# 改进yolo11-AKConv等200+全套创新点大全：农业病害检测系统源码＆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  
  
随着全球人口的持续增长和城市化进程的加快，农业生产面临着越来越大的压力。病害是影响农作物产量和质量的重要因素，及时准确地检测和识别病害对于保障农业生产、提高农作物产量具有重要意义。近年来，计算机视觉和深度学习技术的快速发展为农业病害检测提供了新的解决方案。其中，YOLO（You Only Look Once）系列模型因其高效的实时目标检测能力，逐渐成为农业病害检测领域的研究热点。  
  
本研究旨在基于改进的YOLOv11模型，构建一个高效的农业病害检测系统。该系统将利用一个包含5500张图像的数据集，涵盖12种不同类型的农作物病害，包括豆类、草莓和番茄等常见作物的病害。这些病害的种类多样，涉及到不同的生物特征和生长环境，因此，对其进行准确的检测和分类将有助于农民及时采取相应的防治措施，降低病害对作物的影响。  
  
在现有的农业病害检测研究中，尽管已有多种深度学习模型被提出，但仍存在检测精度不足、实时性差等问题。通过对YOLOv11模型的改进，我们希望能够提升模型在复杂背景下的检测能力，并增强其对不同病害的识别准确性。此外，利用丰富的标注数据集，我们将进行模型的训练和验证，以确保其在实际应用中的有效性和可靠性。  
  
综上所述，本研究不仅为农业病害检测提供了一种新的技术路径，也为农民提供了更为高效的病害管理工具，具有重要的理论价值和实际应用意义。通过实现高效、准确的病害检测，我们期待能够为农业可持续发展贡献一份力量。

## 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“Detecting diseases”，旨在为改进YOLOv11的农业病害检测系统提供强有力的支持。该数据集包含12个不同的类别，涵盖了多种农作物的病害，具体包括：豆类的角斑病、锈病，以及草莓的多种病害，如角斑病、果腐病、花朵枯萎病、灰霉病、叶斑病和粉霉病。此外，数据集中还包括西红柿的病害，如枯萎病、叶霉病和蜘蛛螨的影响。这些类别的选择不仅反映了当前农业生产中常见的病害类型，也为研究人员提供了一个全面的基础，以便于进行深入的分析和模型训练。  
  
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具有代表性和多样性。这种多样性对于训练深度学习模型至关重要，因为它能够帮助模型学习到不同病害在不同生长阶段、不同环境条件下的表现特征。通过使用“Detecting diseases”数据集，研究人员能够更好地理解和识别农业病害，从而提高作物的产量和质量。  
  
在数据集的构建过程中，特别注重了图像的质量和标注的准确性，确保每个图像都能清晰地展示出病害的特征。这样的高质量数据集不仅为YOLOv11的训练提供了坚实的基础，也为后续的农业病害检测研究奠定了良好的数据支持。通过对这些数据的深入分析，研究人员希望能够开发出更为精准和高效的病害检测系统，从而为现代农业的可持续发展贡献力量。

## 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

```以下是对代码的核心部分进行分析和详细注释的结果：  
  
```python  
import torch  
import torch.nn as nn  
  
class KANConvNDLayer(nn.Module):  
 def \_\_init\_\_(self, conv\_class, norm\_class, input\_dim, output\_dim, spline\_order, kernel\_size,  
 groups=1, padding=0, stride=1, dilation=1,  
 ndim: int = 2, grid\_size=5, base\_activation=nn.GELU, grid\_range=[-1, 1], dropout=0.0):  
 super(KANConvNDLayer, self).\_\_init\_\_()  
   
 # 初始化参数  
 self.inputdim = input\_dim # 输入维度  
 self.outdim = output\_dim # 输出维度  
 self.spline\_order = spline\_order # 样条阶数  
 self.kernel\_size = kernel\_size # 卷积核大小  
 self.padding = padding # 填充  
 self.stride = stride # 步幅  
 self.dilation = dilation # 膨胀  
 self.groups = groups # 分组数  
 self.ndim = ndim # 维度  
 self.grid\_size = grid\_size # 网格大小  
 self.base\_activation = base\_activation() # 基础激活函数  
 self.grid\_range = grid\_range # 网格范围  
  
 # Dropout层的初始化  
 self.dropout = None  
 if dropout > 0:  
 if ndim == 1:  
 self.dropout = nn.Dropout1d(p=dropout)  
 elif ndim == 2:  
 self.dropout = nn.Dropout2d(p=dropout)  
 elif ndim == 3:  
 self.dropout = nn.Dropout3d(p=dropout)  
  
