# 改进yolo11-RFCBAMConv等200+全套创新点大全：无人机视角水面漂浮物体检测系统源码＆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  
  
随着全球水资源的日益紧张和环境污染问题的加剧，水面漂浮物体的监测与管理显得尤为重要。漂浮物体不仅影响水体的生态环境，还可能对水质和水生生物造成严重威胁。因此，开发高效的水面漂浮物体检测系统，能够及时识别和处理水面污染物，具有重要的现实意义和应用价值。近年来，随着无人机技术的快速发展，利用无人机进行水面监测已成为一种新兴的研究方向。无人机搭载高分辨率摄像头，可以从空中获取大范围的水面图像，极大地提高了监测的效率和准确性。  
  
在此背景下，基于改进YOLOv11的无人机视角水面漂浮物体检测系统应运而生。YOLO（You Only Look Once）系列模型因其高效的实时检测能力而广泛应用于目标检测领域。YOLOv11作为该系列的最新版本，具备更强的特征提取能力和更快的推理速度，适合在动态环境中进行漂浮物体的实时检测。通过对YOLOv11模型的改进，可以进一步提升其在水面漂浮物体检测中的准确性和鲁棒性。  
  
本研究将利用“Aerial Data of Floating Objects”数据集，该数据集包含25幅图像，涵盖2类漂浮物体。尽管数据量相对较小，但通过数据增强和迁移学习等技术，可以有效提升模型的泛化能力。此外，研究还将探讨不同环境条件下的检测性能，以确保系统在实际应用中的可靠性和有效性。综上所述，基于改进YOLOv11的无人机视角水面漂浮物体检测系统不仅为水体监测提供了一种创新的解决方案，也为环境保护和水资源管理提供了重要的技术支持。

## 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“Aerial Data of Floating Objects”，旨在为改进YOLOv11的无人机视角水面漂浮物体检测系统提供高质量的训练数据。该数据集专注于水面漂浮物体的检测，特别是针对一种特定类别的物体——“paddle-board”。在数据集中，类别数量为1，体现了该项目对特定物体的深度学习和精准识别的需求。  
  
“Aerial Data of Floating Objects”数据集包含了多种场景下的无人机拍摄图像，这些图像展示了不同环境条件下的水面及其上漂浮的桨板。数据集中的图像不仅涵盖了不同的光照条件和天气变化，还包括了多种角度和距离的拍摄，确保了模型在实际应用中的鲁棒性。每幅图像都经过精确标注，确保了训练过程中所需的高质量标签信息，从而提升了模型的学习效果。  
  
通过使用这一数据集，研究人员能够有效地训练和优化YOLOv11模型，使其在检测水面漂浮物体时具备更高的准确性和实时性。数据集的设计考虑到了无人机在实际应用中的多样性和复杂性，力求在不同的环境中实现稳定的检测性能。随着无人机技术的不断发展，水面漂浮物体的检测将变得愈发重要，尤其是在海洋监测、环境保护和水上运动等领域。因此，“Aerial Data of Floating Objects”数据集的构建不仅为本项目提供了必要的基础数据支持，也为未来相关研究提供了重要的参考价值。

