# 改进yolo11-EfficientHead等200+全套创新点大全：快递包装类型检测系统源码＆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  
  
随着电子商务的迅猛发展，快递行业的业务量不断攀升，快递包装的种类和形态也日益多样化。这一现象不仅给快递物流的管理带来了挑战，也对包装的分类、回收和再利用提出了更高的要求。传统的人工分类方法不仅效率低下，而且容易出现错误，无法满足现代快递行业对快速、准确处理的需求。因此，开发一种高效的自动化快递包装类型检测系统显得尤为重要。  
  
本研究旨在基于改进的YOLOv11模型，构建一个能够准确识别和分类快递包装类型的系统。YOLO（You Only Look Once）系列模型因其快速的检测速度和较高的准确率，已成为目标检测领域的主流方法。通过对YOLOv11的改进，我们期望能够进一步提升其在快递包装检测中的性能，尤其是在处理复杂背景和多样化包装形态时的鲁棒性。  
  
在本项目中，我们使用了包含1200张图像的数据集，涵盖了四种快递包装类型：完整的箱子（Box）、破损的箱子（Box\_broken）、打开的包裹（Open\_package）以及普通包裹（Package）。这些数据不仅为模型的训练提供了丰富的样本，也为其在实际应用中的推广奠定了基础。通过对这些不同类型的快递包装进行有效的检测与分类，我们希望能够为快递行业的智能化发展提供有力支持。  
  
综上所述，基于改进YOLOv11的快递包装类型检测系统，不仅能够提升快递物流的处理效率，降低人工成本，还能为包装的回收和再利用提供数据支持，推动绿色物流的发展。这一研究不仅具有重要的学术价值，也在实际应用中具有广泛的社会意义。

## 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“Detection\_box”，旨在为改进YOLOv11的快递包装类型检测系统提供支持。该数据集专注于快递包装的多样性与复杂性，包含四个主要类别，分别为“Box”、“Box\_broken”、“Open\_package”和“Package”。这些类别的选择不仅反映了快递行业中常见的包装状态，还涵盖了不同的包装形式，以便于系统能够准确识别和分类各种快递包装情况。  
  
在数据集的构建过程中，我们确保了样本的多样性和代表性。每个类别均包含大量的图像样本，涵盖了不同的拍摄角度、光照条件和背景环境。这种多样性使得模型在训练过程中能够学习到更为丰富的特征，从而提高其在实际应用中的泛化能力。尤其是在“Box\_broken”和“Open\_package”这两个类别中，我们特别注重收集那些在运输过程中可能出现的损坏或打开状态的包装图像，以增强模型对异常情况的识别能力。  
  
此外，数据集中的图像经过精心标注，确保每个样本的准确性和一致性。这一过程不仅提高了数据集的质量，也为后续的模型训练和评估提供了可靠的基础。通过使用“Detection\_box”数据集，我们期望能够有效提升YOLOv11在快递包装类型检测任务中的性能，使其能够在实际应用中更好地服务于快递行业的需求，减少人工干预，提高工作效率。整体而言，本项目的数据集为实现智能化的快递包装检测系统奠定了坚实的基础。

## 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

```以下是对给定代码的核心部分进行提炼和详细注释的结果。主要保留了模型的核心结构和重要功能，去掉了一些冗余的部分。  
  
```python  
import torch  
import torch.nn as nn  
import torch.nn.functional as F  
from timm.layers import trunc\_normal\_, DropPath, to\_2tuple  
  
class GRNwithNHWC(nn.Module):  
 """ GRN (Global Response Normalization) 层  
 该层用于对输入进行全局响应归一化。  
 输入假设为 (N, H, W, C) 格式。  
 """  
 def \_\_init\_\_(self, dim, use\_bias=True):  
 super().\_\_init\_\_()  
 self.use\_bias = use\_bias  
 self.gamma = nn.Parameter(torch.zeros(1, 1, 1, dim)) # 可学习参数  
 if self.use\_bias:  
 self.beta = nn.Parameter(torch.zeros(1, 1, 1, dim)) # 可学习偏置  
  
 def forward(self, x):  
 Gx = torch.norm(x, p=2, dim=(1, 2), keepdim=True) # 计算L2范数  
 Nx = Gx / (Gx.mean(dim=-1, keepdim=True) + 1e-6) # 归一化  
 if self.use\_bias:  
 return (self.gamma \* Nx + 1) \* x + self.beta # 应用归一化和偏置  
 else:  
 return (self.gamma \* Nx + 1) \* x  
  
