# 改进yolo11-DySnakeConv等200+全套创新点大全：货架空缺检测系统源码＆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  
  
在现代零售环境中，货架管理的有效性直接影响到消费者的购物体验和零售商的销售业绩。随着电子商务的迅猛发展，实体店面面临着前所未有的竞争压力，如何提升货架的商品可用性成为了零售商亟待解决的问题。货架空缺不仅导致消费者无法购买所需商品，还可能影响品牌形象和顾客忠诚度。因此，及时准确地检测货架空缺情况，对于零售商进行库存管理、优化补货策略具有重要意义。  
  
近年来，计算机视觉技术的快速发展为货架空缺检测提供了新的解决方案。YOLO（You Only Look Once）系列模型因其高效的实时目标检测能力而受到广泛关注。YOLOv11作为该系列的最新版本，具备更强的特征提取能力和更快的处理速度，适合于复杂的零售环境中进行货架空缺检测。然而，现有的YOLOv11模型在特定应用场景下仍存在一定的局限性，尤其是在小样本数据集上表现不佳。因此，改进YOLOv11模型以适应货架空缺检测的需求显得尤为重要。  
  
本研究基于141张图像的货架空缺检测数据集，旨在通过改进YOLOv11模型，提升其在实际应用中的检测精度和鲁棒性。通过引入数据增强技术、优化模型结构和调整训练策略，我们期望能够有效提高模型对货架空缺的识别能力。此外，本研究的成果不仅能够为零售商提供实时的货架监控解决方案，还能为后续的智能零售系统发展奠定基础，推动零售行业的数字化转型。

## 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“Empty Shelf Detector”，旨在为改进YOLOv11的货架空缺检测系统提供强有力的支持。该数据集专注于识别货架上的缺货情况，特别是针对“Out-of-Stock-Instances”这一类别进行深入分析。数据集中仅包含一个类别，旨在简化模型的训练过程，同时确保其在特定任务上的准确性和有效性。  
  
“Empty Shelf Detector”数据集的构建过程涉及大量真实场景的图像采集，涵盖了多种零售环境下的货架布局。这些图像不仅展示了货架的不同角度和光照条件，还考虑了不同商品的摆放方式和货架的整体设计。通过多样化的样本，数据集确保了模型在实际应用中的鲁棒性和适应性。  
  
在数据标注方面，所有图像均经过专业人员的仔细审核与标注，确保每个缺货实例都被准确识别。数据集中的图像标注不仅包括缺货商品的具体位置，还提供了相应的上下文信息，以帮助模型更好地理解货架的整体状态。这种细致的标注方式为后续的模型训练提供了丰富的训练样本，极大地提升了YOLOv11在货架空缺检测任务中的表现。  
  
通过使用“Empty Shelf Detector”数据集，研究团队期望能够显著提高YOLOv11在实时货架监测中的检测精度和响应速度，为零售行业提供更为智能化的解决方案。该数据集的独特性和针对性使其成为本项目的核心组成部分，为实现高效的货架管理和库存监控奠定了坚实的基础。

## 5. 全套项目环境部署教程（零基础手把手教学）

5.1 所需软件PyCharm和Anaconda安装教程（第一步）

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

5.2 安装Python虚拟环境创建和依赖库安装视频教程（第二步）

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

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

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

## 7.原始YOLOv11算法讲解

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

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

https://gitee.com/qunmasj/good

## 9.系统功能展示（检测对象为举例，实际内容以本项目数据集为准）

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

## 10. YOLOv11核心改进模块源码讲解

```以下是经过简化和注释的核心代码部分：  
  
```python  
import torch  
import torch.nn as nn  
from einops import rearrange  
  
class LayerNorm2d(nn.Module):  
 """自定义的二维层归一化类"""  
 def \_\_init\_\_(self, normalized\_shape, eps=1e-6, elementwise\_affine=True):  
 super().\_\_init\_\_()  
 self.norm = nn.LayerNorm(normalized\_shape, eps, elementwise\_affine)  
  
