# 改进yolo11-GhostDynamicConv等200+全套创新点大全：家庭门窗开闭状态安全监控系统源码＆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  
  
随着智能家居技术的迅速发展，家庭安全监控系统逐渐成为现代家庭生活中不可或缺的一部分。传统的安全监控手段往往依赖于人工监控或简单的报警系统，难以实现实时、智能化的安全防护。近年来，基于深度学习的目标检测技术，尤其是YOLO（You Only Look Once）系列模型，因其高效性和准确性而受到广泛关注。YOLOv11作为该系列的最新版本，具有更快的检测速度和更高的精度，适合应用于家庭安全监控领域。  
  
本研究旨在基于改进的YOLOv11模型，开发一套家庭门窗开闭状态的安全监控系统。该系统将通过实时监测家庭门窗的状态，及时识别潜在的安全隐患，提升家庭安全防护能力。我们所使用的数据集包含5900张图像，涵盖了三类主要目标：门的开闭状态、窗户的开闭状态以及气体设备的开关状态。这些类别不仅包括门窗的开闭状态（door\_closed、door\_opened、window\_closed、window\_opened），还涉及到家庭日常生活中可能存在的安全隐患，如气体设备的开启与关闭（gas\_turn\_on、gas\_turn\_off）以及冰箱的状态（refrigerator\_closed、refrigerator\_opened）。通过对这些状态的监测，系统能够及时发出警报，防止意外事故的发生。  
  
此外，家庭安全监控系统的智能化升级，不仅能够提升家庭成员的安全感，还能有效降低因人为疏忽而导致的财产损失。通过引入深度学习技术，我们期望能够实现更高效的监测与识别，推动家庭安全监控技术的进一步发展。因此，本研究不仅具有重要的理论意义，还有着广泛的实际应用前景，为智能家居的安全防护提供了新的解决方案。

## 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“Security-Detection”，旨在为改进YOLOv11的家庭门窗开闭状态安全监控系统提供强有力的支持。该数据集共包含9个类别，涵盖了家庭安全监控中常见的状态，具体包括“Fall\_Detected”（跌倒检测）、“door\_closed”（门关闭）、“door\_opened”（门打开）、“gas\_turn\_off”（燃气关闭）、“gas\_turn\_on”（燃气开启）、“refrigerator\_closed”（冰箱关闭）、“refrigerator\_opened”（冰箱打开）、“window\_closed”（窗户关闭）以及“window\_opened”（窗户打开）。这些类别的设计充分考虑了家庭日常生活中的安全隐患，旨在通过智能监控系统及时识别和响应潜在的危险。  
  
数据集中的每个类别均经过精心标注，确保模型在训练过程中能够准确学习到不同状态的特征。跌倒检测类别尤其重要，因为它直接关系到家庭成员的安全，尤其是老年人。门窗和燃气的状态监控则是家庭安全的基础，及时的状态变化检测能够有效预防事故的发生。冰箱的开闭状态监控不仅有助于食品安全，还能通过智能化管理提升家庭生活的便利性。  
  
在数据集的构建过程中，考虑到了多样性和真实场景的复杂性，数据来源于不同的家庭环境，以确保模型的泛化能力。通过对这些状态的有效监测，改进后的YOLOv11模型将能够实时分析家庭环境的安全状况，提供及时的警报和反馈，从而大幅提升家庭安全防护的智能化水平。整体而言，“Security-Detection”数据集为本项目的成功实施奠定了坚实的基础，期待通过这一数据集的应用，推动家庭安全监控技术的进一步发展。

