# 改进yolo11-DBB等200+全套创新点大全：人员异常行为识别检测系统源码＆数据集全套

## 3.背景

研究背景与意义  
  
随着社会的快速发展和城市化进程的加快，公共安全问题日益凸显，尤其是人员异常行为的识别与监测，成为保障社会安全的重要课题。异常行为的识别不仅涉及到犯罪预防、公共安全管理，还与人们的日常生活息息相关。因此，构建一个高效、准确的人员异常行为识别检测系统显得尤为重要。近年来，深度学习技术的飞速发展为计算机视觉领域带来了新的机遇，尤其是目标检测算法的不断改进，使得异常行为识别的准确性和实时性得到了显著提升。  
  
本研究基于改进的YOLOv11算法，旨在开发一套高效的人员异常行为识别检测系统。YOLO（You Only Look Once）系列算法因其快速的检测速度和较高的准确率而广泛应用于各种目标检测任务。通过对YOLOv11的改进，我们希望能够进一步提升系统在复杂环境下的表现，尤其是在低光照、拥挤场景等情况下的异常行为检测能力。  
  
本项目所使用的数据集包含300张经过精确标注的异常行为图像，涵盖了各种可能的异常行为场景。该数据集的单一类别“abnormal-behavior”使得研究更加聚焦，有助于深入分析和理解异常行为的特征及其在不同环境下的表现。通过对数据集的分析和模型的训练，我们期望能够实现对异常行为的高效识别，为公共安全监控、智能安防系统的应用提供技术支持。  
  
综上所述，基于改进YOLOv11的人员异常行为识别检测系统的研究，不仅具有重要的理论意义，也具备广泛的实际应用价值。通过该系统的开发与应用，我们期望能够为提升社会安全水平贡献一份力量。

## 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  
  
本项目所使用的数据集旨在为改进YOLOv11的人员异常行为识别检测系统提供坚实的基础。该数据集的主题为“abnormal behavior7”，专注于识别和分析人类在特定环境下的异常行为模式。数据集中包含的类别数量为1，具体类别为“abnormal-behavior”。这一类别的设定反映了我们对异常行为的深刻理解与研究，旨在通过计算机视觉技术提升对潜在危险或异常事件的检测能力。  
  
在数据集的构建过程中，我们广泛收集了来自不同场景的视频和图像数据，这些场景包括公共场所、工作环境及其他人群密集区域。数据集中的样本涵盖了多种类型的异常行为，如激烈争吵、突然倒地、异常奔跑等。这些行为的多样性确保了模型在训练过程中能够学习到丰富的特征，从而提高其在实际应用中的泛化能力。  
  
此外，为了确保数据集的高质量和准确性，我们对每个样本进行了严格的标注和审核。标注过程不仅考虑了行为的明显特征，还结合了上下文信息，以便更好地理解行为的背景。这种细致的标注方式使得数据集不仅具备了良好的可用性，也为后续的模型训练提供了可靠的依据。  
  
总之，本项目的数据集“abnormal behavior7”通过丰富的样本和精确的标注，为改进YOLOv11的人员异常行为识别检测系统奠定了坚实的基础。我们期待通过这一数据集的应用，推动异常行为检测技术的发展，提升公共安全和人群管理的效率。

## 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  
 """  
 # 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，并捕获返回结果  
 result = subprocess.run(command, shell=True)  
   
 # 检查脚本运行是否成功  
 if result.returncode != 0:  
 print("脚本运行出错。")  
  
# 主程序入口  
if \_\_name\_\_ == "\_\_main\_\_":  
 # 指定要运行的脚本路径  
 script\_path = "web.py" # 这里可以直接指定脚本名称  
  
 # 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. \*\*函数 `run\_script`\*\*：  
 - 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。  
 - 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，并捕获返回结果。  
 - 检查返回码，如果不为 0，表示脚本运行出错，打印错误信息。  
  
3. \*\*主程序入口\*\*：  
 - 在 `\_\_main\_\_` 块中，指定要运行的脚本名称（`web.py`）。  
 - 调用 `run\_script` 函数，传入脚本路径以执行。```

