AdaptiveAvgPool2d(1), nn.Conv2d(in\_channels, 1, 1),  
 nn.ReLU(inplace=True), build\_activation\_layer(act\_cfg)  
 )  
 self.task\_attn\_module = DyReLU(in\_channels)  
 self.\_init\_weights()  
  
 def \_init\_weights(self):  
 for m in self.modules():  
 if isinstance(m, nn.Conv2d):  
 normal\_init(m, 0, 0.01) # 初始化卷积层  
  
 def forward(self, x, level):  
 offset\_and\_mask = self.spatial\_conv\_offset(x[level]) # 计算偏移和掩码  
 offset = offset\_and\_mask[:, :18, :, :] # 提取偏移  
 mask = offset\_and\_mask[:, 18:, :, :].sigmoid() # 提取掩码并应用sigmoid  
  
 mid\_feat = self.spatial\_conv\_mid(x[level], offset, mask) # 中间特征  
 sum\_feat = mid\_feat \* self.scale\_attn\_module(mid\_feat) # 加权特征  
  
 # 处理低层和高层特征  
 if level > 0:  
 low\_feat = self.spatial\_conv\_low(x[level - 1], offset, mask)  
 sum\_feat += low\_feat \* self.scale\_attn\_module(low\_feat)  
 if level < len(x) - 1:  
 high\_feat = F.interpolate(self.spatial\_conv\_high(x[level + 1], offset, mask), size=x[level].shape[-2:], mode='bilinear', align\_corners=True)  
 sum\_feat += high\_feat \* self.scale\_attn\_module(high\_feat)  
  
 return self.task\_attn\_module(sum\_feat / 3) # 返回最终特征  
```  
  
### 代码核心部分总结：  
1. \*\*激活函数\*\*：实现了多种激活函数，包括Swish、Hard Swish和Hard Sigmoid，适用于不同的网络需求。  
2. \*\*动态ReLU\*\*：通过自适应的方式调整激活函数的输出，增强了模型的表达能力。  
3. \*\*动态卷积\*\*：实现了一个支持偏移和掩码的动态卷积模块，能够根据输入特征动态调整卷积操作。  
4. \*\*DyHead模块\*\*：整合了动态卷积和注意力机制，能够有效地处理多层特征，提升模型的性能。  
  
这些模块的组合使得模型在处理复杂任务时能够更加灵活和高效。```

这个文件 `dyhead\_prune.py` 实现了一个动态头部（Dynamic Head）模块，主要用于计算机视觉任务中的特征提取和处理。代码中使用了 PyTorch 框架，并且定义了一些自定义的激活函数和卷积层，结合了动态卷积和注意力机制。以下是对代码的逐步分析。  
  
首先，导入了必要的库，包括 PyTorch 的核心模块和一些特定的操作模块，如 `ModulatedDeformConv2d`。接着定义了一个 `\_make\_divisible` 函数，用于确保某个值是可被指定的除数整除的，并且在必要时会调整该值以满足最小值的要求。  
  
接下来，定义了几个自定义的激活函数类，包括 `swish`、`h\_swish` 和 `h\_sigmoid`。这些激活函数在深度学习中常用于提升模型的非线性表达能力。`h\_swish` 和 `h\_sigmoid` 特别是基于 ReLU6 的变体，适用于一些特定的网络架构。  
  
`DyReLU` 类是一个动态激活函数模块，具有可调的参数。它根据输入特征的统计信息（通过全局平均池化获得）来动态调整激活函数的输出。该模块的设计允许在训练过程中根据特征的分布自适应地调整激活值，从而增强模型的表达能力。  
  
`DyDCNv2` 类实现了带有归一化层的可调变形卷积（Modulated Deformable Convolution），这是一个灵活的卷积操作，能够根据输入特征动态调整卷积核的位置和形状。该类支持多种归一化方法，具体取决于传入的配置。  
  
`DyHeadBlock\_Prune` 类是整个模块的核心部分，结合了多个注意力机制。它使用了三个不同层次的卷积操作来处理输入特征，并通过计算偏移量和掩码来动态调整卷积操作。这个模块的设计使得它能够在不同的特征层次上进行信息融合，从而提升模型的性能。  
  
在 `DyHeadBlock\_Prune` 的构造函数中，初始化了多个卷积层和注意力模块，并且定义了权重初始化的方法。`forward` 方法则实现了前向传播过程，计算特征的偏移量和掩码，并通过不同层次的卷积操作融合特征。  
  
总体来说，这个文件实现了一个复杂的动态特征提取模块，结合了动态卷积、注意力机制和自适应激活函数，旨在提升计算机视觉任务中的特征表示能力。