 # 参数有效性检查  
 if groups <= 0:  
 raise ValueError('groups must be a positive integer')  
 if input\_dim % groups != 0:  
 raise ValueError('input\_dim must be divisible by groups')  
 if output\_dim % groups != 0:  
 raise ValueError('output\_dim must be divisible by groups')  
  
 # 基础卷积层的初始化  
 self.base\_conv = nn.ModuleList([conv\_class(input\_dim // groups,  
 output\_dim // groups,  
 kernel\_size,  
 stride,  
 padding,  
 dilation,  
 groups=1,  
 bias=False) for \_ in range(groups)])  
  
 # 样条卷积层的初始化  
 self.spline\_conv = nn.ModuleList([conv\_class((grid\_size + spline\_order) \* input\_dim // groups,  
 output\_dim // groups,  
 kernel\_size,  
 stride,  
 padding,  
 dilation,  
 groups=1,  
 bias=False) for \_ in range(groups)])  
  
 # 归一化层的初始化  
 self.layer\_norm = nn.ModuleList([norm\_class(output\_dim // groups) for \_ in range(groups)])  
  
 # PReLU激活函数的初始化  
 self.prelus = nn.ModuleList([nn.PReLU() for \_ in range(groups)])  
  
 # 初始化网格  
 h = (self.grid\_range[1] - self.grid\_range[0]) / grid\_size  
 self.grid = torch.linspace(  
 self.grid\_range[0] - h \* spline\_order,  
 self.grid\_range[1] + h \* spline\_order,  
 grid\_size + 2 \* spline\_order + 1,  
 dtype=torch.float32  
 )  
  
 # 使用Kaiming均匀分布初始化卷积层权重  
 for conv\_layer in self.base\_conv:  
 nn.init.kaiming\_uniform\_(conv\_layer.weight, nonlinearity='linear')  
  
 for conv\_layer in self.spline\_conv:  
 nn.init.kaiming\_uniform\_(conv\_layer.weight, nonlinearity='linear')  
  
 def forward\_kan(self, x, group\_index):  
 # 对输入应用基础激活函数并进行线性变换  
 base\_output = self.base\_conv[group\_index](self.base\_activation(x))  
  
 x\_uns = x.unsqueeze(-1) # 扩展维度以进行样条操作  
 target = x.shape[1:] + self.grid.shape # 计算目标形状  
 grid = self.grid.view(\*list([1 for \_ in range(self.ndim + 1)] + [-1, ])).expand(target).contiguous().to(x.device)  
  
 # 计算样条基  
 bases = ((x\_uns >= grid[..., :-1]) & (x\_uns < grid[..., 1:])).to(x.dtype)  
  
 # 计算多阶样条基  
 for k in range(1, self.spline\_order + 1):  
 left\_intervals = grid[..., :-(k + 1)]  
 right\_intervals = grid[..., k:-1]  
 delta = torch.where(right\_intervals == left\_intervals, torch.ones\_like(right\_intervals),  
 right\_intervals - left\_intervals)  
 bases = ((x\_uns - left\_intervals) / delta \* bases[..., :-1]) + \  
 ((grid[..., k + 1:] - x\_uns) / (grid[..., k + 1:] - grid[..., 1:(-k)]) \* bases[..., 1:])  
 bases = bases.contiguous()  
 bases = bases.moveaxis(-1, 2).flatten(1, 2) # 调整基的形状以适应卷积输入  
 spline\_output = self.spline\_conv[group\_index](bases) # 通过样条卷积层  
  
