# 改进yolo11-RevCol等200+全套创新点大全：可回收垃圾塑料检测图像分割系统源码＆数据集全套

## 3.背景

研究背景与意义  
  
随着全球塑料污染问题的日益严重，塑料垃圾的回收与处理已成为环境保护的重要议题。塑料作为一种广泛使用的材料，其在生活中的便利性与经济性使得其使用量不断增加，但随之而来的却是大量的塑料废弃物对生态环境的威胁。因此，开发高效的垃圾分类与回收系统显得尤为重要。尤其是在城市化进程加快的背景下，如何快速、准确地识别和分类可回收垃圾，尤其是塑料垃圾，成为了亟待解决的技术难题。  
  
本研究旨在基于改进的YOLOv11模型，构建一个高效的可回收垃圾塑料检测图像分割系统。该系统将通过实例分割技术，精确识别和分离图像中的塑料垃圾与非塑料垃圾，进而提高垃圾分类的自动化水平。为实现这一目标，我们采用了一个包含3000张图像的数据集，该数据集专门标注了两类物体：可回收的“Trash plastic”和不可回收的“Non-plastic”。通过对这些图像的分析与处理，系统能够在实际应用中提供高精度的分类结果。  
  
此外，改进的YOLOv11模型在实时性和准确性方面的优势，使其成为处理大规模图像数据的理想选择。通过引入深度学习技术，尤其是卷积神经网络（CNN），我们可以有效地提取图像特征，从而实现对塑料垃圾的快速识别与分割。这不仅有助于提高垃圾分类的效率，还能为后续的回收处理提供数据支持，推动可持续发展目标的实现。  
  
综上所述，本研究不仅具有重要的学术价值，也为实际的环境保护工作提供了技术支持，期望通过智能化的垃圾分类系统，促进塑料垃圾的有效回收与利用，减轻塑料污染对生态环境的影响。

## 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  
  
本项目所使用的数据集专注于可回收垃圾塑料的检测与图像分割，旨在改进YOLOv11模型的性能，以实现更高效的垃圾分类和回收。数据集的主题为“Instance Segmentation Trash Plastic”，其中包含两大主要类别：非塑料（Non-plastic）和垃圾塑料（Trash plastic）。这两类的划分不仅反映了实际垃圾分类的需求，也为模型训练提供了清晰的目标，帮助算法在复杂的环境中准确识别和分割不同类型的垃圾。  
  
数据集的构建经过精心设计，确保了图像的多样性和代表性。所收集的图像涵盖了不同场景、光照条件和背景，旨在模拟实际垃圾投放环境中的各种情况。这种多样性不仅提高了模型的泛化能力，还增强了其在实际应用中的实用性。此外，数据集中每个类别的样本数量经过合理配置，以确保模型在训练过程中能够充分学习到每一类的特征。  
  
在数据标注方面，采用了精确的实例分割技术，为每个垃圾塑料实例提供了细致的边界框和掩膜。这种标注方式使得模型能够在像素级别上进行学习，从而提升了对复杂场景中塑料垃圾的识别精度。通过这种方式，模型不仅能够识别出垃圾的存在，还能准确地分割出每个塑料实例，为后续的处理和回收提供了重要的数据支持。  
  
总之，本项目的数据集为改进YOLOv11的可回收垃圾塑料检测图像分割系统提供了坚实的基础，旨在通过先进的图像处理技术，推动垃圾分类的智能化和自动化进程，为环境保护和资源回收做出积极贡献。

## 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  
  
# 定义 MobileNetV4 的不同模型规格  
MODEL\_SPECS = {  
 "MobileNetV4ConvSmall": MNV4ConvSmall\_BLOCK\_SPECS,  
 "MobileNetV4ConvMedium": MNV4ConvMedium\_BLOCK\_SPECS,  
 "MobileNetV4ConvLarge": MNV4ConvLarge\_BLOCK\_SPECS,  
 "MobileNetV4HybridMedium": MNV4HybridConvMedium\_BLOCK\_SPECS,  
 "MobileNetV4HybridLarge": MNV4HybridConvLarge\_BLOCK\_SPECS,  
}  
  
def conv\_2d(inp, oup, kernel\_size=3, stride=1, groups=1, bias=False, norm=True, act=True):  
 """  
 创建一个2D卷积层，包含卷积、批归一化和激活函数（ReLU6）。  
  
 Args:  
 inp: 输入通道数  
 oup: 输出通道数  
 kernel\_size: 卷积核大小  
 stride: 步幅  
 groups: 分组卷积的组数  
 bias: 是否使用偏置  
 norm: 是否使用批归一化  
 act: 是否使用激活函数  
  
