```python
import numpy as np
from keras.saving.save import load_model
import cv2
from matplotlib import pyplot as plt
from tqdm.notebook import tqdm
import random
from keras.utils import CustomObjectScope
import PIL.Image as pil_image
import tensorflow.compat.v1 as tf
from tensorflow.keras import models
from tensorflow.keras import regularizers
from tensorflow.keras.activations import softmax
from tensorflow.keras.layers import Conv2D, Conv2DTranspose, \
    GlobalAveragePooling2D, AveragePooling2D, MaxPool2D, UpSampling2D, \
    Activation, Dense, Input, \
    Add, Multiply, Concatenate, concatenate
from tensorflow.keras.models import Model
import os
os.environ['TF_CPP_MIN_LOG_LEVEL'] = '2'
# 使用 tf2.0 以上版本声明
# model
# 第一部分定义模型的各层
# 定义预处理卷积层操作，提取图像特征如纹理特征，颜色特征等
class Convolutional_block(tf.keras.layers.Layer):
    def __init__(self, **kwargs):
        super().__init__(**kwargs)
        self.conv_1 = Conv2D(filters=64, kernel_size=(3, 3), strides=1, padding='same')
        self.conv_2 = Conv2D(filters=64, kernel_size=(3, 3), strides=1, padding='same')
        self.conv_3 = Conv2D(filters=64, kernel_size=(3, 3), strides=1, padding='same')
        self.conv_4 = Conv2D(filters=64, kernel_size=(3, 3), strides=1, padding='same')

    def call(self, X):
        X_1 = self.conv_1(X)
        X_1 = Activation('relu')(X_1)

        X_2 = self.conv_2(X_1)
        X_2 = Activation('relu')(X_2)

        X_3 = self.conv_3(X_2)
        X_3 = Activation('relu')(X_3)

        X_4 = self.conv_4(X_3)
        X_4 = Activation('relu')(X_4)
```

```
        # 定义四层卷积，设置每层 64 个卷积核，卷积核大小为 33，步长 1，输出图像大小一样自动计算 padding 值

        return X_4


# 通道注意力机制：获取到特征图的每个通道的重要程度，然后用这个重要程度去给每个特征赋予一个权重值，从而让神经网络重点关注某些特征通道。
# 提升对当前任务有用的特征图的通道，并抑制对当前任务用处不大的特征通道
class Channel_attention(tf.keras.layers.Layer):
    def __init__(self, C=64, **kwargs):
        super().__init__(**kwargs)
        self.C = C
        self.gap = GlobalAveragePooling2D()
        # 通过全局平均池化，将每个通道的二维特征（H*W）压缩为 1 个实数，将特征图从 [h, w, c] ==> [1,1,c]
        self.dense_middle = Dense(units=2, activation='relu')
        self.dense_sigmoid = Dense(units=self.C, activation='sigmoid')
        # 给每个特征通道生成一个权重值，通过两个全连接层构建通道间的相关性，输出的权重值数目和输入特征图的通道数相同。[1,1,c] ==> [1,1,c]

    def get_config(self):
        config = super().get_config().copy()
        config.update({
            'C': self.C
        })
        return config

    def call(self, X):
        v = self.gap(X)
        # 进行全局平均池化
        fc1 = self.dense_middle(v)
        mu = self.dense_sigmoid(fc1)
        # 通过两个全连接层给每个特征通道生成一个权重值
        U_out = Multiply()([X, mu])
        # 将前面得到的归一化权重加权到每个通道的特征上,逐通道乘以权重系数。[h,w,c]*[1,1,c] ==> [h,w,c]
        return U_out


# 使用平均池化和 Unet 获取上下文的信息和位置信息，进行前置调用定义
# 设计一种五层金字塔结构，通过 5 种并行采样方式将输入的特征图采样到不同大小，
# 帮助分支获取不同尺度的接受域，可以获得原始、局部和全局的信息
# 由深度编码-解码和跳过连接组成，5 个 Unet 间不共享权重，最后使用双线性插值采样到
```

相同大小

```
class Avg_pool_Unet_Upsample_msfe(tf.keras.layers.Layer):
    def __init__(self, avg_pool_size, upsample_rate, **kwargs):
        super().__init__(**kwargs)
        # ---二维平均池化--
        self.avg_pool = AveragePooling2D(pool_size=avg_pool_size, padding='same')

        # --- Unet---
        self.deconv_lst = []