 def forward(self, x):  
 # 将输入张量从 (B, C, H, W) 转换为 (B, H, W, C)  
 x = rearrange(x, 'b c h w -> b h w c').contiguous()  
 x = self.norm(x) # 应用层归一化  
 # 再将张量转换回 (B, C, H, W)  
 x = rearrange(x, 'b h w c -> b c h w').contiguous()  
 return x  
  
class CrossScan(torch.autograd.Function):  
 """交叉扫描操作的自定义函数"""  
 @staticmethod  
 def forward(ctx, x: torch.Tensor):  
 B, C, H, W = x.shape  
 ctx.shape = (B, C, H, W)  
 xs = x.new\_empty((B, 4, C, H \* W)) # 创建新的张量  
 xs[:, 0] = x.flatten(2, 3) # 展平 H 和 W 维度  
 xs[:, 1] = x.transpose(dim0=2, dim1=3).flatten(2, 3) # 转置并展平  
 xs[:, 2:4] = torch.flip(xs[:, 0:2], dims=[-1]) # 翻转  
 return xs  
  
 @staticmethod  
 def backward(ctx, ys: torch.Tensor):  
 B, C, H, W = ctx.shape  
 L = H \* W  
 # 反向传播计算  
 ys = ys[:, 0:2] + ys[:, 2:4].flip(dims=[-1]).view(B, 2, -1, L)  
 y = ys[:, 0] + ys[:, 1].view(B, -1, W, H).transpose(dim0=2, dim1=3).contiguous().view(B, -1, L)  
 return y.view(B, -1, H, W)  
  
class SelectiveScanCore(torch.autograd.Function):  
 """选择性扫描的核心操作"""  
 @staticmethod  
 @torch.cuda.amp.custom\_fwd  
 def forward(ctx, u, delta, A, B, C, D=None, delta\_bias=None, delta\_softplus=False, nrows=1, backnrows=1):  
 # 确保输入张量是连续的  
 if u.stride(-1) != 1:  
 u = u.contiguous()  
 if delta.stride(-1) != 1:  
 delta = delta.contiguous()  
 if D is not None and D.stride(-1) != 1:  
 D = D.contiguous()  
 if B.stride(-1) != 1:  
 B = B.contiguous()  
 if C.stride(-1) != 1:  
 C = C.contiguous()  
   
 # 进行选择性扫描的前向计算  
 out, x, \*rest = selective\_scan\_cuda\_core.fwd(u, delta, A, B, C, D, delta\_bias, delta\_softplus, 1)  
 ctx.save\_for\_backward(u, delta, A, B, C, D, delta\_bias, x) # 保存反向传播需要的张量  
 return out  
  
 @staticmethod  
 @torch.cuda.amp.custom\_bwd  
 def backward(ctx, dout):  
 u, delta, A, B, C, D, delta\_bias, x = ctx.saved\_tensors  
 # 进行选择性扫描的反向计算  
 du, ddelta, dA, dB, dC, dD, ddelta\_bias, \*rest = selective\_scan\_cuda\_core.bwd(  
 u, delta, A, B, C, D, delta\_bias, dout, x, ctx.delta\_softplus, 1  
 )  
 return (du, ddelta, dA, dB, dC, dD, ddelta\_bias, None, None, None, None)  
  
class SS2D(nn.Module):  
 """选择性扫描的二维实现"""  
 def \_\_init\_\_(self, d\_model=96, d\_state=16, ssm\_ratio=2.0, act\_layer=nn.SiLU, dropout=0.0):  
 super().\_\_init\_\_()  
 self.in\_proj = nn.Conv2d(d\_model, d\_model \* ssm\_ratio, kernel\_size=1) # 输入投影  
 self.out\_proj = nn.Conv2d(d\_model \* ssm\_ratio, d\_model, kernel\_size=1) # 输出投影  
 self.act = act\_layer() # 激活函数  
 self.dropout = nn.Dropout(dropout) # Dropout层  
  
