# 改进yolo11-DRBNCSPELAN等200+全套创新点大全：可回收金属垃圾检测系统源码＆数据集全套

## 3.背景

研究背景与意义  
  
随着全球对可持续发展和资源循环利用的关注日益增强，废弃物管理与回收利用成为了现代社会面临的重要挑战之一。金属垃圾，作为一种常见的可回收物品，其有效检测与分类对于提升资源回收率、减少环境污染具有重要意义。传统的金属垃圾检测方法往往依赖人工分拣，不仅效率低下，而且容易受到人为因素的影响，导致分类不准确。因此，基于计算机视觉技术的自动化检测系统应运而生，成为提升金属垃圾回收效率的有效解决方案。  
  
在众多计算机视觉算法中，YOLO（You Only Look Once）系列算法因其高效的实时目标检测能力而备受青睐。YOLOv11作为该系列的最新版本，进一步优化了检测精度和速度，适合于处理复杂的环境和多样化的物体。在本研究中，我们将基于改进的YOLOv11算法，构建一个专门针对可回收金属垃圾的检测系统。该系统旨在通过深度学习技术，实现对四类金属垃圾（瓶盖、硬币、可乐罐和回形针）的高效识别与分类。  
  
为支持这一目标，我们使用了一个包含2217张图像的数据集，涵盖了上述四类物品。数据集经过精心标注，并进行了多种预处理和增强，以提高模型的泛化能力和鲁棒性。通过对这些数据的深入分析与训练，我们期望不仅能够提升金属垃圾的检测精度，还能为后续的智能垃圾分类系统提供坚实的基础。  
  
综上所述，基于改进YOLOv11的可回收金属垃圾检测系统的研究，不仅有助于推动智能垃圾分类技术的发展，还将为实现资源的高效回收和环境保护贡献一份力量。这一研究的成功实施，将为可持续发展目标的实现提供重要的技术支持和实践经验。

## 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  
  
本项目旨在开发一个改进版的YOLOv11模型，以实现高效的可回收金属垃圾检测系统。为此，我们构建了一个专门的数据集，专注于物体检测领域，涵盖了四种主要类别的可回收物品。这些类别包括瓶盖（bottlecap）、硬币（coin）、可乐罐（coke）和回形针（paperclips）。每个类别在数据集中都经过精心标注，以确保模型能够准确识别和分类这些物品。  
  
数据集的构建过程涉及多种数据采集方法，包括实地拍摄和合成图像生成。我们从不同的环境和光照条件下收集了大量样本，以增强模型的鲁棒性和适应性。每个类别的样本数量均衡，确保模型在训练过程中不会偏向某一特定类别，从而提高整体检测性能。数据集中包含的图像经过预处理，确保了其质量和清晰度，以便于模型的学习和训练。  
  
在数据标注方面，我们采用了高精度的标注工具，确保每个物体的边界框准确无误。这对于YOLOv11模型的训练至关重要，因为准确的标注能够直接影响模型的检测精度和召回率。此外，数据集还包括多种场景下的图像，以模拟实际应用中的复杂情况，例如不同的背景、物体的遮挡以及各种拍摄角度。  
  
通过这一数据集的构建，我们希望为可回收金属垃圾的自动检测提供一个坚实的基础，从而推动环境保护和资源回收的相关研究。我们相信，改进后的YOLOv11模型将能够在实际应用中展现出卓越的性能，为可持续发展贡献一份力量。

## 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 os  
import torch  
from ultralytics.data import build\_dataloader, build\_yolo\_dataset, converter  
from ultralytics.engine.validator import BaseValidator  
from ultralytics.utils import LOGGER, ops  
from ultralytics.utils.metrics import ConfusionMatrix, DetMetrics, box\_iou  
  
class DetectionValidator(BaseValidator):  
 """  
 该类扩展了BaseValidator类，用于基于检测模型的验证。  
 """  
  