class DilatedReparamBlock(nn.Module):  
 """ Dilated Reparam Block  
 该模块用于实现扩张卷积的重参数化。  
 输入假设为 (N, C, H, W) 格式。  
 """  
 def \_\_init\_\_(self, channels, kernel\_size, deploy, use\_sync\_bn=False):  
 super().\_\_init\_\_()  
 self.lk\_origin = nn.Conv2d(channels, channels, kernel\_size=kernel\_size, stride=1,  
 padding=kernel\_size//2, dilation=1, groups=channels, bias=True)  
 self.attempt\_use\_lk\_impl = True # 是否尝试使用高效的卷积实现  
  
 # 定义扩张卷积的参数  
 self.kernel\_sizes = [3, 5] # 示例：可根据需要调整  
 self.dilates = [1, 2] # 示例：可根据需要调整  
  
 if not deploy:  
 self.origin\_bn = nn.BatchNorm2d(channels) # 训练模式下使用BatchNorm  
 for k, r in zip(self.kernel\_sizes, self.dilates):  
 self.\_\_setattr\_\_('dil\_conv\_k{}\_{}'.format(k, r),  
 nn.Conv2d(channels, channels, kernel\_size=k, stride=1,  
 padding=(r \* (k - 1) + 1) // 2, dilation=r, groups=channels,  
 bias=False))  
 self.\_\_setattr\_\_('dil\_bn\_k{}\_{}'.format(k, r), nn.BatchNorm2d(channels))  
  
 def forward(self, x):  
 if not hasattr(self, 'origin\_bn'): # 部署模式  
 return self.lk\_origin(x)  
 out = self.origin\_bn(self.lk\_origin(x)) # 应用原始卷积和BN  
 for k, r in zip(self.kernel\_sizes, self.dilates):  
 conv = self.\_\_getattr\_\_('dil\_conv\_k{}\_{}'.format(k, r))  
 bn = self.\_\_getattr\_\_('dil\_bn\_k{}\_{}'.format(k, r))  
 out = out + bn(conv(x)) # 添加扩张卷积的输出  
 return out  
  
class UniRepLKNetBlock(nn.Module):  
 """ UniRepLKNet中的基本块  
 包含卷积、归一化、激活和Squeeze-and-Excitation模块。  
 """  
 def \_\_init\_\_(self, dim, kernel\_size, drop\_path=0., deploy=False):  
 super().\_\_init\_\_()  
 self.dwconv = DilatedReparamBlock(dim, kernel\_size, deploy=deploy) # 扩张卷积  
 self.norm = nn.BatchNorm2d(dim) # 归一化层  
 self.se = SEBlock(dim, dim // 4) # Squeeze-and-Excitation模块  
  
 ffn\_dim = dim \* 4 # 前馈网络维度  
 self.pwconv1 = nn.Linear(dim, ffn\_dim) # 第一层前馈网络  
 self.act = nn.GELU() # 激活函数  
 self.pwconv2 = nn.Linear(ffn\_dim, dim) # 第二层前馈网络  
  
 def forward(self, inputs):  
 y = self.se(self.norm(self.dwconv(inputs))) # 通过扩张卷积和SE模块  
 y = self.pwconv2(self.act(self.pwconv1(y))) # 前馈网络  
 return y + inputs # 残差连接  
  
class UniRepLKNet(nn.Module):  
 """ UniRepLKNet模型  
 包含多个UniRepLKNetBlock，构成深度神经网络。  
 """  
 def \_\_init\_\_(self, in\_chans=3, num\_classes=1000, depths=(3, 3, 27, 3), dims=(96, 192, 384, 768)):  
 super().\_\_init\_\_()  
 self.stages = nn.ModuleList() # 存储不同阶段的模块  
 for i in range(4):  
 stage = nn.Sequential(  
 \*[UniRepLKNetBlock(dim=dims[i], kernel\_size=3) for \_ in range(depths[i])]  
 )  
 self.stages.append(stage)  
  
 def forward(self, x):  
 for stage in self.stages:  
 x = stage(x) # 逐阶段传递输入  
 return x  
  
