# 改进yolo11-EMSC等200+全套创新点大全：牛结节性皮肤病检测系统源码＆数据集全套

## 3.背景

研究背景与意义  
  
牛结节性皮肤病是一种影响牛只健康的重要疾病，其主要特征是皮肤上出现结节或肿块。这种疾病不仅对牛只的生长和生产性能造成影响，还可能导致经济损失和食品安全隐患。因此，及时、准确地检测和诊断牛结节性皮肤病显得尤为重要。传统的检测方法往往依赖于兽医的经验和人工检查，效率低下且容易受到主观因素的影响。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系统逐渐成为解决这一问题的有效手段。  
  
本研究旨在基于改进的YOLOv11模型，构建一个高效的牛结节性皮肤病检测系统。YOLO（You Only Look Once）系列模型以其实时性和高准确率在目标检测领域取得了显著成果。通过对YOLOv11进行改进，我们希望能够进一步提升其在特定场景下的检测性能，尤其是在牛结节性皮肤病的检测任务中。我们的数据集包含278张经过标注的牛只图像，涵盖了两类目标：牛和结节。这一数据集为模型的训练和评估提供了坚实的基础。  
  
此外，随着农业现代化的推进，数字化技术在畜牧业中的应用日益广泛。通过构建智能化的检测系统，不仅可以提高牛结节性皮肤病的检测效率，还能为养殖户提供科学的管理依据，促进健康养殖和可持续发展。因此，本研究不仅具有重要的学术价值，也具有广泛的应用前景，能够为提高畜牧业的生产效率和动物福利做出贡献。

## 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模型，以实现对牛结节性皮肤病的高效检测。为此，我们构建了一个专门的数据集，旨在为模型的训练和评估提供丰富的样本和准确的标注。该数据集的主题为“annotate image”，主要包含两类目标：牛（cow）和肿块（lump）。在数据集中，我们精心挑选了多种场景下的牛只图像，确保覆盖不同的环境、光照条件以及牛只的姿态变化，以增强模型的泛化能力。  
  
数据集中包含的牛只图像均经过专业的标注，确保每一张图像中的牛只和肿块都被准确识别和框定。标注过程中，我们采用了严格的质量控制措施，以确保数据的准确性和一致性。这一过程不仅包括对牛只的识别，还特别关注肿块的形态特征，力求在不同的病理状态下提供详尽的标注信息。  
  
在类别数量方面，本数据集共设定了两个类别，分别为“cow”和“lump”。这种简化的类别设置使得模型在训练过程中能够更加专注于关键特征的学习，从而提高检测的准确性和效率。通过对这两个类别的深入学习，YOLOv11模型将能够在实际应用中快速识别出牛只及其可能存在的皮肤病变，进而为养殖业提供科学的疾病预警和管理方案。  
  
总之，本项目的数据集不仅在数量上满足了训练需求，更在质量上力求精益求精，为改进YOLOv11的牛结节性皮肤病检测系统奠定了坚实的基础。通过这一数据集的应用，我们期待能够推动牛只健康管理的智能化进程，提高养殖效率，降低经济损失。

## 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

```以下是经过精简和注释的核心代码部分，保留了主要的功能和结构：  
  
```python  
import torch  
import torch.nn as nn  
import torch.nn.functional as F  
from timm.layers import trunc\_normal\_  
  
# 定义相对位置函数，用于生成卷积核的坐标  
def rel\_pos(kernel\_size):  
 tensors = [torch.linspace(-1, 1, steps=kernel\_size) for \_ in range(2)]  
 kernel\_coord = torch.stack(torch.meshgrid(\*tensors), dim=-0) # 生成网格坐标  
 kernel\_coord = kernel\_coord.unsqueeze(0) # 增加一个维度  
 return kernel\_coord  
  
# 定义SMP卷积层  
class SMPConv(nn.Module):  
 def \_\_init\_\_(self, planes, kernel\_size, n\_points, stride, padding, groups):  
 super().\_\_init\_\_()  
 self.planes = planes  
 self.kernel\_size = kernel\_size  
 self.n\_points = n\_points  
 self.init\_radius = 2 \* (2/kernel\_size)  
  
 # 生成卷积核坐标  
 kernel\_coord = rel\_pos(kernel\_size)  
 self.register\_buffer('kernel\_coord', kernel\_coord)  
  
 # 初始化权重坐标  
 weight\_coord = torch.empty(1, n\_points, 2)  
 nn.init.trunc\_normal\_(weight\_coord, std=0.2, a=-1., b=1.)  
 self.weight\_coord = nn.Parameter(weight\_coord)  
  