 Returns:  
 nn.Sequential: 包含卷积层、批归一化和激活函数的序列  
 """  
 conv = nn.Sequential()  
 padding = (kernel\_size - 1) // 2 # 计算填充  
 conv.add\_module('conv', nn.Conv2d(inp, oup, kernel\_size, stride, padding, bias=bias, groups=groups))  
 if norm:  
 conv.add\_module('BatchNorm2d', nn.BatchNorm2d(oup)) # 添加批归一化  
 if act:  
 conv.add\_module('Activation', nn.ReLU6()) # 添加激活函数  
 return conv  
  
class InvertedResidual(nn.Module):  
 """  
 反向残差块，包含扩展卷积、深度卷积和投影卷积。  
 """  
 def \_\_init\_\_(self, inp, oup, stride, expand\_ratio, act=False):  
 super(InvertedResidual, self).\_\_init\_\_()  
 self.stride = stride  
 assert stride in [1, 2] # 步幅只能是1或2  
 hidden\_dim = int(round(inp \* expand\_ratio)) # 计算隐藏层维度  
 self.block = nn.Sequential()  
 if expand\_ratio != 1:  
 self.block.add\_module('exp\_1x1', conv\_2d(inp, hidden\_dim, kernel\_size=1, stride=1)) # 扩展卷积  
 self.block.add\_module('conv\_3x3', conv\_2d(hidden\_dim, hidden\_dim, kernel\_size=3, stride=stride, groups=hidden\_dim)) # 深度卷积  
 self.block.add\_module('red\_1x1', conv\_2d(hidden\_dim, oup, kernel\_size=1, stride=1, act=act)) # 投影卷积  
 self.use\_res\_connect = self.stride == 1 and inp == oup # 判断是否使用残差连接  
  
 def forward(self, x):  
 if self.use\_res\_connect:  
 return x + self.block(x) # 使用残差连接  
 else:  
 return self.block(x)  
  
class MobileNetV4(nn.Module):  
 """  
 MobileNetV4 模型，支持不同规格的网络结构。  
 """  
 def \_\_init\_\_(self, model):  
 super().\_\_init\_\_()  
 assert model in MODEL\_SPECS.keys() # 确保模型名称有效  
 self.model = model  
 self.spec = MODEL\_SPECS[self.model] # 获取模型规格  
  
 # 构建网络的各个层  
 self.conv0 = build\_blocks(self.spec['conv0'])  
 self.layer1 = build\_blocks(self.spec['layer1'])  
 self.layer2 = build\_blocks(self.spec['layer2'])  
 self.layer3 = build\_blocks(self.spec['layer3'])  
 self.layer4 = build\_blocks(self.spec['layer4'])  
 self.layer5 = build\_blocks(self.spec['layer5'])  
 self.features = nn.ModuleList([self.conv0, self.layer1, self.layer2, self.layer3, self.layer4, self.layer5]) # 将所有层放入一个模块列表  
  
 def forward(self, x):  
 features = [None, None, None, None] # 存储特征图  
 for f in self.features:  
 x = f(x) # 通过每一层  
 # 根据输入大小记录特征图  
 if x.size(2) in [x.size(2) // 4, x.size(2) // 8, x.size(2) // 16, x.size(2) // 32]:  
 features.append(x)  
 return features # 返回特征图  
  
# 各种 MobileNetV4 模型的构造函数  
def MobileNetV4ConvSmall():  
 return MobileNetV4('MobileNetV4ConvSmall')  
  
def MobileNetV4ConvMedium():  
 return MobileNetV4('MobileNetV4ConvMedium')  
  
def MobileNetV4ConvLarge():  
 return MobileNetV4('MobileNetV4ConvLarge')  
  
def MobileNetV4HybridMedium():  
 return MobileNetV4('MobileNetV4HybridMedium')  
  
