# 改进yolo11-aux等200+全套创新点大全：小鼠行为识别系统源码＆数据集全套

## 3.背景

研究背景与意义  
  
小鼠作为一种重要的实验动物，广泛应用于生物医学研究、药物开发和行为科学等领域。通过对小鼠行为的深入分析，研究人员能够获得关于神经生物学、心理学和药理学等方面的重要信息。传统的行为观察方法往往依赖于人工记录，不仅耗时耗力，而且容易受到观察者主观因素的影响。因此，开发一种高效、准确的小鼠行为识别系统显得尤为重要。  
  
近年来，计算机视觉技术的快速发展为动物行为识别提供了新的解决方案。YOLO（You Only Look Once）系列模型因其实时性和高精度的特点，成为目标检测领域的热门选择。YOLOv11作为该系列的最新版本，进一步提升了模型的性能，能够在复杂环境中实现更为精确的行为识别。通过对小鼠行为的自动化识别，不仅可以提高实验数据的可靠性，还能为行为学研究提供更为丰富的定量分析。  
  
本研究旨在基于改进的YOLOv11模型，构建一个小鼠行为识别系统。该系统将利用一个包含2200张图像的数据集，涵盖了小鼠的十种典型行为，包括前爪舔舐、前爪抬起、梳理、后爪舔舐、后爪抬起、原地、跳跃、直立、伸展和转身等。这些行为的多样性为模型的训练提供了丰富的样本，有助于提高识别的准确性和鲁棒性。  
  
通过本项目的实施，期望能够为小鼠行为研究提供一种高效、自动化的工具，推动相关领域的研究进展。同时，基于YOLOv11的改进模型也将为其他动物行为识别的研究提供借鉴，促进计算机视觉技术在生物医学领域的应用与发展。

## 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  
  
本项目旨在通过改进YOLOv11模型，提升小鼠行为识别系统的准确性与效率。为实现这一目标，我们采用了“Hotplate\_behavior\_annotations”数据集，该数据集专注于小鼠在热板实验中的多种行为表现，涵盖了丰富的行为类别，适合用于训练和验证深度学习模型。数据集中包含10个主要的行为类别，具体包括：前爪舔舐、前爪抬起、梳理、后爪舔舐、后爪抬起、原地活动、跳跃、竖立、伸展以及转身。这些行为类别不仅具有显著的生物学意义，还能为研究小鼠的疼痛反应、活动水平及其他生理行为提供重要的参考。  
  
在数据集的构建过程中，所有行为均经过精确标注，确保每个样本的行为特征清晰可辨。这一过程涉及到对小鼠在热板上行为的细致观察与记录，确保数据的真实性和可靠性。通过对这些行为的深入分析，我们能够更好地理解小鼠在不同环境刺激下的反应机制，为相关生物医学研究提供坚实的数据基础。  
  
此外，数据集的多样性和丰富性为模型的训练提供了广泛的样本支持，使得YOLOv11在识别小鼠行为时能够具备更强的泛化能力。通过使用这一数据集，我们希望能够在小鼠行为识别领域取得突破性进展，推动相关研究的深入发展。整体而言，“Hotplate\_behavior\_annotations”数据集不仅为本项目提供了必要的训练数据，也为未来在小鼠行为学及其相关领域的研究奠定了坚实的基础。

## 5. 全套项目环境部署教程（零基础手把手教学）

5.1 所需软件PyCharm和Anaconda安装教程（第一步）

https://www.bilibili.com/video/BV1BoC1YCEKi/?spm\_id\_from=333.999.0.0&vd\_source=bc9aec86d164b67a7004b996143742dc

5.2 安装Python虚拟环境创建和依赖库安装视频教程（第二步）

https://www.bilibili.com/video/BV1ZoC1YCEBw?spm\_id\_from=333.788.videopod.sections&vd\_source=bc9aec86d164b67a7004b996143742dc

## 6. 改进YOLOv11训练教程和Web\_UI前端加载模型视频教程（零基础手把手教学）

https://www.bilibili.com/video/BV1BoC1YCEhR?spm\_id\_from=333.788.videopod.sections&vd\_source=bc9aec86d164b67a7004b996143742dc