# 示例模型创建和前向传播  
if \_\_name\_\_ == '\_\_main\_\_':  
 inputs = torch.randn((1, 3, 640, 640)) # 随机输入  
 model = UniRepLKNet() # 创建模型  
 res = model(inputs) # 前向传播  
 print(res.shape) # 输出结果形状  
```  
  
### 代码核心部分说明：  
1. \*\*GRNwithNHWC\*\*: 实现了全局响应归一化层，用于对输入进行归一化处理。  
2. \*\*DilatedReparamBlock\*\*: 实现了扩张卷积的重参数化，包含多个扩张卷积和BatchNorm层。  
3. \*\*UniRepLKNetBlock\*\*: 该模块是UniRepLKNet的基本构建块，包含扩张卷积、归一化、激活和Squeeze-and-Excitation模块。  
4. \*\*UniRepLKNet\*\*: 整个模型的实现，包含多个阶段，每个阶段由多个UniRepLKNetBlock组成。  
  
以上代码经过简化，保留了核心功能和结构，并添加了详细的中文注释以帮助理解。```

该文件实现了一个名为UniRepLKNet的深度学习模型，主要用于音频、视频、点云、时间序列和图像识别等任务。该模型的设计灵感来源于多个前沿的卷积神经网络架构，包括RepLKNet、ConvNeXt、DINO和DeiT等。文件中包含了一系列的类和函数，构成了模型的基础结构。  
  
首先，文件导入了必要的PyTorch库以及一些辅助函数和模块。接着，定义了多个模块，包括GRN（全局响应归一化）层、NCHW与NHWC格式的转换层、Squeeze-and-Excitation（SE）块等。这些模块的设计旨在提高模型的表达能力和计算效率。  
  
GRN层通过对输入进行归一化处理，增强了模型对特征的响应能力。NCHWtoNHWC和NHWCtoNCHW类则负责在不同的数据格式之间进行转换，以适应不同的卷积实现。SE块通过自适应地调整通道的权重，进一步提升了特征的表达能力。  
  
文件中还定义了一个函数`get\_conv2d`，用于根据输入参数选择合适的卷积实现，支持原生卷积和iGEMM大核卷积的选择。该函数会根据卷积的特性（如核大小、步幅、填充等）决定使用哪种实现方式。  
  
在`DilatedReparamBlock`类中，定义了一个带有扩张卷积的重参数化块，该块通过多个不同核大小和扩张率的卷积层组合，增强了模型对不同尺度特征的捕捉能力。`UniRepLKNetBlock`类则是模型的基本构建单元，结合了卷积、归一化、激活和SE块，形成了一个完整的模块。  
  
`UniRepLKNet`类是整个模型的核心，负责构建模型的不同阶段，包括下采样层和多个UniRepLKNetBlock。模型的输入通道数、类别数、每个阶段的深度和特征维度等都可以通过参数进行配置。模型还支持部署模式和检查点功能，以便在训练和推理时灵活使用。  
  
最后，文件提供了一些函数（如`unireplknet\_a`、`unireplknet\_f`等）用于实例化不同配置的UniRepLKNet模型，并加载预训练权重。主函数部分则展示了如何创建模型实例并进行推理。  
  
总体而言，该文件实现了一个灵活且高效的卷积神经网络架构，适用于多种视觉和音频任务，具有良好的扩展性和可配置性。

