# 改进yolo11-slimneck等200+全套创新点大全：装卸码头场景检测系统源码＆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  
  
随着全球贸易的不断增长，港口和装卸码头的作业效率显得尤为重要。装卸码头作为货物运输的重要节点，其作业过程涉及多种设备和人员的协同工作。为了提高作业效率和安全性，实时监测和智能识别装卸码头场景中的各种目标显得尤为必要。近年来，深度学习技术在计算机视觉领域取得了显著进展，尤其是目标检测算法的快速发展，为实现这一目标提供了强有力的技术支持。YOLO（You Only Look Once）系列算法因其高效的实时检测能力，已成为目标检测领域的热门选择。  
  
本研究旨在基于改进的YOLOv11算法，构建一个高效的装卸码头场景检测系统。该系统将针对装卸码头特有的目标进行识别，包括叉车、装卸码头、半挂车、终端拖车、拖拉机单元以及工作人员等。通过对247幅图像的分析和训练，我们希望系统能够在复杂的工作环境中，准确识别和定位各类目标，从而为装卸作业提供实时数据支持，降低人为错误和事故发生的风险。  
  
此外，随着自动化和智能化技术的不断发展，装卸码头的作业模式也在逐步转变。基于改进YOLOv11的检测系统，不仅能够提升作业效率，还能为未来的智能港口建设提供数据基础和技术保障。通过本研究，我们期望为装卸码头的智能化管理提供一种新的解决方案，推动港口行业的数字化转型与升级，最终实现更高效、更安全的货物运输与管理。

## 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“dock”，旨在为改进YOLOv11的装卸码头场景检测系统提供丰富的训练素材。该数据集包含七个类别，具体包括“dock number”、“forklift”、“loading dock”、“person”、“semi-trailer”、“terminal tractor”和“tractor unit”。这些类别涵盖了装卸码头场景中常见的各种元素，能够有效支持模型在复杂环境中的目标检测任务。  
  
在数据集的构建过程中，特别注重了数据的多样性和真实性，以确保模型能够在实际应用中表现出色。数据集中的图像来源于不同的装卸码头场景，包含了多种天气条件、光照变化以及不同的视角，旨在模拟真实世界中可能遇到的各种情况。这种多样性不仅增强了模型的鲁棒性，还提高了其在不同环境下的适应能力。  
  
此外，数据集中的每个类别都经过精确标注，确保了训练过程中数据的准确性。比如，“forklift”类别的图像标注涵盖了不同类型和颜色的叉车，以帮助模型学习识别各种叉车的特征；而“loading dock”类别则包含了不同设计和布局的装卸码头，以提高模型对不同环境的理解能力。通过这种细致的标注和多样化的样本选择，数据集为YOLOv11的训练提供了坚实的基础。  
  
总之，“dock”数据集不仅为改进YOLOv11的装卸码头场景检测系统提供了丰富的训练数据，还通过多样性和准确性确保了模型在实际应用中的有效性和可靠性。这将有助于提升装卸码头的作业效率和安全性，为相关行业的发展贡献力量。

## 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

```以下是对给定代码的核心部分进行分析和详细注释的结果。代码主要实现了跨层通道注意力和空间注意力机制，以下是简化和注释后的核心代码：  
  
```python  
import torch  
import torch.nn as nn  
import torch.nn.functional as F  
from timm.layers import to\_2tuple, trunc\_normal\_  
  
class CrossLayerPosEmbedding3D(nn.Module):  
 def \_\_init\_\_(self, num\_heads=4, window\_size=(5, 3, 1), spatial=True):  
 super(CrossLayerPosEmbedding3D, self).\_\_init\_\_()  
 self.spatial = spatial # 是否使用空间位置嵌入  
 self.num\_heads = num\_heads # 注意力头的数量  
 self.layer\_num = len(window\_size) # 层数  
 # 初始化相对位置偏置表  
 self.relative\_position\_bias\_table = nn.Parameter(  
 torch.zeros((2 \* window\_size[0] - 1) \* (2 \* window\_size[0] - 1), num\_heads)  
 )  
 # 计算相对位置索引  
 self.relative\_position\_index = self.\_compute\_relative\_position\_index(window\_size)  
 trunc\_normal\_(self.relative\_position\_bias\_table, std=.02) # 初始化偏置表  
  
