# 改进yolo11-ASF等200+全套创新点大全：骨折检测系统源码＆数据集全套

## 3.背景

研究背景与意义  
  
骨折作为一种常见的骨骼损伤，其诊断和治疗对患者的康复至关重要。传统的骨折检测方法主要依赖于医生的经验和影像学检查，如X光、CT等，这不仅耗时，而且容易受到主观因素的影响。随着计算机视觉和深度学习技术的迅猛发展，基于人工智能的自动化骨折检测系统逐渐成为研究热点。特别是YOLO（You Only Look Once）系列模型因其高效的实时检测能力，已被广泛应用于各类物体检测任务中。  
  
本研究旨在基于改进的YOLOv11模型，构建一个高效的骨折检测系统。我们采用了一个包含4000张图像的数据集，专注于骨骼类别的检测。该数据集经过精心标注，确保了模型训练的准确性和有效性。通过对YOLOv11模型的改进，我们希望在检测精度和速度上实现显著提升，以满足临床应用的需求。  
  
在骨折检测领域，准确的早期诊断可以显著提高治疗效果，减少并发症的发生。我们的研究不仅有助于提高骨折检测的自动化水平，还能为医生提供辅助决策支持，减轻其工作负担。此外，基于深度学习的骨折检测系统能够通过不断学习和优化，适应不同类型的骨折情况，从而在实际应用中展现出更强的灵活性和适应性。  
  
总之，基于改进YOLOv11的骨折检测系统的研究，不仅具有重要的学术价值，也对临床医学实践具有深远的影响。通过提高骨折检测的效率和准确性，我们期望为患者提供更优质的医疗服务，推动骨科领域的技术进步。

## 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  
  
本项目旨在开发和改进基于YOLOv11的骨折检测系统，为此我们构建了一个专门的数据集，以支持深度学习模型的训练和评估。该数据集的核心主题为“Convert COCO JSON to Yolov7 TXT”，通过这一过程，我们将COCO格式的数据转换为YOLO所需的TXT格式，以便于更高效地进行模型训练和推理。数据集中包含的类别数量为1，具体类别为“bone”，这表明我们的模型专注于识别和检测骨折相关的图像信息。  
  
在数据集的构建过程中，我们精心挑选了大量的医学影像数据，包括X光片、CT扫描图像等，这些图像均标注了骨折的具体位置和类型。通过对这些数据的标注，我们确保了模型能够学习到骨折的特征，从而在实际应用中实现高效、准确的检测。为了提高模型的泛化能力，我们的数据集涵盖了不同类型的骨折，包括但不限于横断骨折、斜骨折和粉碎骨折等。这种多样性使得模型在面对不同病例时，能够更好地适应和识别。  
  
此外，为了增强数据集的鲁棒性，我们还进行了数据增强处理，包括旋转、缩放、翻转等操作。这些处理不仅丰富了数据集的多样性，还有效提升了模型在真实场景中的表现。通过对数据集的不断优化和扩充，我们期望最终构建出一个能够在临床环境中提供高效、准确骨折检测的系统，为医疗工作者提供有力的支持，提升骨折诊断的效率和准确性。

