# 改进yolo11-AggregatedAtt等200+全套创新点大全：停车位状态检测系统源码＆数据集全套

## 3.背景

### 研究背景与意义  
  
随着城市化进程的加快，城市交通问题日益严重，停车难已成为许多城市面临的普遍挑战。有效的停车管理不仅能够提高城市交通的流动性，还能提升居民的生活质量。因此，开发一种高效的停车位状态检测系统显得尤为重要。基于深度学习的计算机视觉技术，尤其是目标检测算法的快速发展，为停车位状态的实时监测提供了新的解决方案。  
  
本研究旨在基于改进的YOLOv11模型，构建一个高效的停车位状态检测系统。该系统能够实时识别停车位的状态，分类为“占用”（Busy）和“空闲”（Free），从而为驾驶员提供及时的停车信息，减少寻找停车位的时间，降低交通拥堵。通过使用包含3100张标注图像的数据集，系统能够在多种环境下进行训练和测试，确保其在实际应用中的鲁棒性和准确性。  
  
停车位状态检测不仅对个人驾驶者有直接的益处，还能为城市交通管理提供数据支持。通过分析停车位的使用情况，城市管理者可以优化停车资源的配置，制定更合理的停车政策。此外，结合实时数据，智能交通系统可以为驾驶者提供更为精准的导航服务，提升整体交通效率。  
  
综上所述，基于改进YOLOv11的停车位状态检测系统的研究具有重要的理论意义和实际应用价值。它不仅推动了计算机视觉技术在智能交通领域的应用，还为解决城市停车难题提供了创新的思路和方法。通过该系统的开发与应用，期待能够为未来的智慧城市建设贡献一份力量。

## 2.视频效果

## 3. 图片效果展示

注意：本项目提供训练的数据集和训练教程,由于版本持续更新,暂不提供权重文件（best.pt）,请按照6.训练教程进行训练后实现上图演示的效果。

## 4. 数据集信息

本项目数据集信息介绍  
  
本项目旨在改进YOLOv11的停车位状态检测系统，因此所使用的数据集围绕“ParkingSlots”主题进行构建。该数据集专注于停车位的状态分类，包含两种主要类别：‘Busy’（占用）和‘Free’（空闲）。通过对这两种状态的精确识别，系统能够实时监测停车场的使用情况，从而为驾驶员提供有效的停车指导，提升停车效率，减少寻找停车位所需的时间。  
  
数据集的构建过程涉及大量的图像采集与标注工作。为了确保数据的多样性和代表性，数据来源于不同类型的停车场，包括城市街道、商业区、住宅区等。每张图像均经过严格的标注，确保其准确反映停车位的真实状态。这一过程不仅提高了数据集的质量，也为后续的模型训练提供了坚实的基础。  
  
在数据集的规模方面，涵盖了数千张图像，充分考虑了不同天气、光照条件及时间段的变化，确保模型在各种环境下均能保持良好的性能。此外，数据集还包含了不同角度和距离拍摄的停车位图像，以增强模型的泛化能力。这种多样化的训练数据将有助于YOLOv11在实际应用中更准确地识别停车位状态，从而有效降低误报率和漏报率。  
  
通过使用这一数据集，研究团队期望能够提升停车位状态检测系统的准确性和可靠性，为智能交通系统的发展贡献力量。最终，改进后的系统将为城市交通管理提供有力支持，帮助解决停车难的问题，提升城市的整体交通效率。

## 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  
import torch.nn.functional as F  
from timm.layers import weight\_init  
  
# 定义激活函数类，继承自 ReLU  
class Activation(nn.ReLU):  
 def \_\_init\_\_(self, dim, act\_num=3, deploy=False):  
 super(Activation, self).\_\_init\_\_()  
 self.deploy = deploy # 是否为部署模式  
 # 权重参数初始化  
 self.weight = torch.nn.Parameter(torch.randn(dim, 1, act\_num \* 2 + 1, act\_num \* 2 + 1))  
 self.bias = None  
 self.bn = nn.BatchNorm2d(dim, eps=1e-6) # 批归一化  
 self.dim = dim  
 self.act\_num = act\_num  
 weight\_init.trunc\_normal\_(self.weight, std=.02) # 权重初始化  
  