 def \_compute\_relative\_position\_index(self, window\_size):  
 # 计算相对位置索引  
 coords\_h = [torch.arange(ws) - ws // 2 for ws in window\_size]  
 coords\_w = [torch.arange(ws) - ws // 2 for ws in window\_size]  
 coords = [torch.stack(torch.meshgrid([coord\_h, coord\_w])) for coord\_h, coord\_w in zip(coords\_h, coords\_w)]  
 coords\_flatten = torch.cat([torch.flatten(coord, 1) for coord in coords], dim=-1)  
 relative\_coords = coords\_flatten[:, :, None] - coords\_flatten[:, None, :]  
 relative\_coords = relative\_coords.permute(1, 2, 0).contiguous()  
 relative\_coords[:, :, 0] += window\_size[0] - 1  
 relative\_coords[:, :, 1] += window\_size[0] - 1  
 relative\_coords[:, :, 0] \*= 2 \* window\_size[0] - 1  
 return relative\_coords.sum(-1)  
  
 def forward(self):  
 # 计算位置嵌入  
 pos\_indicies = self.relative\_position\_index.view(-1)  
 pos\_indicies\_floor = torch.floor(pos\_indicies).long()  
 pos\_indicies\_ceil = torch.ceil(pos\_indicies).long()  
 value\_floor = self.relative\_position\_bias\_table[pos\_indicies\_floor]  
 value\_ceil = self.relative\_position\_bias\_table[pos\_indicies\_ceil]  
 weights\_ceil = pos\_indicies - pos\_indicies\_floor.float()  
 weights\_floor = 1.0 - weights\_ceil  
  
 pos\_embed = weights\_floor.unsqueeze(-1) \* value\_floor + weights\_ceil.unsqueeze(-1) \* value\_ceil  
 return pos\_embed.reshape(1, 1, -1, -1, self.num\_heads).permute(0, 4, 1, 2, 3)  
  
class CrossLayerSpatialAttention(nn.Module):  
 def \_\_init\_\_(self, in\_dim, layer\_num=3, num\_heads=4):  
 super(CrossLayerSpatialAttention, self).\_\_init\_\_()  
 self.num\_heads = num\_heads  
 self.window\_sizes = [2 \*\* i for i in range(layer\_num)][::-1] # 窗口大小  
 self.token\_num\_per\_layer = [ws \*\* 2 for ws in self.window\_sizes] # 每层的token数量  
 self.token\_num = sum(self.token\_num\_per\_layer) # 总token数量  
  
 # 定义各层的卷积、归一化和注意力机制  
 self.cpe = nn.ModuleList([ConvPosEnc(in\_dim) for \_ in range(layer\_num)]) # 卷积位置编码  
 self.qkv = nn.ModuleList(nn.Conv2d(in\_dim, in\_dim \* 3, kernel\_size=1) for \_ in range(layer\_num)) # QKV变换  
 self.softmax = nn.Softmax(dim=-1) # Softmax层  
 self.pos\_embed = CrossLayerPosEmbedding3D(num\_heads=num\_heads, window\_size=self.window\_sizes, spatial=True) # 位置嵌入  
  
 def forward(self, x\_list):  
 q\_list, k\_list, v\_list = [], [], []  
 for i, x in enumerate(x\_list):  
 qkv = self.qkv[i](x) # 计算QKV  
 q, k, v = qkv.chunk(3, dim=1) # 分割QKV  
 q\_list.append(q)  
 k\_list.append(k)  
 v\_list.append(v)  
  
 # 计算注意力  
 q\_stack = torch.cat(q\_list, dim=1)  
 k\_stack = torch.cat(k\_list, dim=1)  
 v\_stack = torch.cat(v\_list, dim=1)  
 attn = F.normalize(q\_stack, dim=-1) @ F.normalize(k\_stack, dim=-1).transpose(-1, -2) # 计算注意力矩阵  
 attn = attn + self.pos\_embed() # 加入位置嵌入  
 attn = self.softmax(attn) # 归一化  
  
 out = attn @ v\_stack # 计算输出  
 return out  
  
class ConvPosEnc(nn.Module):  
 def \_\_init\_\_(self, dim, k=3):  
 super(ConvPosEnc, self).\_\_init\_\_()  
 self.proj = nn.Conv2d(dim, dim, kernel\_size=k, padding=k // 2, groups=dim) # 深度可分离卷积  
  