## 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，保留了最重要的结构和功能：  
  
```python  
import torch  
import torch.nn as nn  
from functools import partial  
  
class Mlp(nn.Module):  
 """ 多层感知机模块，包含两个卷积层和一个深度卷积层 """  
 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.Conv2d(in\_features, hidden\_features, 1) # 第一个卷积层  
 self.dwconv = DWConv(hidden\_features) # 深度卷积层  
 self.act = act\_layer() # 激活函数  
 self.fc2 = nn.Conv2d(hidden\_features, out\_features, 1) # 第二个卷积层  
 self.drop = nn.Dropout(drop) # Dropout层  
  
 def forward(self, x):  
 x = self.fc1(x)  
 x = self.dwconv(x)  
 x = self.act(x)  
 x = self.drop(x)  
 x = self.fc2(x)  
 x = self.drop(x)  
 return x  
  
  
class LSKblock(nn.Module):  
 """ LSK模块，包含空间卷积和注意力机制 """  
 def \_\_init\_\_(self, dim):  
 super().\_\_init\_\_()  
 self.conv0 = nn.Conv2d(dim, dim, 5, padding=2, groups=dim) # 深度卷积  
 self.conv\_spatial = nn.Conv2d(dim, dim, 7, stride=1, padding=9, groups=dim, dilation=3) # 空间卷积  
 self.conv1 = nn.Conv2d(dim, dim//2, 1) # 1x1卷积  
 self.conv2 = nn.Conv2d(dim, dim//2, 1) # 1x1卷积  
 self.conv\_squeeze = nn.Conv2d(2, 2, 7, padding=3) # 压缩卷积  
 self.conv = nn.Conv2d(dim//2, dim, 1) # 1x1卷积  
  
 def forward(self, x):   
 attn1 = self.conv0(x) # 通过深度卷积  
 attn2 = self.conv\_spatial(attn1) # 通过空间卷积  
  
 attn1 = self.conv1(attn1) # 1x1卷积  
 attn2 = self.conv2(attn2) # 1x1卷积  
   
 attn = torch.cat([attn1, attn2], dim=1) # 拼接  
 avg\_attn = torch.mean(attn, dim=1, keepdim=True) # 平均注意力  
 max\_attn, \_ = torch.max(attn, dim=1, keepdim=True) # 最大注意力  
 agg = torch.cat([avg\_attn, max\_attn], dim=1) # 拼接平均和最大注意力  
 sig = self.conv\_squeeze(agg).sigmoid() # 压缩并激活  
 attn = attn1 \* sig[:,0,:,:].unsqueeze(1) + attn2 \* sig[:,1,:,:].unsqueeze(1) # 加权组合  
 attn = self.conv(attn) # 最终卷积  
 return x \* attn # 乘以输入  
  
  
class Attention(nn.Module):  
 """ 注意力模块 """  
 def \_\_init\_\_(self, d\_model):  
 super().\_\_init\_\_()  
 self.proj\_1 = nn.Conv2d(d\_model, d\_model, 1) # 投影层  
 self.activation = nn.GELU() # 激活函数  
 self.spatial\_gating\_unit = LSKblock(d\_model) # LSK模块  
 self.proj\_2 = nn.Conv2d(d\_model, d\_model, 1) # 投影层  
  
 def forward(self, x):  
 shortcut = x.clone() # 残差连接  
 x = self.proj\_1(x)  
 x = self.activation(x)  
 x = self.spatial\_gating\_unit(x)  
 x = self.proj\_2(x)  
 x = x + shortcut # 残差连接  
 return x  
  
  
class Block(nn.Module):  
 """ 网络中的基本块，包含注意力和多层感知机 """  
 def \_\_init\_\_(self, dim, mlp\_ratio=4., drop=0., drop\_path=0., act\_layer=nn.GELU):  
 super().\_\_init\_\_()  
 self.norm1 = nn.BatchNorm2d(dim) # 第一层归一化  
 self.norm2 = nn.BatchNorm2d(dim) # 第二层归一化  
 self.attn = Attention(dim) # 注意力模块  
 self.mlp = Mlp(in\_features=dim, hidden\_features=int(dim \* mlp\_ratio), act\_layer=act\_layer, drop=drop) # MLP模块  
  
 def forward(self, x):  
 x = x + self.attn(self.norm1(x)) # 注意力连接  
 x = x + self.mlp(self.norm2(x)) # MLP连接  
 return x  
  
  
class OverlapPatchEmbed(nn.Module):  
 """ 图像到补丁的嵌入 """  
 def \_\_init\_\_(self, img\_size=224, patch\_size=7, stride=4, in\_chans=3, embed\_dim=768):  
 super().\_\_init\_\_()  
 self.proj = nn.Conv2d(in\_chans, embed\_dim, kernel\_size=patch\_size, stride=stride, padding=(patch\_size // 2)) # 卷积嵌入  
 self.norm = nn.BatchNorm2d(embed\_dim) # 归一化层  
  
 def forward(self, x):  
 x = self.proj(x) # 卷积嵌入  
 x = self.norm(x) # 归一化  
 return x  
  
  
class LSKNet(nn.Module):  
 """ LSK网络结构 """  
 def \_\_init\_\_(self, img\_size=224, in\_chans=3, embed\_dims=[64, 128, 256, 512], depths=[3, 4, 6, 3]):  
 super().\_\_init\_\_()  
 self.num\_stages = len(embed\_dims) # 网络阶段数  
  
 for i in range(self.num\_stages):  
 patch\_embed = OverlapPatchEmbed(img\_size=img\_size // (2 \*\* i), in\_chans=in\_chans if i == 0 else embed\_dims[i - 1], embed\_dim=embed\_dims[i])  
 block = nn.ModuleList([Block(dim=embed\_dims[i]) for \_ in range(depths[i])]) # 生成多个Block  
 norm = nn.BatchNorm2d(embed\_dims[i]) # 归一化层  
 setattr(self, f"patch\_embed{i + 1}", patch\_embed) # 动态添加属性  
 setattr(self, f"block{i + 1}", block)  
 setattr(self, f"norm{i + 1}", norm)  
  
 def forward(self, x):  
 outs = []  
 for i in range(self.num\_stages):  
 patch\_embed = getattr(self, f"patch\_embed{i + 1}")  
 block = getattr(self, f"block{i + 1}")  
 norm = getattr(self, f"norm{i + 1}")  
 x = patch\_embed(x) # 嵌入  
 for blk in block:  
 x = blk(x) # 通过每个Block  
 x = norm(x) # 归一化  
 outs.append(x) # 保存输出  
 return outs  
  
  
class DWConv(nn.Module):  
 """ 深度卷积模块 """  
 def \_\_init\_\_(self, dim=768):  
 super(DWConv, self).\_\_init\_\_()  
 self.dwconv = nn.Conv2d(dim, dim, 3, padding=1, groups=dim) # 深度卷积  
  
 def forward(self, x):  
 return self.dwconv(x) # 直接返回深度卷积结果  
  
  
def lsknet\_t(weights=''):  
 """ LSKNet的一个变体，加载权重 """  
 model = LSKNet(embed\_dims=[32, 64, 160, 256], depths=[3, 3, 5, 2])  
 if weights:  
 model.load\_state\_dict(torch.load(weights)['state\_dict']) # 加载权重  
 return model  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 model = lsknet\_t('lsk\_t\_backbone-2ef8a593.pth') # 实例化模型  
 inputs = torch.randn((1, 3, 640, 640)) # 随机输入  
 for i in model(inputs):  
 print(i.size()) # 打印输出尺寸  
```  
  
