# 改进yolo11-DLKA等200+全套创新点大全：牛肉分级识别图像分割系统源码＆数据集全套

## 3.背景

研究背景与意义  
  
随着全球肉类消费的不断增长，牛肉的质量与安全性问题日益受到关注。牛肉分级不仅影响消费者的购买决策，还直接关系到肉类市场的经济效益。因此，建立一个高效、准确的牛肉分级识别系统，能够为牛肉产业链的各个环节提供重要的技术支持，提升产品质量与市场竞争力。传统的牛肉分级方法多依赖人工评估，效率低下且主观性强，容易受到评估者经验和判断的影响，导致分级结果的不一致性和不准确性。  
  
近年来，计算机视觉技术的迅猛发展为牛肉分级提供了新的解决方案。基于深度学习的图像分割技术，尤其是YOLO（You Only Look Once）系列模型，因其在实时物体检测和分割任务中的优越性能，成为研究的热点。YOLOv11作为该系列的最新版本，具备更高的检测精度和更快的处理速度，能够有效应对复杂的牛肉图像特征，进而实现准确的分级识别。  
  
本研究基于改进的YOLOv11模型，构建一个牛肉分级识别图像分割系统，旨在通过深度学习技术对牛肉进行自动化分级。数据集包含五个类别（M2、M3、M4、M5、M6），并通过高质量的图像和精确的标注，确保模型训练的有效性。通过对193幅图像的处理与分析，系统能够实现对牛肉质量的快速、准确评估，进而为牛肉产业的智能化发展提供支持。  
  
综上所述，本研究不仅具有重要的理论意义，推动了计算机视觉技术在农业领域的应用，同时也为牛肉产业的现代化管理提供了实用的技术手段，助力提升整体行业的生产效率与产品质量。

## 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以“IMF - BEEF”为主题，旨在为改进YOLOv11的牛肉分级识别图像分割系统提供强有力的支持。该数据集包含五个主要类别，分别为M2、M3、M4、M5和M6，涵盖了牛肉分级的不同层次。这些类别代表了牛肉在市场上的不同品质等级，具体包括M2（优质）、M3（良好）、M4（中等）、M5（较差）和M6（低质）。每个类别的图像样本均经过精心挑选和标注，以确保数据的准确性和代表性，从而为模型的训练提供高质量的输入。  
  
数据集的构建过程中，考虑到了牛肉分级的实际应用场景，采集了多种角度和光照条件下的牛肉图像，以增强模型的鲁棒性和泛化能力。这些图像不仅展示了牛肉的外观特征，还包含了与分级相关的细微差别，如纹理、颜色和脂肪分布等。这些特征在牛肉分级中起着至关重要的作用，因此，数据集中的图像将为YOLOv11模型的训练提供丰富的信息来源。  
  
此外，数据集的规模和多样性也经过精心设计，以确保模型能够在不同的实际应用环境中表现出色。通过对牛肉图像进行标注和分类，研究团队能够有效地训练出一个高效的图像分割系统，进而提升牛肉分级的自动化程度和准确性。最终，本项目的数据集将为牛肉分级的智能化提供坚实的基础，推动相关领域的技术进步和应用发展。