 def forward(self, x):  
 return x + self.proj(x) # 添加卷积特征  
  
# 代码的核心部分是 CrossLayerSpatialAttention 和 CrossLayerPosEmbedding3D 类  
# 这些类实现了跨层的空间注意力机制和位置嵌入功能，适用于处理图像数据的深度学习模型。  
```  
  
### 代码分析  
1. \*\*CrossLayerPosEmbedding3D\*\*: 该类负责计算相对位置嵌入，使用了相对位置偏置表来增强模型对空间关系的理解。  
2. \*\*CrossLayerSpatialAttention\*\*: 该类实现了跨层空间注意力机制，能够在不同层之间传递信息，并通过注意力机制聚合特征。  
3. \*\*ConvPosEnc\*\*: 该类实现了卷积位置编码，利用深度可分离卷积对输入特征进行增强。  
  
### 主要功能  
- 通过注意力机制和位置嵌入，增强模型对图像特征的学习能力。  
- 支持多层次特征融合，提高模型的表达能力。  
  
### 注意事项  
- 代码中涉及到的参数如`num\_heads`、`window\_size`等，可以根据具体任务进行调整，以达到最佳效果。```

该程序文件 `cfpt.py` 实现了一个基于深度学习的跨层注意力机制，主要用于图像处理任务。文件中定义了多个类和函数，涉及到位置编码、卷积操作、注意力机制等内容。以下是对文件中主要部分的逐步分析。  
  
首先，文件导入了一些必要的库，包括 `torch`、`math`、`einops`、`torch.nn` 等，这些库提供了构建和训练神经网络所需的基本功能。`timm.layers` 中的 `DropPath`、`to\_2tuple` 和 `trunc\_normal\_` 也被引入，用于特定的层和初始化。  
  
接下来，定义了 `LayerNormProxy` 类，这是一个简单的层归一化模块。它重写了 `forward` 方法，使用 `einops` 库对输入张量的维度进行重排，以适应层归一化的要求。  
  
`CrossLayerPosEmbedding3D` 类实现了三维位置嵌入。该类的构造函数根据给定的窗口大小和头数初始化了一些参数，包括相对位置偏置表和绝对位置偏置。`forward` 方法计算并返回位置嵌入。  
  
`ConvPosEnc` 类实现了卷积位置编码。它使用卷积层对输入进行处理，并可选择性地应用激活函数。  
  
`DWConv` 类实现了深度卷积操作，主要用于对输入进行空间上的特征提取。  
  
`Mlp` 类实现了多层感知机结构，包括两个线性层和一个激活函数。它用于对输入特征进行非线性变换。  
  
接下来，定义了一些辅助函数，如 `overlaped\_window\_partition` 和 `overlaped\_window\_reverse`，这些函数用于在窗口内进行分区和重构，主要用于处理图像的重叠窗口。  
  
`CrossLayerSpatialAttention` 类实现了跨层空间注意力机制。它包含多个层，每层都有卷积位置编码、归一化、查询-键-值（QKV）计算、注意力计算等。`forward` 方法处理输入的多个层，并计算注意力输出。  
  
`CrossLayerChannelAttention` 类实现了跨层通道注意力机制，结构与 `CrossLayerSpatialAttention` 类似，但处理方式有所不同。它同样包含多个层，并在 `forward` 方法中实现了通道注意力的计算。  
  
总体而言，该文件实现了一个复杂的注意力机制，结合了空间和通道的信息，通过多层结构来增强特征提取能力。这种机制在图像处理和计算机视觉任务中具有重要的应用价值。

