# 改进yolo11-SDI等200+全套创新点大全：眼科疾病检测系统源码＆数据集全套

## 3.背景

研究背景与意义  
  
随着全球人口老龄化的加剧，眼科疾病的发病率逐年上升，成为影响人们生活质量的重要健康问题。根据世界卫生组织的统计，眼科疾病不仅导致了大量的视觉障碍，还给患者及其家庭带来了沉重的经济负担。因此，早期检测和及时治疗眼科疾病显得尤为重要。传统的眼科疾病诊断主要依赖于医生的经验和专业知识，然而这种方法往往受到主观因素的影响，导致诊断结果的准确性和一致性存在差异。为了解决这一问题，基于计算机视觉的自动化检测系统应运而生，成为提高眼科疾病诊断效率和准确性的重要工具。  
  
本研究旨在基于改进的YOLOv11模型，开发一套高效的眼科疾病检测系统。该系统将专注于四种主要眼科疾病的检测，包括白内障、正常眼、翼状胬肉和麦粒肿。通过利用包含792幅图像的数据集，系统将能够实现对这些疾病的快速识别和分类。YOLOv11作为一种先进的目标检测算法，具有实时性和高准确率的优势，能够有效地处理复杂的视觉信息，并在各种环境下保持稳定的性能。  
  
此外，本研究的意义还在于推动计算机视觉技术在医疗领域的应用，尤其是在眼科疾病的早期筛查和诊断方面。通过自动化检测系统的开发，不仅可以减轻医生的工作负担，还能提高患者的就医体验。随着数据集的不断扩展和模型的持续优化，未来该系统有望实现更广泛的眼科疾病检测，助力于公共卫生的改善和眼科医疗资源的合理配置。因此，本研究不仅具有重要的学术价值，也将为眼科临床实践提供有力的技术支持。

## 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  
  
本项目旨在开发一个改进版的YOLOv11眼科疾病检测系统，为眼科疾病的早期诊断和治疗提供支持。为实现这一目标，我们构建了一个专门的数据集，聚焦于四种主要的眼科疾病。这些疾病包括白内障（Cataract）、正常眼睛（Normal）、翼状胬肉（Pterygium）和麦粒肿（Stye）。数据集的设计旨在涵盖不同年龄段、性别和种族的患者，以确保模型在多样化的临床场景中具有良好的泛化能力。  
  
在数据集的构建过程中，我们严格筛选了高质量的眼部图像，确保每一类疾病的样本数量均衡且具有代表性。每一张图像都经过专业眼科医生的标注，确保数据的准确性和可靠性。此外，为了增强模型的鲁棒性，我们还对图像进行了多种数据增强处理，包括旋转、缩放、亮度调整等，以模拟不同的拍摄条件和患者状态。  
  
数据集的类别数量为四个，分别代表了常见的眼科疾病和正常眼睛的状态。这种分类不仅有助于模型在检测时进行准确的判断，也为后续的临床应用提供了明确的参考依据。通过使用这一数据集，我们期望能够显著提高YOLOv11在眼科疾病检测中的性能，帮助医生更快速、准确地识别眼部疾病，从而提高患者的治疗效果和生活质量。整体而言，本项目的数据集不仅是技术开发的基础，也是推动眼科医疗智能化的重要一步。

## 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

```以下是经过精简和注释的核心代码部分：  
  
```python  
import torch  
import torch.nn as nn  
from timm.models.layers import SqueezeExcite  
  
def \_make\_divisible(v, divisor, min\_value=None):  
 """  
 确保所有层的通道数是可被8整除的  
 :param v: 输入的通道数  
 :param divisor: 需要整除的数  
 :param min\_value: 最小值  
 :return: 调整后的通道数  
 """  
 if min\_value is None:  
 min\_value = divisor  
 new\_v = max(min\_value, int(v + divisor / 2) // divisor \* divisor)  
 if new\_v < 0.9 \* v: # 确保不低于原值的90%  
 new\_v += divisor  
 return new\_v  
  
class Conv2d\_BN(nn.Sequential):  
 """  
 包含卷积层和批归一化层的组合  
 """  
 def \_\_init\_\_(self, in\_channels, out\_channels, kernel\_size=1, stride=1, padding=0, dilation=1, groups=1):  
 super().\_\_init\_\_()  
 # 添加卷积层  
 self.add\_module('conv', nn.Conv2d(in\_channels, out\_channels, kernel\_size, stride, padding, dilation, groups, bias=False))  
 # 添加批归一化层  
 self.add\_module('bn', nn.BatchNorm2d(out\_channels))  
  