这个程序文件名为 `ui.py`，其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。  
  
首先，程序导入了必要的模块，包括 `sys`、`os` 和 `subprocess`。其中，`sys` 模块用于访问与 Python 解释器相关的变量和函数，`os` 模块提供了与操作系统交互的功能，而 `subprocess` 模块则用于创建新进程、连接到它们的输入/输出/错误管道，并获取它们的返回码。  
  
接下来，程序从 `QtFusion.path` 模块中导入了 `abs\_path` 函数，这个函数的作用是获取给定路径的绝对路径，确保后续的脚本路径是正确的。  
  
在 `run\_script` 函数中，程序接收一个参数 `script\_path`，表示要运行的脚本的路径。函数首先获取当前 Python 解释器的路径，存储在 `python\_path` 变量中。然后，构建一个命令字符串，使用 Streamlit 运行指定的脚本。命令的格式是 `"{python\_path}" -m streamlit run "{script\_path}"`，这意味着使用当前的 Python 解释器来运行 Streamlit 模块，并指定要运行的脚本。  
  
接着，程序使用 `subprocess.run` 方法执行构建好的命令。这个方法会在一个新的 shell 中运行命令，并等待其完成。执行后，程序检查返回码，如果返回码不为 0，表示脚本运行出错，则打印出错误信息。  
  
最后，在 `if \_\_name\_\_ == "\_\_main\_\_":` 语句块中，程序指定了要运行的脚本路径，这里使用 `abs\_path("web.py")` 来获取 `web.py` 的绝对路径。然后调用 `run\_script` 函数来执行这个脚本。  
  
总的来说，这个程序的主要目的是提供一个简单的接口来运行一个 Streamlit Web 应用，确保脚本能够在当前的 Python 环境中正确执行。

```以下是代码中最核心的部分，并附上详细的中文注释：  
  
```python  
import torch  
import torch.nn as nn  
from 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 # 输入维度  
 self.outdim = output\_dim # 输出维度  
 self.degree = degree # 多项式的度数  
 self.kernel\_size = kernel\_size # 卷积核大小  
 self.padding = padding # 填充  
 self.stride = stride # 步幅  
 self.dilation = dilation # 膨胀  
 self.groups = groups # 分组卷积的组数  
 self.base\_activation = nn.SiLU() # 基础激活函数  
 self.conv\_w\_fun = conv\_w\_fun # 卷积权重函数  
 self.ndim = ndim # 数据的维度（1D, 2D, 3D）  
 self.dropout = None # Dropout层初始化  
  
 # 根据维度选择合适的Dropout层  
 if dropout > 0:  
 if ndim == 1:  
 self.dropout = nn.Dropout1d(p=dropout)  
 elif ndim == 2:  
 self.dropout = nn.Dropout2d(p=dropout)  
 elif ndim == 3:  
 self.dropout = nn.Dropout3d(p=dropout)  
  
 # 验证分组参数的有效性  
 if groups <= 0:  
 raise ValueError('groups must be a positive integer')  
 if input\_dim % groups != 0:  
 raise ValueError('input\_dim must be divisible by groups')  
 if output\_dim % groups != 0:  
 raise ValueError('output\_dim must be divisible by groups')  
  
 # 创建基础卷积层和归一化层  
 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)])  
  
 # 多项式权重的形状  
 poly\_shape = (groups, output\_dim // groups, (input\_dim // groups) \* (degree + 1)) + tuple(  
 kernel\_size for \_ in range(ndim))  
  
 # 初始化多项式权重和beta权重  
 self.poly\_weights = nn.Parameter(torch.randn(\*poly\_shape))  
 self.beta\_weights = nn.Parameter(torch.zeros(degree + 1, dtype=torch.float32))  
  
 # 使用Kaiming均匀分布初始化卷积层权重  
 for conv\_layer in self.base\_conv:  
 nn.init.kaiming\_uniform\_(conv\_layer.weight, nonlinearity='linear')  
  
 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)),  
 )  
  