## 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：  
  
```python  
import math  
import torch  
import torch.nn as nn  
import torch.nn.functional as F  
  
\_\_all\_\_ = ['ReparamLargeKernelConv']  
  
def get\_conv2d(in\_channels, out\_channels, kernel\_size, stride, padding, dilation, groups, bias):  
 """  
 创建一个2D卷积层，返回nn.Conv2d对象。  
 """  
 paddings = (kernel\_size[0] // 2, kernel\_size[1] // 2) # 计算填充  
 return nn.Conv2d(  
 in\_channels, out\_channels, kernel\_size, stride, paddings, dilation, groups, bias  
 )  
  
def get\_bn(channels):  
 """  
 创建一个Batch Normalization层，返回nn.BatchNorm2d对象。  
 """  
 return nn.BatchNorm2d(channels)  
  
class Mask(nn.Module):  
 """  
 自定义Mask类，用于生成可学习的权重掩码。  
 """  
 def \_\_init\_\_(self, size):  
 super().\_\_init\_\_()  
 self.weight = torch.nn.Parameter(data=torch.Tensor(\*size), requires\_grad=True) # 初始化权重  
 self.weight.data.uniform\_(-1, 1) # 权重初始化为[-1, 1]之间的均匀分布  
  
 def forward(self, x):  
 w = torch.sigmoid(self.weight) # 对权重应用sigmoid函数  
 masked\_wt = w.mul(x) # 将权重应用于输入x  
 return masked\_wt  
  
class ReparamLargeKernelConv(nn.Module):  
 """  
 自定义卷积层，支持大核卷积的重参数化。  
 """  
 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 # 计算填充  
   
 # 根据是否合并小卷积来选择卷积层  
 if small\_kernel\_merged:  
 self.lkb\_reparam = get\_conv2d(  
 in\_channels=in\_channels,  
 out\_channels=out\_channels,  
 kernel\_size=kernel\_size,  
 stride=stride,  
 padding=padding,  
 dilation=1,  
 groups=groups,  
 bias=True,  
 )  
 else:  
 if self.Decom:  
 self.LoRA = conv\_bn(  
 in\_channels=in\_channels,  
 out\_channels=out\_channels,  
 kernel\_size=(kernel\_size, small\_kernel),  
 stride=stride,  
 padding=padding,  
 dilation=1,  
 groups=groups,  
 bn=bn  
 )  
 else:  
 self.lkb\_origin = conv\_bn(  
 in\_channels=in\_channels,  
 out\_channels=out\_channels,  
 kernel\_size=kernel\_size,  
 stride=stride,  
 padding=padding,  
 dilation=1,  
 groups=groups,  
 bn=bn,  
 )  
  
 if (small\_kernel is not None) and small\_kernel < kernel\_size:  
 self.small\_conv = conv\_bn(  
 in\_channels=in\_channels,  
 out\_channels=out\_channels,  
 kernel\_size=small\_kernel,  
 stride=stride,  
 padding=small\_kernel // 2,  
 groups=groups,  
 dilation=1,  
 bn=bn,  
 )  
   
 self.bn = get\_bn(out\_channels) # 创建Batch Normalization层  
 self.act = nn.SiLU() # 使用SiLU激活函数  
  
 def forward(self, inputs):  
 """  
 前向传播函数，根据是否使用重参数化卷积进行计算。  
 """  
 if hasattr(self, "lkb\_reparam"):  
 out = self.lkb\_reparam(inputs) # 使用重参数化卷积  
 elif self.Decom:  
 out = self.LoRA(inputs) # 使用LoRA卷积  
 if hasattr(self, "small\_conv"):  
 out += self.small\_conv(inputs) # 加上小卷积的输出  
 else:  
 out = self.lkb\_origin(inputs) # 使用原始卷积  
 if hasattr(self, "small\_conv"):  
 out += self.small\_conv(inputs) # 加上小卷积的输出  
 return self.act(self.bn(out)) # 返回经过BN和激活函数的输出  
  
 def get\_equivalent\_kernel\_bias(self):  
 """  
 获取等效的卷积核和偏置，用于将Batch Normalization融合到卷积层中。  
 """  
 eq\_k, eq\_b = fuse\_bn(self.lkb\_origin.conv, self.lkb\_origin.bn) # 融合BN  
 if hasattr(self, "small\_conv"):  
 small\_k, small\_b = fuse\_bn(self.small\_conv.conv, self.small\_conv.bn) # 融合小卷积的BN  
 eq\_b += small\_b # 更新偏置  
 eq\_k += nn.functional.pad(small\_k, [(self.kernel\_size - self.small\_kernel) // 2] \* 4) # 填充小卷积核  
 return eq\_k, eq\_b  
  
 def switch\_to\_deploy(self):  
 """  
 切换到部署模式，替换为重参数化卷积。  
 """  
 if hasattr(self, 'lkb\_origin'):  
 eq\_k, eq\_b = self.get\_equivalent\_kernel\_bias() # 获取等效卷积核和偏置  
 self.lkb\_reparam = get\_conv2d(  
 in\_channels=self.lkb\_origin.conv.in\_channels,  
 out\_channels=self.lkb\_origin.conv.out\_channels,  
 kernel\_size=self.lkb\_origin.conv.kernel\_size,  
 stride=self.lkb\_origin.conv.stride,  
 padding=self.lkb\_origin.conv.padding,  
 dilation=self.lkb\_origin.conv.dilation,  
 groups=self.lkb\_origin.conv.groups,  
 bias=True,  
 )  
 self.lkb\_reparam.weight.data = eq\_k # 设置卷积核权重  
 self.lkb\_reparam.bias.data = eq\_b # 设置偏置  
 self.\_\_delattr\_\_("lkb\_origin") # 删除原始卷积属性  
 if hasattr(self, "small\_conv"):  
 self.\_\_delattr\_\_("small\_conv") # 删除小卷积属性  
```  
  
