# 改进yolo11-ASF-DySample等200+全套创新点大全：发票信息提取系统源码＆数据集全套

## 1. 图片效果展示

注意：由于项目一直在更新迭代，上面“1.图片效果展示”和“2.视频效果展示”展示的系统图片或者视频可能为老版本，新版本在老版本的基础上升级如下：（实际效果以升级的新版本为准）  
  
 （1）适配了YOLOV11的“目标检测”模型和“实例分割”模型，通过加载相应的权重（.pt）文件即可自适应加载模型。  
  
 （2）支持“图片识别”、“视频识别”、“摄像头实时识别”三种识别模式。  
  
 （3）支持“图片识别”、“视频识别”、“摄像头实时识别”三种识别结果保存导出，解决手动导出（容易卡顿出现爆内存）存在的问题，识别完自动保存结果并导出到tempDir中。  
  
 （4）支持Web前端系统中的标题、背景图等自定义修改。  
  
 另外本项目提供训练的数据集和训练教程,暂不提供权重文件（best.pt）,需要您按照教程进行训练后实现图片演示和Web前端界面演示的效果。

## 2.视频效果展示

## 3.背景

研究背景与意义  
  
随着电子商务的迅猛发展，发票作为商业交易的重要凭证，其信息的自动提取与处理变得愈发重要。传统的发票信息提取方法多依赖人工输入，效率低下且易出错，无法满足现代企业对数据处理速度和准确性的高要求。因此，开发一种高效、准确的发票信息提取系统显得尤为必要。近年来，深度学习技术的飞速发展为图像识别和信息提取提供了新的解决方案，其中YOLO（You Only Look Once）系列模型因其实时性和高精度而备受关注。  
  
本研究旨在基于改进的YOLOv11模型，构建一个高效的发票信息提取系统。我们所使用的数据集包含756张发票图像，涵盖了多个重要信息类别，如地址、商品、日期、价格、税务信息等。这些类别不仅是发票信息提取的核心内容，也是后续财务分析和数据挖掘的基础。通过对这些类别的准确识别与提取，企业能够实现自动化的财务管理，提升工作效率，降低人工成本。  
  
在技术实现上，YOLOv11模型通过其创新的网络结构和训练方法，能够在保证高精度的同时，提升信息提取的速度。针对发票图像的特性，我们将对模型进行针对性的改进，以适应不同发票格式和排版的多样性。此外，结合数据增强技术和迁移学习策略，我们期望进一步提升模型的泛化能力，使其在实际应用中能够有效应对各种复杂场景。  
  
综上所述，本研究不仅具有重要的理论意义，推动了深度学习在信息提取领域的应用，也具备广泛的实际应用价值，能够为企业的财务管理提供有力支持，助力数字化转型的进程。

## 4. 数据集信息展示

本项目数据集信息介绍  
  
本项目所使用的数据集名为“Invoice Detection”，旨在为改进YOLOv11的发票信息提取系统提供支持。该数据集包含12个类别，涵盖了发票中常见的关键信息，具体类别包括：Billa、Hofer、Spar、Unimarkt、地址、商品、日期、发票、支付信息、价格、总计以及税务信息。这些类别的设计充分考虑了发票的实际应用场景，确保系统能够准确提取并识别各种重要信息。  
  
在数据集的构建过程中，我们采集了多种类型的发票样本，确保数据的多样性和代表性。这些样本来自不同的商家和服务提供商，涵盖了多种格式和布局，以模拟真实世界中可能遇到的各种情况。通过这种方式，我们的目标是提高YOLOv11在处理不同发票时的鲁棒性和准确性，进而提升信息提取的效率。  
  
数据集中的每个类别都经过精心标注，以确保训练过程中模型能够学习到每种信息的特征。例如，商品类别不仅包括商品名称，还涵盖了数量和单价等信息，而支付信息则涉及到支付方式和交易编号等细节。这种细致的标注使得模型在训练时能够更好地理解和区分不同类别之间的关系，从而提高提取的准确性。  
  
此外，为了确保模型的泛化能力，我们还进行了数据增强处理，包括旋转、缩放和颜色调整等。这些技术手段不仅丰富了训练数据的多样性，还有效降低了模型对特定样本的过拟合风险。通过这样的数据集构建和处理策略，我们期望能够显著提升YOLOv11在发票信息提取任务中的表现，为实际应用提供更加高效和可靠的解决方案。