## 7.原始YOLOv11算法讲解

## 8. 70+全套YOLOV8创新点原理讲解（非科班也可以轻松写刊发刊，V12版本正在科研待更新）

由于篇幅限制，每个创新点的具体原理讲解就不全部展开，具体见下列网址中的改进模块对应项目的技术原理博客网址【Blog】（创新点均为模块化搭建，原理适配YOLOv5~YOLOv11等各种版本）

https://gitee.com/qunmasj/good

## 9.系统功能展示（检测对象为举例，实际内容以本项目数据集为准）

图9.1.系统支持检测结果表格显示  
  
 图9.2.系统支持置信度和IOU阈值手动调节  
  
 图9.3.系统支持自定义加载权重文件best.pt(需要你通过步骤5中训练获得)  
  
 图9.4.系统支持摄像头实时识别  
  
 图9.5.系统支持图片识别  
  
 图9.6.系统支持视频识别  
  
 图9.7.系统支持识别结果文件自动保存  
  
 图9.8.系统支持Excel导出检测结果数据

## 10. YOLOv11核心改进模块源码讲解

```以下是对给定代码的核心部分进行提炼和详细注释的结果。主要保留了模型的定义和前向传播的逻辑，去掉了一些不必要的部分，如加载预训练模型的代码和主函数的测试部分。  
  
```python  
import torch  
import torch.nn as nn  
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):  
 """局部增强注意力（LePE）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  
 head\_dim = dim // num\_heads # 每个头的维度  
 self.scale = head\_dim \*\* -0.5 # 缩放因子  
 self.get\_v = nn.Conv2d(dim, dim, kernel\_size=3, stride=1, padding=1, groups=dim) # 用于获取v的卷积层  
 self.attn\_drop = nn.Dropout(attn\_drop) # 注意力Dropout  
  
 def im2cswin(self, x):  
 """将输入转换为窗口形式"""  
 B, N, C = x.shape  
 H = W = int(np.sqrt(N)) # 计算高度和宽度  
 x = x.transpose(-2, -1).contiguous().view(B, C, H, W) # 转换形状  
 x = img2windows(x, self.split\_size, self.split\_size) # 切分窗口  
 return x  
  
 def forward(self, qkv):  
 """前向传播"""  
 q, k, v = qkv # 获取q, k, v  
 q = self.im2cswin(q) # 将q转换为窗口形式  
 k = self.im2cswin(k) # 将k转换为窗口形式  
 v = self.get\_v(v) # 通过卷积获取v  
 q = q \* self.scale # 缩放q  
 attn = (q @ k.transpose(-2, -1)) # 计算注意力  
 attn = nn.functional.softmax(attn, dim=-1) # softmax归一化  
 attn = self.attn\_drop(attn) # 应用Dropout  
 x = attn @ v # 计算加权和  
 return x  
  
class CSWinBlock(nn.Module):  
 """CSWin Transformer的基本块"""  
 def \_\_init\_\_(self, dim, reso, num\_heads, split\_size=7, 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.norm1 = nn.LayerNorm(dim) # 归一化层  
 self.attn = LePEAttention(dim, resolution=reso, idx=0, split\_size=split\_size, num\_heads=num\_heads) # 注意力层  
 self.mlp = Mlp(in\_features=dim, hidden\_features=int(dim \* mlp\_ratio), out\_features=dim) # MLP层  
 self.norm2 = nn.LayerNorm(dim) # 归一化层  
  
 def forward(self, x):  
 """前向传播"""  
 x = self.norm1(x) # 归一化  
 qkv = self.qkv(x).reshape(x.shape[0], -1, 3, self.dim).permute(2, 0, 1, 3) # 生成q, k, v  
 x = self.attn(qkv) # 计算注意力  
 x = x + self.norm2(x) # 加上归一化后的输入  
 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, in\_chans=3, num\_classes=1000, embed\_dim=96, depth=[2, 2, 6, 2], num\_heads=12):  
 super().\_\_init\_\_()  
 self.stage1\_conv\_embed = nn.Sequential(  
 nn.Conv2d(in\_chans, embed\_dim, 7, 4, 2), # 初始卷积层  
 nn.LayerNorm(embed\_dim) # 归一化层  
 )  
 self.stage1 = nn.ModuleList([  
 CSWinBlock(dim=embed\_dim, num\_heads=num\_heads) for \_ in range(depth[0]) # 第一阶段的CSWin块  
 ])  
 # 其他阶段的定义省略...  
  
 def forward(self, x):  
 """前向传播"""  
 x = self.stage1\_conv\_embed(x) # 通过初始卷积层  
 for blk in self.stage1:  
 x = blk(x) # 通过每个CSWin块  
 return x  
  