## 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取和详细注释。代码主要涉及一个基于注意力机制的卷积神经网络模块和一个仓库管理器，用于动态管理卷积核的权重。  
  
```python  
import torch  
import torch.nn as nn  
import torch.nn.functional as F  
import collections.abc  
  
# 定义核心模块  
class Attention(nn.Module):  
 def \_\_init\_\_(self, in\_planes, reduction, num\_static\_cell, num\_local\_mixture, norm\_layer=nn.BatchNorm1d):  
 super(Attention, self).\_\_init\_\_()  
 # 计算隐藏层的通道数  
 hidden\_planes = max(int(in\_planes \* reduction), 16)  
 self.kw\_planes\_per\_mixture = num\_static\_cell + 1 # 每个混合的通道数  
 self.num\_local\_mixture = num\_local\_mixture # 本地混合数  
 self.kw\_planes = self.kw\_planes\_per\_mixture \* num\_local\_mixture # 总通道数  
  
 # 定义网络层  
 self.avgpool = nn.AdaptiveAvgPool1d(1) # 自适应平均池化  
 self.fc1 = nn.Linear(in\_planes, hidden\_planes) # 全连接层  
 self.norm1 = norm\_layer(hidden\_planes) # 归一化层  
 self.act1 = nn.ReLU(inplace=True) # 激活函数  
  
 # 初始化权重  
 self.\_initialize\_weights()  
  
 def \_initialize\_weights(self):  
 # 权重初始化  
 for m in self.modules():  
 if isinstance(m, nn.Linear):  
 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.BatchNorm1d):  
 nn.init.constant\_(m.weight, 1)  
 nn.init.constant\_(m.bias, 0)  
  
 def forward(self, x):  
 # 前向传播  
 x = self.avgpool(x.reshape(\*x.shape[:2], -1)).squeeze(dim=-1) # 池化  
 x = self.act1(self.norm1(self.fc1(x))) # 线性变换 + 归一化 + 激活  
 return x # 返回处理后的特征  
  
class KWconvNd(nn.Module):  
 def \_\_init\_\_(self, in\_planes, out\_planes, kernel\_size, stride=1, padding=0, dilation=1, groups=1):  
 super(KWconvNd, self).\_\_init\_\_()  
 self.in\_planes = in\_planes # 输入通道数  
 self.out\_planes = out\_planes # 输出通道数  
 self.kernel\_size = kernel\_size # 卷积核大小  
 self.stride = stride # 步幅  
 self.padding = padding # 填充  
 self.dilation = dilation # 膨胀  
 self.groups = groups # 分组卷积  
  
 def forward(self, x):  
 # 前向传播  
 weight = self.get\_weight() # 获取权重  
 output = F.conv2d(x, weight, stride=self.stride, padding=self.padding, groups=self.groups) # 卷积操作  
 return output # 返回卷积结果  
  
class Warehouse\_Manager(nn.Module):  
 def \_\_init\_\_(self):  
 super(Warehouse\_Manager, self).\_\_init\_\_()  
 self.warehouse\_list = {} # 存储卷积核的仓库  
  
 def reserve(self, in\_planes, out\_planes, kernel\_size=1, stride=1, padding=0, groups=1):  
 # 创建卷积层并记录其信息  
 weight\_shape = [out\_planes, in\_planes, kernel\_size, kernel\_size] # 权重形状  
 self.warehouse\_list['default'] = weight\_shape # 存储权重形状  
 return KWconvNd(in\_planes, out\_planes, kernel\_size, stride, padding, groups) # 返回卷积层  
  
 def take\_cell(self):  
 # 获取卷积核  
 return self.warehouse\_list['default'] # 返回默认仓库的卷积核  
  
# 使用示例  
wm = Warehouse\_Manager() # 创建仓库管理器  
conv\_layer = wm.reserve(3, 16, kernel\_size=3) # 创建卷积层  
```  
  
### 代码核心部分说明：  
1. \*\*Attention类\*\*：实现了一个注意力机制，用于对输入特征进行加权处理。包含了多个层（全连接层、归一化层和激活函数），并在前向传播中计算注意力权重。  
   
2. \*\*KWconvNd类\*\*：实现了一个可扩展的卷积层，支持动态管理卷积核的权重。其前向传播中调用了PyTorch的卷积函数。  
  
3. \*\*Warehouse\_Manager类\*\*：用于管理卷积核的仓库，支持动态创建和存储卷积层的权重信息。可以根据输入参数生成相应的卷积层。  
  
### 总结：  
以上代码展示了如何构建一个动态卷积网络，利用注意力机制来增强特征提取能力，同时通过仓库管理器来管理卷积核的权重，便于在不同层之间共享和复用卷积核。```

这个文件 `kernel\_warehouse.py` 是一个用于实现深度学习中卷积操作的模块，特别是涉及到“内核仓库”（Kernel Warehouse）管理的卷积层。它主要包括几个类和函数，下面是对这些内容的逐一分析。  
  
首先，文件导入了一些必要的库，包括 PyTorch 的核心模块和一些数学函数。接着，定义了一个 `parse` 函数，用于处理输入参数，确保它们是可迭代的，并根据需要将其扩展到指定的长度。  
  
接下来，定义了一个 `Attention` 类，这是一个神经网络模块，负责计算注意力权重。它的构造函数接受多个参数，包括输入通道数、缩减比例、静态单元数量、局部混合数量等。该类的主要功能是通过全连接层和激活函数对输入进行处理，并生成注意力权重，这些权重用于后续的卷积操作。  
  