```以下是代码中最核心的部分，并附上详细的中文注释：  
  
```python  
import math  
import torch  
import torch.nn as nn  
import torch.nn.functional as F  
  
def get\_conv2d(in\_channels, out\_channels, kernel\_size, stride, padding, dilation, groups, bias):  
 # 创建一个2D卷积层  
 return nn.Conv2d(  
 in\_channels, out\_channels, kernel\_size, stride, padding, dilation, groups, bias  
 )  
  
def get\_bn(channels):  
 # 创建一个批归一化层  
 return nn.BatchNorm2d(channels)  
  
class Mask(nn.Module):  
 def \_\_init\_\_(self, size):  
 super().\_\_init\_\_()  
 # 初始化权重参数，范围在-1到1之间  
 self.weight = torch.nn.Parameter(data=torch.Tensor(\*size), requires\_grad=True)  
 self.weight.data.uniform\_(-1, 1)  
  
 def forward(self, x):  
 # 使用sigmoid函数对权重进行激活  
 w = torch.sigmoid(self.weight)  
 # 将输入x与权重w相乘，进行掩码操作  
 masked\_wt = w.mul(x)  
 return masked\_wt  
  
class LoRAConvsByWeight(nn.Module):  
 def \_\_init\_\_(self, in\_channels, out\_channels, big\_kernel, small\_kernel, stride=1, group=1, bn=True, use\_small\_conv=True):  
 super().\_\_init\_\_()  
 self.kernels = (small\_kernel, big\_kernel) # 小卷积核和大卷积核  
 self.stride = stride  
 self.small\_conv = use\_small\_conv  
 # 计算填充和索引  
 padding, after\_padding\_index, index = self.shift(self.kernels)  
 self.pad = padding, after\_padding\_index, index  
 self.nk = math.ceil(big\_kernel / small\_kernel) # 计算小卷积核的数量  
 out\_n = out\_channels \* self.nk # 输出通道数  
 # 创建分离卷积层  
 self.split\_convs = nn.Conv2d(in\_channels, out\_n, kernel\_size=small\_kernel, stride=stride, padding=padding, groups=group, bias=False)  
  
 # 创建两个掩码层  
 self.lora1 = Mask((1, out\_n, 1, 1))  
 self.lora2 = Mask((1, out\_n, 1, 1))  
 self.use\_bn = bn  
  
 # 如果需要，创建批归一化层  
 if bn:  
 self.bn\_lora1 = get\_bn(out\_channels)  
 self.bn\_lora2 = get\_bn(out\_channels)  
 else:  
 self.bn\_lora1 = None  
 self.bn\_lora2 = None  
  
 def forward(self, inputs):  
 # 前向传播  
 out = self.split\_convs(inputs) # 通过分离卷积层  
 \*\_, ori\_h, ori\_w = inputs.shape # 获取输入的高度和宽度  
 # 分别通过两个掩码层  
 lora1\_x = self.forward\_lora(self.lora1(out), ori\_h, ori\_w, VH='H', bn=self.bn\_lora1)  
 lora2\_x = self.forward\_lora(self.lora2(out), ori\_h, ori\_w, VH='W', bn=self.bn\_lora2)  
 x = lora1\_x + lora2\_x # 合并结果  
 return x  
  
 def forward\_lora(self, out, ori\_h, ori\_w, VH='H', bn=None):  
 # 处理输出，按组进行数据重排  
 b, c, h, w = out.shape  
 out = torch.split(out.reshape(b, -1, self.nk, h, w), 1, 2) # 将输出按组分割  
 x = 0  
 for i in range(self.nk):  
 outi = self.rearrange\_data(out[i], i, ori\_h, ori\_w, VH) # 重排数据  
 x = x + outi # 累加结果  
 if self.use\_bn:  
 x = bn(x) # 如果需要，进行批归一化  
 return x  
  
 def rearrange\_data(self, x, idx, ori\_h, ori\_w, VH):  
 # 根据索引重排数据  
 padding, \_, index = self.pad  
 x = x.squeeze(2) # 去掉维度  
 \*\_, h, w = x.shape  
 k = min(self.kernels)  
 ori\_k = max(self.kernels)  
 ori\_p = ori\_k // 2  
 stride = self.stride  
 # 计算填充和起始点  
 if (idx + 1) >= index:  
 pad\_l = 0  
 s = (idx + 1 - index) \* (k // stride)  
 else:  
 pad\_l = (index - 1 - idx) \* (k // stride)  
 s = 0  
 if VH == 'H':  
 suppose\_len = (ori\_w + 2 \* ori\_p - ori\_k) // stride + 1  
 pad\_r = 0 if (s + suppose\_len) <= (w + pad\_l) else s + suppose\_len - w - pad\_l  
 new\_pad = (pad\_l, pad\_r, 0, 0)  
 dim = 3  
 else:  
 suppose\_len = (ori\_h + 2 \* ori\_p - ori\_k) // stride + 1  
 pad\_r = 0 if (s + suppose\_len) <= (h + pad\_l) else s + suppose\_len - h - pad\_l  