# img2windows 和 windows2img 函数省略...  
```  
  
### 主要部分说明：  
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。  
2. \*\*LePEAttention类\*\*：实现了局部增强注意力机制，负责将输入转换为窗口形式并计算注意力。  
3. \*\*CSWinBlock类\*\*：定义了CSWin Transformer的基本块，包含注意力层和MLP层。  
4. \*\*CSWinTransformer类\*\*：整体模型的定义，包含多个CSWin块和初始卷积层。  
  
以上代码保留了模型的核心结构和前向传播逻辑，去掉了与模型加载和测试相关的部分。```

该文件实现了一个名为CSWin Transformer的视觉变换器模型，主要用于图像分类等计算机视觉任务。代码的结构分为多个类和函数，每个部分都有其特定的功能。  
  
首先，文件导入了必要的库，包括PyTorch及其相关模块、NumPy和一些用于图像处理的工具。文件的开头部分定义了一些常量和全局变量，接着定义了多个类，分别实现了模型的不同组件。  
  
Mlp类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），用于特征的非线性变换。LePEAttention类实现了局部增强注意力机制，包含多个头的自注意力机制。它的构造函数接收多个参数，包括输入维度、分辨率、分割大小、头数等。该类的forward方法实现了输入的查询、键、值的计算，并通过注意力机制对输入进行加权。  
  
CSWinBlock类是CSWin Transformer的基本构建块，结合了MLP和注意力机制。它的构造函数定义了多个参数，包括输入维度、分辨率、头数等。forward方法实现了块内的前向传播，首先对输入进行归一化，然后计算查询、键、值，并通过注意力机制和MLP进行特征提取。  
  
接下来，img2windows和windows2img函数用于将图像从空间域转换为窗口域和反向转换，支持对图像进行分块处理。Merge\_Block类用于合并特征图，使用卷积层和归一化层来处理特征。  
  
CSWinTransformer类是整个模型的核心，包含多个阶段的构建，每个阶段由多个CSWinBlock组成。构造函数定义了输入图像的大小、补丁大小、通道数、类别数等参数，并逐步构建模型的各个阶段。forward\_features方法负责特征提取，逐层通过各个块并合并特征。  
  
最后，文件定义了一些辅助函数，例如\_conv\_filter用于转换权重，update\_weight用于更新模型权重，以及CSWin\_tiny、CSWin\_small、CSWin\_base和CSWin\_large函数用于创建不同规模的CSWin Transformer模型，并可选择加载预训练权重。  
  
在文件的最后部分，包含了一个测试代码块，创建了不同规模的模型并对随机输入进行前向传播，输出每个模型的特征图大小。这段代码用于验证模型的正确性和功能。  
  
整体而言，该文件实现了一个高效的视觉变换器模型，利用局部增强注意力机制和多层感知机的组合，适用于各种计算机视觉任务。

```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  
import torch.nn.functional as F  
  
def autopad(k, p=None, d=1):  
 """自动填充以保持输出形状相同。"""  
 if d > 1:  
 k = d \* (k - 1) + 1 if isinstance(k, int) else [d \* (x - 1) + 1 for x in k] # 实际的卷积核大小  
 if p is None:  
 p = k // 2 if isinstance(k, int) else [x // 2 for x in k] # 自动填充  
 return p  
  
class Conv(nn.Module):  
 """卷积层，包含卷积、批归一化和激活函数。"""  
 def \_\_init\_\_(self, in\_channels, out\_channels, kernel\_size=3, stride=1, padding=None, groups=1, act=True):  
 super().\_\_init\_\_()  
 self.conv = nn.Conv2d(in\_channels, out\_channels, kernel\_size, stride, autopad(kernel\_size, padding), groups=groups, bias=False)  
 self.bn = nn.BatchNorm2d(out\_channels)  
 self.act = nn.SiLU() if act else nn.Identity() # 使用SiLU激活函数  
  
 def forward(self, x):  
 """前向传播。"""  
 return self.act(self.bn(self.conv(x)))  
  
class Bottleneck(nn.Module):  
 """标准瓶颈结构。"""  
 def \_\_init\_\_(self, c1, c2, shortcut=True, g=1, k=(3, 3), e=0.5):  
 super().\_\_init\_\_()  
 c\_ = int(c2 \* e) # 隐藏通道数  
 self.cv1 = Conv(c1, c\_, k[0], 1) # 第一个卷积  
 self.cv2 = Conv(c\_, c2, k[1], 1, g=g) # 第二个卷积  
 self.add = shortcut and c1 == c2 # 是否使用shortcut连接  
  