 def \_\_init\_\_(self, dataloader=None, save\_dir=None, pbar=None, args=None, \_callbacks=None):  
 """初始化检测模型所需的变量和设置。"""  
 super().\_\_init\_\_(dataloader, save\_dir, pbar, args, \_callbacks)  
 self.metrics = DetMetrics(save\_dir=self.save\_dir, on\_plot=self.on\_plot) # 初始化检测指标  
 self.iouv = torch.linspace(0.5, 0.95, 10) # 定义mAP@0.5:0.95的IOU向量  
 self.niou = self.iouv.numel() # IOU的数量  
  
 def preprocess(self, batch):  
 """对YOLO训练的图像批次进行预处理。"""  
 batch["img"] = batch["img"].to(self.device, non\_blocking=True) # 将图像移动到设备上  
 batch["img"] = (batch["img"].half() if self.args.half else batch["img"].float()) / 255 # 归一化图像  
 for k in ["batch\_idx", "cls", "bboxes"]:  
 batch[k] = batch[k].to(self.device) # 将其他数据移动到设备上  
 return batch  
  
 def init\_metrics(self, model):  
 """初始化YOLO的评估指标。"""  
 self.names = model.names # 获取模型的类别名称  
 self.nc = len(model.names) # 类别数量  
 self.metrics.names = self.names # 设置指标的类别名称  
 self.confusion\_matrix = ConfusionMatrix(nc=self.nc) # 初始化混淆矩阵  
  
 def postprocess(self, preds):  
 """对预测输出应用非极大值抑制。"""  
 return ops.non\_max\_suppression(  
 preds,  
 self.args.conf,  
 self.args.iou,  
 multi\_label=True,  
 agnostic=self.args.single\_cls,  
 max\_det=self.args.max\_det,  
 )  
  
 def update\_metrics(self, preds, batch):  
 """更新指标。"""  
 for si, pred in enumerate(preds):  
 npr = len(pred) # 当前预测的数量  
 pbatch = self.\_prepare\_batch(si, batch) # 准备当前批次的数据  
 cls, bbox = pbatch.pop("cls"), pbatch.pop("bbox") # 获取真实标签  
 if npr == 0:  
 continue # 如果没有预测，跳过  
 predn = self.\_prepare\_pred(pred, pbatch) # 准备预测数据  
 # 计算TP（真正例）  
 stat["tp"] = self.\_process\_batch(predn, bbox, cls)   
 # 更新混淆矩阵  
 self.confusion\_matrix.process\_batch(predn, bbox, cls)  
  
 def finalize\_metrics(self, \*args, \*\*kwargs):  
 """设置指标速度和混淆矩阵的最终值。"""  
 self.metrics.speed = self.speed  
 self.metrics.confusion\_matrix = self.confusion\_matrix  
  
 def get\_stats(self):  
 """返回指标统计和结果字典。"""  
 stats = {k: torch.cat(v, 0).cpu().numpy() for k, v in self.stats.items()} # 转换为numpy  
 if len(stats) and stats["tp"].any():  
 self.metrics.process(\*\*stats) # 处理指标  
 return self.metrics.results\_dict # 返回结果字典  
  
 def print\_results(self):  
 """打印每个类别的训练/验证集指标。"""  
 pf = "%22s" + "%11i" \* 2 + "%11.3g" \* len(self.metrics.keys) # 打印格式  
 LOGGER.info(pf % ("all", self.seen, self.nt\_per\_class.sum(), \*self.metrics.mean\_results()))  
  
 def \_process\_batch(self, detections, gt\_bboxes, gt\_cls):  
 """  
 返回正确的预测矩阵。  
 """  
 iou = box\_iou(gt\_bboxes, detections[:, :4]) # 计算IOU  
 return self.match\_predictions(detections[:, 5], gt\_cls, iou) # 匹配预测与真实标签  
  
 def build\_dataset(self, img\_path, mode="val", batch=None):  
 """  
 构建YOLO数据集。  
 """  
 return build\_yolo\_dataset(self.args, img\_path, batch, self.data, mode=mode)  
  