### 代码核心部分分析  
1. \*\*Mlp\*\*: 实现了一个多层感知机，包含两个卷积层和一个深度卷积层，用于特征提取。  
2. \*\*LSKblock\*\*: 通过深度卷积和空间卷积生成注意力特征，并结合平均和最大池化进行加权。  
3. \*\*Attention\*\*: 通过残差连接和激活函数，结合LSK模块实现注意力机制。  
4. \*\*Block\*\*: 组合了注意力模块和多层感知机，形成网络的基本构建块。  
5. \*\*OverlapPatchEmbed\*\*: 将输入图像转换为补丁，进行特征嵌入。  
6. \*\*LSKNet\*\*: 整体网络结构，包含多个阶段，每个阶段由嵌入层、多个Block和归一化层组成。  
7. \*\*DWConv\*\*: 实现深度卷积，用于特征提取。  
  
以上代码构成了一个深度学习模型的基础结构，适用于图像处理任务。```

这个文件 `lsknet.py` 实现了一个名为 LSKNet 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，包含了多个模块和类，构成了整个网络的结构。  
  
首先，文件引入了必要的库，包括 PyTorch 的核心模块和一些辅助函数。接着，定义了几个重要的类。`Mlp` 类实现了一个多层感知机结构，包含了卷积层、深度卷积层、激活函数（GELU）和 dropout 层。这个类的 `forward` 方法定义了前向传播的过程。  
  
`LSKblock` 类是一个核心模块，主要实现了一种特定的注意力机制。它通过多个卷积层和激活函数对输入进行处理，并使用 sigmoid 函数生成的权重对输入进行加权，最终输出与输入相乘的结果。  
  
`Attention` 类则封装了一个注意力机制，它首先通过卷积层对输入进行线性变换，然后经过激活函数和 `LSKblock` 处理，最后再通过一个卷积层输出结果。这个类的设计使得网络能够关注输入特征的不同部分。  
  
`Block` 类是网络的基本构建块，结合了注意力机制和多层感知机。它使用了批归一化和 dropout 技术来提高模型的稳定性和泛化能力。每个 Block 由两个主要部分组成：注意力部分和 MLP 部分，二者通过残差连接相结合。  
  
`OverlapPatchEmbed` 类负责将输入图像转换为补丁嵌入。它使用卷积层来提取特征，并进行归一化处理。这个过程将输入图像划分为多个小块，便于后续的处理。  
  
`LSKNet` 类是整个模型的主类，负责构建网络的不同阶段。它根据输入的参数设置不同的嵌入维度、深度和其他超参数。模型的前向传播方法将输入逐层传递，并在每个阶段进行处理，最终输出多个特征图。  
  
`DWConv` 类实现了深度卷积操作，主要用于 `Mlp` 类中。深度卷积是一种特殊的卷积方式，可以有效减少参数数量和计算量。  
  
此外，文件中还定义了 `update\_weight` 函数，用于更新模型的权重。`lsknet\_t` 和 `lsknet\_s` 函数则分别创建了两种不同配置的 LSKNet 模型，并可以加载预训练的权重。  
  
