# 改进yolo11-FocusedLinearAttention等200+全套创新点大全：【无人零售】自动售卖机饮料检测系统源码＆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  
  
随着科技的迅猛发展和消费者购物习惯的改变，无人零售逐渐成为现代商业的一种新兴模式。无人零售自动售卖机以其便捷性和高效性，受到了越来越多消费者的青睐。然而，自动售卖机在商品管理和库存监控方面仍面临诸多挑战，尤其是在饮料产品的实时检测与识别方面。为了解决这一问题，基于改进YOLOv11的饮料检测系统应运而生。  
  
YOLO（You Only Look Once）系列模型以其快速且准确的目标检测能力而广泛应用于各种计算机视觉任务。YOLOv11作为该系列的最新版本，进一步提升了检测精度和速度，适合在资源有限的自动售卖机环境中进行实时处理。针对饮料产品的检测，研究团队选取了包含2200张图像的“soda bottles”数据集，数据集中涵盖了多种知名饮料品牌，如可口可乐、芬达和雪碧等。该数据集的单一类别特性，使得模型在训练过程中能够集中精力于特定对象的识别，从而提高检测的准确性。  
  
本研究的意义在于，通过改进YOLOv11模型，构建一个高效的饮料检测系统，不仅能够实时监控自动售卖机内的饮料库存，还能通过数据分析优化补货策略，提升运营效率。此外，该系统的成功实施将为无人零售行业提供一种可行的技术解决方案，推动无人零售的进一步发展。同时，基于深度学习的目标检测技术在其他领域的应用也将得到启示，具有广泛的推广价值。通过本项目的研究，期望能够为无人零售行业的智能化转型提供坚实的技术支持。

## 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“soda bottles”，专门用于训练和改进YOLOv11模型，以实现无人零售自动售卖机中的饮料检测系统。该数据集包含三种主要类别的饮料，分别是可口可乐（coca-cola）、芬达（fanta）和雪碧（sprite）。这些类别的选择不仅反映了市场上最受欢迎的饮料品牌，也为模型的训练提供了丰富的样本，确保其在实际应用中的准确性和可靠性。  
  
数据集的构建过程经过精心设计，确保每个类别的样本数量均衡且多样化，以便模型能够有效学习到不同饮料的特征。数据集中包含了各种环境下的饮料图像，包括不同的光照条件、拍摄角度和背景，这些因素都可能影响模型的识别能力。因此，数据集中的图像涵盖了多种场景，以增强模型的泛化能力，确保其在实际应用中能够准确识别和分类。  
  
此外，数据集还经过了详细的标注，每张图像中的饮料瓶都被精确框定，提供了必要的位置信息。这一过程不仅提高了数据集的质量，也为YOLOv11模型的训练提供了坚实的基础。通过使用“soda bottles”数据集，我们期望能够提升无人零售自动售卖机的饮料检测精度，从而优化消费者的购物体验，推动无人零售技术的发展。  
  
综上所述，“soda bottles”数据集为本项目提供了丰富的训练素材和高质量的标注信息，确保了模型在实际应用中的有效性和准确性，为无人零售领域的创新与发展奠定了坚实的基础。

## 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  
  
class KANConvNDLayer(nn.Module):  
 def \_\_init\_\_(self, conv\_class, norm\_class, input\_dim, output\_dim, spline\_order, kernel\_size,  
 groups=1, padding=0, stride=1, dilation=1,  
 ndim: int = 2, grid\_size=5, base\_activation=nn.GELU, grid\_range=[-1, 1], dropout=0.0):  
 super(KANConvNDLayer, self).\_\_init\_\_()  
   
 # 初始化参数  
 self.inputdim = input\_dim # 输入维度  
 self.outdim = output\_dim # 输出维度  
 self.spline\_order = spline\_order # 样条的阶数  
 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 # 分组数  
 self.ndim = ndim # 维度（1D, 2D, 3D）  
 self.grid\_size = grid\_size # 网格大小  
 self.base\_activation = base\_activation() # 基础激活函数  
 self.grid\_range = grid\_range # 网格范围  
  