 @torch.no\_grad()  
 def fuse\_self(self):  
 """  
 融合卷积层和批归一化层为一个卷积层  
 """  
 conv, bn = self.\_modules.values()  
 # 计算融合后的权重和偏置  
 w = bn.weight / (bn.running\_var + bn.eps)\*\*0.5  
 w = conv.weight \* w[:, None, None, None]  
 b = bn.bias - bn.running\_mean \* bn.weight / (bn.running\_var + bn.eps)\*\*0.5  
 # 创建新的卷积层  
 fused\_conv = nn.Conv2d(w.size(1) \* conv.groups, w.size(0), w.shape[2:], stride=conv.stride, padding=conv.padding, dilation=conv.dilation, groups=conv.groups)  
 fused\_conv.weight.data.copy\_(w)  
 fused\_conv.bias.data.copy\_(b)  
 return fused\_conv  
  
class RepViTBlock(nn.Module):  
 """  
 RepViT模块，包含token混合和channel混合  
 """  
 def \_\_init\_\_(self, inp, hidden\_dim, oup, kernel\_size, stride, use\_se, use\_hs):  
 super(RepViTBlock, self).\_\_init\_\_()  
 self.identity = stride == 1 and inp == oup # 判断是否为恒等映射  
 if stride == 2:  
 # 使用下采样  
 self.token\_mixer = nn.Sequential(  
 Conv2d\_BN(inp, inp, kernel\_size, stride, (kernel\_size - 1) // 2, groups=inp),  
 SqueezeExcite(inp, 0.25) if use\_se else nn.Identity(),  
 Conv2d\_BN(inp, oup, ks=1, stride=1, pad=0)  
 )  
 self.channel\_mixer = nn.Sequential(  
 Conv2d\_BN(oup, 2 \* oup, 1, 1, 0),  
 nn.GELU() if use\_hs else nn.Identity(),  
 Conv2d\_BN(2 \* oup, oup, 1, 1, 0)  
 )  
 else:  
 assert self.identity  
 self.token\_mixer = nn.Sequential(  
 Conv2d\_BN(inp, inp, 3, 1, 1, groups=inp),  
 SqueezeExcite(inp, 0.25) if use\_se else nn.Identity(),  
 )  
 self.channel\_mixer = nn.Sequential(  
 Conv2d\_BN(inp, hidden\_dim, 1, 1, 0),  
 nn.GELU() if use\_hs else nn.Identity(),  
 Conv2d\_BN(hidden\_dim, oup, 1, 1, 0)  
 )  
  
 def forward(self, x):  
 return self.channel\_mixer(self.token\_mixer(x))  
  
class RepViT(nn.Module):  
 """  
 RepViT模型  
 """  
 def \_\_init\_\_(self, cfgs):  
 super(RepViT, self).\_\_init\_\_()  
 self.cfgs = cfgs # 配置反向残差块  
 layers = []  
 input\_channel = cfgs[0][2] # 获取输入通道数  
 # 构建第一个层  
 patch\_embed = nn.Sequential(Conv2d\_BN(3, input\_channel // 2, 3, 2, 1), nn.GELU(),  
 Conv2d\_BN(input\_channel // 2, input\_channel, 3, 2, 1))  
 layers.append(patch\_embed)  
 # 构建反向残差块  
 for k, t, c, use\_se, use\_hs, s in self.cfgs:  
 output\_channel = \_make\_divisible(c, 8)  
 exp\_size = \_make\_divisible(input\_channel \* t, 8)  
 layers.append(RepViTBlock(input\_channel, exp\_size, output\_channel, k, s, use\_se, use\_hs))  
 input\_channel = output\_channel  
 self.features = nn.ModuleList(layers)  
  