 def forward(self, x):  
 """前向传播。"""  
 return x + self.cv2(self.cv1(x)) if self.add else self.cv2(self.cv1(x))  
  
class C3k(nn.Module):  
 """C3k模块，包含多个瓶颈结构。"""  
 def \_\_init\_\_(self, c1, c2, n=1, shortcut=False, g=1, e=0.5, k=3):  
 super().\_\_init\_\_()  
 self.m = nn.Sequential(\*(Bottleneck(c1, c2, shortcut, g, k=(k, k), e=e) for \_ in range(n))) # 创建n个瓶颈层  
  
 def forward(self, x):  
 """前向传播。"""  
 return self.m(x)  
  
class C3k2(nn.Module):  
 """C3k2模块，包含多个C3k模块。"""  
 def \_\_init\_\_(self, c1, c2, n=1, c3k=False, e=0.5, g=1, shortcut=True):  
 super().\_\_init\_\_()  
 self.m = nn.ModuleList(C3k(c1, c2, n, shortcut, g, e) for \_ in range(n)) # 创建n个C3k层  
  
 def forward(self, x):  
 """前向传播。"""  
 for layer in self.m:  
 x = layer(x)  
 return x  
  
class DynamicConv(nn.Module):  
 """动态卷积层。"""  
 def \_\_init\_\_(self, c1, c2, k=1, s=1, p=None, g=1, d=1, act=True, num\_experts=4):  
 super().\_\_init\_\_()  
 self.conv = nn.Sequential(  
 DynamicConv\_Single(c1, c2, kernel\_size=k, stride=s, padding=autopad(k, p, d), dilation=d, groups=g, num\_experts=num\_experts),  
 nn.BatchNorm2d(c2),  
 nn.SiLU() if act else nn.Identity()  
 )  
  
 def forward(self, x):  
 """前向传播。"""  
 return self.conv(x)  
  
class DynamicConv\_Single(nn.Module):  
 """动态卷积单元。"""  
 def \_\_init\_\_(self, in\_features, out\_features, kernel\_size=1, stride=1, padding='', dilation=1,  
 groups=1, bias=False, num\_experts=4):  
 super().\_\_init\_\_()  
 self.routing = nn.Linear(in\_features, num\_experts) # 路由层  
 self.cond\_conv = CondConv2d(in\_features, out\_features, kernel\_size, stride, padding, dilation,  
 groups, bias, num\_experts) # 条件卷积层  
  
 def forward(self, x):  
 """前向传播。"""  
 pooled\_inputs = F.adaptive\_avg\_pool2d(x, 1).flatten(1) # 计算条件卷积的路由  
 routing\_weights = torch.sigmoid(self.routing(pooled\_inputs)) # 计算路由权重  
 x = self.cond\_conv(x, routing\_weights) # 应用条件卷积  
 return x  
  
class C3k\_DynamicConv(C3k):  
 """C3k模块，使用动态卷积。"""  
 def \_\_init\_\_(self, c1, c2, n=1, shortcut=False, g=1, e=0.5, k=3):  
 super().\_\_init\_\_(c1, c2, n, shortcut, g, e, k)  
 c\_ = int(c2 \* e) # 隐藏通道数  
 self.m = nn.Sequential(\*(Bottleneck\_DynamicConv(c\_, c\_, shortcut, g, k=(k, k), e=1.0) for \_ in range(n))) # 创建n个动态卷积瓶颈层  
  
 def forward(self, x):  
 """前向传播。"""  
 return self.m(x)  
  