## 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提炼和详细注释。主要包括`OmniAttention`类、`FrequencySelection`类和`AdaptiveDilatedConv`类。其他辅助函数和类被省略以保持核心代码的简洁性。  
  
```python  
import torch  
import torch.nn as nn  
import torch.nn.functional as F  
  
class OmniAttention(nn.Module):  
 """OmniAttention类实现了全局注意力机制，主要用于通道、过滤器、空间和内核的注意力计算。"""  
  
 def \_\_init\_\_(self, in\_planes, out\_planes, kernel\_size, groups=1, reduction=0.0625, kernel\_num=4, min\_channel=16):  
 super(OmniAttention, self).\_\_init\_\_()  
 # 计算注意力通道数  
 attention\_channel = max(int(in\_planes \* reduction), min\_channel)  
 self.kernel\_size = kernel\_size  
 self.kernel\_num = kernel\_num  
 self.temperature = 1.0 # 温度参数用于控制注意力的平滑度  
  
 # 定义用于计算通道注意力的层  
 self.avgpool = nn.AdaptiveAvgPool2d(1)  
 self.fc = nn.Conv2d(in\_planes, attention\_channel, 1, bias=False)  
 self.bn = nn.BatchNorm2d(attention\_channel)  
 self.relu = nn.ReLU(inplace=True)  
  
 # 定义其他注意力机制的层  
 self.channel\_fc = nn.Conv2d(attention\_channel, in\_planes, 1, bias=True)  
 self.func\_channel = self.get\_channel\_attention  
  
 if in\_planes == groups and in\_planes == out\_planes: # 深度卷积  
 self.func\_filter = self.skip  
 else:  
 self.filter\_fc = nn.Conv2d(attention\_channel, out\_planes, 1, bias=True)  
 self.func\_filter = self.get\_filter\_attention  
  
 if kernel\_size == 1: # 点卷积  
 self.func\_spatial = self.skip  
 else:  
 self.spatial\_fc = nn.Conv2d(attention\_channel, kernel\_size \* kernel\_size, 1, bias=True)  
 self.func\_spatial = self.get\_spatial\_attention  
  
 if kernel\_num == 1:  
 self.func\_kernel = self.skip  
 else:  
 self.kernel\_fc = nn.Conv2d(attention\_channel, kernel\_num, 1, bias=True)  
 self.func\_kernel = self.get\_kernel\_attention  
  
 self.\_initialize\_weights() # 初始化权重  
  
 def \_initialize\_weights(self):  
 """初始化卷积层和批归一化层的权重"""  
 for m in self.modules():  
 if isinstance(m, nn.Conv2d):  
 nn.init.kaiming\_normal\_(m.weight, mode='fan\_out', nonlinearity='relu')  
 if m.bias is not None:  
 nn.init.constant\_(m.bias, 0)  
 if isinstance(m, nn.BatchNorm2d):  
 nn.init.constant\_(m.weight, 1)  
 nn.init.constant\_(m.bias, 0)  
  
 @staticmethod  
 def skip(\_):  
 """跳过函数，返回1.0"""  
 return 1.0  
  
 def get\_channel\_attention(self, x):  
 """计算通道注意力"""  
 channel\_attention = torch.sigmoid(self.channel\_fc(x).view(x.size(0), -1, 1, 1) / self.temperature)  
 return channel\_attention  
  
 def get\_filter\_attention(self, x):  
 """计算过滤器注意力"""  
 filter\_attention = torch.sigmoid(self.filter\_fc(x).view(x.size(0), -1, 1, 1) / self.temperature)  
 return filter\_attention  
  
 def get\_spatial\_attention(self, x):  
 """计算空间注意力"""  
 spatial\_attention = self.spatial\_fc(x).view(x.size(0), 1, 1, 1, self.kernel\_size, self.kernel\_size)  
 spatial\_attention = torch.sigmoid(spatial\_attention / self.temperature)  
 return spatial\_attention  
  
 def get\_kernel\_attention(self, x):  
 """计算内核注意力"""  
 kernel\_attention = self.kernel\_fc(x).view(x.size(0), -1, 1, 1, 1, 1)  
 kernel\_attention = F.softmax(kernel\_attention / self.temperature, dim=1)  
 return kernel\_attention  
  
 def forward(self, x):  
 """前向传播，计算各类注意力"""  
 x = self.avgpool(x) # 全局平均池化  
 x = self.fc(x) # 通过全连接层  
 x = self.bn(x) # 批归一化  
 x = self.relu(x) # 激活函数  
 return self.func\_channel(x), self.func\_filter(x), self.func\_spatial(x), self.func\_kernel(x)  
  
  
class FrequencySelection(nn.Module):  
 """FrequencySelection类用于选择频率特征并应用注意力机制。"""  
  
 def \_\_init\_\_(self, in\_channels, k\_list=[2], lowfreq\_att=True, fs\_feat='feat', lp\_type='freq', act='sigmoid', spatial='conv', spatial\_group=1):  
 super().\_\_init\_\_()  
 self.k\_list = k\_list # 频率列表  
 self.lp\_list = nn.ModuleList() # 存储不同频率的池化层  
 self.freq\_weight\_conv\_list = nn.ModuleList() # 存储频率权重卷积层  
 self.in\_channels = in\_channels  
 self.lowfreq\_att = lowfreq\_att  
  
 # 定义频率权重卷积层  
 for i in range(len(k\_list) + (1 if lowfreq\_att else 0)):  
 freq\_weight\_conv = nn.Conv2d(in\_channels, spatial\_group, kernel\_size=3, padding=1, groups=spatial\_group, bias=True)  
 self.freq\_weight\_conv\_list.append(freq\_weight\_conv)  
  
 # 定义不同类型的池化层  
 if lp\_type == 'avgpool':  
 for k in k\_list:  
 self.lp\_list.append(nn.AvgPool2d(kernel\_size=k, stride=1, padding=0))  
  
 def sp\_act(self, freq\_weight):  
 """应用激活函数"""  
 if self.act == 'sigmoid':  
 freq\_weight = freq\_weight.sigmoid() \* 2  
 elif self.act == 'softmax':  
 freq\_weight = freq\_weight.softmax(dim=1) \* freq\_weight.shape[1]  
 return freq\_weight  
  
 def forward(self, x):  
 """前向传播，计算频率选择和注意力"""  
 x\_list = []  
 pre\_x = x  
 b, \_, h, w = x.shape  
 for idx, avg in enumerate(self.lp\_list):  
 low\_part = avg(x) # 低频部分  
 high\_part = pre\_x - low\_part # 高频部分  
 pre\_x = low\_part  
 freq\_weight = self.freq\_weight\_conv\_list[idx](x) # 计算频率权重  
 freq\_weight = self.sp\_act(freq\_weight) # 应用激活函数  
 x\_list.append(freq\_weight \* high\_part) # 结合高频部分  
 if self.lowfreq\_att:  
 freq\_weight = self.freq\_weight\_conv\_list[len(x\_list)](x)  
 x\_list.append(freq\_weight \* pre\_x) # 结合低频部分  
 else:  
 x\_list.append(pre\_x)  
 return sum(x\_list) # 返回所有部分的和  
  
  
class AdaptiveDilatedConv(nn.Module):  
 """AdaptiveDilatedConv类实现了可调的膨胀卷积，结合了频率选择和注意力机制。"""  
  
 def \_\_init\_\_(self, in\_channels, out\_channels, kernel\_size, stride=1, padding=1, dilation=1):  
 super().\_\_init\_\_()  
 self.conv = nn.Conv2d(in\_channels, out\_channels, kernel\_size, stride=stride, padding=padding, dilation=dilation)  
 self.frequency\_selection = FrequencySelection(in\_channels) # 初始化频率选择模块  
  
 def forward(self, x):  
 """前向传播，执行卷积和频率选择"""  
 x = self.frequency\_selection(x) # 选择频率特征  
 x = self.conv(x) # 执行卷积  
 return x  
```  
  
