# 改进yolo11-RVB等200+全套创新点大全：麻将识别与定位系统源码＆数据集全套

## 3.背景

研究背景与意义  
  
随着人工智能技术的迅猛发展，计算机视觉在各个领域的应用日益广泛，尤其是在物体识别和定位方面。麻将作为一种深受欢迎的传统棋牌游戏，其复杂的牌面结构和多样的牌型使得麻将识别成为计算机视觉领域的一项挑战。传统的麻将识别方法往往依赖于手工特征提取和规则设定，难以适应不同的环境和牌局变化。因此，基于深度学习的自动化麻将识别与定位系统的研究显得尤为重要。  
  
本研究旨在基于改进的YOLOv11模型，构建一个高效的麻将识别与定位系统。YOLO（You Only Look Once）系列模型以其快速的检测速度和较高的准确率在实时物体检测中表现出色。通过对YOLOv11进行改进，我们期望能够提升其在麻将牌识别中的性能，尤其是在复杂背景和不同光照条件下的适应能力。  
  
本项目使用的数据集包含36个类别的麻将牌，包括各种花色和特殊牌型，共计414张经过标注的图像。这些图像经过了多种预处理和数据增强技术，确保了模型训练的多样性和鲁棒性。通过引入数据增强方法，如随机高斯模糊，进一步提升了模型的泛化能力，使其能够更好地适应现实场景中的变化。  
  
本研究不仅具有重要的学术价值，也为麻将游戏的智能化发展提供了技术支持。通过实现高效的麻将识别与定位系统，可以为在线麻将游戏、智能麻将桌等应用提供基础，推动传统文化与现代科技的融合，具有广泛的社会和经济意义。

## 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  
  
本项目的数据集专注于麻将识别与定位系统的训练，旨在改进YOLOv11模型的性能，以实现对麻将牌的高效识别与定位。数据集的主题围绕“4”展开，涵盖了麻将游戏中常见的多种牌型，具有丰富的多样性和复杂性。数据集共包含36个类别，具体类别包括：b1至b9的基本牌，baopai（宝牌）、baoting（报听）、chi（吃牌）、gang（杠牌）、guo（过牌）、peng（碰牌），以及多种特定的牌型标识（t1至t9和w1至w9），最后还有x2、x3和z等类别。这些类别不仅涵盖了麻将的基本构成，还涉及了游戏中的策略和玩法，为模型的训练提供了全面的基础。  
  
数据集分为训练集、验证集和测试集，分别存放在指定的路径下，确保模型在不同阶段的学习和评估。训练集包含丰富的样本，旨在帮助模型学习麻将牌的特征和识别规律；验证集则用于调整模型参数，以提高其在未见数据上的表现；测试集则是对模型最终性能的检验，确保其在实际应用中的有效性和准确性。  
  
通过使用这一数据集，研究人员可以深入探索麻将牌的视觉特征，并利用YOLOv11的先进技术进行实时识别与定位。这一系统的成功实现不仅能够提升麻将游戏的智能化水平，还能为相关领域的研究提供重要的数据支持和实践基础。整体而言，本项目的数据集为麻将识别与定位系统的研究提供了坚实的基础，具有广泛的应用前景和研究价值。

