# 改进yolo11-DCNV2-Dynamic等200+全套创新点大全：柠檬质量检测系统源码＆数据集全套

## 3.背景

研究背景与意义  
  
随着全球经济的发展和人们生活水平的提高，食品安全和质量问题日益受到重视。柠檬作为一种广泛消费的水果，其质量直接影响到消费者的健康和市场的经济效益。因此，建立一个高效、准确的柠檬质量检测系统具有重要的现实意义。传统的柠檬质量检测方法多依赖人工检验，不仅耗时耗力，而且容易受到主观因素的影响，导致检测结果的不一致性。随着计算机视觉技术的快速发展，基于深度学习的自动化检测方法逐渐成为研究的热点。  
  
本研究旨在基于改进的YOLOv11模型，构建一个高效的柠檬质量检测系统。YOLO（You Only Look Once）系列模型因其高效的实时检测能力和良好的准确性而广泛应用于物体检测领域。通过对YOLOv11的改进，结合针对柠檬质量的特定需求，我们可以实现对柠檬的快速、准确分类，从而提高生产效率，降低人工成本。  
  
本项目所使用的数据集包含2100张柠檬图像，分为“好”和“坏”两类，能够有效支持模型的训练与评估。通过对这些图像的实例分割处理，系统能够识别柠檬的质量特征，进而实现对不同质量柠檬的自动分类。这不仅有助于提升柠檬的市场竞争力，也为相关企业提供了数据支持，促进其生产流程的优化。  
  
综上所述，基于改进YOLOv11的柠檬质量检测系统的研究，不仅能够提升柠檬的质量控制水平，还将推动计算机视觉技术在农业领域的应用，为实现智能农业提供新的思路和方法。

## 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  
  
本项目旨在开发一个改进版的YOLOv11模型，以实现柠檬质量检测系统的高效运作。为此，我们构建了一个专门针对柠檬质量评估的数据集，命名为“Lemon\_quality”。该数据集包含两类柠檬样本，分别为“bad”（劣质）和“good”（优质），总类别数量为2。这一分类不仅为模型提供了清晰的目标，也为后续的质量检测算法提供了坚实的基础。  
  
在数据集的构建过程中，我们精心挑选了来自不同环境和生长条件下的柠檬样本，以确保数据的多样性和代表性。每个类别的样本均经过严格筛选，确保“bad”类别的柠檬具有明显的缺陷特征，如表面斑点、变色、腐烂等，而“good”类别的柠檬则展现出鲜亮的颜色、光滑的表面和良好的形态。这种清晰的标注方式不仅有助于模型的训练，还能提高模型在实际应用中的准确性和可靠性。  
  
数据集中的图像数量经过精心设计，以确保每个类别的样本均衡，避免模型在训练过程中出现偏差。此外，为了增强模型的泛化能力，我们还对图像进行了多种数据增强处理，包括旋转、缩放、翻转等操作，从而丰富了训练样本的多样性。通过这种方式，我们希望模型能够更好地适应不同环境下的柠檬质量检测任务。  
  
综上所述，“Lemon\_quality”数据集为改进YOLOv11模型提供了坚实的基础，旨在通过高效的质量检测手段，提升柠檬的市场竞争力，促进农业生产的可持续发展。

## 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

```以下是经过精简和详细注释的核心代码部分：  
  
```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，使用 streamlit 运行指定的脚本  
 command = f'"{python\_path}" -m streamlit run "{script\_path}"'  
  
