# 改进yolo11-AKConv等200+全套创新点大全：起重机与载荷检测定位系统源码＆数据集全套

## 3.背景

研究背景与意义  
  
随着工业自动化和智能化的快速发展，起重机在建筑、物流和制造等领域的应用愈发广泛。起重机的安全性和效率直接影响到生产过程的顺利进行，因此，如何实现对起重机及其载荷的实时检测与定位，成为了当前研究的热点之一。传统的检测方法往往依赖于人工监控，不仅效率低下，而且容易受到人为因素的影响，存在一定的安全隐患。因此，基于计算机视觉的自动检测系统应运而生。  
  
在众多计算机视觉算法中，YOLO（You Only Look Once）系列因其高效的实时检测能力而备受关注。YOLOv11作为该系列的最新版本，具有更高的准确性和更快的处理速度，适合于复杂环境下的目标检测。然而，现有的YOLOv11模型在特定场景下的应用仍存在一定的局限性，尤其是在起重机与载荷的检测与定位方面。因此，改进YOLOv11以适应这一特定任务，具有重要的理论与实践意义。  
  
本研究将基于一个包含2600张图像的数据集，该数据集专注于起重机和载荷的检测，旨在通过改进YOLOv11模型，提升其在该领域的检测精度和鲁棒性。通过对图像数据的深入分析和模型的优化调整，我们期望能够实现对起重机及其载荷的高效、准确定位。这不仅能够提高起重机的操作安全性，还能在一定程度上降低因操作失误导致的事故风险，从而推动相关行业的安全生产和智能化发展。最终，本研究的成果将为智能起重机系统的推广应用提供有力的技术支持，为行业的进步贡献新的思路与方法。

## 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名为“vinc”，旨在为改进YOLOv11的起重机与载荷检测定位系统提供强有力的支持。该数据集包含两类主要对象，分别是“crane”（起重机）和“load”（载荷），共计类别数量为2。这一精简而明确的类别设置，使得模型在训练过程中能够专注于识别和定位这两种关键元素，从而提高检测的准确性和效率。  
  
“vinc”数据集的构建过程充分考虑了实际应用场景，涵盖了多种不同环境下的起重机与载荷图像。这些图像不仅包括了不同类型和型号的起重机，还涉及了多种形状、尺寸和颜色的载荷，确保了数据集的多样性和代表性。此外，数据集中还包含了不同光照条件、天气状况和视角下的图像，以增强模型的鲁棒性和适应性。这种多样化的图像来源将有助于模型在实际应用中更好地应对各种复杂情况。  
  
在数据标注方面，所有图像均经过精确标注，确保每个起重机和载荷的边界框位置准确无误。这一细致的标注工作为模型的训练提供了高质量的监督信号，使得YOLOv11能够在学习过程中有效捕捉到目标的特征。通过使用“vinc”数据集，研究团队期望能够显著提升起重机与载荷的检测精度，为后续的智能化施工和安全监控提供可靠的技术支持。  
  
综上所述，“vinc”数据集不仅为本项目的研究提供了丰富的素材，还为改进YOLOv11的检测能力奠定了坚实的基础。通过对该数据集的深入分析和应用，研究团队相信能够在起重机与载荷检测领域取得突破性进展。

## 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

```以下是经过简化和注释的核心代码，保留了最重要的部分，并对每个部分进行了详细的中文注释：  
  
```python  
import torch  
import torch.nn as nn  
import torch.nn.functional as F  
from timm.layers import weight\_init  
  
# 定义激活函数类  
class Activation(nn.ReLU):  
 def \_\_init\_\_(self, dim, act\_num=3, deploy=False):  
 super(Activation, self).\_\_init\_\_()  
 self.deploy = deploy  
 # 初始化权重  
 self.weight = torch.nn.Parameter(torch.randn(dim, 1, act\_num \* 2 + 1, act\_num \* 2 + 1))  
 self.bias = None  
 self.bn = nn.BatchNorm2d(dim, eps=1e-6) # 批归一化  
 self.dim = dim  
 self.act\_num = act\_num  
 weight\_init.trunc\_normal\_(self.weight, std=.02) # 权重初始化  
  
 def forward(self, x):  
 # 前向传播  
 if self.deploy:  
 return F.conv2d(  
 super(Activation, self).forward(x),   
 self.weight, self.bias, padding=(self.act\_num \* 2 + 1) // 2, groups=self.dim)  
 else:  
 return self.bn(F.conv2d(  
 super(Activation, self).forward(x),  
 self.weight, padding=self.act\_num, groups=self.dim))  
  
