# 改进yolo11-ASF-P2等200+全套创新点大全：航拍图油罐检测系统源码＆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  
  
随着城市化进程的加快，油罐作为重要的能源储存设施，其安全性和监测需求日益凸显。传统的油罐检测方法多依赖人工巡检，效率低下且容易受到人为因素的影响，难以满足现代化管理的需求。因此，基于计算机视觉和深度学习技术的自动化检测系统应运而生，成为提升油罐监测效率的重要手段。近年来，YOLO（You Only Look Once）系列目标检测算法因其高效性和实时性，广泛应用于各类目标检测任务中，尤其是在复杂环境下的图像处理方面表现突出。  
  
本研究旨在基于改进的YOLOv11算法，构建一个高效的航拍图油罐检测系统。我们选用的“2020 Tanks 360”数据集包含352幅图像，涵盖了四种油罐类别（AB、DT、FAT、FXT），为模型的训练和测试提供了丰富的样本。这一数据集的多样性和代表性，使得模型能够在不同场景下进行有效的油罐识别和分类。通过对YOLOv11算法的改进，我们期望在提升检测精度的同时，减少计算资源的消耗，以实现实时监测的目标。  
  
此外，航拍技术的应用为油罐检测提供了全新的视角，能够有效克服地面视角下的遮挡和视野限制，提升检测的全面性和准确性。随着无人机技术的不断发展，结合深度学习的油罐检测系统不仅能够提高检测效率，还能在一定程度上降低人力成本，提升油罐管理的智能化水平。因此，本研究不仅具有重要的理论意义，也为实际应用提供了切实可行的解决方案，推动油气行业的数字化转型与智能化发展。

## 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“2020 Tanks 360”，旨在为改进YOLOv11的航拍图油罐检测系统提供丰富的训练数据。该数据集包含四个主要类别，分别为AB、DT、FAT和FXT，这些类别涵盖了不同类型的油罐，具有显著的特征差异。通过对这些类别的深入分析和标注，我们能够有效地训练模型，使其在复杂的航拍场景中准确识别和分类油罐。  
  
“2020 Tanks 360”数据集的构建基于多种航拍图像，这些图像来源于不同的地理位置和环境条件，确保了数据集的多样性和广泛性。每个类别的样本数量经过精心设计，以保证模型在训练过程中能够充分学习到每个类别的特征，同时避免过拟合现象的发生。数据集中不仅包含了清晰的油罐图像，还包括了在不同光照、天气和视角下拍摄的样本，进一步增强了模型的鲁棒性。  
  
此外，数据集还配备了详细的标注信息，包括每个油罐的边界框和类别标签，这为YOLOv11的训练提供了必要的监督信号。通过使用“2020 Tanks 360”数据集，我们期望能够显著提升航拍图像中油罐检测的准确性和效率，为相关领域的应用提供更为可靠的技术支持。随着模型的不断优化，我们相信这一数据集将为未来的油罐检测系统奠定坚实的基础，并推动相关研究的深入发展。

## 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创新点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核心改进源码讲解

```以下是经过精简和详细注释的核心代码部分：  
  
```python  
# 导入必要的模块  
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。  
  
 参数:  
 preds: 模型的预测结果  
 img: 输入图像  
 orig\_imgs: 原始图像（可能是torch.Tensor或numpy数组）  
  
 返回:  
 results: 包含处理后结果的Results对象列表  
 """  
 # 应用非极大值抑制（NMS）来过滤预测框  
 preds = ops.non\_max\_suppression(  
 preds,  
 self.args.conf, # 置信度阈值  
 self.args.iou, # IOU阈值  
 agnostic=self.args.agnostic\_nms, # 是否使用类别无关的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):  
 orig\_imgs = ops.convert\_torch2numpy\_batch(orig\_imgs)  
  
 results = [] # 存储处理后的结果  
 for i, pred in enumerate(preds):  
 orig\_img = orig\_imgs[i] # 获取对应的原始图像  
 # 将预测框的坐标缩放到原始图像的尺寸  
 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  
 # 创建Results对象并添加到结果列表中  
 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  
```  
  
### 代码分析：  
1. \*\*类定义\*\*：`DetectionPredictor`类继承自`BasePredictor`，用于处理检测模型的预测。  
2. \*\*后处理方法\*\*：`postprocess`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）和缩放预测框。  
3. \*\*非极大值抑制\*\*：通过`ops.non\_max\_suppression`函数过滤掉重叠的预测框，以提高检测精度。  
4. \*\*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。  
5. \*\*结果构建\*\*：遍历每个预测结果，缩放预测框，并将结果存储在`Results`对象中，最终返回结果列表。```

这个文件 `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。它继承自 `BasePredictor` 类，专门用于处理检测模型的预测任务。  
  
在这个文件中，定义了一个名为 `DetectionPredictor` 的类。这个类的主要功能是进行目标检测，并对检测结果进行后处理。类的文档字符串中提供了一个示例，展示了如何使用这个类进行预测。用户可以通过传入模型文件路径和数据源来初始化 `DetectionPredictor` 的实例，并调用 `predict\_cli()` 方法进行预测。  
  
类中有一个名为 `postprocess` 的方法，负责对模型的预测结果进行后处理。这个方法接收三个参数：`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（NMS），以去除冗余的检测框。这个过程会根据设定的置信度阈值、IoU（Intersection over Union）阈值以及其他参数来筛选出最终的检测框。  
  
接下来，方法检查 `orig\_imgs` 是否为列表，如果不是，则将其转换为 NumPy 数组，以便后续处理。然后，方法会遍历每个预测结果，使用 `ops.scale\_boxes` 函数将检测框的坐标从缩放后的图像空间转换回原始图像空间。最后，将每个检测结果封装成 `Results` 对象，并将其添加到结果列表中。  
  
最终，`postprocess` 方法返回一个包含所有检测结果的列表，每个结果都包含原始图像、图像路径、模型的类别名称和检测框信息。这使得用户能够方便地获取和使用目标检测的结果。