## 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：  
  
```python  
import torch  
import torch.nn as nn  
from timm.models.layers import SqueezeExcite  
  
def \_make\_divisible(v, divisor, min\_value=None):  
 """  
 确保所有层的通道数是8的倍数。  
 :param v: 输入的通道数  
 :param divisor: 需要被整除的数  
 :param min\_value: 最小值  
 :return: 处理后的通道数  
 """  
 if min\_value is None:  
 min\_value = divisor  
 new\_v = max(min\_value, int(v + divisor / 2) // divisor \* divisor)  
 # 确保向下取整不会超过原值的90%  
 if new\_v < 0.9 \* v:  
 new\_v += divisor  
 return new\_v  
  
class Conv2d\_BN(nn.Sequential):  
 """  
 包含卷积层和批归一化层的模块。  
 """  
 def \_\_init\_\_(self, in\_channels, out\_channels, kernel\_size=1, stride=1, padding=0, dilation=1, groups=1, bn\_weight\_init=1):  
 super().\_\_init\_\_()  
 # 添加卷积层  
 self.add\_module('c', nn.Conv2d(in\_channels, out\_channels, kernel\_size, stride, padding, dilation, groups, bias=False))  
 # 添加批归一化层  
 self.add\_module('bn', nn.BatchNorm2d(out\_channels))  
 # 初始化批归一化层的权重  
 nn.init.constant\_(self.bn.weight, bn\_weight\_init)  
 nn.init.constant\_(self.bn.bias, 0)  
  
 @torch.no\_grad()  
 def fuse\_self(self):  
 """  
 融合卷积层和批归一化层为一个卷积层。  
 """  
 c, bn = self.\_modules.values()  
 # 计算融合后的权重和偏置  
 w = bn.weight / (bn.running\_var + bn.eps)\*\*0.5  
 w = c.weight \* w[:, None, None, None]  
 b = bn.bias - bn.running\_mean \* bn.weight / (bn.running\_var + bn.eps)\*\*0.5  
 # 创建新的卷积层  
 m = nn.Conv2d(w.size(1) \* c.groups, w.size(0), w.shape[2:], stride=c.stride, padding=c.padding, dilation=c.dilation, groups=c.groups)  
 m.weight.data.copy\_(w)  
 m.bias.data.copy\_(b)  
 return m  
  
class Residual(nn.Module):  
 """  
 残差连接模块。  
 """  
 def \_\_init\_\_(self, m, drop=0.):  
 super().\_\_init\_\_()  
 self.m = m # 子模块  
 self.drop = drop # dropout比例  
  
 def forward(self, x):  
 # 如果在训练模式且drop大于0，则进行随机丢弃  
 if self.training and self.drop > 0:  
 return x + self.m(x) \* torch.rand(x.size(0), 1, 1, 1, device=x.device).ge\_(self.drop).div(1 - self.drop).detach()  
 else:  
 return x + self.m(x)  
  
class RepViTBlock(nn.Module):  
 """  
 RepViT的基本模块，包含token混合和channel混合。  
 """  
 def \_\_init\_\_(self, inp, hidden\_dim, oup, kernel\_size, stride, use\_se, use\_hs):  
 super(RepViTBlock, self).\_\_init\_\_()  
 assert stride in [1, 2]  
 self.identity = stride == 1 and inp == oup # 是否为身份连接  
 assert(hidden\_dim == 2 \* inp) # 隐藏层维度是输入的两倍  
  
 if stride == 2:  
 # 当stride为2时，使用token混合  
 self.token\_mixer = nn.Sequential(  
 Conv2d\_BN(inp, inp, kernel\_size, stride, (kernel\_size - 1) // 2, groups=inp),  
 SqueezeExcite(inp, 0.25) if use\_se else nn.Identity(),  
 Conv2d\_BN(inp, oup, ks=1, stride=1, pad=0)  
 )  
 # channel混合  
 self.channel\_mixer = Residual(nn.Sequential(  
 Conv2d\_BN(oup, 2 \* oup, 1, 1, 0),  
 nn.GELU() if use\_hs else nn.GELU(),  
 Conv2d\_BN(2 \* oup, oup, 1, 1, 0, bn\_weight\_init=0),  
 ))  
 else:  
 assert(self.identity)  
 self.token\_mixer = nn.Sequential(  
 RepVGGDW(inp),  
 SqueezeExcite(inp, 0.25) if use\_se else nn.Identity(),  
 )  
 self.channel\_mixer = Residual(nn.Sequential(  
 Conv2d\_BN(inp, hidden\_dim, 1, 1, 0),  
 nn.GELU() if use\_hs else nn.GELU(),  
 Conv2d\_BN(hidden\_dim, oup, 1, 1, 0, bn\_weight\_init=0),  
 ))  
  
 def forward(self, x):  
 return self.channel\_mixer(self.token\_mixer(x))  
  
class RepViT(nn.Module):  
 """  
 RepViT模型的定义。  
 """  
 def \_\_init\_\_(self, cfgs):  
 super(RepViT, self).\_\_init\_\_()  
 self.cfgs = cfgs # 模型配置  
 input\_channel = self.cfgs[0][2] # 输入通道数  
 # 构建初始层  
 patch\_embed = nn.Sequential(Conv2d\_BN(3, input\_channel // 2, 3, 2, 1), nn.GELU(),  
 Conv2d\_BN(input\_channel // 2, input\_channel, 3, 2, 1))  
 layers = [patch\_embed]  
 # 构建RepViT块  
 block = RepViTBlock  
 for k, t, c, use\_se, use\_hs, s in self.cfgs:  
 output\_channel = \_make\_divisible(c, 8)  
 exp\_size = \_make\_divisible(input\_channel \* t, 8)  
 layers.append(block(input\_channel, exp\_size, output\_channel, k, s, use\_se, use\_hs))  
 input\_channel = output\_channel  
 self.features = nn.ModuleList(layers)  
  
 def forward(self, x):  
 features = []  
 for f in self.features:  
 x = f(x)  
 features.append(x)  
 return features  
  
def repvit\_m2\_3(weights=''):  
 """  
 构建RepViT模型的一个变体。  
 """  
 cfgs = [  
 # k, t, c, SE, HS, s   
 [3, 2, 80, 1, 0, 1],  
 # 省略其他配置...  
 ]  
 model = RepViT(cfgs)  
 if weights:  
 model.load\_state\_dict(torch.load(weights)['model'])  
 return model  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 model = repvit\_m2\_3('repvit\_m2\_3\_distill\_450e.pth') # 加载模型  
 inputs = torch.randn((1, 3, 640, 640)) # 创建输入张量  
 res = model(inputs) # 前向传播  
 for i in res:  
 print(i.size()) # 输出每层的输出尺寸  
```  
  