### 代码核心部分说明  
1. \*\*OmniAttention\*\*: 该类实现了多种注意力机制，包括通道、过滤器、空间和内核注意力。通过全局平均池化和一系列卷积层来计算注意力权重，并在前向传播中返回这些权重。  
  
2. \*\*FrequencySelection\*\*: 该类用于选择特定频率的特征，并通过卷积层计算频率权重。它支持不同类型的池化操作，以提取低频和高频特征。  
  
3. \*\*AdaptiveDilatedConv\*\*: 该类封装了可调的膨胀卷积，结合了频率选择模块。通过卷积操作和频率选择来处理输入特征。  
  
这些核心部分构成了整个模型的基础，能够实现复杂的特征提取和注意力机制。```

这个 `fadc.py` 文件包含了多个深度学习模型的实现，主要是基于 PyTorch 框架。文件中定义了几个类，包括 `OmniAttention`、`FrequencySelection`、`AdaptiveDilatedConv` 和 `AdaptiveDilatedDWConv`，它们都涉及到卷积操作和注意力机制。以下是对这些类的逐一分析和说明。  
  
首先，`OmniAttention` 类实现了一种全局注意力机制。它的构造函数接收多个参数，包括输入和输出通道数、卷积核大小、分组数、通道缩减比例等。该类的核心在于计算通道注意力、过滤器注意力、空间注意力和卷积核注意力。它通过自适应平均池化、全连接层和卷积层来实现这些注意力机制，并在前向传播中计算这些注意力权重。该类还包含了权重初始化和温度更新的方法。  
  
接下来，`generate\_laplacian\_pyramid` 函数用于生成拉普拉斯金字塔，它可以用于图像处理中的多尺度特征提取。该函数接收输入张量、金字塔层数和其他参数，逐层下采样并计算拉普拉斯差分，最终返回金字塔的各层特征。  
  
`FrequencySelection` 类实现了一种频率选择机制，允许对输入特征进行频率域的处理。它支持多种操作模式，包括平均池化和拉普拉斯金字塔等。该类的构造函数接收输入通道数、频率列表、空间卷积参数等，并根据这些参数初始化多个卷积层。它的前向传播方法根据不同的模式计算频率选择的特征。  
  