```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  
from timm.models.layers import DropPath, trunc\_normal\_  
  
# 定义一个卷积层加批归一化的组合模块  
class ConvBN(torch.nn.Sequential):  
 def \_\_init\_\_(self, in\_planes, out\_planes, kernel\_size=1, stride=1, padding=0, dilation=1, groups=1, with\_bn=True):  
 super().\_\_init\_\_()  
 # 添加卷积层  
 self.add\_module('conv', torch.nn.Conv2d(in\_planes, out\_planes, kernel\_size, stride, padding, dilation, groups))  
 # 如果需要，添加批归一化层  
 if with\_bn:  
 self.add\_module('bn', torch.nn.BatchNorm2d(out\_planes))  
 # 初始化批归一化的权重和偏置  
 torch.nn.init.constant\_(self.bn.weight, 1)  
 torch.nn.init.constant\_(self.bn.bias, 0)  
  
# 定义网络中的基本模块  
class Block(nn.Module):  
 def \_\_init\_\_(self, dim, mlp\_ratio=3, drop\_path=0.):  
 super().\_\_init\_\_()  
 # 深度可分离卷积  
 self.dwconv = ConvBN(dim, dim, 7, 1, (7 - 1) // 2, groups=dim, with\_bn=True)  
 # 线性变换层  
 self.f1 = ConvBN(dim, mlp\_ratio \* dim, 1, with\_bn=False)  
 self.f2 = ConvBN(dim, mlp\_ratio \* dim, 1, with\_bn=False)  
 self.g = ConvBN(mlp\_ratio \* dim, dim, 1, with\_bn=True)  
 self.dwconv2 = ConvBN(dim, dim, 7, 1, (7 - 1) // 2, groups=dim, with\_bn=False)  
 self.act = nn.ReLU6() # 激活函数  
 self.drop\_path = DropPath(drop\_path) if drop\_path > 0. else nn.Identity() # 随机深度  
  
 def forward(self, x):  
 input = x # 保存输入  
 x = self.dwconv(x) # 深度可分离卷积  
 x1, x2 = self.f1(x), self.f2(x) # 线性变换  
 x = self.act(x1) \* x2 # 元素级乘法  
 x = self.dwconv2(self.g(x)) # 进一步处理  
 x = input + self.drop\_path(x) # 残差连接  
 return x  
  
# 定义StarNet网络结构  
class StarNet(nn.Module):  
 def \_\_init\_\_(self, base\_dim=32, depths=[3, 3, 12, 5], mlp\_ratio=4, drop\_path\_rate=0.0, num\_classes=1000, \*\*kwargs):  
 super().\_\_init\_\_()  
 self.num\_classes = num\_classes  
 self.in\_channel = 32  
 # stem层  
 self.stem = nn.Sequential(ConvBN(3, self.in\_channel, kernel\_size=3, stride=2, padding=1), nn.ReLU6())  
 dpr = [x.item() for x in torch.linspace(0, drop\_path\_rate, sum(depths))] # 随机深度  
 # 构建网络的各个阶段  
 self.stages = nn.ModuleList()  
 cur = 0  
 for i\_layer in range(len(depths)):  
 embed\_dim = base\_dim \* 2 \*\* i\_layer # 当前层的嵌入维度  
 down\_sampler = ConvBN(self.in\_channel, embed\_dim, 3, 2, 1) # 下采样层  
 self.in\_channel = embed\_dim  
 blocks = [Block(self.in\_channel, mlp\_ratio, dpr[cur + i]) for i in range(depths[i\_layer])] # 构建Block  
 cur += depths[i\_layer]  
 self.stages.append(nn.Sequential(down\_sampler, \*blocks)) # 将下采样层和Block组合成一个阶段  
   
 self.channel = [i.size(1) for i in self.forward(torch.randn(1, 3, 640, 640))] # 计算每个阶段的输出通道数  
 self.apply(self.\_init\_weights) # 初始化权重  
  
 def \_init\_weights(self, m):  
 # 权重初始化  
 if isinstance(m, (nn.Linear, nn.Conv2d)):  
 trunc\_normal\_(m.weight, std=.02) # 截断正态分布初始化  
 if isinstance(m, nn.Linear) and m.bias is not None:  
 nn.init.constant\_(m.bias, 0) # 偏置初始化为0  
 elif isinstance(m, (nn.LayerNorm, nn.BatchNorm2d)):  
 nn.init.constant\_(m.bias, 0) # 偏置初始化为0  
 nn.init.constant\_(m.weight, 1.0) # 权重初始化为1.0  
  
 def forward(self, x):  
 features = [] # 存储特征  
 x = self.stem(x) # 通过stem层  
 features.append(x) # 保存特征  
 for stage in self.stages:  
 x = stage(x) # 通过每个阶段  
 features.append(x) # 保存特征  
 return features # 返回所有特征  
  