### 代码核心部分分析：  
1. \*\*卷积层和BN的创建\*\*：`get\_conv2d`和`get\_bn`函数用于创建卷积层和Batch Normalization层。  
2. \*\*Mask类\*\*：用于生成可学习的权重掩码，应用于输入数据。  
3. \*\*ReparamLargeKernelConv类\*\*：这是主要的卷积层实现，支持大核卷积的重参数化，包含了对小卷积的支持。  
4. \*\*前向传播\*\*：`forward`方法根据不同的配置选择合适的卷积层进行计算，并返回经过激活函数处理的输出。  
5. \*\*等效卷积核和偏置的获取\*\*：`get\_equivalent\_kernel\_bias`方法用于在融合BN后获取等效的卷积核和偏置。  
6. \*\*部署模式切换\*\*：`switch\_to\_deploy`方法用于将模型切换到部署模式，替换为重参数化的卷积层，以提高推理效率。  
  
通过以上分析和注释，可以更好地理解代码的结构和功能。```

这个文件 `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的情况，并结合了小卷积核的特性，以提高计算效率和灵活性。文件中定义了多个类和函数，下面是对其功能的逐步分析。  
  
首先，文件导入了必要的库，包括 `math` 和 `torch` 相关的模块。接着，定义了一个名为 `get\_conv2d` 的函数，用于创建一个标准的二维卷积层。这个函数接受多个参数，包括输入通道数、输出通道数、卷积核大小、步幅、填充、扩张率、分组数和是否使用偏置等。它会根据给定的卷积核大小计算填充，并返回一个 `nn.Conv2d` 对象。  
  
接下来，定义了一个 `Mask` 类，这个类用于创建一个可学习的权重掩码。掩码的权重通过 Sigmoid 函数进行归一化，然后与输入进行逐元素相乘，从而实现对输入的加权。  
  
`conv\_bn\_ori` 函数用于创建一个包含卷积层和可选的批归一化层的序列。这个函数会根据输入参数构建一个 `nn.Sequential` 对象，首先添加卷积层，然后根据需要添加批归一化层。  
  
`LoRAConvsByWeight` 类实现了一种新的卷积结构，旨在通过权重的方式合并两个不同大小的卷积核。它在初始化时接受多个参数，包括输入和输出通道数、卷积核大小、步幅、分组数等。该类的核心在于其 `forward` 方法，它通过对输入进行卷积，然后将输出分为两个部分，分别通过不同的掩码进行处理，最后将结果相加。  
  
`forward\_lora` 方法负责处理经过掩码后的输出，并根据输入的高度和宽度进行数据重排。`rearrange\_data` 方法则负责根据卷积核的大小和步幅对数据进行适当的填充和裁剪，以确保输出的形状符合预期。  
  
`shift` 方法用于计算在进行卷积时所需的填充量和窗口的起始索引，确保卷积操作不会改变特征图的大小。  
  
`conv\_bn` 函数是一个工厂函数，根据传入的卷积核大小决定使用 `conv\_bn\_ori` 还是 `LoRAConvsByWeight`。  
  
`fuse\_bn` 函数用于将卷积层和批归一化层融合，以减少推理时的计算开销。  
  
最后，`ReparamLargeKernelConv` 类是文件的核心部分，它实现了一个可重参数化的大卷积核卷积层。这个类的构造函数接受多个参数，并根据这些参数初始化相应的卷积层和小卷积层。`forward` 方法定义了前向传播的过程，计算最终的输出。  
  
`get\_equivalent\_kernel\_bias` 方法用于获取融合后的卷积核和偏置，`switch\_to\_deploy` 方法则用于在推理阶段切换到更高效的卷积层配置。  
  