 def forward(self, x):  
 for f in self.features:  
 x = f(x)  
 return x  
  
def repvit\_m2\_3(weights=''):  
 """  
 构建RepViT模型  
 """  
 cfgs = [  
 # k, t, c, SE, HS, s   
 [3, 2, 80, 1, 0, 1],  
 [3, 2, 80, 0, 0, 1],  
 # 省略其他配置...  
 [3, 2, 640, 0, 1, 2],  
 [3, 2, 640, 1, 1, 1],  
 ]  
 model = RepViT(cfgs)  
 if weights:  
 model.load\_state\_dict(torch.load(weights)['model'])  
 return model  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 model = repvit\_m2\_3('repvit\_m2\_3\_distill\_450e.pth')  
 inputs = torch.randn((1, 3, 640, 640))  
 res = model(inputs)  
 for i in res:  
 print(i.size())  
```  
  
### 代码说明：  
1. \*\*\_make\_divisible\*\*: 确保通道数是8的倍数，以便在硬件上优化性能。  
2. \*\*Conv2d\_BN\*\*: 组合卷积层和批归一化层，并提供融合方法以提高推理速度。  
3. \*\*RepViTBlock\*\*: 实现RepViT的基本模块，包含token混合和channel混合。  
4. \*\*RepViT\*\*: 构建整个RepViT模型，利用配置参数生成多个模块。  
5. \*\*repvit\_m2\_3\*\*: 根据给定的配置构建RepViT模型，并可加载预训练权重。  
  
通过这些核心部分的实现，RepViT模型能够有效地进行图像处理任务。```

这个文件 `repvit.py` 是一个用于构建和训练 RepViT（Residual Vision Transformer）模型的 PyTorch 实现。该模型结合了卷积神经网络（CNN）和变换器（Transformer）的优点，适用于图像分类等任务。  
  
首先，文件导入了必要的库，包括 PyTorch 的神经网络模块 `torch.nn`、NumPy 以及 `timm` 库中的 SqueezeExcite 层。接着，定义了一个名为 `replace\_batchnorm` 的函数，该函数用于替换网络中的 BatchNorm2d 层为 Identity 层，主要用于模型推理时的优化。  
  
`\_make\_divisible` 函数确保所有层的通道数都是 8 的倍数，以满足某些硬件加速的要求。接下来，定义了 `Conv2d\_BN` 类，这是一个包含卷积层和 BatchNorm 层的组合，具有自我融合的能力，能够在推理时减少计算量。  
  
`Residual` 类实现了残差连接的机制，允许在训练时使用随机丢弃，以增强模型的鲁棒性。它同样具有自我融合的功能，以便在推理时进行优化。  
  
`RepVGGDW` 类实现了一种特殊的卷积结构，结合了深度可分离卷积和残差连接。该类的 `forward` 方法将输入通过卷积和 BatchNorm 层进行处理，并添加输入的残差。  
  
`RepViTBlock` 类是模型的基本构建块，包含了通道混合和令牌混合的机制。它根据输入的步幅选择不同的结构，以适应不同的下采样需求。  
  
`RepViT` 类是整个模型的主体，负责构建网络的各个层次。它根据配置参数 `cfgs` 来创建多个 `RepViTBlock`，并定义了前向传播的逻辑。该类还实现了 `switch\_to\_deploy` 方法，用于在推理时替换 BatchNorm 层。  
  
`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相匹配。  
  
接下来，定义了一系列的模型构造函数，如 `repvit\_m0\_9`、`repvit\_m1\_0` 等，这些函数根据不同的配置参数构建不同版本的 RepViT 模型，并可选择加载预训练权重。  
  
最后，在文件的主程序部分，创建了一个 `repvit\_m2\_3` 模型实例，并生成随机输入进行测试，输出每个特征图的尺寸。  
  
总体而言，这个文件实现了一个灵活且高效的视觉模型，结合了现代深度学习中的多种技术，适用于各种计算机视觉任务。

```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 torch  
import torch.nn as nn  
import torch.nn.functional as F  
import numpy as np  
  
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):  
 """ 前向传播函数。 """  
 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  
  
 # 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() # 重新排列维度  
 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  
 relative\_position\_index = relative\_coords.sum(-1) # 计算相对位置索引  
 self.register\_buffer("relative\_position\_index", relative\_position\_index) # 注册为缓冲区  
  
 self.qkv = nn.Linear(dim, dim \* 3, bias=qkv\_bias) # 线性变换用于生成Q、K、V  
 self.attn\_drop = nn.Dropout(attn\_drop) # 注意力权重的Dropout  
 self.proj = nn.Linear(dim, dim) # 输出线性变换  
 self.proj\_drop = nn.Dropout(proj\_drop) # 输出的Dropout  
  
 def forward(self, x, mask=None):  
 """ 前向传播函数。 """  
 B\_, N, C = x.shape # 获取输入形状  
 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)) # 计算注意力分数  
  