# 其他模块省略，保留上述核心部分及注释  
```  
  
上述代码包含了动态卷积、瓶颈结构和C3k模块的实现，注释详细解释了每个类和方法的功能。```

这个程序文件 `block.py` 是一个用于构建深度学习模型的模块，特别是用于计算机视觉任务的模型。文件中包含了多个类和函数，这些类和函数实现了不同的卷积层、注意力机制、块结构等，主要用于构建神经网络的各个部分。以下是对文件内容的详细说明：  
  
首先，文件导入了许多必要的库，包括 PyTorch 及其子模块（如 `torch.nn` 和 `torch.nn.functional`），以及一些其他工具库（如 `numpy` 和 `math`）。这些库为后续的深度学习模型构建提供了基础。  
  
文件的开头定义了一些通用的函数，例如 `autopad`，用于自动计算卷积操作的填充，以确保输出尺寸与输入尺寸相同。  
  
接下来，文件中定义了多个类，这些类可以分为几大类：  
  
1. \*\*卷积层和注意力机制\*\*：文件中实现了多种卷积层（如 `Conv`、`DWConv`、`DSConv`、`RepConv`、`GhostConv` 等），这些卷积层可以用于不同的网络结构。此外，还实现了多种注意力机制（如 `Attention`、`ChannelAttention`、`SpatialAttention` 等），用于增强模型对特征的关注能力。  
  
2. \*\*块结构\*\*：文件中定义了多个网络块（如 `Bottleneck`、`C3k`、`C3k2` 等），这些块通常由多个卷积层和激活函数组成，用于构建更复杂的网络结构。每个块可以包含不同的卷积层、注意力机制和其他操作，以实现特定的功能。  
  
3. \*\*特定功能模块\*\*：文件中还实现了一些特定功能的模块，例如 `PyramidPoolAgg`、`FocusFeature`、`SDFM` 等，这些模块用于特定的任务，如特征融合、上下文信息提取等。  
  
4. \*\*动态卷积和自适应卷积\*\*：文件中实现了动态卷积（如 `DynamicConv` 和 `DynamicConv\_Single`），这些卷积层可以根据输入的特征动态调整其参数，以提高模型的灵活性和适应性。  
  
5. \*\*融合模块\*\*：实现了一些融合模块（如 `DynamicInterpolationFusion` 和 `FeaturePyramidSharedConv`），这些模块用于将来自不同层的特征进行融合，以增强模型的表达能力。  
  
6. \*\*自适应模块\*\*：如 `AdaptiveDilatedConv` 和 `SEAttention`，这些模块用于在卷积操作中引入自适应机制，以便更好地捕捉特征。  
  
7. \*\*特定网络结构\*\*：文件中还实现了一些特定的网络结构，如 `CSPStage`、`C3k2` 等，这些结构通常用于实现更复杂的网络模型，如 YOLO 系列模型。  
  
最后，文件中还包含了一些辅助函数和类，用于初始化权重、处理输入数据等。这些函数和类为构建和训练深度学习模型提供了必要的支持。  
  
总体而言，`block.py` 文件是一个功能丰富的模块，提供了多种构建深度学习模型所需的组件，适用于计算机视觉任务的实现。

```以下是经过简化和注释的代码，保留了核心部分：  
  
```python  
import torch  
import torch.nn as nn  
import torch.nn.functional as F  
import numpy as np  
from timm.models.layers import DropPath, to\_2tuple, trunc\_normal\_  
  
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):  
 """ 窗口注意力机制模块。 """  
  
 def \_\_init\_\_(self, dim, window\_size, num\_heads, qkv\_bias=True, attn\_drop=0., proj\_drop=0.):  
 super().\_\_init\_\_()  
 self.dim = dim  
 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  
  
 # 定义相对位置偏置参数  
 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  
 relative\_coords[:, :, 0] += self.window\_size[0] - 1 # 偏移  
 relative\_coords[:, :, 1] += self.window\_size[1] - 1  
 relative\_coords[:, :, 0] \*= 2 \* self.window\_size[1] - 1  
 self.relative\_position\_index = relative\_coords.sum(-1) # 计算相对位置索引  
  
 self.qkv = nn.Linear(dim, dim \* 3, bias=qkv\_bias) # 线性变换生成Q, K, V  
 self.attn\_drop = nn.Dropout(attn\_drop) # 注意力Dropout  
 self.proj = nn.Linear(dim, dim) # 输出线性变换  
 self.proj\_drop = nn.Dropout(proj\_drop) # 输出Dropout  
 trunc\_normal\_(self.relative\_position\_bias\_table, std=.02) # 初始化相对位置偏置  
 self.softmax = nn.Softmax(dim=-1) # Softmax层  
  
 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)  
 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()  
 attn = attn + relative\_position\_bias.unsqueeze(0)  
  
 attn = self.softmax(attn) # 应用Softmax  
 attn = self.attn\_drop(attn) # 注意力Dropout  
  
 x = (attn @ v).transpose(1, 2).reshape(B\_, N, C) # 计算输出  
 x = self.proj(x) # 输出线性变换  
 x = self.proj\_drop(x) # 输出Dropout  
 return x  
  
  
class SwinTransformer(nn.Module):  
 """ Swin Transformer 主体。 """  
  
 def \_\_init\_\_(self, patch\_size=4, embed\_dim=96, depths=[2, 2, 6, 2], num\_heads=[3, 6, 12, 24]):  
 super().\_\_init\_\_()  
 self.patch\_embed = PatchEmbed(patch\_size=patch\_size, in\_chans=3, embed\_dim=embed\_dim) # 图像分块  
 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=7,  
 mlp\_ratio=4.)  
 self.layers.append(layer)  
  
 def forward(self, x):  
 """ 前向传播函数。 """  
 x = self.patch\_embed(x) # 图像分块  
 outs = []  
 for layer in self.layers:  
 x = layer(x) # 通过每一层  
 outs.append(x) # 收集输出  
 return outs  
  
  
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])  
 if weights:  
 model.load\_state\_dict(torch.load(weights)['model']) # 加载权重  
 return model  
```  
  