 # 初始化半径  
 self.radius = nn.Parameter(torch.empty(1, n\_points).unsqueeze(-1).unsqueeze(-1))  
 self.radius.data.fill\_(value=self.init\_radius)  
  
 # 初始化权重  
 weights = torch.empty(1, planes, n\_points)  
 trunc\_normal\_(weights, std=.02)  
 self.weights = nn.Parameter(weights)  
  
 def forward(self, x):  
 kernels = self.make\_kernels().unsqueeze(1) # 生成卷积核  
 x = x.contiguous()  
 kernels = kernels.contiguous()  
  
 # 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)  
 elif x.dtype == torch.float16:  
 x = \_DepthWiseConv2dImplicitGEMMFP16.apply(x, kernels)  
 else:  
 raise TypeError("Only support fp32 and fp16, get {}".format(x.dtype))  
 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)  
 diff = F.relu(1 - torch.sum(torch.abs(diff), dim=2) / self.radius) # 计算权重  
  
 # 生成最终的卷积核  
 kernels = torch.matmul(self.weights, diff.reshape(1, self.n\_points, -1)) # 加权求和  
 kernels = kernels.reshape(1, self.planes, \*self.kernel\_coord.shape[2:]) # 重新调整形状  
 kernels = kernels.squeeze(0)  
 kernels = torch.flip(kernels.permute(0, 2, 1), dims=(1,)) # 翻转维度  
 return kernels  
  
# 定义SMPCNN模块  
class SMPCNN(nn.Module):  
 def \_\_init\_\_(self, in\_channels, out\_channels, kernel\_size, stride, groups, n\_points=None):  
 super().\_\_init\_\_()  
 padding = kernel\_size // 2  
 self.smp = SMPConv(in\_channels, kernel\_size, n\_points, stride, padding, groups) # 使用SMP卷积  
 self.small\_conv = nn.Conv2d(in\_channels, out\_channels, kernel\_size=5, stride=stride, padding=2, groups=groups, bias=False) # 小卷积  
  
 def forward(self, inputs):  
 out = self.smp(inputs) # 通过SMP卷积  
 out += self.small\_conv(inputs) # 加上小卷积的输出  
 return out  
  
# 定义SMPBlock模块  
class SMPBlock(nn.Module):  
 def \_\_init\_\_(self, in\_channels, dw\_channels, lk\_size, drop\_path):  
 super().\_\_init\_\_()  
 self.pw1 = nn.Sequential(nn.Conv2d(in\_channels, dw\_channels, kernel\_size=1, stride=1, padding=0), nn.BatchNorm2d(dw\_channels), nn.ReLU())  
 self.pw2 = nn.Conv2d(dw\_channels, in\_channels, kernel\_size=1, stride=1, padding=0)  
 self.large\_kernel = SMPCNN(in\_channels=dw\_channels, out\_channels=dw\_channels, kernel\_size=lk\_size, stride=1, groups=dw\_channels)  
 self.drop\_path = nn.Identity() # 可选的DropPath  
  