 # 执行命令，并等待其完成  
 result = subprocess.run(command, shell=True)  
   
 # 检查命令执行结果，如果返回码不为0，则表示出错  
 if result.returncode != 0:  
 print("脚本运行出错。")  
  
# 程序入口  
if \_\_name\_\_ == "\_\_main\_\_":  
 # 指定要运行的脚本路径  
 script\_path = "web.py" # 假设脚本在当前目录下  
  
 # 调用函数运行脚本  
 run\_script(script\_path)  
```  
  
### 代码注释说明：  
  
1. \*\*导入模块\*\*：  
 - `sys`：用于获取当前 Python 解释器的路径。  
 - `subprocess`：用于执行外部命令。  
  
2. \*\*`run\_script` 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。  
 - 构建一个命令字符串，使用 `streamlit` 模块运行指定的脚本。  
 - 使用 `subprocess.run` 执行命令，并等待其完成。  
 - 检查命令的返回码，如果不为0，打印错误信息。  
  
3. \*\*程序入口\*\*：  
 - 使用 `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为 `web.py`（假设该脚本在当前目录下）。  
 - 调用 `run\_script` 函数，传入脚本路径以运行该脚本。```

这个程序文件 `ui.py` 的主要功能是运行一个指定的 Python 脚本，具体来说是通过 Streamlit 框架来启动一个 Web 应用。程序首先导入了必要的模块，包括 `sys`、`os` 和 `subprocess`，这些模块分别用于获取系统信息、处理文件路径和执行外部命令。  
  
在 `run\_script` 函数中，首先获取当前 Python 解释器的路径，这样可以确保使用正确的 Python 环境来运行脚本。接着，构建一个命令字符串，这个命令会调用 Streamlit 来运行指定的脚本。命令的格式是 `python -m streamlit run script\_path`，其中 `script\_path` 是传入的参数。  
  
然后，使用 `subprocess.run` 方法执行这个命令。该方法会在新的进程中运行命令，并等待其完成。如果命令执行后返回的状态码不为零，表示脚本运行过程中出现了错误，此时会打印出“脚本运行出错”的提示信息。  
  
在文件的最后部分，使用 `if \_\_name\_\_ == "\_\_main\_\_":` 语句来确保当这个文件作为主程序运行时才会执行后面的代码。这里指定了要运行的脚本路径为 `web.py`，并调用 `run\_script` 函数来启动这个脚本。  
  
总体来说，这个程序的设计简单明了，主要用于在当前 Python 环境中运行一个 Streamlit Web 应用，方便用户进行开发和测试。