## 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  
  
# 定义一个h\_sigmoid激活函数  
class h\_sigmoid(nn.Module):  
 def \_\_init\_\_(self, inplace=True):  
 super(h\_sigmoid, self).\_\_init\_\_()  
 self.relu = nn.ReLU6(inplace=inplace) # 使用ReLU6作为基础  
  
 def forward(self, x):  
 return self.relu(x + 3) / 6 # 计算h\_sigmoid  
  
# 定义一个h\_swish激活函数  
class h\_swish(nn.Module):  
 def \_\_init\_\_(self, inplace=True):  
 super(h\_swish, self).\_\_init\_\_()  
 self.sigmoid = h\_sigmoid(inplace=inplace) # 使用h\_sigmoid作为基础  
  
 def forward(self, x):  
 return x \* self.sigmoid(x) # 计算h\_swish  
  
# 定义RFAConv模块  
class RFAConv(nn.Module):  
 def \_\_init\_\_(self, in\_channel, out\_channel, kernel\_size, stride=1):  
 super().\_\_init\_\_()  
 self.kernel\_size = kernel\_size  
  
 # 生成权重的卷积操作  
 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)  
 )  
   
 # 生成特征的卷积操作  
 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()  
 )  
   
 # 最终的卷积操作  
 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batch size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)  
 feature = self.generate\_feature(x).view(b, c, self.kernel\_size \*\* 2, h, w) # 生成特征  
   
 # 权重与特征相乘  
 weighted\_data = feature \* weighted  
 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) # 重排数据  
   
 return self.conv(conv\_data) # 返回卷积结果  
  
# 定义SE模块（Squeeze-and-Excitation）  
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到c/r  
 nn.ReLU(),  
 nn.Linear(ratio, in\_channel, bias=False), # 从c/r到c  
 nn.Sigmoid()  
 )  
  
 def forward(self, x):  
 b, c = x.shape[0:2] # 获取输入的batch size和通道数  
 y = self.gap(x).view(b, c) # 进行全局平均池化  
 y = self.fc(y).view(b, c, 1, 1) # 通过全连接层  
 return y # 返回通道注意力  
  
# 定义RFCBAMConv模块  
class RFCBAMConv(nn.Module):  
 def \_\_init\_\_(self, in\_channel, out\_channel, kernel\_size=3, stride=1):  
 super().\_\_init\_\_()  
 self.kernel\_size = kernel\_size  
   
 # 生成特征的卷积操作  
 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()  
 )  
   
 # 获取权重的卷积操作  
 self.get\_weight = nn.Sequential(nn.Conv2d(2, 1, kernel\_size=3, padding=1, bias=False), nn.Sigmoid())  
 self.se = SE(in\_channel) # 初始化SE模块  
  
 # 最终的卷积操作  
 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batch size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) # 重排特征  
   
 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) # 重排数据  
   
 unfold\_feature = generate\_feature \* channel\_attention # 应用通道注意力  
 max\_feature, \_ = torch.max(generate\_feature, dim=1, keepdim=True) # 最大特征  
 mean\_feature = torch.mean(generate\_feature, dim=1, keepdim=True) # 平均特征  
   
 # 计算接收场注意力  
 receptive\_field\_attention = self.get\_weight(torch.cat((max\_feature, mean\_feature), dim=1))  
 conv\_data = unfold\_feature \* receptive\_field\_attention # 应用接收场注意力  
   