 # 归一化和激活  
 x = self.prelus[group\_index](self.layer\_norm[group\_index](base\_output + spline\_output))  
  
 # 应用Dropout  
 if self.dropout is not None:  
 x = self.dropout(x)  
  
 return x  
  
 def forward(self, x):  
 # 将输入按组分割  
 split\_x = torch.split(x, self.inputdim // self.groups, dim=1)  
 output = []  
 for group\_ind, \_x in enumerate(split\_x):  
 y = self.forward\_kan(\_x.clone(), group\_ind) # 对每组进行前向传播  
 output.append(y.clone())  
 y = torch.cat(output, dim=1) # 合并输出  
 return y  
```  
  
### 代码核心部分分析  
1. \*\*KANConvNDLayer类\*\*: 这是一个通用的N维卷积层，支持1D、2D和3D卷积。它使用样条基函数进行卷积操作，并结合基础卷积和归一化层。  
2. \*\*初始化方法\*\*: 在构造函数中，初始化了卷积层、归一化层、激活函数和Dropout层，并进行了参数有效性检查。  
3. \*\*forward\_kan方法\*\*: 该方法实现了卷积的前向传播，包括基础卷积、样条基计算和最终的激活。它处理了输入的扩展和样条基的计算。  
4. \*\*forward方法\*\*: 该方法将输入按组分割，并对每组调用`forward\_kan`进行处理，最后将所有组的输出合并。  
  
### 其他类  
`KANConv1DLayer`、`KANConv2DLayer`和`KANConv3DLayer`是`KANConvNDLayer`的具体实现，分别用于1D、2D和3D卷积，主要是通过调用父类的构造函数来实现。```

这个文件定义了一个名为 `KANConvNDLayer` 的神经网络层，主要用于实现一种新的卷积操作，结合了基础卷积和样条插值。这个层可以处理多维数据（如1D、2D和3D），并且通过继承机制，分别实现了1D、2D和3D的卷积层（`KANConv1DLayer`、`KANConv2DLayer`、`KANConv3DLayer`）。  
  
在 `KANConvNDLayer` 的构造函数中，初始化了一些参数，包括输入和输出维度、卷积核大小、样条阶数、分组数、填充、步幅、扩张、网格大小、激活函数、网格范围和丢弃率。构造函数中还会检查一些参数的有效性，比如分组数必须为正整数，并且输入和输出维度必须能够被分组数整除。  
  
该类的核心部分是 `forward\_kan` 方法，它实现了前向传播的逻辑。首先，它对输入应用基础激活函数，然后通过基础卷积层进行线性变换。接着，计算样条基函数，使用输入值和预定义的网格进行插值。样条基函数的计算涉及到多个阶数的插值，通过一系列的条件判断和数学运算，生成最终的样条输出。然后，将基础卷积的输出和样条卷积的输出相加，并通过层归一化和激活函数进行处理，最后应用丢弃层（如果设置了丢弃率）。  
  
`forward` 方法则负责将输入数据分成多个组，并对每个组调用 `forward\_kan` 方法，最后将所有组的输出拼接在一起。  
  
`KANConv3DLayer`、`KANConv2DLayer` 和 `KANConv1DLayer` 这三个类分别继承自 `KANConvNDLayer`，并在构造函数中指定了相应的卷积和归一化类（`nn.Conv3d`、`nn.Conv2d`、`nn.Conv1d` 和 `nn.InstanceNorm3d`、`nn.InstanceNorm2d`、`nn.InstanceNorm1d`），从而实现对不同维度数据的支持。  
  
整体来看，这个文件实现了一种灵活且强大的卷积层，能够处理多维数据，并结合了样条插值的优点，适用于需要高效特征提取的深度学习任务。

```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  
  
# 定义一个自定义的批量归一化层，带有可学习的参数  
class RepBN(nn.Module):  
 def \_\_init\_\_(self, channels):  
 super(RepBN, self).\_\_init\_\_()  
 # alpha是一个可学习的参数，初始化为1  
 self.alpha = nn.Parameter(torch.ones(1))  
 # 使用1D批量归一化  
 self.bn = nn.BatchNorm1d(channels)  
  
 def forward(self, x):  
 # 将输入的维度进行转置，以适应BatchNorm1d的输入格式  
 x = x.transpose(1, 2)  
 # 进行批量归一化，并加上alpha乘以输入x  
 x = self.bn(x) + self.alpha \* x  
 # 再次转置回原来的维度  
 x = x.transpose(1, 2)  
 return x  
  
# 定义一个线性归一化层，支持动态调整归一化方式  
class LinearNorm(nn.Module):  
 def \_\_init\_\_(self, dim, norm1, norm2, warm=0, step=300000, r0=1.0):  
 super(LinearNorm, self).\_\_init\_\_()  
 # 注册缓冲区，用于存储预热阶段的步数  
 self.register\_buffer('warm', torch.tensor(warm))  
 # 注册缓冲区，用于存储当前迭代步数  
 self.register\_buffer('iter', torch.tensor(step))  
 # 注册缓冲区，用于存储总步数  
 self.register\_buffer('total\_step', torch.tensor(step))  
 self.r0 = r0 # 初始权重  
 # norm1和norm2是两个不同的归一化方法  
 self.norm1 = norm1(dim)  
 self.norm2 = norm2(dim)  
  