总体而言，这个文件实现了一个灵活且高效的卷积层结构，适用于需要处理大卷积核的深度学习模型，同时结合了小卷积核的优势，以提高计算效率。

```以下是保留的核心代码部分，并添加了详细的中文注释：  
  
```python  
import torch  
import torch.nn as nn  
import torch.nn.functional as F  
from timm.layers import 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.register\_buffer("relative\_position\_index", self.calculate\_relative\_position\_index(window\_size))  
 trunc\_normal\_(self.relative\_position\_bias\_table, std=.02) # 初始化相对位置偏置  
  
 # 初始化绝对位置偏置  
 self.absolute\_position\_bias = nn.Parameter(torch.zeros(len(window\_size), num\_heads, 1, 1, 1))  
 trunc\_normal\_(self.absolute\_position\_bias, std=.02)  
  
 def calculate\_relative\_position\_index(self, window\_size):  
 # 计算相对位置索引的逻辑  
 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\_flatten = torch.cat([torch.flatten(coord) for coord in torch.meshgrid(coords\_h, coords\_w)], 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):  
 # 前向传播计算位置嵌入  
 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  
 pos\_embed = pos\_embed.reshape(1, 1, -1, -1, self.num\_heads).permute(0, 4, 1, 2, 3)  
 return pos\_embed + self.absolute\_position\_bias  
  
class CrossLayerSpatialAttention(nn.Module):  
 def \_\_init\_\_(self, in\_dim, layer\_num=3, num\_heads=4):  
 super(CrossLayerSpatialAttention, self).\_\_init\_\_()  
 self.num\_heads = num\_heads # 注意力头的数量  
 self.hidden\_dim = in\_dim // 4 # 隐藏层维度  
 self.qkv = nn.Conv2d(in\_dim, self.hidden\_dim \* 3, kernel\_size=1) # 线性变换生成Q、K、V  
 self.softmax = nn.Softmax(dim=-1) # Softmax层  
 self.pos\_embed = CrossLayerPosEmbedding3D(num\_heads=num\_heads) # 位置嵌入  
  
 def forward(self, x\_list):  
 q\_list, k\_list, v\_list = [], [], []  
 for x in x\_list:  
 qkv = self.qkv(x) # 计算Q、K、V  
 q, k, v = qkv.chunk(3, dim=1) # 分割Q、K、V  
 q\_list.append(q)  
 k\_list.append(k)  
 v\_list.append(v)  
  
 # 将所有Q、K、V拼接  
 q\_stack = torch.cat(q\_list, dim=1)  
 k\_stack = torch.cat(k\_list, dim=1)  
 v\_stack = torch.cat(v\_list, dim=1)  
  
 # 计算注意力  
 attn = F.normalize(q\_stack, dim=-1) @ F.normalize(k\_stack, dim=-1).transpose(-1, -2)  
 attn = attn + self.pos\_embed() # 加入位置嵌入  
 attn = self.softmax(attn) # 应用Softmax  
  