 return self.conv(conv\_data) # 返回卷积结果  
```  
  
### 代码分析：  
1. \*\*激活函数\*\*：  
 - `h\_sigmoid` 和 `h\_swish` 是自定义的激活函数，分别实现了h-sigmoid和h-swish的功能，主要用于网络中的非线性变换。  
  
2. \*\*RFAConv\*\*：  
 - 该模块实现了一种基于注意力机制的卷积操作，首先通过平均池化和卷积生成权重，然后生成特征，最后将特征与权重相乘，经过重排后进行卷积。  
  
3. \*\*SE模块\*\*：  
 - Squeeze-and-Excitation模块通过全局平均池化和全连接层生成通道注意力，用于增强网络对重要特征的关注。  
  
4. \*\*RFCBAMConv\*\*：  
 - 该模块结合了特征生成、通道注意力和接收场注意力，通过多个卷积操作和特征重排，增强了特征的表达能力。  
  
这些模块的设计旨在提高卷积神经网络的性能，特别是在处理复杂图像任务时。```

这个程序文件 `RFAConv.py` 定义了一些用于深度学习卷积操作的模块，主要包括 RFAConv、RFCBAMConv 和 RFCAConv。这些模块利用了深度学习中的一些新技术，比如注意力机制和非线性激活函数，来增强卷积神经网络的性能。  
  
首先，文件中导入了必要的库，包括 PyTorch 和 einops。PyTorch 是一个流行的深度学习框架，而 einops 是一个用于重排张量的库。接着，定义了两个自定义的激活函数：`h\_sigmoid` 和 `h\_swish`。`h\_sigmoid` 是一种改进的 sigmoid 函数，`h\_swish` 则是基于 `h\_sigmoid` 的一种新型激活函数，它结合了 ReLU 和 sigmoid 的特性。  
  
接下来是 `RFAConv` 类的定义。这个类实现了一种新的卷积操作，称为 RFA（Receptive Field Attention）卷积。构造函数中，`get\_weight` 用于生成权重，`generate\_feature` 用于生成特征。`forward` 方法中，输入数据经过处理后生成加权特征，并通过重排操作将其格式化为适合卷积的形状，最后通过 `self.conv` 进行卷积操作。  
  
然后是 `SE` 类的定义，它实现了 Squeeze-and-Excitation（SE）模块。该模块通过全局平均池化和全连接层来生成通道注意力，增强特征表示。`forward` 方法中，输入数据经过池化和全连接层处理后生成通道注意力。  
  
接着是 `RFCBAMConv` 类的定义，它结合了 RFA 和 SE 模块。构造函数中，`generate` 用于生成特征，`get\_weight` 用于生成注意力权重。`forward` 方法中，首先计算通道注意力，然后生成特征并进行重排，最后通过注意力机制和卷积操作生成输出。  
  
最后是 `RFCAConv` 类的定义，它结合了 RFA 和通道注意力机制。构造函数中，`generate` 用于生成特征，`pool\_h` 和 `pool\_w` 用于对特征进行池化。`forward` 方法中，生成特征后分别进行水平和垂直池化，生成通道注意力，并最终通过卷积操作生成输出。  
  
总体来说，这个文件实现了一些复杂的卷积操作和注意力机制，旨在提高卷积神经网络在图像处理任务中的表现。通过这些模块，用户可以构建更为高效和强大的深度学习模型。