 def switch\_to\_deploy(self):  
 # 切换到部署模式，融合BN层  
 if not self.deploy:  
 kernel, bias = self.\_fuse\_bn\_tensor(self.weight, self.bn)  
 self.weight.data = kernel  
 self.bias = torch.nn.Parameter(torch.zeros(self.dim))  
 self.bias.data = bias  
 self.\_\_delattr\_\_('bn') # 删除bn属性  
 self.deploy = True  
  
 def \_fuse\_bn\_tensor(self, weight, bn):  
 # 融合权重和批归一化层  
 kernel = weight  
 running\_mean = bn.running\_mean  
 running\_var = bn.running\_var  
 gamma = bn.weight  
 beta = bn.bias  
 eps = bn.eps  
 std = (running\_var + eps).sqrt()  
 t = (gamma / std).reshape(-1, 1, 1, 1)  
 return kernel \* t, beta + (0 - running\_mean) \* gamma / std  
  
# 定义网络块  
class Block(nn.Module):  
 def \_\_init\_\_(self, dim, dim\_out, act\_num=3, stride=2, deploy=False):  
 super().\_\_init\_\_()  
 self.deploy = deploy  
 if self.deploy:  
 self.conv = nn.Conv2d(dim, dim\_out, kernel\_size=1) # 部署模式下的卷积  
 else:  
 self.conv1 = nn.Sequential(  
 nn.Conv2d(dim, dim, kernel\_size=1),  
 nn.BatchNorm2d(dim, eps=1e-6),  
 )  
 self.conv2 = nn.Sequential(  
 nn.Conv2d(dim, dim\_out, kernel\_size=1),  
 nn.BatchNorm2d(dim\_out, eps=1e-6)  
 )  
 self.pool = nn.MaxPool2d(stride) if stride != 1 else nn.Identity() # 池化层  
 self.act = Activation(dim\_out, act\_num) # 激活函数  
  
 def forward(self, x):  
 # 前向传播  
 if self.deploy:  
 x = self.conv(x)  
 else:  
 x = self.conv1(x)  
 x = F.leaky\_relu(x, negative\_slope=1) # 使用Leaky ReLU激活  
 x = self.conv2(x)  
  
 x = self.pool(x) # 池化  
 x = self.act(x) # 激活  
 return x  
  
 def switch\_to\_deploy(self):  
 # 切换到部署模式  
 if not self.deploy:  
 kernel, bias = self.\_fuse\_bn\_tensor(self.conv1[0], self.conv1[1])  
 self.conv = self.conv2[0] # 选择第二个卷积层  
 self.conv.weight.data = kernel  
 self.conv.bias.data = bias  
 self.\_\_delattr\_\_('conv1')  
 self.\_\_delattr\_\_('conv2')  
 self.act.switch\_to\_deploy() # 切换激活函数  
 self.deploy = True  
  
# 定义主网络结构  
class VanillaNet(nn.Module):  
 def \_\_init\_\_(self, in\_chans=3, num\_classes=1000, dims=[96, 192, 384, 768], strides=[2, 2, 2, 1], deploy=False):  
 super().\_\_init\_\_()  
 self.deploy = deploy  
 if self.deploy:  
 self.stem = nn.Sequential(  
 nn.Conv2d(in\_chans, dims[0], kernel\_size=4, stride=4),  
 Activation(dims[0])  
 )  
 else:  
 self.stem1 = nn.Sequential(  
 nn.Conv2d(in\_chans, dims[0], kernel\_size=4, stride=4),  
 nn.BatchNorm2d(dims[0], eps=1e-6),  
 )  
 self.stem2 = nn.Sequential(  
 nn.Conv2d(dims[0], dims[0], kernel\_size=1, stride=1),  
 nn.BatchNorm2d(dims[0], eps=1e-6),  
 Activation(dims[0])  
 )  
  
 self.stages = nn.ModuleList()  
 for i in range(len(strides)):  
 stage = Block(dim=dims[i], dim\_out=dims[i + 1], stride=strides[i], deploy=deploy)  
 self.stages.append(stage) # 添加网络块  
  
 def forward(self, x):  
 # 前向传播  
 if self.deploy:  
 x = self.stem(x)  
 else:  
 x = self.stem1(x)  
 x = F.leaky\_relu(x, negative\_slope=1)  
 x = self.stem2(x)  
  
 for stage in self.stages:  
 x = stage(x) # 逐层通过网络块  
 return x  
  
 def switch\_to\_deploy(self):  
 # 切换到部署模式  
 if not self.deploy:  
 self.stem2[2].switch\_to\_deploy() # 切换激活函数  
 self.deploy = True  
  