 # 初始化dropout层  
 self.dropout = None  
 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)  
  
 # 检查分组参数的有效性  
 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')  
  
 # 初始化基础卷积层  
 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)])  
  
 # 初始化样条卷积层  
 self.spline\_conv = nn.ModuleList([conv\_class((grid\_size + spline\_order) \* input\_dim // groups,  
 output\_dim // groups,  
 kernel\_size,  
 stride,  
 padding,  
 dilation,  
 groups=1,  
 bias=False) for \_ in range(groups)])  
  
 # 初始化归一化层  
 self.layer\_norm = nn.ModuleList([norm\_class(output\_dim // groups) for \_ in range(groups)])  
  
 # 初始化PReLU激活函数  
 self.prelus = nn.ModuleList([nn.PReLU() for \_ in range(groups)])  
  
 # 生成样条网格  
 h = (self.grid\_range[1] - self.grid\_range[0]) / grid\_size  
 self.grid = torch.linspace(  
 self.grid\_range[0] - h \* spline\_order,  
 self.grid\_range[1] + h \* spline\_order,  
 grid\_size + 2 \* spline\_order + 1,  
 dtype=torch.float32  
 )  
  
 # 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')  
  
 for conv\_layer in self.spline\_conv:  
 nn.init.kaiming\_uniform\_(conv\_layer.weight, nonlinearity='linear')  
  
 def forward\_kan(self, x, group\_index):  
 # 基础卷积的前向传播  
 base\_output = self.base\_conv[group\_index](self.base\_activation(x))  
  
 # 扩展输入维度以进行样条操作  
 x\_uns = x.unsqueeze(-1)  
 target = x.shape[1:] + self.grid.shape  
 grid = self.grid.view(\*list([1 for \_ in range(self.ndim + 1)] + [-1, ])).expand(target).contiguous().to(x.device)  
  
 # 计算样条基  
 bases = ((x\_uns >= grid[..., :-1]) & (x\_uns < grid[..., 1:])).to(x.dtype)  
  
 # 计算多阶样条基  
 for k in range(1, self.spline\_order + 1):  
 left\_intervals = grid[..., :-(k + 1)]  
 right\_intervals = grid[..., k:-1]  
 delta = torch.where(right\_intervals == left\_intervals, torch.ones\_like(right\_intervals),  
 right\_intervals - left\_intervals)  
 bases = ((x\_uns - left\_intervals) / delta \* bases[..., :-1]) + \  
 ((grid[..., k + 1:] - x\_uns) / (grid[..., k + 1:] - grid[..., 1:(-k)]) \* bases[..., 1:])  
 bases = bases.contiguous()  
 bases = bases.moveaxis(-1, 2).flatten(1, 2)  
  
 # 通过样条卷积层进行前向传播  
 spline\_output = self.spline\_conv[group\_index](bases)  
 x = self.prelus[group\_index](self.layer\_norm[group\_index](base\_output + spline\_output))  
  
 # 应用dropout  
 if self.dropout is not None:  
 x = self.dropout(x)  
  
 return x  
  
 def forward(self, x):  
 # 将输入按组分割并进行前向传播  
 split\_x = torch.split(x, self.inputdim // self.groups, dim=1)  
 output = []  
 for group\_ind, \_x in enumerate(split\_x):  
 y = self.forward\_kan(\_x.clone(), group\_ind)  
 output.append(y.clone())  
 y = torch.cat(output, dim=1) # 合并输出  
 return y  
  