 def forward(self, x: torch.Tensor):  
 x = self.in\_proj(x) # 输入投影  
 x = self.act(x) # 激活  
 x = self.dropout(x) # Dropout  
 x = self.out\_proj(x) # 输出投影  
 return x  
  
class VSSBlock\_YOLO(nn.Module):  
 """YOLO模型中的选择性扫描块"""  
 def \_\_init\_\_(self, in\_channels: int, hidden\_dim: int, drop\_path: float):  
 super().\_\_init\_\_()  
 self.proj\_conv = nn.Conv2d(in\_channels, hidden\_dim, kernel\_size=1) # 投影卷积  
 self.ss2d = SS2D(d\_model=hidden\_dim) # 选择性扫描层  
 self.drop\_path = nn.Dropout(drop\_path) # DropPath层  
  
 def forward(self, input: torch.Tensor):  
 input = self.proj\_conv(input) # 投影  
 x = self.ss2d(input) # 选择性扫描  
 x = self.drop\_path(x) # DropPath  
 return x  
```  
  
### 代码分析与注释  
1. \*\*LayerNorm2d\*\*: 自定义的二维层归一化模块，输入为四维张量，进行归一化处理。  
2. \*\*CrossScan\*\*: 实现交叉扫描的前向和反向传播，主要用于对输入张量进行不同方向的处理。  
3. \*\*SelectiveScanCore\*\*: 选择性扫描的核心操作，包含前向和反向传播的实现，使用CUDA加速。  
4. \*\*SS2D\*\*: 选择性扫描的二维实现，包含输入和输出的卷积投影，以及激活和Dropout层。  
5. \*\*VSSBlock\_YOLO\*\*: YOLO模型中的选择性扫描块，负责对输入进行投影和选择性扫描处理。  
  
以上代码是模型的核心部分，其他部分如辅助函数和模块可以根据需要进行扩展和使用。```

这个程序文件 `mamba\_yolo.py` 实现了一个基于深度学习的视觉模型，主要用于目标检测任务，采用了 YOLO（You Only Look Once）系列的结构。文件中定义了多个类和函数，构成了整个模型的框架。以下是对代码的逐部分分析和说明。  
  
首先，文件引入了一些必要的库，包括 `torch`、`math`、`functools` 和 `torch.nn` 等。它还使用了 `einops` 库来处理张量的重排，以及 `timm` 库中的 `DropPath` 层来实现随机深度丢弃。  
  
接下来，定义了一个 `LayerNorm2d` 类，继承自 `nn.Module`，用于实现二维层归一化。它在前向传播中对输入张量进行形状重排，以适应 `LayerNorm` 的要求。  
  
`autopad` 函数用于自动计算卷积操作的填充，使得输出形状与输入形状相同。它根据给定的卷积核大小和膨胀因子来确定填充大小。  
  
然后，定义了 `CrossScan` 和 `CrossMerge` 类，这两个类分别实现了交叉扫描和交叉合并的功能，主要用于在模型中处理输入特征图的交叉信息。  
  
`SelectiveScanCore` 类实现了选择性扫描的核心功能，利用 CUDA 加速进行前向和反向传播。选择性扫描是一种用于高效处理序列数据的技术，能够在计算中保持较低的复杂度。  
  
`cross\_selective\_scan` 函数是一个封装函数，调用 `SelectiveScanCore` 进行选择性扫描，并处理输入的各种参数和张量。它支持多种输出归一化方式。  
  
`SS2D` 类是一个核心模块，包含了多种参数和层，主要用于实现选择性扫描的功能。它的构造函数中定义了输入和输出的投影层、卷积层以及其他相关参数。  
  
`RGBlock` 和 `LSBlock` 类分别实现了不同的模块，前者包含了全连接层和深度卷积层，后者则使用了批归一化和激活函数。  
  
`XSSBlock` 类是一个复杂的模块，结合了多个子模块，使用选择性扫描和其他操作来处理输入特征图。它的前向传播方法通过多个步骤处理输入，并结合残差连接。  
  
`VSSBlock\_YOLO` 类是 YOLO 模型的一个具体实现，包含了选择性扫描模块和其他必要的层。它的前向传播方法整合了输入的特征图，并通过不同的分支进行处理。  
  
`SimpleStem` 类实现了模型的初始卷积层，用于处理输入图像并生成特征图。它使用了两个卷积层和激活函数。  
  
最后，`VisionClueMerge` 类用于合并不同来源的特征图，以便在后续的处理步骤中使用。  
  
整体来看，这个文件实现了一个复杂的视觉模型，结合了选择性扫描、卷积操作和多种激活函数，旨在提高目标检测的性能和效率。模型的设计考虑了多种因素，包括参数的初始化、前向传播的效率以及不同模块之间的连接方式。