 new\_pad = (0, 0, pad\_l, pad\_r)  
 dim = 2  
 if len(set(new\_pad)) > 1:  
 x = F.pad(x, new\_pad) # 填充  
 if padding \* 2 + 1 != k:  
 pad = padding - k // 2  
 if VH == 'H':  
 x = torch.narrow(x, 2, pad, h - 2 \* pad)  
 else:  
 x = torch.narrow(x, 3, pad, w - 2 \* pad)  
  
 xs = torch.narrow(x, dim, s, suppose\_len) # 根据维度切片  
 return xs  
  
 def shift(self, kernels):  
 # 计算填充和索引  
 mink, maxk = min(kernels), max(kernels)  
 mid\_p = maxk // 2  
 offset\_idx\_left = mid\_p % mink  
 offset\_idx\_right = (math.ceil(maxk / mink) \* mink - mid\_p - 1) % mink  
 padding = offset\_idx\_left % mink  
 while padding < offset\_idx\_right:  
 padding += mink  
 while padding < (mink - 1):  
 padding += mink  
 after\_padding\_index = padding - offset\_idx\_left  
 index = math.ceil((mid\_p + 1) / mink)  
 real\_start\_idx = index - after\_padding\_index // mink  
 return padding, after\_padding\_index, real\_start\_idx  
  
class ReparamLargeKernelConv(nn.Module):  
 def \_\_init\_\_(self, in\_channels, out\_channels, kernel\_size, small\_kernel=5, stride=1, groups=1, small\_kernel\_merged=False, Decom=True, bn=True):  
 super(ReparamLargeKernelConv, self).\_\_init\_\_()  
 self.kernel\_size = kernel\_size  
 self.small\_kernel = small\_kernel  
 self.Decom = Decom  
 padding = kernel\_size // 2 # 填充  
 if small\_kernel\_merged:  
 self.lkb\_reparam = get\_conv2d(in\_channels, out\_channels, kernel\_size, stride, padding, 1, groups, True)  
 else:  
 if self.Decom:  
 self.LoRA = conv\_bn(in\_channels, out\_channels, (kernel\_size, small\_kernel), stride, padding, 1, bn=bn)  
 else:  
 self.lkb\_origin = conv\_bn(in\_channels, out\_channels, kernel\_size, stride, padding, 1, bn=bn)  
  
 if (small\_kernel is not None) and small\_kernel < kernel\_size:  
 self.small\_conv = conv\_bn(in\_channels, out\_channels, small\_kernel, stride, small\_kernel // 2, groups, bn=bn)  
  
 self.bn = get\_bn(out\_channels) # 批归一化  
 self.act = nn.SiLU() # 激活函数  
  
 def forward(self, inputs):  
 # 前向传播  
 if hasattr(self, "lkb\_reparam"):  
 out = self.lkb\_reparam(inputs)  
 elif self.Decom:  
 out = self.LoRA(inputs)  
 if hasattr(self, "small\_conv"):  
 out += self.small\_conv(inputs)  
 else:  
 out = self.lkb\_origin(inputs)  
 if hasattr(self, "small\_conv"):  
 out += self.small\_conv(inputs)  
 return self.act(self.bn(out)) # 激活和归一化  
  
 def get\_equivalent\_kernel\_bias(self):  
 # 获取等效的卷积核和偏置  
 eq\_k, eq\_b = fuse\_bn(self.lkb\_origin.conv, self.lkb\_origin.bn)  
 if hasattr(self, "small\_conv"):  
 small\_k, small\_b = fuse\_bn(self.small\_conv.conv, self.small\_conv.bn)  
 eq\_b += small\_b  
 eq\_k += nn.functional.pad(small\_k, [(self.kernel\_size - self.small\_kernel) // 2] \* 4)  
 return eq\_k, eq\_b  
  
 def switch\_to\_deploy(self):  
 # 切换到部署模式  
 if hasattr(self, 'lkb\_origin'):  
 eq\_k, eq\_b = self.get\_equivalent\_kernel\_bias()  
 self.lkb\_reparam = get\_conv2d(self.lkb\_origin.conv.in\_channels, self.lkb\_origin.conv.out\_channels, self.lkb\_origin.conv.kernel\_size, self.lkb\_origin.conv.stride, self.lkb\_origin.conv.padding, self.lkb\_origin.conv.dilation, self.lkb\_origin.conv.groups, True)  
 self.lkb\_reparam.weight.data = eq\_k  
 self.lkb\_reparam.bias.data = eq\_b  
 self.\_\_delattr\_\_("lkb\_origin")  
 if hasattr(self, "small\_conv"):  
 self.\_\_delattr\_\_("small\_conv")  
```  
  
