# 改进yolo11-DLKA等200+全套创新点大全：路面坑洼检测系统源码＆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，路面交通设施的维护与管理变得愈发重要。路面坑洼不仅影响行车安全，还可能导致交通事故的发生，给社会带来巨大的经济损失。因此，及时、准确地检测和修复路面坑洼成为了交通管理部门亟待解决的问题。传统的人工巡查方法不仅效率低下，而且容易受到主观因素的影响，难以保证检测的准确性和全面性。近年来，计算机视觉技术的迅猛发展为路面坑洼检测提供了新的解决方案。  
  
在众多计算机视觉算法中，YOLO（You Only Look Once）系列因其高效的实时检测能力而受到广泛关注。YOLOv11作为该系列的最新版本，结合了深度学习的优势，能够在复杂的环境中快速识别和定位目标。然而，现有的YOLOv11模型在特定场景下的表现仍有提升空间，尤其是在针对路面坑洼等小目标的检测上。因此，基于改进YOLOv11的路面坑洼检测系统的研究具有重要的现实意义。  
  
本研究将利用“Detection5”数据集，该数据集包含818幅图像，涵盖了多种交通相关类别，包括“car”、“clack”、“no\_stop”、“pothole”和“speed”。通过对这些图像的深入分析与处理，改进YOLOv11模型的特征提取和目标检测能力，旨在提高路面坑洼的检测精度和速度。此外，本研究还将探索如何将改进后的模型应用于实际的路面监测系统中，为城市交通管理提供有效的技术支持，最终实现智能化的路面维护与管理。通过这一研究，不仅能够提升路面坑洼检测的效率和准确性，还将为未来的智能交通系统建设奠定基础，推动交通管理的数字化转型。

## 4. 数据集信息展示

本项目数据集信息介绍  
  
本项目所使用的数据集名为“Detection5”，其主要目的是为了训练和改进YOLOv11模型，以实现高效的路面坑洼检测系统。该数据集包含五个类别，分别为“car”（汽车）、“clack”（车轮碰撞声）、“no\_stop”（不停车标志）、“pothole”（坑洼）和“speed”（速度限制标志）。这些类别的选择旨在全面覆盖与路面状况相关的关键元素，从而提高模型在实际应用中的准确性和鲁棒性。  
  
在数据集的构建过程中，采用了多种数据采集技术，包括实地拍摄和合成数据生成，以确保数据的多样性和代表性。每个类别的样本均经过精心标注，确保模型能够准确识别不同的路面状况和交通标志。例如，汽车类别的样本涵盖了不同品牌、颜色和形状的车辆，以模拟真实世界中的复杂交通环境；而坑洼类别则专注于不同类型和大小的路面缺陷，以提高模型对各种坑洼的检测能力。  
  
此外，数据集还考虑了不同天气和光照条件下的样本，以增强模型的适应性。通过引入多样化的场景，模型能够在各种环境中保持良好的性能，减少因环境变化导致的误检和漏检现象。这一数据集的构建不仅为YOLOv11模型的训练提供了坚实的基础，也为后续的路面监测和维护工作提供了重要的数据支持。通过不断优化和扩展“Detection5”数据集，我们期望能够推动智能交通系统的发展，提高城市道路的安全性和通行效率。

## 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

```以下是经过简化和注释的核心代码部分，保留了最重要的功能和结构。  
  
```python  
import torch  
import torch.nn as nn  
import torch.nn.functional as F  
  
# 定义一些常用的函数  
def transI\_fusebn(kernel, bn):  
 """  
 将卷积核和批归一化层的参数融合  
 :param kernel: 卷积核  
 :param bn: 批归一化层  
 :return: 融合后的卷积核和偏置  
 """  
 gamma = bn.weight  
 std = (bn.running\_var + bn.eps).sqrt()  
 return kernel \* ((gamma / std).reshape(-1, 1, 1, 1)), bn.bias - bn.running\_mean \* gamma / std  
  
def conv\_bn(in\_channels, out\_channels, kernel\_size, stride=1, padding=0, dilation=1, groups=1):  
 """  
 创建一个卷积层和批归一化层的组合  
 :param in\_channels: 输入通道数  
 :param out\_channels: 输出通道数  
 :param kernel\_size: 卷积核大小  
 :param stride: 步幅  
 :param padding: 填充  
 :param dilation: 膨胀  
 :param groups: 分组卷积  
 :return: 包含卷积和批归一化的顺序容器  
 """  
 conv\_layer = nn.Conv2d(in\_channels, out\_channels, kernel\_size, stride=stride, padding=padding, dilation=dilation, groups=groups, bias=False)  
 bn\_layer = nn.BatchNorm2d(out\_channels)  
 return nn.Sequential(conv\_layer, bn\_layer)  
  