### 代码注释说明：  
1. \*\*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，支持Dropout。  
2. \*\*WindowAttention类\*\*：实现了窗口注意力机制，计算Q、K、V，并添加相对位置偏置。  
3. \*\*SwinTransformer类\*\*：构建了Swin Transformer的主要结构，包括图像分块和多个层的堆叠。  
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。  
  
此代码是Swin Transformer的核心实现，包含了模型的基本结构和注意力机制。```

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是一种分层的视觉Transformer，采用了窗口注意力机制，通过在不同层次上处理图像块来提高效率和效果。  
  
首先，文件导入了必要的库，包括PyTorch和一些自定义的模块。接着定义了多个类和函数。  
  
`Mlp`类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），并在每个线性层后面加上了Dropout层，以防止过拟合。  
  
`window\_partition`和`window\_reverse`函数用于将输入特征图分割成窗口，以及将窗口合并回特征图。窗口的大小由参数`window\_size`指定。  
  
`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。该类的构造函数定义了输入通道数、窗口大小、注意力头数等参数，并初始化了相对位置偏置表。`forward`方法计算输入特征的注意力权重，并应用于输入特征。  
  
`SwinTransformerBlock`类实现了Swin Transformer的基本块，包含归一化层、窗口注意力层和MLP。该类支持循环移位，以实现Shifted Window Attention（SW-MSA）。`forward`方法处理输入特征，进行归一化、窗口分割、注意力计算和特征合并。  
  
`PatchMerging`类用于将特征图的块合并，以减少特征图的空间维度。它通过线性层将4个输入通道合并为2个输出通道，并在合并前进行归一化。  
  
`BasicLayer`类表示Swin Transformer的一个基本层，包含多个Swin Transformer块，并在必要时进行下采样。它还计算SW-MSA的注意力掩码。  
  
`PatchEmbed`类将输入图像分割成不重叠的块，并通过卷积层进行嵌入。它还支持对嵌入后的特征进行归一化。  
  
`SwinTransformer`类是整个模型的主类，负责构建模型的各个层次，包括图像块嵌入、绝对位置嵌入、多个基本层和输出层。它的`forward`方法处理输入图像，生成多个输出特征图。  
  
最后，`update\_weight`函数用于更新模型的权重，从给定的权重字典中加载匹配的权重。`SwinTransformer\_Tiny`函数创建一个Swin Transformer模型实例，并可选择加载预训练权重。  
  