```以下是经过简化并添加详细中文注释的核心代码部分：  
  
```python  
import torch  
import torch.nn as nn  
import torch.nn.functional as F  
  
class DWConv2d(nn.Module):  
 """ 深度可分离卷积层 """  
 def \_\_init\_\_(self, dim, kernel\_size, stride, padding):  
 super().\_\_init\_\_()  
 # 使用深度可分离卷积，groups=dim表示每个输入通道独立卷积  
 self.conv = nn.Conv2d(dim, dim, kernel\_size, stride, padding, groups=dim)  
  
 def forward(self, x: torch.Tensor):  
 '''  
 x: 输入张量，形状为 (b, h, w, c)  
 '''  
 x = x.permute(0, 3, 1, 2) # 转换为 (b, c, h, w)  
 x = self.conv(x) # 进行卷积操作  
 x = x.permute(0, 2, 3, 1) # 转换回 (b, h, w, c)  
 return x  
  
class MaSA(nn.Module):  
 """ 多头自注意力机制 """  
 def \_\_init\_\_(self, embed\_dim, num\_heads, value\_factor=1):  
 super().\_\_init\_\_()  
 self.factor = value\_factor  
 self.embed\_dim = embed\_dim  
 self.num\_heads = num\_heads  
 self.head\_dim = self.embed\_dim \* self.factor // num\_heads  
 self.key\_dim = self.embed\_dim // num\_heads  
 self.scaling = self.key\_dim \*\* -0.5  
   
 # 线性变换层  
 self.q\_proj = nn.Linear(embed\_dim, embed\_dim, bias=True)  
 self.k\_proj = nn.Linear(embed\_dim, embed\_dim, bias=True)  
 self.v\_proj = nn.Linear(embed\_dim, embed\_dim \* self.factor, bias=True)  
 self.out\_proj = nn.Linear(embed\_dim \* self.factor, embed\_dim, bias=True)  
  
 def forward(self, x: torch.Tensor, rel\_pos):  
 '''  
 x: 输入张量，形状为 (b, h, w, c)  
 rel\_pos: 位置关系的掩码  
 '''  
 bsz, h, w, \_ = x.size()  
 q = self.q\_proj(x) # 计算查询  
 k = self.k\_proj(x) # 计算键  
 v = self.v\_proj(x) # 计算值  
  
 # 计算注意力权重  
 qk\_mat = (q @ k.transpose(-1, -2)) \* self.scaling + rel\_pos # 加入位置关系  
 qk\_mat = torch.softmax(qk\_mat, dim=-1) # 归一化  
  
 # 计算输出  
 output = (qk\_mat @ v) # 注意力加权  
 output = self.out\_proj(output) # 最终线性变换  
 return output  
  
class FeedForwardNetwork(nn.Module):  
 """ 前馈神经网络 """  
 def \_\_init\_\_(self, embed\_dim, ffn\_dim, activation\_fn=F.gelu, dropout=0.0):  
 super().\_\_init\_\_()  
 self.fc1 = nn.Linear(embed\_dim, ffn\_dim) # 第一层线性变换  
 self.fc2 = nn.Linear(ffn\_dim, embed\_dim) # 第二层线性变换  
 self.dropout = nn.Dropout(dropout) # Dropout层  
 self.activation\_fn = activation\_fn # 激活函数  
  
 def forward(self, x: torch.Tensor):  
 '''  
 x: 输入张量，形状为 (b, h, w, c)  
 '''  
 x = self.fc1(x) # 第一层  
 x = self.activation\_fn(x) # 激活  
 x = self.dropout(x) # Dropout  
 x = self.fc2(x) # 第二层  
 return x  
  
class VisRetNet(nn.Module):  
 """ 可视化恢复网络 """  
 def \_\_init\_\_(self, in\_chans=3, num\_classes=1000, embed\_dims=[96, 192, 384, 768], depths=[2, 2, 6, 2], num\_heads=[3, 6, 12, 24]):  
 super().\_\_init\_\_()  
 self.patch\_embed = PatchEmbed(in\_chans=in\_chans, embed\_dim=embed\_dims[0]) # 图像分块嵌入  
 self.layers = nn.ModuleList() # 存储各层  
  
 # 构建网络层  
 for i\_layer in range(len(depths)):  
 layer = BasicLayer(embed\_dim=embed\_dims[i\_layer], depth=depths[i\_layer], num\_heads=num\_heads[i\_layer])  
 self.layers.append(layer)  
  
 def forward(self, x):  
 x = self.patch\_embed(x) # 图像嵌入  
 for layer in self.layers:  
 x = layer(x) # 逐层传递  
 return x  
  
# 这里省略了 PatchEmbed 和 BasicLayer 的实现，假设它们已定义  
  
# 模型实例化示例  
if \_\_name\_\_ == '\_\_main\_\_':  
 model = VisRetNet()  
 inputs = torch.randn((1, 3, 640, 640)) # 随机输入  
 res = model(inputs) # 前向传播  
 print(res.size()) # 输出结果的形状  
```  
  
### 代码说明：  
1. \*\*DWConv2d\*\*: 实现了深度可分离卷积，适用于图像特征提取。  
2. \*\*MaSA\*\*: 实现了多头自注意力机制，能够捕捉输入特征之间的关系。  
3. \*\*FeedForwardNetwork\*\*: 实现了前馈神经网络，包含两层线性变换和激活函数。  
4. \*\*VisRetNet\*\*: 主要网络结构，包含图像嵌入和多个基本层的堆叠。```

这个程序文件 `rmt.py` 实现了一个视觉变换器（Vision Transformer）模型，名为 `VisRetNet`，并提供了不同规模的模型构造函数（如 `RMT\_T`, `RMT\_S`, `RMT\_B`, `RMT\_L`）。该模型主要用于图像处理任务，尤其是在计算机视觉领域。  
  