 # 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() # 重新排列维度  
 attn = attn + relative\_position\_bias.unsqueeze(0) # 加入偏置  
  
 attn = F.softmax(attn, dim=-1) # 计算softmax以获得注意力权重  
 attn = self.attn\_drop(attn) # 应用Dropout  
  
 x = (attn @ v).transpose(1, 2).reshape(B\_, N, C) # 计算加权和  
 x = self.proj(x) # 输出线性变换  
 x = self.proj\_drop(x) # 应用Dropout  
 return x  
  
  
class SwinTransformer(nn.Module):  
 """ Swin Transformer主干网络。 """  
  
 def \_\_init\_\_(self, pretrain\_img\_size=224, patch\_size=4, in\_chans=3, embed\_dim=96, depths=[2, 2, 6, 2],  
 num\_heads=[3, 6, 12, 24], window\_size=7, mlp\_ratio=4., drop\_rate=0., attn\_drop\_rate=0.):  
 super().\_\_init\_\_()  
  
 self.patch\_embed = PatchEmbed(patch\_size=patch\_size, in\_chans=in\_chans, embed\_dim=embed\_dim) # 图像分块嵌入  
 self.layers = nn.ModuleList() # 存储每一层  
  
 # 构建每一层  
 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],  
 window\_size=window\_size,  
 mlp\_ratio=mlp\_ratio,  
 drop=drop\_rate,  
 attn\_drop=attn\_drop\_rate)  
 self.layers.append(layer)  
  
 def forward(self, x):  
 """ 前向传播函数。 """  
 x = self.patch\_embed(x) # 图像分块嵌入  
 for layer in self.layers:  
 x = layer(x) # 逐层传递  
 return x # 返回最终输出  
  
  
def SwinTransformer\_Tiny(weights=''):  
 """ 创建一个小型的Swin Transformer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，包含两个线性层和一个激活函数，支持Dropout。  
2. \*\*WindowAttention类\*\*：实现了窗口注意力机制，支持相对位置偏置，计算Q、K、V并进行注意力计算。  
3. \*\*SwinTransformer类\*\*：实现了Swin Transformer的主干结构，包含图像分块嵌入和多个Transformer层。  
4. \*\*SwinTransformer\_Tiny函数\*\*：创建一个小型的Swin Transformer模型，并可选择加载预训练权重。```

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是一种层次化的视觉Transformer，采用了移动窗口机制以提高计算效率。以下是代码的主要组成部分和功能说明。  
  
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，它实现了一个多层感知机（MLP），包含两个线性层和一个激活函数（默认为GELU），以及可选的Dropout层。  
  
接下来，定义了两个辅助函数`window\_partition`和`window\_reverse`，它们用于将输入特征图分割成窗口和将窗口合并回特征图。这是Swin Transformer中窗口注意力机制的基础。  
  
`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。该类中定义了查询、键、值的线性变换，以及相对位置偏置的计算和应用。注意力权重通过Softmax函数进行归一化。  
  
`SwinTransformerBlock`类是Swin Transformer的基本构建块，包含了规范化层、窗口注意力层和MLP层。它还实现了循环移位的功能，以支持SW-MSA（Shifted Window Multi-Head Self-Attention）。  
  
`PatchMerging`类用于将特征图的补丁合并，以减少特征图的分辨率。它通过线性层将四个相邻的补丁合并为一个新的补丁。  
  
`BasicLayer`类表示Swin Transformer中的一个基本层，包含多个Swin Transformer块，并在必要时执行下采样。  
  
`PatchEmbed`类将输入图像划分为不重叠的补丁，并通过卷积层将这些补丁嵌入到高维空间中。  
  
`SwinTransformer`类是整个模型的核心，负责构建模型的各个层次，并实现前向传播。它支持绝对位置嵌入、Dropout等功能，并在每个阶段后应用规范化层。  
  
最后，定义了一个`update\_weight`函数，用于加载预训练模型的权重，并定义了一个`SwinTransformer\_Tiny`函数，用于创建一个小型的Swin Transformer模型实例。  
  