class DiverseBranchBlock(nn.Module):  
 def \_\_init\_\_(self, in\_channels, out\_channels, kernel\_size, stride=1, padding=None, dilation=1, groups=1):  
 """  
 多分支卷积块的构造函数  
 :param in\_channels: 输入通道数  
 :param out\_channels: 输出通道数  
 :param kernel\_size: 卷积核大小  
 :param stride: 步幅  
 :param padding: 填充  
 :param dilation: 膨胀  
 :param groups: 分组卷积  
 """  
 super(DiverseBranchBlock, self).\_\_init\_\_()  
   
 # 默认填充  
 if padding is None:  
 padding = kernel\_size // 2  
   
 # 原始卷积和批归一化  
 self.dbb\_origin = conv\_bn(in\_channels, out\_channels, kernel\_size, stride, padding, dilation, groups)  
   
 # 平均池化分支  
 self.dbb\_avg = nn.Sequential(  
 nn.Conv2d(in\_channels, out\_channels, kernel\_size=1, stride=1, padding=0, groups=groups, bias=False),  
 nn.BatchNorm2d(out\_channels),  
 nn.AvgPool2d(kernel\_size=kernel\_size, stride=stride, padding=0)  
 )  
  
 # 1x1卷积分支  
 self.dbb\_1x1\_kxk = nn.Sequential(  
 nn.Conv2d(in\_channels, out\_channels, kernel\_size=1, stride=1, padding=0, groups=groups, bias=False),  
 nn.BatchNorm2d(out\_channels)  
 )  
  
 def forward(self, inputs):  
 """  
 前向传播  
 :param inputs: 输入张量  
 :return: 输出张量  
 """  
 out = self.dbb\_origin(inputs) # 原始卷积输出  
 out += self.dbb\_avg(inputs) # 加上平均池化分支输出  
 out += self.dbb\_1x1\_kxk(inputs) # 加上1x1卷积分支输出  
 return out # 返回最终输出  
  
# 示例使用  
if \_\_name\_\_ == "\_\_main\_\_":  
 model = DiverseBranchBlock(in\_channels=3, out\_channels=16, kernel\_size=3)  
 x = torch.randn(1, 3, 32, 32) # 输入张量  
 output = model(x) # 前向传播  
 print(output.shape) # 输出形状  
```  
  
### 代码说明：  
1. \*\*transI\_fusebn\*\*: 该函数用于将卷积层的权重与批归一化层的参数融合，以便在推理时减少计算量。  
2. \*\*conv\_bn\*\*: 创建一个组合层，包括卷积层和批归一化层，便于后续使用。  
3. \*\*DiverseBranchBlock\*\*: 这是一个多分支卷积块，包含多个卷积分支和一个平均池化分支，适用于多种输入特征的提取。  
4. \*\*forward\*\*: 定义了前向传播过程，计算各个分支的输出并相加。  
  
### 主要功能：  
- 该模块可以用于构建复杂的卷积神经网络，支持多种输入特征的处理，适合于图像分类、目标检测等任务。```

该文件 `rep\_block.py` 定义了一系列用于深度学习的模块，主要包括多种不同的卷积块，旨在实现多样化的特征提取。文件中使用了 PyTorch 框架，并且包含了一些自定义的卷积和批归一化操作。  
  
首先，文件导入了必要的库，包括 `torch` 和 `torch.nn`，并定义了一些用于卷积操作的辅助函数。这些函数主要用于处理卷积核和偏置的转换、融合以及多尺度处理等。  
  
接下来，定义了多个类，主要包括 `DiverseBranchBlock`、`WideDiverseBranchBlock` 和 `DeepDiverseBranchBlock`。这些类的构造函数中，用户可以指定输入和输出通道数、卷积核大小、步幅、填充方式等参数。  
  
`DiverseBranchBlock` 类是一个多分支卷积块，包含多个不同的卷积路径，允许模型在不同的特征空间中进行学习。它通过不同的卷积操作（如 1x1、3x3 卷积）和池化操作来实现多样化的特征提取。在前向传播中，它会将不同路径的输出相加，经过非线性激活函数后返回。  
  
`WideDiverseBranchBlock` 类则扩展了这一概念，增加了水平和垂直卷积的处理，能够在不同方向上提取特征。它使用了两个额外的卷积层，分别处理水平和垂直方向的特征，最终将这些特征与其他分支的输出结合。  
  
`DeepDiverseBranchBlock` 类则进一步增加了深度，结合了多个卷积块的特征。它的设计目的是在更深的网络中保持多样化的特征提取能力。  
  
此外，文件中还定义了一些辅助类，如 `IdentityBasedConv1x1` 和 `BNAndPadLayer`，前者实现了带有身份映射的 1x1 卷积，后者则结合了批归一化和填充操作。这些类的设计使得网络在进行卷积操作时能够保持更好的特征传递和信息流动。  
  
总的来说，`rep\_block.py` 文件中的模块设计旨在提高卷积神经网络的灵活性和表达能力，通过多样化的卷积结构和分支设计，使得模型能够在复杂的任务中表现得更加出色。