# 创建模型实例的函数  
def vanillanet\_10(pretrained='', \*\*kwargs):  
 model = VanillaNet(dims=[128 \* 4, 128 \* 4, 256 \* 4, 512 \* 4, 512 \* 4, 512 \* 4, 512 \* 4, 1024 \* 4], \*\*kwargs)  
 if pretrained:  
 weights = torch.load(pretrained)['model\_ema']  
 model.load\_state\_dict(weights) # 加载预训练权重  
 return model  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 inputs = torch.randn((1, 3, 640, 640)) # 输入数据  
 model = vanillanet\_10() # 创建模型  
 pred = model(inputs) # 前向传播  
 for i in pred:  
 print(i.size()) # 输出每层的尺寸  
```  
  
### 代码说明：  
1. \*\*Activation类\*\*：实现了自定义的激活函数，支持训练和部署模式，并融合了批归一化。  
2. \*\*Block类\*\*：表示网络中的一个基本块，包含卷积层、池化层和激活函数。  
3. \*\*VanillaNet类\*\*：主网络结构，包含多个Block，负责前向传播和权重加载。  
4. \*\*vanillanet\_10函数\*\*：用于创建VanillaNet模型的实例，并可选择加载预训练权重。  
5. \*\*主程序\*\*：创建输入数据，实例化模型并进行前向传播，输出每层的尺寸。```

该文件 `VanillaNet.py` 实现了一个名为 VanillaNet 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，包含多个模块和功能，以下是对其代码的逐步分析和说明。  
  
首先，文件中包含了版权声明和许可证信息，表明该程序是自由软件，可以在 MIT 许可证下进行再分发和修改。接着，导入了必要的库，包括 PyTorch 和一些用于初始化权重的函数。  
  
接下来，定义了一个名为 `activation` 的类，继承自 `nn.ReLU`。这个类的主要功能是实现一种自定义的激活函数，包含了权重和偏置的初始化，以及在前向传播时的不同处理方式（如部署模式和训练模式）。在 `forward` 方法中，根据是否处于部署模式，选择不同的卷积操作。该类还实现了批归一化的融合方法，以便在模型部署时减少计算量。  
  
然后，定义了一个名为 `Block` 的类，表示模型中的基本构建块。这个类的构造函数中根据输入和输出维度、步幅和是否处于部署模式来初始化卷积层和池化层。`forward` 方法中实现了前向传播的逻辑，处理输入数据并通过卷积和激活函数生成输出。该类同样实现了批归一化的融合方法和切换到部署模式的功能。  
  
接下来，定义了 `VanillaNet` 类，这是整个模型的核心。构造函数中根据输入通道数、类别数、各层的维度、丢弃率、激活函数数量、步幅等参数初始化模型的结构。模型的前向传播方法会依次通过各个阶段（即 `Block`），并在适当的尺度下保存特征图。  
  
此外，模型还包含了一些辅助方法，例如 `\_init\_weights` 用于初始化权重，`change\_act` 用于修改激活函数的学习率，`switch\_to\_deploy` 用于将模型切换到部署模式。  
  
文件中还定义了一些函数，如 `update\_weight` 用于更新模型的权重，`vanillanet\_5` 到 `vanillanet\_13\_x1\_5\_ada\_pool` 函数用于创建不同配置的 VanillaNet 模型，并可选择加载预训练权重。  
  
最后，在文件的主程序部分，创建了一个输入张量并实例化了 `vanillanet\_10` 模型，进行前向传播并打印输出特征图的尺寸。  
  
总体来说，该文件实现了一个灵活且高效的卷积神经网络模型，支持多种配置和预训练权重的加载，适用于图像分类等任务。通过自定义的激活函数和模块化的设计，模型在训练和部署时都能保持高效性。