        filter = 512
        #  转置卷积，根据卷积核大小和输出的大小，恢复卷积前的图像尺寸

        for i in range(4):
            self.deconv_lst.append(
                Conv2DTranspose(filters=int(filter   /   2),   kernel_size=[3,   3],   strides=2,
padding='same'))
            filter = int(filter / 2)
        #  分别设置 64、128、256、512 个卷积核的卷积
        self.conv_32_down_lst = []
        for i in range(4):
            self.conv_32_down_lst.append(Conv2D(filters=64,         kernel_size=[3,         3],
activation='relu', padding='same',

kernel_regularizer=regularizers.l2(l=0.001)))
        #  权重 l2 正则化，减少模型的过拟合效果
        self.conv_64_down_lst = []
        for i in range(4):
            self.conv_64_down_lst.append(Conv2D(filters=128,        kernel_size=[3,        3],
activation='relu', padding='same',

kernel_regularizer=regularizers.l2(l=0.001)))

        self.conv_128_down_lst = []
        for i in range(4):
            self.conv_128_down_lst.append(Conv2D(filters=256,       kernel_size=[3,       3],
activation='relu', padding='same',

kernel_regularizer=regularizers.l2(l=0.001)))

        self.conv_256_down_lst = []
        for i in range(4):
            self.conv_256_down_lst.append(Conv2D(filters=512,       kernel_size=[3,       3],
activation='relu', padding='same',
```

```
kernel_regularizer=regularizers.l2(l=0.001)))

        self.conv_512_down_lst = []
        for i in range(4):
            self.conv_512_down_lst.append(Conv2D(filters=1024,      kernel_size=[3,      3],
activation='relu', padding='same',

kernel_regularizer=regularizers.l2(l=0.001)))

        self.conv_32_up_lst = []
        for i in range(3):
            self.conv_32_up_lst.append(Conv2D(filters=64,          kernel_size=[3,          3],
activation='relu', padding='same',

kernel_regularizer=regularizers.l2(l=0.001)))

        self.conv_64_up_lst = []
        for i in range(3):
            self.conv_64_up_lst.append(Conv2D(filters=128,         kernel_size=[3,         3],
activation='relu', padding='same',

kernel_regularizer=regularizers.l2(l=0.001)))

        self.conv_128_up_lst = []
        for i in range(3):
            self.conv_128_up_lst.append(Conv2D(filters=256,        kernel_size=[3,        3],
activation='relu', padding='same',

kernel_regularizer=regularizers.l2(l=0.001)))

        self.conv_256_up_lst = []
        for i in range(3):
            self.conv_256_up_lst.append(Conv2D(filters=512,        kernel_size=[3,        3],
activation='relu', padding='same',

kernel_regularizer=regularizers.l2(l=0.001)))

        self.conv_3 = Conv2D(filters=3, kernel_size=[1, 1])
        # 最大池化层进行特征融合和降维
        self.pooling1_unet = MaxPool2D(pool_size=[2, 2], padding='same')
        self.pooling2_unet = MaxPool2D(pool_size=[2, 2], padding='same')
        self.pooling3_unet = MaxPool2D(pool_size=[2, 2], padding='same')
        self.pooling4_unet = MaxPool2D(pool_size=[2, 2], padding='same')
```

```
        # ---Unet 上采样--
        self.upsample = UpSampling2D(upsample_rate, interpolation='bilinear')

    def get_config(self):
        config = super().get_config().copy()
        config.update({
            'avg_pool_size': self.avg_pool_size,
            'upsample_rate': self.upsample_rate