```以下是经过简化并注释的核心代码部分，主要保留了 MobileNetV4 的结构和关键组件。  
  
```python  
import torch  
import torch.nn as nn  
  
# 定义模型的块规格，包括不同版本的 MobileNetV4  
MODEL\_SPECS = {  
 "MobileNetV4ConvSmall": {  
 "conv0": {"block\_name": "convbn", "num\_blocks": 1, "block\_specs": [[3, 32, 3, 2]]},  
 "layer1": {"block\_name": "convbn", "num\_blocks": 2, "block\_specs": [[32, 32, 3, 2], [32, 32, 1, 1]]},  
 "layer2": {"block\_name": "convbn", "num\_blocks": 2, "block\_specs": [[32, 96, 3, 2], [96, 64, 1, 1]]},  
 "layer3": {"block\_name": "uib", "num\_blocks": 6, "block\_specs": [[64, 96, 5, 5, True, 2, 3], [96, 96, 0, 3, True, 1, 2]]},  
 "layer4": {"block\_name": "uib", "num\_blocks": 6, "block\_specs": [[96, 128, 3, 3, True, 2, 6], [128, 128, 5, 5, True, 1, 4]]},  
 "layer5": {"block\_name": "convbn", "num\_blocks": 2, "block\_specs": [[128, 960, 1, 1], [960, 1280, 1, 1]]}  
 },  
 # 其他模型规格可以在这里添加  
}  
  
def conv\_2d(inp, oup, kernel\_size=3, stride=1, groups=1, bias=False, norm=True, act=True):  
 """  
 创建一个2D卷积层，包含可选的批归一化和激活函数。  
   
 Args:  
 inp: 输入通道数  
 oup: 输出通道数  
 kernel\_size: 卷积核大小  
 stride: 步幅  
 groups: 分组卷积  
 bias: 是否使用偏置  
 norm: 是否使用批归一化  
 act: 是否使用激活函数  
   
 Returns:  
 nn.Sequential: 包含卷积层、批归一化和激活函数的序列  
 """  
 conv = nn.Sequential()  
 padding = (kernel\_size - 1) // 2 # 计算填充  
 conv.add\_module('conv', nn.Conv2d(inp, oup, kernel\_size, stride, padding, bias=bias, groups=groups))  
 if norm:  
 conv.add\_module('BatchNorm2d', nn.BatchNorm2d(oup)) # 添加批归一化  
 if act:  
 conv.add\_module('Activation', nn.ReLU6()) # 添加ReLU6激活函数  
 return conv  
  
class InvertedResidual(nn.Module):  
 """  
 倒残差块，用于构建MobileNetV4的基本单元。  
 """  
 def \_\_init\_\_(self, inp, oup, stride, expand\_ratio, act=False):  
 super(InvertedResidual, self).\_\_init\_\_()  
 self.stride = stride  
 hidden\_dim = int(round(inp \* expand\_ratio)) # 计算扩展后的维度  
 self.block = nn.Sequential()  
 if expand\_ratio != 1:  
 self.block.add\_module('exp\_1x1', conv\_2d(inp, hidden\_dim, kernel\_size=1, stride=1)) # 扩展卷积  
 self.block.add\_module('conv\_3x3', conv\_2d(hidden\_dim, hidden\_dim, kernel\_size=3, stride=stride, groups=hidden\_dim)) # 深度卷积  
 self.block.add\_module('red\_1x1', conv\_2d(hidden\_dim, oup, kernel\_size=1, stride=1, act=act)) # 投影卷积  
 self.use\_res\_connect = self.stride == 1 and inp == oup # 判断是否使用残差连接  
  
 def forward(self, x):  
 if self.use\_res\_connect:  
 return x + self.block(x) # 使用残差连接  
 else:  
 return self.block(x)  
  
def build\_blocks(layer\_spec):  
 """  
 根据层规格构建相应的层。  
   
 Args:  
 layer\_spec: 层的规格信息  
   
 Returns:  
 nn.Sequential: 构建的层序列  
 """  
 if not layer\_spec.get('block\_name'):  
 return nn.Sequential()  
 block\_names = layer\_spec['block\_name']  
 layers = nn.Sequential()  
 if block\_names == "convbn":  
 for i in range(layer\_spec['num\_blocks']):  
 args = dict(zip(['inp', 'oup', 'kernel\_size', 'stride'], layer\_spec['block\_specs'][i]))  
 layers.add\_module(f"convbn\_{i}", conv\_2d(\*\*args))  
 elif block\_names == "uib":  
 for i in range(layer\_spec['num\_blocks']):  
 args = dict(zip(['inp', 'oup', 'start\_dw\_kernel\_size', 'middle\_dw\_kernel\_size', 'middle\_dw\_downsample', 'stride', 'expand\_ratio'], layer\_spec['block\_specs'][i]))  
 layers.add\_module(f"uib\_{i}", InvertedResidual(\*\*args))  
 return layers  
  
class MobileNetV4(nn.Module):  
 """  
 MobileNetV4模型类，构建整个网络结构。  
 """  
 def \_\_init\_\_(self, model):  
 super().\_\_init\_\_()  
 assert model in MODEL\_SPECS.keys() # 确保模型在支持的规格中  
 self.spec = MODEL\_SPECS[model]  
 # 构建各个层  
 self.conv0 = build\_blocks(self.spec['conv0'])  
 self.layer1 = build\_blocks(self.spec['layer1'])  
 self.layer2 = build\_blocks(self.spec['layer2'])  
 self.layer3 = build\_blocks(self.spec['layer3'])  
 self.layer4 = build\_blocks(self.spec['layer4'])  
 self.layer5 = build\_blocks(self.spec['layer5'])  
 self.features = nn.ModuleList([self.conv0, self.layer1, self.layer2, self.layer3, self.layer4, self.layer5]) # 将所有层放入ModuleList  
  
 def forward(self, x):  
 features = []  
 for f in self.features:  
 x = f(x) # 逐层前向传播  
 features.append(x) # 收集特征  
 return features  
  
# 创建不同版本的MobileNetV4模型  
def MobileNetV4ConvSmall():  
 return MobileNetV4('MobileNetV4ConvSmall')  
  
# 主程序入口  
if \_\_name\_\_ == '\_\_main\_\_':  
 model = MobileNetV4ConvSmall() # 创建MobileNetV4ConvSmall模型  
 inputs = torch.randn((1, 3, 640, 640)) # 随机输入  
 res = model(inputs) # 前向传播  
 for i in res:  
 print(i.size()) # 打印每层输出的尺寸  
```  
  