# KANConv3DLayer, KANConv2DLayer, KANConv1DLayer 类分别继承自 KANConvNDLayer  
# 这些类的构造函数只需传入相应的卷积和归一化类  
class KANConv3DLayer(KANConvNDLayer):  
 def \_\_init\_\_(self, input\_dim, output\_dim, kernel\_size, spline\_order=3, groups=1, padding=0, stride=1, dilation=1,  
 grid\_size=5, base\_activation=nn.GELU, grid\_range=[-1, 1], dropout=0.0):  
 super(KANConv3DLayer, self).\_\_init\_\_(nn.Conv3d, nn.InstanceNorm3d,  
 input\_dim, output\_dim,  
 spline\_order, kernel\_size,  
 groups=groups, padding=padding, stride=stride, dilation=dilation,  
 ndim=3,  
 grid\_size=grid\_size, base\_activation=base\_activation,  
 grid\_range=grid\_range, dropout=dropout)  
  
class KANConv2DLayer(KANConvNDLayer):  
 def \_\_init\_\_(self, input\_dim, output\_dim, kernel\_size, spline\_order=3, groups=1, padding=0, stride=1, dilation=1,  
 grid\_size=5, base\_activation=nn.GELU, grid\_range=[-1, 1], dropout=0.0):  
 super(KANConv2DLayer, self).\_\_init\_\_(nn.Conv2d, nn.InstanceNorm2d,  
 input\_dim, output\_dim,  
 spline\_order, kernel\_size,  
 groups=groups, padding=padding, stride=stride, dilation=dilation,  
 ndim=2,  
 grid\_size=grid\_size, base\_activation=base\_activation,  
 grid\_range=grid\_range, dropout=dropout)  
  
class KANConv1DLayer(KANConvNDLayer):  
 def \_\_init\_\_(self, input\_dim, output\_dim, kernel\_size, spline\_order=3, groups=1, padding=0, stride=1, dilation=1,  
 grid\_size=5, base\_activation=nn.GELU, grid\_range=[-1, 1], dropout=0.0):  
 super(KANConv1DLayer, self).\_\_init\_\_(nn.Conv1d, nn.InstanceNorm1d,  
 input\_dim, output\_dim,  
 spline\_order, kernel\_size,  
 groups=groups, padding=padding, stride=stride, dilation=dilation,  
 ndim=1,  
 grid\_size=grid\_size, base\_activation=base\_activation,  
 grid\_range=grid\_range, dropout=dropout)  
```  
  
### 代码核心部分说明：  
1. \*\*KANConvNDLayer\*\*: 这是一个通用的N维卷积层，支持1D、2D和3D卷积。它结合了基础卷积和样条卷积，并提供了归一化和激活功能。  
2. \*\*forward\_kan\*\*: 这个方法实现了卷积层的前向传播，包括基础卷积和样条卷积的计算。  
3. \*\*网格生成\*\*: 在初始化时生成样条网格，用于后续的样条基计算。  
4. \*\*分组卷积\*\*: 支持分组卷积，允许在多个组上并行处理输入。  
5. \*\*子类\*\*: `KANConv3DLayer`、`KANConv2DLayer`和`KANConv1DLayer`分别针对3D、2D和1D卷积的具体实现，简化了构造函数的调用。  
  
通过这些核心部分，用户可以构建复杂的卷积神经网络，利用样条卷积的优势来提高模型的表现。```

这个文件定义了一个名为 `KANConvNDLayer` 的神经网络层，主要用于实现一种基于样条（spline）插值的卷积操作。该层支持多维输入（如1D、2D、3D），并且可以根据需要进行分组卷积。下面是对代码的逐步解析。  
  
首先，`KANConvNDLayer` 类继承自 `torch.nn.Module`，构造函数接受多个参数，包括卷积类型、归一化类型、输入和输出维度、样条的阶数、卷积核大小、分组数、填充、步幅、扩张率、维度数量、网格大小、激活函数、网格范围和丢弃率等。构造函数中，首先对这些参数进行初始化，并检查一些有效性条件，比如分组数必须为正且输入和输出维度必须能被分组数整除。  
  
接下来，类中定义了几个重要的模块列表：`base\_conv` 和 `spline\_conv` 分别用于基础卷积和样条卷积，这些卷积层是通过 `conv\_class` 创建的。`layer\_norm` 列表用于对每个分组的输出进行归一化，`prelus` 列表则包含了 PReLU 激活函数。网格（grid）用于样条插值的计算，初始化时根据给定的范围和大小生成。  
  
在 `forward\_kan` 方法中，首先对输入进行基础激活处理，然后通过基础卷积层进行线性变换。接着，输入的维度被扩展，以便进行样条操作。样条基函数的计算涉及到输入值与网格的比较，并通过多个阶数的样条进行插值计算。最终，样条卷积的输出与基础卷积的输出相加，经过归一化和激活函数处理后，如果设置了丢弃率，则应用丢弃层。  
  
`forward` 方法则是将输入按照分组进行切分，并对每个分组调用 `forward\_kan` 方法进行处理，最后将所有分组的输出拼接在一起。  
  
此外，文件中还定义了三个子类：`KANConv3DLayer`、`KANConv2DLayer` 和 `KANConv1DLayer`，分别对应于三维、二维和一维卷积。这些子类通过调用父类的构造函数，指定相应的卷积和归一化类型。  
  
整体来看，这个文件实现了一种灵活且强大的卷积层，结合了样条插值和深度学习中的卷积操作，适用于多种维度的数据处理任务。