 def get\_dataloader(self, dataset\_path, batch\_size):  
 """构造并返回数据加载器。"""  
 dataset = self.build\_dataset(dataset\_path, batch=batch\_size, mode="val")  
 return build\_dataloader(dataset, batch\_size, self.args.workers, shuffle=False) # 返回数据加载器  
```  
  
### 核心部分分析  
1. \*\*类定义\*\*：`DetectionValidator`类继承自`BaseValidator`，用于YOLO模型的验证。  
2. \*\*初始化方法\*\*：`\_\_init\_\_`方法中初始化了一些必要的变量和设置，包括指标、IOU向量等。  
3. \*\*预处理\*\*：`preprocess`方法对输入的图像批次进行归一化和设备转移。  
4. \*\*指标初始化\*\*：`init\_metrics`方法初始化评估指标和混淆矩阵。  
5. \*\*后处理\*\*：`postprocess`方法应用非极大值抑制来过滤预测结果。  
6. \*\*更新指标\*\*：`update\_metrics`方法更新当前批次的指标，包括计算TP和更新混淆矩阵。  
7. \*\*最终指标\*\*：`finalize\_metrics`方法设置最终的指标速度和混淆矩阵。  
8. \*\*获取统计信息\*\*：`get\_stats`方法返回指标统计信息。  
9. \*\*打印结果\*\*：`print\_results`方法打印每个类别的验证结果。  
10. \*\*数据集构建\*\*：`build\_dataset`和`get\_dataloader`方法用于构建YOLO数据集和数据加载器。  
  
这些核心部分是YOLO模型验证的基础，涵盖了从数据预处理到指标计算和结果输出的整个流程。```

这个程序文件 `val.py` 是一个用于验证目标检测模型（如 YOLO）的类和函数的实现，主要是为了评估模型在验证集上的性能。文件中包含了许多与数据处理、模型评估和结果可视化相关的功能。  
  
首先，文件导入了一些必要的库和模块，包括 `os`、`numpy` 和 `torch`，以及一些来自 `ultralytics` 库的工具和类，这些工具和类用于构建数据加载器、处理数据集、计算指标等。  
  
`DetectionValidator` 类继承自 `BaseValidator`，它的主要职责是进行目标检测模型的验证。构造函数中初始化了一些必要的变量和设置，包括处理设备、任务类型、评估指标等。特别地，它定义了一个 `iouv` 向量，用于计算不同 IoU（Intersection over Union）阈值下的 mAP（mean Average Precision）。  
  
在 `preprocess` 方法中，输入的图像批次会被处理，包括将图像转换为合适的格式和范围（如归一化），并将目标框（bounding boxes）进行缩放，以适应当前图像的尺寸。  
  
`init\_metrics` 方法用于初始化评估指标，包括判断数据集是否为 COCO 格式，并设置相应的类别映射和指标对象。`get\_desc` 方法返回一个格式化的字符串，用于描述每个类别的评估指标。  
  
`postprocess` 方法应用非极大值抑制（NMS）来处理模型的预测输出，以减少冗余的检测框。`\_prepare\_batch` 和 `\_prepare\_pred` 方法则用于准备验证过程中所需的批次数据和预测数据。  
  
在 `update\_metrics` 方法中，模型的预测结果与真实标签进行比较，更新各种统计信息，包括正确预测的数量、置信度等。此方法还负责保存预测结果到 JSON 或 TXT 文件中。  
  
`finalize\_metrics` 方法用于设置最终的指标值，而 `get\_stats` 方法则返回统计信息和结果字典。`print\_results` 方法打印出每个类别的训练或验证集指标，并在需要时绘制混淆矩阵。  
  
此外，文件中还包含了一些用于构建数据集和数据加载器的方法，如 `build\_dataset` 和 `get\_dataloader`，这些方法帮助加载验证集的图像和标签。  
  
最后，文件提供了一些可视化功能，如 `plot\_val\_samples` 和 `plot\_predictions`，用于绘制验证图像和模型的预测结果。`save\_one\_txt` 和 `pred\_to\_json` 方法则负责将预测结果保存为特定格式的文件，以便后续分析和评估。  
  
整体来看，这个文件是一个完整的目标检测模型验证框架，涵盖了从数据预处理到结果评估和可视化的各个方面。