 def forward(self, x):  
 # 如果模型处于训练状态  
 if self.training:  
 # 如果还有预热步数  
 if self.warm > 0:  
 # 减少预热步数  
 self.warm.copy\_(self.warm - 1)  
 # 使用第一种归一化方法  
 x = self.norm1(x)  
 else:  
 # 计算当前的lambda值，控制两种归一化方法的混合比例  
 lamda = self.r0 \* self.iter / self.total\_step  
 if self.iter > 0:  
 # 减少当前迭代步数  
 self.iter.copy\_(self.iter - 1)  
 # 使用第一种归一化方法  
 x1 = self.norm1(x)  
 # 使用第二种归一化方法  
 x2 = self.norm2(x)  
 # 按照lambda值混合两种归一化的结果  
 x = lamda \* x1 + (1 - lamda) \* x2  
 else:  
 # 如果模型不在训练状态，直接使用第二种归一化方法  
 x = self.norm2(x)  
 return x  
```  
  
### 代码说明：  
1. \*\*RepBN类\*\*：自定义的批量归一化层，除了进行标准的批量归一化外，还引入了一个可学习的参数`alpha`，使得模型能够学习到对输入的加权调整。  
2. \*\*LinearNorm类\*\*：一个线性归一化层，支持在训练过程中根据预热阶段和迭代次数动态调整归一化方式。它可以在训练初期使用一种归一化方法，随着训练的进行逐渐过渡到另一种归一化方法。```

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`，它们都是基于 PyTorch 框架构建的。  
  
首先，`RepBN` 类是一个自定义的批量归一化（Batch Normalization）模块。它的构造函数接收一个参数 `channels`，表示输入数据的通道数。在构造函数中，定义了一个可学习的参数 `alpha`，初始值为 1，并且创建了一个标准的 `BatchNorm1d` 层。`forward` 方法实现了前向传播的逻辑，首先将输入 `x` 的维度进行转置，以适应 `BatchNorm1d` 的输入格式，然后对转置后的数据进行批量归一化处理，并加上 `alpha` 乘以原始输入 `x`。最后，再次对结果进行转置，返回处理后的数据。  
  
接下来是 `LinearNorm` 类，它实现了一种线性归一化的机制。构造函数接收多个参数，包括 `dim`（维度）、`norm1` 和 `norm2`（两个归一化函数）、`warm`（预热步数）、`step`（迭代步数）和 `r0`（初始比例）。在构造函数中，使用 `register\_buffer` 方法注册了一些缓冲区，用于存储预热步数和迭代步数等信息。`forward` 方法中，如果模型处于训练状态且 `warm` 大于 0，则执行 `norm1` 进行归一化，并减少 `warm` 的值；否则，计算一个动态的比例 `lamda`，该比例随着迭代次数的减少而变化。接着，分别对输入 `x` 应用 `norm1` 和 `norm2`，并根据 `lamda` 的值线性组合这两个归一化的结果。如果模型不在训练状态，则直接使用 `norm2` 对输入进行归一化处理。  
  
总体而言，这个文件实现了两个模块，分别用于不同的归一化操作，具有一定的灵活性和可扩展性，适用于深度学习模型中的特定需求。