```以下是代码中最核心的部分，并附上详细的中文注释：  
  
```python  
from ultralytics.engine.predictor import BasePredictor  
from ultralytics.engine.results import Results  
from ultralytics.utils import ops  
  
class DetectionPredictor(BasePredictor):  
 """  
 DetectionPredictor类，继承自BasePredictor类，用于基于检测模型进行预测。  
 """  
  
 def postprocess(self, preds, img, orig\_imgs):  
 """对预测结果进行后处理，并返回Results对象的列表。"""  
   
 # 使用非极大值抑制（NMS）来过滤预测框  
 preds = ops.non\_max\_suppression(  
 preds, # 预测框  
 self.args.conf, # 置信度阈值  
 self.args.iou, # IOU阈值  
 agnostic=self.args.agnostic\_nms, # 是否进行类别无关的NMS  
 max\_det=self.args.max\_det, # 最大检测框数量  
 classes=self.args.classes, # 需要检测的类别  
 )  
  
 # 如果输入的原始图像不是列表，则将其转换为numpy数组  
 if not isinstance(orig\_imgs, list): # 输入图像是torch.Tensor，而不是列表  
 orig\_imgs = ops.convert\_torch2numpy\_batch(orig\_imgs) # 转换为numpy数组  
  
 results = [] # 初始化结果列表  
 for i, pred in enumerate(preds): # 遍历每个预测结果  
 orig\_img = orig\_imgs[i] # 获取对应的原始图像  
 # 将预测框的坐标从缩放后的图像尺寸转换为原始图像尺寸  
 pred[:, :4] = ops.scale\_boxes(img.shape[2:], pred[:, :4], orig\_img.shape)  
 img\_path = self.batch[0][i] # 获取图像路径  
 # 将结果添加到结果列表中  
 results.append(Results(orig\_img, path=img\_path, names=self.model.names, boxes=pred))  
   
 return results # 返回处理后的结果列表  
```  
  
### 代码注释说明：  
1. \*\*导入模块\*\*：导入必要的模块和类，以便进行预测和结果处理。  
2. \*\*DetectionPredictor类\*\*：该类专门用于处理检测模型的预测，继承自`BasePredictor`。  
3. \*\*postprocess方法\*\*：该方法负责对模型的预测结果进行后处理，包括应用非极大值抑制（NMS）和调整预测框的坐标。  
4. \*\*非极大值抑制\*\*：通过`ops.non\_max\_suppression`函数来减少重叠的预测框，只保留最有可能的框。  
5. \*\*图像转换\*\*：如果原始图像不是列表格式，则将其转换为numpy数组，以便后续处理。  
6. \*\*结果构建\*\*：遍历每个预测结果，调整框的坐标，并将每个结果封装成`Results`对象，最终返回所有结果的列表。```

这个程序文件 `predict.py` 是一个用于目标检测的预测模块，基于 Ultralytics YOLO（You Only Look Once）模型构建。文件中定义了一个名为 `DetectionPredictor` 的类，该类继承自 `BasePredictor`，用于处理目标检测模型的预测。  
  