 # 计算输出  
 out = attn @ v\_stack  
 return out  
  
# 代码的核心部分是 CrossLayerPosEmbedding3D 和 CrossLayerSpatialAttention 类。  
# CrossLayerPosEmbedding3D 负责计算位置嵌入，而 CrossLayerSpatialAttention 负责实现跨层的空间注意力机制。  
```  
  
以上代码保留了最核心的部分，主要是与位置嵌入和空间注意力相关的功能，并添加了详细的中文注释以便理解。```

这个程序文件 `cfpt.py` 实现了一个深度学习模型中的跨层注意力机制，主要包括两个类：`CrossLayerSpatialAttention` 和 `CrossLayerChannelAttention`。这两个类通过不同的方式处理输入特征图，以实现跨层的信息交互和特征增强。  
  
首先，文件中引入了一些必要的库，包括 `torch`、`math`、`einops` 和 `torch.nn` 等。这些库提供了构建神经网络所需的基础功能和操作。  
  
`LayerNormProxy` 类是一个简单的层归一化实现，它将输入的张量进行维度重排，然后应用 `nn.LayerNorm`，最后再重排回原来的形状。这种处理方式有助于在处理图像数据时保持通道的顺序。  
  
`CrossLayerPosEmbedding3D` 类用于生成跨层的位置信息嵌入。它根据给定的窗口大小和头数，计算相对位置偏置，并将其存储为可训练的参数。这个类的 `forward` 方法会生成位置嵌入，考虑到不同层之间的相对位置关系。  
  
`ConvPosEnc` 类实现了一个卷积位置编码模块，它通过卷积操作对输入特征进行处理，并可选择性地应用激活函数。  
  
`DWConv` 类实现了深度可分离卷积，用于在特征图上进行卷积操作，以减少计算量。  
  
`Mlp` 类实现了一个简单的多层感知机，包括两个线性层和一个激活函数。  
  
接下来，定义了一些辅助函数，如 `overlaped\_window\_partition` 和 `overlaped\_window\_reverse`，用于处理特征图的窗口划分和重组。这些函数在注意力机制中用于提取局部特征。  
  
`CrossLayerSpatialAttention` 类实现了空间注意力机制。它接受多个层的输入特征图，并通过卷积、注意力计算和归一化等步骤，生成增强后的特征图。这个类的 `forward` 方法中，首先对每一层的输入进行处理，计算查询、键、值，然后通过注意力机制聚合信息，最后将结果与原始特征图相加并经过归一化和激活函数处理。  
  
`CrossLayerChannelAttention` 类实现了通道注意力机制，结构与空间注意力类似，但它在处理特征时更关注通道之间的信息交互。这个类的 `forward` 方法同样处理输入特征图，计算注意力权重，并通过反向操作恢复特征图的形状。  
  
总体而言，这个程序文件实现了一个复杂的跨层注意力机制，旨在通过空间和通道的注意力计算，增强特征提取的能力，适用于各种深度学习任务，尤其是在计算机视觉领域。

```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  
from torch.autograd import Function  
import pywt  
  
# 创建小波滤波器  
def create\_wavelet\_filter(wave, in\_size, out\_size, type=torch.float):  
 # 使用pywt库创建小波对象  
 w = pywt.Wavelet(wave)  
 # 获取小波的分解高通和低通滤波器，并反转  
 dec\_hi = torch.tensor(w.dec\_hi[::-1], dtype=type)  
 dec\_lo = torch.tensor(w.dec\_lo[::-1], dtype=type)  
   
 # 创建分解滤波器  
 dec\_filters = torch.stack([  
 dec\_lo.unsqueeze(0) \* dec\_lo.unsqueeze(1),  
 dec\_lo.unsqueeze(0) \* dec\_hi.unsqueeze(1),  
 dec\_hi.unsqueeze(0) \* dec\_lo.unsqueeze(1),  
 dec\_hi.unsqueeze(0) \* dec\_hi.unsqueeze(1)  
 ], dim=0)  
  
 # 扩展滤波器以适应输入通道数  
 dec\_filters = dec\_filters[:, None].repeat(in\_size, 1, 1, 1)  
  
 # 获取重构高通和低通滤波器，并反转  
 rec\_hi = torch.tensor(w.rec\_hi[::-1], dtype=type).flip(dims=[0])  
 rec\_lo = torch.tensor(w.rec\_lo[::-1], dtype=type).flip(dims=[0])  
   
 # 创建重构滤波器  
 rec\_filters = torch.stack([  
 rec\_lo.unsqueeze(0) \* rec\_lo.unsqueeze(1),  
 rec\_lo.unsqueeze(0) \* rec\_hi.unsqueeze(1),  
 rec\_hi.unsqueeze(0) \* rec\_lo.unsqueeze(1),  
 rec\_hi.unsqueeze(0) \* rec\_hi.unsqueeze(1)  
 ], dim=0)  
  
 # 扩展滤波器以适应输出通道数  
 rec\_filters = rec\_filters[:, None].repeat(out\_size, 1, 1, 1)  
  
 return dec\_filters, rec\_filters  
  
# 小波变换函数  
def wavelet\_transform(x, filters):  
 b, c, h, w = x.shape # 获取输入的形状  
 pad = (filters.shape[2] // 2 - 1, filters.shape[3] // 2 - 1) # 计算填充  
 # 进行2D卷积操作，进行小波变换  
 x = F.conv2d(x, filters.to(x.dtype).to(x.device), stride=2, groups=c, padding=pad)  
 x = x.reshape(b, c, 4, h // 2, w // 2) # 重新调整形状  
 return x  
  