最后，在文件的主程序部分，创建了一个 LSKNet 模型实例，并生成了一个随机输入进行测试，打印出每个输出的尺寸。这部分代码可以用于验证模型的构建是否正确。  
  
整体来看，这个文件实现了一个复杂的图像处理模型，结合了多种深度学习技术，具有较强的灵活性和可扩展性。

```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  
  
class RadialBasisFunction(nn.Module):  
 def \_\_init\_\_(self, grid\_min: float = -2., grid\_max: float = 2., num\_grids: int = 8, denominator: float = None):  
 """  
 初始化径向基函数（RBF）模块。  
   
 参数：  
 - grid\_min: 网格的最小值  
 - grid\_max: 网格的最大值  
 - num\_grids: 网格的数量  
 - denominator: 用于控制基函数平滑度的分母  
 """  
 super().\_\_init\_\_()  
 # 创建从 grid\_min 到 grid\_max 的线性空间  
 grid = torch.linspace(grid\_min, grid\_max, num\_grids)  
 self.grid = torch.nn.Parameter(grid, requires\_grad=False) # 将网格参数化，不需要梯度  
 self.denominator = denominator or (grid\_max - grid\_min) / (num\_grids - 1) # 计算分母  
  
 def forward(self, x):  
 """  
 前向传播，计算径向基函数的输出。  
   
 参数：  
 - x: 输入张量  
   
 返回：  
 - 经过径向基函数变换后的输出  
 """  
 return torch.exp(-((x[..., None] - self.grid) / self.denominator) \*\* 2) # 计算RBF值  
  
  
class FastKANConvNDLayer(nn.Module):  
 def \_\_init\_\_(self, conv\_class, norm\_class, input\_dim, output\_dim, kernel\_size, groups=1, padding=0, stride=1, dilation=1, ndim: int = 2, grid\_size=8, base\_activation=nn.SiLU, grid\_range=[-2, 2], dropout=0.0):  
 """  
 初始化FastKAN卷积层。  
   
 参数：  
 - conv\_class: 卷积层的类（如nn.Conv2d）  
 - norm\_class: 归一化层的类（如nn.InstanceNorm2d）  
 - input\_dim: 输入维度  
 - output\_dim: 输出维度  
 - kernel\_size: 卷积核大小  
 - groups: 分组数  
 - padding: 填充  
 - stride: 步幅  
 - dilation: 膨胀  
 - ndim: 维度（1D, 2D, 3D）  
 - grid\_size: 网格大小  
 - base\_activation: 基础激活函数  
 - grid\_range: 网格范围  
 - dropout: dropout比率  
 """  
 super(FastKANConvNDLayer, self).\_\_init\_\_()  
 self.inputdim = input\_dim  
 self.outdim = output\_dim  
 self.kernel\_size = kernel\_size  
 self.padding = padding  
 self.stride = stride  
 self.dilation = dilation  
 self.groups = groups  
 self.ndim = ndim  
 self.grid\_size = grid\_size  
 self.base\_activation = base\_activation() # 实例化基础激活函数  
 self.grid\_range = grid\_range  
  
 # 验证分组参数  
 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')  
  
 # 创建基础卷积层和样条卷积层  
 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)])  
 self.spline\_conv = nn.ModuleList([conv\_class(grid\_size \* input\_dim // groups, output\_dim // groups, kernel\_size, stride, padding, dilation, groups=1, bias=False) for \_ in range(groups)])  
   
 # 创建归一化层  
 self.layer\_norm = nn.ModuleList([norm\_class(output\_dim // groups) for \_ in range(groups)])  
  
 # 初始化径向基函数  
 self.rbf = RadialBasisFunction(grid\_range[0], grid\_range[1], grid\_size)  
  
 # 初始化dropout层  
 self.dropout = nn.Dropout(p=dropout) if dropout > 0 else None  
  
 # 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\_fast\_kan(self, x, group\_index):  
 """  
 快速KAN前向传播，处理每个分组的输入。  
   
 参数：  
 - x: 输入张量  
 - group\_index: 当前处理的分组索引  
   
 返回：  
 - 输出张量  
 """  
 # 应用基础激活函数并进行线性变换  
 base\_output = self.base\_conv[group\_index](self.base\_activation(x))  
 if self.dropout is not None:  
 x = self.dropout(x) # 应用dropout  
 spline\_basis = self.rbf(self.layer\_norm[group\_index](x)) # 计算样条基  
 spline\_basis = spline\_basis.moveaxis(-1, 2).flatten(1, 2) # 调整维度以适应卷积层  
 spline\_output = self.spline\_conv[group\_index](spline\_basis) # 计算样条卷积输出  
 x = base\_output + spline\_output # 合并基础输出和样条输出  
  
 return x  
  
 def forward(self, x):  
 """  
 前向传播，处理所有分组的输入。  
   
 参数：  
 - x: 输入张量  
   
 返回：  
 - 输出张量  
 """  
 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\_fast\_kan(\_x.clone(), group\_ind) # 处理每个分组  
 output.append(y.clone())  
 y = torch.cat(output, dim=1) # 合并所有分组的输出  
 return y  
```  
  
### 代码核心部分说明：  
1. \*\*RadialBasisFunction类\*\*：实现了径向基函数的计算，主要用于将输入映射到高维空间，以便于后续的卷积操作。  
2. \*\*FastKANConvNDLayer类\*\*：实现了一个灵活的卷积层，支持多维卷积，结合了基础卷积和样条卷积，使用径向基函数来增强模型的表达能力。  
3. \*\*前向传播方法\*\*：`forward\_fast\_kan`和`forward`方法实现了输入的处理逻辑，包括分组处理、激活函数应用、dropout、样条基计算和输出合并。```

这个程序文件 `fast\_kan\_conv.py` 定义了一些用于快速卷积操作的神经网络层，主要包括径向基函数（Radial Basis Function）和多维卷积层（FastKANConvNDLayer），以及其一维、二维和三维的具体实现。  
  