```以下是代码中最核心的部分，并附上详细的中文注释：  
  
```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，模块不存在），则执行以下代码  
 # 导入 TransNext\_native 中的所有内容  
 from ultralytics.nn.backbone.TransNeXt.TransNext\_native import \*  
 pass  
```  
  
### 注释说明：  
1. \*\*try 块\*\*：尝试执行一段代码，如果代码中发生异常（如模块未找到），则会跳转到 except 块。  
2. \*\*import swattention\*\*：导入名为 `swattention` 的模块，可能是用于某种特定的注意力机制。  
3. \*\*from ... import \*\*\*：从 `TransNext\_cuda` 模块中导入所有内容，这通常是一个 CUDA 加速的实现，适用于支持 GPU 的环境。  
4. \*\*except ImportError as e\*\*：捕获导入时发生的 `ImportError` 异常，`as e` 语句可以将异常信息存储在变量 `e` 中（虽然在这里并没有使用）。  
5. \*\*from ... import \*\*\*：如果前面的导入失败，则从 `TransNext\_native` 模块中导入所有内容，这通常是一个不依赖于 CUDA 的实现，适用于 CPU 环境或没有 GPU 的情况。  
6. \*\*pass\*\*：在 except 块中使用 `pass` 表示不执行任何操作，继续执行后面的代码。```

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。代码的结构使用了 `try...except` 语句，这是一种异常处理机制。  
  
首先，程序尝试导入 `swattention` 模块和 `TransNext\_cuda` 中的所有内容。如果这两个模块能够成功导入，程序将继续执行这些导入的内容。`TransNext\_cuda` 可能是一个专门为 CUDA（即用于并行计算的 NVIDIA 图形处理单元）优化的版本，通常用于加速深度学习模型的训练和推理过程。  
  
如果在导入过程中发生 `ImportError` 异常，程序将进入 `except` 块。在这里，它将尝试导入 `TransNext\_native` 模块，而不是 CUDA 版本。这表明 `TransNext\_native` 是一个不依赖于 CUDA 的实现，可能适用于不支持 CUDA 的环境。  
  
总的来说，这段代码的设计目的是为了确保在不同的环境中都能找到合适的 `TransNeXt` 实现，从而提高程序的兼容性和灵活性。

```以下是代码中最核心的部分，并附上详细的中文注释：  
  
```python  
from ultralytics.engine.predictor import BasePredictor  
from ultralytics.engine.results import Results  
from ultralytics.utils import ops  
  
class DetectionPredictor(BasePredictor):  
 """  
 DetectionPredictor类用于基于检测模型进行预测。  
 该类继承自BasePredictor类，主要实现了后处理预测结果的功能。  
 """  
  
 def postprocess(self, preds, img, orig\_imgs):  
 """  
 对预测结果进行后处理，并返回一个Results对象的列表。  
  
 参数:  
 preds: 模型的原始预测结果  
 img: 输入图像  
 orig\_imgs: 原始输入图像（可能是torch.Tensor或numpy数组）  
  
 返回:  
 results: 包含后处理结果的Results对象列表  
 """  
 # 使用非极大值抑制（NMS）来过滤预测框  
 preds = ops.non\_max\_suppression(  
 preds,  
 self.args.conf, # 置信度阈值  
 self.args.iou, # IOU阈值  
 agnostic=self.args.agnostic\_nms, # 是否进行类别无关的NMS  
 max\_det=self.args.max\_det, # 最大检测框数量  
 classes=self.args.classes, # 需要检测的类别  
 )  
  