```以下是保留的核心代码部分，并附上详细的中文注释：  
  
```python  
import torch  
import torch.nn as nn  
from functools import lru\_cache  
  
class KAGNConvNDLayer(nn.Module):  
 def \_\_init\_\_(self, conv\_class, norm\_class, conv\_w\_fun, input\_dim, output\_dim, degree, kernel\_size,  
 groups=1, padding=0, stride=1, dilation=1, dropout: float = 0.0, ndim: int = 2):  
 super(KAGNConvNDLayer, self).\_\_init\_\_()  
 # 初始化输入和输出维度、卷积参数等  
 self.inputdim = input\_dim  
 self.outdim = output\_dim  
 self.degree = degree  
 self.kernel\_size = kernel\_size  
 self.padding = padding  
 self.stride = stride  
 self.dilation = dilation  
 self.groups = groups  
 self.base\_activation = nn.SiLU() # 使用SiLU激活函数  
 self.conv\_w\_fun = conv\_w\_fun # 卷积权重函数  
 self.ndim = ndim # 维度  
 self.dropout = None # 初始化dropout  
 if dropout > 0:  
 # 根据维度选择相应的dropout层  
 if ndim == 1:  
 self.dropout = nn.Dropout1d(p=dropout)  
 if ndim == 2:  
 self.dropout = nn.Dropout2d(p=dropout)  
 if ndim == 3:  
 self.dropout = nn.Dropout3d(p=dropout)  
  
 # 检查groups参数的有效性  
 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')  
  
 # 创建卷积层和归一化层  
 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)])  
  
 self.layer\_norm = nn.ModuleList([norm\_class(output\_dim // groups) for \_ in range(groups)])  
  
 # 初始化多项式权重  
 poly\_shape = (groups, output\_dim // groups, (input\_dim // groups) \* (degree + 1)) + tuple(  
 kernel\_size for \_ in range(ndim))  
  
 self.poly\_weights = nn.Parameter(torch.randn(\*poly\_shape)) # 多项式权重  
 self.beta\_weights = nn.Parameter(torch.zeros(degree + 1, dtype=torch.float32)) # beta权重  
  
 # 使用Kaiming均匀分布初始化权重  
 for conv\_layer in self.base\_conv:  
 nn.init.kaiming\_uniform\_(conv\_layer.weight, nonlinearity='linear')  
  
 nn.init.kaiming\_uniform\_(self.poly\_weights, nonlinearity='linear')  
 nn.init.normal\_(  
 self.beta\_weights,  
 mean=0.0,  
 std=1.0 / ((kernel\_size \*\* ndim) \* self.inputdim \* (self.degree + 1.0)),  
 )  
  
 def beta(self, n, m):  
 # 计算beta值  
 return (  
 ((m + n) \* (m - n) \* n \*\* 2) / (m \*\* 2 / (4.0 \* n \*\* 2 - 1.0))  
 ) \* self.beta\_weights[n]  
  
 @lru\_cache(maxsize=128) # 使用缓存避免重复计算Legendre多项式  
 def gram\_poly(self, x, degree):  
 # 计算Gram多项式  
 p0 = x.new\_ones(x.size()) # p0初始化为1  
  
 if degree == 0:  
 return p0.unsqueeze(-1)  
  
 p1 = x # p1初始化为x  
 grams\_basis = [p0, p1] # 存储多项式基  
  
 for i in range(2, degree + 1):  
 p2 = x \* p1 - self.beta(i - 1, i) \* p0 # 计算下一个多项式  
 grams\_basis.append(p2)  
 p0, p1 = p1, p2 # 更新p0和p1  
  
 return torch.cat(grams\_basis, dim=1) # 合并多项式基  
  
 def forward\_kag(self, x, group\_index):  
 # 前向传播，处理每个组的输入  
 basis = self.base\_conv[group\_index](self.base\_activation(x)) # 基础卷积  
  
 # 将x归一化到[-1, 1]范围  
 x = torch.tanh(x).contiguous()  
  
 if self.dropout is not None:  
 x = self.dropout(x) # 应用dropout  
  
 grams\_basis = self.base\_activation(self.gram\_poly(x, self.degree)) # 计算Gram多项式基  
  
 # 使用卷积权重函数进行卷积操作  
 y = self.conv\_w\_fun(grams\_basis, self.poly\_weights[group\_index],  
 stride=self.stride, dilation=self.dilation,  
 padding=self.padding, groups=1)  
  
 y = self.base\_activation(self.layer\_norm[group\_index](y + basis)) # 归一化和激活  
  
 return y  
  
 def forward(self, x):  
 # 前向传播，处理所有组的输入  
 split\_x = torch.split(x, self.inputdim // self.groups, dim=1) # 按组分割输入  
 output = []  
 for group\_ind, \_x in enumerate(split\_x):  
 y = self.forward\_kag(\_x.clone(), group\_ind) # 处理每个组  
 output.append(y.clone())  
 y = torch.cat(output, dim=1) # 合并输出  
 return y  
```  
  
### 代码说明：  
1. \*\*KAGNConvNDLayer类\*\*：这是一个自定义的卷积层，支持N维卷积（1D、2D、3D），结合了多项式特征和归一化。  
2. \*\*初始化方法\*\*：初始化输入输出维度、卷积参数、激活函数、dropout等，并创建相应的卷积层和归一化层。  
3. \*\*beta方法\*\*：计算多项式中的beta值。  
4. \*\*gram\_poly方法\*\*：计算Gram多项式，使用缓存以提高效率。  
5. \*\*forward\_kag方法\*\*：实现前向传播，处理每个组的输入，并进行卷积和归一化。  
6. \*\*forward方法\*\*：处理所有组的输入，将结果合并。```

这个程序文件定义了一个名为 `kagn\_conv.py` 的深度学习模块，主要用于实现具有可变阶数的卷积层，支持一维、二维和三维卷积。代码中使用了 PyTorch 框架，包含了多个类来实现不同维度的卷积层。  
  
首先，`KAGNConvNDLayer` 是一个基类，接受多个参数以初始化卷积层的属性，包括输入和输出维度、卷积核大小、组数、填充、步幅、扩张、丢弃率等。该类使用了一个可调用的函数 `conv\_w\_fun` 来执行卷积操作，并支持通过不同的归一化层进行数据处理。构造函数中会检查输入参数的有效性，并初始化卷积层和归一化层。  
  
在权重初始化方面，使用了 Kaiming 均匀分布来提高训练的起始效果。类中还定义了一个计算 Legendre 多项式的函数 `gram\_poly`，并使用了 `lru\_cache` 装饰器来缓存计算结果，以避免重复计算。  
  