# 逆小波变换函数  
def inverse\_wavelet\_transform(x, filters):  
 b, c, \_, h\_half, w\_half = x.shape # 获取输入的形状  
 pad = (filters.shape[2] // 2 - 1, filters.shape[3] // 2 - 1) # 计算填充  
 x = x.reshape(b, c \* 4, h\_half, w\_half) # 重新调整形状  
 # 进行转置卷积操作，进行逆小波变换  
 x = F.conv\_transpose2d(x, filters.to(x.dtype).to(x.device), stride=2, groups=c, padding=pad)  
 return x  
  
# 定义小波变换的类  
class WaveletTransform(Function):  
 @staticmethod  
 def forward(ctx, input, filters):  
 ctx.filters = filters # 保存滤波器  
 with torch.no\_grad():  
 x = wavelet\_transform(input, filters) # 进行小波变换  
 return x  
  
 @staticmethod  
 def backward(ctx, grad\_output):  
 grad = inverse\_wavelet\_transform(grad\_output, ctx.filters) # 计算梯度  
 return grad, None  
  
# 定义卷积层类  
class WTConv2d(nn.Module):  
 def \_\_init\_\_(self, in\_channels, out\_channels, kernel\_size=5, stride=1, bias=True, wt\_levels=1, wt\_type='db1'):  
 super(WTConv2d, self).\_\_init\_\_()  
  
 assert in\_channels == out\_channels # 输入通道数和输出通道数必须相等  
  
 self.in\_channels = in\_channels  
 self.wt\_levels = wt\_levels  
 self.stride = stride  
  
 # 创建小波滤波器  
 self.wt\_filter, self.iwt\_filter = create\_wavelet\_filter(wt\_type, in\_channels, in\_channels, torch.float)  
 self.wt\_filter = nn.Parameter(self.wt\_filter, requires\_grad=False) # 不需要训练的小波滤波器  
 self.iwt\_filter = nn.Parameter(self.iwt\_filter, requires\_grad=False) # 不需要训练的逆小波滤波器  
  
 # 基础卷积层  
 self.base\_conv = nn.Conv2d(in\_channels, in\_channels, kernel\_size, padding='same', stride=1, groups=in\_channels, bias=bias)  
  
 def forward(self, x):  
 # 进行小波变换和逆小波变换的前向传播  
 # 省略具体实现细节  
 return x # 返回输出  
```  
  
### 代码核心部分解释：  
1. \*\*创建小波滤波器\*\*：`create\_wavelet\_filter` 函数用于生成小波变换和逆变换所需的滤波器。  
2. \*\*小波变换和逆变换\*\*：`wavelet\_transform` 和 `inverse\_wavelet\_transform` 函数实现了小波变换和逆变换的核心操作。  
3. \*\*小波变换类\*\*：`WaveletTransform` 类定义了小波变换的前向和反向传播方法。  
4. \*\*卷积层\*\*：`WTConv2d` 类是一个自定义的卷积层，结合了小波变换的特性，能够在卷积操作中使用小波变换。```

这个程序文件 `wtconv2d.py` 实现了一个基于小波变换的二维卷积层，名为 `WTConv2d`。该文件主要使用 PyTorch 框架，结合小波变换来处理输入数据，提供了一种新的卷积操作方式。  
  
首先，文件导入了必要的库，包括 PyTorch 的核心库、神经网络模块、功能模块以及小波变换库 `pywt`。接着，定义了一个函数 `create\_wavelet\_filter`，该函数用于创建小波变换的滤波器。它接受小波类型、输入通道数和输出通道数作为参数，并生成相应的滤波器。滤波器分为降采样和重构两种，分别用于小波变换和逆小波变换。  
  
接下来，定义了两个函数 `wavelet\_transform` 和 `inverse\_wavelet\_transform`，分别用于执行小波变换和逆小波变换。这两个函数使用了 PyTorch 的卷积操作，处理输入张量并返回变换后的结果。  
  
随后，定义了两个类 `WaveletTransform` 和 `InverseWaveletTransform`，这两个类继承自 `Function`，实现了小波变换和逆小波变换的前向和反向传播。它们的 `forward` 方法调用了之前定义的变换函数，而 `backward` 方法则计算梯度。  
  
在 `WTConv2d` 类中，构造函数初始化了输入和输出通道数、小波变换的层数、步幅等参数。通过调用 `create\_wavelet\_filter` 函数生成小波滤波器，并将其转化为不可训练的参数。接着，初始化了小波变换和逆小波变换的函数，并创建了基础卷积层和一系列小波卷积层。  
  
在 `forward` 方法中，首先对输入数据进行小波变换，分离出低频和高频成分。然后，依次通过每一层的小波卷积处理这些成分，并在最后进行逆小波变换，重构出输出数据。最后，基础卷积层的输出与小波卷积的输出相加，得到最终结果。  
  
此外，文件中还定义了一个 `\_ScaleModule` 类，用于对输入进行缩放操作，确保输出的数值范围适合后续处理。  
  
总体来说，这个程序文件实现了一个结合小波变换的卷积神经网络层，能够在卷积操作中引入频域信息，从而可能提高模型对特征的提取能力。