`KWconvNd` 类是一个抽象的卷积层类，继承自 `nn.Module`。它的构造函数接受输入和输出通道数、卷积核大小、步幅、填充、扩张、分组等参数，并初始化相应的属性。该类的 `init\_attention` 方法用于初始化注意力机制，`forward` 方法则实现了前向传播过程，利用注意力权重和仓库中的卷积核进行卷积操作。  
  
随后，定义了三个具体的卷积类 `KWConv1d`、`KWConv2d` 和 `KWConv3d`，分别对应一维、二维和三维卷积操作。这些类通过设置不同的维度和排列方式来实现具体的卷积功能。  
  
`KWLinear` 类是一个线性层的实现，内部使用 `KWConv1d` 来实现线性变换。  
  
`Warehouse\_Manager` 类是整个模块的核心，负责管理卷积层的内核仓库。它的构造函数接受多个参数，用于配置仓库的行为。该类的 `reserve` 方法用于创建动态卷积层而不分配权重，并记录相关信息。`store` 方法则用于计算并存储仓库中的权重。`allocate` 方法负责将权重分配给网络中的卷积层。  
  
最后，`KWConv` 类是一个高层次的卷积封装，结合了卷积层、批归一化和激活函数。它的 `forward` 方法实现了输入的前向传播。  
  
此外，文件中还定义了一个 `get\_temperature` 函数，用于根据当前的训练迭代和总的训练周期计算温度值，这在某些训练策略中可能用于调整模型的学习率或其他超参数。  
  
总体而言，这个文件实现了一个灵活的卷积层管理系统，允许动态地管理和分配卷积核，适用于需要优化卷积操作的深度学习模型。

```以下是经过简化并添加详细中文注释的核心代码部分：  
  
```python  
import math  
import torch  
import torch.nn as nn  
  
class DropPath(nn.Module):  
 """实现随机深度（Drop Path）机制的模块，适用于残差块的主路径。"""  
   
 def \_\_init\_\_(self, drop\_prob: float = 0.1):  
 super().\_\_init\_\_()  
 self.drop\_prob = drop\_prob # 路径被置零的概率  
  
 def forward(self, x: torch.Tensor) -> torch.Tensor:  
 """前向传播，应用随机深度机制。"""  
 if self.drop\_prob == 0. or not self.training:  
 return x # 如果不丢弃路径，直接返回输入  
 keep\_prob = 1 - self.drop\_prob  
 shape = (x.shape[0], ) + (1, ) \* (x.ndim - 1) # 处理不同维度的张量  
 random\_tensor = keep\_prob + torch.rand(shape, dtype=x.dtype, device=x.device)  
 output = x.div(keep\_prob) \* random\_tensor.floor() # 应用随机丢弃  
 return output  
  
class ConvFFN(nn.Module):  
 """使用卷积模块实现的多层感知机（Feed Forward Network）。"""  
   
 def \_\_init\_\_(self, in\_channels: int, out\_channels: int, hidden\_channels\_scale: float = 4.0, dropout\_rate: float = 0.):  
 super().\_\_init\_\_()  
 hidden\_channels = int(in\_channels \* hidden\_channels\_scale) # 隐藏层通道数  
  
 self.ffn\_layers = nn.Sequential(  
 nn.LayerNorm(in\_channels), # 归一化层  
 nn.Conv2d(in\_channels, hidden\_channels, kernel\_size=1), # 1x1卷积  
 nn.ReLU(), # 激活函数  
 nn.Dropout(dropout\_rate), # Dropout层  
 nn.Conv2d(hidden\_channels, out\_channels, kernel\_size=1), # 1x1卷积  
 nn.Dropout(dropout\_rate), # Dropout层  
 )  
  
 def forward(self, x):  
 """前向传播，应用FFN层。"""  
 return self.ffn\_layers(x)  
  
class PKIBlock(nn.Module):  
 """多核Inception模块，结合了卷积和注意力机制。"""  
   