```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 numpy as np  
from torch.nn import Dropout, Softmax, LayerNorm  
  
class Channel\_Embeddings(nn.Module):  
 """构建通道嵌入，包括位置嵌入和补丁嵌入。"""  
 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  
  
 # 使用最大池化和卷积层来提取补丁嵌入  
 self.patch\_embeddings = nn.Sequential(  
 nn.MaxPool2d(kernel\_size=5, stride=5),  
 nn.Conv2d(in\_channels=in\_channels,  
 out\_channels=in\_channels,  
 kernel\_size=patchsize // 5,  
 stride=patchsize // 5)  
 )  
  
 # 初始化位置嵌入  
 self.position\_embeddings = nn.Parameter(torch.zeros(1, n\_patches, in\_channels))  
 self.dropout = Dropout(0.1) # Dropout层以防止过拟合  
  
 def forward(self, x):  
 """前向传播，计算嵌入。"""  
 if x is None:  
 return None  
 x = self.patch\_embeddings(x) # 提取补丁嵌入  
 x = x.flatten(2) # 将特征展平  
 x = x.transpose(-1, -2) # 转置以适应位置嵌入  
 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):  
 """实现多头注意力机制。"""  
 def \_\_init\_\_(self, vis, channel\_num):  
 super(Attention\_org, self).\_\_init\_\_()  
 self.vis = vis # 可视化标志  
 self.KV\_size = sum(channel\_num) # 键值对的大小  
 self.channel\_num = channel\_num # 通道数量  
 self.num\_attention\_heads = 4 # 注意力头的数量  
  
 # 初始化查询、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  
 self.proj\_dropout = Dropout(0.1) # Dropout层  
  
 def forward(self, \*embeddings):  
 """前向传播，计算注意力输出。"""  
 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)) # 将所有嵌入拼接  
 multi\_head\_V = self.value(torch.cat(embeddings, dim=2)) # 将所有嵌入拼接  
  
 # 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  
  
 # 返回输出和注意力权重  
 outputs = [self.proj\_dropout(layer) for layer in context\_layers]  
 return outputs  
  
class ChannelTransformer(nn.Module):  
 """通道变换器，整合多个嵌入并进行编码。"""  
 def \_\_init\_\_(self, channel\_num=[64, 128, 256, 512], img\_size=640, vis=False, patchSize=[40, 20, 10, 5]):  
 super().\_\_init\_\_()  
 self.embeddings = nn.ModuleList([Channel\_Embeddings(patch, img\_size // (2 \*\* i), c)   
 for i, (patch, c) in enumerate(zip(patchSize, channel\_num))])  
 self.encoder = Encoder(vis, channel\_num) # 编码器  
 self.reconstruct = nn.ModuleList([Reconstruct(c, c, kernel\_size=1, scale\_factor=(patch, patch))   
 for patch, c in zip(patchSize, channel\_num)])  
  
 def forward(self, en):  
 """前向传播，计算最终输出。"""  
 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]  
 return reconstructed  
  
class GetIndexOutput(nn.Module):  
 """获取特定索引的输出。"""  
 def \_\_init\_\_(self, index):  
 super().\_\_init\_\_()  
 self.index = index  
  