### 代码说明：  
1. \*\*模型规格\*\*：定义了不同版本的 MobileNetV4 的结构，包括每一层的卷积块和参数。  
2. \*\*卷积函数\*\*：`conv\_2d` 函数用于创建一个包含卷积、批归一化和激活函数的序列。  
3. \*\*倒残差块\*\*：`InvertedResidual` 类实现了 MobileNetV4 的基本构建块，支持残差连接。  
4. \*\*构建层\*\*：`build\_blocks` 函数根据给定的规格构建相应的层。  
5. \*\*MobileNetV4 类\*\*：定义了整个模型的结构，包含多个层的构建和前向传播逻辑。  
6. \*\*模型实例化\*\*：提供了创建不同版本 MobileNetV4 模型的函数，并在主程序中展示了如何使用这些模型。```

这个程序文件定义了一个名为 `mobilenetv4.py` 的深度学习模型，主要实现了 MobileNetV4 的不同变体，包括小型、中型、大型和混合型模型。文件中使用了 PyTorch 框架来构建神经网络。  
  
首先，文件引入了一些必要的库和模块，包括 `torch` 和 `torch.nn`，并定义了一些常量和模型规格。模型规格以字典的形式存储，包含了不同层的结构信息，例如卷积层的数量、参数和类型。这些规格分为多个部分，分别对应不同的模型变体（如 `MobileNetV4ConvSmall`、`MobileNetV4ConvMedium` 等）。  
  
接下来，定义了一个 `make\_divisible` 函数，用于确保网络中所有层的通道数都是8的倍数。这是为了提高模型的效率和兼容性。  
  
然后，定义了一个 `conv\_2d` 函数，用于创建2D卷积层，并可选择性地添加批归一化和激活函数（ReLU6）。接着，定义了 `InvertedResidual` 类和 `UniversalInvertedBottleneckBlock` 类，这两个类实现了 MobileNetV4 中的倒残差块和通用倒瓶颈块。它们通过不同的卷积层组合来实现特征提取和维度变换。  
  
`build\_blocks` 函数根据给定的层规格构建相应的网络层。它根据层的类型（如 `convbn`、`uib`、`fused\_ib`）选择合适的构建方式，并返回一个包含多个层的序列。  
  
`MobileNetV4` 类是整个模型的核心，它根据传入的模型类型（如 `MobileNetV4ConvSmall` 等）构建相应的网络结构。构造函数中调用 `build\_blocks` 函数来构建各个层，并将它们存储在一个模块列表中。`forward` 方法定义了前向传播的过程，并在特定的缩放比例下提取特征。  
  
最后，文件提供了几个函数（如 `MobileNetV4ConvSmall`、`MobileNetV4ConvMedium` 等）用于实例化不同的 MobileNetV4 模型。主程序部分测试了 `MobileNetV4ConvSmall` 模型，生成一个随机输入并打印输出特征的尺寸。  
  
整体来看，这个文件实现了 MobileNetV4 的多种变体，结构清晰，便于扩展和修改。通过使用模块化的设计，用户可以方便地构建和使用不同的模型，同时保持代码的可读性和可维护性。

```以下是经过简化和注释的核心代码部分，保留了模型的主要结构和功能。  
  