# 定义不同规模的StarNet模型  
def starnet\_s1(pretrained=False, \*\*kwargs):  
 model = StarNet(24, [2, 2, 8, 3], \*\*kwargs) # 创建模型  
 return model  
  
def starnet\_s2(pretrained=False, \*\*kwargs):  
 model = StarNet(32, [1, 2, 6, 2], \*\*kwargs) # 创建模型  
 return model  
  
def starnet\_s3(pretrained=False, \*\*kwargs):  
 model = StarNet(32, [2, 2, 8, 4], \*\*kwargs) # 创建模型  
 return model  
  
def starnet\_s4(pretrained=False, \*\*kwargs):  
 model = StarNet(32, [3, 3, 12, 5], \*\*kwargs) # 创建模型  
 return model  
```  
  
### 代码说明：  
1. \*\*ConvBN 类\*\*：定义了一个包含卷积层和批归一化层的组合模块，方便构建网络。  
2. \*\*Block 类\*\*：定义了网络中的基本构建块，包含深度可分离卷积、线性变换和元素级乘法等操作。  
3. \*\*StarNet 类\*\*：定义了整个网络结构，包括stem层和多个阶段，每个阶段由下采样层和多个Block组成。  
4. \*\*权重初始化\*\*：使用截断正态分布初始化卷积层和线性层的权重。  
5. \*\*模型构建函数\*\*：提供了不同规模的StarNet模型构建函数，便于创建不同配置的模型。```

这个程序文件实现了一个名为StarNet的神经网络模型，主要用于深度学习任务。文件开头的文档字符串说明了该模型的设计理念，强调了简化网络结构以突出逐元素乘法的关键贡献，并提到在训练过程中没有使用层缩放和指数移动平均（EMA），这些设计选择可能会进一步提升性能。  
  
程序中首先导入了必要的库，包括PyTorch和一些模型层的组件。接着定义了一个包含多个模型版本的列表和相应的模型权重下载链接。  
  
接下来定义了一个名为`ConvBN`的类，它是一个组合层，包含卷积层和批归一化层。该类的构造函数允许用户指定输入和输出通道数、卷积核大小、步幅、填充、扩张率、分组卷积以及是否使用批归一化。若使用批归一化，则初始化其权重为1，偏置为0。  
  
然后定义了一个`Block`类，它是StarNet的基本构建块。该类包含多个卷积层和激活函数，采用逐元素乘法的方式将两个特征图结合。`Block`类的前向传播方法实现了特征的提取和处理，最后通过残差连接将输入与输出相加。  
  
`StarNet`类是整个模型的核心，包含多个阶段，每个阶段由下采样层和多个`Block`组成。构造函数中，模型的深度、基础维度、每个阶段的层数等参数可以灵活设置。通过随机深度的方式，模型在训练时可以随机丢弃某些层，以增强模型的泛化能力。模型的权重初始化通过`\_init\_weights`方法进行，确保各层的权重和偏置在训练开始时具有合适的初始值。  
  
`StarNet`的前向传播方法返回每个阶段的特征图，便于后续的特征提取和分析。  
  
最后，文件定义了多个函数，用于创建不同配置的StarNet模型，包括是否加载预训练权重的选项。这些函数分别对应不同的模型规模，提供了灵活的接口供用户使用。  
  
总体来说，这个程序文件展示了StarNet模型的设计与实现，强调了其结构的简洁性和高效性，适合用于各种计算机视觉任务。

```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 as nn  
import math  
import itertools  
  
class Attention4D(nn.Module):  
 def \_\_init\_\_(self, dim=384, key\_dim=32, num\_heads=8, attn\_ratio=4, resolution=7, act\_layer=nn.ReLU, stride=None):  
 super().\_\_init\_\_()  
 self.num\_heads = num\_heads # 注意力头的数量  
 self.scale = key\_dim \*\* -0.5 # 缩放因子  
 self.key\_dim = key\_dim # 键的维度  
 self.nh\_kd = key\_dim \* num\_heads # 头的总维度  
  
 # 如果有步幅，则调整分辨率并添加卷积层  
 if stride is not None:  
 self.resolution = math.ceil(resolution / stride)  
 self.stride\_conv = nn.Sequential(  
 nn.Conv2d(dim, dim, kernel\_size=3, stride=stride, padding=1, groups=dim),  
 nn.BatchNorm2d(dim),  
 )  
 self.upsample = nn.Upsample(scale\_factor=stride, mode='bilinear')  
 else:  
 self.resolution = resolution  
 self.stride\_conv = None  
 self.upsample = None  
  
 self.N = self.resolution \*\* 2 # 每个头的注意力矩阵的大小  
 self.d = int(attn\_ratio \* key\_dim) # 注意力输出的维度  
 self.dh = self.d \* num\_heads # 所有头的输出维度  
 self.attn\_ratio = attn\_ratio # 注意力比率  
  