在类的文档字符串中，提供了一个使用示例，展示了如何从 `ultralytics.utils` 导入必要的模块，并创建一个 `DetectionPredictor` 实例。示例中，模型文件 `yolov8n.pt` 和数据源 `ASSETS` 被传递给 `DetectionPredictor`，然后调用 `predict\_cli()` 方法进行预测。  
  
类中定义了一个名为 `postprocess` 的方法，该方法用于对模型的预测结果进行后处理。具体来说，它接受三个参数：`preds`（模型的预测结果）、`img`（输入图像）和 `orig\_imgs`（原始图像）。在方法内部，首先调用 `ops.non\_max\_suppression` 函数对预测结果进行非极大值抑制，以减少冗余的检测框。这个过程使用了一些参数，如置信度阈值、IOU（Intersection over Union）阈值、是否进行类别无关的非极大值抑制、最大检测框数量以及需要检测的类别。  
  
接下来，方法检查 `orig\_imgs` 是否为列表。如果不是，说明输入图像是一个 PyTorch 张量，此时调用 `ops.convert\_torch2numpy\_batch` 将其转换为 NumPy 数组格式。  
  
然后，方法会遍历每个预测结果，调整检测框的坐标，使其适应原始图像的尺寸。具体地，使用 `ops.scale\_boxes` 函数来缩放检测框的坐标。每个预测结果和对应的原始图像路径会被封装成 `Results` 对象，并添加到结果列表中。  
  
最后，`postprocess` 方法返回一个包含所有处理后结果的列表。这个类和方法的设计使得用户能够方便地对目标检测模型的输出进行处理，并提取出有用的信息。

```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 build\_selective\_scan\_fn(selective\_scan\_cuda: object = None, mode="mamba\_ssm", tag=None):  
 """  
 构建选择性扫描函数的工厂函数。  
   
 参数:  
 selective\_scan\_cuda: CUDA实现的选择性扫描函数  
 mode: 模式选择  
 tag: 标签，用于标识不同的选择性扫描实现  
 """  
   
 class SelectiveScanFn(torch.autograd.Function):  
 @staticmethod  
 def forward(ctx, u, delta, A, B, C, D=None, z=None, delta\_bias=None, delta\_softplus=False, return\_last\_state=False, nrows=1, backnrows=-1):  
 """  
 前向传播函数，执行选择性扫描操作。  
   
 参数:  
 ctx: 上下文对象，用于保存状态  
 u: 输入张量  
 delta: 增量张量  
 A, B, C: 权重张量  
 D: 可选的偏置张量  
 z: 可选的张量  
 delta\_bias: 可选的增量偏置  
 delta\_softplus: 是否应用softplus激活  
 return\_last\_state: 是否返回最后状态  
 nrows: 行数  
 backnrows: 回溯行数  
   
 返回:  
 输出张量或输出张量和最后状态的元组  
 """  
 # 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:  
 D = D.contiguous()  
 if B.stride(-1) != 1:  
 B = B.contiguous()  
 if C.stride(-1) != 1:  
 C = C.contiguous()  
 if z is not None and z.stride(-1) != 1:  
 z = z.contiguous()  
  
 # 处理张量的维度和形状  
 if B.dim() == 3:  
 B = rearrange(B, "b dstate l -> b 1 dstate l")  
 ctx.squeeze\_B = True  
 if C.dim() == 3:  
 C = rearrange(C, "b dstate l -> b 1 dstate l")  
 ctx.squeeze\_C = True  
  
 # 检查输入形状的有效性  
 assert u.shape[1] % (B.shape[1] \* nrows) == 0   
 assert nrows in [1, 2, 3, 4] # 限制行数为1到4  
  
 if backnrows > 0:  
 assert u.shape[1] % (B.shape[1] \* backnrows) == 0   
 assert backnrows in [1, 2, 3, 4] # 限制回溯行数为1到4  
 else:  
 backnrows = nrows  
 ctx.backnrows = backnrows  
   
 # 调用CUDA实现的选择性扫描前向函数  
 if mode in ["mamba\_ssm"]:  
 out, x, \*rest = selective\_scan\_cuda.fwd(u, delta, A, B, C, D, z, delta\_bias, delta\_softplus)  
 else:  
 raise NotImplementedError  
  
 # 保存状态以便反向传播  
 ctx.delta\_softplus = delta\_softplus  
 ctx.has\_z = z is not None  
 last\_state = x[:, :, -1, 1::2] # 获取最后状态  
   
 # 返回输出或输出和最后状态  
 ctx.save\_for\_backward(u, delta, A, B, C, D, delta\_bias, x)  
 return out if not return\_last\_state else (out, last\_state)  
  
 @staticmethod  
 def backward(ctx, dout, \*args):  
 """  
 反向传播函数，计算梯度。  
   
 参数:  
 ctx: 上下文对象  
 dout: 输出的梯度  
   
 返回:  
 输入张量的梯度  
 """  
 u, delta, A, B, C, D, delta\_bias, x = ctx.saved\_tensors  
   
 # 调用CUDA实现的选择性扫描反向函数  
 du, ddelta, dA, dB, dC, dD, ddelta\_bias, \*rest = selective\_scan\_cuda.bwd(  
 u, delta, A, B, C, D, delta\_bias, dout, x, ctx.delta\_softplus, ctx.backnrows  
 )  
   
 return (du, ddelta, dA, dB, dC, dD if D is not None else None, None, ddelta\_bias if delta\_bias is not None else None)  
  
 def selective\_scan\_fn(u, delta, A, B, C, D=None, z=None, delta\_bias=None, delta\_softplus=False, return\_last\_state=False, nrows=1, backnrows=-1):  
 """  
 封装选择性扫描函数的调用。  
 """  
 return SelectiveScanFn.apply(u, delta, A, B, C, D, z, delta\_bias, delta\_softplus, return\_last\_state, nrows, backnrows)  
  
 return selective\_scan\_fn  
```  
  