```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  
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 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 # 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  
 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() # 调整维度  
 relative\_coords[:, :, 0] += self.window\_size[0] - 1 # 调整坐标范围  
 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) # 线性变换用于生成Q、K、V  
 self.attn\_drop = nn.Dropout(attn\_drop) # 注意力的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)) # 计算注意力得分  
  
 # 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() # 调整维度  
 attn = attn + relative\_position\_bias.unsqueeze(0) # 加入相对位置偏置  
  
 if mask is not None:  
 attn = attn + mask.unsqueeze(1).unsqueeze(0) # 加入mask  
 attn = self.softmax(attn) # Softmax归一化  
 attn = self.attn\_drop(attn) # 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) # Dropout  
 return x  
  
  
class SwinTransformer(nn.Module):  
 """ Swin Transformer 主体 """  
  
 def \_\_init\_\_(self, embed\_dim=96, depths=[2, 2, 6, 2], num\_heads=[3, 6, 12, 24], window\_size=7):  
 super().\_\_init\_\_()  
 self.embed\_dim = embed\_dim # 嵌入维度  
 self.layers = nn.ModuleList() # 存储各层  
  
 # 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=window\_size)  
 self.layers.append(layer)  
  
 def forward(self, x):  
 """ 前向传播 """  
 for layer in self.layers:  
 x = layer(x) # 逐层传递  
 return x # 返回最终输出  
```  
  
### 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。  
2. \*\*WindowAttention类\*\*：实现了窗口注意力机制，支持相对位置偏置。  
3. \*\*SwinTransformer类\*\*：实现了Swin Transformer的主体结构，包含多个基本层（BasicLayer），每层使用窗口注意力机制进行特征提取。  
  
该代码片段是Swin Transformer模型的核心部分，负责实现特征提取和注意力机制。```

这个程序文件实现了Swin Transformer模型的结构，Swin Transformer是一种基于视觉的Transformer架构，具有分层和移动窗口的特性。代码中定义了多个类和函数，以实现模型的各个组成部分。  
  
首先，程序导入了必要的库，包括PyTorch的核心模块和一些用于构建模型的辅助函数。接着，定义了一个名为`Mlp`的类，这是一个多层感知机（MLP），用于在Transformer的每个块中进行前馈神经网络的计算。`Mlp`类的构造函数接受输入特征、隐藏特征和输出特征的维度，以及激活函数和丢弃率。其`forward`方法实现了前向传播。  
  
接下来，定义了`window\_partition`和`window\_reverse`函数，这两个函数用于将输入特征分割成窗口以及将窗口合并回原始特征。窗口的概念是Swin Transformer的核心之一，它允许模型在局部区域内进行自注意力计算。  
  
`WindowAttention`类实现了基于窗口的多头自注意力机制，支持相对位置偏置。它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，并初始化了一些必要的权重。`forward`方法实现了自注意力的计算过程，包括查询、键、值的计算，以及相对位置偏置的应用。  
  
`SwinTransformerBlock`类则是Swin Transformer的基本构建块，包含了归一化层、窗口注意力层和前馈网络。它的`forward`方法实现了输入特征的处理，包括规范化、窗口划分、注意力计算和前馈网络的应用。  
  
`PatchMerging`类用于将特征图的补丁合并，以减少特征图的分辨率。它的`forward`方法实现了特征的重组和归一化。  
  
`BasicLayer`类表示Swin Transformer中的一个基本层，包含多个`SwinTransformerBlock`。它的构造函数接受层的深度、头数、窗口大小等参数，并构建出多个块。`forward`方法计算输入特征的处理，并在必要时进行下采样。  
  
`PatchEmbed`类用于将输入图像分割成补丁并进行嵌入。它的`forward`方法实现了图像的填充、卷积操作和归一化。  
  
`SwinTransformer`类是整个模型的主类，负责将所有的组件组合在一起。它的构造函数定义了输入图像的大小、补丁大小、嵌入维度、每个阶段的深度和头数等参数，并构建出各个层。`forward`方法实现了模型的前向传播过程，包括补丁嵌入、位置编码的添加、各层的处理以及输出的归一化。  
  
最后，`update\_weight`函数用于更新模型的权重，确保加载的权重与模型的结构匹配。`SwinTransformer\_Tiny`函数则是一个工厂函数，用于创建一个小型的Swin Transformer模型，并在提供权重文件时加载相应的权重。  
  
总体而言，这个程序文件实现了Swin Transformer的完整结构，涵盖了模型的各个部分，包括嵌入、注意力机制、前馈网络和层的组合，适用于图像处理任务。