```python  
import torch  
import torch.nn as nn  
import torch.nn.functional as F  
from timm.models.layers import SqueezeExcite  
  
# 定义卷积层和批归一化的组合  
class Conv2d\_BN(torch.nn.Sequential):  
 def \_\_init\_\_(self, in\_channels, out\_channels, kernel\_size=1, stride=1, padding=0):  
 super().\_\_init\_\_()  
 # 添加卷积层  
 self.add\_module('conv', torch.nn.Conv2d(in\_channels, out\_channels, kernel\_size, stride, padding, bias=False))  
 # 添加批归一化层  
 self.add\_module('bn', torch.nn.BatchNorm2d(out\_channels))  
  
 @torch.no\_grad()  
 def switch\_to\_deploy(self):  
 # 将训练模式下的卷积和批归一化合并为一个卷积层  
 conv, bn = self.\_modules.values()  
 w = bn.weight / (bn.running\_var + bn.eps)\*\*0.5  
 w = conv.weight \* w[:, None, None, None]  
 b = bn.bias - bn.running\_mean \* bn.weight / (bn.running\_var + bn.eps)\*\*0.5  
 # 创建新的卷积层并赋值权重和偏置  
 new\_conv = torch.nn.Conv2d(w.size(1) \* conv.groups, w.size(0), w.shape[2:], stride=conv.stride, padding=conv.padding, dilation=conv.dilation, groups=conv.groups)  
 new\_conv.weight.data.copy\_(w)  
 new\_conv.bias.data.copy\_(b)  
 return new\_conv  
  
# 定义高效ViT的基本块  
class EfficientViTBlock(torch.nn.Module):  
 def \_\_init\_\_(self, embed\_dim, key\_dim, num\_heads=8, window\_size=7):  
 super().\_\_init\_\_()  
 # 第一层卷积和激活  
 self.dw0 = Conv2d\_BN(embed\_dim, embed\_dim, kernel\_size=3, stride=1, padding=1)  
 # 注意力机制  
 self.mixer = LocalWindowAttention(embed\_dim, key\_dim, num\_heads, window\_resolution=window\_size)  
 # 第二层卷积和激活  
 self.dw1 = Conv2d\_BN(embed\_dim, embed\_dim, kernel\_size=3, stride=1, padding=1)  
  
 def forward(self, x):  
 # 前向传播  
 x = self.dw0(x)  
 x = self.mixer(x)  
 x = self.dw1(x)  
 return x  
  
# 定义局部窗口注意力机制  
class LocalWindowAttention(torch.nn.Module):  
 def \_\_init\_\_(self, dim, key\_dim, num\_heads=8, window\_resolution=7):  
 super().\_\_init\_\_()  
 self.attn = CascadedGroupAttention(dim, key\_dim, num\_heads, resolution=window\_resolution)  
  
 def forward(self, x):  
 # 直接应用注意力机制  
 return self.attn(x)  
  
# 定义高效ViT模型  
class EfficientViT(torch.nn.Module):  
 def \_\_init\_\_(self, img\_size=224, embed\_dim=[64, 128, 192], depth=[1, 2, 3], num\_heads=[4, 4, 4]):  
 super().\_\_init\_\_()  
 self.patch\_embed = Conv2d\_BN(3, embed\_dim[0] // 8, kernel\_size=3, stride=2, padding=1)  
 self.blocks = nn.ModuleList()  
 for i in range(len(depth)):  
 for \_ in range(depth[i]):  
 self.blocks.append(EfficientViTBlock(embed\_dim[i], embed\_dim[i] // num\_heads[i], num\_heads[i]))  
  
 def forward(self, x):  
 x = self.patch\_embed(x)  
 for block in self.blocks:  
 x = block(x)  
 return x  
  