 def forward(self, x):  
 """返回指定索引的输出。"""  
 return x[self.index]  
```  
  
### 代码说明：  
1. \*\*Channel\_Embeddings\*\*：构建通道嵌入，包含补丁嵌入和位置嵌入，使用卷积和池化层提取特征。  
2. \*\*Attention\_org\*\*：实现多头注意力机制，计算注意力分数和上下文层，支持可视化。  
3. \*\*ChannelTransformer\*\*：整合多个嵌入，进行编码和重构，返回最终输出。  
4. \*\*GetIndexOutput\*\*：根据指定索引获取输出。  
  
以上是代码的核心部分和详细注释，帮助理解其功能和实现方式。```

这个程序文件 `CTrans.py` 实现了一个基于通道的变换器（Channel Transformer），主要用于图像处理任务。程序中定义了多个类，每个类负责不同的功能，整体结构体现了深度学习中常见的模块化设计。  
  
首先，文件导入了一些必要的库，包括 PyTorch 及其模块，NumPy，以及其他用于深度学习的工具。接下来，定义了几个主要的类。  
  
`Channel\_Embeddings` 类用于构建图像的嵌入表示。它通过最大池化和卷积操作将输入图像分割成多个小块（patch），并为每个小块生成位置嵌入。该类的 `forward` 方法接收输入图像并返回经过处理的嵌入。  
  
`Reconstruct` 类则负责将嵌入的特征图重建为原始图像的大小。它使用卷积和批归一化层来处理输入的嵌入，并通过上采样将其恢复到更高的分辨率。  
  
`Attention\_org` 类实现了多头注意力机制。它将输入的嵌入通过线性变换生成查询（Query）、键（Key）和值（Value），并计算注意力分数。注意力分数经过归一化后用于加权求和，生成上下文向量。该类的 `forward` 方法处理多个输入嵌入，并返回加权后的输出。  
  
`Mlp` 类定义了一个简单的多层感知机（MLP），用于对嵌入进行进一步的非线性变换。它包含两个全连接层和一个激活函数（GELU），并在每层后应用 dropout。  
  
`Block\_ViT` 类实现了一个变换器块，包含了注意力机制和前馈网络。它首先对输入的嵌入进行层归一化，然后通过注意力层和前馈网络进行处理，最后将结果与原始输入相加以实现残差连接。  
  
`Encoder` 类由多个 `Block\_ViT` 组成，负责对输入的嵌入进行多次处理。它同样实现了层归一化，并在每个块的输出中收集注意力权重。  
  
`ChannelTransformer` 类是整个模型的核心，负责将输入的多个通道（图像）进行嵌入、编码和重建。它通过多个 `Channel\_Embeddings` 和 `Reconstruct` 实例来处理不同通道的输入，并将编码后的特征进行重建。  
  
最后，`GetIndexOutput` 类用于从模型的输出中提取特定索引的结果，方便后续处理。  
  
整体来看，这个程序实现了一个复杂的深度学习模型，结合了卷积神经网络和变换器的优点，适用于图像特征提取和重建任务。每个模块的设计都考虑到了深度学习中的常见操作，如注意力机制、残差连接和层归一化等，使得模型在处理复杂任务时能够保持良好的性能。

```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 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\_\_()  
 # 初始化输入和输出维度、卷积参数等  
 self.inputdim = input\_dim  
 self.outdim = output\_dim  
 self.degree = degree  
 self.kernel\_size = kernel\_size  
 self.padding = padding  
 self.stride = stride  
 self.dilation = dilation  
 self.groups = groups  
 self.base\_activation = nn.SiLU() # 使用SiLU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 = None # 初始化dropout层  
  
 # 根据维度选择合适的dropout层  
 if dropout > 0:  
 if ndim == 1:  
 self.dropout = nn.Dropout1d(p=dropout)  
 elif ndim == 2:  
 self.dropout = nn.Dropout2d(p=dropout)  
 elif ndim == 3:  
 self.dropout = nn.Dropout3d(p=dropout)  
  