 def forward(self, x):  
 # 前向传播  
 if self.deploy:  
 # 部署模式下的卷积操作  
 return F.conv2d(  
 super(Activation, self).forward(x),   
 self.weight, self.bias, padding=(self.act\_num \* 2 + 1) // 2, groups=self.dim)  
 else:  
 # 训练模式下的卷积操作  
 return self.bn(F.conv2d(  
 super(Activation, self).forward(x),  
 self.weight, padding=self.act\_num, groups=self.dim))  
  
 def switch\_to\_deploy(self):  
 # 切换到部署模式  
 if not self.deploy:  
 kernel, bias = self.\_fuse\_bn\_tensor(self.weight, self.bn) # 融合BN层  
 self.weight.data = kernel  
 self.bias = torch.nn.Parameter(torch.zeros(self.dim))  
 self.bias.data = bias  
 self.\_\_delattr\_\_('bn') # 删除bn属性  
 self.deploy = True  
  
 def \_fuse\_bn\_tensor(self, weight, bn):  
 # 融合卷积层和BN层的权重  
 kernel = weight  
 running\_mean = bn.running\_mean  
 running\_var = bn.running\_var  
 gamma = bn.weight  
 beta = bn.bias  
 eps = bn.eps  
 std = (running\_var + eps).sqrt()  
 t = (gamma / std).reshape(-1, 1, 1, 1)  
 return kernel \* t, beta + (0 - running\_mean) \* gamma / std  
  
# 定义网络块  
class Block(nn.Module):  
 def \_\_init\_\_(self, dim, dim\_out, act\_num=3, stride=2, deploy=False):  
 super().\_\_init\_\_()  
 self.deploy = deploy  
 # 根据是否为部署模式选择不同的卷积结构  
 if self.deploy:  
 self.conv = nn.Conv2d(dim, dim\_out, kernel\_size=1)  
 else:  
 self.conv1 = nn.Sequential(  
 nn.Conv2d(dim, dim, kernel\_size=1),  
 nn.BatchNorm2d(dim, eps=1e-6),  
 )  
 self.conv2 = nn.Sequential(  
 nn.Conv2d(dim, dim\_out, kernel\_size=1),  
 nn.BatchNorm2d(dim\_out, eps=1e-6)  
 )  
 # 池化层  
 self.pool = nn.MaxPool2d(stride) if stride != 1 else nn.Identity()  
 self.act = Activation(dim\_out, act\_num) # 激活函数  
  
 def forward(self, x):  
 # 前向传播  
 if self.deploy:  
 x = self.conv(x)  
 else:  
 x = self.conv1(x)  
 x = F.leaky\_relu(x, negative\_slope=1) # 使用Leaky ReLU激活  
 x = self.conv2(x)  
  
 x = self.pool(x) # 池化  
 x = self.act(x) # 激活  
 return x  
  
 def switch\_to\_deploy(self):  
 # 切换到部署模式  
 if not self.deploy:  
 # 融合卷积和BN层  
 kernel, bias = self.\_fuse\_bn\_tensor(self.conv1[0], self.conv1[1])  
 self.conv1[0].weight.data = kernel  
 self.conv1[0].bias.data = bias  
 kernel, bias = self.\_fuse\_bn\_tensor(self.conv2[0], self.conv2[1])  
 self.conv = self.conv2[0] # 更新卷积层  
 self.conv.weight.data = kernel  
 self.conv.bias.data = bias  
 self.\_\_delattr\_\_('conv1')  
 self.\_\_delattr\_\_('conv2')  
 self.act.switch\_to\_deploy() # 切换激活函数  
 self.deploy = True  
  
# 定义主网络结构  
class VanillaNet(nn.Module):  
 def \_\_init\_\_(self, in\_chans=3, num\_classes=1000, dims=[96, 192, 384, 768],   
 drop\_rate=0, act\_num=3, strides=[2, 2, 2, 1], deploy=False):  
 super().\_\_init\_\_()  
 self.deploy = deploy  
 # 网络的stem部分  
 if self.deploy:  
 self.stem = nn.Sequential(  
 nn.Conv2d(in\_chans, dims[0], kernel\_size=4, stride=4),  
 Activation(dims[0], act\_num)  
 )  
 else:  
 self.stem1 = nn.Sequential(  
 nn.Conv2d(in\_chans, dims[0], kernel\_size=4, stride=4),  
 nn.BatchNorm2d(dims[0], eps=1e-6),  
 )  
 self.stem2 = nn.Sequential(  
 nn.Conv2d(dims[0], dims[0], kernel\_size=1, stride=1),  
 nn.BatchNorm2d(dims[0], eps=1e-6),  
 Activation(dims[0], act\_num)  
 )  
  