```以下是简化后的代码，保留了核心部分，并添加了详细的中文注释：  
  
```python  
import torch  
import torch.nn as nn  
import torch.nn.functional as F  
from timm.models.layers import DropPath, to\_2tuple, trunc\_normal\_  
  
class Mlp(nn.Module):  
 """ 多层感知机 (MLP) 模块 """  
  
 def \_\_init\_\_(self, in\_features, hidden\_features=None, out\_features=None, act\_layer=nn.GELU, drop=0.):  
 super().\_\_init\_\_()  
 out\_features = out\_features or in\_features # 输出特征数  
 hidden\_features = hidden\_features or in\_features # 隐藏层特征数  
 self.fc1 = nn.Linear(in\_features, hidden\_features) # 第一层线性变换  
 self.act = act\_layer() # 激活函数  
 self.fc2 = nn.Linear(hidden\_features, out\_features) # 第二层线性变换  
 self.drop = nn.Dropout(drop) # Dropout层  
  
 def forward(self, x):  
 """ 前向传播 """  
 x = self.fc1(x) # 线性变换  
 x = self.act(x) # 激活  
 x = self.drop(x) # Dropout  
 x = self.fc2(x) # 线性变换  
 x = self.drop(x) # Dropout  
 return x  
  
  
class WindowAttention(nn.Module):  
 """ 基于窗口的多头自注意力 (W-MSA) 模块 """  
  
 def \_\_init\_\_(self, dim, window\_size, num\_heads, qkv\_bias=True, attn\_drop=0., proj\_drop=0.):  
 super().\_\_init\_\_()  
 self.dim = dim # 输入通道数  
 self.window\_size = window\_size # 窗口大小  
 self.num\_heads = num\_heads # 注意力头数  
 head\_dim = dim // num\_heads # 每个头的维度  
 self.scale = head\_dim \*\* -0.5 # 缩放因子  
  
 # 定义相对位置偏置参数表  
 self.relative\_position\_bias\_table = nn.Parameter(  
 torch.zeros((2 \* window\_size[0] - 1) \* (2 \* window\_size[1] - 1), num\_heads))  
  
 # 计算相对位置索引  
 coords\_h = torch.arange(self.window\_size[0])  
 coords\_w = torch.arange(self.window\_size[1])  
 coords = torch.stack(torch.meshgrid([coords\_h, coords\_w])) # 生成坐标网格  
 coords\_flatten = torch.flatten(coords, 1) # 展平坐标  
 relative\_coords = coords\_flatten[:, :, None] - coords\_flatten[:, None, :] # 计算相对坐标  
 relative\_coords = relative\_coords.permute(1, 2, 0).contiguous() # 调整维度  
 relative\_coords[:, :, 0] += self.window\_size[0] - 1 # 位置偏移  
 relative\_coords[:, :, 1] += self.window\_size[1] - 1  
 relative\_coords[:, :, 0] \*= 2 \* self.window\_size[1] - 1  
 self.relative\_position\_index = relative\_coords.sum(-1) # 计算相对位置索引  
  
 self.qkv = nn.Linear(dim, dim \* 3, bias=qkv\_bias) # Q, K, V的线性变换  
 self.attn\_drop = nn.Dropout(attn\_drop) # 注意力的Dropout  
 self.proj = nn.Linear(dim, dim) # 输出的线性变换  
 self.proj\_drop = nn.Dropout(proj\_drop) # 输出的Dropout  
 trunc\_normal\_(self.relative\_position\_bias\_table, std=.02) # 初始化相对位置偏置  
 self.softmax = nn.Softmax(dim=-1) # Softmax层  
  
 def forward(self, x, mask=None):  
 """ 前向传播 """  
 B\_, N, C = x.shape # 获取输入的形状  
 qkv = self.qkv(x).reshape(B\_, N, 3, self.num\_heads, C // self.num\_heads).permute(2, 0, 3, 1, 4) # 计算Q, K, V  
 q, k, v = qkv[0], qkv[1], qkv[2] # 分离Q, K, V  
  
 q = q \* self.scale # 缩放Q  
 attn = (q @ k.transpose(-2, -1)) # 计算注意力  
  
 # 添加相对位置偏置  
 relative\_position\_bias = self.relative\_position\_bias\_table[self.relative\_position\_index.view(-1)].view(  
 self.window\_size[0] \* self.window\_size[1], self.window\_size[0] \* self.window\_size[1], -1)  
 relative\_position\_bias = relative\_position\_bias.permute(2, 0, 1).contiguous() # 调整维度  
 attn = attn + relative\_position\_bias.unsqueeze(0) # 加入相对位置偏置  
  
 attn = self.softmax(attn) # 计算Softmax  
 attn = self.attn\_drop(attn) # Dropout  
  
 x = (attn @ v).transpose(1, 2).reshape(B\_, N, C) # 计算输出  
 x = self.proj(x) # 线性变换  
 x = self.proj\_drop(x) # Dropout  
 return x  
  
  
class SwinTransformer(nn.Module):  
 """ Swin Transformer 主体 """  
  
 def \_\_init\_\_(self, patch\_size=4, embed\_dim=96, depths=[2, 2, 6, 2], num\_heads=[3, 6, 12, 24]):  
 super().\_\_init\_\_()  
 self.patch\_embed = PatchEmbed(patch\_size=patch\_size, in\_chans=3, embed\_dim=embed\_dim) # 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(  
 dim=int(embed\_dim \* 2 \*\* i\_layer),  
 depth=depths[i\_layer],  
 num\_heads=num\_heads[i\_layer]  
 )  
 self.layers.append(layer)  
  
 def forward(self, x):  
 """ 前向传播 """  
 x = self.patch\_embed(x) # 进行图像分块嵌入  
 for layer in self.layers:  
 x = layer(x) # 逐层前向传播  
 return x  
  
  
def SwinTransformer\_Tiny(weights=''):  
 """ 创建Swin Transformer Tiny模型 """  
 model = SwinTransformer(depths=[2, 2, 6, 2], num\_heads=[3, 6, 12, 24]) # 初始化模型  
 if weights:  
 model.load\_state\_dict(torch.load(weights)['model']) # 加载权重  
 return model  
```  
  