`AdaptiveDilatedConv` 类是一个可调节的膨胀卷积层，继承自 `ModulatedDeformConv2d`。它支持多种卷积类型和频率选择机制，并通过注意力机制来调整卷积权重。该类的构造函数接收多个参数，包括输入输出通道、卷积核大小、步幅、填充、膨胀因子等。它在前向传播中计算偏移量和掩码，并根据注意力权重和偏移量进行卷积操作。  
  
最后，`AdaptiveDilatedDWConv` 类是一个适应性膨胀深度卷积层，类似于 `AdaptiveDilatedConv`，但它专门针对深度卷积进行优化。它的构造函数和前向传播方法与 `AdaptiveDilatedConv` 类似，但增加了对深度卷积的支持。  
  
总体来说，这个文件实现了一系列复杂的卷积和注意力机制，适用于图像处理和计算机视觉任务。通过结合频率选择和注意力机制，这些类能够有效地提取和处理多尺度特征，从而提高模型的性能。

```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  
from functools import partial  
  
class Mlp(nn.Module):  
 """多层感知机模块，包含卷积层和激活函数"""  
 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.Conv2d(in\_features, hidden\_features, 1) # 第一个卷积层  
 self.dwconv = DWConv(hidden\_features) # 深度卷积层  
 self.act = act\_layer() # 激活函数  
 self.fc2 = nn.Conv2d(hidden\_features, out\_features, 1) # 第二个卷积层  
 self.drop = nn.Dropout(drop) # Dropout层  
  
 def forward(self, x):  
 x = self.fc1(x) # 通过第一个卷积层  
 x = self.dwconv(x) # 通过深度卷积层  
 x = self.act(x) # 激活  
 x = self.drop(x) # Dropout  
 x = self.fc2(x) # 通过第二个卷积层  
 x = self.drop(x) # Dropout  
 return x  
  
class Attention(nn.Module):  
 """注意力模块"""  
 def \_\_init\_\_(self, d\_model):  
 super().\_\_init\_\_()  
 self.proj\_1 = nn.Conv2d(d\_model, d\_model, 1) # 投影层1  
 self.activation = nn.GELU() # 激活函数  
 self.spatial\_gating\_unit = LSKblock(d\_model) # 空间门控单元  
 self.proj\_2 = nn.Conv2d(d\_model, d\_model, 1) # 投影层2  
  
 def forward(self, x):  
 shortcut = x.clone() # 保存输入用于残差连接  
 x = self.proj\_1(x) # 通过投影层1  
 x = self.activation(x) # 激活  
 x = self.spatial\_gating\_unit(x) # 通过空间门控单元  
 x = self.proj\_2(x) # 通过投影层2  
 x = x + shortcut # 残差连接  
 return x  
  
class Block(nn.Module):  
 """网络的基本构建块，包含注意力和多层感知机"""  
 def \_\_init\_\_(self, dim, mlp\_ratio=4., drop=0., drop\_path=0., act\_layer=nn.GELU):  
 super().\_\_init\_\_()  
 self.norm1 = nn.BatchNorm2d(dim) # 归一化层1  
 self.norm2 = nn.BatchNorm2d(dim) # 归一化层2  
 self.attn = Attention(dim) # 注意力模块  
 self.mlp = Mlp(in\_features=dim, hidden\_features=int(dim \* mlp\_ratio), act\_layer=act\_layer, drop=drop) # MLP模块  
  
 def forward(self, x):  
 x = x + self.attn(self.norm1(x)) # 通过注意力模块并添加残差  
 x = x + self.mlp(self.norm2(x)) # 通过MLP模块并添加残差  
 return x  
  
class LSKNet(nn.Module):  
 """LSKNet网络结构"""  
 def \_\_init\_\_(self, img\_size=224, in\_chans=3, embed\_dims=[64, 128, 256, 512], depths=[3, 4, 6, 3]):  
 super().\_\_init\_\_()  
 self.num\_stages = len(depths) # 网络阶段数  
 for i in range(self.num\_stages):  
 # 每个阶段的嵌入和块  
 patch\_embed = OverlapPatchEmbed(img\_size=img\_size // (2 \*\* i), in\_chans=in\_chans if i == 0 else embed\_dims[i - 1], embed\_dim=embed\_dims[i])  
 block = nn.ModuleList([Block(dim=embed\_dims[i]) for \_ in range(depths[i])])  
 setattr(self, f"patch\_embed{i + 1}", patch\_embed)  
 setattr(self, f"block{i + 1}", block)  
  
 def forward(self, x):  
 outs = []  
 for i in range(self.num\_stages):  
 patch\_embed = getattr(self, f"patch\_embed{i + 1}")  
 block = getattr(self, f"block{i + 1}")  
 x, \_, \_ = patch\_embed(x) # 嵌入  
 for blk in block:  
 x = blk(x) # 通过每个块  
 outs.append(x) # 收集输出  
 return outs  
  
class DWConv(nn.Module):  
 """深度卷积模块"""  
 def \_\_init\_\_(self, dim=768):  
 super(DWConv, self).\_\_init\_\_()  
 self.dwconv = nn.Conv2d(dim, dim, 3, 1, 1, bias=True, groups=dim) # 深度卷积层  
  
 def forward(self, x):  
 return self.dwconv(x) # 通过深度卷积层  
  
def lsknet\_t(weights=''):  
 """构建LSKNet-t模型"""  
 model = LSKNet(embed\_dims=[32, 64, 160, 256], depths=[3, 3, 5, 2])  
 if weights:  
 model.load\_state\_dict(torch.load(weights)['state\_dict']) # 加载权重  
 return model  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 model = lsknet\_t('lsk\_t\_backbone-2ef8a593.pth') # 实例化模型  
 inputs = torch.randn((1, 3, 640, 640)) # 随机输入  
 for i in model(inputs):  
 print(i.size()) # 输出每层的尺寸  
```  
  