 # 定义查询、键、值的卷积层  
 self.q = nn.Sequential(nn.Conv2d(dim, self.num\_heads \* self.key\_dim, 1), nn.BatchNorm2d(self.num\_heads \* self.key\_dim))  
 self.k = nn.Sequential(nn.Conv2d(dim, self.num\_heads \* self.key\_dim, 1), nn.BatchNorm2d(self.num\_heads \* self.key\_dim))  
 self.v = nn.Sequential(nn.Conv2d(dim, self.num\_heads \* self.d, 1), nn.BatchNorm2d(self.num\_heads \* self.d))  
  
 # 定义局部值的卷积层  
 self.v\_local = nn.Sequential(  
 nn.Conv2d(self.num\_heads \* self.d, self.num\_heads \* self.d, kernel\_size=3, stride=1, padding=1, groups=self.num\_heads \* self.d),  
 nn.BatchNorm2d(self.num\_heads \* self.d),  
 )  
  
 # 定义用于注意力的卷积层  
 self.talking\_head1 = nn.Conv2d(self.num\_heads, self.num\_heads, kernel\_size=1, stride=1, padding=0)  
 self.talking\_head2 = nn.Conv2d(self.num\_heads, self.num\_heads, kernel\_size=1, stride=1, padding=0)  
  
 # 定义输出的卷积层  
 self.proj = nn.Sequential(act\_layer(), nn.Conv2d(self.dh, dim, 1), nn.BatchNorm2d(dim))  
  
 # 计算注意力偏置  
 points = list(itertools.product(range(self.resolution), range(self.resolution)))  
 N = len(points)  
 attention\_offsets = {}  
 idxs = []  
 for p1 in points:  
 for p2 in points:  
 offset = (abs(p1[0] - p2[0]), abs(p1[1] - p2[1]))  
 if offset not in attention\_offsets:  
 attention\_offsets[offset] = len(attention\_offsets)  
 idxs.append(attention\_offsets[offset])  
 self.attention\_biases = nn.Parameter(torch.zeros(num\_heads, len(attention\_offsets))) # 注意力偏置参数  
 self.register\_buffer('attention\_bias\_idxs', torch.LongTensor(idxs).view(N, N)) # 注册注意力偏置索引  
  
 @torch.no\_grad()  
 def train(self, mode=True):  
 super().train(mode)  
 if mode and hasattr(self, 'ab'):  
 del self.ab # 删除训练模式下的偏置  
 else:  
 self.ab = self.attention\_biases[:, self.attention\_bias\_idxs] # 在训练模式下使用偏置  
  
 def forward(self, x): # 前向传播  
 B, C, H, W = x.shape # 获取输入的形状  
 if self.stride\_conv is not None:  
 x = self.stride\_conv(x) # 如果有步幅，应用卷积  
  
 # 计算查询、键、值  
 q = self.q(x).flatten(2).reshape(B, self.num\_heads, -1, self.N).permute(0, 1, 3, 2)  
 k = self.k(x).flatten(2).reshape(B, self.num\_heads, -1, self.N).permute(0, 1, 2, 3)  
 v = self.v(x)  
 v\_local = self.v\_local(v)  
 v = v.flatten(2).reshape(B, self.num\_heads, -1, self.N).permute(0, 1, 3, 2)  
  
 # 计算注意力权重  
 attn = (q @ k) \* self.scale + (self.attention\_biases[:, self.attention\_bias\_idxs] if self.training else self.ab)  
 attn = self.talking\_head1(attn)  
 attn = attn.softmax(dim=-1) # 应用softmax  
 attn = self.talking\_head2(attn)  
  
 # 计算输出  
 x = (attn @ v)  
 out = x.transpose(2, 3).reshape(B, self.dh, self.resolution, self.resolution) + v\_local  
 if self.upsample is not None:  
 out = self.upsample(out) # 如果有上采样，应用上采样  
  
 out = self.proj(out) # 应用输出卷积  
 return out  
  
  
class EfficientFormerV2(nn.Module):  
 def \_\_init\_\_(self, layers, embed\_dims=None, mlp\_ratios=4, downsamples=None, num\_classes=1000):  
 super().\_\_init\_\_()  
 self.patch\_embed = nn.Sequential(  
 nn.Conv2d(3, embed\_dims[0], kernel\_size=3, stride=2, padding=1),  
 nn.BatchNorm2d(embed\_dims[0]),  
 nn.ReLU(),  
 ) # 初始的卷积层用于嵌入  
  
 network = []  
 for i in range(len(layers)):  
 # 为每一层添加注意力和前馈网络块  
 network.append(Attention4D(dim=embed\_dims[i], resolution=7))  
 if downsamples[i]: # 如果需要下采样  
 network.append(nn.Conv2d(embed\_dims[i], embed\_dims[i + 1], kernel\_size=3, stride=2, padding=1))  
  