 # 检查groups参数的有效性  
 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)])  
  
 # 初始化多项式权重  
 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  
  
 # 使用Kaiming均匀分布初始化卷积层权重  
 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Legendre多项式  
 def gram\_poly(self, x, degree):  
 # 计算Legendre多项式  
 p0 = x.new\_ones(x.size()) # p0初始化为1  
  
 if degree == 0:  
 return p0.unsqueeze(-1)  
  
 p1 = x # p1初始化为x  
 grams\_basis = [p0, p1] # 存储多项式基  
  
 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 # 更新p0和p1  
  
 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)) # 通过基本卷积层计算基础输出  
  
 x = torch.tanh(x).contiguous() # 对输入进行tanh归一化  
  
 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) # 计算卷积输出  
  
 y = self.base\_activation(self.layer\_norm[group\_index](y + basis)) # 归一化并激活  
  
 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  
```  
  
### 代码核心部分说明：  
1. \*\*KAGNConvNDLayer类\*\*：这是一个自定义的卷积层，支持多维卷积操作。它使用了基于Legendre多项式的卷积方法，并且支持不同的归一化和激活函数。  
2. \*\*初始化方法\*\*：在初始化时设置了输入输出维度、卷积参数、dropout等，并且创建了多个卷积层和归一化层。  
3. \*\*beta和gram\_poly方法\*\*：这两个方法用于计算Legendre多项式的相关值，`gram\_poly`方法使用了缓存以提高效率。  
4. \*\*forward\_kag方法\*\*：实现了前向传播的具体逻辑，计算基础卷积输出和多项式基的卷积输出，并进行归一化和激活。  
5. \*\*forward方法\*\*：处理输入数据的整体前向传播，将输入按组分割并调用`forward\_kag`进行处理，最后合并输出。  
  
这个类的设计使得它能够灵活地处理不同维度的卷积操作，并且通过多项式权重的学习增强了模型的表达能力。```

这个程序文件 `kagn\_conv.py` 定义了一个名为 `KAGNConvNDLayer` 的神经网络层，以及其一维、二维和三维的特定实现。该层的设计灵感来源于一种特殊的卷积结构，旨在通过引入多项式基函数来增强卷积操作的表达能力。  
  
首先，`KAGNConvNDLayer` 类是一个通用的卷积层，支持任意维度的卷积操作。它的构造函数接受多个参数，包括输入和输出维度、卷积核大小、分组数、填充、步幅、扩张率、丢弃率等。构造函数中还定义了基本的激活函数（使用 SiLU 激活函数），并根据输入的维度选择相应的丢弃层（Dropout）。在参数检查中，确保分组数是正整数，并且输入和输出维度能够被分组数整除。  
  
该类还定义了基础卷积层和归一化层，使用 `nn.ModuleList` 来管理这些层。多项式权重 `poly\_weights` 和 `beta\_weights` 被定义为可学习的参数，并使用 Kaiming 均匀分布进行初始化，以提高训练的稳定性。  
  
`beta` 方法用于计算与 Legendre 多项式相关的系数，而 `gram\_poly` 方法则计算给定输入和多项式的度数的 Gram 多项式。为了避免重复计算，`gram\_poly` 方法使用了 LRU 缓存。  
  
在 `forward\_kag` 方法中，首先对输入应用基本激活函数，然后进行线性变换。接着，将输入标准化到 [-1, 1] 的范围，以便于计算 Legendre 多项式。之后，计算 Gram 基，并使用传入的卷积函数和多项式权重进行卷积操作，最后进行归一化和激活。  
  
`forward` 方法将输入分割成多个组，并对每个组调用 `forward\_kag` 方法，最后将所有组的输出拼接在一起。  
  
接下来的三个类 `KAGNConv3DLayer`、`KAGNConv2DLayer` 和 `KAGNConv1DLayer` 分别是 `KAGNConvNDLayer` 的具体实现，针对三维、二维和一维卷积进行了相应的封装。这些类通过调用父类的构造函数，传入适当的卷积类和归一化类，简化了多维卷积层的创建过程。  
  
整体来看，这个文件实现了一个灵活且强大的卷积层，能够处理不同维度的数据，并通过多项式基函数增强了卷积操作的能力。