 self.stages = nn.ModuleList()  
 for i in range(len(strides)):  
 stage = Block(dim=dims[i], dim\_out=dims[i + 1], act\_num=act\_num, stride=strides[i], deploy=deploy)  
 self.stages.append(stage) # 添加网络块  
  
 def forward(self, x):  
 # 前向传播  
 if self.deploy:  
 x = self.stem(x)  
 else:  
 x = self.stem1(x)  
 x = F.leaky\_relu(x, negative\_slope=1)  
 x = self.stem2(x)  
  
 for stage in self.stages:  
 x = stage(x) # 依次通过每个网络块  
 return x  
  
 def switch\_to\_deploy(self):  
 # 切换到部署模式  
 if not self.deploy:  
 self.stem2[2].switch\_to\_deploy()  
 kernel, bias = self.\_fuse\_bn\_tensor(self.stem1[0], self.stem1[1])  
 self.stem1[0].weight.data = kernel  
 self.stem1[0].bias.data = bias  
 self.stem = nn.Sequential(self.stem1[0], self.stem2[2])  
 self.\_\_delattr\_\_('stem1')  
 self.\_\_delattr\_\_('stem2')  
  
 for stage in self.stages:  
 stage.switch\_to\_deploy() # 切换每个网络块  
 self.deploy = True  
  
# 用于加载预训练权重的函数  
def update\_weight(model\_dict, weight\_dict):  
 temp\_dict = {}  
 for k, v in weight\_dict.items():  
 if k in model\_dict.keys() and model\_dict[k].shape == v.shape:  
 temp\_dict[k] = v  
 model\_dict.update(temp\_dict)  
 return model\_dict  
  
# 创建不同版本的VanillaNet模型  
def vanillanet\_5(pretrained='', \*\*kwargs):  
 model = VanillaNet(dims=[128 \* 4, 256 \* 4, 512 \* 4, 1024 \* 4], strides=[2, 2, 2], \*\*kwargs)  
 if pretrained:  
 weights = torch.load(pretrained)['model\_ema']  
 model.load\_state\_dict(update\_weight(model.state\_dict(), weights))  
 return model  
  
# 其他版本的创建函数类似...  
```  
  
### 代码分析  
1. \*\*Activation 类\*\*: 这个类实现了一个自定义的激活函数，支持训练和部署模式。它包含了权重和偏置的初始化，并提供了前向传播的实现。  
  
2. \*\*Block 类\*\*: 这个类定义了网络的基本构建块，包含了卷积层、批归一化层和激活函数。根据输入的步幅决定是否使用池化层。  
  
3. \*\*VanillaNet 类\*\*: 这是主网络结构，包含了stem部分和多个Block。它支持不同的输入通道数和输出类别数，并可以通过传入的参数配置网络的深度和宽度。  
  
4. \*\*update\_weight 函数\*\*: 这个函数用于更新模型的权重，确保加载的权重与模型的结构匹配。  
  
5. \*\*模型创建函数\*\*: 这些函数根据不同的配置创建 VanillaNet 模型，并支持加载预训练权重。  
  
以上是对代码的核心部分和详细注释，希望能帮助理解代码的结构和功能。```

该文件 `VanillaNet.py` 实现了一个名为 VanillaNet 的深度学习模型，主要用于图像处理任务。该模型的设计灵感来源于卷积神经网络（CNN），并且提供了多种不同配置的网络结构。以下是对文件中主要内容的逐步分析和说明。  
  
首先，文件开头包含版权信息和许可声明，表明该程序是开源的，可以在 MIT 许可下进行修改和分发。接着，导入了必要的库，包括 PyTorch 和一些辅助函数。  
  
接下来，定义了一个名为 `activation` 的类，该类继承自 `nn.ReLU`，用于实现自定义的激活函数。这个类中包含了一个可学习的权重和偏置，利用卷积操作对输入进行处理，并在前向传播中应用批归一化（Batch Normalization）。该类还提供了一个方法 `switch\_to\_deploy`，用于在推理阶段将模型切换到部署模式，合并卷积层和批归一化层的参数。  
  
然后，定义了一个 `Block` 类，表示网络中的基本构建块。该类包含多个卷积层和激活函数，并根据步幅选择不同的池化策略。与 `activation` 类类似，`Block` 类也实现了 `switch\_to\_deploy` 方法，用于在推理阶段优化模型结构。  
  
接下来，`VanillaNet` 类是整个模型的核心。它接受输入通道数、类别数、各层的维度、丢弃率、激活函数数量、步幅等参数进行初始化。该类构建了一个由多个 `Block` 组成的网络结构，并在前向传播中处理输入数据。`VanillaNet` 还提供了一个 `\_init\_weights` 方法，用于初始化网络权重。  
  
此外，文件中还定义了一些辅助函数，如 `update\_weight`，用于更新模型的权重；以及多个函数（如 `vanillanet\_5`、`vanillanet\_6` 等），用于创建不同配置的 VanillaNet 模型，并可选择加载预训练权重。  
  
最后，在文件的主程序部分，创建了一个输入张量并实例化了一个 VanillaNet 模型，随后进行前向传播并打印输出特征图的尺寸。  
  
整体而言，该文件实现了一个灵活且可扩展的卷积神经网络架构，适用于多种图像处理任务，并提供了多种配置选项以满足不同需求。