```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  
  
 # 用于获取权重的卷积模块  
 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)  
 )  
   
 # 用于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] # 获取输入的批量大小和通道数  
 weight = self.get\_weight(x) # 获取权重  
 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  
   
 # 调整形状以进行卷积  
 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] # 获取输入的批量大小和通道数  
 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) # 通过全连接层并调整形状  
 return y # 返回通道注意力权重  
  
  
class RFCBAMConv(nn.Module):  
 def \_\_init\_\_(self, in\_channel, out\_channel, kernel\_size=3, stride=1):  
 super().\_\_init\_\_()  
 self.kernel\_size = kernel\_size  
   
 # 特征生成模块  
 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()  
 )  
   
 # 权重获取模块  
 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] # 获取输入的批量大小和通道数  
 channel\_attention = self.se(x) # 获取通道注意力  
 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  
   
 # 调整形状以进行卷积  
 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)  
   
 # 获取感受野注意力  
 receptive\_field\_attention = self.get\_weight(torch.cat((max\_feature, mean\_feature), dim=1))  
   
 # 返回卷积结果  
 conv\_data = unfold\_feature \* receptive\_field\_attention  
 return self.conv(conv\_data)  
```  
  
### 代码说明：  
1. \*\*RFAConv\*\*: 该类实现了一种卷积层，结合了特征生成和权重计算，通过自适应的方式来增强特征表示。  
2. \*\*SE\*\*: 该类实现了Squeeze-and-Excitation模块，用于生成通道注意力权重，增强网络对重要特征的关注。  
3. \*\*RFCBAMConv\*\*: 该类结合了RFAConv和SE模块，进一步增强了特征提取能力，并引入了感受野注意力机制。  
  
这些模块可以用于构建更复杂的神经网络架构，提升图像处理任务的性能。```

这个程序文件 `RFAConv.py` 定义了一些用于卷积神经网络的模块，主要包括 `RFAConv`、`RFCBAMConv` 和 `RFCAConv` 三个类，以及一些辅助的激活函数和注意力机制。下面是对代码的详细说明。  
  
首先，文件导入了必要的库，包括 PyTorch 的核心模块和一些自定义的卷积模块。接着，定义了两个激活函数类 `h\_sigmoid` 和 `h\_swish`，这两个类分别实现了 h-sigmoid 和 h-swish 激活函数，通常用于提高网络的非线性表达能力。  
  
`RFAConv` 类是一个自定义的卷积层，它通过加权的方式生成特征图。初始化时，它接受输入通道数、输出通道数、卷积核大小和步幅作为参数。该类内部定义了两个主要的子模块：`get\_weight` 和 `generate\_feature`。`get\_weight` 通过平均池化和卷积操作生成权重，而 `generate\_feature` 则通过卷积、批归一化和 ReLU 激活生成特征。前向传播中，首先计算权重，然后生成特征，接着将特征与权重相乘，最后通过重排和卷积得到输出。  
  
`SE` 类实现了 Squeeze-and-Excitation (SE) 模块，用于增强特征的表达能力。它通过全局平均池化和全连接层来生成通道注意力权重，进而调整输入特征的通道信息。  
  
`RFCBAMConv` 类是一个结合了通道注意力和空间注意力的卷积模块。它在初始化时定义了生成特征的卷积层、获取权重的卷积层以及 SE 模块。在前向传播中，首先计算通道注意力，然后生成特征并重排，接着计算最大值和均值特征以生成空间注意力权重，最后将特征与注意力权重相乘并通过卷积得到输出。  
  
`RFCAConv` 类则实现了一种更复杂的卷积模块，结合了空间和通道注意力机制。它在初始化时定义了生成特征的卷积层、用于计算通道注意力的卷积层以及最终的卷积层。在前向传播中，生成特征后，通过自适应平均池化计算水平和垂直方向的特征，然后将它们拼接并通过一系列卷积和激活操作生成通道注意力，最后将特征与注意力权重相乘并通过卷积得到最终输出。  
  
总体而言，这个文件实现了一些先进的卷积操作和注意力机制，旨在提高卷积神经网络的性能，尤其是在处理图像等高维数据时。