### 代码说明  
1. \*\*Mlp类\*\*：实现了一个简单的多层感知机，包含两层线性变换和激活函数。  
2. \*\*WindowAttention类\*\*：实现了窗口注意力机制，计算Q、K、V，并添加相对位置偏置。  
3. \*\*SwinTransformer类\*\*：构建了Swin Transformer的主体，包含图像分块嵌入和多个Transformer层。  
4. \*\*SwinTransformer\_Tiny函数\*\*：用于创建一个小型的Swin Transformer模型，并可选择加载预训练权重。  
  
通过以上注释，代码的核心功能和结构得以清晰展现。```

这个程序文件实现了Swin Transformer模型，主要用于计算机视觉任务。Swin Transformer是一种分层的视觉Transformer，采用了移动窗口机制以提高计算效率。代码中定义了多个类和函数，下面是对这些部分的详细说明。  
  
首先，程序导入了必要的库，包括PyTorch和一些用于构建神经网络的模块。接着，定义了一个名为`Mlp`的类，这是一个多层感知机（MLP），包含两个线性层和一个激活函数（默认为GELU），同时支持Dropout操作。该类的`forward`方法实现了前向传播。  
  
接下来，定义了`window\_partition`和`window\_reverse`函数，用于将输入特征图划分为窗口和将窗口合并回特征图。这种窗口划分的方式是Swin Transformer的核心思想之一，可以有效减少计算量。  
  
`WindowAttention`类实现了基于窗口的多头自注意力机制（W-MSA），支持相对位置偏置。该类的构造函数中定义了输入通道数、窗口大小、注意力头数等参数，并初始化了一些必要的权重。`forward`方法实现了自注意力的计算，包括查询、键、值的线性变换，以及相对位置偏置的添加。  
  
`SwinTransformerBlock`类是Swin Transformer的基本构建块，包含了窗口注意力和前馈网络（FFN）。在`forward`方法中，输入特征经过归一化、窗口划分、注意力计算、窗口合并等步骤，最后通过残差连接和DropPath进行输出。  
  
`PatchMerging`类用于将特征图的patch进行合并，减少特征图的空间维度。它通过线性层将4个输入通道合并为2个输出通道，并进行归一化。  
  
`BasicLayer`类表示Swin Transformer中的一个基本层，包含多个Swin Transformer块，并在最后进行下采样。它还计算了SW-MSA的注意力掩码，以支持循环位移的注意力机制。  
  
`PatchEmbed`类负责将输入图像划分为patch并进行嵌入，使用卷积层进行线性投影，并可选择性地进行归一化。  
  
`SwinTransformer`类是整个模型的主类，负责构建Swin Transformer的各个层，并实现前向传播。它支持绝对位置嵌入、Dropout等功能，并在输出时返回指定层的特征图。  
  
最后，`update\_weight`函数用于更新模型的权重，确保模型的权重与加载的权重字典相匹配。`SwinTransformer\_Tiny`函数则是一个便捷的工厂函数，用于创建一个小型的Swin Transformer模型，并可选择性地加载预训练权重。  
  