首先，`RadialBasisFunction` 类是一个自定义的神经网络模块，使用了径向基函数来生成平滑的基函数。它的构造函数接受最小和最大网格值、网格数量以及分母参数。网格值通过 `torch.linspace` 生成，并且被定义为不可训练的参数。`forward` 方法计算输入 `x` 和网格之间的距离，并返回经过指数函数处理的结果，这个结果可以用于后续的卷积操作。  
  
接下来，`FastKANConvNDLayer` 类是一个多维卷积层的基类，支持一维、二维和三维卷积。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、网格大小、基础激活函数、网格范围和丢弃率等。构造函数中会检查分组数的有效性，并确保输入和输出维度可以被分组数整除。然后，它初始化了基础卷积层、样条卷积层、层归一化层和径向基函数模块。根据提供的丢弃率，选择适当的丢弃层。权重初始化使用了 Kaiming 均匀分布，以便于训练的开始。  
  
`forward\_fast\_kan` 方法是该类的核心部分，负责处理输入数据。它首先对输入应用基础激活函数，然后通过基础卷积层进行线性变换。接着，如果使用了丢弃层，则对输入进行丢弃处理。之后，计算样条基函数并通过样条卷积层得到输出，最后将基础卷积的输出和样条卷积的输出相加，返回最终结果。  
  
`forward` 方法将输入数据按组进行分割，并对每个组调用 `forward\_fast\_kan` 方法，最后将所有组的输出拼接在一起。  
  
接下来的三个类 `FastKANConv3DLayer`、`FastKANConv2DLayer` 和 `FastKANConv1DLayer` 分别继承自 `FastKANConvNDLayer`，并指定了卷积类型（3D、2D 和 1D）和归一化类型（实例归一化）。这些类的构造函数调用了基类的构造函数，并传递了适当的参数。  
  
总体而言，这个程序文件实现了一种高效的卷积层，结合了径向基函数和多维卷积操作，适用于深度学习中的图像和信号处理任务。

```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.functional as F  
  
def selective\_scan\_easy(us, dts, As, Bs, Cs, Ds, delta\_bias=None, delta\_softplus=False, return\_last\_state=False, chunksize=64):  
 """  
 选择性扫描函数，执行状态更新和输出计算。  
   
 参数：  
 us: 输入张量，形状为 (B, G \* D, L)  
 dts: 时间增量张量，形状为 (B, G \* D, L)  
 As: 状态转移矩阵，形状为 (G \* D, N)  
 Bs: 影响输入的矩阵，形状为 (B, G, N, L)  
 Cs: 输出矩阵，形状为 (B, G, N, L)  
 Ds: 额外的输入影响，形状为 (G \* D)  
 delta\_bias: 可选的偏置，形状为 (G \* D)  
 delta\_softplus: 是否对 dts 应用 softplus  
 return\_last\_state: 是否返回最后的状态  
 chunksize: 处理的块大小  
  
 返回：  
 输出张量和（可选的）最后状态  
 """  
   
 def selective\_scan\_chunk(us, dts, As, Bs, Cs, hprefix):  
 """  
 处理每个块的选择性扫描。  
   
 参数：  
 us: 当前块的输入张量  
 dts: 当前块的时间增量  
 As: 状态转移矩阵  
 Bs: 当前块的影响输入矩阵  
 Cs: 当前块的输出矩阵  
 hprefix: 前一个状态的输出  
   
 返回：  
 当前块的输出和更新后的状态  
 """  
 ts = dts.cumsum(dim=0) # 计算时间增量的累积和  
 Ats = torch.einsum("gdn,lbgd->lbgdn", As, ts).exp() # 计算状态转移的指数  
 scale = 1 # 缩放因子  
 rAts = Ats / scale # 归一化的状态转移  
 duts = dts \* us # 计算输入的影响  
 dtBus = torch.einsum("lbgd,lbgn->lbgdn", duts, Bs) # 计算影响输入的影响  
 hs\_tmp = rAts \* (dtBus / rAts).cumsum(dim=0) # 更新状态  
 hs = hs\_tmp + Ats \* hprefix.unsqueeze(0) # 加上前一个状态  
 ys = torch.einsum("lbgn,lbgdn->lbgd", Cs, hs) # 计算输出  
 return ys, hs # 返回输出和更新后的状态  
  
 # 数据类型设置  
 dtype = torch.float32  
 inp\_dtype = us.dtype # 输入数据类型  
 has\_D = Ds is not None # 检查是否有额外的输入影响  
 if chunksize < 1:  
 chunksize = Bs.shape[-1] # 设置块大小  
  
 # 处理输入数据  
 dts = dts.to(dtype)  
 if delta\_bias is not None:  
 dts = dts + delta\_bias.view(1, -1, 1).to(dtype) # 应用偏置  
 if delta\_softplus:  
 dts = F.softplus(dts) # 应用 softplus  
  
 # 处理维度  
 Bs = Bs.unsqueeze(1) if len(Bs.shape) == 3 else Bs  
 Cs = Cs.unsqueeze(1) if len(Cs.shape) == 3 else Cs  
 B, G, N, L = Bs.shape  
 us = us.view(B, G, -1, L).permute(3, 0, 1, 2).to(dtype)  
 dts = dts.view(B, G, -1, L).permute(3, 0, 1, 2).to(dtype)  
 As = As.view(G, -1, N).to(dtype)  
 Bs = Bs.permute(3, 0, 1, 2).to(dtype)  
 Cs = Cs.permute(3, 0, 1, 2).to(dtype)  
 Ds = Ds.view(G, -1).to(dtype) if has\_D else None  
 D = As.shape[1] # 状态维度  
  
 oys = [] # 输出列表  
 hprefix = us.new\_zeros((B, G, D, N), dtype=dtype) # 初始化前一个状态  
 for i in range(0, L, chunksize):  
 ys, hs = selective\_scan\_chunk(  
 us[i:i + chunksize], dts[i:i + chunksize],   
 As, Bs[i:i + chunksize], Cs[i:i + chunksize], hprefix,   
 )  
 oys.append(ys) # 收集输出  
 hprefix = hs[-1] # 更新前一个状态  
  
 oys = torch.cat(oys, dim=0) # 合并输出  
 if has\_D:  
 oys = oys + Ds \* us # 添加额外的输入影响  
 oys = oys.permute(1, 2, 3, 0).view(B, -1, L) # 调整输出维度  
  
 return oys.to(inp\_dtype) if not return\_last\_state else (oys.to(inp\_dtype), hprefix.view(B, G \* D, N).float()) # 返回输出和状态  
```  
  