整体来看，这个文件实现了Swin Transformer的核心组件和结构，适用于图像分类、目标检测等视觉任务。

```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 as nn  
from einops import rearrange  
  
class RFAConv(nn.Module):  
 def \_\_init\_\_(self, in\_channel, out\_channel, kernel\_size, stride=1):  
 super().\_\_init\_\_()  
 self.kernel\_size = kernel\_size  
  
 # 生成权重的卷积模块  
 self.get\_weight = nn.Sequential(  
 nn.AvgPool2d(kernel\_size=kernel\_size, padding=kernel\_size // 2, stride=stride),  
 nn.Conv2d(in\_channel, in\_channel \* (kernel\_size \*\* 2), kernel\_size=1, groups=in\_channel, bias=False)  
 )  
   
 # 生成特征的卷积模块  
 self.generate\_feature = nn.Sequential(  
 nn.Conv2d(in\_channel, in\_channel \* (kernel\_size \*\* 2), kernel\_size=kernel\_size, padding=kernel\_size // 2, stride=stride, groups=in\_channel, bias=False),  
 nn.BatchNorm2d(in\_channel \* (kernel\_size \*\* 2)),  
 nn.ReLU()  
 )  
   
 # 最终的卷积层  
 self.conv = nn.Conv2d(in\_channel, out\_channel, kernel\_size=kernel\_size, stride=kernel\_size)  
  
 def forward(self, x):  
 b, c = x.shape[0:2] # 获取输入的批次大小和通道数  
 weight = self.get\_weight(x) # 计算权重  
 h, w = weight.shape[2:] # 获取特征图的高和宽  
   
 # 对权重进行softmax归一化  
 weighted = weight.view(b, c, self.kernel\_size \*\* 2, h, w).softmax(2) # b c\*kernel\*\*2, h, w  
   
 # 生成特征并调整形状  
 feature = self.generate\_feature(x).view(b, c, self.kernel\_size \*\* 2, h, w) # b c\*kernel\*\*2, h, w  
   
 # 加权特征  
 weighted\_data = feature \* weighted  
   
 # 重排特征数据以适应卷积层的输入  
 conv\_data = rearrange(weighted\_data, 'b c (n1 n2) h w -> b c (h n1) (w n2)', n1=self.kernel\_size, n2=self.kernel\_size)  
   
 return self.conv(conv\_data) # 返回卷积结果  
  
  
class SE(nn.Module):  
 def \_\_init\_\_(self, in\_channel, ratio=16):  
 super(SE, self).\_\_init\_\_()  
 self.gap = nn.AdaptiveAvgPool2d((1, 1)) # 全局平均池化  
 self.fc = nn.Sequential(  
 nn.Linear(in\_channel, ratio, bias=False), # 从 c -> c/r  
 nn.ReLU(),  
 nn.Linear(ratio, in\_channel, bias=False), # 从 c/r -> c  
 nn.Sigmoid()  
 )  
  
 def forward(self, x):  
 b, c = x.shape[0:2] # 获取输入的批次大小和通道数  
 y = self.gap(x).view(b, c) # 进行全局平均池化并调整形状  
 y = self.fc(y).view(b, c, 1, 1) # 通过全连接层  
 return y # 返回通道注意力  
  
  
class RFCBAMConv(nn.Module):  
 def \_\_init\_\_(self, in\_channel, out\_channel, kernel\_size=3, stride=1):  
 super().\_\_init\_\_()  
 self.kernel\_size = kernel\_size  
   
 # 生成特征的卷积模块  
 self.generate = nn.Sequential(  
 nn.Conv2d(in\_channel, in\_channel \* (kernel\_size \*\* 2), kernel\_size, padding=kernel\_size // 2, stride=stride, groups=in\_channel, bias=False),  
 nn.BatchNorm2d(in\_channel \* (kernel\_size \*\* 2)),  
 nn.ReLU()  
 )  
   
 # 计算权重的卷积模块  
 self.get\_weight = nn.Sequential(nn.Conv2d(2, 1, kernel\_size=3, padding=1, bias=False), nn.Sigmoid())  
 self.se = SE(in\_channel) # 通道注意力模块  
  
 # 最终的卷积层  
 self.conv = nn.Conv2d(in\_channel, out\_channel, kernel\_size=kernel\_size, stride=kernel\_size)  
  
 def forward(self, x):  
 b, c = x.shape[0:2] # 获取输入的批次大小和通道数  
 channel\_attention = self.se(x) # 计算通道注意力  
 generate\_feature = self.generate(x) # 生成特征  
  
 h, w = generate\_feature.shape[2:] # 获取特征图的高和宽  
 generate\_feature = generate\_feature.view(b, c, self.kernel\_size \*\* 2, h, w) # 调整形状  
   
 # 重排特征数据  
 generate\_feature = rearrange(generate\_feature, 'b c (n1 n2) h w -> b c (h n1) (w n2)', n1=self.kernel\_size, n2=self.kernel\_size)  
   
 # 加权特征  
 unfold\_feature = generate\_feature \* channel\_attention  
   
 # 计算最大值和均值特征  
 max\_feature, \_ = torch.max(generate\_feature, dim=1, keepdim=True)  
 mean\_feature = torch.mean(generate\_feature, dim=1, keepdim=True)  
   
 # 计算接收场注意力  
 receptive\_field\_attention = self.get\_weight(torch.cat((max\_feature, mean\_feature), dim=1))  
   
 # 返回卷积结果  
 return self.conv(unfold\_feature \* receptive\_field\_attention) # 返回卷积结果  
```  
  