### 代码核心部分说明：  
1. \*\*\_make\_divisible\*\*: 确保通道数是8的倍数，适用于某些网络结构的要求。  
2. \*\*Conv2d\_BN\*\*: 结合卷积层和批归一化层的模块，提供了融合功能以优化推理速度。  
3. \*\*Residual\*\*: 实现残差连接，支持在训练时随机丢弃部分输出以增强模型的鲁棒性。  
4. \*\*RepViTBlock\*\*: RepViT的基本构建块，负责token和channel的混合。  
5. \*\*RepViT\*\*: 整个RepViT模型的实现，负责根据配置构建网络结构。  
6. \*\*repvit\_m2\_3\*\*: 用于构建特定配置的RepViT模型并加载预训练权重。  
  
这些核心部分构成了RepViT模型的基础，负责处理输入数据并生成特征输出。```

这个文件 `repvit.py` 实现了一个名为 RepViT 的深度学习模型，主要用于计算机视觉任务。该模型的结构灵感来源于 MobileNet 和 Vision Transformer（ViT），并通过不同的模块组合来提高模型的性能和效率。  
  
首先，文件导入了必要的库，包括 PyTorch 的神经网络模块（`torch.nn`）、NumPy、以及来自 timm 库的 SqueezeExcite 层。接着，定义了一个公共接口 `\_\_all\_\_`，列出了可供外部调用的模型构造函数。  
  
接下来，定义了几个辅助函数。`replace\_batchnorm` 函数用于将模型中的 BatchNorm2d 层替换为 Identity 层，以便在推理时减少计算开销。`\_make\_divisible` 函数确保每一层的通道数是 8 的倍数，以满足某些硬件加速的要求。  
  
`Conv2d\_BN` 类是一个组合模块，包含卷积层和批归一化层，并提供了自我融合的方法 `fuse\_self`，该方法将卷积和批归一化合并为一个卷积层，以提高推理速度。  
  
`Residual` 类实现了残差连接，允许输入直接与经过卷积层处理的输出相加。该类同样提供了融合方法，以便在推理时优化模型。  
  
`RepVGGDW` 类是一个特定的卷积模块，包含深度可分离卷积和残差连接，能够在不同的通道维度上进行特征融合。  
  
`RepViTBlock` 类是模型的核心构建块，包含了通道混合和标记混合的功能，能够处理不同的输入通道和输出通道，支持 SqueezeExcite 和其他激活函数。  
  
`RepViT` 类是整个模型的实现，构造函数接收配置参数并构建模型的各个层。模型的前向传播方法 `forward` 处理输入数据并返回特征图。  
  
此外，`switch\_to\_deploy` 方法用于将模型切换到推理模式，替换掉 BatchNorm 层以优化性能。  
  
文件中还定义了多个函数（如 `repvit\_m0\_9`, `repvit\_m1\_0`, 等），这些函数根据不同的配置构建不同版本的 RepViT 模型，并可以加载预训练权重。  
  
最后，在 `\_\_main\_\_` 块中，示例代码展示了如何实例化一个 RepViT 模型并对随机输入进行前向传播，输出每个特征图的尺寸。  
  
整体而言，这个文件提供了一个灵活且高效的深度学习模型构建框架，适用于各种计算机视觉任务。

```以下是代码中最核心的部分，并附上详细的中文注释：  
  
```python  
import torch  
import torch.nn 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 # 输出特征数默认为输入特征数  
 hidden\_features = hidden\_features or in\_features # 隐藏层特征数默认为输入特征数  
 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 WindowAttention(nn.Module):  
 """ 窗口基础的多头自注意力 (W-MSA) 模块 """  
  
 def \_\_init\_\_(self, dim, window\_size, num\_heads, qkv\_bias=True, attn\_drop=0., proj\_drop=0.):  
 super().\_\_init\_\_()  
 self.dim = dim # 输入通道数  
 self.window\_size = window\_size # 窗口大小  
 self.num\_heads = num\_heads # 注意力头数  
 head\_dim = dim // num\_heads # 每个头的维度  
 self.scale = head\_dim \*\* -0.5 # 缩放因子  
  
 # 定义相对位置偏置参数表  
 self.relative\_position\_bias\_table = nn.Parameter(  
 torch.zeros((2 \* window\_size[0] - 1) \* (2 \* window\_size[1] - 1), num\_heads))  
  