### 代码说明：  
1. \*\*Mlp类\*\*：实现了一个多层感知机，包括两个卷积层和一个深度卷积层。  
2. \*\*Attention类\*\*：实现了一个注意力机制，包含两个投影层和一个空间门控单元。  
3. \*\*Block类\*\*：构建了网络的基本模块，结合了注意力和多层感知机。  
4. \*\*LSKNet类\*\*：构建了整个网络结构，包含多个阶段，每个阶段由嵌入层和多个Block组成。  
5. \*\*DWConv类\*\*：实现了深度卷积，用于特征提取。  
6. \*\*lsknet\_t函数\*\*：用于创建LSKNet-t模型，并可选择加载预训练权重。  
  
该代码实现了一个深度学习模型的核心结构，适用于图像处理任务。```

这个程序文件定义了一个名为 `lsknet.py` 的深度学习模型，主要用于图像处理任务。它使用了 PyTorch 框架，并包含了多个类和函数，构成了一个复杂的神经网络结构。  
  
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，它是一个多层感知机，包含两个卷积层和一个深度可分离卷积（`DWConv`）层，以及激活函数和 dropout 层。这个类的 `forward` 方法定义了数据的前向传播过程。  
  
接下来，定义了 `LSKblock` 类，它实现了一种新的卷积块结构，使用了多种卷积操作来提取特征。该类通过空间注意力机制对输入进行处理，计算平均和最大注意力，并通过卷积层进行加权组合，最终输出与输入相乘的结果。  
  
然后，定义了 `Attention` 类，它包含了一个注意力机制，使用 `LSKblock` 来增强特征提取能力。该类在前向传播中对输入进行线性变换、激活、注意力处理和残差连接。  
  
`Block` 类是网络的基本构建块，结合了归一化、注意力机制和多层感知机。它使用了 dropout 和层缩放技术，以增强模型的稳定性和性能。  
  
`OverlapPatchEmbed` 类负责将输入图像分割成重叠的补丁，并进行嵌入。它使用卷积层将输入通道映射到嵌入维度，并进行归一化处理。  
  
`LSKNet` 类是整个网络的核心，定义了网络的结构，包括多个阶段，每个阶段由补丁嵌入、多个块和归一化层组成。它的 `forward` 方法实现了输入数据的逐层处理，并将每个阶段的输出保存到列表中。  
  
`DWConv` 类实现了深度可分离卷积，主要用于减少模型的参数量和计算复杂度。  
  
此外，程序还定义了 `update\_weight` 函数，用于更新模型的权重，确保加载的权重与模型结构相匹配。最后，提供了两个函数 `lsknet\_t` 和 `lsknet\_s`，用于创建不同配置的 `LSKNet` 模型，并可选择加载预训练权重。  
  