### 代码核心部分说明：  
1. \*\*RFAConv\*\*：该类实现了一种卷积操作，结合了权重生成和特征生成。它通过对输入特征进行加权，生成新的特征图，并通过卷积层输出最终结果。  
   
2. \*\*SE（Squeeze-and-Excitation）\*\*：该类实现了通道注意力机制，通过全局平均池化和全连接层来计算每个通道的重要性，并生成相应的权重。  
  
3. \*\*RFCBAMConv\*\*：该类结合了特征生成、通道注意力和接收场注意力的机制，进一步增强了卷积操作的效果。它通过最大池化和均值池化来计算接收场注意力，并与生成的特征相结合，最终通过卷积层输出结果。```

这个程序文件定义了几个用于卷积神经网络的模块，主要包括RFAConv、RFCBAMConv和RFCAConv。文件中使用了PyTorch库，并且结合了一些自定义的激活函数和卷积操作。  
  
首先，文件引入了必要的库，包括torch和torch.nn，以及用于重排张量的einops库。接着，定义了两个自定义的激活函数：h\_sigmoid和h\_swish。h\_sigmoid是一个改进的sigmoid函数，使用ReLU6来限制输出范围；h\_swish则是将输入与h\_sigmoid的输出相乘，形成一个新的激活函数。  
  
RFAConv类是一个卷积模块，初始化时接收输入通道数、输出通道数、卷积核大小和步幅。它包含两个主要部分：一个用于生成权重的卷积层和一个用于生成特征的卷积层。权重通过对输入进行平均池化和卷积得到，特征则通过卷积、批归一化和ReLU激活得到。在前向传播中，首先计算权重，然后对特征进行处理，最后通过重排和卷积得到输出。  
  
RFCBAMConv类在RFAConv的基础上增加了通道注意力机制。它通过SE（Squeeze-and-Excitation）模块来生成通道注意力，并结合最大池化和平均池化来获取特征的全局信息。生成的特征经过重排后与通道注意力相乘，最后通过卷积得到输出。  
  
RFCAConv类则进一步扩展了功能，结合了空间注意力机制。它在生成特征后，分别对特征在高度和宽度方向进行自适应平均池化，生成两个特征图。然后将这两个特征图拼接，经过一系列卷积和激活操作后，得到空间注意力。最终，生成的特征与空间注意力相乘，并通过卷积得到最终输出。  
  
整个文件实现了基于注意力机制的卷积操作，旨在提升特征提取的能力，使得网络在处理图像时能够更加关注重要的区域和通道。这些模块可以被集成到更大的神经网络中，以提高性能。

### 程序整体功能和构架概括  
  
该程序的整体功能是实现多个先进的视觉变换器模型和卷积模块，主要用于计算机视觉任务，如图像分类、目标检测等。程序由多个文件组成，每个文件负责不同的功能模块，包括特定的网络结构、卷积操作、注意力机制等。这些模块可以组合在一起，形成复杂的深度学习模型，以提高在视觉任务中的表现。  
  
- \*\*CSwomTransformer.py\*\*：实现了CSWin Transformer模型，采用局部增强注意力机制，适用于图像分类等任务。  
- \*\*block.py\*\*：提供了多种卷积层和网络块的实现，支持构建更复杂的网络结构，包含注意力机制和特定功能模块。  
- \*\*SwinTransformer.py\*\*：实现了Swin Transformer模型，采用窗口注意力机制，分层处理图像块，提高了模型的效率和效果。  
- \*\*RFAConv.py\*\*：定义了基于注意力机制的卷积模块，增强了特征提取能力，适用于各种卷积神经网络。  
  
### 文件功能整理表  
  
| 文件名 | 功能描述 |  
|---------------------------|--------------------------------------------------------------------------|  
| CSwomTransformer.py | 实现CSWin Transformer模型，采用局部增强注意力机制，适用于图像分类等任务。 |  
| block.py | 提供多种卷积层和网络块的实现，支持构建复杂网络结构，包含注意力机制和特定功能模块。 |  
| SwinTransformer.py | 实现Swin Transformer模型，采用窗口注意力机制，分层处理图像块，提高模型效率和效果。 |  
| RFAConv.py | 定义基于注意力机制的卷积模块，增强特征提取能力，适用于各种卷积神经网络。 |  
  
通过这些模块的组合，程序能够构建出高效且强大的视觉变换器模型，适应多种计算机视觉任务的需求。