        })
        return config

    def upsample_and_concat(self, x1, x2, i):
        # skip connection
        deconv = self.deconv_lst[i](x1)
        deconv_output = Concatenate()([deconv, x2])
        return deconv_output

    def unet(self, input):
        # ---Unet  下采样---
        conv1 = input
        for c_32 in self.conv_32_down_lst:
            conv1 = c_32(conv1)
        pool1 = self.pooling1_unet(conv1)

        conv2 = pool1
        for c_64 in self.conv_64_down_lst:
            conv2 = c_64(conv2)
        pool2 = self.pooling2_unet(conv2)

        conv3 = pool2
        for c_128 in self.conv_128_down_lst:
            conv3 = c_128(conv3)
        pool3 = self.pooling3_unet(conv3)

        conv4 = pool3
        for c_256 in self.conv_256_down_lst:
            conv4 = c_256(conv4)
        pool4 = self.pooling4_unet(conv4)

        conv5 = pool4
        for c_512 in self.conv_512_down_lst:
            conv5 = c_512(conv5)

        # ---Unet upsampling---
```

```
        up6 = self.upsample_and_concat(conv5, conv4, 0)

        conv6 = up6
        for c_256 in self.conv_256_up_lst:
            conv6 = c_256(conv6)

        up7 = self.upsample_and_concat(conv6, conv3, 1)
        conv7 = up7
        for c_128 in self.conv_128_up_lst:
            conv7 = c_128(conv7)

        up8 = self.upsample_and_concat(conv7, conv2, 2)
        conv8 = up8
        for c_64 in self.conv_64_up_lst:
            conv8 = c_64(conv8)

        up9 = self.upsample_and_concat(conv8, conv1, 3)
        conv9 = up9
        for c_32 in self.conv_32_up_lst:
            conv9 = c_32(conv9)

        conv10 = self.conv_3(conv9)
        return conv10

    def call(self, X):
        # 池化+unet+双线性插值采样
        avg_pool = self.avg_pool(X)
        unet = self.unet(avg_pool)
        upsample = self.upsample(unet)
        return upsample


# 多尺度特征融合层，调用 Avg_pool_Unet_Upsample_msfe 的操作
class Multi_scale_feature_extraction(tf.keras.layers.Layer):
    def __init__(self, **kwargs):
        super().__init__(**kwargs)

        self.msfe_16 = Avg_pool_Unet_Upsample_msfe(avg_pool_size=16, upsample_rate=16)
        self.msfe_8 = Avg_pool_Unet_Upsample_msfe(avg_pool_size=8, upsample_rate=8)
        self.msfe_4 = Avg_pool_Unet_Upsample_msfe(avg_pool_size=4, upsample_rate=4)
        self.msfe_2 = Avg_pool_Unet_Upsample_msfe(avg_pool_size=2, upsample_rate=2)
        self.msfe_1 = Avg_pool_Unet_Upsample_msfe(avg_pool_size=1, upsample_rate=1)
```

```
    def call(self, X):
        up_sample_16 = self.msfe_16(X)
        up_sample_8 = self.msfe_8(X)
        up_sample_4 = self.msfe_4(X)
        up_sample_2 = self.msfe_2(X)
        up_sample_1 = self.msfe_1(X)
        # 将输入与融合了全局感受野
        msfe_out = Concatenate()([X, up_sample_16, up_sample_8, up_sample_4,
up_sample_2, up_sample_1])
        return msfe_out


# 核选择层
class Kernel_selecting_module(tf.keras.layers.Layer):
    def __init__(self, C=21, **kwargs):
        super().__init__(**kwargs)
        self.C = C
        # 分别使用 33、55、77 尺寸的卷积核提取特征
        self.c_3 = Conv2D(filters=self.C, kernel_size=(3, 3), strides=1, padding='same',
                          kernel_regularizer=regularizers.l2(l=0.001))