在 `\_\_main\_\_` 部分，程序实例化了一个 `lsknet\_t` 模型，并生成了一个随机输入张量，随后打印出模型对输入的输出尺寸。这部分代码用于测试模型的基本功能。整体来看，这个程序实现了一个复杂的图像处理神经网络，结合了多种先进的技术和结构。

```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  
from torch.autograd import Function  
import pywt  
  
def create\_wavelet\_filter(wave, in\_size, out\_size, type=torch.float):  
 # 创建小波滤波器  
 w = pywt.Wavelet(wave)  
 # 反转并转换为张量  
 dec\_hi = torch.tensor(w.dec\_hi[::-1], dtype=type)  
 dec\_lo = torch.tensor(w.dec\_lo[::-1], dtype=type)  
   
 # 生成分解滤波器  
 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)  
  
 # 生成重构滤波器  
 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])  
 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  
  
def wavelet\_transform(x, filters):  
 # 执行小波变换  
 b, c, h, w = x.shape  
 pad = (filters.shape[2] // 2 - 1, filters.shape[3] // 2 - 1)  
 # 使用卷积进行小波变换  
 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  
 return x  
  
def inverse\_wavelet\_transform(x, filters):  
 # 执行逆小波变换  
 b, c, \_, h\_half, w\_half = x.shape  
 pad = (filters.shape[2] // 2 - 1, filters.shape[3] // 2 - 1)  
 x = x.reshape(b, c \* 4, h\_half, w\_half) # 重塑输入  
 # 使用转置卷积进行逆小波变换  
 x = F.conv\_transpose2d(x, filters.to(x.dtype).to(x.device), stride=2, groups=c, padding=pad)  
 return x  
  
class WaveletTransform(Function):  
 @staticmethod  
 def forward(ctx, input, filters):  
 ctx.filters = filters  
 with torch.no\_grad():  
 x = wavelet\_transform(input, filters) # 前向小波变换  
 return x  
  
 @staticmethod  
 def backward(ctx, grad\_output):  
 grad = inverse\_wavelet\_transform(grad\_output, ctx.filters) # 反向传播  
 return grad, None  
  
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)  
  
 # 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  
 x\_ll\_in\_levels = []  
 x\_h\_in\_levels = []  
 curr\_x\_ll = x  
  
 for i in range(self.wt\_levels):  
 curr\_x = wavelet\_transform(curr\_x\_ll, self.wt\_filter) # 小波变换  
 curr\_x\_ll = curr\_x[:, :, 0, :, :] # 提取低频部分  
 x\_ll\_in\_levels.append(curr\_x\_ll)  
 x\_h\_in\_levels.append(curr\_x[:, :, 1:4, :, :]) # 提取高频部分  
  
 # 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  
 next\_x\_ll = inverse\_wavelet\_transform(curr\_x, self.iwt\_filter) # 逆小波变换  
  
 x = self.base\_conv(x) + next\_x\_ll # 加上基础卷积的输出  
 return x  
```  
  
### 代码说明：  
1. \*\*创建小波滤波器\*\*：`create\_wavelet\_filter` 函数生成小波变换和逆变换所需的滤波器。  
2. \*\*小波变换\*\*：`wavelet\_transform` 和 `inverse\_wavelet\_transform` 函数分别实现小波变换和逆小波变换。  
3. \*\*WaveletTransform 类\*\*：自定义的 PyTorch 函数，用于在前向和反向传播中应用小波变换。  
4. \*\*WTConv2d 类\*\*：自定义的卷积层，结合了小波变换和基础卷积操作。它在前向传播中执行小波变换，并在后续步骤中使用逆小波变换。```

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，名为 `WTConv2d`。它结合了小波变换和卷积操作，能够对输入的特征图进行多层次的处理。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 PyWavelets。接着，定义了一些函数和类，用于创建小波滤波器、执行小波变换和逆变换，以及实现自定义的 PyTorch 操作。  
  
`create\_wavelet\_filter` 函数用于生成小波变换的滤波器。它接受小波类型、输入通道数和输出通道数作为参数，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。  
  
`wavelet\_transform` 和 `inverse\_wavelet\_transform` 函数分别实现小波变换和逆小波变换。小波变换通过对输入张量应用卷积操作，将其转换为包含低频和高频信息的多个子带；而逆小波变换则将这些子带重构回原始的特征图。  
  
接下来，定义了两个类 `WaveletTransform` 和 `InverseWaveletTransform`，它们继承自 `torch.autograd.Function`。这两个类实现了前向和反向传播的方法，允许在训练过程中自动计算梯度。  
  