```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 torch.nn.functional as F  
  
# 定义相对位置的函数  
def rel\_pos(kernel\_size):  
 # 创建一个从-1到1的线性空间，生成两个维度的坐标  
 tensors = [torch.linspace(-1, 1, steps=kernel\_size) for \_ in range(2)]  
 # 生成网格坐标  
 kernel\_coord = torch.stack(torch.meshgrid(\*tensors), dim=-0)  
 kernel\_coord = kernel\_coord.unsqueeze(0) # 增加一个维度  
 return kernel\_coord  
  
# 定义自定义卷积层  
class SMPConv(nn.Module):  
 def \_\_init\_\_(self, planes, kernel\_size, n\_points, stride, padding, groups):  
 super().\_\_init\_\_()  
  
 self.planes = planes # 输出通道数  
 self.kernel\_size = kernel\_size # 卷积核大小  
 self.n\_points = n\_points # 关键点数量  
 self.init\_radius = 2 \* (2/kernel\_size) # 初始化半径  
  
 # 生成卷积核坐标  
 kernel\_coord = rel\_pos(kernel\_size)  
 self.register\_buffer('kernel\_coord', kernel\_coord) # 注册为缓冲区，不会被优化  
  
 # 权重坐标初始化  
 weight\_coord = torch.empty(1, n\_points, 2)  
 nn.init.trunc\_normal\_(weight\_coord, std=0.2, a=-1., b=1.) # 使用截断正态分布初始化  
 self.weight\_coord = nn.Parameter(weight\_coord) # 注册为可学习参数  
  
 # 半径初始化  
 self.radius = nn.Parameter(torch.empty(1, n\_points).unsqueeze(-1).unsqueeze(-1))  
 self.radius.data.fill\_(value=self.init\_radius) # 填充初始值  
  
 # 权重初始化  
 weights = torch.empty(1, planes, n\_points)  
 nn.init.trunc\_normal\_(weights, std=.02) # 使用截断正态分布初始化  
 self.weights = nn.Parameter(weights) # 注册为可学习参数  
  
 def forward(self, x):  
 # 生成卷积核并进行前向传播  
 kernels = self.make\_kernels().unsqueeze(1) # 生成卷积核  
 x = x.contiguous() # 确保输入是连续的  
 kernels = kernels.contiguous() # 确保卷积核是连续的  
  
 # 根据输入数据类型选择相应的卷积实现  
 if x.dtype == torch.float32:  
 x = \_DepthWiseConv2dImplicitGEMMFP32.apply(x, kernels) # FP32卷积  
 elif x.dtype == torch.float16:  
 x = \_DepthWiseConv2dImplicitGEMMFP16.apply(x, kernels) # FP16卷积  
 else:  
 raise TypeError("Only support fp32 and fp16, get {}".format(x.dtype)) # 抛出异常  
 return x   
  
 def make\_kernels(self):  
 # 计算卷积核  
 diff = self.weight\_coord.unsqueeze(-2) - self.kernel\_coord.reshape(1, 2, -1).transpose(1, 2) # 计算差值  
 diff = diff.transpose(2, 3).reshape(1, self.n\_points, 2, self.kernel\_size, self.kernel\_size) # 重塑形状  
 diff = F.relu(1 - torch.sum(torch.abs(diff), dim=2) / self.radius) # 计算ReLU激活  
  