def MobileNetV4HybridLarge():  
 return MobileNetV4('MobileNetV4HybridLarge')  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 model = MobileNetV4ConvSmall() # 实例化 MobileNetV4ConvSmall 模型  
 inputs = torch.randn((1, 3, 640, 640)) # 创建一个随机输入  
 res = model(inputs) # 前向传播  
 for i in res:  
 print(i.size()) # 打印输出特征图的尺寸  
```  
  
### 代码核心部分分析  
1. \*\*卷积层构建\*\* (`conv\_2d`): 该函数创建一个包含卷积、批归一化和激活函数的序列，适用于构建网络的基本单元。  
   
2. \*\*反向残差块\*\* (`InvertedResidual`): 这是 MobileNetV4 的核心模块，包含扩展卷积、深度卷积和投影卷积，支持残差连接。  
  
3. \*\*MobileNetV4 类\*\*: 该类构建整个 MobileNetV4 网络，包含多个层的构建和前向传播逻辑。  
  
4. \*\*模型构造函数\*\*: 提供了不同规格的 MobileNetV4 模型的构造函数，便于用户根据需求创建模型实例。  
  
通过以上分析，可以看出代码的核心在于卷积层的构建和反向残差块的实现，这些是构成 MobileNetV4 网络的基础。```

这个文件定义了一个名为 MobileNetV4 的深度学习模型，主要用于计算机视觉任务。MobileNetV4 是一种轻量级的卷积神经网络，适合在资源受限的设备上运行。代码中包含了多个模块和功能，下面是对其主要部分的详细说明。  
  
首先，文件导入了必要的库，包括 `torch` 和 `torch.nn`，这些是构建和训练深度学习模型的基础库。接着，定义了一些模型的结构参数，包括不同版本的 MobileNetV4 的卷积层配置（如 `MNV4ConvSmall\_BLOCK\_SPECS`、`MNV4ConvMedium\_BLOCK\_SPECS` 等）。这些配置以字典的形式存储，描述了每一层的名称、块的数量以及具体的参数设置（如输入输出通道数、卷积核大小、步幅等）。  
  
`make\_divisible` 函数用于确保每一层的通道数都是 8 的倍数，这是为了提高模型的兼容性和性能。该函数接受一个值、一个除数以及其他可选参数，返回一个调整后的整数值。  
  
`conv\_2d` 函数是一个辅助函数，用于创建一个包含卷积层、批归一化层和激活函数的序列。这个函数简化了卷积层的构建过程。  
  
`InvertedResidual` 类实现了倒残差块，这是 MobileNetV4 的核心构建块之一。该类的构造函数定义了三个主要的卷积层：扩展卷积、深度卷积和投影卷积。`forward` 方法则实现了前向传播的逻辑，支持残差连接。  
  
`UniversalInvertedBottleneckBlock` 类是另一个重要的构建块，支持不同的卷积核大小和下采样选项。它的构造函数中定义了起始深度卷积、扩展卷积和投影卷积，并在 `forward` 方法中实现了前向传播。  
  
`build\_blocks` 函数根据给定的层规格构建相应的层。它根据块的名称（如 `convbn`、`uib` 或 `fused\_ib`）选择相应的构建逻辑，并返回一个包含所有构建层的序列。  
  
`MobileNetV4` 类是整个模型的主体，构造函数中根据指定的模型类型（如 `MobileNetV4ConvSmall`）构建相应的层。它将所有层组合成一个模块列表，并在 `forward` 方法中实现了前向传播逻辑，返回特定尺度的特征图。  
  
最后，文件提供了几个函数用于实例化不同版本的 MobileNetV4 模型（如 `MobileNetV4ConvSmall`、`MobileNetV4ConvMedium` 等）。在主程序中，创建了一个 `MobileNetV4ConvSmall` 模型，并通过随机输入进行测试，输出每个特征图的尺寸。  
  
总体来说，这个文件实现了 MobileNetV4 模型的结构，提供了灵活的构建方式和高效的前向传播机制，适合在多种应用场景中使用。