总体来说，这个程序文件实现了Swin Transformer的各个组件，并提供了一个完整的模型结构，适用于图像分类、目标检测等计算机视觉任务。

```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 multi\_scale\_deformable\_attn\_pytorch(  
 value: torch.Tensor,  
 value\_spatial\_shapes: torch.Tensor,  
 sampling\_locations: torch.Tensor,  
 attention\_weights: torch.Tensor,  
) -> torch.Tensor:  
 """  
 多尺度可变形注意力机制。  
  
 参数:  
 value: 输入特征图，形状为 (bs, C, num\_heads, embed\_dims)  
 value\_spatial\_shapes: 特征图的空间形状，形状为 (num\_levels, 2)  
 sampling\_locations: 采样位置，形状为 (bs, num\_queries, num\_heads, num\_levels, num\_points, 2)  
 attention\_weights: 注意力权重，形状为 (bs, num\_heads, num\_queries, num\_levels, num\_points)  
  
 返回:  
 output: 经过多尺度可变形注意力机制处理后的输出，形状为 (bs, num\_queries, num\_heads \* embed\_dims)  
 """  
 bs, \_, num\_heads, embed\_dims = value.shape # 获取输入特征图的批次大小、通道数、头数和嵌入维度  
 \_, num\_queries, \_, num\_levels, num\_points, \_ = sampling\_locations.shape # 获取采样位置的相关维度  
  
 # 将输入特征图按照空间形状分割成多个特征图  
 value\_list = value.split([H\_ \* W\_ for H\_, W\_ in value\_spatial\_shapes], dim=1)  
   
 # 将采样位置转换到[-1, 1]范围  
 sampling\_grids = 2 \* sampling\_locations - 1  
 sampling\_value\_list = [] # 用于存储每个层级的采样值  
  
 for level, (H\_, W\_) in enumerate(value\_spatial\_shapes):  
 # 对于每个层级，进行特征图的处理  
 value\_l\_ = value\_list[level].flatten(2).transpose(1, 2).reshape(bs \* num\_heads, embed\_dims, H\_, W\_)  
   
 # 处理采样网格  
 sampling\_grid\_l\_ = sampling\_grids[:, :, :, level].transpose(1, 2).flatten(0, 1)  
   
 # 使用grid\_sample进行双线性插值采样  
 sampling\_value\_l\_ = F.grid\_sample(  
 value\_l\_, sampling\_grid\_l\_, mode="bilinear", padding\_mode="zeros", align\_corners=False  
 )  
 sampling\_value\_list.append(sampling\_value\_l\_) # 将采样值添加到列表中  
  
 # 将注意力权重进行维度变换  
 attention\_weights = attention\_weights.transpose(1, 2).reshape(  
 bs \* num\_heads, 1, num\_queries, num\_levels \* num\_points  
 )  
   
 # 计算最终输出  
 output = (  
 (torch.stack(sampling\_value\_list, dim=-2).flatten(-2) \* attention\_weights)  
 .sum(-1)  
 .view(bs, num\_heads \* embed\_dims, num\_queries)  
 )  
   
 return output.transpose(1, 2).contiguous() # 返回输出，调整维度顺序并确保内存连续  
```  
  
### 代码说明：  
1. \*\*输入参数\*\*：  
 - `value`：输入特征图，包含多个头的特征。  
 - `value\_spatial\_shapes`：特征图的空间形状，用于分割特征图。  
 - `sampling\_locations`：指定的采样位置，包含多个查询和层级。  
 - `attention\_weights`：每个查询的注意力权重。  
  
2. \*\*处理流程\*\*：  
 - 将输入特征图根据空间形状分割成多个特征图。  
 - 将采样位置转换到[-1, 1]范围，以适应`grid\_sample`函数。  
 - 对每个层级的特征图进行双线性插值采样，得到对应的采样值。  
 - 通过注意力权重对采样值进行加权求和，最终得到输出。  
  
3. \*\*输出\*\*：  
 - 返回经过多尺度可变形注意力机制处理后的输出，形状为 `(bs, num\_queries, num\_heads \* embed\_dims)`。```

这个程序文件 `utils.py` 是一个用于实现一些常用功能的模块，主要与深度学习和神经网络相关，特别是在使用 PyTorch 框架时。文件中包含了一些工具函数和多尺度可变形注意力机制的实现。  
  
首先，文件导入了一些必要的库，包括 `copy`、`math`、`numpy` 和 `torch` 及其子模块。这些库提供了深度学习所需的基本功能和数学运算支持。  
  