 # 计算最终的卷积核  
 kernels = torch.matmul(self.weights, diff.reshape(1, self.n\_points, -1)) # 矩阵乘法  
 kernels = kernels.reshape(1, self.planes, \*self.kernel\_coord.shape[2:]) # 重塑形状  
 kernels = kernels.squeeze(0) # 去掉多余的维度  
 kernels = torch.flip(kernels.permute(0, 2, 1), dims=(1,)) # 反转维度  
 return kernels  
  
# 定义SMPBlock模块  
class SMPBlock(nn.Module):  
 def \_\_init\_\_(self, in\_channels, dw\_channels, lk\_size, drop\_path, n\_points=None, n\_points\_divide=4):  
 super().\_\_init\_\_()  
 self.pw1 = conv\_bn\_relu(in\_channels, dw\_channels, 1, 1, 0, groups=1) # 逐点卷积  
 self.pw2 = conv\_bn(dw\_channels, in\_channels, 1, 1, 0, groups=1) # 逐点卷积  
 self.large\_kernel = SMPCNN(in\_channels=dw\_channels, out\_channels=dw\_channels, kernel\_size=lk\_size,  
 stride=1, groups=dw\_channels, n\_points=n\_points, n\_points\_divide=n\_points\_divide) # 大卷积核  
 self.lk\_nonlinear = nn.ReLU() # 非线性激活  
 self.drop\_path = DropPath(drop\_path) if drop\_path > 0. else nn.Identity() # DropPath层  
  
 def forward(self, x):  
 out = self.pw1(x) # 逐点卷积  
 out = self.large\_kernel(out) # 大卷积核  
 out = self.lk\_nonlinear(out) # 激活  
 out = self.pw2(out) # 逐点卷积  
 return x + self.drop\_path(out) # 残差连接  
```  
  
### 代码说明：  
1. \*\*SMPConv类\*\*：实现了一种自定义的卷积层，支持动态生成卷积核，利用权重坐标和卷积核坐标的差值来计算卷积核，并根据输入数据类型选择不同的卷积实现。  
2. \*\*SMPBlock类\*\*：结合逐点卷积和大卷积核的特性，通过残差连接来增强特征传递，使用DropPath来实现随机丢弃路径的效果，有助于模型的正则化。```

这个程序文件 `SMPConv.py` 定义了一些用于构建深度学习模型的类和函数，主要集中在卷积操作和模块的实现上。文件中使用了 PyTorch 框架，并引入了一些额外的模块来实现特定的功能。  
  
首先，文件导入了必要的 PyTorch 模块，包括神经网络模块（`torch.nn`）、功能模块（`torch.nn.functional`）以及检查点工具（`torch.utils.checkpoint`）。同时，还尝试导入了一些自定义的深度可分离卷积实现，分别用于 FP16 和 FP32 精度的计算。  
  
文件中定义了一个 `rel\_pos` 函数，用于生成相对位置的坐标，这对于后续的卷积核生成是必要的。  
  
接下来，定义了 `SMPConv` 类，这是一个自定义的卷积层。它的构造函数接收多个参数，包括输出通道数、卷积核大小、采样点数、步幅、填充和分组数。在初始化过程中，计算并注册了卷积核的坐标，并随机初始化了权重坐标和半径。`forward` 方法中实现了卷积操作，支持 FP32 和 FP16 的输入数据类型。  
  
`make\_kernels` 方法用于生成卷积核，通过计算权重坐标与卷积核坐标之间的差异，并利用这些差异生成最终的卷积核。`radius\_clip` 方法则用于限制半径的范围，确保其在指定的最小值和最大值之间。  
  
`get\_conv2d` 函数用于根据输入参数返回适当的卷积层，支持自定义的 `SMPConv` 或标准的 `nn.Conv2d`。此外，文件中还定义了一些用于批归一化的函数，包括 `get\_bn` 和 `conv\_bn`，这些函数根据全局变量 `use\_sync\_bn` 决定使用同步批归一化还是普通批归一化。  
  
`SMPCNN` 类是一个包含多个卷积层的网络模块，它结合了自定义的 `SMPConv` 和标准卷积层。`SMPCNN\_ConvFFN` 类则实现了一个前馈网络模块，包含两个逐点卷积层和一个非线性激活函数。  
  
最后，`SMPBlock` 类实现了一个包含多个卷积操作的复合模块，结合了逐点卷积和自定义的 `SMPCNN`，并使用残差连接和丢弃路径（Drop Path）机制。  
  
整体来看，这个文件提供了一种灵活的方式来构建复杂的卷积神经网络，尤其适用于需要自定义卷积操作和高效计算的场景。