 # 计算每个token在窗口内的相对位置索引  
 coords\_h = torch.arange(self.window\_size[0])  
 coords\_w = torch.arange(self.window\_size[1])  
 coords = torch.stack(torch.meshgrid([coords\_h, coords\_w])) # 生成坐标网格  
 coords\_flatten = torch.flatten(coords, 1) # 展平坐标  
 relative\_coords = coords\_flatten[:, :, None] - coords\_flatten[:, None, :] # 计算相对坐标  
 relative\_coords = relative\_coords.permute(1, 2, 0).contiguous() # 重新排列维度  
 relative\_coords[:, :, 0] += self.window\_size[0] - 1 # 偏移  
 relative\_coords[:, :, 1] += self.window\_size[1] - 1  
 relative\_coords[:, :, 0] \*= 2 \* self.window\_size[1] - 1  
 relative\_position\_index = relative\_coords.sum(-1) # 计算相对位置索引  
 self.register\_buffer("relative\_position\_index", relative\_position\_index) # 注册为缓冲区  
  
 self.qkv = nn.Linear(dim, dim \* 3, bias=qkv\_bias) # 线性变换生成Q、K、V  
 self.attn\_drop = nn.Dropout(attn\_drop) # 注意力权重的Dropout  
 self.proj = nn.Linear(dim, dim) # 输出线性变换  
 self.proj\_drop = nn.Dropout(proj\_drop) # 输出的Dropout  
 self.softmax = nn.Softmax(dim=-1) # Softmax层  
  
 def forward(self, x, mask=None):  
 """ 前向传播 """  
 B\_, N, C = x.shape # 获取输入的形状  
 qkv = self.qkv(x).reshape(B\_, N, 3, self.num\_heads, C // self.num\_heads).permute(2, 0, 3, 1, 4) # 计算Q、K、V  
 q, k, v = qkv[0], qkv[1], qkv[2] # 分离Q、K、V  
  
 q = q \* self.scale # 缩放Q  
 attn = (q @ k.transpose(-2, -1)) # 计算注意力分数  
  
 # 添加相对位置偏置  
 relative\_position\_bias = self.relative\_position\_bias\_table[self.relative\_position\_index.view(-1)].view(  
 self.window\_size[0] \* self.window\_size[1], self.window\_size[0] \* self.window\_size[1], -1)  
 relative\_position\_bias = relative\_position\_bias.permute(2, 0, 1).contiguous() # 重新排列维度  
 attn = attn + relative\_position\_bias.unsqueeze(0) # 加入相对位置偏置  
  
 if mask is not None:  
 attn = attn.view(B\_ // mask.shape[0], mask.shape[0], self.num\_heads, N, N) + mask.unsqueeze(1).unsqueeze(0)  
 attn = attn.view(-1, self.num\_heads, N, N)  
 attn = self.softmax(attn) # 应用Softmax  
 else:  
 attn = self.softmax(attn) # 应用Softmax  
  
 attn = self.attn\_drop(attn) # Dropout  
  
 x = (attn @ v).transpose(1, 2).reshape(B\_, N, C) # 计算输出  
 x = self.proj(x) # 线性变换  
 x = self.proj\_drop(x) # Dropout  
 return x  
  
  
class SwinTransformer(nn.Module):  
 """ Swin Transformer 主体 """  
  
 def \_\_init\_\_(self, patch\_size=4, in\_chans=3, embed\_dim=96, depths=[2, 2, 6, 2], num\_heads=[3, 6, 12, 24], window\_size=7):  
 super().\_\_init\_\_()  
  
 # 图像分块嵌入  
 self.patch\_embed = PatchEmbed(patch\_size=patch\_size, in\_chans=in\_chans, embed\_dim=embed\_dim)  
  
 # 构建各层  
 self.layers = nn.ModuleList()  
 for i\_layer in range(len(depths)):  
 layer = BasicLayer(  
 dim=int(embed\_dim \* 2 \*\* i\_layer),  
 depth=depths[i\_layer],  
 num\_heads=num\_heads[i\_layer],  
 window\_size=window\_size)  
 self.layers.append(layer)  
  