```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.functional as F  
  
def multi\_scale\_deformable\_attn\_pytorch(  
 value: torch.Tensor,  
 value\_spatial\_shapes: torch.Tensor,  
 sampling\_locations: torch.Tensor,  
 attention\_weights: torch.Tensor,  
) -> torch.Tensor:  
 """  
 多尺度可变形注意力机制。  
  
 参数:  
 value: 输入特征图，形状为 (bs, C, num\_heads, embed\_dims)  
 value\_spatial\_shapes: 特征图的空间形状，形状为 (num\_levels, 2)  
 sampling\_locations: 采样位置，形状为 (bs, num\_queries, num\_heads, num\_levels, num\_points, 2)  
 attention\_weights: 注意力权重，形状为 (bs, num\_heads, num\_queries, num\_levels, num\_points)  
  
 返回:  
 output: 经过多尺度可变形注意力机制处理后的输出，形状为 (bs, num\_queries, num\_heads \* embed\_dims)  
 """  
 bs, \_, num\_heads, embed\_dims = value.shape # 获取输入特征图的批次大小、通道数、头数和嵌入维度  
 \_, num\_queries, \_, num\_levels, num\_points, \_ = sampling\_locations.shape # 获取采样位置的相关维度  
  
 # 将输入特征图按照空间形状拆分成多个特征图  
 value\_list = value.split([H\_ \* W\_ for H\_, W\_ in value\_spatial\_shapes], dim=1)  
   
 # 将采样位置转换到[-1, 1]范围  
 sampling\_grids = 2 \* sampling\_locations - 1  
 sampling\_value\_list = [] # 用于存储每个尺度的采样值  
  
 for level, (H\_, W\_) in enumerate(value\_spatial\_shapes):  
 # 对于每个尺度，调整特征图的形状以便进行采样  
 value\_l\_ = value\_list[level].flatten(2).transpose(1, 2).reshape(bs \* num\_heads, embed\_dims, H\_, W\_)  
   
 # 调整采样网格的形状  
 sampling\_grid\_l\_ = sampling\_grids[:, :, :, level].transpose(1, 2).flatten(0, 1)  
   
 # 使用grid\_sample进行双线性插值采样  
 sampling\_value\_l\_ = F.grid\_sample(  
 value\_l\_, sampling\_grid\_l\_, mode="bilinear", padding\_mode="zeros", align\_corners=False  
 )  
 sampling\_value\_list.append(sampling\_value\_l\_) # 将采样结果添加到列表中  
  
 # 调整注意力权重的形状以便进行加权求和  
 attention\_weights = attention\_weights.transpose(1, 2).reshape(  
 bs \* num\_heads, 1, num\_queries, num\_levels \* num\_points  
 )  
   
 # 计算最终输出，进行加权求和  
 output = (  
 (torch.stack(sampling\_value\_list, dim=-2).flatten(-2) \* attention\_weights)  
 .sum(-1)  
 .view(bs, num\_heads \* embed\_dims, num\_queries)  
 )  
   