```以下是经过简化和注释的核心代码部分，主要保留了 `CrossLayerSpatialAttention` 和 `CrossLayerChannelAttention` 类的实现，以及与其相关的关键方法和类。  
  
```python  
import torch  
import math  
import torch.nn as nn  
import torch.nn.functional as F  
  
class LayerNormProxy(nn.Module):  
 """自定义的LayerNorm代理，用于处理通道维度的归一化"""  
 def \_\_init\_\_(self, dim):  
 super().\_\_init\_\_()  
 self.norm = nn.LayerNorm(dim)  
  
 def forward(self, x):  
 # 将输入的形状从 (B, C, H, W) 转换为 (B, H, W, C)  
 x = x.permute(0, 2, 3, 1)  
 x = self.norm(x) # 进行LayerNorm  
 # 将形状转换回 (B, C, H, W)  
 return x.permute(0, 3, 1, 2)  
  
class CrossLayerPosEmbedding3D(nn.Module):  
 """跨层位置嵌入，计算相对位置偏置"""  
 def \_\_init\_\_(self, num\_heads=4, window\_size=(5, 3, 1), spatial=True):  
 super(CrossLayerPosEmbedding3D, self).\_\_init\_\_()  
 self.spatial = spatial  
 self.num\_heads = num\_heads  
 self.window\_size = window\_size  
  
 # 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.register\_buffer("relative\_position\_index", self.\_calculate\_relative\_position\_index())  
  
 def \_calculate\_relative\_position\_index(self):  
 # 计算相对位置索引的逻辑  
 # 省略具体实现，保持原有逻辑  
 pass  
  
 def forward(self):  
 # 计算位置嵌入  
 # 省略具体实现，保持原有逻辑  
 pass  
  
class CrossLayerSpatialAttention(nn.Module):  
 """跨层空间注意力机制"""  
 def \_\_init\_\_(self, in\_dim, layer\_num=3, beta=1, num\_heads=4, mlp\_ratio=2, reduction=4):  
 super(CrossLayerSpatialAttention, self).\_\_init\_\_()  
 self.num\_heads = num\_heads  
 self.reduction = reduction  
 self.window\_sizes = [(2 \*\* i + beta) if i != 0 else (2 \*\* i + beta - 1) for i in range(layer\_num)][::-1]  
 self.token\_num\_per\_layer = [i \*\* 2 for i in self.window\_sizes]  
 self.token\_num = sum(self.token\_num\_per\_layer)  
  
 # 初始化卷积层、归一化层和线性层  
 self.cpe = nn.ModuleList([ConvPosEnc(dim=in\_dim, k=3) for \_ in range(layer\_num)])  
 self.norm1 = nn.ModuleList(LayerNormProxy(in\_dim) for \_ in range(layer\_num))  
 self.norm2 = nn.ModuleList(nn.LayerNorm(in\_dim) for \_ in range(layer\_num))  
 self.qkv = nn.ModuleList(nn.Conv2d(in\_dim, in\_dim, kernel\_size=1) for \_ in range(layer\_num))  
 self.mlp = nn.ModuleList(Mlp(in\_features=in\_dim, hidden\_features=int(in\_dim \* mlp\_ratio)) for \_ in range(layer\_num))  
 self.softmax = nn.Softmax(dim=-1)  
 self.proj = nn.ModuleList(nn.Conv2d(in\_dim, in\_dim, kernel\_size=1) for \_ in range(layer\_num))  
 self.pos\_embed = CrossLayerPosEmbedding3D(num\_heads=num\_heads, window\_size=self.window\_sizes, spatial=True)  
  
 def forward(self, x\_list):  
 """前向传播，计算跨层空间注意力"""  
 # 省略具体实现，保持原有逻辑  
 pass  
  
class CrossLayerChannelAttention(nn.Module):  
 """跨层通道注意力机制"""  
 def \_\_init\_\_(self, in\_dim, layer\_num=3, alpha=1, num\_heads=4, mlp\_ratio=2, reduction=4):  
 super(CrossLayerChannelAttention, self).\_\_init\_\_()  
 self.num\_heads = num\_heads  
 self.reduction = reduction  
 self.hidden\_dim = in\_dim // reduction  
 self.window\_sizes = [(4 \*\* i + alpha) if i != 0 else (4 \*\* i + alpha - 1) for i in range(layer\_num)][::-1]  
 self.token\_num\_per\_layer = [i for i in self.window\_sizes]  
 self.token\_num = sum(self.token\_num\_per\_layer)  
  
 # 初始化卷积层、归一化层和线性层  
 self.cpe = nn.ModuleList([ConvPosEnc(dim=in\_dim, k=3) for \_ in range(layer\_num)])  
 self.norm1 = nn.ModuleList(LayerNormProxy(in\_dim) for \_ in range(layer\_num))  
 self.norm2 = nn.ModuleList(nn.LayerNorm(in\_dim) for \_ in range(layer\_num))  
 self.qkv = nn.ModuleList(nn.Conv2d(in\_dim, self.hidden\_dim \* 3, kernel\_size=1) for \_ in range(layer\_num))  
 self.softmax = nn.Softmax(dim=-1)  
 self.proj = nn.ModuleList(nn.Conv2d(self.hidden\_dim, in\_dim, kernel\_size=1) for \_ in range(layer\_num))  
 self.mlp = nn.ModuleList(Mlp(in\_features=in\_dim, hidden\_features=int(in\_dim \* mlp\_ratio)) for \_ in range(layer\_num))  
 self.pos\_embed = CrossLayerPosEmbedding3D(num\_heads=num\_heads, window\_size=self.window\_sizes, spatial=False)  
  
 def forward(self, x\_list):  
 """前向传播，计算跨层通道注意力"""  
 # 省略具体实现，保持原有逻辑  
 pass  
  
# 省略 ConvPosEnc 和 Mlp 类的实现，保持原有逻辑  
```  
  