        self.c_5 = Conv2D(filters=self.C, kernel_size=(5, 5), strides=1, padding='same',
                          kernel_regularizer=regularizers.l2(l=0.001))
        self.c_7 = Conv2D(filters=self.C, kernel_size=(7, 7), strides=1, padding='same',
                          kernel_regularizer=regularizers.l2(l=0.001))
        # 全局平均池化操作，对每个通道求平均值
        self.gap = GlobalAveragePooling2D()
        # 使用 dense 全连接层，目的是将前面提取的特征，在 dense 经过非线性变化，
提取这些特征之间的关联，最后映射到输出空间上
        self.dense_two = Dense(units=2, activation='relu')
        # 输出一个二维的激活函数为 relu 的全连接层
        self.dense_c1 = Dense(units=self.C)
        # 输出 21 维度的全连接层
        self.dense_c2 = Dense(units=self.C)
        self.dense_c3 = Dense(units=self.C)

    def get_config(self):
        config = super().get_config().copy()
        config.update({
            'C': self.C
        })
        return config

    def call(self, X):
```

```
        X_1 = self.c_3(X)
        X_2 = self.c_5(X)
        X_3 = self.c_7(X)

        X_dash = Add()([X_1, X_2, X_3])
        # 对三个卷积后的特征图相加融合
        v_gap = self.gap(X_dash)
        # 全局平均池化操作，得到通道对应权重
        v_gap = tf.reshape(v_gap, [-1, 1, 1, self.C])
        fc1 = self.dense_two(v_gap)
        # 对池化效果输出二维的全连接层
        alpha = self.dense_c1(fc1)
        beta = self.dense_c2(fc1)
        gamma = self.dense_c3(fc1)
        # 生成α、β、γ三个全连接层
        before_softmax = concatenate([alpha, beta, gamma], 1)
        after_softmax = softmax(before_softmax, axis=1)
        # 使用 softmax 激活函数
        a1 = after_softmax[:, 0, :, :]
        a1 = tf.reshape(a1, [-1, 1, 1, self.C])
        a2 = after_softmax[:, 1, :, :]
        a2 = tf.reshape(a2, [-1, 1, 1, self.C])
        a3 = after_softmax[:, 2, :, :]
        a3 = tf.reshape(a3, [-1, 1, 1, self.C])

        select_1 = Multiply()([X_1, a1])
        select_2 = Multiply()([X_2, a2])
        select_3 = Multiply()([X_3, a3])
        # 将核特征图与权重相乘得到输出
        out = Add()([select_1, select_2, select_3])

        return out


# 定义创建模型方法
def create_model():
    # ca_block = Channel Attention block
    # msfe_block = Multi scale feature extraction block
    # ksm = Kernel Selecting Module
    tf.keras.backend.clear_session()
    input = Input(shape=(256, 256, 3), name="input_layer")
    # 图像卷积预处理
    conv_block = Convolutional_block()(input)
    # 通道注意力机制获取通道权重
```

```
    ca_block = Channel_attention()(conv_block)
    # 用 3 个卷积核提取 64 维度的信息
    ca_block = Conv2D(filters=3, kernel_size=(3, 3), strides=1, padding='same')(ca_block)
    # 将输入图像与特征图像融合
    ca_block = Concatenate()([input, ca_block])
    # 多尺度特征融合层，使用平均池化和 Unet 获取原始、局部、全局感受野
    msfe_block = Multi_scale_feature_extraction()(ca_block)
    # 核选择模块，对多尺度特征使用三种卷积核处理后进行权重分配
    ksm = Kernel_selecting_module()(msfe_block)
    ksm = Conv2D(filters=3, kernel_size=(3, 3), strides=1, padding='same')(ksm)
    # 卷积输出与原图相同维度
    model = Model(inputs=[input], outputs=[ksm])
    return model


# 定义图像处理函数
def inference_single_image(model, noisy_image):
    # 增加维度
    input_image = np.expand_dims(noisy_image, axis=0)
    # 调用模型输出结果
    predicted_image = model.predict(input_image)
    return predicted_image[0]


# 清楚去噪缓存的切割图像
def clean():