```以下是经过简化和注释的核心代码部分，主要集中在 `DetectionValidator` 类的关键功能上：  
  
```python  
import os  
import torch  
from ultralytics.utils import LOGGER, ops  
from ultralytics.utils.metrics import DetMetrics, box\_iou  
from ultralytics.utils.plotting import output\_to\_target, plot\_images  
  
class DetectionValidator:  
 """  
 用于基于检测模型的验证的类。  
 """  
  
 def \_\_init\_\_(self, dataloader=None, save\_dir=None, args=None):  
 """初始化检测模型所需的变量和设置。"""  
 self.dataloader = dataloader # 数据加载器  
 self.save\_dir = save\_dir # 保存目录  
 self.args = args # 参数  
 self.metrics = DetMetrics(save\_dir=self.save\_dir) # 初始化检测指标  
 self.iouv = torch.linspace(0.5, 0.95, 10) # 用于计算mAP的IoU阈值  
 self.niou = self.iouv.numel() # IoU的数量  
  
 def preprocess(self, batch):  
 """预处理图像批次以进行YOLO训练。"""  
 # 将图像数据转移到设备上并进行归一化  
 batch["img"] = batch["img"].to(self.device, non\_blocking=True) / 255  
 # 将其他数据转移到设备上  
 for k in ["batch\_idx", "cls", "bboxes"]:  
 batch[k] = batch[k].to(self.device)  
 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):  
 """更新指标统计信息。"""  
 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  
  
 # 处理预测  
 predn = self.\_prepare\_pred(pred, pbatch) # 准备预测数据  
 # 计算正确预测  
 stat = self.\_process\_batch(predn, bbox, cls)  
 # 更新统计信息  
 for k in self.stats.keys():  
 self.stats[k].append(stat[k])  
  
 def \_process\_batch(self, detections, gt\_bboxes, gt\_cls):  
 """  
 返回正确预测矩阵。  
 """  
 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 plot\_predictions(self, batch, preds, ni):  
 """在输入图像上绘制预测的边界框并保存结果。"""  
 plot\_images(  
 batch["img"],  
 \*output\_to\_target(preds),  
 paths=batch["im\_file"],  
 fname=self.save\_dir / f"val\_batch{ni}\_pred.jpg",  
 names=self.names,  
 ) # 绘制预测结果  
  
 def save\_one\_txt(self, predn, save\_conf, shape, file):  
 """将YOLO检测结果保存到txt文件中。"""  
 gn = torch.tensor(shape)[[1, 0, 1, 0]] # 归一化增益  
 for \*xyxy, conf, cls in predn.tolist():  
 xywh = (ops.xyxy2xywh(torch.tensor(xyxy).view(1, 4)) / gn).view(-1).tolist() # 转换为归一化的xywh格式  
 line = (cls, \*xywh, conf) if save\_conf else (cls, \*xywh) # 保存格式  
 with open(file, "a") as f:  
 f.write(("%g " \* len(line)).rstrip() % line + "\n") # 写入文件  
  
 def eval\_json(self, stats):  
 """评估YOLO输出的JSON格式并返回性能统计信息。"""  
 # 省略具体实现，主要用于评估模型性能  
 pass  
```  
  
### 代码注释说明：  
1. \*\*类初始化\*\*：`\_\_init\_\_` 方法初始化了数据加载器、保存目录、参数和指标。  
2. \*\*预处理\*\*：`preprocess` 方法对输入批次进行归一化处理，并将数据转移到指定设备。  
3. \*\*后处理\*\*：`postprocess` 方法使用非极大值抑制（NMS）来过滤预测结果。  
4. \*\*更新指标\*\*：`update\_metrics` 方法更新检测指标，通过比较预测和真实标签来计算正确预测。  
5. \*\*绘制预测\*\*：`plot\_predictions` 方法在图像上绘制预测的边界框并保存结果。  
6. \*\*保存结果\*\*：`save\_one\_txt` 方法将检测结果保存为txt文件，格式为YOLO所需的格式。  
7. \*\*评估\*\*：`eval\_json` 方法用于评估模型性能，具体实现省略。  
  