 self.network = nn.ModuleList(network) # 将网络模块化  
  
 def forward(self, x):  
 x = self.patch\_embed(x) # 嵌入输入  
 for block in self.network:  
 x = block(x) # 通过网络块前向传播  
 return x # 返回输出  
```  
  
### 代码核心部分说明：  
1. \*\*Attention4D类\*\*：实现了一个四维注意力机制，包括查询、键、值的计算和注意力权重的生成。该类支持不同的分辨率和步幅，并在前向传播中计算输出。  
2. \*\*EfficientFormerV2类\*\*：构建了一个高效的Transformer模型，包括嵌入层和多个注意力块。它将输入图像嵌入到特征空间，并通过多个层进行处理，最终输出特征图。  
  
这些部分是模型的核心，负责特征提取和注意力机制的实现。```

这个程序文件实现了一个名为 `EfficientFormerV2` 的深度学习模型，主要用于图像处理任务。代码中包含了模型的定义、各个组件的实现以及模型的实例化和测试。  
  
首先，文件导入了一些必要的库，包括 PyTorch 及其相关模块，数学库和一些类型提示工具。接着，定义了一些与模型结构相关的参数，包括不同规模（如 S0、S1、S2 和 L）的宽度和深度，这些参数将用于构建不同的模型变体。  
  
在模型的核心部分，定义了多个类，每个类实现了模型的不同功能。例如，`Attention4D` 类实现了一个四维注意力机制，`LGQuery` 类用于局部查询，`Embedding` 类用于输入的嵌入处理，`Mlp` 类实现了多层感知机，`AttnFFN` 和 `FFN` 类则分别实现了带有注意力机制和普通前馈网络的功能。  
  
`EfficientFormerV2` 类是模型的主类，构造函数中接受多个参数来定义模型的结构，包括层数、嵌入维度、下采样策略等。模型的前向传播方法 `forward` 将输入数据传递通过网络的各个层，并在需要时进行规范化处理。  
  
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，`efficientformerv2\_s0`、`efficientformerv2\_s1`、`efficientformerv2\_s2` 和 `efficientformerv2\_l` 函数用于创建不同规模的模型实例，并可选择加载预训练权重。  
  
在文件的最后部分，提供了一个测试代码块，通过随机生成的输入数据来测试不同规模的模型，并打印出每个模型输出的尺寸。这一部分代码在直接运行文件时会被执行，确保模型的正确性和功能性。  
  
总体而言，这个文件实现了一个高效的图像处理模型，结合了注意力机制和多层感知机，适用于各种计算机视觉任务。

```以下是经过简化和注释的核心代码部分，保留了主要的功能和逻辑：  
  
```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  
 参数:  
 us: 输入张量，形状为 (B, G \* D, L)  
 dts: 时间增量，形状为 (B, G \* D, L)  
 As: 权重矩阵，形状为 (G \* D, N)  
 Bs: 权重矩阵，形状为 (B, G, N, L)  
 Cs: 权重矩阵，形状为 (B, G, N, L)  
 Ds: 偏置项，形状为 (G \* D)  
 delta\_bias: 可选的偏置项，形状为 (G \* D)  
 delta\_softplus: 是否应用softplus激活  
 return\_last\_state: 是否返回最后的状态  
 chunksize: 每次处理的块大小  
 """  
   
 def selective\_scan\_chunk(us, dts, As, Bs, Cs, hprefix):  
 """  
 处理单个块的选择性扫描  
 参数:  
 us: 输入张量，形状为 (L, B, G, D)  
 dts: 时间增量，形状为 (L, B, G, D)  
 As: 权重矩阵，形状为 (G, D, N)  
 Bs: 权重矩阵，形状为 (L, B, G, N)  
 Cs: 权重矩阵，形状为 (B, G, N, L)  
 hprefix: 前一个状态，形状为 (B, G, D, N)  
 返回:  
 ys: 输出张量，形状为 (L, B, G, D)  
 hs: 状态张量，形状为 (L, B, G, D, N)  
 """  
 ts = dts.cumsum(dim=0) # 计算时间增量的累积和  
 Ats = torch.einsum("gdn,lbgd->lbgdn", As, ts).exp() # 计算权重的指数  
 rAts = Ats # 归一化权重  
 duts = dts \* us # 计算输入与时间增量的乘积  
 dtBus = torch.einsum("lbgd,lbgn->lbgdn", duts, Bs) # 计算中间结果  
 hs\_tmp = rAts \* (dtBus / rAts).cumsum(dim=0) # 更新状态  
 hs = hs\_tmp + Ats \* hprefix.unsqueeze(0) # 结合前一个状态  
 ys = torch.einsum("lbgn,lbgdn->lbgd", Cs, hs) # 计算输出  
 return ys, hs  
  
 # 数据类型和设备设置  
 dtype = torch.float32  
 inp\_dtype = us.dtype  
 has\_D = Ds is not None  
 if chunksize < 1:  
 chunksize = Bs.shape[-1]  
  
 # 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)  
 if delta\_softplus:  
 dts = F.softplus(dts)  
  
 # 维度调整  
 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]  
  
 oys = [] # 存储输出  
 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)  
 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 # 加上偏置项  
 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())  
```  
  