``````python  
import sys  
import subprocess  
  
def run\_script(script\_path):  
 """  
 使用当前 Python 环境运行指定的脚本。  
  
 Args:  
 script\_path (str): 要运行的脚本路径  
  
 Returns:  
 None  
 """  
 # 获取当前 Python 解释器的路径  
 python\_path = sys.executable  
  
 # 构建运行命令  
 command = f'"{python\_path}" -m streamlit run "{script\_path}"'  
  
 # 执行命令  
 result = subprocess.run(command, shell=True)  
 if result.returncode != 0:  
 print("脚本运行出错。")  
  
  
# 实例化并运行应用  
if \_\_name\_\_ == "\_\_main\_\_":  
 # 指定您的脚本路径  
 script\_path = "web.py" # 这里可以直接指定脚本名称，假设它在当前目录下  
  
 # 运行脚本  
 run\_script(script\_path)  
```  
  
### 代码核心部分及注释  
  
1. \*\*导入必要的模块\*\*：  
 - `sys`：用于访问与 Python 解释器紧密相关的变量和函数。  
 - `subprocess`：用于执行外部命令。  
  
2. \*\*定义 `run\_script` 函数\*\*：  
 - 该函数接受一个参数 `script\_path`，表示要运行的 Python 脚本的路径。  
 - 使用 `sys.executable` 获取当前 Python 解释器的路径，以确保使用正确的 Python 环境来运行脚本。  
 - 构建命令字符串，使用 `streamlit` 模块运行指定的脚本。  
 - 使用 `subprocess.run` 执行构建的命令，并检查返回码以确定脚本是否成功运行。  
  
3. \*\*主程序块\*\*：  
 - 通过 `if \_\_name\_\_ == "\_\_main\_\_":` 确保只有在直接运行该脚本时才会执行以下代码。  
 - 指定要运行的脚本路径 `script\_path`，这里假设脚本文件名为 `web.py`。  
 - 调用 `run\_script` 函数来执行指定的脚本。```

这个程序文件的主要功能是使用当前的 Python 环境来运行一个指定的脚本，具体是通过 Streamlit 框架来启动一个 Web 应用。首先，程序导入了必要的模块，包括 `sys`、`os` 和 `subprocess`，以及一个自定义的 `abs\_path` 函数，用于获取文件的绝对路径。  
  
在 `run\_script` 函数中，首先获取当前 Python 解释器的路径，这样可以确保使用的是正确的 Python 环境。接着，构建一个命令字符串，该命令会调用 Streamlit 来运行指定的脚本。这里使用了 `-m streamlit run` 的方式来执行脚本，这样可以确保 Streamlit 正确加载。  
  
然后，使用 `subprocess.run` 方法来执行构建好的命令。这个方法会在新的 shell 中运行命令，并等待其完成。如果脚本运行过程中出现错误（即返回码不为 0），程序会打印出“脚本运行出错”的提示信息。  
  
在文件的最后部分，使用 `if \_\_name\_\_ == "\_\_main\_\_":` 语句来确保只有在直接运行该文件时才会执行后面的代码。这里指定了要运行的脚本路径，即 `web.py`，并调用 `run\_script` 函数来启动这个脚本。  
  
总体来说，这个程序的目的是简化通过 Streamlit 启动 Web 应用的过程，确保在正确的 Python 环境中运行指定的脚本。