 # 如果输入的原始图像不是列表，说明是torch.Tensor格式，需转换为numpy数组  
 if not isinstance(orig\_imgs, list):  
 orig\_imgs = ops.convert\_torch2numpy\_batch(orig\_imgs)  
  
 results = [] # 存储后处理结果的列表  
 for i, pred in enumerate(preds):  
 orig\_img = orig\_imgs[i] # 获取对应的原始图像  
 # 将预测框的坐标从当前图像的尺寸缩放到原始图像的尺寸  
 pred[:, :4] = ops.scale\_boxes(img.shape[2:], pred[:, :4], orig\_img.shape)  
 img\_path = self.batch[0][i] # 获取当前图像的路径  
 # 创建Results对象并添加到结果列表中  
 results.append(Results(orig\_img, path=img\_path, names=self.model.names, boxes=pred))  
   
 return results # 返回后处理的结果列表  
```  
  
### 代码核心部分解析：  
1. \*\*类定义\*\*：`DetectionPredictor`类用于处理基于YOLO模型的目标检测预测，继承自`BasePredictor`。  
2. \*\*后处理方法\*\*：`postprocess`方法对模型的预测结果进行后处理，包括非极大值抑制和坐标缩放。  
3. \*\*非极大值抑制\*\*：通过`ops.non\_max\_suppression`函数来过滤掉重叠的预测框，保留最有可能的检测结果。  
4. \*\*图像格式转换\*\*：确保原始图像为numpy数组格式，以便后续处理。  
5. \*\*结果存储\*\*：将处理后的结果封装成`Results`对象，包含原始图像、图像路径、类别名称和预测框信息，最终返回结果列表。```

这个程序文件 `predict.py` 是一个用于目标检测的预测类，基于 Ultralytics YOLO 模型。它扩展了 `BasePredictor` 类，提供了一个专门用于处理检测模型预测的功能。  
  
在文件的开头，首先导入了必要的模块，包括 `BasePredictor`、`Results` 和一些实用的操作函数 `ops`。接着定义了 `DetectionPredictor` 类，这个类主要负责处理模型的预测结果。  
  
类中包含一个方法 `postprocess`，这个方法的作用是对模型的预测结果进行后处理，并返回一个包含结果的列表。具体来说，`postprocess` 方法接收三个参数：`preds`（模型的预测结果）、`img`（输入图像）和 `orig\_imgs`（原始图像）。在方法内部，首先使用非极大值抑制（NMS）来过滤掉冗余的检测框，确保每个目标只保留一个最佳的检测框。这个过程是通过调用 `ops.non\_max\_suppression` 实现的，参数包括置信度阈值、IoU 阈值、是否使用类别无关的 NMS、最大检测框数量以及要检测的类别。  
  
接下来，方法检查 `orig\_imgs` 是否为列表。如果不是，说明输入图像是一个 PyTorch 张量，此时需要将其转换为 NumPy 数组。然后，程序会遍历每个预测结果，使用 `ops.scale\_boxes` 方法将检测框的坐标从缩放后的图像尺寸转换回原始图像的尺寸。每个结果都会被封装成一个 `Results` 对象，包含原始图像、图像路径、模型名称和检测框信息。  
  
最后，所有的结果对象会被收集到一个列表中并返回。这段代码展示了如何将模型的输出进行有效的后处理，以便于后续的结果分析和可视化。