```以下是经过简化和注释的核心代码部分：  
  
```python  
import torch  
import torch.nn as nn  
from functools import lru\_cache  
  
class KAGNConvNDLayer(nn.Module):  
 def \_\_init\_\_(self, conv\_class, norm\_class, conv\_w\_fun, input\_dim, output\_dim, degree, kernel\_size,  
 groups=1, padding=0, stride=1, dilation=1, dropout: float = 0.0, ndim: int = 2):  
 super(KAGNConvNDLayer, self).\_\_init\_\_()  
   
 # 初始化参数  
 self.inputdim = input\_dim # 输入维度  
 self.outdim = output\_dim # 输出维度  
 self.degree = degree # 多项式的度数  
 self.kernel\_size = kernel\_size # 卷积核大小  
 self.padding = padding # 填充  
 self.stride = stride # 步幅  
 self.dilation = dilation # 膨胀  
 self.groups = groups # 分组卷积的组数  
 self.base\_activation = nn.SiLU() # 基础激活函数  
 self.conv\_w\_fun = conv\_w\_fun # 卷积权重函数  
 self.ndim = ndim # 数据的维度  
 self.dropout = nn.Dropout2d(p=dropout) if dropout > 0 else None # Dropout层  
   
 # 检查参数有效性  
 if groups <= 0:  
 raise ValueError('groups must be a positive integer')  
 if input\_dim % groups != 0:  
 raise ValueError('input\_dim must be divisible by groups')  
 if output\_dim % groups != 0:  
 raise ValueError('output\_dim must be divisible by groups')  
  
 # 创建基础卷积层和归一化层  
 self.base\_conv = nn.ModuleList([conv\_class(input\_dim // groups,  
 output\_dim // groups,  
 kernel\_size,  
 stride,  
 padding,  
 dilation,  
 groups=1,  
 bias=False) for \_ in range(groups)])  
  
 self.layer\_norm = nn.ModuleList([norm\_class(output\_dim // groups) for \_ in range(groups)])  
  
 # 多项式权重初始化  
 poly\_shape = (groups, output\_dim // groups, (input\_dim // groups) \* (degree + 1)) + tuple(  
 kernel\_size for \_ in range(ndim))  
 self.poly\_weights = nn.Parameter(torch.randn(\*poly\_shape)) # 多项式权重  
 self.beta\_weights = nn.Parameter(torch.zeros(degree + 1, dtype=torch.float32)) # beta权重  
  
 # 权重初始化  
 for conv\_layer in self.base\_conv:  
 nn.init.kaiming\_uniform\_(conv\_layer.weight, nonlinearity='linear')  
 nn.init.kaiming\_uniform\_(self.poly\_weights, nonlinearity='linear')  
 nn.init.normal\_(  
 self.beta\_weights,  
 mean=0.0,  
 std=1.0 / ((kernel\_size \*\* ndim) \* self.inputdim \* (self.degree + 1.0)),  
 )  
  
 def beta(self, n, m):  
 # 计算beta值  
 return (  
 ((m + n) \* (m - n) \* n \*\* 2) / (m \*\* 2 / (4.0 \* n \*\* 2 - 1.0))  
 ) \* self.beta\_weights[n]  
  
 @lru\_cache(maxsize=128) # 使用缓存避免重复计算  
 def gram\_poly(self, x, degree):  
 # 计算Gram多项式  
 p0 = x.new\_ones(x.size()) # p0初始化为1  
  
 if degree == 0:  
 return p0.unsqueeze(-1)  
  
 p1 = x  
 grams\_basis = [p0, p1]  
  
 for i in range(2, degree + 1):  
 p2 = x \* p1 - self.beta(i - 1, i) \* p0 # 递归计算多项式  
 grams\_basis.append(p2)  
 p0, p1 = p1, p2  
  
 return torch.cat(grams\_basis, dim=1) # 合并多项式基  
  
 def forward\_kag(self, x, group\_index):  
 # 前向传播过程  
 basis = self.base\_conv[group\_index](self.base\_activation(x)) # 基础卷积  
  
 x = torch.tanh(x).contiguous() # 归一化输入  
  
 if self.dropout is not None:  
 x = self.dropout(x) # 应用Dropout  
  
 grams\_basis = self.base\_activation(self.gram\_poly(x, self.degree)) # 计算Gram多项式基  
 y = self.conv\_w\_fun(grams\_basis, self.poly\_weights[group\_index],  
 stride=self.stride, dilation=self.dilation,  
 padding=self.padding, groups=1) # 卷积操作  
  
 y = self.base\_activation(self.layer\_norm[group\_index](y + basis)) # 归一化和激活  
  
 return y  
  
 def forward(self, x):  
 # 整体前向传播  
 split\_x = torch.split(x, self.inputdim // self.groups, dim=1) # 按组分割输入  
 output = []  
 for group\_ind, \_x in enumerate(split\_x):  
 y = self.forward\_kag(\_x.clone(), group\_ind) # 对每个组进行前向传播  
 output.append(y.clone())  
 y = torch.cat(output, dim=1) # 合并输出  
 return y  
```  
  
### 代码说明：  
1. \*\*KAGNConvNDLayer\*\*: 这是一个自定义的卷积层，支持多维卷积（1D、2D、3D）。它使用了分组卷积和多项式基函数。  
2. \*\*初始化参数\*\*: 在构造函数中，初始化了输入输出维度、卷积参数、Dropout层等，并检查参数的有效性。  
3. \*\*权重初始化\*\*: 使用Kaiming均匀分布初始化卷积层和多项式权重，以帮助模型更好地训练。  
4. \*\*Gram多项式计算\*\*: `gram\_poly`方法计算Gram多项式，使用递归方式生成多项式基。  
5. \*\*前向传播\*\*: `forward\_kag`方法处理每个组的前向传播，应用卷积、激活和归一化。`forward`方法则负责整体的前向传播，将输入分组并处理每个组的输出。```