 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) # 前馈网络  
  
 def forward(self, x):  
 """前向传播，应用Inception模块。"""  
 x = self.conv1(x)  
 x = self.conv2(x)  
 x = self.ffn(x)  
 return x  
  
class PKINet(nn.Module):  
 """多核Inception网络的实现。"""  
   
 def \_\_init\_\_(self):  
 super().\_\_init\_\_()  
 self.stem = nn.Conv2d(3, 32, kernel\_size=3, stride=2, padding=1) # Stem层  
 self.block1 = PKIBlock(32, 64) # 第一个Inception块  
 self.block2 = PKIBlock(64, 128) # 第二个Inception块  
  
 def forward(self, x):  
 """前向传播，依次通过Stem层和多个Inception块。"""  
 x = self.stem(x)  
 x = self.block1(x)  
 x = self.block2(x)  
 return x  
  
def PKINET\_T():  
 """创建并返回一个T型的PKINet模型。"""  
 return PKINet()  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 model = PKINET\_T() # 实例化模型  
 inputs = torch.randn((1, 3, 640, 640)) # 创建输入张量  
 res = model(inputs) # 通过模型进行前向传播  
 print(res.size()) # 输出结果的尺寸  
```  
  
### 代码说明：  
1. \*\*DropPath\*\*: 实现了随机深度的机制，能够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\*\*: 实现了一个简单的前馈网络，包含了层归一化、卷积层和激活函数。  
3. \*\*PKIBlock\*\*: 这是一个多核Inception模块，结合了多个卷积层和前馈网络，旨在提取特征。  
4. \*\*PKINet\*\*: 这是整个网络的主体，包含了Stem层和多个Inception块，负责处理输入并生成输出。  
5. \*\*PKINET\_T\*\*: 用于创建一个特定配置的PKINet模型。  
  
通过这些核心模块的组合，构建了一个具有多层特征提取能力的深度学习模型。```

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。该模型的结构基于多种卷积模块和注意力机制，旨在提高图像特征提取的能力。以下是对代码的详细说明。  
  
首先，文件导入了一些必要的库，包括 `math`、`torch` 和 `torch.nn`，以及一些可选的模块。通过 `try-except` 语句，程序尝试导入来自 `mmcv` 和 `mmengine` 的一些组件，这些组件通常用于构建和训练深度学习模型。如果导入失败，则使用 PyTorch 的基本模块。  
  
接下来，定义了一些实用函数和类。`drop\_path` 函数实现了随机深度（Stochastic Depth）机制，用于在训练过程中随机丢弃某些路径，以增强模型的泛化能力。`DropPath` 类则封装了这一功能，使其可以作为一个可训练的模块。  
  
`autopad` 函数用于自动计算卷积层的填充，以确保输出尺寸与输入尺寸相符。`make\_divisible` 函数确保通道数是可被指定除数整除的，常用于模型设计中以优化计算效率。  
  
接下来的几个类（如 `BCHW2BHWC` 和 `BHWC2BCHW`）用于处理张量的维度转换，以适应不同的卷积操作。`GSiLU` 类实现了一种新的激活函数，结合了全局平均池化和 Sigmoid 激活。  
  
`CAA` 类实现了上下文锚点注意力机制，通过多个卷积层和池化层来计算注意力因子。`ConvFFN` 类则实现了一个多层感知机，使用卷积模块来替代传统的全连接层。  
  
`Stem` 和 `DownSamplingLayer` 类分别定义了模型的初始层和下采样层，后者用于减小特征图的尺寸并增加通道数。`InceptionBottleneck` 类实现了带有多个卷积核的瓶颈结构，增强了模型的特征提取能力。  
  
`PKIBlock` 类是一个核心模块，结合了前述的各个组件，构成了多核的 Inception 结构。`PKIStage` 类则将多个 `PKIBlock` 组合在一起，形成模型的一个阶段。  
  
`PKINet` 类是整个模型的主体，包含多个阶段，并定义了模型的架构设置。该类的构造函数接受多个参数，包括模型的架构类型、输出索引、随机丢弃率等。根据不同的架构设置，模型会动态构建相应的层和模块。  
  
最后，文件提供了三个函数 `PKINET\_T`、`PKINET\_S` 和 `PKINET\_B`，分别用于创建不同规模的 PKINet 模型。`\_\_main\_\_` 部分用于测试模型的创建和前向传播，生成随机输入并输出各层的尺寸。  
  
整体而言，这个文件实现了一个复杂的深度学习模型，采用了多种现代卷积神经网络的设计理念，旨在提升图像处理任务的性能。