`forward\_kag` 方法是该类的前向传播函数，首先对输入进行激活，然后进行线性变换和归一化，最后计算输出。它会将输入数据分组处理，并对每个组调用 `forward\_kag` 方法进行卷积操作。  
  
`forward` 方法则是处理整个输入数据的前向传播，首先将输入数据按组分割，然后对每个组调用 `forward\_kag`，最后将所有组的输出拼接在一起。  
  
接下来，`KAGNConv3DLayer`、`KAGNConv2DLayer` 和 `KAGNConv1DLayer` 是针对三维、二维和一维卷积的具体实现类，它们都继承自 `KAGNConvNDLayer`，并在初始化时指定了相应的卷积和归一化层。  
  
整体而言，这个模块提供了一种灵活的方式来构建具有多项式特性的卷积层，适用于不同维度的输入数据，能够在深度学习模型中实现更复杂的特征提取。

```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 math  
import itertools  
  
class Attention4D(nn.Module):  
 """4D注意力机制模块"""  
 def \_\_init\_\_(self, dim=384, key\_dim=32, num\_heads=8, attn\_ratio=4, resolution=7, act\_layer=nn.ReLU, stride=None):  
 super().\_\_init\_\_()  
 self.num\_heads = num\_heads # 注意力头的数量  
 self.scale = key\_dim \*\* -0.5 # 缩放因子  
 self.key\_dim = key\_dim # 键的维度  
 self.nh\_kd = key\_dim \* num\_heads # 每个头的键的总维度  
  
 # 如果有步幅，则调整分辨率并添加卷积层  
 if stride is not None:  
 self.resolution = math.ceil(resolution / stride)  
 self.stride\_conv = nn.Sequential(  
 nn.Conv2d(dim, dim, kernel\_size=3, stride=stride, padding=1, groups=dim),  
 nn.BatchNorm2d(dim),  
 )  
 self.upsample = nn.Upsample(scale\_factor=stride, mode='bilinear')  
 else:  
 self.resolution = resolution  
 self.stride\_conv = None  
 self.upsample = None  
  
 self.N = self.resolution \*\* 2 # 分辨率的平方  
 self.d = int(attn\_ratio \* key\_dim) # 输出维度  
 self.dh = self.d \* num\_heads # 总输出维度  
  
 # 定义查询、键、值的卷积层  
 self.q = nn.Sequential(nn.Conv2d(dim, self.num\_heads \* self.key\_dim, 1), nn.BatchNorm2d(self.num\_heads \* self.key\_dim))  
 self.k = nn.Sequential(nn.Conv2d(dim, self.num\_heads \* self.key\_dim, 1), nn.BatchNorm2d(self.num\_heads \* self.key\_dim))  
 self.v = nn.Sequential(nn.Conv2d(dim, self.num\_heads \* self.d, 1), nn.BatchNorm2d(self.num\_heads \* self.d))  
  
 # 定义局部值的卷积层  
 self.v\_local = nn.Sequential(  
 nn.Conv2d(self.num\_heads \* self.d, self.num\_heads \* self.d, kernel\_size=3, stride=1, padding=1, groups=self.num\_heads \* self.d),  
 nn.BatchNorm2d(self.num\_heads \* self.d),  
 )  
  
 # 定义投影层  
 self.proj = nn.Sequential(act\_layer(), nn.Conv2d(self.dh, dim, 1), nn.BatchNorm2d(dim))  
  
 # 计算注意力偏置  
 points = list(itertools.product(range(self.resolution), range(self.resolution)))  
 attention\_offsets = {}  
 idxs = []  
 for p1 in points:  
 for p2 in points:  
 offset = (abs(p1[0] - p2[0]), abs(p1[1] - p2[1]))  
 if offset not in attention\_offsets:  
 attention\_offsets[offset] = len(attention\_offsets)  
 idxs.append(attention\_offsets[offset])  
 self.attention\_biases = nn.Parameter(torch.zeros(num\_heads, len(attention\_offsets)))  
 self.register\_buffer('attention\_bias\_idxs', torch.LongTensor(idxs).view(self.N, self.N))  
  
 def forward(self, x):  
 """前向传播"""  
 B, C, H, W = x.shape # 获取输入的形状  
 if self.stride\_conv is not None:  
 x = self.stride\_conv(x) # 应用步幅卷积  
  
 # 计算查询、键、值  
 q = self.q(x).flatten(2).reshape(B, self.num\_heads, -1, self.N).permute(0, 1, 3, 2)  
 k = self.k(x).flatten(2).reshape(B, self.num\_heads, -1, self.N).permute(0, 1, 2, 3)  
 v = self.v(x)  
 v\_local = self.v\_local(v)  
 v = v.flatten(2).reshape(B, self.num\_heads, -1, self.N).permute(0, 1, 3, 2)  
  
 # 计算注意力  
 attn = (q @ k) \* self.scale + self.attention\_biases[:, self.attention\_bias\_idxs]  
 attn = attn.softmax(dim=-1) # 归一化  
 x = (attn @ v) # 应用注意力  
  
 out = x.transpose(2, 3).reshape(B, self.dh, self.resolution, self.resolution) + v\_local  
 if self.upsample is not None:  
 out = self.upsample(out) # 上采样  
  
 out = self.proj(out) # 投影到输出维度  
 return out  
  
  
class EfficientFormerV2(nn.Module):  
 """EfficientFormer V2模型"""  
 def \_\_init\_\_(self, layers, embed\_dims=None, mlp\_ratios=4, downsamples=None, num\_classes=1000):  
 super().\_\_init\_\_()  
 self.patch\_embed = nn.Conv2d(3, embed\_dims[0], kernel\_size=3, stride=2, padding=1) # 初始嵌入层  
  
 network = []  
 for i in range(len(layers)):  
 # 添加每个阶段的块  
 stage = self.\_build\_stage(embed\_dims[i], layers[i])  
 network.append(stage)  
 if downsamples[i] or embed\_dims[i] != embed\_dims[i + 1]:  
 # 添加下采样层  
 network.append(nn.Conv2d(embed\_dims[i], embed\_dims[i + 1], kernel\_size=3, stride=2, padding=1))  
  
 self.network = nn.ModuleList(network) # 将网络模块化  
  
 def \_build\_stage(self, dim, layers):  
 """构建一个阶段的块"""  
 blocks = []  
 for \_ in range(layers):  
 blocks.append(Attention4D(dim)) # 添加注意力块  
 return nn.Sequential(\*blocks)  
  
 def forward(self, x):  
 """前向传播"""  
 x = self.patch\_embed(x) # 嵌入  
 for block in self.network:  
 x = block(x) # 通过网络块  
 return x  
  
  
def efficientformerv2\_s0(weights='', \*\*kwargs):  
 """创建EfficientFormer V2 S0模型"""  
 model = EfficientFormerV2(  
 layers=[2, 2, 6, 4], # 每个阶段的层数  
 embed\_dims=[32, 48, 96, 176], # 嵌入维度  
 downsamples=[True, True, True, True], # 是否下采样  
 \*\*kwargs  
 )  
 if weights:  
 pretrained\_weight = torch.load(weights)['model']  
 model.load\_state\_dict(pretrained\_weight) # 加载预训练权重  
 return model  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 inputs = torch.randn((1, 3, 640, 640)) # 输入张量  
 model = efficientformerv2\_s0() # 创建模型  
 res = model(inputs) # 前向传播  
 print(res.size()) # 输出结果的形状  
```  
  