```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 torch.nn.functional as F  
from ..modules.conv import Conv  
  
# 定义基本的卷积块  
class BasicBlock(nn.Module):  
 def \_\_init\_\_(self, filter\_in, filter\_out):  
 super(BasicBlock, self).\_\_init\_\_()  
 # 定义两个卷积层  
 self.conv1 = Conv(filter\_in, filter\_out, 3) # 第一个卷积层，卷积核大小为3  
 self.conv2 = Conv(filter\_out, filter\_out, 3, act=False) # 第二个卷积层，卷积核大小为3，不使用激活函数  
  
 def forward(self, x):  
 residual = x # 保存输入作为残差  
 out = self.conv1(x) # 通过第一个卷积层  
 out = self.conv2(out) # 通过第二个卷积层  
 out += residual # 添加残差  
 return self.conv1.act(out) # 返回经过激活函数处理的输出  
  
  
# 定义上采样模块  
class Upsample(nn.Module):  
 def \_\_init\_\_(self, in\_channels, out\_channels, scale\_factor=2):  
 super(Upsample, self).\_\_init\_\_()  
 # 定义上采样操作，使用1x1卷积层后接双线性插值上采样  
 self.upsample = nn.Sequential(  
 Conv(in\_channels, out\_channels, 1), # 1x1卷积  
 nn.Upsample(scale\_factor=scale\_factor, mode='bilinear') # 双线性插值上采样  
 )  
  
 def forward(self, x):  
 return self.upsample(x) # 前向传播  
  
  
# 定义自适应特征融合模块（ASFF）  
class ASFF\_2(nn.Module):  
 def \_\_init\_\_(self, inter\_dim=512):  
 super(ASFF\_2, self).\_\_init\_\_()  
 self.inter\_dim = inter\_dim  
 compress\_c = 8 # 压缩通道数  
  
 # 定义权重计算卷积层  
 self.weight\_level\_1 = Conv(self.inter\_dim, compress\_c, 1)  
 self.weight\_level\_2 = Conv(self.inter\_dim, compress\_c, 1)  
 self.weight\_levels = nn.Conv2d(compress\_c \* 2, 2, kernel\_size=1, stride=1, padding=0) # 计算融合权重  
 self.conv = Conv(self.inter\_dim, self.inter\_dim, 3) # 最后的卷积层  
  
 def forward(self, input1, input2):  
 # 计算每个输入的权重  
 level\_1\_weight\_v = self.weight\_level\_1(input1)  
 level\_2\_weight\_v = self.weight\_level\_2(input2)  
  
 # 合并权重并计算最终权重  
 levels\_weight\_v = torch.cat((level\_1\_weight\_v, level\_2\_weight\_v), 1)  
 levels\_weight = self.weight\_levels(levels\_weight\_v)  
 levels\_weight = F.softmax(levels\_weight, dim=1) # 使用softmax归一化权重  
  
 # 融合输入特征  
 fused\_out\_reduced = input1 \* levels\_weight[:, 0:1, :, :] + input2 \* levels\_weight[:, 1:2, :, :]  
 out = self.conv(fused\_out\_reduced) # 通过卷积层  
 return out # 返回融合后的输出  
  
  
# 定义特征金字塔网络（AFPN）  
class AFPN\_P345(nn.Module):  
 def \_\_init\_\_(self, in\_channels=[256, 512, 1024], out\_channels=256, factor=4):  
 super(AFPN\_P345, self).\_\_init\_\_()  
 # 定义输入通道到压缩通道的卷积层  
 self.conv0 = Conv(in\_channels[0], in\_channels[0] // factor, 1)  
 self.conv1 = Conv(in\_channels[1], in\_channels[1] // factor, 1)  
 self.conv2 = Conv(in\_channels[2], in\_channels[2] // factor, 1)  
  
 # 定义特征块  
 self.body = BlockBody\_P345([in\_channels[0] // factor, in\_channels[1] // factor, in\_channels[2] // factor])  
  