 def forward(self, x):  
 """ 前向传播 """  
 x = self.patch\_embed(x) # 图像分块嵌入  
 for layer in self.layers:  
 x, \_, \_, \_, \_, \_ = layer(x, x.size(2), x.size(3)) # 逐层前向传播  
 return x # 返回最终输出  
```  
  
### 代码核心部分解释：  
1. \*\*Mlp类\*\*：实现了一个简单的多层感知机，包含两个线性层和一个激活函数，支持Dropout。  
2. \*\*WindowAttention类\*\*：实现了窗口基础的多头自注意力机制，支持相对位置偏置。通过计算Q、K、V来获取注意力分数，并应用Softmax。  
3. \*\*SwinTransformer类\*\*：实现了Swin Transformer的主体结构，包含图像分块嵌入和多个基本层。每个基本层包含窗口自注意力和前馈网络。  
  
这些模块是Swin Transformer的核心构建块，结合起来实现了高效的视觉特征提取。```

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是一种基于Transformer架构的视觉模型，通过局部窗口注意力机制来提高计算效率和模型性能。以下是代码的详细说明。  
  
首先，导入了必要的库，包括PyTorch和一些辅助函数。接着定义了一个名为`Mlp`的类，它实现了一个多层感知机（MLP），包含两个线性层和一个激活函数（默认为GELU），并支持Dropout。  
  
接下来，定义了两个辅助函数`window\_partition`和`window\_reverse`，用于将输入特征图分割成小窗口和将窗口合并回原来的特征图。这对于局部窗口注意力机制非常重要。  
  
`WindowAttention`类实现了窗口基础的多头自注意力机制，支持相对位置偏置。该类中定义了查询、键、值的线性变换，并计算注意力权重。注意力权重会加上相对位置偏置，以捕捉窗口内的位置信息。  
  
`SwinTransformerBlock`类是Swin Transformer的基本构建块，包含了窗口注意力机制和前馈网络。它支持循环移位（shifted window）机制，通过在每个块之间进行窗口的循环移位来增强模型的表达能力。  
  
`PatchMerging`类用于将特征图的补丁合并，降低特征图的分辨率，同时增加通道数。这个过程在Swin Transformer的不同阶段之间进行。  
  
`BasicLayer`类代表了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。它还计算了SW-MSA的注意力掩码。  
  
`PatchEmbed`类负责将输入图像分割成补丁并进行嵌入，使用卷积层实现补丁的线性投影。  
  
`SwinTransformer`类是整个模型的主体，负责构建各个层次的结构，并处理输入数据。它支持绝对位置嵌入、随机深度、层归一化等功能。  
  
最后，定义了一个`update\_weight`函数，用于更新模型的权重，并提供了一个`SwinTransformer\_Tiny`函数，用于创建一个小型的Swin Transformer模型实例，并可选择加载预训练权重。  
  
整体来看，这个程序实现了Swin Transformer的完整结构，包含了模型的各个组件和必要的辅助函数，适合用于图像分类、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  
import torch.nn.functional as F  
  
# 定义相对位置的函数  
def rel\_pos(kernel\_size):  
 # 创建一个从-1到1的线性空间，步数为kernel\_size  
 tensors = [torch.linspace(-1, 1, steps=kernel\_size) for \_ in range(2)]  
 # 生成网格坐标  
 kernel\_coord = torch.stack(torch.meshgrid(\*tensors), dim=-0)  
 kernel\_coord = kernel\_coord.unsqueeze(0) # 增加一个维度  
 return kernel\_coord  
  
# 定义SMP卷积层  
class SMPConv(nn.Module):  
 def \_\_init\_\_(self, planes, kernel\_size, n\_points, stride, padding, groups):  
 super().\_\_init\_\_()  
  
 self.planes = planes # 输出通道数  
 self.kernel\_size = kernel\_size # 卷积核大小  
 self.n\_points = n\_points # 点的数量  
 self.init\_radius = 2 \* (2/kernel\_size) # 初始化半径  
  
 # 计算卷积核的相对位置  
 kernel\_coord = rel\_pos(kernel\_size)  
 self.register\_buffer('kernel\_coord', kernel\_coord) # 注册为buffer，不会被优化  
  
 # 初始化权重坐标  
 weight\_coord = torch.empty(1, n\_points, 2)  
 nn.init.trunc\_normal\_(weight\_coord, std=0.2, a=-1., b=1.) # 使用截断正态分布初始化  
 self.weight\_coord = nn.Parameter(weight\_coord) # 权重坐标作为可学习参数  
  
 # 初始化半径  
 self.radius = nn.Parameter(torch.empty(1, n\_points).unsqueeze(-1).unsqueeze(-1))  
 self.radius.data.fill\_(value=self.init\_radius)  
  
 # 初始化权重  
 weights = torch.empty(1, planes, n\_points)  
 nn.init.trunc\_normal\_(weights, std=.02) # 使用截断正态分布初始化  
 self.weights = nn.Parameter(weights) # 权重作为可学习参数  
  
 def forward(self, x):  
 # 生成卷积核  
 kernels = self.make\_kernels().unsqueeze(1)  
 x = x.contiguous() # 确保输入张量是连续的  
 kernels = kernels.contiguous()  
  
 # 根据输入数据类型选择合适的深度可分离卷积实现  
 if x.dtype == torch.float32:  
 x = \_DepthWiseConv2dImplicitGEMMFP32.apply(x, kernels)  
 elif x.dtype == torch.float16:  
 x = \_DepthWiseConv2dImplicitGEMMFP16.apply(x, kernels)  
 else:  
 raise TypeError("Only support fp32 and fp16, get {}".format(x.dtype))  
 return x   
  
 def make\_kernels(self):  
 # 计算卷积核的差异  
 diff = self.weight\_coord.unsqueeze(-2) - self.kernel\_coord.reshape(1, 2, -1).transpose(1, 2) # [1, n\_points, kernel\_size^2, 2]  
 diff = diff.transpose(2, 3).reshape(1, self.n\_points, 2, self.kernel\_size, self.kernel\_size)  
 diff = F.relu(1 - torch.sum(torch.abs(diff), dim=2) / self.radius) # 计算卷积核的权重  
  
 # 计算最终的卷积核  
 kernels = torch.matmul(self.weights, diff.reshape(1, self.n\_points, -1)) # [1, planes, kernel\_size\*kernel\_size]  
 kernels = kernels.reshape(1, self.planes, \*self.kernel\_coord.shape[2:]) # [1, planes, kernel\_size, kernel\_size]  
 kernels = kernels.squeeze(0)  
 kernels = torch.flip(kernels.permute(0, 2, 1), dims=(1,)) # 反转卷积核  
 return kernels  
  
# 定义SMPCNN模块  
class SMPCNN(nn.Module):  
 def \_\_init\_\_(self, in\_channels, out\_channels, kernel\_size, stride, groups, n\_points=None, n\_points\_divide=4):  
 super().\_\_init\_\_()  
 self.kernel\_size = kernel\_size  
 if n\_points is None:  
 n\_points = int((kernel\_size\*\*2) // n\_points\_divide) # 计算点的数量  
  
 padding = kernel\_size // 2  
 # 使用SMP卷积和标准卷积组合  
 self.smp = conv\_bn(in\_channels=in\_channels, out\_channels=out\_channels, kernel\_size=kernel\_size,  
 stride=stride, padding=padding, groups=groups, n\_points=n\_points)  
 self.small\_kernel = 5  
 self.small\_conv = Conv(in\_channels, out\_channels, self.small\_kernel, stride, self.small\_kernel // 2, groups, act=False)  
  
 def forward(self, inputs):  
 out = self.smp(inputs) # 通过SMP卷积层  
 out += self.small\_conv(inputs) # 加上小卷积的输出  
 return out  
  
# 定义SMPBlock模块  
class SMPBlock(nn.Module):  
 def \_\_init\_\_(self, in\_channels, dw\_channels, lk\_size, drop\_path, n\_points=None, n\_points\_divide=4):  
 super().\_\_init\_\_()  
 self.pw1 = conv\_bn\_relu(in\_channels, dw\_channels, 1, 1, 0, groups=1) # 逐点卷积  
 self.pw2 = conv\_bn(dw\_channels, in\_channels, 1, 1, 0, groups=1) # 逐点卷积  
 self.large\_kernel = SMPCNN(in\_channels=dw\_channels, out\_channels=dw\_channels, kernel\_size=lk\_size,  
 stride=1, groups=dw\_channels, n\_points=n\_points, n\_points\_divide=n\_points\_divide)  
 self.lk\_nonlinear = nn.ReLU() # 激活函数  
 self.drop\_path = DropPath(drop\_path) if drop\_path > 0. else nn.Identity() # 路径丢弃  
  
 def forward(self, x):  
 out = self.pw1(x) # 逐点卷积  
 out = self.large\_kernel(out) # 大卷积核处理  
 out = self.lk\_nonlinear(out) # 激活  
 out = self.pw2(out) # 逐点卷积  
 return x + self.drop\_path(out) # 残差连接  
```  
  