 def forward(self, x):  
 out = self.pw1(x) # 通过第一个卷积块  
 out = self.large\_kernel(out) # 通过大卷积  
 out = self.pw2(out) # 通过第二个卷积块  
 return x + self.drop\_path(out) # 残差连接  
```  
  
### 代码说明：  
1. \*\*rel\_pos\*\*: 生成卷积核的相对位置坐标。  
2. \*\*SMPConv\*\*: 自定义的卷积层，支持动态生成卷积核，利用权重坐标和半径来调整卷积核的形状。  
3. \*\*SMPCNN\*\*: 结合了SMP卷积和小卷积的模块，增强了特征提取能力。  
4. \*\*SMPBlock\*\*: 组合多个卷积层，使用残差连接，支持DropPath功能。  
  
这些模块构成了一个灵活的卷积神经网络架构，能够适应不同的输入和需求。```

该文件 `SMPConv.py` 定义了一些用于构建深度学习模型的模块，主要包括一个自定义的卷积层 `SMPConv` 以及一些其他的辅助类和函数。以下是对代码的逐行分析和解释。  
  
首先，文件导入了必要的库，包括 PyTorch 的核心模块和一些功能模块，如 `nn` 和 `F`。还尝试导入深度可分离卷积的实现，如果导入失败则会捕获异常。  
  
接下来，定义了一个 `rel\_pos` 函数，用于生成相对位置的坐标，这在卷积操作中可能用于计算卷积核的相对位置。  
  
`SMPConv` 类是文件的核心部分，继承自 `nn.Module`。在初始化方法中，定义了一些参数，包括输出通道数、卷积核大小、点数、步幅和填充。使用 `rel\_pos` 函数生成卷积核的坐标，并初始化权重坐标和半径。权重和半径被定义为可学习的参数。  
  
`forward` 方法实现了前向传播逻辑。根据输入张量的类型（32位或16位浮点数），选择不同的深度可分离卷积实现进行计算。`make\_kernels` 方法则负责生成卷积核，计算过程中使用了权重坐标和相对位置坐标的差异，并通过一些数学操作生成最终的卷积核。  
  
`radius\_clip` 方法用于限制半径的值在指定范围内，确保模型的稳定性。  
  
接下来，定义了 `get\_conv2d` 函数，用于根据输入参数选择使用自定义的 `SMPConv` 还是标准的 `nn.Conv2d`。如果满足特定条件，则返回 `SMPConv`，否则返回标准卷积层。  
  
`enable\_sync\_bn` 和 `get\_bn` 函数用于处理批归一化的选择，支持同步批归一化。  
  
`conv\_bn` 和 `conv\_bn\_relu` 函数则是封装了卷积层和批归一化层的组合，后者还添加了 ReLU 激活函数。  
  
`fuse\_bn` 函数用于将卷积层和批归一化层融合，以减少推理时的计算开销。  
  
`SMPCNN` 类是一个包含多个卷积层的网络结构，它在初始化时创建了一个 `SMPConv` 和一个小卷积层。前向传播中将这两个部分的输出相加。  
  
`SMPCNN\_ConvFFN` 类实现了一个前馈网络，包含两个逐点卷积层和一个非线性激活函数。它还实现了残差连接。  
  
最后，`SMPBlock` 类定义了一个块结构，包含逐点卷积、卷积层和非线性激活。它同样实现了残差连接，并支持 dropout。  
  
整体来看，这个文件实现了一种新的卷积层和网络结构，结合了深度可分离卷积和自定义的卷积核生成方法，旨在提高模型的表达能力和计算效率。

```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  
  
# 定义一个自定义的批量归一化类  
class RepBN(nn.Module):  
 def \_\_init\_\_(self, channels):  
 super(RepBN, self).\_\_init\_\_()  
 # 初始化一个可学习的参数 alpha，初始值为 1  
 self.alpha = nn.Parameter(torch.ones(1))  
 # 初始化一个一维批量归一化层  
 self.bn = nn.BatchNorm1d(channels)  
  
 def forward(self, x):  
 # 将输入张量的维度进行转置，以适应 BatchNorm1d 的输入要求  
 x = x.transpose(1, 2)  
 # 进行批量归一化，并加上 alpha 乘以输入 x  
 x = self.bn(x) + self.alpha \* x  
 # 再次转置回原来的维度  
 x = x.transpose(1, 2)  
 return x  
  
# 定义一个线性归一化类  
class LinearNorm(nn.Module):  
 def \_\_init\_\_(self, dim, norm1, norm2, warm=0, step=300000, r0=1.0):  
 super(LinearNorm, self).\_\_init\_\_()  
 # 注册缓冲区用于存储预热步数、当前迭代步数和总步数  
 self.register\_buffer('warm', torch.tensor(warm))  
 self.register\_buffer('iter', torch.tensor(step))  
 self.register\_buffer('total\_step', torch.tensor(step))  
 self.r0 = r0 # 初始比例  
 # 初始化两个归一化层  
 self.norm1 = norm1(dim)  
 self.norm2 = norm2(dim)  
  