整体来看，这个程序文件提供了Swin Transformer的完整实现，涵盖了从输入图像处理到特征提取的各个环节，适合用于各种视觉任务。

```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.functional as F  
  
def build\_selective\_scan\_fn(selective\_scan\_cuda: object = None, mode="mamba\_ssm", tag=None):  
 """  
 构建选择性扫描函数的工厂函数。  
   
 参数:  
 selective\_scan\_cuda: 用于CUDA实现的选择性扫描模块。  
 mode: 选择性扫描的模式。  
 tag: 标签，用于标识不同的选择性扫描实现。  
   
 返回:  
 selective\_scan\_fn: 构建的选择性扫描函数。  
 """  
   
 class SelectiveScanFn(torch.autograd.Function):  
 @staticmethod  
 def forward(ctx, u, delta, A, B, C, D=None, z=None, delta\_bias=None, delta\_softplus=False, return\_last\_state=False, nrows=1, backnrows=-1):  
 """  
 前向传播函数，执行选择性扫描的计算。  
   
 参数:  
 ctx: 上下文对象，用于保存信息以便在反向传播中使用。  
 u: 输入张量。  
 delta: 增量张量。  
 A, B, C: 参与计算的参数张量。  
 D: 可选的额外参数张量。  
 z: 可选的张量。  
 delta\_bias: 可选的增量偏置。  
 delta\_softplus: 是否应用softplus函数。  
 return\_last\_state: 是否返回最后的状态。  
 nrows: 行数。  
 backnrows: 反向传播时的行数。  
   
 返回:  
 out: 输出张量。  
 last\_state: 最后状态（如果return\_last\_state为True）。  
 """  
 # 确保输入张量是连续的  
 if u.stride(-1) != 1:  
 u = u.contiguous()  
 if delta.stride(-1) != 1:  
 delta = delta.contiguous()  
 if D is not None:  
 D = D.contiguous()  
 if B.stride(-1) != 1:  
 B = B.contiguous()  
 if C.stride(-1) != 1:  
 C = C.contiguous()  
 if z is not None and z.stride(-1) != 1:  
 z = z.contiguous()  
  
 # 检查输入张量的形状是否符合要求  
 assert u.shape[1] % (B.shape[1] \* nrows) == 0   
 assert nrows in [1, 2, 3, 4] # 限制行数为1到4  
  
 if backnrows > 0:  
 assert u.shape[1] % (B.shape[1] \* backnrows) == 0   
 assert backnrows in [1, 2, 3, 4] # 限制反向传播行数为1到4  
 else:  
 backnrows = nrows  
 ctx.backnrows = backnrows  
   
 # 根据模式调用不同的CUDA实现  
 if mode == "mamba\_ssm":  
 out, x, \*rest = selective\_scan\_cuda.fwd(u, delta, A, B, C, D, z, delta\_bias, delta\_softplus)  
 else:  
 raise NotImplementedError  
  
 ctx.delta\_softplus = delta\_softplus  
 ctx.has\_z = z is not None  
  
 last\_state = x[:, :, -1, 1::2] # 获取最后的状态  
 ctx.save\_for\_backward(u, delta, A, B, C, D, delta\_bias, x)  
 return out if not return\_last\_state else (out, last\_state)  
  
 @staticmethod  
 def backward(ctx, dout):  
 """  
 反向传播函数，计算梯度。  
   
 参数:  
 ctx: 上下文对象，包含前向传播时保存的信息。  
 dout: 输出的梯度。  
   
 返回:  
 梯度张量。  
 """  
 u, delta, A, B, C, D, delta\_bias, x = ctx.saved\_tensors  
   
 # 调用CUDA实现的反向传播  
 du, ddelta, dA, dB, dC, dD, ddelta\_bias, \*rest = selective\_scan\_cuda.bwd(  
 u, delta, A, B, C, D, delta\_bias, dout, x, ctx.delta\_softplus, ctx.backnrows  
 )  
  
 return (du, ddelta, dA, dB, dC, dD if D is not None else None, ddelta\_bias if delta\_bias is not None else None)  
  
 def selective\_scan\_fn(u, delta, A, B, C, D=None, z=None, delta\_bias=None, delta\_softplus=False, return\_last\_state=False, nrows=1, backnrows=-1):  
 """  
 封装选择性扫描函数的调用。  
   
 返回:  
 选择性扫描的输出。  
 """  
 return SelectiveScanFn.apply(u, delta, A, B, C, D, z, delta\_bias, delta\_softplus, return\_last\_state, nrows, backnrows)  
  
 return selective\_scan\_fn  
```  
  
### 代码核心部分说明：  
1. \*\*`build\_selective\_scan\_fn` 函数\*\*：这是一个工厂函数，用于构建选择性扫描的函数。它接受CUDA实现、模式和标签作为参数，并返回一个选择性扫描函数。  
  
2. \*\*`SelectiveScanFn` 类\*\*：这个类继承自 `torch.autograd.Function`，实现了前向传播和反向传播的逻辑。前向传播中，调用了CUDA实现的选择性扫描函数，并保存必要的上下文信息。反向传播中，计算并返回梯度。  
  