```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  
from functools import partial  
  
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 # 输出特征数默认为输入特征数  
 hidden\_features = hidden\_features or in\_features # 隐藏层特征数默认为输入特征数  
 self.fc1 = nn.Conv2d(in\_features, hidden\_features, 1) # 第一层卷积  
 self.dwconv = DWConv(hidden\_features) # 深度卷积  
 self.act = act\_layer() # 激活函数  
 self.fc2 = nn.Conv2d(hidden\_features, out\_features, 1) # 第二层卷积  
 self.drop = nn.Dropout(drop) # Dropout层  
  
 def forward(self, x):  
 x = self.fc1(x) # 通过第一层卷积  
 x = self.dwconv(x) # 通过深度卷积  
 x = self.act(x) # 激活  
 x = self.drop(x) # Dropout  
 x = self.fc2(x) # 通过第二层卷积  
 x = self.drop(x) # 再次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() # 保留输入以便后续残差连接  
 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):  
 """网络的基本块，包括注意力和MLP"""  
 def \_\_init\_\_(self, dim, mlp\_ratio=4., drop=0., drop\_path=0., act\_layer=nn.GELU):  
 super().\_\_init\_\_()  
 self.norm1 = nn.BatchNorm2d(dim) # 第一层归一化  
 self.norm2 = nn.BatchNorm2d(dim) # 第二层归一化  
 self.attn = Attention(dim) # 注意力模块  
 self.drop\_path = nn.Identity() if drop\_path <= 0. else DropPath(drop\_path) # 随机深度  
 mlp\_hidden\_dim = int(dim \* mlp\_ratio) # MLP隐藏层维度  
 self.mlp = Mlp(in\_features=dim, hidden\_features=mlp\_hidden\_dim, act\_layer=act\_layer, drop=drop) # MLP模块  
  
 def forward(self, x):  
 x = x + self.drop\_path(self.attn(self.norm1(x))) # 通过注意力模块并添加残差  
 x = x + self.drop\_path(self.mlp(self.norm2(x))) # 通过MLP模块并添加残差  
 return x  
  
class LSKNet(nn.Module):  
 """LSKNet模型"""  
 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(embed\_dims) # 网络阶段数  
 for i in range(self.num\_stages):  
 # 创建每个阶段的嵌入层和块  
 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])]) # 创建多个Block  
 setattr(self, f"patch\_embed{i + 1}", patch\_embed) # 将嵌入层添加到模型中  
 setattr(self, f"block{i + 1}", block) # 将Block添加到模型中  
  
 def forward(self, x):  
 outs = []  
 for i in range(self.num\_stages):  
 patch\_embed = getattr(self, f"patch\_embed{i + 1}") # 获取嵌入层  
 block = getattr(self, f"block{i + 1}") # 获取Block  
 x, H, W = patch\_embed(x) # 嵌入输入  
 for blk in block:  
 x = blk(x) # 通过每个Block  
 outs.append(x) # 保存输出  
 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  
  