### 代码注释说明：  
1. \*\*Attention4D类\*\*：实现了一个4D注意力机制，包含查询、键、值的计算和注意力权重的应用。  
2. \*\*EfficientFormerV2类\*\*：构建了EfficientFormer V2模型，包含多个阶段的注意力块和下采样层。  
3. \*\*efficientformerv2\_s0函数\*\*：用于创建EfficientFormer V2 S0模型，并加载预训练权重。  
4. \*\*主程序\*\*：创建一个输入张量并通过模型进行前向传播，输出结果的形状。```

这个程序文件定义了一个名为 `EfficientFormerV2` 的深度学习模型，主要用于图像处理任务。该模型的设计灵感来源于高效的视觉变换器（Vision Transformer），并通过不同的模块和层次结构来提高性能和效率。  
  
首先，文件中定义了一些模型的超参数，包括不同规模的模型（如 S0、S1、S2 和 L）的宽度和深度。这些参数通过字典 `EfficientFormer\_width` 和 `EfficientFormer\_depth` 进行管理，便于后续模型的构建。  
  
接下来，文件中定义了多个类来实现模型的不同组件。`Attention4D` 类实现了一个四维注意力机制，支持多头注意力，并可以通过步幅进行下采样。`LGQuery` 类用于生成局部查询，`Attention4DDownsample` 类则实现了带下采样的四维注意力机制。  
  
`Embedding` 类负责将输入图像转换为嵌入特征，支持不同的卷积层和归一化层。`Mlp` 类实现了多层感知机（MLP），用于特征的进一步处理。`AttnFFN` 和 `FFN` 类则分别实现了带注意力机制的前馈网络和普通前馈网络。  
  
`eformer\_block` 函数用于构建模型的各个块，结合了注意力机制和前馈网络。`EfficientFormerV2` 类是模型的主类，负责整体结构的搭建，包括输入嵌入、各个块的组合以及下采样。  
  