### 代码说明  
1. \*\*函数定义\*\*：`selective\_scan\_easy` 是一个选择性扫描的函数，主要用于在给定输入和时间增量的情况下，计算输出和状态更新。  
2. \*\*内部函数\*\*：`selective\_scan\_chunk` 用于处理每个块的计算，使用张量运算来更新状态和计算输出。  
3. \*\*参数处理\*\*：函数首先处理输入参数，包括时间增量、状态转移矩阵等，并进行必要的维度调整。  
4. \*\*输出计算\*\*：通过循环处理每个块，调用内部函数进行状态更新和输出计算，最后将结果合并并返回。  
  
这个核心部分是实现选择性扫描的基础，涉及张量运算和状态更新的逻辑。```

这个程序文件 `test\_selective\_scan\_easy.py` 实现了一个名为 `selective\_scan\_easy` 的函数，该函数用于执行选择性扫描操作，通常用于序列数据的处理，尤其是在深度学习模型中。文件中包含了多个函数和类，主要功能是计算和反向传播，适用于 PyTorch 框架。  
  
首先，文件导入了一些必要的库，包括 `torch` 和 `pytest`，并定义了一些常量和模式。`selective\_scan\_easy` 函数的输入包括多个张量（`us`, `dts`, `As`, `Bs`, `Cs`, `Ds`），这些张量代表了批量数据、时间步长、状态转移矩阵等。函数内部定义了一个嵌套函数 `selective\_scan\_chunk`，用于处理数据块的选择性扫描逻辑。  
  
在 `selective\_scan\_easy` 函数中，首先对输入数据进行类型转换和形状调整，以确保它们符合后续计算的要求。接着，函数通过循环分块处理输入数据，调用 `selective\_scan\_chunk` 进行计算，最终将结果合并并返回。  
  
`SelectiveScanEasy` 类是一个自定义的 PyTorch 自动求导函数，包含 `forward` 和 `backward` 方法。`forward` 方法实现了前向传播逻辑，计算输出和最后的状态；`backward` 方法实现了反向传播逻辑，计算梯度。该类支持混合精度计算，使用了 `torch.cuda.amp.custom\_fwd` 和 `torch.cuda.amp.custom\_bwd` 装饰器。  
  
此外，文件还定义了多个版本的选择性扫描函数（如 `selective\_scan\_easyv2` 和 `selective\_scan\_easyv3`），这些版本在实现细节上有所不同，可能会针对不同的场景进行优化。  
  
文件的最后部分是一个测试函数 `test\_selective\_scan`，使用 `pytest` 框架对选择性扫描函数进行单元测试。测试中使用了多种参数组合，验证了函数的正确性和性能。  
  