### 代码说明  
1. \*\*LayerNormProxy\*\*: 自定义的 LayerNorm 类，用于在特定维度上进行归一化。  
2. \*\*CrossLayerPosEmbedding3D\*\*: 计算相对位置嵌入，使用了相对位置偏置表。  
3. \*\*CrossLayerSpatialAttention\*\*: 实现跨层空间注意力机制，包含了卷积层、归一化层、注意力计算等。  
4. \*\*CrossLayerChannelAttention\*\*: 实现跨层通道注意力机制，结构与空间注意力类似，但在通道维度上进行操作。  
  
此代码的具体实现细节（如某些方法的实现）已被省略，保留了主要结构和功能。```

这个程序文件 `cfpt.py` 定义了一个用于深度学习的模型，主要涉及跨层通道注意力和空间注意力机制。文件中使用了 PyTorch 框架，包含多个类和函数，具体功能如下：  
  
首先，导入了一些必要的库，包括 `torch`、`math`、`einops`、`torch.nn` 等，利用这些库来构建神经网络模型和实现各种操作。  
  
`LayerNormProxy` 类是一个简单的层归一化代理，它对输入进行重排，然后应用层归一化，最后再重排回原来的形状。这个类的主要作用是方便地对特定维度进行归一化处理。  
  
`CrossLayerPosEmbedding3D` 类实现了一个三维的跨层位置嵌入。它根据给定的窗口大小和头数，计算相对位置偏置，并生成绝对位置偏置。这个类在模型中用于增强特征的空间和通道信息，帮助模型更好地理解输入数据的结构。  
  
`ConvPosEnc` 类定义了一个卷积位置编码模块，它通过卷积操作来增强输入特征，并可选择性地应用激活函数。  
  
`DWConv` 类实现了深度卷积操作，适用于通道数较多的输入数据，能够有效提取特征。  
  
`Mlp` 类定义了一个多层感知机，包含两个线性层和一个激活函数，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`，这些函数用于处理输入特征的窗口划分和重组，支持重叠窗口的操作，适用于注意力机制的实现。  
  
`CrossLayerSpatialAttention` 类实现了跨层空间注意力机制。它包含多个层，每层都使用卷积位置编码、层归一化、注意力计算等模块。该类的 `forward` 方法接收多个输入特征图，计算注意力权重，并将结果通过一系列的变换和融合，输出处理后的特征图。  
  
`CrossLayerChannelAttention` 类实现了跨层通道注意力机制，结构与 `CrossLayerSpatialAttention` 类似，但它的注意力计算是基于通道的。该类同样使用卷积位置编码和层归一化，处理输入特征并计算注意力权重，最终输出经过注意力加权的特征图。  
  