```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  
from timm.layers import weight\_init  
  
# 定义激活函数类，继承自ReLU  
class activation(nn.ReLU):  
 def \_\_init\_\_(self, dim, act\_num=3, deploy=False):  
 super(activation, self).\_\_init\_\_()  
 self.deploy = deploy # 是否处于部署模式  
 # 初始化权重和偏置  
 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:  
 # 在部署模式下，直接进行卷积  
 return torch.nn.functional.conv2d(  
 super(activation, self).forward(x),   
 self.weight, self.bias, padding=(self.act\_num\*2 + 1)//2, groups=self.dim)  
 else:  
 # 否则，先进行批归一化再卷积  
 return self.bn(torch.nn.functional.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  
 if not self.deploy:  
 kernel, bias = self.\_fuse\_bn\_tensor(self.weight, self.bn) # 融合权重和偏置  
 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  
  
class Block(nn.Module):  
 def \_\_init\_\_(self, dim, dim\_out, act\_num=3, stride=2, deploy=False):  
 super().\_\_init\_\_()  
 self.deploy = deploy  
 # 根据是否部署选择不同的卷积结构  
 if self.deploy:  
 self.conv = nn.Conv2d(dim, dim\_out, kernel\_size=1)  
 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)  
 )  
 # 池化层的选择  
 self.pool = nn.MaxPool2d(stride) if stride != 1 else nn.Identity()  
 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  
  
class VanillaNet(nn.Module):  
 def \_\_init\_\_(self, in\_chans=3, num\_classes=1000, dims=[96, 192, 384, 768],   
 drop\_rate=0, act\_num=3, strides=[2,2,2,1], deploy=False):  
 super().\_\_init\_\_()  
 self.deploy = deploy  
 # 定义网络的stem部分  
 if self.deploy:  
 self.stem = nn.Sequential(  
 nn.Conv2d(in\_chans, dims[0], kernel\_size=4, stride=4),  
 activation(dims[0], act\_num)  
 )  
 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], act\_num)  
 )  
  
 self.stages = nn.ModuleList() # 存储网络的各个阶段  
 for i in range(len(strides)):  
 stage = Block(dim=dims[i], dim\_out=dims[i+1], act\_num=act\_num, stride=strides[i], deploy=deploy)  
 self.stages.append(stage) # 添加Block到网络中  
  