### 代码说明：  
1. \*\*导入必要的库\*\*：导入了PyTorch和其功能模块。  
2. \*\*`build\_selective\_scan\_fn`函数\*\*：构建选择性扫描函数的工厂函数，接收CUDA实现和模式作为参数。  
3. \*\*`SelectiveScanFn`类\*\*：继承自`torch.autograd.Function`，实现了前向和反向传播的方法。  
 - \*\*`forward`方法\*\*：执行选择性扫描的前向计算，处理输入张量的形状和连续性，调用CUDA实现的前向函数，并保存必要的状态以供反向传播使用。  
 - \*\*`backward`方法\*\*：计算梯度，调用CUDA实现的反向函数，并返回输入张量的梯度。  
4. \*\*`selective\_scan\_fn`函数\*\*：封装了对`SelectiveScanFn`的调用，提供了一个简单的接口。  
  
该代码实现了选择性扫描的前向和反向传播功能，适用于深度学习中的动态计算图。```

这个程序文件 `test\_selective\_scan\_speed.py` 是一个用于测试选择性扫描（Selective Scan）算法性能的脚本，主要依赖于 PyTorch 库进行深度学习计算。代码中包含了选择性扫描的前向和反向传播实现，以及对不同实现的性能测试。  
  
首先，文件导入了必要的库，包括 `torch`、`torch.nn.functional`、`pytest`、`time` 等。接着，定义了一个构建选择性扫描函数的工厂函数 `build\_selective\_scan\_fn`，该函数接收一个 CUDA 实现的选择性扫描函数和其他参数，返回一个自定义的 PyTorch 函数 `selective\_scan\_fn`。这个函数内部定义了一个 `SelectiveScanFn` 类，继承自 `torch.autograd.Function`，实现了前向传播和反向传播的方法。  
  
在 `SelectiveScanFn` 的 `forward` 方法中，首先对输入张量进行处理，确保它们是连续的，并根据需要调整维度。接着，依据不同的模式（如 `mamba\_ssm`、`sscore` 等）调用相应的 CUDA 实现进行前向计算。最后，保存必要的中间结果以供反向传播使用，并返回输出结果。  
  