在模型的初始化过程中，定义了不同层次的网络结构，并根据输入参数配置相应的层和模块。模型还支持加载预训练权重，以便在特定任务上进行微调。  
  
最后，文件中定义了一些函数，如 `efficientformerv2\_s0`、`efficientformerv2\_s1`、`efficientformerv2\_s2` 和 `efficientformerv2\_l`，用于创建不同规模的 EfficientFormerV2 模型实例，并可选择性地加载预训练权重。  
  
在 `\_\_main\_\_` 部分，程序会创建不同规模的模型实例，并对随机生成的输入进行前向传播，输出各层的特征图尺寸。这部分代码用于测试模型的构建和前向传播是否正常。  
  
总体而言，这个程序文件实现了一个灵活且高效的视觉变换器模型，适用于各种图像处理任务，并通过模块化设计提高了可维护性和可扩展性。

```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  
  
# 定义一个反向传播的自定义函数  
class ReverseFunction(torch.autograd.Function):  
 @staticmethod  
 def forward(ctx, run\_functions, alpha, \*args):  
 # 提取运行的函数和缩放因子  
 l0, l1, l2, l3 = run\_functions  
 alpha0, alpha1, alpha2, alpha3 = alpha  
   
 # 保存上下文信息  
 ctx.run\_functions = run\_functions  
 ctx.alpha = alpha  
   
 # 断开梯度追踪  
 with torch.no\_grad():  
 # 执行前向传播  
 x, c0, c1, c2, c3 = args  
 c0 = l0(x, c1) + c0 \* alpha0  
 c1 = l1(c0, c2) + c1 \* alpha1  
 c2 = l2(c1, c3) + c2 \* alpha2  
 c3 = l3(c2, None) + c3 \* alpha3  
   
 # 保存输入以便在反向传播时使用  
 ctx.save\_for\_backward(x, c0, c1, c2, c3)  
 return x, c0, c1, c2, c3  
  
 @staticmethod  
 def backward(ctx, \*grad\_outputs):  
 # 从上下文中恢复保存的张量  
 x, c0, c1, c2, c3 = ctx.saved\_tensors  
 l0, l1, l2, l3 = ctx.run\_functions  
 alpha0, alpha1, alpha2, alpha3 = ctx.alpha  
   
 # 提取梯度  
 gx\_right, g0\_right, g1\_right, g2\_right, g3\_right = grad\_outputs  
   
 # 计算每一层的梯度  
 # 这里省略了详细的计算过程，主要是使用保存的输入和运行的函数进行反向传播  
   
 # 返回各层的梯度  
 return None, None, gx\_up, g0\_left, g1\_left, g2\_left, g3\_left  
  
# 定义一个子网络  
class SubNet(nn.Module):  
 def \_\_init\_\_(self, channels, layers, kernel, first\_col, save\_memory) -> None:  
 super().\_\_init\_\_()  
 # 初始化缩放因子  
 self.alpha0 = nn.Parameter(torch.ones((1, channels[0], 1, 1)), requires\_grad=True)  
 self.alpha1 = nn.Parameter(torch.ones((1, channels[1], 1, 1)), requires\_grad=True)  
 self.alpha2 = nn.Parameter(torch.ones((1, channels[2], 1, 1)), requires\_grad=True)  
 self.alpha3 = nn.Parameter(torch.ones((1, channels[3], 1, 1)), requires\_grad=True)  
  
 # 创建网络层  
 self.level0 = Level(0, channels, layers, kernel, first\_col)  
 self.level1 = Level(1, channels, layers, kernel, first\_col)  
 self.level2 = Level(2, channels, layers, kernel, first\_col)  
 self.level3 = Level(3, channels, layers, kernel, first\_col)  
  
 def forward(self, \*args):  
 # 根据是否保存内存选择前向传播方式  
 if self.save\_memory:  
 return self.\_forward\_reverse(\*args)  
 else:  
 return self.\_forward\_nonreverse(\*args)  
  
# 定义主网络结构  
class RevCol(nn.Module):  
 def \_\_init\_\_(self, kernel='C2f', channels=[32, 64, 96, 128], layers=[2, 3, 6, 3], num\_subnet=5, save\_memory=True) -> None:  
 super().\_\_init\_\_()  
 self.num\_subnet = num\_subnet  
 self.channels = channels  
 self.layers = layers  
  
 # 初始化网络的输入层  
 self.stem = Conv(3, channels[0], k=4, s=4, p=0)  
  
 # 创建多个子网络  
 for i in range(num\_subnet):  
 first\_col = (i == 0)  
 self.add\_module(f'subnet{str(i)}', SubNet(channels, layers, kernel, first\_col, save\_memory=save\_memory))  
  
 def forward(self, x):  
 # 进行前向传播  
 c0, c1, c2, c3 = 0, 0, 0, 0  
 x = self.stem(x)   
 for i in range(self.num\_subnet):  
 c0, c1, c2, c3 = getattr(self, f'subnet{str(i)}')(x, c0, c1, c2, c3)   
 return [c0, c1, c2, c3]  
```  
  