    top = 'D:\\AIPicture\\OriginalPicture\\cut\\'
    for root, dirs, files in os.walk(top, topdown=False):
        for name in files:
            os.remove(os.path.join(root, name))
        for name in dirs:
            os.rmdir(os.path.join(root, name))


# 切割图像
def cut(pic_path):
    pic_target = 'D:\\AIPicture\\OriginalPicture\\cut\\'
    # 分割后的图片的文件夹，以及拼接后要保存的文件夹
    pic_target_out = 'D:\\AIPicture\\OriginalPicture\\'
    cut_high = 256
    cut_width = 256
    picture = cv2.imread(pic_path)
    (high, width, depth) = picture.shape
```

```
    # 预处理生成 0 矩阵
    pic = np.zeros((cut_high, cut_width, depth))
    # 计算可以划分的横纵的个数
    num_high = int(high / cut_high) + 1
    num_width = int(width / cut_width) + 1


    temp_pic = np.zeros((num_high * cut_high, cut_width * num_width, depth))


    temp_pic[0: high, 0: width, :] = picture

    # 生成一个填充图像以便完整切割
    # for 循环迭代生成
    for i in range(0, num_high):
        for j in range(0, num_width):
            pic = temp_pic[i * cut_high: (i + 1) * cut_high, j * cut_width: (j + 1) * cut_width, :]
            pic = inference_single_image(model, pic)
            # 调用模型处理
            result_path = pic_target + '{}_{}.jpg'.format(i + 1, j + 1)
            cv2.imwrite(result_path, pic)

    temp_pic1 = np.zeros((num_high * cut_high, cut_width * num_width, depth))

    picture_names = os.listdir(pic_target)
    if len(picture_names) == 0:
        print("没有文件")

    else:
        # 循环复制
        for i in range(1, num_high + 1):
            for j in range(1, num_width + 1):
                # 数组保存分割后图片的列数和行数，注意分割后图片的格式为 x_x.jpg，
x 从 1 开始
                img_part = cv2.imread(pic_target + '{}_{}.jpg'.format(i, j))
                temp_pic1[cut_high * (i - 1): cut_high * i, cut_width * (j - 1): cut_width * j, :]
= img_part
        # 保存图片，大功告成

    temp_pic2 = temp_pic1[0:high, 0:width, :]

    cv2.imwrite(pic_target_out + 'result.jpg', temp_pic2)
# 不全部占满显存，按需分配
```

```
#主函数
if __name__ =='__main__':
        config = tf.ConfigProto()
        config.gpu_options.allow_growth = True
        sess = tf.Session(config=config)
        tf.compat.v1.keras.backend.set_session(sess)    #  设置 session
        #  设置显存自动增长，不足时转入 CPU
        tf.keras.backend.set_image_data_format('channels_last')
#  设置返回的默认图像维度 channels_last：返回(256,256,3)

        best_models_path                                                                                  =
"D:\\AIPicture\\back\\src\\main\\java\\org\\app\\Python\\pridnet\\model\\best_PRIDNet.h5
"
#  模型参数路径
        custom_obj = {"Convolutional_block": Convolutional_block,
                      "Channel_attention": Channel_attention,
                      "Avg_pool_Unet_Upsample_msfe": Avg_pool_Unet_Upsample_msfe,
                      "Multi_scale_feature_extraction": Multi_scale_feature_extraction,
                      "Kernel_selecting_module": Kernel_selecting_module}
#  自定义模型各层

        model = load_model(best_models_path,
                             custom_objects=custom_obj, compile=False)
#  生成模型方法
        clean()
#  清除去噪分割文件
        img_path = "D:\\AIPicture\\OriginalPicture\\OriginalPicture.png"
#  输入图像地址
    cut(img_path)
#  调用函数切割并使用模型处理后拼接保存
```