`backward` 方法则实现了反向传播的逻辑，计算梯度并返回给定的输入张量的梯度。  
  
文件中还定义了几个选择性扫描的参考实现函数 `selective\_scan\_ref`、`selective\_scan\_easy` 和 `selective\_scan\_easy\_v2`，这些函数实现了选择性扫描的核心算法，支持不同的输入格式和参数设置。  
  
在 `test\_speed` 函数中，设置了一系列参数，包括数据类型、序列长度、批量大小等。接着，生成了测试所需的随机输入数据，并使用不同的选择性扫描实现进行性能测试。通过记录每个实现的运行时间，比较它们的性能。  
  
最后，程序调用 `test\_speed` 函数开始性能测试，并输出每个实现的前向和前向反向计算的时间。这种测试可以帮助开发者了解不同实现的效率，并在实际应用中选择最佳的实现方案。

```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  
  
class KACNConvNDLayer(nn.Module):  
 def \_\_init\_\_(self, conv\_class, norm\_class, input\_dim, output\_dim, degree, kernel\_size,  
 groups=1, padding=0, stride=1, dilation=1,  
 ndim: int = 2, dropout=0.0):  
 super(KACNConvNDLayer, 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.ndim = ndim # 数据的维度（1D, 2D, 3D）  
 self.dropout = None # 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)  
  
 # 检查分组参数的有效性  
 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')  
  
 # 创建归一化层  
 self.layer\_norm = nn.ModuleList([norm\_class(output\_dim // groups) for \_ in range(groups)])  
  
 # 创建多项式卷积层  
 self.poly\_conv = nn.ModuleList([conv\_class((degree + 1) \* input\_dim // groups,  
 output\_dim // groups,  
 kernel\_size,  
 stride,  
 padding,  
 dilation,  
 groups=1,  
 bias=False) for \_ in range(groups)])  
   
 # 创建一个缓冲区，用于存储多项式的系数  
 arange\_buffer\_size = (1, 1, -1,) + tuple(1 for \_ in range(ndim))  
 self.register\_buffer("arange", torch.arange(0, degree + 1, 1).view(\*arange\_buffer\_size))  
   
 # 使用Kaiming均匀分布初始化卷积层的权重  
 for conv\_layer in self.poly\_conv:  
 nn.init.normal\_(conv\_layer.weight, mean=0.0, std=1 / (input\_dim \* (degree + 1) \* kernel\_size \*\* ndim))  
  
 def forward\_kacn(self, x, group\_index):  
 # 前向传播，处理每个组的输入  
 x = torch.tanh(x) # 应用tanh激活函数  
 x = x.acos().unsqueeze(2) # 计算反余弦并增加一个维度  
 x = (x \* self.arange).flatten(1, 2) # 乘以多项式系数并展平  
 x = x.cos() # 计算余弦  
 x = self.poly\_conv[group\_index](x) # 通过对应的卷积层  
 x = self.layer\_norm[group\_index](x) # 归一化  
 if self.dropout is not None:  
 x = self.dropout(x) # 应用Dropout  
 return x  
  
 def forward(self, x):  
 # 前向传播，处理整个输入  
 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\_kacn(\_x.clone(), group\_ind) # 对每个组调用forward\_kacn  
 output.append(y.clone()) # 收集输出  
 y = torch.cat(output, dim=1) # 合并所有组的输出  
 return y  
```  
  
### 代码核心部分说明：  
1. \*\*KACNConvNDLayer类\*\*：这是一个自定义的卷积层，支持多维卷积（1D、2D、3D），并结合了多项式卷积和归一化。  
2. \*\*初始化方法\*\*：初始化卷积层、归一化层和Dropout层，并进行参数有效性检查。  
3. \*\*forward\_kacn方法\*\*：实现了对输入的前向传播，包含了激活函数、卷积和归一化等操作。  
4. \*\*forward方法\*\*：处理整个输入，将其分割为多个组，并对每个组调用`forward\_kacn`进行处理，最后合并输出。```