总的来说，这个程序文件实现了一个高效的选择性扫描机制，适用于处理序列数据的深度学习任务，并通过测试确保了其功能的正确性。

```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 LayerNorm2d(nn.Module):  
 """  
 2D层归一化模块  
 """  
 def \_\_init\_\_(self, normalized\_shape, eps=1e-6, elementwise\_affine=True):  
 super().\_\_init\_\_()  
 # 初始化LayerNorm，normalized\_shape为归一化的形状  
 self.norm = nn.LayerNorm(normalized\_shape, eps, elementwise\_affine)  
  
 def forward(self, x):  
 # 将输入x的形状从 (B, C, H, W) 转换为 (B, H, W, C)  
 x = rearrange(x, 'b c h w -> b h w c').contiguous()  
 # 应用LayerNorm  
 x = self.norm(x)  
 # 将输出的形状转换回 (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  
 """  
 @staticmethod  
 def forward(ctx, x: torch.Tensor):  
 B, C, H, W = x.shape  
 ctx.shape = (B, C, H, W)  
 # 创建一个新的张量，存储交叉扫描的结果  
 xs = x.new\_empty((B, 4, C, H \* W))  
 xs[:, 0] = x.flatten(2, 3) # 原始顺序  
 xs[:, 1] = x.transpose(dim0=2, dim1=3).flatten(2, 3) # 转置后的顺序  
 xs[:, 2:4] = torch.flip(xs[:, 0:2], dims=[-1]) # 翻转的顺序  
 return xs  
  
 @staticmethod  
 def backward(ctx, ys: torch.Tensor):  
 B, C, H, W = ctx.shape  
 L = H \* W  
 # 计算反向传播的梯度  
 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()  
   
 # 处理输入的维度  
 if B.dim() == 3:  
 B = B.unsqueeze(dim=1)  
 ctx.squeeze\_B = True  
 if C.dim() == 3:  
 C = C.unsqueeze(dim=1)  
 ctx.squeeze\_C = True  
   
 # 选择性扫描的前向操作  
 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)  
 return out  
  
 @staticmethod  
 @torch.cuda.amp.custom\_bwd  
 def backward(ctx, dout, \*args):  
 u, delta, A, B, C, D, delta\_bias, x = ctx.saved\_tensors  
 if dout.stride(-1) != 1:  
 dout = dout.contiguous()  
 # 选择性扫描的反向操作  
 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)  
  
def cross\_selective\_scan(x: torch.Tensor, x\_proj\_weight: torch.Tensor, dt\_projs\_weight: torch.Tensor, A\_logs: torch.Tensor, Ds: torch.Tensor, out\_norm: torch.nn.Module = None):  
 """  
 交叉选择性扫描操作  
 """  
 B, D, H, W = x.shape  
 D, N = A\_logs.shape  
 K, D, R = dt\_projs\_weight.shape  
 L = H \* W  
  
 # 进行交叉扫描  
 xs = CrossScan.apply(x)  
  
 # 进行权重投影  
 x\_dbl = torch.einsum("b k d l, k c d -> b k c l", xs, x\_proj\_weight)  
 dts, Bs, Cs = torch.split(x\_dbl, [R, N, N], dim=2)  
 dts = torch.einsum("b k r l, k d r -> b k d l", dts, dt\_projs\_weight)  
  
 # HiPPO矩阵  
 As = -torch.exp(A\_logs.to(torch.float)) # (k \* c, d\_state)  
 Bs = Bs.contiguous()  
 Cs = Cs.contiguous()  
 Ds = Ds.to(torch.float) # (K \* c)  
  
 # 选择性扫描  
 ys: torch.Tensor = SelectiveScanCore.apply(xs, dts, As, Bs, Cs, Ds)  
 return ys  
  
class SS2D(nn.Module):  
 """  
 2D选择性扫描模块  
 """  
 def \_\_init\_\_(self, d\_model=96, d\_state=16, ssm\_ratio=2.0, ssm\_rank\_ratio=2.0):  
 super().\_\_init\_\_()  
 self.d\_model = d\_model  
 self.d\_state = d\_state  
 self.ssm\_ratio = ssm\_ratio  
 self.ssm\_rank\_ratio = ssm\_rank\_ratio  
   
 # 输入投影层  
 self.in\_proj = nn.Conv2d(d\_model, int(ssm\_ratio \* d\_model), kernel\_size=1)  
 # 输出投影层  
 self.out\_proj = nn.Conv2d(int(ssm\_ratio \* d\_model), d\_model, kernel\_size=1)  
  
 def forward(self, x: torch.Tensor):  
 # 前向传播  
 x = self.in\_proj(x)  
 x = cross\_selective\_scan(x, self.x\_proj\_weight, self.dt\_projs\_weight, self.A\_logs, self.Ds)  
 x = self.out\_proj(x)  
 return x  
  
class VSSBlock\_YOLO(nn.Module):  
 """  
 YOLO模型中的VSSBlock模块  
 """  
 def \_\_init\_\_(self, in\_channels: int, hidden\_dim: int):  
 super().\_\_init\_\_()  
 self.proj\_conv = nn.Conv2d(in\_channels, hidden\_dim, kernel\_size=1)  
 self.ss2d = SS2D(d\_model=hidden\_dim)  
  
 def forward(self, input: torch.Tensor):  
 input = self.proj\_conv(input) # 投影  
 x = self.ss2d(input) # 选择性扫描  
 return x  
```  
  
### 代码说明  
1. \*\*LayerNorm2d\*\*: 实现了2D层归一化，适用于图像数据。  
2. \*\*CrossScan\*\*: 实现了交叉扫描操作，前向传播和反向传播均定义。  
3. \*\*SelectiveScanCore\*\*: 实现了选择性扫描的核心功能，包含前向和反向传播。  
4. \*\*cross\_selective\_scan\*\*: 结合交叉扫描和选择性扫描的操作。  
5. \*\*SS2D\*\*: 2D选择性扫描模块，包含输入和输出投影层。  
6. \*\*VSSBlock\_YOLO\*\*: YOLO模型中的VSSBlock模块，整合了投影和选择性扫描。  
  