### 代码说明：  
1. \*\*函数定义\*\*：`selective\_scan\_easy`是主要的选择性扫描函数，接收多个输入参数并返回处理后的输出。  
2. \*\*内部函数\*\*：`selective\_scan\_chunk`处理每个块的输入数据，计算输出和状态。  
3. \*\*数据预处理\*\*：对输入数据进行类型转换和维度调整，以便后续计算。  
4. \*\*循环处理\*\*：使用循环对输入数据进行分块处理，更新状态并存储输出。  
5. \*\*返回结果\*\*：根据参数决定是否返回最后的状态。  
  
此代码实现了选择性扫描的基本逻辑，适用于处理时间序列数据的模型。```

这个程序文件 `test\_selective\_scan\_easy.py` 主要实现了一个名为 `selective\_scan\_easy` 的函数，该函数用于在深度学习模型中进行选择性扫描操作。该操作通常用于处理序列数据，特别是在递归神经网络（RNN）或类似的模型中。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`（用于深度学习的主要库）、`math`、`functools`、`pytest`（用于测试）以及 `einops`（用于张量重排）。接着，定义了一个常量 `MODE`，可以选择不同的模式。  
  
`selective\_scan\_easy` 函数的参数包括：  
- `us`：输入张量，形状为 `(B, G \* D, L)`，其中 `B` 是批量大小，`G` 是组数，`D` 是维度，`L` 是序列长度。  
- `dts`：时间增量张量，形状与 `us` 相同。  
- `As`、`Bs`、`Cs`、`Ds`：用于计算的参数张量，分别表示不同的权重矩阵。  
- `delta\_bias` 和 `delta\_softplus`：可选参数，用于调整时间增量。  
- `return\_last\_state`：布尔值，指示是否返回最后的状态。  
- `chunksize`：用于分块处理的大小。  
  
在函数内部，首先定义了一个内部函数 `selective\_scan\_chunk`，用于处理输入的一个块。该函数实现了选择性扫描的核心逻辑，包括计算中间状态和输出。  
  
接下来，函数对输入数据进行了一系列的预处理，包括数据类型转换、形状调整等。然后，函数使用循环对输入序列进行分块处理，调用 `selective\_scan\_chunk` 函数计算每个块的输出，并将结果存储在 `oys` 列表中。  
  
最后，函数将所有块的输出合并，并根据需要返回最后的状态。返回的结果是一个张量，形状为 `(B, -1, L)`，其中 `-1` 表示维度的自动推导。  
  
接下来，定义了一个 `SelectiveScanEasy` 类，继承自 `torch.autograd.Function`，用于实现自定义的前向和反向传播操作。该类的 `forward` 方法实现了选择性扫描的前向计算，`backward` 方法实现了反向传播计算，计算梯度。  
  
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`），这些版本可能在实现细节上有所不同，以优化性能或适应不同的使用场景。  
  
最后，文件中包含了一个测试函数 `test\_selective\_scan`，使用 `pytest` 进行参数化测试，以验证选择性扫描函数的正确性。测试涉及多个参数组合，确保函数在各种情况下都能正常工作。  
  
整体来看，这个程序文件实现了一个复杂的选择性扫描机制，结合了深度学习中的自定义操作和自动微分功能，适用于处理序列数据的任务。

```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 as nn  
import torch.nn.functional as F  
  
# 定义一个函数用于生成相对位置的坐标  
def rel\_pos(kernel\_size):  
 # 生成一个从-1到1的线性空间，步数为kernel\_size  
 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SMPConv类，继承自nn.Module  
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 # 点的数量  
 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，不会被视为模型参数  
  
 # 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.) # 截断正态分布初始化  
 self.weight\_coord = nn.Parameter(weight\_coord) # 权重坐标作为可训练参数  
  
 # 半径参数初始化  
 self.radius = nn.Parameter(torch.empty(1, n\_points).unsqueeze(-1).unsqueeze(-1))  
 self.radius.data.fill\_(value=self.init\_radius) # 填充初始半径  
  
 # 权重初始化  
 weights = torch.empty(1, planes, n\_points)  
 nn.init.trunc\_normal\_(weights, std=.02) # 截断正态分布初始化  
 self.weights = nn.Parameter(weights) # 权重作为可训练参数  
  
 def forward(self, x):  
 # 生成卷积核并增加维度  
 kernels = self.make\_kernels().unsqueeze(1)  
 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不同的卷积实现  
 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):  
 # 计算卷积核的差异  
 diff = self.weight\_coord.unsqueeze(-2) - self.kernel\_coord.reshape(1, 2, -1).transpose(1, 2) # [1, n\_points, kernel\_size^2, 2]  
 diff = diff.transpose(2, 3).reshape(1, self.n\_points, 2, self.kernel\_size, self.kernel\_size) # 重新调整形状  
 diff = F.relu(1 - torch.sum(torch.abs(diff), dim=2) / self.radius) # 计算ReLU激活后的差异  
  
 # 计算卷积核  
 kernels = torch.matmul(self.weights, diff.reshape(1, self.n\_points, -1)) # [1, planes, kernel\_size\*kernel\_size]  
 kernels = kernels.reshape(1, self.planes, \*self.kernel\_coord.shape[2:]) # [1, planes, kernel\_size, kernel\_size]  
 kernels = kernels.squeeze(0) # 去掉多余的维度  
 kernels = torch.flip(kernels.permute(0, 2, 1), dims=(1,)) # 翻转卷积核  
 return kernels  
  
# 定义SMPCNN类，继承自nn.Module  
class SMPCNN(nn.Module):  
 def \_\_init\_\_(self, in\_channels, out\_channels, kernel\_size, stride, groups, n\_points=None, n\_points\_divide=4):  
 super().\_\_init\_\_()  
 self.kernel\_size = kernel\_size  
 if n\_points is None:  
 n\_points = int((kernel\_size\*\*2) // n\_points\_divide) # 计算点的数量  
  
 padding = kernel\_size // 2 # 计算填充  
 self.smp = conv\_bn(in\_channels=in\_channels, out\_channels=out\_channels, kernel\_size=kernel\_size,  
 stride=stride, padding=padding, dilation=1, groups=groups, n\_points=n\_points)  
   
 self.small\_kernel = 5 # 小卷积核大小  
 self.small\_conv = Conv(in\_channels, out\_channels, self.small\_kernel, stride, self.small\_kernel // 2, groups, act=False) # 小卷积层  
  
 def forward(self, inputs):  
 out = self.smp(inputs) # 通过SMP卷积层  
 out += self.small\_conv(inputs) # 加上小卷积层的输出  
 return out  
```  
  