这个程序文件 `kagn\_conv.py` 定义了一组用于卷积神经网络的层，主要是 KAGN（Kochawongwat's Adaptive Graph Neural Network）卷积层，支持一维、二维和三维卷积。程序中使用了 PyTorch 框架，主要包含一个基类 `KAGNConvNDLayer` 和三个子类 `KAGNConv1DLayer`、`KAGNConv2DLayer` 和 `KAGNConv3DLayer`，分别用于处理一维、二维和三维数据。  
  
在 `KAGNConvNDLayer` 类的构造函数中，首先初始化了一些卷积层所需的参数，包括输入和输出维度、卷积核大小、步幅、填充、扩张、组数、丢弃率等。然后根据组数创建基础卷积层和归一化层的列表。接着，定义了多项式权重和 beta 权重，并使用 Kaiming 均匀分布初始化这些权重，以确保网络在训练开始时的表现更好。  
  
`beta` 方法用于计算与 Legendre 多项式相关的 beta 值。`gram\_poly` 方法计算 Legendre 多项式的基函数，并使用缓存机制来避免重复计算，提高效率。`forward\_kag` 方法是核心的前向传播逻辑，首先对输入进行激活和卷积操作，然后计算归一化和多项式的结果，最后将这些结果结合并返回。  
  
`forward` 方法将输入数据按组分割，然后对每个组调用 `forward\_kag` 方法进行处理，最后将所有组的输出拼接在一起，形成最终的输出。  
  
三个子类分别继承自 `KAGNConvNDLayer`，并在构造函数中指定卷积和归一化的具体实现，分别对应一维、二维和三维卷积操作。这样设计使得代码结构清晰，便于扩展和维护，同时可以处理不同维度的数据。整体上，这个文件实现了一个灵活且高效的卷积层，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，主要保留了 YOLO 检测验证器的核心功能和逻辑：  
  
```python  
import os  
import torch  
from ultralytics.engine.validator import BaseValidator  
from ultralytics.utils.metrics import DetMetrics, box\_iou  
from ultralytics.utils import LOGGER, ops  
  
class DetectionValidator(BaseValidator):  
 """  
 检测验证器类，继承自 BaseValidator，用于基于检测模型的验证。  
 """  
  
 def \_\_init\_\_(self, dataloader=None, save\_dir=None, args=None):  
 """初始化检测模型，设置必要的变量和参数。"""  
 super().\_\_init\_\_(dataloader, save\_dir, args)  
 self.metrics = DetMetrics(save\_dir=self.save\_dir) # 初始化检测指标  
 self.iouv = torch.linspace(0.5, 0.95, 10) # 定义 IoU 向量  
  
 def preprocess(self, batch):  
 """预处理图像批次以适应 YOLO 训练。"""  
 batch["img"] = batch["img"].to(self.device) / 255 # 将图像归一化到 [0, 1]  
 for k in ["batch\_idx", "cls", "bboxes"]:  
 batch[k] = batch[k].to(self.device) # 将标签和边界框转移到设备上  
 return batch  
  
 def postprocess(self, preds):  
 """对预测输出应用非极大值抑制（NMS）。"""  
 return ops.non\_max\_suppression(  
 preds,  
 self.args.conf,  
 self.args.iou,  
 multi\_label=True,  
 max\_det=self.args.max\_det,  
 )  
  
 def update\_metrics(self, preds, batch):  
 """更新检测指标。"""  
 for si, pred in enumerate(preds):  
 npr = len(pred) # 当前预测的数量  
 pbatch = self.\_prepare\_batch(si, batch) # 准备当前批次的标签  
 cls, bbox = pbatch.pop("cls"), pbatch.pop("bbox") # 获取当前批次的类和边界框  
  
 if npr == 0: # 如果没有预测  
 continue  
  
 predn = self.\_prepare\_pred(pred, pbatch) # 准备预测  
 stat = dict(conf=predn[:, 4], pred\_cls=predn[:, 5]) # 提取置信度和预测类  
 stat["tp"] = self.\_process\_batch(predn, bbox, cls) # 计算真阳性  
 self.stats["tp"].append(stat["tp"]) # 更新统计信息  
  
 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 get\_stats(self):  
 """返回指标统计信息和结果字典。"""  
 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())) # 打印总结果  
```  
  
### 代码注释说明：  
1. \*\*类定义\*\*：`DetectionValidator` 继承自 `BaseValidator`，用于处理 YOLO 模型的验证。  
2. \*\*初始化方法\*\*：设置指标、IoU 向量等基本参数。  
3. \*\*预处理方法\*\*：将输入图像归一化并将数据转移到指定设备（如 GPU）。  
4. \*\*后处理方法\*\*：应用非极大值抑制，过滤掉重叠的检测框。  
5. \*\*更新指标方法\*\*：根据预测结果和真实标签更新检测指标。  
6. \*\*处理批次方法\*\*：计算预测框与真实框之间的 IoU，并匹配预测和真实标签。  
7. \*\*获取统计信息方法\*\*：返回计算后的指标统计信息。  
8. \*\*打印结果方法\*\*：输出每个类的验证结果，包括总的检测结果。  
  