这些模块和功能构成了深度学习模型中的关键部分，尤其是在处理图像数据时。```

这个程序文件 `mamba\_yolo.py` 是一个用于实现 YOLO（You Only Look Once）目标检测模型的 PyTorch 代码。文件中定义了多个类和函数，主要用于构建模型的不同模块，包括层归一化、卷积、选择性扫描等。以下是对代码的详细说明。  
  
首先，文件导入了一些必要的库，包括 `torch`、`math`、`functools` 和 `torch.nn` 等，后者用于构建神经网络层。还使用了 `einops` 库来进行张量的重排和重复操作，`timm.layers` 中的 `DropPath` 用于实现随机丢弃路径的功能。  
  
接下来，定义了一个 `LayerNorm2d` 类，它是对二维数据进行层归一化的实现。其 `forward` 方法将输入张量的维度从 `(batch, channels, height, width)` 转换为 `(batch, height, width, channels)`，然后进行归一化处理，最后再转换回原来的维度。  
  
`autopad` 函数用于根据卷积核的大小自动计算填充量，以确保输出形状与输入形状相同。  
  
接下来的 `CrossScan` 和 `CrossMerge` 类实现了交叉扫描和交叉合并的功能，这些操作对于处理特征图的不同维度和方向非常重要。它们的 `forward` 和 `backward` 方法定义了前向传播和反向传播的计算过程。  
  
`SelectiveScanCore` 类实现了选择性扫描的核心功能，提供了前向和反向传播的计算方法。选择性扫描是一种高效的计算方法，用于处理序列数据。  
  
`cross\_selective\_scan` 函数是一个高层次的接口，用于执行选择性扫描操作，接受多个输入参数并返回处理后的输出。  
  
`SS2D` 类是一个主要的模块，结合了选择性扫描和卷积操作。它的构造函数定义了多个参数，包括模型的维度、状态维度、卷积层的参数等。`forward` 方法实现了前向传播的逻辑，结合了输入投影、卷积和选择性扫描的操作。  
  
`RGBlock` 和 `LSBlock` 类实现了残差块和层次块的功能，分别用于构建模型的不同部分。它们包含了卷积层、激活函数和丢弃层。  
  
`XSSBlock` 和 `VSSBlock\_YOLO` 类是 YOLO 模型的关键模块，结合了多个子模块以实现复杂的特征提取和处理。它们的构造函数中定义了输入和输出的维度、激活函数、丢弃率等参数，并在 `forward` 方法中实现了具体的前向传播逻辑。  
  
`SimpleStem` 类是模型的输入处理模块，负责将输入图像通过卷积层进行特征提取。  
  
最后，`VisionClueMerge` 类用于合并不同特征图的输出，以便于后续的处理。  
  
整体来看，这个文件实现了一个复杂的神经网络结构，结合了多种技术以提高目标检测的性能。通过使用选择性扫描、残差连接和多种卷积操作，模型能够有效地提取和处理图像特征。

### 程序整体功能和构架概括  
  
该工程包含多个 Python 文件，每个文件实现了深度学习模型的不同组件，主要用于图像处理和目标检测任务。整体架构围绕 PyTorch 框架构建，结合了多种深度学习技术，包括卷积神经网络（CNN）、选择性扫描、注意力机制等。具体功能如下：  
  
- \*\*lsknet.py\*\*：实现了 LSKNet 模型，主要用于图像特征提取，结合了多层感知机和注意力机制，具有灵活的网络结构和强大的特征学习能力。  
- \*\*fast\_kan\_conv.py\*\*：定义了快速卷积操作的模块，支持一维、二维和三维卷积，结合了径向基函数，旨在提高卷积计算的效率。  
- \*\*test\_selective\_scan\_easy.py\*\*：实现了选择性扫描的功能，适用于序列数据处理，并通过单元测试确保功能的正确性。  
- \*\*mamba\_yolo.py\*\*：构建了 YOLO 目标检测模型，结合了选择性扫描和多种卷积操作，旨在实现高效的目标检测。  
  
### 文件功能整理表  
  
| 文件名 | 功能描述 |  
|---------------------------------|----------------------------------------------------------------------------------------------|  
| `lsknet.py` | 实现 LSKNet 模型，包含多层感知机、注意力机制和特征提取模块，适用于图像处理任务。 |  
| `fast\_kan\_conv.py` | 定义快速卷积操作模块，支持一维、二维和三维卷积，结合径向基函数，提高卷积计算效率。 |  
| `test\_selective\_scan\_easy.py` | 实现选择性扫描功能，适用于序列数据处理，并包含单元测试以验证功能的正确性。 |  
| `mamba\_yolo.py` | 构建 YOLO 目标检测模型，结合选择性扫描和多种卷积操作，实现高效的目标检测。 |  
  
这个表格总结了每个文件的主要功能，展示了整个工程的模块化设计和深度学习任务的多样性。

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。