文件中定义了几个函数。`\_get\_clones(module, n)` 函数用于创建一个给定模块的克隆列表，返回一个 `nn.ModuleList`，其中包含 `n` 个深拷贝的模块。这在构建具有多个相同层的网络时非常有用。  
  
`bias\_init\_with\_prob(prior\_prob=0.01)` 函数用于根据给定的概率值初始化卷积或全连接层的偏置值。它通过计算负对数几率来实现，返回一个浮点数作为偏置的初始化值。  
  
`linear\_init(module)` 函数用于初始化线性模块的权重和偏置。它使用均匀分布在一个特定的范围内初始化权重和偏置，范围是基于模块权重的形状计算得出的。  
  
`inverse\_sigmoid(x, eps=1e-5)` 函数计算张量的反 sigmoid 函数。它首先将输入限制在 [0, 1] 的范围内，然后通过对数运算计算反 sigmoid 值，避免了数值不稳定性。  
  
最后，`multi\_scale\_deformable\_attn\_pytorch` 函数实现了多尺度可变形注意力机制。该函数接受四个参数：`value`、`value\_spatial\_shapes`、`sampling\_locations` 和 `attention\_weights`。它首先提取输入张量的形状信息，然后对每个尺度的特征图进行处理。通过对采样位置进行变换，使用 `F.grid\_sample` 函数从特征图中采样，得到不同尺度的特征值。接着，函数将这些采样值与注意力权重结合，最终输出一个经过变形注意力机制处理的结果张量。  
  
总体而言，这个文件提供了一些基础的工具函数和一个复杂的注意力机制实现，适用于深度学习模型的构建和训练，尤其是在目标检测和图像处理等任务中。

```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 torch  
import torch.nn as nn  
import torch.nn.functional as F  
  
class GRNwithNHWC(nn.Module):  
 """   
 GRN (全局响应归一化) 层  
 该层的输入假设为 (N, H, W, C) 格式  
 """  
 def \_\_init\_\_(self, dim, use\_bias=True):  
 super().\_\_init\_\_()  
 self.use\_bias = use\_bias  
 # 初始化可学习参数 gamma 和 beta  
 self.gamma = nn.Parameter(torch.zeros(1, 1, 1, dim))  
 if self.use\_bias:  
 self.beta = nn.Parameter(torch.zeros(1, 1, 1, dim))  
  
 def forward(self, x):  
 # 计算输入的 L2 范数  
 Gx = torch.norm(x, p=2, dim=(1, 2), keepdim=True)  
 # 计算归一化因子  
 Nx = Gx / (Gx.mean(dim=-1, keepdim=True) + 1e-6)  
 # 返回归一化后的输出  
 if self.use\_bias:  
 return (self.gamma \* Nx + 1) \* x + self.beta  
 else:  
 return (self.gamma \* Nx + 1) \* x  
  
class UniRepLKNetBlock(nn.Module):  
 """  
 UniRepLKNet 的基本模块  
 """  
 def \_\_init\_\_(self, dim, kernel\_size, drop\_path=0., deploy=False):  
 super().\_\_init\_\_()  
 # 卷积层  
 self.dwconv = nn.Conv2d(dim, dim, kernel\_size=kernel\_size, stride=1, padding=kernel\_size // 2, groups=dim)  
 # 归一化层  
 self.norm = nn.BatchNorm2d(dim)  
 # Squeeze-and-Excitation (SE) 块  
 self.se = SEBlock(dim, dim // 4)  
 # 前馈网络  
 self.pwconv1 = nn.Linear(dim, dim \* 4)  
 self.pwconv2 = nn.Linear(dim \* 4, dim)  
  
 def forward(self, inputs):  
 # 前向传播  
 x = self.dwconv(inputs) # 卷积操作  
 x = self.norm(x) # 归一化  
 x = self.se(x) # SE 块  
 x = F.gelu(x) # 激活函数  
 x = self.pwconv1(x) # 前馈网络第一层  
 x = self.pwconv2(x) # 前馈网络第二层  
 return x  
  
class UniRepLKNet(nn.Module):  
 """  
 UniRepLKNet 模型  
 """  
 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()  
 # 构建各个阶段的模块  
 for i in range(len(depths)):  
 stage = nn.Sequential(  
 \*[UniRepLKNetBlock(dim=dims[i], kernel\_size=3) for \_ in range(depths[i])]  
 )  
 self.stages.append(stage)  
  
 def forward(self, x):  
 # 前向传播  
 for stage in self.stages:  
 x = stage(x)  
 return x  
  
# 示例用法  
if \_\_name\_\_ == '\_\_main\_\_':  
 inputs = torch.randn((1, 3, 640, 640)) # 随机输入  
 model = UniRepLKNet() # 实例化模型  
 res = model(inputs) # 前向传播  
 print(res.shape) # 输出结果的形状  
```  
  