以上代码展示了 YOLO 检测验证器的核心逻辑和结构，注释详细说明了每个方法的功能和作用。```

这个程序文件 `val.py` 是一个用于基于YOLO（You Only Look Once）检测模型进行验证的类实现，继承自 `BaseValidator`。其主要功能是处理验证数据集，计算模型在目标检测任务中的性能指标，并可视化结果。  
  
在文件的开头，导入了一些必要的库和模块，包括操作系统模块、路径处理模块、NumPy、PyTorch以及Ultralytics库中的多个功能模块。接着定义了 `DetectionValidator` 类，包含多个方法用于不同的处理步骤。  
  
在初始化方法 `\_\_init\_\_` 中，设置了一些基本的变量和参数，包括数据加载器、保存目录、进度条、参数字典等。它还初始化了一些性能指标的计算对象，比如 `DetMetrics` 和 `ConfusionMatrix`，并定义了一个用于计算mAP（mean Average Precision）的IoU（Intersection over Union）向量。  
  
`preprocess` 方法用于对输入的图像批次进行预处理，包括将图像转换为适合模型输入的格式，并根据需要进行归一化处理。此方法还支持自动标注功能。  
  
`init\_metrics` 方法用于初始化评估指标，包括检查数据集是否为COCO格式，设置类别映射，并准备混淆矩阵等。  
  
`get\_desc` 方法返回一个格式化的字符串，用于总结YOLO模型的类别指标。  
  
`postprocess` 方法对模型的预测结果应用非极大值抑制（NMS），以减少重复的检测框。  
  
`\_prepare\_batch` 和 `\_prepare\_pred` 方法分别用于准备输入批次和预测结果，以便进行后续的指标计算。  
  
`update\_metrics` 方法负责更新当前批次的性能指标，包括正确预测的数量、置信度、预测类别等，并根据需要保存结果到文件。  
  
`finalize\_metrics` 方法用于设置最终的指标值，包括速度和混淆矩阵。  
  
`get\_stats` 方法返回当前的统计信息和结果字典，计算每个类别的目标数量。  
  
`print\_results` 方法用于打印训练或验证集的每个类别的指标结果，并根据需要绘制混淆矩阵。  
  
`\_process\_batch` 方法计算正确预测的矩阵，基于IoU值匹配预测框和真实框。  
  
`build\_dataset` 和 `get\_dataloader` 方法用于构建YOLO数据集和返回数据加载器，支持不同的批次大小和模式。  
  
`plot\_val\_samples` 和 `plot\_predictions` 方法用于可视化验证样本和预测结果，生成相应的图像文件。  
  
`save\_one\_txt` 方法将YOLO检测结果保存为文本文件，采用特定的格式。  
  
`pred\_to\_json` 方法将YOLO的预测结果序列化为COCO格式的JSON文件，便于后续评估。  
  
`eval\_json` 方法用于评估YOLO输出的JSON格式结果，并返回性能统计信息，支持与COCO API进行交互以计算mAP。  
  
总体来说，这个文件实现了YOLO模型验证的完整流程，包括数据预处理、指标计算、结果保存和可视化，适用于目标检测任务的评估和分析。

```以下是代码中最核心的部分，并附上详细的中文注释：  
  
```python  
import math  
import torch  
from torch import nn  
from einops.layers.torch import Rearrange  
  
# 定义一个二维卷积层的类，带有特定的权重获取方式  
class Conv2d\_cd(nn.Module):  
 def \_\_init\_\_(self, in\_channels, out\_channels, kernel\_size=3, stride=1,  
 padding=1, dilation=1, groups=1, bias=False, theta=1.0):  
 super(Conv2d\_cd, self).\_\_init\_\_()   
 # 初始化标准的卷积层  
 self.conv = nn.Conv2d(in\_channels, out\_channels, kernel\_size=kernel\_size, stride=stride, padding=padding, dilation=dilation, groups=groups, bias=bias)  
 self.theta = theta # 可调参数 theta  
   
 def get\_weight(self):  
 # 获取卷积层的权重  
 conv\_weight = self.conv.weight  
 conv\_shape = conv\_weight.shape # 权重的形状  
 # 重排权重的维度  
 conv\_weight = Rearrange('c\_in c\_out k1 k2 -> c\_in c\_out (k1 k2)')(conv\_weight)  
   