这个程序文件定义了一个名为 `kacn\_conv.py` 的模块，主要实现了一个新的卷积层 `KACNConvNDLayer` 及其针对不同维度的特化版本（1D、2D 和 3D）。该模块使用 PyTorch 框架构建深度学习模型。  
  
首先，`KACNConvNDLayer` 类是一个自定义的神经网络层，继承自 `nn.Module`。在初始化方法中，用户可以传入卷积类、归一化类、输入和输出维度、卷积的次数（degree）、卷积核大小等参数。该类还支持分组卷积（groups）、填充（padding）、步幅（stride）、扩张（dilation）以及 dropout 的设置。初始化过程中，程序会检查分组数是否为正整数，并确保输入和输出维度可以被分组数整除。  
  
在初始化时，`KACNConvNDLayer` 会创建多个卷积层和归一化层的实例，存储在 `ModuleList` 中，以便后续使用。卷积层的权重会使用 Kaiming 正态分布进行初始化，以便于模型的训练。  
  
`forward\_kacn` 方法实现了该层的前向传播逻辑。首先对输入进行非线性激活（使用双曲正切函数），然后通过反余弦函数处理，并进行维度变换。接着，使用指定的卷积层对处理后的数据进行卷积操作，并应用归一化层。若设置了 dropout，则会在此处应用。  
  
`forward` 方法负责处理输入数据，首先将输入按照分组进行拆分，然后对每个分组调用 `forward\_kacn` 方法进行处理，最后将所有分组的输出拼接在一起，形成最终的输出。  
  
接下来，`KACNConv3DLayer`、`KACNConv2DLayer` 和 `KACNConv1DLayer` 类分别是 `KACNConvNDLayer` 的特化版本，分别用于处理三维、二维和一维的卷积操作。它们在初始化时调用父类的构造函数，并传入相应的卷积和归一化类（如 `nn.Conv3d`、`nn.Conv2d` 和 `nn.Conv1d`）以及其他参数。  
  
总体而言，这个模块实现了一个灵活且可扩展的卷积层设计，支持多种维度的卷积操作，并结合了归一化和 dropout 机制，适用于构建复杂的深度学习模型。

### 整体功能和构架概括  
  
该项目包含多个模块，主要用于实现深度学习中的卷积操作、目标检测预测以及性能测试。具体来说，项目的结构包括自定义卷积层的实现、目标检测的预测功能，以及选择性扫描算法的性能测试。每个模块都使用 PyTorch 框架构建，具有良好的可扩展性和灵活性。  
  
- \*\*kan\_conv.py\*\*：实现了基于样条插值的卷积层，支持多维输入，适用于复杂的卷积操作。  
- \*\*predict.py\*\*：实现了目标检测的预测功能，使用 YOLO 模型，能够对输入图像进行目标检测并返回结果。  
- \*\*test\_selective\_scan\_speed.py\*\*：用于测试选择性扫描算法的性能，比较不同实现的运行时间，帮助开发者选择最佳实现。  
- \*\*kacn\_conv.py\*\*：实现了一个新的卷积层，支持多维卷积操作，结合了归一化和激活机制，适用于深度学习模型的构建。  
  
### 文件功能整理表  
  
| 文件名 | 功能描述 |  
|---------------------------------|----------------------------------------------------------------------------------------------|  
| `kan\_conv.py` | 实现基于样条插值的卷积层 `KANConvNDLayer`，支持多维输入和分组卷积，结合归一化和激活机制。 |  
| `predict.py` | 实现目标检测的预测功能，使用 YOLO 模型对输入图像进行目标检测，并返回处理后的结果。 |  
| `test\_selective\_scan\_speed.py` | 测试选择性扫描算法的性能，比较不同实现的前向和反向传播运行时间，帮助优化算法选择。 |  
| `kacn\_conv.py` | 实现新的卷积层 `KACNConvNDLayer`，支持多维卷积操作，结合归一化和 dropout 机制，适用于深度学习。 |  
  
这些模块共同构成了一个功能丰富的深度学习框架，能够处理卷积操作、目标检测和性能测试等任务。

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。