# 实例化模型  
if \_\_name\_\_ == '\_\_main\_\_':  
 model = EfficientViT()  
 inputs = torch.randn((1, 3, 224, 224)) # 输入图像的尺寸  
 res = model(inputs) # 前向传播  
 print(res.size()) # 输出结果的尺寸  
```  
  
### 代码注释说明：  
1. \*\*Conv2d\_BN\*\*: 这个类结合了卷积层和批归一化层，提供了一个方便的接口来初始化和切换到部署模式。  
2. \*\*EfficientViTBlock\*\*: 这是高效ViT的基本构建块，包含卷积层和局部窗口注意力机制。  
3. \*\*LocalWindowAttention\*\*: 实现了局部窗口注意力机制，使用了级联组注意力。  
4. \*\*EfficientViT\*\*: 这是整个模型的定义，包含了多个块和输入的嵌入层。  
5. \*\*主程序\*\*: 实例化模型并进行前向传播，打印输出结果的尺寸。```

这个程序文件 `efficientViT.py` 实现了一个高效的视觉变换器（Efficient Vision Transformer，EfficientViT）模型架构，主要用于图像处理的下游任务。代码中包含了多个类和函数，构成了整个模型的结构和功能。  
  
首先，文件导入了必要的库，包括 PyTorch 及其相关模块，此外还引入了 `SqueezeExcite` 层，这是一个用于增强特征表示的机制。接着，定义了一些基础的组件，例如 `Conv2d\_BN` 类，它结合了卷积层和批归一化层，并提供了一个用于推理时的转换方法 `switch\_to\_deploy`，可以将批归一化层的参数融合到卷积层中，从而提高推理效率。  
  
接下来，定义了 `replace\_batchnorm` 函数，用于在模型中替换掉所有的批归一化层，以便在推理时使用更高效的结构。然后是 `PatchMerging` 类，它实现了将输入特征图进行合并的操作，使用了卷积和激活函数，结合了 SqueezeExcite 机制，增强了特征的表达能力。  
  
`Residual` 类实现了残差连接，允许在训练时引入一定的随机性，以提高模型的鲁棒性。`FFN` 类实现了前馈神经网络的结构，包含两个卷积层和一个激活函数。  
  
`CascadedGroupAttention` 和 `LocalWindowAttention` 类实现了不同类型的注意力机制，前者用于处理全局特征，后者则专注于局部窗口内的特征。注意力机制是视觉变换器的核心部分，通过计算输入特征之间的关系来增强模型的表现。  
  
`EfficientViTBlock` 类则是整个模型的基本构建块，结合了卷积、前馈网络和注意力机制，形成一个完整的处理单元。`EfficientViT` 类是模型的主要结构，负责将输入图像进行嵌入、经过多个块的处理，并输出特征。  
  
在模型的初始化过程中，定义了不同的超参数，例如图像大小、补丁大小、嵌入维度、深度、头数等。模型的不同变体（如 `EfficientViT\_M0` 到 `EfficientViT\_M5`）通过字典形式定义了不同的配置。  
  
最后，文件提供了一些函数用于创建不同配置的模型，并加载预训练权重，若需要，还可以融合批归一化层。`update\_weight` 函数用于更新模型的权重，确保加载的权重与模型结构匹配。  
  
在主程序部分，创建了一个 `EfficientViT\_M0` 模型实例，并生成随机输入进行测试，输出每个阶段的特征图大小。这段代码展示了如何使用该模型进行推理。整体来看，这个文件实现了一个灵活且高效的视觉变换器架构，适用于多种图像处理任务。

```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 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  
 # 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:  
 # 部署模式下的卷积操作  
 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:  
 # 非部署模式下的卷积操作，先经过批归一化  
 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  
  
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) # 池化操作  
 x = self.act(x) # 激活操作  
 return x  
  
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  
 # 初始化网络的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()  
 # 构建多个Block  
 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)  
  