```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  
import torch.nn.functional as F  
  
class DyReLU(nn.Module):  
 """动态ReLU模块，支持自适应激活函数的调整。"""  
 def \_\_init\_\_(self, inp, reduction=4, lambda\_a=1.0, use\_bias=True):  
 super(DyReLU, self).\_\_init\_\_()  
 self.oup = inp # 输出通道数  
 self.lambda\_a = lambda\_a \* 2 # 调整因子  
 self.avg\_pool = nn.AdaptiveAvgPool2d(1) # 自适应平均池化层  
  
 # 确定压缩比  
 squeeze = inp // reduction  
 self.fc = nn.Sequential(  
 nn.Linear(inp, squeeze), # 全连接层  
 nn.ReLU(inplace=True), # ReLU激活  
 nn.Linear(squeeze, self.oup \* 2), # 输出两个参数  
 h\_sigmoid() # 使用h\_sigmoid激活  
 )  
  
 def forward(self, x):  
 """前向传播函数。"""  
 b, c, h, w = x.size() # 获取输入的批量大小、通道数、高度和宽度  
 y = self.avg\_pool(x).view(b, c) # 进行平均池化并调整形状  
 y = self.fc(y).view(b, self.oup \* 2, 1, 1) # 通过全连接层  
  
 # 分割y为两个参数a和b  
 a, b = torch.split(y, self.oup, dim=1)  
 a = (a - 0.5) \* self.lambda\_a + 1.0 # 调整a的值  
 out = x \* a + b # 计算输出  
  
 return out  
  
class DyDCNv2(nn.Module):  
 """带有归一化层的可调变形卷积模块。"""  
 def \_\_init\_\_(self, in\_channels, out\_channels, stride=1, norm\_cfg=None):  
 super().\_\_init\_\_()  
 self.conv = ModulatedDeformConv2d(  
 in\_channels, out\_channels, 3, stride=stride, padding=1, bias=norm\_cfg is None)  
 if norm\_cfg:  
 self.norm = build\_norm\_layer(norm\_cfg, out\_channels)[1] # 构建归一化层  
  
 def forward(self, x, offset, mask):  
 """前向传播函数。"""  
 x = self.conv(x.contiguous(), offset, mask) # 进行可调变形卷积  
 if hasattr(self, 'norm'):  
 x = self.norm(x) # 如果有归一化层，则进行归一化  
 return x  
  
class DyHeadBlock\_Prune(nn.Module):  
 """DyHead模块，结合多种注意力机制。"""  
 def \_\_init\_\_(self, in\_channels, norm\_type='GN'):  
 super().\_\_init\_\_()  
 self.spatial\_conv\_high = DyDCNv2(in\_channels, in\_channels) # 高层卷积  
 self.spatial\_conv\_mid = DyDCNv2(in\_channels, in\_channels) # 中层卷积  
 self.spatial\_conv\_low = DyDCNv2(in\_channels, in\_channels, stride=2) # 低层卷积  
 self.spatial\_conv\_offset = nn.Conv2d(in\_channels, 27, 3, padding=1) # 偏移和掩码卷积  
  
 def forward(self, x, level):  
 """前向传播函数。"""  
 # 计算偏移和掩码  
 offset\_and\_mask = self.spatial\_conv\_offset(x[level])  
 offset = offset\_and\_mask[:, :18, :, :] # 提取偏移  
 mask = offset\_and\_mask[:, 18:, :, :].sigmoid() # 提取掩码并应用sigmoid  
  
 mid\_feat = self.spatial\_conv\_mid(x[level], offset, mask) # 中层特征  
 sum\_feat = mid\_feat # 初始化总特征  
  
 # 结合低层和高层特征  
 if level > 0:  
 low\_feat = self.spatial\_conv\_low(x[level - 1], offset, mask) # 低层特征  
 sum\_feat += low\_feat  
 if level < len(x) - 1:  
 high\_feat = F.interpolate(self.spatial\_conv\_high(x[level + 1], offset, mask),  
 size=x[level].shape[-2:], mode='bilinear', align\_corners=True) # 高层特征  
 sum\_feat += high\_feat  
  