```以下是经过简化和注释的核心代码部分，保留了EfficientViT模型的主要结构和功能。  
  
```python  
import torch  
import torch.nn as nn  
import torch.nn.functional as F  
import itertools  
  
# 定义卷积层和批归一化的组合类  
class Conv2d\_BN(torch.nn.Sequential):  
 def \_\_init\_\_(self, in\_channels, out\_channels, kernel\_size=1, stride=1, padding=0, dilation=1, groups=1, bn\_weight\_init=1):  
 super().\_\_init\_\_()  
 # 添加卷积层  
 self.add\_module('conv', torch.nn.Conv2d(in\_channels, out\_channels, kernel\_size, stride, padding, dilation, groups, bias=False))  
 # 添加批归一化层  
 self.add\_module('bn', torch.nn.BatchNorm2d(out\_channels))  
 # 初始化批归一化的权重  
 torch.nn.init.constant\_(self.bn.weight, bn\_weight\_init)  
 torch.nn.init.constant\_(self.bn.bias, 0)  
  
 @torch.no\_grad()  
 def switch\_to\_deploy(self):  
 # 将训练模式下的卷积和批归一化合并为一个卷积层  
 conv, bn = self.\_modules.values()  
 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  
 return torch.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, bias=True).weight.data.copy\_(w), b  
  
# 定义高效ViT块  
class EfficientViTBlock(torch.nn.Module):  
 def \_\_init\_\_(self, in\_channels, key\_dim, num\_heads=8, resolution=14):  
 super().\_\_init\_\_()  
 # 定义残差卷积层  
 self.dw0 = nn.Sequential(Conv2d\_BN(in\_channels, in\_channels, kernel\_size=3, stride=1, padding=1, groups=in\_channels))  
 # 定义前馈网络  
 self.ffn0 = nn.Sequential(Conv2d\_BN(in\_channels, in\_channels \* 2), nn.ReLU(), Conv2d\_BN(in\_channels \* 2, in\_channels))  
  
 # 定义局部窗口注意力机制  
 self.mixer = LocalWindowAttention(in\_channels, key\_dim, num\_heads, resolution=resolution)  
  
 self.dw1 = nn.Sequential(Conv2d\_BN(in\_channels, in\_channels, kernel\_size=3, stride=1, padding=1, groups=in\_channels))  
 self.ffn1 = nn.Sequential(Conv2d\_BN(in\_channels, in\_channels \* 2), nn.ReLU(), Conv2d\_BN(in\_channels \* 2, in\_channels))  
  
 def forward(self, x):  
 # 前向传播  
 return self.ffn1(self.dw1(self.mixer(self.ffn0(self.dw0(x)))))  
  
  
class EfficientViT(torch.nn.Module):  
 def \_\_init\_\_(self, img\_size=400, patch\_size=16, embed\_dim=[64, 128, 192], depth=[1, 2, 3], num\_heads=[4, 4, 4]):  
 super().\_\_init\_\_()  
 # 初始化图像嵌入层  
 self.patch\_embed = nn.Sequential(  
 Conv2d\_BN(3, embed\_dim[0] // 8, kernel\_size=3, stride=2, padding=1),  
 nn.ReLU(),  
 Conv2d\_BN(embed\_dim[0] // 8, embed\_dim[0] // 4, kernel\_size=3, stride=2, padding=1),  
 nn.ReLU(),  
 Conv2d\_BN(embed\_dim[0] // 4, embed\_dim[0] // 2, kernel\_size=3, stride=2, padding=1),  
 nn.ReLU(),  
 Conv2d\_BN(embed\_dim[0] // 2, embed\_dim[0], kernel\_size=3, stride=1, padding=1)  
 )  
  
 # 构建多个高效ViT块  
 self.blocks = nn.ModuleList()  
 for i in range(len(depth)):  
 for \_ in range(depth[i]):  
 self.blocks.append(EfficientViTBlock(embed\_dim[i], key\_dim=16, num\_heads=num\_heads[i]))  
  
 def forward(self, x):  
 # 前向传播  
 x = self.patch\_embed(x)  
 for block in self.blocks:  
 x = block(x)  
 return x  
  
# 示例模型创建  
if \_\_name\_\_ == '\_\_main\_\_':  
 model = EfficientViT()  
 inputs = torch.randn((1, 3, 640, 640)) # 创建一个随机输入  
 res = model(inputs) # 通过模型进行前向传播  
 print(res.size()) # 输出结果的尺寸  
```  
  
### 代码注释说明：  
1. \*\*Conv2d\_BN类\*\*：这是一个自定义的卷积层，包含卷积操作和批归一化，方便在训练和推理阶段进行切换。  
2. \*\*EfficientViTBlock类\*\*：定义了高效ViT的基本构建块，包含卷积层、前馈网络和局部窗口注意力机制。  
3. \*\*EfficientViT类\*\*：整体模型结构，包含图像嵌入层和多个高效ViT块的堆叠。  
4. \*\*前向传播\*\*：通过模型的输入数据，逐层进行处理，最终输出特征图。  
  
该代码的核心部分是EfficientViT的结构，主要关注卷积层、前馈网络和注意力机制的实现。```

这个程序文件 `efficientViT.py` 实现了一个高效的视觉变换器（Efficient Vision Transformer）模型架构，主要用于图像处理任务。代码中包含了多个类和函数，构成了整个模型的结构和功能。  
  