 # 创建一个新的权重张量，初始化为0  
 conv\_weight\_cd = torch.zeros(conv\_shape[0], conv\_shape[1], 3 \* 3, device=conv\_weight.device, dtype=conv\_weight.dtype)  
 # 将原始权重复制到新的权重张量中  
 conv\_weight\_cd[:, :, :] = conv\_weight[:, :, :]  
 # 调整权重  
 conv\_weight\_cd[:, :, 4] = conv\_weight[:, :, 4] - conv\_weight[:, :, :].sum(2)  
 # 重排回原来的形状  
 conv\_weight\_cd = Rearrange('c\_in c\_out (k1 k2) -> c\_in c\_out k1 k2', k1=conv\_shape[2], k2=conv\_shape[3])(conv\_weight\_cd)  
 return conv\_weight\_cd, self.conv.bias # 返回调整后的权重和偏置  
  
  
# 定义一个用于特定权重调整的卷积层  
class Conv2d\_ad(nn.Module):  
 def \_\_init\_\_(self, in\_channels, out\_channels, kernel\_size=3, stride=1,  
 padding=1, dilation=1, groups=1, bias=False, theta=1.0):  
 super(Conv2d\_ad, self).\_\_init\_\_()   
 self.conv = nn.Conv2d(in\_channels, out\_channels, kernel\_size=kernel\_size, stride=stride, padding=padding, dilation=dilation, groups=groups, bias=bias)  
 self.theta = theta # 可调参数 theta  
   
 def get\_weight(self):  
 # 获取卷积层的权重  
 conv\_weight = self.conv.weight  
 conv\_shape = conv\_weight.shape  
 # 重排权重的维度  
 conv\_weight = Rearrange('c\_in c\_out k1 k2 -> c\_in c\_out (k1 k2)')(conv\_weight)  
 # 根据 theta 调整权重  
 conv\_weight\_ad = conv\_weight - self.theta \* conv\_weight[:, :, [3, 0, 1, 6, 4, 2, 7, 8, 5]]  
 # 重排回原来的形状  
 conv\_weight\_ad = Rearrange('c\_in c\_out (k1 k2) -> c\_in c\_out k1 k2', k1=conv\_shape[2], k2=conv\_shape[3])(conv\_weight\_ad)  
 return conv\_weight\_ad, self.conv.bias # 返回调整后的权重和偏置  
  
  
# 定义一个包含多个卷积层的反卷积类  
class DEConv(nn.Module):  
 def \_\_init\_\_(self, dim):  
 super(DEConv, self).\_\_init\_\_()  
 # 初始化多个不同类型的卷积层  
 self.conv1\_1 = Conv2d\_cd(dim, dim, 3, bias=True)  
 self.conv1\_2 = Conv2d\_ad(dim, dim, 3, bias=True)  
 self.conv1\_3 = nn.Conv2d(dim, dim, 3, padding=1, bias=True) # 标准卷积层  
   
 self.bn = nn.BatchNorm2d(dim) # 批归一化层  
 self.act = nn.ReLU() # 激活函数  
  
 def forward(self, x):  
 # 前向传播  
 w1, b1 = self.conv1\_1.get\_weight() # 获取第一个卷积层的权重和偏置  
 w2, b2 = self.conv1\_2.get\_weight() # 获取第二个卷积层的权重和偏置  
 w3, b3 = self.conv1\_3.weight, self.conv1\_3.bias # 获取标准卷积层的权重和偏置  
  
 # 将所有卷积层的权重和偏置相加  
 w = w1 + w2 + w3  
 b = b1 + b2 + b3  
 # 进行卷积操作  
 res = nn.functional.conv2d(input=x, weight=w, bias=b, stride=1, padding=1, groups=1)  
   
 # 进行批归一化  
 res = self.bn(res)  
   
 return self.act(res) # 返回激活后的结果  
  
 def switch\_to\_deploy(self):  
 # 切换到部署模式，合并权重和偏置  
 w1, b1 = self.conv1\_1.get\_weight()  
 w2, b2 = self.conv1\_2.get\_weight()  
 w3, b3 = self.conv1\_3.weight, self.conv1\_3.bias  
  
 # 合并权重和偏置  
 self.conv1\_3.weight = torch.nn.Parameter(w1 + w2 + w3)  
 self.conv1\_3.bias = torch.nn.Parameter(b1 + b2 + b3)  
   