 # 定义输出卷积层  
 self.conv00 = Conv(in\_channels[0] // factor, out\_channels, 1)  
 self.conv11 = Conv(in\_channels[1] // factor, out\_channels, 1)  
 self.conv22 = Conv(in\_channels[2] // factor, out\_channels, 1)  
  
 def forward(self, x):  
 x0, x1, x2 = x # 解包输入特征  
 x0 = self.conv0(x0) # 通过卷积层  
 x1 = self.conv1(x1)  
 x2 = self.conv2(x2)  
  
 out0, out1, out2 = self.body([x0, x1, x2]) # 通过特征块  
 out0 = self.conv00(out0) # 输出卷积  
 out1 = self.conv11(out1)  
 out2 = self.conv22(out2)  
 return [out0, out1, out2] # 返回输出特征  
```  
  
### 代码核心部分说明：  
1. \*\*BasicBlock\*\*: 定义了一个基本的卷积块，包含两个卷积层和残差连接。  
2. \*\*Upsample\*\*: 实现了上采样功能，使用1x1卷积和双线性插值。  
3. \*\*ASFF\_2\*\*: 自适应特征融合模块，计算输入特征的权重并融合特征。  
4. \*\*AFPN\_P345\*\*: 特征金字塔网络的实现，包含输入特征的卷积、特征块处理和输出卷积。  
  
这些模块是构建特征金字塔网络的基础，能够有效地处理多尺度特征并进行融合。```

这个程序文件 `afpn.py` 实现了一个用于计算机视觉任务的特征金字塔网络（FPN），具体来说是自适应特征金字塔网络（AFPN）。该网络主要用于目标检测和图像分割等任务，通过不同尺度的特征融合来提高模型的性能。  
  
文件中定义了多个类，首先是 `BasicBlock` 类，它是一个基本的卷积块，包含两个卷积层，第二个卷积层不使用激活函数。它的前向传播方法实现了残差连接，即输入与卷积结果相加，形成更深层次的特征。  
  
接下来是几个下采样和上采样的类，包括 `Downsample\_x2`、`Downsample\_x4`、`Downsample\_x8` 和 `Upsample`，这些类分别实现了不同的下采样和上采样操作，利用卷积层和插值方法来改变特征图的尺寸。  
  
`ASFF\_2`、`ASFF\_3` 和 `ASFF\_4` 类实现了自适应特征融合模块，分别用于处理两个、三个和四个输入特征图。每个模块通过计算权重来融合不同尺度的特征图，并通过卷积层进一步处理融合后的特征。  
  
`BlockBody\_P345` 和 `BlockBody\_P2345` 类是网络的主体部分，分别处理三个和四个输入特征图。它们包含多个卷积块和自适应特征融合模块，通过不同的上采样和下采样操作实现特征的多尺度融合。  
  
`AFPN\_P345` 和 `AFPN\_P2345` 类是网络的顶层结构，负责输入特征图的初步处理和最终输出。它们将输入特征图通过卷积层进行通道数的调整，然后将处理后的特征图传递给主体部分进行进一步的特征融合和处理。  
  
最后，`BlockBody\_P345\_Custom` 和 `BlockBody\_P2345\_Custom` 类允许用户自定义使用的卷积块类型，以便在不同的任务中灵活调整网络结构。  
  
整体来看，这个程序文件实现了一个灵活且强大的特征金字塔网络，能够有效地处理多尺度特征，并在目标检测和图像分割等任务中取得良好的效果。

```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  
 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 # 维度  
 self.dropout = nn.Dropout2d(p=dropout) if ndim == 2 else nn.Dropout3d(p=dropout) if ndim == 3 else nn.Dropout1d(p=dropout) if ndim == 1 else None  
  
 # 检查参数有效性  
 if groups <= 0 or input\_dim % groups != 0 or output\_dim % groups != 0:  
 raise ValueError('groups must be a positive integer and input/output dimensions 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)])  
  
 # 多项式权重初始化  
 poly\_shape = (groups, output\_dim // groups, (input\_dim // groups) \* (degree + 1)) + (kernel\_size,) \* 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) # 缓存以避免重复计算  
 def gram\_poly(self, x, degree):  
 # 计算Gram多项式  
 p0 = x.new\_ones(x.size())  
 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  
 grams\_basis.append(p2)  
 p0, p1 = p1, p2  
  
 return torch.cat(grams\_basis, dim=1)  
  