 def forward(self, x):  
 return self.dwconv(x) # 通过深度卷积  
  
# 定义模型的构造函数  
def lsknet\_t(weights=''):  
 model = LSKNet(embed\_dims=[32, 64, 160, 256], depths=[3, 3, 5, 2]) # 创建LSKNet模型  
 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') # 实例化模型并加载权重  
 inputs = torch.randn((1, 3, 640, 640)) # 创建随机输入  
 for i in model(inputs):  
 print(i.size()) # 打印每个阶段的输出尺寸  
```  
  
### 代码核心部分解释：  
1. \*\*Mlp类\*\*：实现了一个多层感知机模块，包含两个卷积层和一个深度卷积层，使用GELU激活函数和Dropout。  
2. \*\*Attention类\*\*：实现了一个注意力机制模块，包含两个投影层和一个空间门控单元（LSKblock）。  
3. \*\*Block类\*\*：构建了一个基本的网络块，结合了注意力模块和MLP，使用BatchNorm进行归一化。  
4. \*\*LSKNet类\*\*：构建了整个网络架构，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，用于在MLP中进行特征提取。  
6. \*\*lsknet\_t函数\*\*：用于创建LSKNet模型并加载预训练权重。  
  
通过这些核心模块的组合，构成了一个完整的神经网络模型。```

这个程序文件 `lsknet.py` 实现了一个名为 LSKNet 的深度学习模型，主要用于图像处理任务。代码中使用了 PyTorch 框架，定义了多个类和函数来构建和管理模型的结构和功能。  
  
首先，文件导入了必要的库，包括 PyTorch 的核心模块和一些辅助函数。接着，定义了几个重要的类。  
  
`Mlp` 类实现了一个多层感知机（MLP），它包含两个卷积层和一个深度卷积层（DWConv），并使用 GELU 激活函数和 Dropout 层。这个类的 `forward` 方法定义了数据流经 MLP 的过程。  
  
`LSKblock` 类是模型的一个核心组件，包含多个卷积层和注意力机制。它通过空间卷积和通道卷积提取特征，并使用 sigmoid 函数生成的权重对特征进行加权组合。这个类的 `forward` 方法实现了特征的处理和加权。  
  
`Attention` 类实现了一个注意力机制，包含两个卷积层和一个 LSKblock。它通过对输入进行投影、激活和注意力计算，增强了特征的表达能力。  
  
`Block` 类结合了归一化、注意力机制和 MLP，构成了模型的基本构建块。它在 `forward` 方法中使用了残差连接和随机丢弃路径（DropPath）来增强模型的鲁棒性。  
  
`OverlapPatchEmbed` 类负责将输入图像转换为补丁嵌入。它使用卷积层将图像分割成多个补丁，并进行归一化处理。  
  
`LSKNet` 类是整个模型的主类，定义了模型的结构，包括不同阶段的嵌入、块和归一化层。它在初始化时根据输入参数设置各个阶段的特征维度、深度和其他超参数。`forward` 方法负责将输入数据通过各个阶段进行处理，并返回不同阶段的输出。  
  
`DWConv` 类实现了深度卷积操作，作为特征提取的一部分。  
  
`update\_weight` 函数用于更新模型的权重，确保加载的权重与模型的结构匹配。  
  
最后，`lsknet\_t` 和 `lsknet\_s` 函数分别定义了两种不同配置的 LSKNet 模型，并提供了加载预训练权重的功能。  
  
在文件的最后部分，提供了一个简单的测试代码，创建了一个 LSKNet 模型实例，并对随机生成的输入数据进行前向传播，打印输出的尺寸。这部分代码可以用于验证模型的基本功能和结构是否正确。  
  
整体来看，这个文件实现了一个复杂的深度学习模型，结合了多种现代神经网络设计理念，如注意力机制、残差连接和多层感知机，适合用于图像分类、目标检测等任务。