文件首先导入了必要的库，包括 PyTorch 和一些特定的模块。接下来定义了一些基础组件，如 `DWConv2d`（深度可分离卷积）、`RelPos2d`（二维相对位置编码）、`MaSAd` 和 `MaSA`（不同类型的注意力机制），以及 `FeedForwardNetwork`（前馈神经网络）等。这些组件构成了模型的基础。  
  
`DWConv2d` 类实现了深度可分离卷积，它在输入的通道维度上进行卷积操作，适用于处理图像数据。`RelPos2d` 类用于生成相对位置编码，支持在二维空间中计算位置关系，帮助模型理解输入数据的空间结构。  
  
`MaSAd` 和 `MaSA` 类实现了多头自注意力机制，分别支持不同的注意力计算方式。它们通过查询（Q）、键（K）和值（V）的线性变换来计算注意力权重，并结合相对位置编码来增强模型的上下文理解能力。  
  
`FeedForwardNetwork` 类实现了一个简单的前馈网络，包含两个线性层和激活函数，常用于 transformer 模型中的每个层。  
  
`RetBlock` 类是一个包含注意力机制和前馈网络的基本模块，它通过残差连接和层归一化来增强模型的稳定性。`PatchMerging` 类则用于将输入特征图进行下采样，合并成更小的特征图。  
  
`BasicLayer` 类构建了一个基本的 transformer 层，包含多个 `RetBlock` 组成的堆叠，并在最后可选择性地进行下采样。  
  
`LayerNorm2d` 和 `PatchEmbed` 类分别实现了二维层归一化和图像到补丁的嵌入，将输入图像转换为适合 transformer 处理的格式。  
  
`VisRetNet` 类是整个模型的核心，负责将所有层和模块组合在一起。它支持不同的输入通道、嵌入维度、层数、头数等超参数，并通过 `forward` 方法实现前向传播。  
  
最后，文件提供了四个函数（`RMT\_T`, `RMT\_S`, `RMT\_B`, `RMT\_L`）用于创建不同规模的 `VisRetNet` 模型。这些函数根据预设的参数配置模型的各个部分。  
  
在 `\_\_main\_\_` 部分，代码示例展示了如何创建一个小型模型并进行一次前向传播，输出每个特征图的尺寸。  
  
总体而言，这个文件实现了一个灵活且可扩展的视觉变换器模型，适用于各种计算机视觉任务。

```以下是经过简化并添加详细中文注释的核心代码部分：  
  
```python  
import torch  
import torch.nn as nn  
import 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 # 是否处于部署模式  
 # 权重参数初始化  
 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，使用卷积层处理输入  
 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，融合批归一化  
 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的参数  
 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  
  
# 定义基本块  
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  
 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  
  
# 定义VanillaNet模型  
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()  
 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) # 添加每个Block到模型中  
  