 def forward\_kag(self, x, group\_index):  
 # 前向传播，处理每个组的输入  
 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):  
 # 前向传播，处理所有组的输入  
 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\_kag(\_x.clone(), group\_ind) # 处理每个组  
 output.append(y.clone())  
 y = torch.cat(output, dim=1) # 合并输出  
 return y  
```  
  
### 代码说明：  
1. \*\*KAGNConvNDLayer类\*\*：这是一个自定义的神经网络层，支持多维卷积（1D、2D、3D）。  
2. \*\*初始化方法\*\*：设置输入输出维度、卷积参数、激活函数等，并初始化卷积层和归一化层。  
3. \*\*beta方法\*\*：计算用于Gram多项式的beta值。  
4. \*\*gram\_poly方法\*\*：计算Gram多项式的基，使用缓存以提高效率。  
5. \*\*forward\_kag方法\*\*：实现前向传播，处理每个组的输入，进行卷积和归一化。  
6. \*\*forward方法\*\*：处理所有组的输入，返回最终输出。```

这个程序文件 `kagn\_conv.py` 定义了一系列用于卷积神经网络的层，主要是基于 KAGN（Kochawongwat 的一种卷积层）设计的。程序使用了 PyTorch 框架，包含了多维卷积层的实现，包括一维、二维和三维卷积层。  
  
首先，`KAGNConvNDLayer` 类是一个通用的卷积层类，支持任意维度的卷积。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、丢弃率等。构造函数中还定义了基础激活函数（使用 SiLU 激活函数），并根据输入的维度选择相应的丢弃层（Dropout）。在初始化过程中，程序会检查分组数和输入输出维度的有效性，并创建多个卷积层和归一化层（Layer Normalization）。  
  
接下来，程序定义了多项式权重和 beta 权重，这些权重用于后续的计算。权重的初始化使用了 Kaiming 均匀分布，以便在训练开始时提供更好的性能。  
  
`beta` 方法用于计算与 Legendre 多项式相关的 beta 值，而 `gram\_poly` 方法则计算给定度数的 Gram 多项式。该方法使用了缓存机制，以避免重复计算。  
  
`forward\_kag` 方法是该层的前向传播逻辑。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，输入会被归一化到 [-1, 1] 的范围，以便于计算 Legendre 多项式。之后，计算 Gram 多项式基，并通过卷积权重函数进行卷积操作，最后通过归一化层和激活函数得到输出。  
  
`forward` 方法负责处理输入数据，首先将输入按组分割，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。  
  
在此基础上，程序还定义了三个具体的卷积层类：`KAGNConv3DLayer`、`KAGNConv2DLayer` 和 `KAGNConv1DLayer`，分别用于三维、二维和一维卷积。这些类通过调用 `KAGNConvNDLayer` 的构造函数，传入相应的卷积和归一化类，简化了不同维度卷积层的实现。  
  
总体来说，这个程序文件实现了一个灵活且可扩展的卷积层结构，适用于多种维度的卷积操作，并引入了 Gram 多项式的概念，以增强网络的表达能力。

### 整体功能和构架概括  
  
该工程包含多个文件，每个文件实现了不同的深度学习模块和功能，主要用于构建和训练卷积神经网络（CNN）以及特征提取和融合。整体架构设计灵活，支持多种卷积操作和特征处理方式，适用于计算机视觉任务，如目标检测和图像分割。  
  
- \*\*VanillaNet.py\*\*：实现了一个基础的卷积神经网络架构，支持多种配置和自定义激活函数，适合于图像处理任务。  
- \*\*SMPConv.py\*\*：定义了自适应卷积层，支持不同精度的计算，提供了特征融合和批归一化的功能，适用于构建复杂的卷积网络。  
- \*\*afpn.py\*\*：实现了自适应特征金字塔网络（AFPN），通过多尺度特征融合来提高目标检测和分割的性能。  
- \*\*kagn\_conv.py\*\*：定义了基于 KAGN 的卷积层，支持多维卷积操作，引入了 Gram 多项式的概念，以增强网络的表达能力。  
  
### 文件功能整理表  
  
| 文件名 | 功能描述 |  
|------------------|----------------------------------------------------------------------------------------------|  
| VanillaNet.py | 实现基础卷积神经网络架构，支持多种配置和自定义激活函数，适用于图像处理任务。 |  
| SMPConv.py | 定义自适应卷积层，支持不同精度计算，提供特征融合和批归一化功能，适合构建复杂卷积网络。 |  
| afpn.py | 实现自适应特征金字塔网络（AFPN），通过多尺度特征融合提高目标检测和分割性能。 |  
| kagn\_conv.py | 定义基于 KAGN 的卷积层，支持多维卷积操作，引入 Gram 多项式概念，增强网络表达能力。 |  
  
这个表格清晰地展示了每个文件的主要功能，便于理解整个工程的结构和目的。