 @lru\_cache(maxsize=128) # 使用缓存避免重复计算Legendre多项式  
 def gram\_poly(self, x, degree):  
 p0 = x.new\_ones(x.size()) # 创建多项式的第0项  
  
 if degree == 0:  
 return p0.unsqueeze(-1) # 如果度数为0，返回常数项  
  
 p1 = x # 第一项为x  
 grams\_basis = [p0, p1] # 初始化多项式基  
  
 # 计算Legendre多项式  
 for i in range(2, degree + 1):  
 p2 = x \* p1 - self.beta(i - 1, i) \* p0 # 递推公式  
 grams\_basis.append(p2) # 添加新项  
 p0, p1 = p1, p2 # 更新p0和p1  
  
 return torch.cat(grams\_basis, dim=1) # 将多项式基拼接成一个张量  
  
 def forward\_kag(self, x, group\_index):  
 # 对输入应用基础激活函数，并进行卷积操作  
 basis = self.base\_conv[group\_index](self.base\_activation(x))  
  
 # 将x归一化到[-1, 1]范围内以稳定Legendre多项式计算  
 x = torch.tanh(x).contiguous()  
  
 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多项式基  
  
 # 使用卷积权重函数进行卷积操作  
 y = self.conv\_w\_fun(grams\_basis, self.poly\_weights[group\_index],  
 stride=self.stride, dilation=self.dilation,  
 padding=self.padding, groups=1)  
  
 # 应用归一化和激活函数  
 y = self.base\_activation(self.layer\_norm[group\_index](y + basis))  
  
 return y  
  
 def forward(self, x):  
 # 将输入x按组分割  
 split\_x = torch.split(x, self.inputdim // self.groups, dim=1)  
 output = []  
 for group\_ind, \_x in enumerate(split\_x):  
 y = self.forward\_kag(\_x.clone(), group\_ind) # 对每个组进行前向传播  
 output.append(y.clone()) # 保存输出  
 y = torch.cat(output, dim=1) # 将所有组的输出拼接  
 return y  
```  
  
### 代码核心部分说明：  
1. \*\*KAGNConvNDLayer类\*\*：这是一个自定义的卷积层，支持任意维度的卷积（1D、2D、3D），结合了多项式基和卷积操作。  
2. \*\*初始化方法\*\*：在初始化中，设置了输入输出维度、卷积参数、激活函数、Dropout层等，并创建了基础卷积层和归一化层。  
3. \*\*Gram多项式计算\*\*：使用`gram\_poly`方法计算Legendre多项式，利用递推公式生成多项式基。  
4. \*\*前向传播\*\*：`forward\_kag`方法实现了对每个组的前向传播，包括激活、卷积和归一化等操作。  
5. \*\*整体前向传播\*\*：`forward`方法将输入分组后，逐组进行前向传播并拼接结果。```

``````python  
import torch  
import torch.nn as nn  
  
class AGLU(nn.Module):  
 """统一激活函数模块，来源于 https://github.com/kostas1515/AGLU。"""  
  
 def \_\_init\_\_(self, device=None, dtype=None) -> None:  
 """初始化统一激活函数模块。"""  
 super().\_\_init\_\_()  
 # 使用Softplus作为基础激活函数，beta设置为-1.0  
 self.act = nn.Softplus(beta=-1.0)  
 # 初始化lambda参数，范围为[0, 1]，并将其设为可学习参数  
 self.lambd = nn.Parameter(nn.init.uniform\_(torch.empty(1, device=device, dtype=dtype)))   
 # 初始化kappa参数，范围为[0, 1]，并将其设为可学习参数  
 self.kappa = nn.Parameter(nn.init.uniform\_(torch.empty(1, device=device, dtype=dtype)))   
  