 # 删除不再需要的卷积层  
 del self.conv1\_1  
 del self.conv1\_2  
```  
  
### 代码核心部分解释：  
1. \*\*卷积层定义\*\*：`Conv2d\_cd` 和 `Conv2d\_ad` 类定义了具有特殊权重获取方式的卷积层。`get\_weight` 方法通过重排和调整权重来实现特定的卷积操作。  
   
2. \*\*反卷积类\*\*：`DEConv` 类整合了多个卷积层，并在前向传播中合并它们的权重和偏置。它还包含批归一化和激活函数的应用。  
  
3. \*\*部署模式\*\*：`switch\_to\_deploy` 方法用于将训练模式下的多个卷积层合并为一个卷积层，以提高推理效率。  
  
这些核心部分构成了一个复杂的卷积神经网络模块，能够灵活地处理卷积操作和权重调整。```

这个程序文件 `deconv.py` 定义了一系列自定义的卷积层，主要用于深度学习中的图像处理任务。文件中包含多个类，每个类都实现了不同类型的卷积操作，主要是针对二维卷积的变体。  
  
首先，文件导入了必要的库，包括 `math`、`torch` 和 `torch.nn`，以及一些用于重排张量的工具和其他模块。接下来，定义了几个卷积类，每个类都继承自 `nn.Module`，并在初始化时创建了一个标准的二维卷积层 `nn.Conv2d`。  
  
`Conv2d\_cd` 类实现了一种特殊的卷积操作，提供了一个 `get\_weight` 方法，该方法通过重排卷积权重的形状并进行一些调整，返回处理后的权重和偏置。  
  
`Conv2d\_ad` 类与 `Conv2d\_cd` 类类似，但在权重的计算上有所不同，使用了一个 `theta` 参数来调整权重。  
  
`Conv2d\_rd` 类则实现了前向传播的方法 `forward`，在其中根据 `theta` 的值决定使用标准卷积还是经过调整的卷积权重。  
  
`Conv2d\_hd` 和 `Conv2d\_vd` 类实现了不同的卷积权重获取方式，主要是通过对权重进行特定的重排和调整，形成新的权重矩阵。  
  
`DEConv` 类是一个更高层次的模型，它将之前定义的卷积层组合在一起，形成一个完整的网络结构。在 `forward` 方法中，所有卷积层的权重和偏置被组合在一起，然后通过一个标准的卷积操作得到最终的输出。此外，该类还包含了批归一化和激活函数的应用。  
  
`switch\_to\_deploy` 方法用于在模型部署时优化模型结构，将所有卷积层的权重和偏置合并到最后一个卷积层中，从而减少计算量和内存占用。该方法还删除了不再需要的卷积层。  
  
最后，在 `\_\_main\_\_` 块中，创建了一个随机输入数据，并实例化了 `DEConv` 模型。通过调用 `forward` 方法和 `switch\_to\_deploy` 方法，验证了模型在合并权重后的输出与原始输出是否一致。  
  
整体来看，这个文件实现了一个复杂的卷积神经网络模块，允许用户通过自定义的卷积层进行灵活的网络设计，同时提供了模型优化的功能。

### 程序整体功能和构架概括  
  
该项目的整体功能是实现一个用于计算机视觉任务的深度学习框架，特别是针对目标检测和图像处理的卷积神经网络。项目包含多个模块，每个模块负责不同的功能，从模型的定义到验证和优化，构成了一个完整的深度学习工作流。  
  
1. \*\*模型定义\*\*：`mobilenetv4.py` 定义了 MobileNetV4 模型，适用于资源受限的设备，提供了高效的卷积层和网络结构。  
2. \*\*自定义卷积层\*\*：`kagn\_conv.py` 实现了一系列自定义的卷积层，支持一维、二维和三维卷积，增强了模型的灵活性和适应性。  
3. \*\*验证与评估\*\*：`val.py` 提供了验证模型性能的功能，包括数据预处理、指标计算、结果可视化等，支持与 COCO 格式的数据交互。  
4. \*\*卷积操作优化\*\*：`deconv.py` 实现了多种卷积层的变体，并提供了模型优化的方法，减少了计算量，提高了推理效率。  
  
### 文件功能整理表  
  
| 文件名 | 功能描述 |  
|---------------------|--------------------------------------------------------------------------|  
| `mobilenetv4.py` | 定义 MobileNetV4 模型结构，包含卷积层配置、倒残差块和前向传播逻辑。 |  
| `kagn\_conv.py` | 实现 KAGN 卷积层，支持一维、二维和三维卷积，提供自定义卷积操作的灵活性。 |  
| `val.py` | 处理模型验证，计算性能指标，支持结果可视化和与 COCO 格式的交互。 |  
| `deconv.py` | 定义多种自定义卷积层，提供卷积操作的优化和合并权重的方法，提升模型效率。 |  
  
通过这些模块的协同工作，该项目能够高效地构建、训练和评估深度学习模型，特别是在目标检测和图像处理领域。