### 代码核心部分说明：  
1. \*\*`rel\_pos`函数\*\*：生成卷积核的相对位置坐标，用于计算卷积核的相对位置。  
2. \*\*`SMPConv`类\*\*：自定义的卷积层，使用可训练的权重坐标和半径来生成卷积核，并根据输入数据类型选择合适的卷积实现。  
3. \*\*`make\_kernels`方法\*\*：计算并生成卷积核，使用ReLU激活函数处理差异。  
4. \*\*`SMPCNN`类\*\*：结合了SMP卷积和小卷积层的网络结构，能够处理输入并输出特征图。```

这个程序文件`SMPConv.py`主要实现了一种新的卷积模块，名为`SMPConv`，以及相关的网络结构和辅助函数。文件中使用了PyTorch库来构建神经网络模型，具体功能和结构如下：  
  
首先，文件导入了必要的库，包括PyTorch的核心模块和一些自定义的模块。`SMPConv`类是文件的核心部分，它继承自`nn.Module`，表示一个可训练的神经网络模块。在`\_\_init\_\_`方法中，定义了多个参数，包括输出通道数、卷积核大小、采样点数、步幅、填充和分组卷积的设置。  
  
`rel\_pos`函数用于生成相对位置的坐标，创建一个形状为`(1, 2, kernel\_size^2)`的张量，表示卷积核中每个点的相对位置。`SMPConv`类中还定义了`make\_kernels`方法，该方法用于生成卷积核，通过计算权重坐标和相对位置坐标之间的差异，并应用ReLU激活函数，最终生成适用于卷积操作的核。  
  
在`forward`方法中，输入数据经过`make\_kernels`生成的卷积核进行卷积操作，使用了自定义的深度可分离卷积实现，支持FP32和FP16两种数据类型。  
  
此外，文件中还定义了一些辅助函数，如`get\_conv2d`、`get\_bn`、`conv\_bn`和`conv\_bn\_relu`，这些函数用于创建不同类型的卷积层和批归一化层。`fuse\_bn`函数则用于将卷积层和批归一化层融合，以提高推理效率。  
  
`SMPCNN`类是一个更复杂的网络结构，它结合了`SMPConv`和一个小卷积层，以实现更好的特征提取。`SMPCNN\_ConvFFN`类则实现了一个前馈网络，包含两个逐点卷积层和一个GELU激活函数。  
  
最后，`SMPBlock`类是一个基本的模块，包含两个逐点卷积层和一个大卷积层，采用残差连接和DropPath策略来提高网络的表现。  
  
总体而言，这个文件实现了一种新型的卷积结构，结合了深度可分离卷积和其他卷积操作，旨在提高模型的表达能力和计算效率。

```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  
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  
 self.conv2 = Conv(filter\_out, filter\_out, 3, act=False) # 第二个卷积层，不使用激活函数  
  
 def forward(self, x):  
 residual = x # 保存输入以便后续加回去  
 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 # 残差连接  
 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\_\_()  
 # 定义上采样的卷积层和上采样操作  
 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) # 返回上采样后的结果  
  
  
# 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  
  
 # 定义用于计算权重的卷积层  
 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) # 计算最终权重  
 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)  
  
 # 将权重拼接并计算最终权重  
 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  
  
  
# 定义主网络结构  
class AFPN\_P345(nn.Module):  
 def \_\_init\_\_(self, in\_channels=[256, 512, 1024], out\_channels=256, factor=4):  
 super(AFPN\_P345, self).\_\_init\_\_()  
  
 # 定义输入通道到输出通道的卷积层  
 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)  
  
 # 定义特征提取的主体部分  
 self.body = BlockBody\_P345([in\_channels[0] // factor, in\_channels[1] // factor, in\_channels[2] // factor])  
  
 # 定义输出通道的卷积层  
 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 # 输入特征图  
  
 # 通过卷积层处理输入  
 x0 = self.conv0(x0)  
 x1 = self.conv1(x1)  
 x2 = self.conv2(x2)  
  
 # 通过主体部分进行特征提取  
 out0, out1, out2 = self.body([x0, x1, x2])  
  
 # 通过输出卷积层  
 out0 = self.conv00(out0)  
 out1 = self.conv11(out1)  
 out2 = self.conv22(out2)  
 return [out0, out1, out2] # 返回输出特征图  
```  
  