首先，程序导入了必要的库，包括 PyTorch 及其子模块，用于构建神经网络和进行张量运算。接着，定义了一些常量和类，`\_\_all\_\_` 列表中列出了可供外部调用的模型类型。  
  
`Conv2d\_BN` 类是一个组合模块，包含卷积层和批归一化层。它的构造函数初始化了卷积层，并为批归一化层的权重和偏置设置了初始值。该类还提供了一个 `switch\_to\_deploy` 方法，用于在推理阶段将卷积和批归一化层融合，以提高推理效率。  
  
`replace\_batchnorm` 函数用于遍历网络中的所有子模块，将批归一化层替换为恒等映射，从而在推理时减少计算开销。  
  
`PatchMerging` 类实现了将输入特征图进行合并的功能，通过多个卷积层和激活函数处理输入数据，并应用了 Squeeze-and-Excitation 模块来增强特征表达。  
  
`Residual` 类实现了残差连接的功能，可以在训练过程中随机丢弃一些输出，以增强模型的鲁棒性。  
  
`FFN` 类实现了前馈神经网络模块，包含两个卷积层和一个激活函数。  
  
`CascadedGroupAttention` 和 `LocalWindowAttention` 类实现了局部窗口注意力机制和级联组注意力机制，这些机制能够有效地捕捉输入特征之间的关系。  
  
`EfficientViTBlock` 类是构建高效视觉变换器的基本模块，包含多个卷积层、前馈网络和注意力机制。该模块的设计使得模型能够灵活地处理不同的输入特征。  
  
`EfficientViT` 类是整个模型的核心，负责将输入图像分割成小块（patch），并通过多个块进行处理。它的构造函数中定义了不同的阶段，每个阶段包含多个 `EfficientViTBlock`，并在必要时进行下采样。  
  
最后，定义了一些不同配置的模型参数（如 `EfficientViT\_m0` 到 `EfficientViT\_m5`），并提供了相应的函数来创建这些模型实例。这些函数允许加载预训练权重，并在需要时进行批归一化的替换。  
  
在 `\_\_main\_\_` 部分，程序实例化了一个模型并对随机生成的输入进行了前向传播，输出了每个阶段的特征图大小。这部分代码展示了如何使用定义的模型进行实际的推理操作。  
  
整体来看，这个程序实现了一个灵活且高效的视觉变换器模型，适用于各种下游视觉任务，具备较好的性能和可扩展性。

### 整体功能和构架概括  
  
该项目包含多个模块，主要用于计算机视觉任务，特别是目标检测和图像处理。每个文件实现了不同的功能模块，结合了深度学习中的多种技术，如动态卷积、视觉变换器、注意力机制等。整体架构设计旨在提高模型的性能和灵活性，适应不同的应用场景。  
  
1. \*\*dyhead\_prune.py\*\*：实现了动态头部模块，结合动态卷积和注意力机制，增强特征提取能力。  
2. \*\*TransNext.py\*\*：负责导入与 `TransNeXt` 相关的模块，确保在不同环境中能够使用合适的实现。  
3. \*\*predict.py\*\*：处理目标检测模型的预测结果，进行后处理并返回可用的检测结果。  
4. \*\*efficientViT.py\*\*：实现高效的视觉变换器模型，结合多种注意力机制和特征处理模块，适用于图像分类和其他视觉任务。  
  
### 文件功能整理表  
  
| 文件名 | 功能描述 |  
|-----------------------|-----------------------------------------------------------------------|  
| dyhead\_prune.py | 实现动态头部模块，结合动态卷积和注意力机制，增强特征提取能力。 |  
| TransNext.py | 导入与 `TransNeXt` 相关的模块，确保在不同环境中使用合适的实现。 |  
| predict.py | 处理目标检测模型的预测结果，进行后处理并返回可用的检测结果。 |  
| efficientViT.py | 实现高效的视觉变换器模型，结合多种注意力机制和特征处理模块，适用于图像分类和其他视觉任务。 |  
  
这个表格总结了每个文件的主要功能，便于理解项目的整体结构和各个模块之间的关系。

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。