 return output.transpose(1, 2).contiguous() # 返回输出，调整形状以符合预期  
```  
  
### 代码说明：  
1. \*\*函数定义\*\*：`multi\_scale\_deformable\_attn\_pytorch`函数实现了多尺度可变形注意力机制。  
2. \*\*参数说明\*\*：  
 - `value`：输入特征图，包含多个头的特征信息。  
 - `value\_spatial\_shapes`：特征图的空间形状，用于确定每个尺度的高和宽。  
 - `sampling\_locations`：指定在特征图上进行采样的位置。  
 - `attention\_weights`：每个查询对应的注意力权重。  
3. \*\*主要步骤\*\*：  
 - 拆分输入特征图为多个尺度。  
 - 将采样位置转换到[-1, 1]范围。  
 - 对每个尺度的特征图进行双线性插值采样。  
 - 调整注意力权重的形状并进行加权求和，最终得到输出。```

这个程序文件 `utils.py` 是一个用于实现一些常用功能的模块，主要涉及深度学习中的模型初始化和多尺度可变形注意力机制。文件中使用了 PyTorch 库，包含了一些重要的函数和工具。  
  
首先，文件引入了必要的库，包括 `copy`、`math`、`numpy` 和 `torch` 及其子模块。接着，定义了一个 `\_\_all\_\_` 列表，指定了该模块公开的接口，方便其他模块导入。  
  
文件中定义了一个 `\_get\_clones` 函数，该函数用于克隆给定的模块 `n` 次，并返回一个 `ModuleList`，这在构建具有多个相同层的网络时非常有用。  
  
接下来是 `bias\_init\_with\_prob` 函数，它根据给定的先验概率初始化卷积或全连接层的偏置值。这个函数使用了对数几率的公式，将概率转换为偏置值，以便在训练过程中更好地控制模型的输出。  
  
`linear\_init` 函数用于初始化线性模块的权重和偏置。它根据权重的形状计算一个边界值，并使用均匀分布在该范围内初始化权重和偏置。  
  
`inverse\_sigmoid` 函数计算输入张量的反 sigmoid 函数。它首先将输入限制在 [0, 1] 范围内，然后计算反 sigmoid 值。这个函数在一些模型中用于处理概率值的转换。  
  
最后，`multi\_scale\_deformable\_attn\_pytorch` 函数实现了多尺度可变形注意力机制。该函数接收多个输入，包括值张量、空间形状、采样位置和注意力权重。函数内部首先对输入的值进行分割，然后计算采样网格，并使用 `F.grid\_sample` 函数进行双线性插值，得到采样值。最后，通过加权求和的方式将采样值与注意力权重结合，输出最终的结果。  
  
整体来看，这个模块提供了一些基础的工具函数，特别是在构建和初始化深度学习模型时非常实用，同时实现了一个复杂的多尺度注意力机制，为后续的模型训练和推理提供了支持。

### 程序整体功能和构架概括  
  
该程序是一个深度学习框架，主要用于实现和验证目标检测模型，特别是基于Swin Transformer架构的模型。整体结构由多个模块组成，每个模块负责特定的功能，包括模型的构建、卷积操作、注意力机制、验证过程和实用工具函数。这种模块化设计使得代码易于维护和扩展，便于在不同的任务中复用。  
  
- \*\*val.py\*\*: 负责模型的验证过程，包括数据预处理、评估指标计算和结果可视化。  
- \*\*SwinTransformer.py\*\*: 实现了Swin Transformer模型的结构，包括各个组成部分，如嵌入、注意力机制和前馈网络。  
- \*\*RFAConv.py\*\*: 定义了一些自定义卷积层和注意力机制，以增强模型的特征提取能力。  
- \*\*utils.py\*\*: 提供了一些常用的工具函数，包括模型初始化、激活函数和多尺度可变形注意力机制的实现。  
  
### 文件功能整理表  
  
| 文件名 | 功能描述 |  
|---------------------|------------------------------------------------------------------------------------------|  
| `val.py` | 实现目标检测模型的验证过程，包括数据预处理、评估指标计算、结果可视化和保存预测结果。 |  
| `SwinTransformer.py`| 实现Swin Transformer模型的结构，包括补丁嵌入、窗口注意力机制、前馈网络和层的组合。 |  
| `RFAConv.py` | 定义自定义卷积层和注意力机制，包括 RFAConv、RFCBAMConv 和 RFCAConv，以增强特征提取能力。 |  
| `utils.py` | 提供工具函数，包括模型权重初始化、激活函数、反 sigmoid 函数和多尺度可变形注意力机制的实现。 |  
  
这种结构和功能划分使得整个程序在处理目标检测任务时更加高效和灵活，能够根据需求进行扩展和修改。