### 代码核心部分解释：  
1. \*\*BasicBlock\*\*: 实现了一个基本的卷积块，包含两个卷积层和残差连接。  
2. \*\*Upsample\*\*: 实现了上采样操作，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。  
4. \*\*AFPN\_P345\*\*: 主网络结构，负责输入特征的处理和输出特征的生成。通过卷积层和自适应特征融合模块来实现特征提取和融合。  
  
这些部分是构建更复杂网络的基础，提供了卷积、上采样和特征融合的基本功能。```

这个程序文件 `afpn.py` 实现了一个自适应特征金字塔网络（AFPN），主要用于计算机视觉任务中的特征提取和融合。程序中定义了多个类，主要包括基本模块、上采样和下采样模块，以及自适应特征融合模块。以下是对代码的详细说明。  
  
首先，程序导入了一些必要的库，包括 `torch` 和 `torch.nn`，并引入了一些自定义的卷积和模块。接着，定义了一个 `BasicBlock` 类，它是一个基本的残差块，包含两个卷积层和一个残差连接。这个块在特征提取中起到重要作用。  
  
接下来，定义了多个上采样和下采样类（如 `Upsample`、`Downsample\_x2`、`Downsample\_x4`、`Downsample\_x8`），这些类通过卷积和插值操作实现特征图的尺寸变化，以便在不同尺度的特征之间进行融合。  
  
自适应特征融合模块（ASFF）是该网络的核心部分，分别定义了 `ASFF\_2`、`ASFF\_3` 和 `ASFF\_4` 类。这些模块通过计算输入特征图的权重，来实现不同尺度特征的加权融合。每个 ASFF 模块根据输入特征图的数量不同而有所区别，使用了 softmax 函数来归一化权重，从而实现特征的自适应融合。  
  
然后，定义了 `BlockBody\_P345` 和 `BlockBody\_P2345` 类，这些类实现了网络的主体结构，包含多个卷积块和自适应特征融合模块。它们通过对不同尺度的特征进行处理和融合，构建了一个多尺度特征提取网络。  
  
`AFPN\_P345` 和 `AFPN\_P2345` 类是整个网络的顶层结构，负责输入特征的处理和输出特征的生成。它们通过调用之前定义的模块，完成特征的下采样、上采样和融合，并最终输出不同尺度的特征图。  
  
此外，程序还提供了 `BlockBody\_P345\_Custom` 和 `BlockBody\_P2345\_Custom` 类，允许用户自定义使用的基本块类型，以便在不同的应用场景中灵活调整网络结构。  
  
最后，整个程序通过定义多个类和模块，构建了一个灵活且强大的自适应特征金字塔网络，能够有效地处理多尺度特征，适用于各种计算机视觉任务。

### 整体功能和构架概括  
  
该项目包含多个模块和文件，主要用于构建和训练深度学习模型，特别是在计算机视觉任务中的应用。每个文件实现了特定的功能，形成了一个完整的深度学习框架。以下是各个文件的功能概述：  
  
- \*\*predict.py\*\*: 实现目标检测的预测模块，负责处理模型的预测任务和后处理。  
- \*\*starnet.py\*\*: 定义了StarNet模型及其组件，专注于高效的特征提取和处理。  
- \*\*EfficientFormerV2.py\*\*: 实现了EfficientFormerV2模型，结合了注意力机制和多层感知机，适用于图像处理任务。  
- \*\*test\_selective\_scan\_easy.py\*\*: 实现选择性扫描操作，主要用于处理序列数据，包含自定义的前向和反向传播操作。  
- \*\*SMPConv.py\*\*: 实现了一种新的卷积模块SMPConv，结合了深度可分离卷积，旨在提高模型的计算效率和表达能力。  
- \*\*afpn.py\*\*: 实现自适应特征金字塔网络（AFPN），用于多尺度特征提取和融合。  
- \*\*attention.py\*\*: 实现注意力机制的相关模块，增强模型在特征提取过程中的表现。  
  
### 文件功能整理表  
  
| 文件名 | 功能描述 |  
|-------------------------------|----------------------------------------------------------|  
| `predict.py` | 实现目标检测的预测模块，处理模型的预测任务和后处理。 |  
| `starnet.py` | 定义StarNet模型及其组件，专注于高效的特征提取和处理。 |  
| `EfficientFormerV2.py` | 实现EfficientFormerV2模型，结合注意力机制和多层感知机。 |  
| `test\_selective\_scan\_easy.py` | 实现选择性扫描操作，处理序列数据，包含自定义的前向和反向传播。 |  
| `SMPConv.py` | 实现新的卷积模块SMPConv，结合深度可分离卷积，提高计算效率。 |  
| `afpn.py` | 实现自适应特征金字塔网络（AFPN），用于多尺度特征提取和融合。 |  
| `attention.py` | 实现注意力机制的相关模块，增强特征提取过程中的表现。 |  
  
这个表格总结了每个文件的主要功能，帮助理解整个项目的结构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。