```以下是经过简化和注释的核心代码部分：  
  
```python  
import torch  
import torch.nn.functional as F  
  
def selective\_scan\_easy(us, dts, As, Bs, Cs, Ds, delta\_bias=None, delta\_softplus=False, return\_last\_state=False, chunksize=64):  
 """  
 执行选择性扫描操作。  
   
 参数:  
 us: 输入张量，形状为 (B, G \* D, L)  
 dts: 时间增量张量，形状为 (B, G \* D, L)  
 As: 系数矩阵，形状为 (G \* D, N)  
 Bs: 系数矩阵，形状为 (B, G, N, L)  
 Cs: 系数矩阵，形状为 (B, G, N, L)  
 Ds: 系数矩阵，形状为 (G \* D)  
 delta\_bias: 偏置项，形状为 (G \* D)  
 delta\_softplus: 布尔值，指示是否应用 softplus 激活  
 return\_last\_state: 布尔值，指示是否返回最后的状态  
 chunksize: 每次处理的序列长度  
   
 返回:  
 输出张量和（可选）最后状态  
 """  
   
 def selective\_scan\_chunk(us, dts, As, Bs, Cs, hprefix):  
 """  
 处理每个块的选择性扫描。  
   
 参数:  
 us: 输入张量块  
 dts: 时间增量块  
 As, Bs, Cs: 系数矩阵  
 hprefix: 前一个状态  
   
 返回:  
 当前块的输出和状态  
 """  
 ts = dts.cumsum(dim=0) # 计算时间增量的累积和  
 Ats = torch.einsum("gdn,lbgd->lbgdn", As, ts).exp() # 计算指数  
 rAts = Ats # 归一化  
 duts = dts \* us # 计算增量  
 dtBus = torch.einsum("lbgd,lbgn->lbgdn", duts, Bs) # 计算中间结果  
 hs\_tmp = rAts \* (dtBus / rAts).cumsum(dim=0) # 计算状态  
 hs = hs\_tmp + Ats \* hprefix.unsqueeze(0) # 更新状态  
 ys = torch.einsum("lbgn,lbgdn->lbgd", Cs, hs) # 计算输出  
 return ys, hs  
  
 # 数据类型设置  
 dtype = torch.float32  
 inp\_dtype = us.dtype  
 has\_D = Ds is not None  
 if chunksize < 1:  
 chunksize = Bs.shape[-1] # 如果 chunksize 小于 1，使用 L  
  
 # 处理输入数据  
 dts = dts.to(dtype)  
 if delta\_bias is not None:  
 dts = dts + delta\_bias.view(1, -1, 1).to(dtype) # 添加偏置  
 if delta\_softplus:  
 dts = F.softplus(dts) # 应用 softplus 激活  
  
 # 调整输入张量的形状  
 Bs = Bs.unsqueeze(1) if len(Bs.shape) == 3 else Bs  
 Cs = Cs.unsqueeze(1) if len(Cs.shape) == 3 else Cs  
 B, G, N, L = Bs.shape  
 us = us.view(B, G, -1, L).permute(3, 0, 1, 2).to(dtype)  
 dts = dts.view(B, G, -1, L).permute(3, 0, 1, 2).to(dtype)  
 As = As.view(G, -1, N).to(dtype)  
 Bs = Bs.permute(3, 0, 1, 2).to(dtype)  
 Cs = Cs.permute(3, 0, 1, 2).to(dtype)  
 Ds = Ds.view(G, -1).to(dtype) if has\_D else None  
  
 oys = [] # 存储输出  
 hprefix = us.new\_zeros((B, G, D, N), dtype=dtype) # 初始化状态  
 for i in range(0, L, chunksize):  
 ys, hs = selective\_scan\_chunk(  
 us[i:i + chunksize], dts[i:i + chunksize],   
 As, Bs[i:i + chunksize], Cs[i:i + chunksize], hprefix,   
 )  
 oys.append(ys) # 收集输出  
 hprefix = hs[-1] # 更新状态  
  
 oys = torch.cat(oys, dim=0) # 合并输出  
 if has\_D:  
 oys = oys + Ds \* us # 添加 D 的影响  
 oys = oys.permute(1, 2, 3, 0).view(B, -1, L) # 调整输出形状  
  
 return oys.to(inp\_dtype) if not return\_last\_state else (oys.to(inp\_dtype), hprefix.view(B, G \* D, N).float())  
```  
  
### 代码说明  
1. \*\*selective\_scan\_easy\*\*: 该函数实现了选择性扫描的核心逻辑，输入为多个张量，输出为经过处理的张量。  
2. \*\*selective\_scan\_chunk\*\*: 这是一个内部函数，用于处理每个块的输入数据，计算输出和状态。  
3. \*\*数据处理\*\*: 在函数开始时，输入数据会被转换为适当的形状和数据类型，确保计算的正确性。  
4. \*\*循环处理\*\*: 使用循环遍历输入数据的块，逐步更新输出和状态。  
5. \*\*返回值\*\*: 最后，函数返回处理后的输出张量和（可选的）最后状态。  
  
这个简化版本保留了核心逻辑，同时通过注释解释了每个部分的功能。```

这个程序文件 `test\_selective\_scan\_easy.py` 实现了一个名为 `selective\_scan\_easy` 的函数，并通过一系列的测试用例来验证其正确性。该函数主要用于处理序列数据的选择性扫描，涉及到深度学习中的张量运算和反向传播。以下是对代码的详细说明。  
  
首先，文件导入了一些必要的库，包括 `torch`、`pytest` 和 `einops`。这些库提供了张量操作、测试框架和张量重排功能。  
  
接下来，定义了 `selective\_scan\_easy` 函数。该函数的输入包括多个张量，如 `us`、`dts`、`As`、`Bs`、`Cs` 和 `Ds`，它们分别代表不同的参数和状态。函数的核心逻辑在于通过选择性扫描的方式，计算出输出张量 `oys` 和最后的状态 `hprefix`。  
  