以上是核心代码部分的简化和注释，保留了主要功能和逻辑。```

这个程序文件 `val.py` 是一个用于YOLO（You Only Look Once）目标检测模型验证的实现。它继承自 `BaseValidator` 类，提供了一系列功能来处理模型验证过程中的数据预处理、指标计算和结果输出。  
  
在初始化时，`DetectionValidator` 类会设置一些必要的变量和参数，包括任务类型、度量标准、IoU（Intersection over Union）阈值等。它会根据传入的参数初始化数据加载器、保存目录和进度条等。类中定义了多个方法来处理不同的验证步骤。  
  
`preprocess` 方法用于对输入的图像批次进行预处理，包括将图像转换为合适的格式和归一化处理，同时也处理边界框的坐标。`init\_metrics` 方法则用于初始化验证所需的指标，比如确定数据集是否为COCO格式，并准备相关的类别映射和混淆矩阵。  
  
在验证过程中，`postprocess` 方法应用非极大值抑制（NMS）来过滤掉重叠的检测结果，确保最终输出的边界框是最优的。`update\_metrics` 方法则用于更新验证过程中收集的指标，包括真实标签和预测结果的比较，计算TP（True Positive）等。  
  
此外，`finalize\_metrics` 方法在验证结束时设置最终的指标值，`get\_stats` 方法则返回计算得到的统计结果。`print\_results` 方法负责输出每个类别的验证结果，包括每个类别的图像数量、实例数量和各类指标。  
  
程序还提供了一些可视化功能，比如 `plot\_val\_samples` 和 `plot\_predictions` 方法，用于绘制验证样本和预测结果的图像。`save\_one\_txt` 和 `pred\_to\_json` 方法则用于将检测结果保存为文本文件或JSON格式，以便后续分析和评估。  
  
最后，`eval\_json` 方法用于评估YOLO模型在COCO格式下的输出，计算mAP（mean Average Precision）等性能指标。它会调用pycocotools库来处理COCO格式的评估，确保验证结果的准确性。  
  
总体来说，这个文件实现了YOLO模型验证的完整流程，从数据预处理到结果输出，涵盖了目标检测任务中的多个关键步骤。

### 程序整体功能和构架概括  
  
该工程包含多个模块，主要用于实现和验证基于深度学习的目标检测模型。整体架构设计灵活且模块化，允许不同的组件（如卷积层、变换器、验证器等）独立开发和测试。以下是各个文件的主要功能：  
  
1. \*\*mamba\_yolo.py\*\*：实现了YOLO目标检测模型的核心结构，结合了选择性扫描和卷积操作，以提高目标检测的性能。  
2. \*\*CTrans.py\*\*：实现了通道变换器（Channel Transformer），用于图像特征提取和重建，结合了卷积神经网络和变换器的优点。  
3. \*\*kagn\_conv.py\*\*：定义了KAGN卷积层，支持多维卷积操作，通过引入多项式基函数增强卷积的表达能力。  
4. \*\*val.py\*\*：实现了YOLO模型的验证过程，包括数据预处理、指标计算、结果输出和可视化，确保模型性能的评估。  
  
### 文件功能整理表  
  
| 文件名 | 功能描述 |  
|-------------------|-------------------------------------------------------------------------------------------|  
| `mamba\_yolo.py` | 实现YOLO目标检测模型的核心结构，结合选择性扫描和卷积操作，优化目标检测性能。 |  
| `CTrans.py` | 实现通道变换器（Channel Transformer），用于图像特征提取和重建，结合卷积和变换器的优点。 |  
| `kagn\_conv.py` | 定义KAGN卷积层，支持多维卷积操作，通过多项式基函数增强卷积表达能力。 |  
| `val.py` | 实现YOLO模型的验证过程，包括数据预处理、指标计算、结果输出和可视化，评估模型性能。 |  
  
这个表格清晰地总结了每个文件的主要功能，便于理解整个工程的构架和各个模块之间的关系。

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。