`wavelet\_transform\_init` 和 `inverse\_wavelet\_transform\_init` 函数用于初始化小波变换和逆变换的应用函数。  
  
`WTConv2d` 类是这个模块的核心，继承自 `nn.Module`。在初始化方法中，它接受输入通道数、输出通道数、卷积核大小、步幅、偏置、以及小波变换的层数和类型。该类创建了小波滤波器、基础卷积层、以及多个小波卷积层和缩放模块。  
  
在 `forward` 方法中，输入特征图经过多层小波变换，提取出低频和高频信息。每一层的小波变换后，低频信息被保留并传递到下一层，而高频信息则被处理并在最后与重构的低频信息相加。最终，经过基础卷积层和缩放模块的处理后，输出结果。  
  
`\_ScaleModule` 类是一个简单的缩放模块，用于对输入进行缩放操作，便于在卷积操作后调整输出的特征图。  
  
整体来看，这个程序实现了一个复杂的卷积层，通过小波变换提取特征，适用于需要多尺度特征提取的深度学习任务。

```以下是经过简化并注释的核心代码部分：  
  
```python  
import torch  
import torch.nn as nn  
  
class DropPath(nn.Module):  
 """实现随机深度（Drop Path）机制的模块"""  
 def \_\_init\_\_(self, drop\_prob: float = 0.1):  
 super().\_\_init\_\_()  
 self.drop\_prob = drop\_prob # 路径被丢弃的概率  
  
 def forward(self, x: torch.Tensor) -> torch.Tensor:  
 """前向传播，应用Drop Path机制"""  
 if self.drop\_prob == 0. or not self.training:  
 return x # 如果不丢弃，直接返回输入  
 keep\_prob = 1 - self.drop\_prob  
 shape = (x.shape[0], ) + (1, ) \* (x.ndim - 1) # 生成与输入相同的形状  
 random\_tensor = keep\_prob + torch.rand(shape, dtype=x.dtype, device=x.device) # 生成随机张量  
 output = x.div(keep\_prob) \* random\_tensor.floor() # 应用Drop Path  
 return output  
  
class ConvFFN(nn.Module):  
 """使用卷积模块实现的多层感知机（FFN）"""  
 def \_\_init\_\_(self, in\_channels: int, out\_channels: int, hidden\_channels\_scale: float = 4.0):  
 super().\_\_init\_\_()  
 hidden\_channels = int(in\_channels \* hidden\_channels\_scale) # 隐藏层通道数  
  
 self.ffn\_layers = nn.Sequential(  
 nn.Conv2d(in\_channels, hidden\_channels, kernel\_size=1), # 1x1卷积  
 nn.ReLU(), # 激活函数  
 nn.Conv2d(hidden\_channels, out\_channels, kernel\_size=1) # 1x1卷积  
 )  
  
 def forward(self, x):  
 return self.ffn\_layers(x) # 前向传播  
  
class PKIBlock(nn.Module):  
 """多核Inception模块"""  
 def \_\_init\_\_(self, in\_channels: int, out\_channels: int):  
 super().\_\_init\_\_()  
 self.conv1 = nn.Conv2d(in\_channels, out\_channels, kernel\_size=1) # 1x1卷积  
 self.conv2 = nn.Conv2d(out\_channels, out\_channels, kernel\_size=3, padding=1) # 3x3卷积  
 self.ffn = ConvFFN(out\_channels, out\_channels) # FFN  
  
 def forward(self, x):  
 x = self.conv1(x) # 通过1x1卷积  
 x = self.conv2(x) # 通过3x3卷积  
 x = self.ffn(x) # 通过FFN  
 return x  
  
class PKINet(nn.Module):  
 """多核Inception网络"""  
 def \_\_init\_\_(self):  
 super().\_\_init\_\_()  
 self.stem = nn.Conv2d(3, 32, kernel\_size=3, stride=2, padding=1) # Stem层  
 self.block = PKIBlock(32, 64) # Inception模块  
  
 def forward(self, x):  
 x = self.stem(x) # 通过Stem层  
 x = self.block(x) # 通过Inception模块  
 return x  
  