在 `selective\_scan\_easy` 函数内部，定义了一个嵌套函数 `selective\_scan\_chunk`，该函数负责处理数据的分块计算。它使用了张量的累积和和指数运算来更新状态，并计算输出。通过对输入张量的维度和形状进行调整，确保它们符合计算要求。  
  
函数的参数中还包括一些可选项，如 `delta\_bias` 和 `delta\_softplus`，用于调整输入数据。计算过程中，如果 `delta\_bias` 不为 `None`，则会将其添加到 `dts` 中；如果 `delta\_softplus` 为 `True`，则会对 `dts` 应用 softplus 激活函数。  
  
函数的输出包括计算得到的 `oys` 和（可选的）最后状态 `hprefix`。这些输出经过必要的形状调整，以确保它们符合预期的格式。  
  
接下来，定义了一个 `SelectiveScanEasy` 类，继承自 `torch.autograd.Function`，用于实现自定义的前向和反向传播。这个类的 `forward` 方法实现了选择性扫描的前向计算，而 `backward` 方法则实现了反向传播的梯度计算。反向传播中，使用了保存的中间结果来计算各个输入的梯度。  
  
在文件的后半部分，定义了一些辅助函数，如 `selective\_scan\_easy\_fwdbwd` 和 `selective\_scan\_ref`，用于不同版本的选择性扫描实现和参考实现。它们提供了与 `selective\_scan\_easy` 类似的功能，但可能在实现细节上有所不同。  
  
最后，使用 `pytest` 框架定义了一系列测试用例，确保 `selective\_scan\_easy` 函数的输出与参考实现的输出一致。测试用例涵盖了多种输入情况，包括不同的张量维度、是否使用偏置、是否使用 softplus 等。通过这些测试，可以验证函数的正确性和鲁棒性。  
  
总的来说，这个程序文件实现了一个复杂的选择性扫描算法，并通过自动微分和测试框架确保其正确性，适用于深度学习中的序列数据处理任务。

### 整体功能和构架概括  
  
在这个工程中，主要实现了多个深度学习模型和相关功能模块，旨在提高图像处理和序列数据处理的能力。以下是各个文件的整体功能和构架概述：  
  
1. \*\*kernel\_warehouse.py\*\*: 实现了一个灵活的卷积层管理系统，允许动态管理和分配卷积核，适用于深度学习模型中的卷积操作。该文件中定义了多个卷积层和注意力机制的实现，增强了模型的特征提取能力。  
  
2. \*\*pkinet.py\*\*: 实现了 PKINet 模型，结合了多种卷积模块和注意力机制，旨在提升图像处理任务的性能。模型结构复杂，采用了现代卷积神经网络设计理念，适合用于图像分类和目标检测等任务。  
  
3. \*\*lsknet.py\*\*: 实现了 LSKNet 模型，采用了多层感知机、注意力机制和深度卷积等结构，旨在提高图像特征提取的能力。该模型同样适用于图像处理任务，具有良好的灵活性和扩展性。  
  
4. \*\*test\_selective\_scan\_easy.py\*\*: 实现了选择性扫描算法的测试和验证，确保该算法在序列数据处理中的正确性。通过自定义的前向和反向传播实现了选择性扫描，并使用 pytest 框架进行了一系列的测试用例，确保算法的鲁棒性。  
  
### 文件功能整理表  
  
| 文件名 | 功能描述 |  
|---------------------------------|----------------------------------------------------------------------------------------------|  
| `kernel\_warehouse.py` | 实现卷积层的动态管理和分配，包含卷积层、注意力机制等，增强特征提取能力。 |  
| `pkinet.py` | 实现 PKINet 模型，结合多种卷积模块和注意力机制，适用于图像分类和目标检测等任务。 |  
| `lsknet.py` | 实现 LSKNet 模型，采用多层感知机和注意力机制，提升图像特征提取能力，适合图像处理任务。 |  
| `test\_selective\_scan\_easy.py` | 实现选择性扫描算法的测试和验证，确保算法在序列数据处理中的正确性，使用 pytest 进行测试。 |  
  
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。

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。