整体来看，这个文件实现了一个复杂的深度学习模型，结合了空间和通道注意力机制，旨在提升模型对输入数据的理解和处理能力。通过模块化的设计，便于扩展和修改，适合用于图像处理等任务。

```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 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"""  
 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 LePEAttention(nn.Module):  
 """局部增强位置编码注意力模块"""  
 def \_\_init\_\_(self, dim, resolution, idx, split\_size=7, num\_heads=8, attn\_drop=0.):  
 super().\_\_init\_\_()  
 self.dim = dim # 输入特征维度  
 self.resolution = resolution # 输入分辨率  
 self.split\_size = split\_size # 切分窗口大小  
 self.num\_heads = num\_heads # 注意力头数  
 self.get\_v = nn.Conv2d(dim, dim, kernel\_size=3, stride=1, padding=1, groups=dim) # 卷积层用于获取V  
 self.attn\_drop = nn.Dropout(attn\_drop) # 注意力Dropout  
  
 def im2cswin(self, x):  
 """将输入图像转换为窗口"""  
 B, N, C = x.shape # B: 批量大小, N: 序列长度, C: 特征维度  
 H = W = int(np.sqrt(N)) # 计算图像的高度和宽度  
 x = x.transpose(-2, -1).contiguous().view(B, C, H, W) # 转换为(B, C, H, W)格式  
 x = img2windows(x, self.split\_size, self.split\_size) # 切分为窗口  
 return x  
  
 def forward(self, qkv):  
 """前向传播"""  
 q, k, v = qkv # 拆分查询、键、值  
 q = self.im2cswin(q) # 将q转换为窗口  
 k = self.im2cswin(k) # 将k转换为窗口  
 # 计算注意力  
 attn = (q @ k.transpose(-2, -1)) \* (self.dim \*\* -0.5) # 计算注意力得分  
 attn = F.softmax(attn, dim=-1) # 归一化  
 attn = self.attn\_drop(attn) # 应用Dropout  
 x = attn @ v # 加权求和  
 return x  
  
class CSWinBlock(nn.Module):  
 """CSWin Transformer的基本块"""  
 def \_\_init\_\_(self, dim, num\_heads, mlp\_ratio=4.):  
 super().\_\_init\_\_()  
 self.dim = dim # 输入特征维度  
 self.num\_heads = num\_heads # 注意力头数  
 self.qkv = nn.Linear(dim, dim \* 3) # 线性层用于生成q, k, v  
 self.attn = LePEAttention(dim, resolution=7, idx=0, num\_heads=num\_heads) # 注意力模块  
 self.mlp = Mlp(in\_features=dim, hidden\_features=int(dim \* mlp\_ratio)) # MLP模块  
  
 def forward(self, x):  
 """前向传播"""  
 qkv = self.qkv(x).reshape(x.shape[0], -1, 3, self.dim).permute(2, 0, 1, 3) # 生成qkv  
 x = self.attn(qkv) # 计算注意力  
 x = x + self.mlp(x) # 加上MLP的输出  
 return x  
  
class CSWinTransformer(nn.Module):  
 """CSWin Transformer模型"""  
 def \_\_init\_\_(self, img\_size=640, num\_classes=1000, embed\_dim=96, depth=[2, 2, 6, 2]):  
 super().\_\_init\_\_()  
 self.num\_classes = num\_classes # 类别数  
 self.embed\_dim = embed\_dim # 嵌入维度  
 self.stage1 = nn.ModuleList([CSWinBlock(embed\_dim, num\_heads=12) for \_ in range(depth[0])]) # 第一阶段  
 self.stage2 = nn.ModuleList([CSWinBlock(embed\_dim \* 2, num\_heads=24) for \_ in range(depth[1])]) # 第二阶段  
 self.stage3 = nn.ModuleList([CSWinBlock(embed\_dim \* 4, num\_heads=48) for \_ in range(depth[2])]) # 第三阶段  
 self.stage4 = nn.ModuleList([CSWinBlock(embed\_dim \* 8, num\_heads=96) for \_ in range(depth[3])]) # 第四阶段  
  
 def forward(self, x):  
 """前向传播"""  
 for stage in [self.stage1, self.stage2, self.stage3, self.stage4]:  
 for block in stage:  
 x = block(x) # 逐层传递  
 return x  
  
# 示例代码，创建模型并进行前向传播  
if \_\_name\_\_ == '\_\_main\_\_':  
 inputs = torch.randn((1, 3, 640, 640)) # 输入随机张量  
 model = CSWinTransformer() # 创建CSWin Transformer模型  
 res = model(inputs) # 前向传播  
 print(res.size()) # 输出结果的尺寸  
```  
  
### 代码核心部分说明：  
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. \*\*LePEAttention类\*\*：实现了局部增强位置编码的注意力机制，能够将输入转换为窗口并计算注意力。  
3. \*\*CSWinBlock类\*\*：构建了CSWin Transformer的基本模块，包含注意力机制和MLP。  
4. \*\*CSWinTransformer类\*\*：整体模型结构，包含多个CSWinBlock的堆叠，最终输出特征。  
  
### 注意：  
- 以上代码中省略了一些细节，例如权重初始化、模型的具体参数等，但保留了模型的核心结构和功能。```

这个程序文件 `CSWinTransformer.py` 实现了 CSWin Transformer 模型，主要用于计算机视觉任务，特别是图像分类。代码的结构分为多个部分，包括模型的定义、各个模块的实现以及一些辅助函数。  
  
首先，程序导入了必要的库，包括 PyTorch 和一些用于模型构建的工具。`timm` 库提供了一些预训练模型和图像处理的工具，`einops` 用于张量的重排列，`torch.utils.checkpoint` 用于模型的检查点功能，帮助减少内存使用。  
  
接下来，定义了几个重要的类。`Mlp` 类实现了一个多层感知机，包含两个线性层和一个激活函数（默认为 GELU），并在层之间使用了 dropout。`LePEAttention` 类实现了局部增强的自注意力机制，支持对输入进行分块处理，以便在计算注意力时能够更好地捕捉局部特征。它包含多个重要的方法，如 `im2cswin` 和 `get\_lepe`，用于将输入图像转换为窗口格式，并计算局部增强特征。  
  
`CSWinBlock` 类则是 CSWin Transformer 的基本构建块，包含了自注意力层和前馈网络。它的 `forward` 方法实现了输入的处理流程，包括归一化、计算 QKV（查询、键、值）向量、应用注意力机制和前馈网络。  
  
`img2windows` 和 `windows2img` 函数用于在图像和窗口之间进行转换，方便在注意力计算中处理局部区域。`Merge\_Block` 类用于合并不同分辨率的特征图，通常在模型的不同阶段使用。  
  
`CSWinTransformer` 类是整个模型的核心，负责构建模型的不同阶段，包括输入的卷积嵌入、多个 CSWinBlock 的堆叠以及特征的合并。模型的构造过程中，使用了深度学习中的随机深度（stochastic depth）技术，以提高模型的泛化能力。  
  