 return sum\_feat # 返回总特征  
```  
  
### 代码注释说明：  
1. \*\*DyReLU\*\*: 这是一个动态ReLU模块，能够根据输入的特征自适应地调整激活函数的参数。它通过全连接层学习两个参数，并结合输入特征进行输出。  
   
2. \*\*DyDCNv2\*\*: 这是一个带有归一化层的可调变形卷积模块，能够在卷积过程中应用偏移和掩码，以增强特征提取能力。  
  
3. \*\*DyHeadBlock\_Prune\*\*: 这是一个结合多种注意力机制的模块，通过不同层次的卷积操作来融合特征。它计算偏移和掩码，并将中层、低层和高层特征结合起来，以便于后续的处理。  
  
以上是代码的核心部分和详细注释，旨在帮助理解每个模块的功能和工作原理。```

这个程序文件 `dyhead\_prune.py` 是一个用于实现动态头部（Dynamic Head）模块的 PyTorch 代码，主要用于计算机视觉任务中的特征提取和注意力机制。代码中定义了多个类和函数，下面是对其主要内容的讲解。  
  
首先，文件导入了必要的 PyTorch 库和一些外部库，如 `mmcv` 和 `mmengine`，这些库提供了构建激活层、归一化层和可调变形卷积等功能。接着，定义了一个 `\_make\_divisible` 函数，用于确保某个值是可被指定的除数整除的，同时还会考虑最小值的限制。  
  
接下来，定义了几个激活函数的类，包括 `swish`、`h\_swish` 和 `h\_sigmoid`。这些类都是继承自 `nn.Module`，并实现了 `forward` 方法，以便在前向传播中使用。  
  
`DyReLU` 类是一个动态激活函数模块，它通过自适应地调整输入的激活值来增强模型的表达能力。该类的构造函数接受多个参数，包括输入通道数、缩减比例、初始化参数等。它通过全连接层和可选的空间注意力机制来实现动态调整。`forward` 方法根据输入的特征图计算输出，支持多种不同的计算方式。  
  
`DyDCNv2` 类是一个封装了可调变形卷积（Modulated Deformable Convolution）的模块，支持归一化层的使用。它的构造函数接受输入和输出通道数、步幅以及归一化配置。`forward` 方法执行卷积操作，并在需要时应用归一化。  
  
`DyHeadBlock\_Prune` 类是动态头部块的实现，包含三种类型的注意力机制。构造函数中定义了多个卷积层和注意力模块，并初始化权重。`forward` 方法计算特征图的偏移量和掩码，并通过不同层次的特征进行融合，最终输出经过任务注意力模块处理的特征图。  
  
整个文件的设计旨在通过动态调整激活函数和特征融合来提升模型的性能，适用于需要多层次特征提取和注意力机制的深度学习任务。

```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  
from timm.layers import weight\_init  
  
# 定义激活函数类  
class activation(nn.ReLU):  
 def \_\_init\_\_(self, dim, act\_num=3, deploy=False):  
 super(activation, self).\_\_init\_\_()  
 self.deploy = deploy # 是否在部署模式  
 # 初始化权重参数  
 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:  
 return torch.nn.functional.conv2d(  
 super(activation, self).forward(x),   
 self.weight, self.bias, padding=(self.act\_num\*2 + 1)//2, groups=self.dim)  
 else:  
 return self.bn(torch.nn.functional.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  
  
# 定义基本模块Block  
class Block(nn.Module):  
 def \_\_init\_\_(self, dim, dim\_out, act\_num=3, stride=2, deploy=False):  
 super().\_\_init\_\_()  
 self.deploy = deploy  
 # 根据是否部署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)  
 )  
 # 池化层的选择  
 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  
  