 def forward(self, x: torch.Tensor) -> torch.Tensor:  
 """计算统一激活函数的前向传播。"""  
 # 将lambda参数限制在最小值0.0001，避免出现负值或零  
 lam = torch.clamp(self.lambd, min=0.0001)  
 # 计算激活值，公式为 exp((1 / lam) \* act(kappa \* x - log(lam)))  
 return torch.exp((1 / lam) \* self.act((self.kappa \* x) - torch.log(lam)))  
```  
  
### 代码注释说明：  
1. \*\*导入必要的库\*\*：导入`torch`和`torch.nn`模块，后者提供了构建神经网络所需的基础组件。  
2. \*\*AGLU类\*\*：定义了一个名为`AGLU`的类，继承自`nn.Module`，表示一个自定义的激活函数模块。  
3. \*\*初始化方法\*\*：  
 - `super().\_\_init\_\_()`：调用父类的初始化方法。  
 - `self.act`：定义了一个基础激活函数`Softplus`，其beta参数设置为-1.0。  
 - `self.lambd`和`self.kappa`：初始化两个可学习参数，分别为lambda和kappa，使用均匀分布进行初始化。  
4. \*\*前向传播方法\*\*：  
 - `torch.clamp(self.lambd, min=0.0001)`：将lambda参数限制在最小值0.0001，以避免计算中的数值不稳定。  
 - `torch.exp(...)`：计算激活值，使用公式`exp((1 / lam) \* act(kappa \* x - log(lam)))`，其中`x`是输入张量。```

这个程序文件定义了一个名为 `activation.py` 的模块，主要用于实现一种名为 AGLU 的激活函数。该模块使用了 PyTorch 库来构建神经网络的激活功能。  
  
首先，文件导入了 PyTorch 的核心库和神经网络模块。接着，定义了一个名为 `AGLU` 的类，该类继承自 `nn.Module`，这是 PyTorch 中所有神经网络模块的基类。  
  
在 `AGLU` 类的初始化方法 `\_\_init\_\_` 中，首先调用了父类的初始化方法。然后，定义了一个激活函数 `self.act`，使用了 `nn.Softplus`，其参数 `beta` 被设置为 -1.0。接下来，定义了两个可学习的参数 `lambd` 和 `kappa`，这两个参数的初始值是通过均匀分布随机初始化的。它们被包装成 `nn.Parameter`，使得在训练过程中可以被优化。  
  
在 `forward` 方法中，定义了前向传播的计算过程。输入的张量 `x` 经过处理后，首先对 `lambd` 参数进行限制，确保其最小值为 0.0001，以避免数值不稳定。然后，计算并返回 AGLU 激活函数的输出。这个输出是通过对输入 `x` 进行一系列变换得到的，包括应用 `self.kappa` 和 `self.act`，以及对 `lam` 的对数进行处理。  
  
总的来说，这个模块实现了一个自定义的激活函数 AGLU，结合了可学习的参数，能够在神经网络的训练过程中进行优化，以提高模型的表现。