### 代码注释说明：  
1. \*\*ReverseFunction\*\*: 这个类实现了一个自定义的反向传播函数，包含前向传播和反向传播的逻辑。前向传播计算了每一层的输出，并保存输入以便在反向传播时使用。  
   
2. \*\*SubNet\*\*: 这个类定义了一个子网络，其中包含多个层级和缩放因子。根据是否需要节省内存，选择不同的前向传播方法。  
  
3. \*\*RevCol\*\*: 这是主网络结构，包含多个子网络的组合。它负责初始化网络的输入层和各个子网络，并在前向传播时依次调用每个子网络。  
  
以上是对核心代码的简化和详细注释，便于理解其主要功能和结构。```

这个程序文件 `revcol.py` 是一个基于 PyTorch 的深度学习模型实现，主要用于构建一种称为“反向列”（Reverse Column）的网络结构。该网络结构通过反向传播的方式来优化模型的参数，具有节省内存的特点。  
  
首先，文件中导入了必要的 PyTorch 库以及一些自定义模块，包括卷积层和不同的网络块。接着，定义了一些辅助函数，用于处理 GPU 状态、设备管理、以及张量的分离和梯度计算等。  
  
`get\_gpu\_states` 函数用于获取当前 GPU 设备的随机数生成器状态，而 `get\_gpu\_device` 函数则用于从输入的张量中提取出使用的 GPU 设备。`set\_device\_states` 函数则是将保存的随机数状态重新设置到对应的设备上。`detach\_and\_grad` 函数用于分离输入张量并设置其需要计算梯度的标志。  
  
接下来，定义了一个名为 `ReverseFunction` 的类，继承自 `torch.autograd.Function`，用于实现自定义的前向和反向传播过程。在 `forward` 方法中，模型通过一系列的运行函数和缩放因子（alpha）来计算输出，并保存中间结果以便在反向传播时使用。反向传播的实现则在 `backward` 方法中完成，通过对保存的张量进行操作，计算出各个层的梯度。  
  
然后，定义了几个网络模块类，包括 `Fusion`、`Level` 和 `SubNet`。`Fusion` 类负责融合不同层的特征，`Level` 类则包含了融合层和多个卷积块。`SubNet` 类则是一个子网络的实现，包含多个层次的处理，并根据是否需要节省内存来选择不同的前向传播方式。  
  
最后，`RevCol` 类是整个模型的主体，初始化时定义了多个子网络，并通过 `forward` 方法将输入数据逐层传递。模型的输入经过一个卷积层（stem）处理后，依次通过每个子网络进行特征提取，最终输出多个特征图。  
  
整体来看，这个文件实现了一个复杂的深度学习模型，结合了反向传播和特征融合的思想，适用于需要高效内存管理的任务。

### 整体功能和构架概括  
  
该项目包含多个深度学习模块，主要用于图像处理和特征提取。每个文件实现了不同的网络结构和功能，形成了一个灵活且高效的深度学习框架。以下是各个文件的主要功能概述：  
  
1. \*\*cfpt.py\*\*：实现了跨层注意力机制，结合了空间和通道的信息，通过多层结构增强特征提取能力，适用于图像处理任务。  
   
2. \*\*kagn\_conv.py\*\*：定义了可变阶数的卷积层，支持一维、二维和三维卷积，灵活地处理不同维度的输入数据，适用于多种深度学习任务。  
  
3. \*\*EfficientFormerV2.py\*\*：构建了高效的视觉变换器模型，结合了注意力机制和前馈网络，优化了图像处理性能，支持不同规模的模型配置。  
  
4. \*\*revcol.py\*\*：实现了一种反向列网络结构，通过反向传播优化模型参数，具有节省内存的特点，适用于需要高效内存管理的深度学习任务。  
  
### 文件功能整理表  
  
| 文件名 | 功能描述 |  
|--------------------------|--------------------------------------------------------------------------------------------|  
| `cfpt.py` | 实现跨层注意力机制，增强特征提取能力，适用于图像处理任务。 |  
| `kagn\_conv.py` | 定义可变阶数的卷积层，支持一维、二维和三维卷积，灵活处理不同维度的输入数据。 |  
| `EfficientFormerV2.py` | 构建高效的视觉变换器模型，结合注意力机制和前馈网络，优化图像处理性能，支持不同规模的模型配置。 |  
| `revcol.py` | 实现反向列网络结构，通过反向传播优化模型参数，具有节省内存的特点，适用于高效内存管理的任务。 |  
  
这些模块可以结合使用，形成一个强大的深度学习框架，适用于多种计算机视觉和图像处理任务。

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。