### 代码核心部分说明：  
1. \*\*SMPConv\*\*: 这是一个自定义的卷积层，使用了相对位置编码和可学习的权重坐标来生成卷积核。  
2. \*\*SMPCNN\*\*: 这个模块结合了SMP卷积和小卷积，增强了特征提取能力。  
3. \*\*SMPBlock\*\*: 这是一个块结构，包含逐点卷积和SMP卷积，支持残差连接和路径丢弃。  
  
这些部分是整个模型的核心，负责特征提取和信息传递。```

这个程序文件 `SMPConv.py` 定义了一些用于深度学习的卷积模块，主要是针对卷积神经网络（CNN）进行优化和扩展。代码中包含了多个类和函数，主要包括 `SMPConv`、`SMPCNN`、`SMPCNN\_ConvFFN` 和 `SMPBlock`，它们的设计目的是为了提高卷积操作的灵活性和效率。  
  
首先，文件中导入了一些必要的库，包括 PyTorch 的核心模块和一些自定义的卷积模块。接着，定义了一个辅助函数 `rel\_pos`，用于生成相对位置的张量，这在后续的卷积操作中会用到。  
  
`SMPConv` 类是文件的核心部分，它继承自 `nn.Module`。在初始化方法中，定义了一些参数，包括输出通道数、卷积核大小、点数、步幅和填充等。通过调用 `rel\_pos` 函数，生成卷积核的坐标，并注册为缓冲区。接着，初始化权重坐标和半径，并定义权重参数。`forward` 方法实现了前向传播，调用 `make\_kernels` 方法生成卷积核，并根据输入数据的类型选择合适的深度可分离卷积实现。  
  
`make\_kernels` 方法计算卷积核的具体形状和权重，使用了重塑和矩阵乘法等操作来生成最终的卷积核。该方法还实现了对半径的限制，以确保在训练过程中保持稳定。  
  
`get\_conv2d` 函数根据输入参数决定使用自定义的 `SMPConv` 还是标准的 `nn.Conv2d`，以便在特定条件下使用优化的卷积实现。`enable\_sync\_bn` 和 `get\_bn` 函数用于选择同步批归一化或普通批归一化。  
  