 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进行前向传播  
 return x  
  
# 示例用法  
if \_\_name\_\_ == '\_\_main\_\_':  
 inputs = torch.randn((1, 3, 640, 640)) # 输入张量  
 model = VanillaNet() # 创建模型实例  
 pred = model(inputs) # 进行前向传播  
 print(pred.size()) # 输出预测结果的尺寸  
```  
  
### 代码说明：  
1. \*\*Activation类\*\*：自定义的激活函数类，包含了卷积和批归一化的融合操作。  
2. \*\*Block类\*\*：网络的基本构建块，包含卷积、池化和激活操作。  
3. \*\*VanillaNet类\*\*：主网络结构，包含stem部分和多个Block的组合。  
4. \*\*前向传播\*\*：每个类都有`forward`方法，定义了数据如何通过网络流动。  
5. \*\*示例用法\*\*：在主程序中创建一个输入张量并通过模型进行前向传播，输出结果的尺寸。```

这个程序文件 `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。代码中使用了 PyTorch 框架，并且包含了一些自定义的层和模块，以增强模型的功能和灵活性。  
  
首先，文件开头包含版权声明和许可证信息，表明该程序是开源的，并且可以在 MIT 许可证下进行修改和分发。  
  
接下来，程序导入了必要的库，包括 PyTorch 的核心模块和一些辅助功能模块。`timm.layers` 中的 `weight\_init` 和 `DropPath` 用于权重初始化和路径丢弃操作。  
  
文件中定义了多个类和函数。`activation` 类是一个自定义的激活函数层，继承自 ReLU。它的构造函数中初始化了权重和偏置，并使用批量归一化。前向传播方法根据是否处于部署模式（`deploy`）来选择不同的计算方式。  
  
`Block` 类表示网络中的一个基本构建块，包含两个卷积层和一个激活层。根据是否处于部署模式，`Block` 的构造函数会选择不同的实现方式。它还包含一个池化层，能够根据步幅的不同选择最大池化或自适应池化。前向传播方法在执行卷积和激活操作后，进行池化处理。  
  
`VanillaNet` 类是整个网络的主体，构造函数中定义了网络的输入通道、类别数、各层的维度、丢弃率等参数。它根据给定的步幅和维度创建多个 `Block` 实例，并将它们添加到一个模块列表中。`forward` 方法实现了网络的前向传播，返回不同尺度的特征图。  
  
此外，文件中还定义了一些辅助函数，例如 `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，并支持加载预训练权重。  
  
最后，在 `\_\_main\_\_` 块中，创建了一个输入张量并实例化了 `vanillanet\_10` 模型，随后进行前向传播并打印输出特征图的尺寸。  
  
总体而言，这个程序文件实现了一个灵活且可扩展的深度学习模型，适用于各种图像处理任务，并提供了多种模型配置以满足不同需求。

```以下是提取后的核心代码部分，并添加了详细的中文注释：  
  
```python  
import torch  
import torch.nn as nn  
import 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 CSWinBlock(nn.Module):  
 """CSWin Transformer的基本块"""  
 def \_\_init\_\_(self, dim, num\_heads, mlp\_ratio=4., drop=0., attn\_drop=0., norm\_layer=nn.LayerNorm):  
 super().\_\_init\_\_()  
 self.dim = dim # 输入特征维度  
 self.num\_heads = num\_heads # 注意力头数  
 self.mlp\_ratio = mlp\_ratio # MLP的扩展比例  
 self.qkv = nn.Linear(dim, dim \* 3) # 线性变换生成Q、K、V  
 self.norm1 = norm\_layer(dim) # 第一层归一化  
 self.attn = LePEAttention(dim, num\_heads=num\_heads, attn\_drop=attn\_drop) # 注意力模块  
 self.mlp = Mlp(in\_features=dim, hidden\_features=int(dim \* mlp\_ratio), out\_features=dim) # MLP模块  
 self.norm2 = norm\_layer(dim) # 第二层归一化  
  
 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) # 注意力计算  
 x = x + self.mlp(self.norm2(x)) # MLP计算  
 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.num\_classes = num\_classes # 类别数  
 self.embed\_dim = embed\_dim # 嵌入维度  
 self.stage1\_conv\_embed = nn.Sequential(  
 nn.Conv2d(in\_chans, embed\_dim, 7, 4, 2), # 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]) # 第一阶段的多个CSWinBlock  
 ])  
 # 后续阶段的定义略去...  
  
 def forward(self, x):  
 """前向传播"""  
 x = self.stage1\_conv\_embed(x) # 输入经过卷积和归一化  
 for blk in self.stage1:  
 x = blk(x) # 通过每个CSWinBlock  
 return x  
  
# 定义模型的实例化函数  
def CSWin\_tiny(pretrained=False, \*\*kwargs):  
 model = CSWinTransformer(embed\_dim=64, depth=[1,2,21,1], num\_heads=2, \*\*kwargs)  
 return model  
  
# 主程序入口  
if \_\_name\_\_ == '\_\_main\_\_':  
 inputs = torch.randn((1, 3, 640, 640)) # 输入数据  
 model = CSWin\_tiny() # 创建模型实例  
 res = model(inputs) # 前向传播  
 print(res.size()) # 输出结果的尺寸  
```  
  