### 代码说明：  
1. \*\*卷积层和批归一化层的创建\*\*：`get\_conv2d`和`get\_bn`函数用于创建卷积层和批归一化层。  
2. \*\*掩码类\*\*：`Mask`类用于创建一个可学习的掩码，通过sigmoid激活来调整输入特征。  
3. \*\*LoRA卷积\*\*：`LoRAConvsByWeight`类实现了LoRA（Low-Rank Adaptation）卷积，通过两个掩码和分离卷积层进行特征处理。  
4. \*\*重参数化卷积\*\*：`ReparamLargeKernelConv`类实现了重参数化的大卷积核，通过组合小卷积核和大卷积核来提高计算效率。  
5. \*\*前向传播\*\*：每个类都有`forward`方法来定义数据的前向传播过程。  
6. \*\*等效卷积核和偏置的获取\*\*：`get\_equivalent\_kernel\_bias`方法用于获取经过批归一化后的等效卷积核和偏置。  
7. \*\*切换到部署模式\*\*：`switch\_to\_deploy`方法用于将模型切换到部署模式，优化计算效率。```

这个程序文件 `shiftwise\_conv.py` 实现了一个自定义的卷积层，主要用于处理大卷积核和小卷积核的组合。代码中使用了 PyTorch 框架，定义了一些辅助函数和类，以便于构建和使用这些卷积层。  
  
首先，文件中导入了必要的库，包括 `math` 和 `torch` 以及 PyTorch 的神经网络模块 `torch.nn` 和功能模块 `torch.nn.functional`。接着，定义了一个函数 `get\_conv2d`，用于创建一个标准的二维卷积层，允许指定输入和输出通道数、卷积核大小、步幅、填充、扩张率、分组和是否使用偏置等参数。  
  
`get\_bn` 函数用于创建一个批归一化层，接受通道数作为参数。  
  
接下来，定义了一个 `Mask` 类，它是一个可学习的参数，使用 sigmoid 函数对其进行激活，并与输入进行逐元素相乘，产生一个掩码效果。  
  
`conv\_bn\_ori` 函数构建一个包含卷积层和可选的批归一化层的序列模型。这个函数会根据输入参数创建相应的卷积层，并在需要时添加批归一化层。  
  
`LoRAConvsByWeight` 类实现了一个特殊的卷积结构，旨在通过权重合并不同的卷积核。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、步幅、分组等。该类的 `forward` 方法会执行卷积操作，并通过 `forward\_lora` 方法对输出进行处理，使用掩码和重排数据的方式来实现卷积的效果。  
  
`rearrange\_data` 方法用于根据输入的参数调整数据的形状和填充，以便在不同的方向上进行卷积操作。`shift` 方法则计算卷积操作所需的填充和索引，以确保特征图的大小不变。  
  
`conv\_bn` 函数根据卷积核的大小选择使用 `conv\_bn\_ori` 或 `LoRAConvsByWeight` 进行卷积操作。  
  
`fuse\_bn` 函数用于将卷积层和批归一化层合并为一个等效的卷积层，以便在推理时提高效率。  
  
最后，`ReparamLargeKernelConv` 类是整个文件的核心，负责实现大卷积核的重参数化。它的构造函数根据输入参数设置卷积层、批归一化层和激活函数。`forward` 方法执行前向传播，返回经过激活的输出。`get\_equivalent\_kernel\_bias` 方法用于获取等效的卷积核和偏置，而 `switch\_to\_deploy` 方法则将模型切换到推理模式，合并卷积和批归一化层。  
  