 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  
  
# 测试模型  
if \_\_name\_\_ == '\_\_main\_\_':  
 inputs = torch.randn((1, 3, 640, 640)) # 随机输入  
 model = VanillaNet() # 创建模型  
 pred = model(inputs) # 前向传播  
 print(pred.size()) # 输出预测结果的尺寸  
```  
  
### 代码核心部分说明：  
1. \*\*激活函数类（activation）\*\*：实现了一个自定义的激活函数，支持在训练和部署模式之间切换，并融合了批归一化（Batch Normalization）层。  
2. \*\*Block类\*\*：表示网络中的一个基本模块，包含卷积层、池化层和激活函数。根据是否处于部署模式，选择不同的结构。  
3. \*\*VanillaNet类\*\*：整个网络的构建，包含stem部分和多个Block。前向传播过程中依次通过这些模块，最终输出特征图。  
4. \*\*测试部分\*\*：在主程序中，创建一个随机输入并通过模型进行前向传播，输出结果的尺寸。```

这个程序文件 `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。文件开头包含版权信息和许可协议，表明该程序是自由软件，可以在 MIT 许可证下进行再分发和修改。  
  
文件中导入了 PyTorch 库及其相关模块，使用了 `timm` 库中的一些功能，如权重初始化和 DropPath。接下来，定义了一个名为 `activation` 的类，它继承自 `nn.ReLU`，用于实现自定义的激活函数。该类在初始化时创建了一个可学习的权重参数和一个批量归一化层，并定义了前向传播的方法。在前向传播中，根据 `deploy` 标志选择不同的计算路径。  
  
接着，定义了一个 `Block` 类，表示模型中的基本构建块。该类包含两个卷积层和一个激活层，支持可选的自适应池化。`Block` 类的 `forward` 方法实现了前向传播逻辑，使用 `activation` 类进行激活处理，并根据步幅选择池化操作。  
  
然后，定义了 `VanillaNet` 类，这是整个模型的核心。它包含多个 `Block`，并在初始化时设置输入通道、类别数、各层的维度、丢弃率等参数。模型的前向传播方法处理输入数据，并在不同的尺度上提取特征。  
  
文件中还实现了权重更新的辅助函数 `update\_weight`，用于加载预训练模型的权重。随后，定义了一系列函数（如 `vanillanet\_5` 到 `vanillanet\_13\_x1\_5\_ada\_pool`），这些函数用于创建不同配置的 VanillaNet 模型，并支持加载预训练权重。  
  
最后，在 `\_\_main\_\_` 部分，创建了一个随机输入并实例化了一个 VanillaNet 模型，进行前向传播并打印输出特征的尺寸。这部分代码用于测试模型的基本功能。  
  
整体来看，该文件实现了一个灵活且可扩展的卷积神经网络结构，适用于多种图像处理任务，具有可调节的激活函数和多种配置选项。

### 程序整体功能和构架概括  
  
该项目包含多个深度学习模型的实现，主要用于图像处理任务。每个文件定义了不同的模型架构，利用 PyTorch 框架构建和训练深度学习模型。项目中包括了不同类型的卷积神经网络（如 MobileNetV4 和 VanillaNet）以及视觉变换器（EfficientViT），这些模型在结构上各具特色，适用于不同的应用场景。整体架构模块化，便于扩展和修改，同时提供了用户友好的接口以便于模型的实例化和推理。  
  
### 文件功能整理表  
  
| 文件名 | 功能描述 |  
|---------------------|-------------------------------------------------------------------------------------------|  
| `ui.py` | 启动一个 Streamlit Web 应用，运行指定的 Python 脚本（如 `web.py`），用于用户界面展示和交互。 |  
| `mobilenetv4.py` | 实现 MobileNetV4 模型的不同变体，定义了卷积层、倒残差块和特征提取的前向传播逻辑。 |  
| `efficientViT.py` | 实现 Efficient Vision Transformer 模型，包含多个注意力机制和特征提取模块，适用于图像处理任务。 |  
| `VanillaNet.py` | 实现 VanillaNet 模型，定义了基本的卷积块和激活函数，支持多种配置选项，适用于图像分类等任务。 |  
  
这个表格清晰地总结了每个文件的主要功能，帮助理解整个项目的结构和目的。