### 代码注释说明：  
1. \*\*GRNwithNHWC 类\*\*：实现了全局响应归一化层，主要用于对输入特征进行归一化处理，提升模型的稳定性和性能。  
2. \*\*UniRepLKNetBlock 类\*\*：构建了 UniRepLKNet 的基本模块，包含卷积层、归一化层、Squeeze-and-Excitation 块和前馈网络。  
3. \*\*UniRepLKNet 类\*\*：构建了整个模型，包含多个 UniRepLKNetBlock 组成的阶段。  
4. \*\*示例用法\*\*：展示了如何实例化模型并进行前向传播，输出结果的形状。```

这个程序文件 `UniRepLKNet.py` 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。该模型基于多个已有的模型架构，包括 RepLKNet、ConvNeXt、DINO 和 DeiT，旨在提供一种通用的感知能力。  
  
程序开始部分引入了必要的库和模块，包括 PyTorch 和一些用于构建神经网络的功能模块。接下来，定义了一些基本的层和功能，包括 GRN（全局响应归一化）层、NCHW 和 NHWC 格式的转换层、以及用于卷积操作的函数 `get\_conv2d`。该函数可以根据输入参数决定使用原生卷积还是高效的 iGEMM 大核卷积实现。  
  
接下来，定义了 Squeeze-and-Excitation（SE）块，这是一个用于增强特征表示的模块。它通过自适应平均池化和两个卷积层来实现特征的压缩和激励。然后，程序中还实现了融合批归一化（Batch Normalization）和卷积层的函数，以优化模型的性能。  
  
在模型的核心部分，定义了 `DilatedReparamBlock` 和 `UniRepLKNetBlock` 类，这些类构成了 UniRepLKNet 的基本构建块。`DilatedReparamBlock` 采用膨胀卷积来捕捉更大范围的上下文信息，而 `UniRepLKNetBlock` 则将多个模块组合在一起，包括卷积、归一化、激活和前馈网络。  
  
接下来，`UniRepLKNet` 类是整个模型的主体，接受多个参数来配置模型的输入通道数、类别数、深度、特征维度等。它通过多层下采样和多个阶段的块构建整个网络，并在前向传播中根据设定的输出模式返回特征或分类结果。  
  
最后，程序提供了一些函数，用于创建不同版本的 UniRepLKNet 模型，并加载预训练权重。程序的入口部分则演示了如何使用该模型进行推理，包括模型的部署模式切换。  
  
总体而言，这个程序文件实现了一个灵活且高效的深度学习模型，适用于多种视觉和感知任务，并提供了多种配置选项以满足不同需求。

### 程序整体功能和构架概括  
  
该程序整体实现了多个深度学习模型，主要用于计算机视觉任务，包括图像分类、目标检测等。每个模型都有其独特的架构和功能，结合了现代深度学习中的多种技术，如卷积神经网络（CNN）、变换器（Transformer）和注意力机制。程序结构清晰，模块化设计使得各个模型的实现和使用变得灵活，便于扩展和维护。  
  
以下是各个文件的功能概述：  
  
| 文件名 | 功能描述 |  
|----------------------|--------------------------------------------------------------------------------------------|  
| `repvit.py` | 实现了 RepViT（Residual Vision Transformer）模型，结合了卷积和变换器的优点，适用于图像分类等任务。 |  
| `SwinTransformer.py` | 实现了 Swin Transformer 模型，采用移动窗口机制，适用于各种计算机视觉任务。 |  
| `utils.py` | 提供了一些工具函数和多尺度可变形注意力机制的实现，支持模型的构建和训练。 |  
| `UniRepLKNet.py` | 实现了 UniRepLKNet 模型，适用于多种感知任务，结合了多个已有模型架构，提供通用的特征提取能力。 |  
  
这些文件共同构成了一个强大的深度学习框架，能够处理多种视觉和感知任务，具备良好的扩展性和灵活性。