整体来看，这个文件实现了一个灵活且高效的卷积层设计，能够处理不同大小的卷积核，并通过重参数化技术提高模型的推理性能。

```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  
from functools import partial  
  
class Mlp(nn.Module):  
 """ 多层感知机模块 """  
 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) # 1x1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) # 1x1卷积层  
 self.drop = nn.Dropout(drop) # Dropout层  
  
 def forward(self, x):  
 """ 前向传播 """  
 x = self.fc1(x)  
 x = self.dwconv(x)  
 x = self.act(x)  
 x = self.drop(x)  
 x = self.fc2(x)  
 x = self.drop(x)  
 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) # 1x1卷积  
 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) # 1x1卷积  
  
 def forward(self, x):  
 """ 前向传播 """  
 shortcut = x.clone() # 保留输入用于残差连接  
 x = self.proj\_1(x)  
 x = self.activation(x)  
 x = self.spatial\_gating\_unit(x)  
 x = self.proj\_2(x)  
 x = x + shortcut # 残差连接  
 return x  
  
  
class Block(nn.Module):  
 """ 基本块，包括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) # 批归一化  
 self.norm2 = nn.BatchNorm2d(dim) # 批归一化  
 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) # 多层感知机  
  
 def forward(self, x):  
 """ 前向传播 """  
 x = x + self.attn(self.norm1(x)) # 添加注意力模块的输出  
 x = x + self.mlp(self.norm2(x)) # 添加多层感知机的输出  
 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(embed\_dims) # 网络阶段数  
 for i in range(self.num\_stages):  
 # 初始化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):  
 """ 前向传播 """  
 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  
  
 def forward(self, x):  
 """ 前向传播 """  
 return self.dwconv(x)  
  
  
def lsknet\_t():  
 """ 创建LSKNet的一个变体 """  
 return LSKNet(embed\_dims=[32, 64, 160, 256], depths=[3, 3, 5, 2])  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 model = lsknet\_t() # 实例化模型  
 inputs = torch.randn((1, 3, 640, 640)) # 创建随机输入  
 for i in model(inputs):  
 print(i.size()) # 输出每个阶段的输出尺寸  
```  
  
### 代码说明：  
1. \*\*Mlp类\*\*：实现了一个多层感知机，包含两个1x1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，包含两个1x1卷积和一个空间门控单元（LSKblock）。  
3. \*\*Block类\*\*：将注意力和多层感知机结合在一起，形成一个基本的网络块。  
4. \*\*LSKNet类\*\*：构建整个网络结构，包含多个阶段，每个阶段有嵌入层和多个块。  
5. \*\*DWConv类\*\*：实现了深度卷积操作。  
6. \*\*lsknet\_t函数\*\*：用于创建LSKNet的一个特定变体。  
7. \*\*主程序\*\*：实例化模型并测试输入的输出尺寸。```

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。该模型基于 PyTorch 框架构建，包含多个模块和类，具体功能如下：  
  
首先，导入了必要的库，包括 PyTorch、神经网络模块、一些实用工具以及 NumPy。然后，定义了一个 Mlp 类，这是一个多层感知机，包含两个卷积层和一个深度卷积层，使用 GELU 激活函数，并且支持 dropout。  
  
接下来，定义了 LSKblock 类，它是模型的核心部分之一。该类通过多个卷积操作提取特征，并使用注意力机制来增强特征表示。它首先通过深度卷积和空间卷积生成特征，然后通过 squeeze 操作和 sigmoid 激活函数生成注意力权重，最后将加权特征与输入特征相乘。  
  
Attention 类则实现了一个简单的注意力机制，包含两个卷积层和一个 LSKblock。它通过残差连接将输入与经过处理的特征相加，增强了模型的学习能力。  
  
Block 类结合了归一化、注意力机制和多层感知机。它通过残差连接和层级缩放来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。它使用卷积层将图像转换为特征图，并进行归一化处理。  
  
LSKNet 类是整个模型的主体，包含多个阶段，每个阶段由补丁嵌入、多个 Block 和归一化层组成。模型的深度和嵌入维度可以通过参数进行调整。  
  
DWConv 类实现了深度卷积，用于特征提取。  
  
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。  
  
lsknet\_t 和 lsknet\_s 函数分别创建了不同配置的 LSKNet 模型，并可选择加载预训练权重。  
  
最后，在主程序中，创建了一个 LSKNet 模型实例，并对一个随机生成的输入进行前向传播，输出每个阶段的特征图的尺寸。  
  
整体来看，这个程序实现了一个灵活且高效的图像处理模型，适用于各种计算机视觉任务。

``````python  
import sys  
import subprocess  
  
def run\_script(script\_path):  
 """  
 使用当前 Python 环境运行指定的脚本。  
  
 Args:  
 script\_path (str): 要运行的脚本路径  
  
 Returns:  
 None  
 """  
 # 获取当前 Python 解释器的路径  
 python\_path = sys.executable  
  
 # 构建运行命令，使用 streamlit 运行指定的脚本  
 command = f'"{python\_path}" -m streamlit run "{script\_path}"'  
  
 # 执行命令并等待其完成  
 result = subprocess.run(command, shell=True)  
   
 # 检查命令执行的返回码，如果不为0则表示出错  
 if result.returncode != 0:  
 print("脚本运行出错。")  
  
  
# 主程序入口  
if \_\_name\_\_ == "\_\_main\_\_":  
 # 指定要运行的脚本路径  
 script\_path = "web.py" # 这里可以直接指定脚本名，假设在当前目录下  
  
 # 调用函数运行脚本  
 run\_script(script\_path)  
```  
  