def PKINET\_T():  
 """返回T版本的PKINet模型"""  
 return PKINet()  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 model = PKINET\_T() # 创建模型实例  
 inputs = torch.randn((1, 3, 640, 640)) # 生成随机输入  
 res = model(inputs) # 前向传播  
 print(res.size()) # 输出结果的尺寸  
```  
  
### 代码注释说明：  
1. \*\*DropPath\*\*: 实现了随机深度的机制，允许在训练过程中随机丢弃某些路径，以增强模型的泛化能力。  
2. \*\*ConvFFN\*\*: 这是一个简单的多层感知机，使用卷积层代替全连接层，适用于处理图像数据。  
3. \*\*PKIBlock\*\*: 这是一个多核Inception模块，包含多个卷积层和一个FFN，旨在提取特征。  
4. \*\*PKINet\*\*: 这是一个整体的网络结构，包含Stem层和多个Inception模块，用于图像处理任务。  
5. \*\*PKINET\_T\*\*: 用于创建PKINet的一个实例。```

这个程序文件 `pkinet.py` 实现了一个名为 PKINet 的深度学习模型，主要用于图像处理任务。它的结构基于一种称为“多核卷积”的网络架构，包含多个模块和层次。以下是对代码的详细讲解。  
  
首先，程序导入了一些必要的库，包括数学库、类型提示、PyTorch 及其神经网络模块。接着，它尝试导入一些来自 `mmcv` 和 `mmengine` 的模块，如果导入失败，则使用 PyTorch 的基础模块。  
  
文件中定义了一些工具函数，例如 `drop\_path` 用于实现随机深度（Stochastic Depth），`autopad` 用于自动计算卷积的填充，`make\_divisible` 用于确保通道数可以被指定的除数整除。此外，还有两个类 `BCHW2BHWC` 和 `BHWC2BCHW` 用于在不同的张量维度之间进行转换。  
  
接下来，定义了几个核心模块。`GSiLU` 是一种激活函数，结合了全局平均池化和 Sigmoid 函数。`CAA` 是上下文锚注意力模块，用于增强特征表示。`ConvFFN` 是一个多层感知机实现，使用卷积模块构建。`Stem` 和 `DownSamplingLayer` 分别是网络的初始层和下采样层。  
  
`InceptionBottleneck` 是一个瓶颈结构，结合了多个卷积层和注意力机制。`PKIBlock` 是多核卷积块，包含多个 `InceptionBottleneck`，并在其中应用了 DropPath 和层缩放机制。`PKIStage` 是网络的一个阶段，包含多个 `PKIBlock`，并在输入和输出之间进行下采样。  
  
`PKINet` 是整个网络的主体，包含多个阶段和一个初始层。它的构造函数根据给定的架构设置（如 T、S、B）初始化各个阶段，并设置参数。`forward` 方法定义了前向传播过程，返回指定的输出索引。  
  
最后，文件提供了三个函数 `PKINET\_T`、`PKINET\_S` 和 `PKINET\_B`，用于实例化不同配置的 PKINet 模型。主程序部分创建了一个 PKINET\_T 模型实例，并生成随机输入以测试模型的输出尺寸。  
  
总体而言，这个文件实现了一个复杂的深度学习模型，结合了多种现代卷积神经网络的设计理念，适用于图像分类或其他视觉任务。

### 程序整体功能和构架概括  
  
该程序包含多个深度学习模型和自定义卷积层，主要用于图像处理和计算机视觉任务。每个文件实现了不同的网络结构和功能模块，结合了现代卷积神经网络的设计理念，如注意力机制、多尺度特征提取和小波变换等。这些模块和模型可以独立使用，也可以组合在一起，以满足特定的应用需求。  
  
以下是每个文件的功能概述：  
  
| 文件名 | 功能描述 |  
|-------------------|--------------------------------------------------------------------------|  
| `fadc.py` | 实现了多个卷积和注意力机制的类，包括 `OmniAttention`、`FrequencySelection`、`AdaptiveDilatedConv` 和 `AdaptiveDilatedDWConv`，用于多尺度特征提取和增强模型性能。 |  
| `lsknet.py` | 定义了 LSKNet 模型，结合了多层感知机、注意力机制和深度卷积，适用于图像处理任务，提供了不同配置的模型实例化函数。 |  
| `wtconv2d.py` | 实现了基于小波变换的二维卷积层 `WTConv2d`，结合小波变换和卷积操作，适用于多尺度特征提取，提供了小波变换和逆变换的功能。 |  
| `pkinet.py` | 定义了 PKINet 模型，采用多核卷积结构，结合了上下文锚注意力和多层感知机，适用于图像分类等视觉任务，提供了不同配置的模型实例化函数。 |  
  
通过这些模块，用户可以构建复杂的深度学习模型，进行图像分类、目标检测等任务，并利用现代卷积神经网络的优势来提高性能。