3. \*\*前向传播 (`forward`)\*\*：该方法执行选择性扫描的主要计算逻辑，确保输入张量的连续性，检查形状，并根据模式调用相应的CUDA实现。  
  
4. \*\*反向传播 (`backward`)\*\*：该方法负责计算梯度，使用保存的上下文信息来调用CUDA实现的反向传播逻辑。  
  
5. \*\*`selective\_scan\_fn` 函数\*\*：这是一个封装函数，用于简化选择性扫描的调用，返回选择性扫描的输出。  
  
通过这些核心部分的实现，代码能够高效地执行选择性扫描操作，并支持自动求导功能。```

这个程序文件 `test\_selective\_scan\_speed.py` 是一个用于测试选择性扫描（Selective Scan）速度的脚本，主要使用 PyTorch 库来实现深度学习中的某些操作。程序的核心部分是实现选择性扫描的前向和反向传播函数，并进行性能测试。  
  
首先，程序导入了必要的库，包括 `torch`、`torch.nn.functional`、`pytest`、`time` 和 `functools.partial`。这些库提供了张量操作、自动求导、测试框架和函数部分应用等功能。  
  
接下来，定义了一个 `build\_selective\_scan\_fn` 函数，它接受一个 CUDA 实现的选择性扫描函数和一些参数。这个函数内部定义了一个名为 `SelectiveScanFn` 的类，继承自 `torch.autograd.Function`，并实现了前向传播（`forward`）和反向传播（`backward`）的方法。在前向传播中，程序会对输入的张量进行一些预处理，比如确保张量是连续的，并且根据输入的维度调整张量的形状。然后根据不同的模式调用相应的 CUDA 函数进行选择性扫描的计算。  
  
在反向传播中，程序根据保存的上下文（`ctx`）恢复输入张量，并调用相应的 CUDA 函数计算梯度。最终，返回计算得到的梯度。  
  
此外，程序还定义了多个选择性扫描的参考实现函数，如 `selective\_scan\_ref`、`selective\_scan\_easy\_v2` 和 `selective\_scan\_easy`，这些函数实现了选择性扫描的不同变体，主要用于在不同情况下的计算。  
  
最后，程序定义了一个 `test\_speed` 函数，进行性能测试。该函数设置了一些参数，包括数据类型、序列长度、批次大小等。然后生成随机输入数据，并调用不同的选择性扫描实现进行前向和反向传播的速度测试。每次测试结束后，程序会输出所用的时间，以便进行比较。  
  
整体来看，这个程序文件主要是为了实现和测试选择性扫描的功能，提供了灵活的接口以支持不同的实现和参数配置，同时也关注性能优化。

### 程序整体功能和构架概括  
  
该程序的整体功能是实现和测试深度学习中的一些关键组件，主要集中在卷积操作、归一化、Transformer架构和选择性扫描等方面。程序的构架由多个模块组成，每个模块实现特定的功能，支持灵活的模型构建和性能测试。具体来说：  
  
1. \*\*kan\_conv.py\*\*：实现了一种新的卷积层，结合了基础卷积和样条插值，支持多维数据处理。  
2. \*\*prepbn.py\*\*：定义了自定义的批量归一化和线性归一化模块，提供灵活的归一化操作。  
3. \*\*SwinTransformer.py\*\*：实现了Swin Transformer模型，适用于计算机视觉任务，包含多种注意力机制和特征提取方法。  
4. \*\*test\_selective\_scan\_speed.py\*\*：测试选择性扫描的速度，提供不同实现的性能比较，帮助优化选择性扫描的计算效率。  
  
### 文件功能整理表  
  
| 文件名 | 功能描述 |  
|-------------------------------|------------------------------------------------------------|  
| `kan\_conv.py` | 实现了一种新的卷积层（KANConvNDLayer），结合基础卷积和样条插值，支持多维数据处理。 |  
| `prepbn.py` | 定义了自定义的批量归一化（RepBN）和线性归一化（LinearNorm）模块，提供灵活的归一化操作。 |  
| `SwinTransformer.py` | 实现了Swin Transformer模型，包含多种注意力机制和特征提取方法，适用于计算机视觉任务。 |  
| `test\_selective\_scan\_speed.py`| 测试选择性扫描的速度，比较不同实现的性能，帮助优化选择性扫描的计算效率。 |  
  
这个表格清晰地总结了每个文件的功能，便于理解整个程序的构架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。