```以下是简化后的代码，保留了核心部分，并添加了详细的中文注释：  
  
```python  
import torch  
import torch.nn as nn  
import torch.nn.functional as F  
import itertools  
  
# 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):  
 # 在推理阶段将BN层与卷积层融合  
 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  
 # 返回融合后的卷积层  
 return torch.nn.Conv2d(w.size(1), w.size(0), w.shape[2:], stride=conv.stride, padding=conv.padding)  
  
# 定义一个基本的EfficientViT模块  
class EfficientViTBlock(torch.nn.Module):  
 def \_\_init\_\_(self, embed\_dim, key\_dim, num\_heads=8):  
 super().\_\_init\_\_()  
 # 定义深度卷积和前馈网络  
 self.dw = Conv2d\_BN(embed\_dim, embed\_dim, kernel\_size=3, stride=1, padding=1)  
 self.ffn = nn.Sequential(  
 Conv2d\_BN(embed\_dim, embed\_dim \* 2, kernel\_size=1),  
 nn.ReLU(),  
 Conv2d\_BN(embed\_dim \* 2, embed\_dim, kernel\_size=1)  
 )  
 # 定义局部窗口注意力机制  
 self.attn = LocalWindowAttention(embed\_dim, key\_dim, num\_heads)  
  
 def forward(self, x):  
 # 前向传播  
 x = self.dw(x) + x # 残差连接  
 x = self.attn(x) # 注意力机制  
 x = self.ffn(x) + x # 残差连接  
 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):  
 super().\_\_init\_\_()  
 self.attn = CascadedGroupAttention(dim, key\_dim, num\_heads)  
  
 def forward(self, x):  
 # 前向传播，执行注意力计算  
 return self.attn(x)  
  
# 定义EfficientViT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  
  
# 创建EfficientViT模型实例  
if \_\_name\_\_ == '\_\_main\_\_':  
 model = EfficientViT()  
 inputs = torch.randn((1, 3, 640, 640)) # 输入一个640x640的图像  
 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\*\*: 该类实现了卷积层和批归一化层的组合，提供了一个`switch\_to\_deploy`方法用于在推理阶段将它们融合，以提高推理效率。  
2. \*\*EfficientViTBlock\*\*: 这是EfficientViT的基本构建块，包含深度卷积、前馈网络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`CascadedGroupAttention`来计算注意力。  
4. \*\*EfficientViT\*\*: 整个模型的定义，包含了输入图像的嵌入和多个EfficientViT块的堆叠。  
5. \*\*主程序\*\*: 创建模型实例并进行一次前向传播，输出结果的尺寸。```

这个程序文件实现了一个名为 EfficientViT 的深度学习模型架构，主要用于图像处理任务。它的设计灵感来源于视觉变换器（Vision Transformer），并结合了一些高效的卷积操作和注意力机制，以提高模型的性能和效率。  
  
文件的开头部分包含了一些必要的库导入，包括 PyTorch 和 timm 库中的 SqueezeExcite 层。接着，定义了一个 `Conv2d\_BN` 类，该类继承自 `torch.nn.Sequential`，用于构建带有批归一化的卷积层。该类还包含一个 `switch\_to\_deploy` 方法，用于在推理阶段将批归一化层融合到卷积层中，以提高推理速度。  
  
接下来，定义了一个 `replace\_batchnorm` 函数，用于在模型中替换批归一化层为恒等映射，以便在推理时提高效率。然后，定义了 `PatchMerging` 类，它实现了对输入特征图的合并操作，通常用于将特征图的分辨率降低。  
  
`Residual` 类实现了残差连接，允许在训练过程中随机丢弃部分输入，以提高模型的鲁棒性。`FFN` 类实现了前馈神经网络结构，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，前者是级联组注意力，后者是局部窗口注意力。这些注意力机制通过对输入特征图进行加权处理，能够有效捕捉图像中的重要信息。  
  
`EfficientViTBlock` 类则是 EfficientViT 模型的基本构建块，结合了卷积层、前馈网络和注意力机制，形成一个完整的模块。`EfficientViT` 类是整个模型的核心，负责构建模型的不同阶段，并通过多个 `EfficientViTBlock` 进行特征提取。  
  
在模型初始化时，用户可以指定输入图像的大小、补丁大小、嵌入维度、深度、注意力头数等参数。模型的前向传播方法会依次通过补丁嵌入层和多个块，最终输出特征图。  
  
最后，文件定义了一些预设的模型配置（如 EfficientViT\_m0 到 EfficientViT\_m5），并提供了相应的构建函数（如 `EfficientViT\_M0` 等），用于创建模型实例并加载预训练权重。  
  
整个程序结构清晰，模块化设计使得各个部分易于理解和扩展，适合用于各种下游视觉任务。

### 程序整体功能和构架概括  
  
该程序是一个基于深度学习的框架，主要用于实现和训练自定义的神经网络模型。它包含多个模块，每个模块负责不同的功能，包括用户界面、激活函数、卷积层的实现以及高效的视觉变换器（ViT）模型。整体架构设计旨在提高模型的可扩展性和可维护性，同时通过自定义的激活函数和网络结构来提升模型的性能。  
  
### 文件功能整理表  
  
| 文件名 | 功能描述 |  
|---------------------|-----------------------------------------------------------------------|  
| `ui.py` | 提供用户界面，允许用户与模型进行交互，启动和管理 Streamlit Web 应用。 |  
| `kagn\_conv.py` | 实现自定义的卷积层，可能结合了特定的卷积操作和参数优化。 |  
| `activation.py` | 定义自定义的激活函数 AGLU，包含可学习的参数以优化神经网络的性能。 |  
| `efficientViT.py` | 实现高效的视觉变换器（ViT）模型，优化了计算效率和模型性能，适用于图像处理任务。 |  
  
以上表格总结了每个文件的主要功能，便于理解程序的整体结构和各个模块的作用。