 def forward(self, x):  
 # 如果模型处于训练模式  
 if self.training:  
 # 如果还有预热步数  
 if self.warm > 0:  
 # 减少预热步数  
 self.warm.copy\_(self.warm - 1)  
 # 使用 norm1 进行归一化  
 x = self.norm1(x)  
 else:  
 # 计算当前的 lambda 值  
 lamda = self.r0 \* self.iter / self.total\_step  
 # 如果还有迭代步数，减少迭代步数  
 if self.iter > 0:  
 self.iter.copy\_(self.iter - 1)  
 # 使用两个不同的归一化方法  
 x1 = self.norm1(x)  
 x2 = self.norm2(x)  
 # 线性组合两个归一化的结果  
 x = lamda \* x1 + (1 - lamda) \* x2  
 else:  
 # 如果模型处于评估模式，直接使用 norm2 进行归一化  
 x = self.norm2(x)  
 return x  
```  
  
### 代码说明：  
1. \*\*RepBN 类\*\*：实现了一个自定义的批量归一化层，除了进行标准的批量归一化外，还引入了一个可学习的参数 `alpha`，用于对输入进行加权。  
   
2. \*\*LinearNorm 类\*\*：实现了一个线性归一化层，支持在训练过程中逐步从 `norm1` 过渡到 `norm2`。它使用预热机制，在训练的初期使用 `norm1`，然后根据迭代次数动态调整使用 `norm1` 和 `norm2` 的比例。```

这个程序文件 `prepbn.py` 定义了两个神经网络模块，分别是 `RepBN` 和 `LinearNorm`，它们都继承自 PyTorch 的 `nn.Module` 类，用于实现特定的归一化操作。  
  
`RepBN` 类实现了一种改进的批量归一化（Batch Normalization）。在初始化方法中，它接受一个参数 `channels`，并创建了一个可学习的参数 `alpha`，初始值为1，以及一个标准的1维批量归一化层 `bn`。在 `forward` 方法中，输入张量 `x` 首先进行维度转换，将其从形状 (batch\_size, channels, seq\_length) 转换为 (batch\_size, seq\_length, channels)，以便适应批量归一化的要求。接着，应用批量归一化，并将归一化后的结果与输入张量乘以 `alpha` 相加。最后，再次进行维度转换，返回处理后的张量。  
  
`LinearNorm` 类则实现了一种线性归一化策略。它的初始化方法接受多个参数，包括 `dim`（维度）、`norm1` 和 `norm2`（归一化函数）、`warm`（预热步数）、`step`（总步数）和 `r0`（初始比例因子）。在初始化过程中，使用 `register\_buffer` 方法注册了一些张量，用于在训练过程中跟踪预热状态和迭代次数。在 `forward` 方法中，如果模型处于训练状态且 `warm` 大于0，则执行第一次归一化 `norm1`，并减少 `warm` 的值。如果 `warm` 为0，则计算一个比例因子 `lamda`，它是基于当前迭代次数与总步数的比例。接着，分别对输入张量 `x` 应用 `norm1` 和 `norm2`，并根据 `lamda` 的值对这两个归一化结果进行线性组合。如果模型不在训练状态，则直接使用 `norm2` 对输入进行归一化处理。  
  
整体来看，这两个模块的设计旨在增强神经网络的归一化能力，以提高模型的训练效果和稳定性。`RepBN` 通过引入可学习的参数 `alpha` 来调整归一化的影响，而 `LinearNorm` 则通过动态调整归一化策略来适应不同的训练阶段。

### 整体功能和构架概括  
  
该项目的整体功能是实现一个深度学习目标检测模型，主要基于 YOLOv8 架构。项目包含多个模块，每个模块负责特定的功能，以便于构建、训练和推理。以下是对各个模块的概述：  
  
1. \*\*SMPConv.py\*\*: 实现了一种自定义的卷积层 `SMPConv`，结合了深度可分离卷积和相对位置编码，旨在提高模型的表达能力和计算效率。  
  
2. \*\*kacn\_conv.py\*\*: 提供了 KACN（Kacn Activation Convolutional Network）卷积层的实现，支持多维度卷积操作，增强了模型的灵活性和性能。  
  
3. \*\*head.py\*\*: 实现了 YOLOv8 的检测头，包含多种检测策略（如动态头、特征金字塔网络等），用于处理目标检测任务的输出。  
  
4. \*\*prepbn.py\*\*: 定义了改进的归一化模块，包括 `RepBN` 和 `LinearNorm`，旨在提高模型的训练稳定性和效果。  
  
### 文件功能整理表  
  
| 文件名 | 功能描述 |  
|------------------|----------------------------------------------------------------------------------------------|  
| `SMPConv.py` | 实现自定义卷积层 `SMPConv`，结合深度可分离卷积和相对位置编码，增强模型表达能力和计算效率。 |  
| `kacn\_conv.py` | 提供 KACN 卷积层的实现，支持 1D、2D 和 3D 卷积操作，适用于多种输入维度的目标检测任务。 |  
| `head.py` | 实现 YOLOv8 的检测头，包含多种检测策略（如动态头、特征金字塔网络），用于处理目标检测输出。 |  
| `prepbn.py` | 定义改进的归一化模块 `RepBN` 和 `LinearNorm`，提高模型训练的稳定性和效果。 |  
  
这个项目通过模块化的设计，使得每个部分的功能清晰，便于后续的扩展和维护。