最后，文件定义了一些辅助函数，如 `\_conv\_filter` 和 `update\_weight`，用于处理模型权重的加载和更新。四个函数 `CSWin\_tiny`、`CSWin\_small`、`CSWin\_base` 和 `CSWin\_large` 用于创建不同规模的 CSWin Transformer 模型，并支持加载预训练权重。  
  
在文件的最后部分，提供了一个示例代码块，用于测试不同规模的模型。通过随机生成的输入数据，模型会输出特征图的尺寸，验证模型的构建是否正确。  
  
总体而言，这个文件实现了一个复杂的视觉 Transformer 模型，采用了多种先进的技术来提升性能，适用于各种计算机视觉任务。

```以下是经过简化和注释的核心代码部分，主要保留了重要的类和方法，并对其进行了详细的中文注释。  
  
```python  
import torch  
import torch.nn as nn  
from einops import rearrange  
  
# 定义一个二维层归一化类  
class LayerNorm2d(nn.Module):  
 def \_\_init\_\_(self, normalized\_shape, eps=1e-6, elementwise\_affine=True):  
 super().\_\_init\_\_()  
 # 使用 PyTorch 的 LayerNorm 进行归一化  
 self.norm = nn.LayerNorm(normalized\_shape, eps, elementwise\_affine)  
  
 def forward(self, x):  
 # 将输入张量的维度从 (B, C, H, W) 转换为 (B, H, W, C)  
 x = rearrange(x, 'b c h w -> b h w c').contiguous()  
 # 进行归一化  
 x = self.norm(x)  
 # 将维度转换回 (B, C, H, W)  
 x = rearrange(x, 'b h w c -> b c h w').contiguous()  
 return x  
  
# 定义一个交叉扫描的自定义操作  
class CrossScan(torch.autograd.Function):  
 @staticmethod  
 def forward(ctx, x: torch.Tensor):  
 B, C, H, W = x.shape  
 ctx.shape = (B, C, H, W)  
 xs = x.new\_empty((B, 4, C, H \* W))  
 # 将输入张量展开并进行转置和翻转  
 xs[:, 0] = x.flatten(2, 3)  
 xs[:, 1] = x.transpose(dim0=2, dim1=3).flatten(2, 3)  
 xs[:, 2:4] = torch.flip(xs[:, 0:2], dims=[-1])  
 return xs  
  
 @staticmethod  
 def backward(ctx, ys: torch.Tensor):  
 B, C, H, W = ctx.shape  
 L = H \* W  
 # 反向传播计算  
 ys = ys[:, 0:2] + ys[:, 2:4].flip(dims=[-1]).view(B, 2, -1, L)  
 y = ys[:, 0] + ys[:, 1].view(B, -1, W, H).transpose(dim0=2, dim1=3).contiguous().view(B, -1, L)  
 return y.view(B, -1, H, W)  
  
# 定义一个简单的卷积层  
class SimpleStem(nn.Module):  
 def \_\_init\_\_(self, inp, embed\_dim, ks=3):  
 super().\_\_init\_\_()  
 self.hidden\_dims = embed\_dim // 2  
 # 定义卷积层和激活函数  
 self.conv = nn.Sequential(  
 nn.Conv2d(inp, self.hidden\_dims, kernel\_size=ks, stride=2, padding=ks // 2, bias=False),  
 nn.BatchNorm2d(self.hidden\_dims),  
 nn.GELU(),  
 nn.Conv2d(self.hidden\_dims, embed\_dim, kernel\_size=ks, stride=2, padding=ks // 2, bias=False),  
 nn.BatchNorm2d(embed\_dim),  
 nn.SiLU(),  
 )  
  
 def forward(self, x):  
 # 前向传播  
 return self.conv(x)  
  
# 定义一个视觉线索合并层  
class VisionClueMerge(nn.Module):  
 def \_\_init\_\_(self, dim, out\_dim):  
 super().\_\_init\_\_()  
 self.hidden = int(dim \* 4)  
 # 定义逐点卷积层  
 self.pw\_linear = nn.Sequential(  
 nn.Conv2d(self.hidden, out\_dim, kernel\_size=1, stride=1, padding=0),  
 nn.BatchNorm2d(out\_dim),  
 nn.SiLU()  
 )  
  