`conv\_bn` 和 `conv\_bn\_relu` 函数分别用于构建带有批归一化和激活函数的卷积层，方便在模型中使用。  
  
`SMPCNN` 类结合了 `SMPConv` 和一个小卷积层，通过前向传播将两者的输出相加，以增强特征提取能力。  
  
`SMPCNN\_ConvFFN` 类实现了一个前馈网络模块，包含了两个逐点卷积层和一个非线性激活函数，结合了 DropPath 技术以提高模型的鲁棒性。  
  
最后，`SMPBlock` 类将多个模块组合在一起，形成一个完整的块结构，包括逐点卷积、SMP 卷积和激活函数，并实现了残差连接。  
  
整体来看，这个文件实现了一种灵活的卷积结构，旨在通过自定义的卷积操作和组合策略来提高卷积神经网络的性能，适用于各种深度学习任务。

```以下是经过简化并注释的核心代码部分：  
  
```python  
import torch  
import torch.nn as nn  
import numpy as np  
from torch.nn import Dropout, Softmax, LayerNorm  
  
class Channel\_Embeddings(nn.Module):  
 """从图像的补丁和位置生成嵌入"""  
 def \_\_init\_\_(self, patchsize, img\_size, in\_channels):  
 super().\_\_init\_\_()  
 img\_size = (img\_size, img\_size) # 将图像大小转换为元组  
 patch\_size = (patchsize, patchsize) # 将补丁大小转换为元组  
 n\_patches = (img\_size[0] // patch\_size[0]) \* (img\_size[1] // patch\_size[1]) # 计算补丁数量  
  
 # 定义补丁嵌入层  
 self.patch\_embeddings = nn.Sequential(  
 nn.MaxPool2d(kernel\_size=5, stride=5), # 最大池化层  
 nn.Conv2d(in\_channels=in\_channels,  
 out\_channels=in\_channels,  
 kernel\_size=patchsize // 5,  
 stride=patchsize // 5) # 卷积层  
 )  
  
 # 定义位置嵌入参数  
 self.position\_embeddings = nn.Parameter(torch.zeros(1, n\_patches, in\_channels))  
 self.dropout = Dropout(0.1) # Dropout层  
  
 def forward(self, x):  
 """前向传播，生成嵌入"""  
 if x is None:  
 return None  
 x = self.patch\_embeddings(x) # 通过补丁嵌入层  
 x = x.flatten(2) # 展平  
 x = x.transpose(-1, -2) # 转置  
 embeddings = x + self.position\_embeddings # 添加位置嵌入  
 embeddings = self.dropout(embeddings) # 应用Dropout  
 return embeddings  
  
class Attention\_org(nn.Module):  
 """自定义的多头注意力机制"""  
 def \_\_init\_\_(self, vis, channel\_num):  
 super(Attention\_org, self).\_\_init\_\_()  
 self.vis = vis  
 self.KV\_size = sum(channel\_num) # 键值对的大小  
 self.channel\_num = channel\_num  
 self.num\_attention\_heads = 4 # 注意力头的数量  
  
 # 定义查询、键、值的线性变换  
 self.query = nn.ModuleList([nn.Linear(c, c, bias=False) for c in channel\_num])  
 self.key = nn.Linear(self.KV\_size, self.KV\_size, bias=False)  
 self.value = nn.Linear(self.KV\_size, self.KV\_size, bias=False)  
 self.softmax = Softmax(dim=3) # Softmax层  
 self.attn\_dropout = Dropout(0.1) # Dropout层  
  
 def forward(self, \*embeddings):  
 """前向传播，计算注意力"""  
 multi\_head\_Q = [query(emb) for query, emb in zip(self.query, embeddings) if emb is not None]  
 multi\_head\_K = self.key(torch.cat(embeddings, dim=2)) # 合并所有嵌入  
 multi\_head\_V = self.value(torch.cat(embeddings, dim=2))  
  
 # 计算注意力分数  
 attention\_scores = [torch.matmul(Q, multi\_head\_K) / np.sqrt(self.KV\_size) for Q in multi\_head\_Q]  
 attention\_probs = [self.softmax(score) for score in attention\_scores] # 计算注意力概率  
  
 # 应用Dropout  
 attention\_probs = [self.attn\_dropout(prob) for prob in attention\_probs]  
 context\_layers = [torch.matmul(prob, multi\_head\_V) for prob in attention\_probs] # 计算上下文层  
  
 return context\_layers # 返回上下文层  
  
class ChannelTransformer(nn.Module):  
 """通道变换器模型"""  
 def \_\_init\_\_(self, channel\_num=[64, 128, 256, 512], img\_size=640, patchSize=[40, 20, 10, 5]):  
 super().\_\_init\_\_()  
 self.embeddings = nn.ModuleList([Channel\_Embeddings(patchSize[i], img\_size // (2 \*\* (i + 2)), channel\_num[i]) for i in range(len(channel\_num))])  
 self.encoder = Encoder(channel\_num) # 编码器  
 self.reconstruct = nn.ModuleList([Reconstruct(channel\_num[i], channel\_num[i], kernel\_size=1, scale\_factor=(patchSize[i], patchSize[i])) for i in range(len(channel\_num))]) # 重构层  
  