# 定义主网络结构VanillaNet  
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) # 添加每个Block到网络中  
  
 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Block  
 return x  
  
# 创建模型的函数  
def vanillanet\_10(pretrained='', \*\*kwargs):  
 model = VanillaNet(dims=[128\*4, 128\*4, 256\*4, 512\*4, 512\*4, 512\*4, 512\*4, 1024\*4, 1024\*4], \*\*kwargs)  
 if pretrained:  
 weights = torch.load(pretrained)['model\_ema']  
 model.load\_state\_dict(weights) # 加载预训练权重  
 return model  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 inputs = torch.randn((1, 3, 640, 640)) # 输入张量  
 model = vanillanet\_10() # 创建模型  
 pred = model(inputs) # 进行前向传播  
 for i in pred:  
 print(i.size()) # 输出每层的尺寸  
```  
  
### 代码说明：  
1. \*\*激活函数类 (`activation`)\*\*：自定义的激活函数类，支持批归一化和卷积操作。  
2. \*\*基本模块 (`Block`)\*\*：网络的基本构建块，包含卷积、池化和激活函数的组合。  
3. \*\*主网络结构 (`VanillaNet`)\*\*：整体网络结构，包含输入层（stem）和多个基本模块（Block）。  
4. \*\*模型创建函数\*\*：提供了一个简单的接口来创建和加载预训练模型。  
5. \*\*主程序\*\*：用于测试模型的输入和输出尺寸。```

这个程序文件定义了一个名为 `VanillaNet` 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，并且包含了一些用于构建和训练神经网络的基本组件。  
  
首先，文件开头包含了一些版权声明和许可证信息，表明该程序是开源的，可以在 MIT 许可证下进行修改和分发。  
  
接下来，程序导入了必要的库，包括 PyTorch 的核心模块和一些辅助功能模块，如 `weight\_init` 和 `DropPath`。这些模块提供了权重初始化和其他功能，帮助构建神经网络。  
  
在代码中，`activation` 类是一个自定义的激活函数类，继承自 `nn.ReLU`。它的构造函数中定义了权重和偏置，并使用批量归一化（Batch Normalization）来提高模型的稳定性和收敛速度。`forward` 方法实现了前向传播的逻辑，支持两种模式：部署模式和训练模式。在部署模式下，使用卷积操作直接处理输入，而在训练模式下，则包括了批量归一化的步骤。  
  
`Block` 类是模型的基本构建块，包含了卷积层、池化层和激活函数。它的构造函数允许设置输入和输出的维度、步幅以及是否使用自适应池化。`forward` 方法实现了块的前向传播逻辑，并根据步幅选择合适的池化操作。  
  
`VanillaNet` 类是整个网络的主类，包含了多个 `Block` 的组合。它的构造函数允许用户指定输入通道数、类别数、各层的维度、丢弃率、激活函数数量、步幅等参数。模型的前向传播逻辑在 `forward` 方法中实现，输入经过多个阶段的处理后，返回特征图。  
  
此外，文件中还定义了一些函数，如 `update\_weight` 用于更新模型的权重，和多个 `vanillanet\_x` 函数用于创建不同配置的 `VanillaNet` 模型。这些函数允许用户加载预训练的权重，从而加速模型的训练过程。  
  
最后，在 `\_\_main\_\_` 部分，创建了一个输入张量并实例化了 `vanillanet\_10` 模型，随后进行了前向传播并打印了输出特征图的尺寸。这部分代码用于测试模型的基本功能。  
  
整体来看，这个程序文件展示了如何使用 PyTorch 构建一个灵活且可扩展的卷积神经网络，适用于多种图像处理任务。

### 整体功能和构架概括  
  
该项目包含多个 Python 文件，每个文件实现了深度学习模型的不同部分，主要用于计算机视觉任务。整体架构包括特征提取模块、动态注意力机制、模型构建以及用户界面等功能。具体来说：  
  
- \*\*`rep\_block.py`\*\*：实现了多种卷积块，支持多分支特征提取，旨在增强模型的表达能力。  
- \*\*`ui.py`\*\*：负责启动一个基于 Streamlit 的 Web 应用，简化了用户与模型交互的过程。  
- \*\*`dyhead\_prune.py`\*\*：实现了动态头部模块，结合动态激活函数和可调变形卷积，增强了模型的特征提取能力。  
- \*\*`VanillaNet.py`\*\*：定义了一个基础的卷积神经网络结构，提供了灵活的构建方式和多种配置选项。  
  
### 文件功能整理表  
  
| 文件名 | 功能描述 |  
|----------------------|-----------------------------------------------------------------------------------------------|  
| `rep\_block.py` | 定义多种卷积块（如多分支卷积块、宽分支卷积块、深分支卷积块），用于增强特征提取能力。 |  
| `ui.py` | 启动基于 Streamlit 的 Web 应用，简化用户与模型的交互过程。 |  
| `dyhead\_prune.py` | 实现动态头部模块，结合动态激活函数和可调变形卷积，提升特征提取能力和模型性能。 |  
| `VanillaNet.py` | 定义基础卷积神经网络结构，提供灵活的构建方式和多种配置选项，适用于图像处理任务。 |  
  
这些文件共同构成了一个完整的深度学习框架，能够处理复杂的计算机视觉任务，并提供用户友好的界面进行交互和可视化。

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。