 def forward(self, x):  
 # 将输入张量进行拼接  
 y = torch.cat([  
 x[..., ::2, ::2],  
 x[..., 1::2, ::2],  
 x[..., ::2, 1::2],  
 x[..., 1::2, 1::2]  
 ], dim=1)  
 return self.pw\_linear(y)  
```  
  
### 代码说明：  
1. \*\*LayerNorm2d\*\*: 这个类实现了二维层归一化，用于在卷积神经网络中对特征图进行归一化处理，以提高训练的稳定性和加速收敛。  
   
2. \*\*CrossScan\*\*: 这是一个自定义的 PyTorch 操作，负责对输入张量进行交叉扫描，生成多个视图以供后续处理。它的前向传播方法生成了一个新的张量，包含了输入张量的不同变换。  
  
3. \*\*SimpleStem\*\*: 这个类实现了一个简单的卷积层结构，通常用于特征提取的初始阶段。它包含两个卷积层和激活函数，逐步减少特征图的尺寸。  
  
4. \*\*VisionClueMerge\*\*: 这个类实现了一个视觉线索合并层，负责将输入特征图的不同部分进行拼接并通过逐点卷积进行处理，通常用于特征融合。  
  
这些核心部分构成了更复杂的网络结构的基础，能够用于各种计算机视觉任务。```

这个程序文件 `mamba\_yolo.py` 是一个用于实现 YOLO（You Only Look Once）目标检测模型的 PyTorch 代码。代码中定义了多个类和函数，主要包括神经网络的不同模块和操作。以下是对代码的详细说明。  
  
首先，代码导入了一些必要的库，包括 PyTorch、数学库、函数式编程工具和类型注解工具。然后，定义了一个 `LayerNorm2d` 类，该类实现了二维层归一化，适用于图像数据。它的 `forward` 方法将输入张量的形状从 `(batch, channels, height, width)` 转换为 `(batch, height, width, channels)`，然后进行归一化处理，最后再转换回原来的形状。  
  
接下来，定义了一个 `autopad` 函数，用于自动计算卷积操作的填充，以确保输出的形状与输入相同。然后，定义了 `CrossScan` 和 `CrossMerge` 类，这两个类实现了交叉扫描和交叉合并的操作，分别用于处理输入张量的不同维度。  
  
`SelectiveScanCore` 类实现了选择性扫描的核心功能，它通过自定义的前向和反向传播方法，利用 CUDA 加速计算。选择性扫描是一种高效的张量操作，适用于处理序列数据。`cross\_selective\_scan` 函数则封装了选择性扫描的具体实现，处理输入张量并进行必要的线性变换。  
  
接下来，定义了 `SS2D` 类，这是一个基于选择性扫描的二维神经网络模块。它包含多个参数，如模型维度、状态维度、卷积层参数等，并在 `forward` 方法中实现了前向传播逻辑。  
  
然后，定义了 `RGBlock` 和 `LSBlock` 类，这两个类实现了不同的网络块结构，分别用于特征提取和特征融合。`XSSBlock` 类则结合了选择性扫描和其他操作，构成了更复杂的网络结构。  
  
`VSSBlock\_YOLO` 类是 YOLO 模型的一个基本构建块，结合了选择性扫描和其他模块，进行特征提取和处理。`SimpleStem` 类实现了模型的初始卷积层，用于将输入图像转换为特征图。  
  
最后，`VisionClueMerge` 类用于将多个特征图合并为一个输出特征图，通常用于多尺度特征融合。  
  
整体来看，这个文件实现了 YOLO 模型的多个关键组件，利用选择性扫描和其他先进的神经网络技术，提高了模型的性能和效率。每个模块的设计都考虑到了深度学习中的常见操作，如卷积、归一化和激活函数，确保了模型的可扩展性和灵活性。

### 程序整体功能和构架概括  
  
该工程包含多个深度学习模型的实现，主要用于计算机视觉任务，包括图像分类和目标检测。每个文件实现了不同的模型结构和功能模块，采用了现代深度学习技术，如自注意力机制、卷积神经网络和选择性扫描等。整体架构模块化，便于扩展和修改，适合用于多种视觉任务。  
  
以下是每个文件的功能概述：  
  
| 文件名 | 功能描述 |  
|-----------------------|----------------------------------------------------------------------------------------------|  
| `VanillaNet.py` | 实现了 VanillaNet 模型，包含自定义激活函数和基本构建块，适用于图像分类任务。通过模块化设计，支持多种配置和预训练权重的加载。 |  
| `cfpt.py` | 实现了跨层通道和空间注意力机制，结合局部增强自注意力，提升模型对输入数据的理解能力，适用于图像处理任务。 |  
| `CSWinTransformer.py` | 实现了 CSWin Transformer 模型，采用多种先进技术（如随机深度和卷积位置编码），用于图像分类等计算机视觉任务。 |  
| `mamba\_yolo.py` | 实现了 YOLO 目标检测模型的多个关键组件，利用选择性扫描和其他模块，提高了模型的性能和效率，适用于实时目标检测。 |  
  
这些文件共同构成了一个强大的计算机视觉框架，支持多种模型和任务，能够满足不同的研究和应用需求。