### 代码注释说明：  
1. \*\*导入模块\*\*：  
 - `sys`：用于获取当前 Python 解释器的路径。  
 - `subprocess`：用于执行外部命令。  
  
2. \*\*`run\_script` 函数\*\*：  
 - 功能：运行指定路径的 Python 脚本。  
 - 参数：`script\_path`，需要运行的脚本的路径。  
 - 获取当前 Python 解释器的路径，构建命令字符串，使用 `subprocess.run` 执行命令。  
 - 检查命令执行的返回码，如果返回码不为0，表示脚本运行出错，打印错误信息。  
  
3. \*\*主程序入口\*\*：  
 - 使用 `if \_\_name\_\_ == "\_\_main\_\_":` 确保只有在直接运行该脚本时才会执行以下代码。  
 - 指定要运行的脚本路径（这里假设脚本名为 `web.py`）。  
 - 调用 `run\_script` 函数，传入脚本路径以执行该脚本。```

这个程序文件的主要功能是使用当前的 Python 环境来运行一个指定的脚本，具体是一个名为 `web.py` 的文件。程序首先导入了必要的模块，包括 `sys`、`os` 和 `subprocess`，这些模块提供了与系统交互的功能。  
  
在 `run\_script` 函数中，首先获取当前 Python 解释器的路径，这通过 `sys.executable` 实现。接着，构建一个命令字符串，该命令使用 `streamlit` 模块来运行指定的脚本。`streamlit` 是一个用于构建数据应用的库，这里通过 `-m` 参数来指定模块运行。  
  
随后，使用 `subprocess.run` 方法执行构建好的命令。这个方法会在新的 shell 中运行命令，并等待其完成。如果命令执行后返回的状态码不为零，表示脚本运行出错，程序会输出相应的错误信息。  
  
在文件的最后部分，使用 `if \_\_name\_\_ == "\_\_main\_\_":` 语句来确保只有在直接运行该文件时才会执行下面的代码。这里指定了要运行的脚本路径，即 `web.py`，并调用 `run\_script` 函数来执行该脚本。  
  
总的来说，这个程序提供了一种简便的方式来启动一个基于 Streamlit 的 web 应用，通过指定脚本路径并在当前 Python 环境中运行它。

### 程序整体功能和构架概括  
  
该程序由多个模块组成，主要用于构建和运行深度学习模型，特别是在计算机视觉和图像处理任务中。整体架构包括模型定义、卷积操作的自定义实现以及一个用户界面（UI）模块来启动和运行相关的应用程序。具体来说：  
  
1. \*\*模型定义\*\*：通过 `UniRepLKNet.py` 和 `lsknet.py` 文件定义了不同的深度学习模型架构，包含了多种卷积层、注意力机制和其他特征提取模块，以提高模型的性能和灵活性。  
   
2. \*\*自定义卷积操作\*\*：`shiftwise\_conv.py` 文件实现了自定义的卷积层，特别是针对大卷积核和小卷积核的组合，利用重参数化技术来优化模型的推理性能。  
  
3. \*\*用户界面\*\*：`ui.py` 文件提供了一个简单的用户界面，用于运行基于 Streamlit 的 web 应用，允许用户与模型进行交互。  
  
### 文件功能整理表  
  
| 文件名 | 功能描述 |  
|---------------------|-----------------------------------------------------------------------------------------------|  
| `UniRepLKNet.py` | 实现了 UniRepLKNet 模型，包含多个卷积模块、注意力机制和特征提取层，适用于多种视觉任务。 |  
| `shiftwise\_conv.py` | 定义了自定义卷积层，支持大卷积核和小卷积核的组合，使用重参数化技术以提高推理性能。 |  
| `lsknet.py` | 实现了 LSKNet 模型，包含多个 Block 和注意力机制，适用于图像处理任务，提供了多种配置选项。 |  
| `ui.py` | 提供了一个用户界面，使用 Streamlit 启动并运行指定的 web 应用，便于用户与模型进行交互。 |  
  
这个程序的整体设计使得模型的构建、训练和部署变得更加高效和灵活，适合于各种计算机视觉和深度学习应用。

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。