 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) # 通过每个Block  
 return x  
  
# 创建模型的函数  
def vanillanet\_10(pretrained='', \*\*kwargs):  
 model = VanillaNet(dims=[128\*4, 128\*4, 256\*4, 512\*4, 512\*4, 512\*4, 512\*4, 1024\*4], strides=[1,2,2,1,1,1,2,1], \*\*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)) # 随机输入  
 model = vanillanet\_10() # 创建模型  
 pred = model(inputs) # 进行预测  
 for i in pred:  
 print(i.size()) # 输出每一层的输出尺寸  
```  
  
### 代码说明：  
1. \*\*激活函数类 (`activation`)\*\*：定义了一个自定义的激活函数类，支持批归一化和卷积操作，能够在部署模式下优化性能。  
2. \*\*Block 类\*\*：定义了网络的基本构建块，包含卷积、批归一化、激活函数和池化操作。  
3. \*\*VanillaNet 类\*\*：构建整个网络结构，包含输入层、多个Block和输出层。  
4. \*\*模型创建函数\*\*：提供了创建不同配置的VanillaNet模型的功能，并支持加载预训练权重。  
5. \*\*主程序\*\*：生成随机输入并通过模型进行前向传播，输出每一层的输出尺寸。```

该程序文件`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。程序首先引入了必要的库，包括PyTorch和一些用于模型初始化和操作的工具。  
  
文件开头包含版权信息和许可声明，表明该程序是自由软件，可以在MIT许可证下进行修改和再分发。接下来，定义了一些全局变量，表示不同版本的VanillaNet模型。  
  
在程序中，定义了一个名为`activation`的类，继承自`nn.ReLU`，用于实现带有可学习参数的激活函数。该类的构造函数中初始化了权重和偏置，并定义了前向传播方法。在前向传播中，如果处于部署模式，则使用卷积操作，否则使用批归一化。该类还提供了一个方法用于将批归一化与卷积层融合，以便在推理时提高效率。  
  
接着，定义了一个`Block`类，表示网络中的基本构建块。每个Block包含两个卷积层和一个激活函数。构造函数中根据是否处于部署模式选择不同的卷积层结构，并根据步幅选择池化层。前向传播方法中依次执行卷积、激活和池化操作。Block类也提供了与`activation`类类似的融合方法和切换到部署模式的方法。  
  
`VanillaNet`类是整个模型的核心，包含多个Block和一个初始卷积层。构造函数中根据输入通道数、类别数、维度、丢弃率、激活函数数量、步幅等参数初始化网络结构。该类还定义了权重初始化方法和前向传播方法。在前向传播中，模型会根据输入大小生成特征图，并在不同的尺度上提取特征。  
  
此外，程序中还定义了一些辅助函数，例如`update\_weight`用于更新模型权重，确保模型的权重与加载的权重形状一致。接下来定义了一系列函数（如`vanillanet\_5`到`vanillanet\_13\_x1\_5\_ada\_pool`），用于创建不同配置的VanillaNet模型，并支持加载预训练权重。  
  
最后，在文件的主程序部分，创建了一个随机输入并实例化了一个VanillaNet模型，进行前向传播并打印输出特征图的尺寸。这部分代码可以用于测试模型的基本功能。  
  
总体来说，该程序实现了一个灵活的深度学习模型结构，允许用户根据需求调整模型的层数、维度和其他参数，并支持在推理时的优化。

### 程序整体功能和构架概括  
  
该项目包含多个文件，每个文件实现了特定的功能模块，主要用于构建和优化深度学习模型。整体上，这些模块结合了不同的卷积操作、注意力机制和网络结构，旨在提高模型在图像处理任务中的性能和效率。  
  
1. \*\*shiftwise\_conv.py\*\*：实现了一种自定义的卷积层，结合了小卷积核和大卷积核的特性，通过权重掩码和重参数化技术，增强了特征提取能力。  
  
2. \*\*cfpt.py\*\*：实现了跨层注意力机制，包括空间和通道注意力，通过不同层之间的信息交互，增强了特征图的表达能力。  
  
3. \*\*wtconv2d.py\*\*：实现了基于小波变换的二维卷积层，结合频域信息来处理输入数据，提供了一种新的卷积操作方式。  
  
4. \*\*VanillaNet.py\*\*：构建了一个灵活的深度学习模型VanillaNet，包含多个卷积块和可学习的激活函数，支持不同配置的模型创建和预训练权重加载。  
  
### 文件功能整理表  
  
| 文件名 | 功能描述 |  
|---------------------|--------------------------------------------------------------------------|  
| `shiftwise\_conv.py` | 实现自定义卷积层，结合小卷积核和大卷积核特性，通过权重掩码和重参数化技术增强特征提取能力。 |  
| `cfpt.py` | 实现跨层注意力机制，包括空间和通道注意力，增强特征图的表达能力。 |  
| `wtconv2d.py` | 实现基于小波变换的二维卷积层，结合频域信息处理输入数据，提供新的卷积操作方式。 |  
| `VanillaNet.py` | 构建灵活的深度学习模型VanillaNet，包含多个卷积块和可学习的激活函数，支持不同配置和预训练权重加载。 |   
  
这个表格总结了每个文件的主要功能，帮助理解整个项目的结构和各个模块之间的关系。