### 代码说明：  
1. \*\*Mlp类\*\*：实现了一个多层感知机，包含两层线性变换和激活函数，使用Dropout来防止过拟合。  
2. \*\*CSWinBlock类\*\*：实现了CSWin Transformer的基本块，包含注意力机制和MLP模块。  
3. \*\*CSWinTransformer类\*\*：定义了整个CSWin Transformer模型的结构，包括输入层和多个CSWinBlock。  
4. \*\*CSWin\_tiny函数\*\*：用于创建一个小型的CSWin Transformer模型实例。  
5. \*\*主程序\*\*：用于测试模型的前向传播，输出结果的尺寸。```

这个程序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。文件中包含了多个类和函数，下面是对代码的详细说明。  
  
首先，文件引入了必要的库，包括PyTorch、timm库以及一些功能性模块。接着，定义了几个全局变量，表示不同规模的CSWin Transformer模型。  
  
接下来，定义了一个名为`Mlp`的类，它是一个多层感知机（MLP），包含两个线性层和一个激活函数（默认为GELU）。这个类的主要功能是对输入进行前向传播，通过两个线性层和激活函数的组合来实现特征变换。  
  
然后，定义了`LePEAttention`类，它实现了一种特殊的注意力机制，称为局部增强位置编码（LePE）。该类的构造函数接收多个参数，包括输入维度、分辨率、头数等。`forward`方法实现了注意力计算，包括将输入分割成窗口、计算注意力权重以及将结果重新组合成图像。  
  
接着，定义了`CSWinBlock`类，它是CSWin Transformer的基本构建块。每个块包含一个注意力层和一个MLP层，使用层归一化进行规范化。`forward`方法实现了块的前向传播，首先对输入进行归一化，然后计算注意力，最后通过MLP进行特征变换。  
  
`img2windows`和`windows2img`是两个辅助函数，用于将图像从常规格式转换为窗口格式，反之亦然。这对于注意力机制的实现至关重要，因为它允许模型在局部窗口内进行计算。  
  
`Merge\_Block`类用于在不同阶段之间合并特征，使用卷积层来减少特征图的分辨率，并进行层归一化。  
  
`CSWinTransformer`类是整个模型的核心，包含多个阶段，每个阶段由多个`CSWinBlock`组成。构造函数中定义了输入层、多个阶段和合并层，并根据给定的参数初始化模型的权重。`forward\_features`方法实现了模型的前向传播，提取特征并在不同阶段之间传递。  
  
此外，文件中还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转换。  
  
最后，文件提供了四个函数（`CSWin\_tiny`、`CSWin\_small`、`CSWin\_base`、`CSWin\_large`），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`\_\_main\_\_`部分，创建了不同规模的模型实例，并对随机输入进行前向传播，输出每个模型的特征图大小。  
  
整体而言，这个程序文件实现了一个复杂的视觉变换器模型，结合了局部增强位置编码和多层感知机，适用于各种计算机视觉任务。

### 整体功能和构架概括  
  
该项目包含多个深度学习模型文件，主要用于计算机视觉任务，如图像分类和特征提取。每个文件实现了不同类型的神经网络架构，利用现代深度学习技术（如卷积、注意力机制和变换器结构）来提高模型的性能和灵活性。整体架构上，各个模型通过模块化设计，使得用户可以根据需求选择和组合不同的模型组件。  
  
以下是每个文件的功能概述：  
  
| 文件名 | 功能描述 |  
|------------------------|----------------------------------------------------------------------------------------------|  
| `RFAConv.py` | 实现了 RFA（Receptive Field Attention）卷积模块，结合了注意力机制和卷积操作，增强特征表示能力。 |  
| `rmt.py` | 实现了视觉变换器（Vision Transformer）模型，包含多个层和模块，适用于图像处理任务。 |  
| `VanillaNet.py` | 实现了 VanillaNet 模型，使用简单的卷积和激活层，提供多种配置选项以适应不同的任务需求。 |  
| `CSwomTransformer.py` | 实现了 CSWin Transformer 模型，结合了局部增强位置编码和多层感知机，适用于图像分类等任务。 |  
  
这些文件通过各自的模型和模块，提供了多样化的选择，使得用户能够根据具体的应用场景和需求，灵活地构建和训练深度学习模型。

注意：由于此博客编辑较早，上面“10.YOLOv11核心改进源码讲解”中部分代码可能会优化升级，仅供参考学习，以“11.完整训练+Web前端界面+200+种全套创新点源码、数据集获取”的内容为准。