 def forward(self, en):  
 """前向传播，处理输入数据"""  
 embeddings = [embed(en[i]) for i, embed in enumerate(self.embeddings) if en[i] is not None]  
 encoded = self.encoder(\*embeddings) # 编码  
 reconstructed = [recon(enc) + en[i] for i, (recon, enc) in enumerate(zip(self.reconstruct, encoded)) if en[i] is not None] # 重构并加上残差  
 return reconstructed # 返回重构后的输出  
```  
  
### 代码说明：  
1. \*\*Channel\_Embeddings\*\*：这个类负责将输入图像转换为补丁嵌入，并添加位置嵌入。  
2. \*\*Attention\_org\*\*：实现了一个多头注意力机制，计算输入嵌入之间的注意力分数并生成上下文层。  
3. \*\*ChannelTransformer\*\*：这是整个模型的核心部分，负责处理输入数据，通过嵌入、编码和重构步骤生成最终输出。```

这个程序文件 `CTrans.py` 实现了一个名为 `ChannelTransformer` 的深度学习模型，主要用于图像处理任务。该模型基于变换器（Transformer）架构，特别关注于通道信息的处理。以下是对代码的详细讲解。  
  
首先，文件引入了一些必要的库，包括 `torch` 和 `torch.nn`，这些库提供了构建神经网络所需的基本模块和函数。接着，定义了几个类，每个类都有其特定的功能。  
  
`Channel\_Embeddings` 类用于构建图像的嵌入表示。它接收图像的尺寸和通道数，并通过卷积和池化操作将图像划分为多个补丁。补丁的嵌入通过位置嵌入进行增强，并添加了 dropout 层以防止过拟合。  
  
`Reconstruct` 类负责将经过处理的嵌入重新构建为图像。它使用卷积层和上采样操作来实现这一点，并通过批归一化和激活函数（ReLU）进行处理。  
  
`Attention\_org` 类实现了多头注意力机制。它接受多个嵌入作为输入，计算查询（Query）、键（Key）和值（Value），并通过注意力得分计算出上下文层。该类还包含 dropout 层以增强模型的鲁棒性。  
  
`Mlp` 类实现了一个简单的多层感知机（MLP），包含两个全连接层和一个激活函数（GELU）。它用于在模型中进行特征变换。  
  
`Block\_ViT` 类将注意力机制和前馈网络结合在一起，形成一个完整的变换器块。它首先对输入的嵌入进行层归一化，然后通过注意力机制进行处理，最后通过 MLP 进行特征转换。  
  
`Encoder` 类包含多个 `Block\_ViT` 层，负责对输入的嵌入进行编码。它通过层归一化对每个通道的嵌入进行处理，并在每个变换器块中传递注意力权重。  
  
`ChannelTransformer` 类是整个模型的核心。它初始化了多个嵌入层、编码器和重构层。模型的前向传播方法接收输入图像，计算嵌入，经过编码器处理后，再通过重构层输出结果。  
  
最后，`GetIndexOutput` 类用于从模型的输出中提取特定索引的结果。  
  
整体来看，这个模型通过结合多头注意力机制和前馈网络，能够有效地处理图像的通道信息，并在图像重建任务中表现出色。每个模块的设计都考虑到了深度学习中的常见技术，如层归一化、dropout 和激活函数等，以提高模型的性能和稳定性。

### 整体功能和构架概括  
  
这个工程包含多个深度学习模型的实现，主要针对计算机视觉任务。每个文件实现了不同类型的模型或模块，采用了现代深度学习中的一些关键技术，如自注意力机制、卷积操作和多层感知机等。整体架构旨在通过不同的模型结构和模块组合，提升图像处理的性能和效率。  
  
- \*\*`repvit.py`\*\*：实现了 RepViT 模型，结合了卷积和 Transformer 的优点，适用于图像分类和目标检测等任务。  
- \*\*`SwinTransformer.py`\*\*：实现了 Swin Transformer 模型，采用局部窗口注意力机制，适合处理高分辨率图像。  
- \*\*`SMPConv.py`\*\*：定义了一些自定义卷积模块，旨在提高卷积操作的灵活性和效率，适用于各种卷积神经网络的构建。  
- \*\*`CTrans.py`\*\*：实现了 Channel Transformer 模型，专注于通道信息的处理，适用于图像重建和处理任务。  
  
### 文件功能整理表  
  
| 文件名 | 功能描述 |  
|---------------------|----------------------------------------------------------|  
| `repvit.py` | 实现 RepViT 模型，结合卷积和 Transformer，适用于图像分类和目标检测。 |  
| `SwinTransformer.py`| 实现 Swin Transformer 模型，采用局部窗口注意力机制，适合高分辨率图像处理。 |  
| `SMPConv.py` | 定义自定义卷积模块，增强卷积操作的灵活性和效率，适用于卷积神经网络。 |  
| `CTrans.py` | 实现 Channel Transformer 模型，专注于通道信息处理，适用于图像重建任务。 |  
  
这个表格总结了每个文件的主要功能，帮助理解整个工程的结构和目的。