```
import torch
import torch.nn.functional as F
import torch.nn as nn
from torch.autograd.variable import Variable
from torch.nn import init

from .densenet import *
from .resnet import *
from .vgg import *
from .crop_resize import CropResize

# from densenet import *
# from resnet import *
# from vgg import *

import numpy as np
import sys
thismodule = sys.modules[__name__]
# from .roi_module import RoIPooling2D
# import cupy as cp
import pdb

img_size = 256

dim_dict = {
    'resnet101': [512, 1024, 2048],
    'resnet152': [512, 1024, 2048],
    'resnet50': [512, 1024, 2048],
    'resnet34': [128, 256, 512],
    'resnet18': [128, 256, 512],
    'densenet121': [256, 512, 1024],
    'densenet161': [384, 1056, 2208],
    'densenet169': [64, 128, 256, 640, 1664],
    'densenet169_par': [64, 128, 256, 640, 1664],
    # 'densenet169': [256, 640, 1664],
    'densenet201': [256, 896, 1920],
    'vgg': [256, 512, 512]
}


def proc_vgg(model):
    # dilation
    model.features[2][-1].stride = 1
    model.features[2][-1].kernel_size = 1
    for m in model.features[3]:
        if isinstance(m, nn.Conv2d):
            m.dilation = (2, 2)
            m.padding = (2, 2)

    model.features[3][-1].stride = 1
    model.features[3][-1].kernel_size = 1
    for m in model.features[4]:
        if isinstance(m, nn.Conv2d):
            m.dilation = (4, 4)
            m.padding = (4, 4)
    model.features[4][-1].stride = 1
    model.features[4][-1].kernel_size = 1
    return model


```

```
def proc_densenet(model):
    # dilation
    def remove_sequential(all_layers, network):
        for layer in network.children():
            if isinstance(layer, nn.Sequential):  # if sequential layer, apply recursively to layers in sequential layer
                remove_sequential(all_layers, layer)
            if list(layer.children()) == []:  # if leaf node, add it to list
                all_layers.append(layer)
    model.features.transition2[-1].kernel_size = 1
    model.features.transition2[-1].stride = 1
    all_layers = []
    remove_sequential(all_layers, model.features.denseblock3)
    for m in all_layers:
        if isinstance(m, nn.Conv2d) and m.kernel_size==(3, 3):
            m.dilation = (2, 2)
            m.padding = (2, 2)

    model.features.transition3[-1].kernel_size = 1
    model.features.transition3[-1].stride = 1
    all_layers = []
    remove_sequential(all_layers, model.features.denseblock4)
    for m in all_layers:
        if isinstance(m, nn.Conv2d) and m.kernel_size==(3, 3):
            m.dilation = (4, 4)
            m.padding = (4, 4)
    return model


procs = {'vgg16_bn': proc_vgg,
         'densenet169': proc_densenet,
         }


def get_upsampling_weight(in_channels, out_channels, kernel_size):
    """Make a 2D bilinear kernel suitable for upsampling"""
    factor = (kernel_size + 1) // 2
    if kernel_size % 2 == 1:
        center = factor - 1
    else:
        center = factor - 0.5
    og = np.ogrid[:kernel_size, :kernel_size]
    filt = (1 - abs(og[0] - center) / factor) * \
           (1 - abs(og[1] - center) / factor)
    weight = np.zeros((in_channels, out_channels, kernel_size, kernel_size),
                      dtype=np.float64)
    weight[range(in_channels), range(out_channels), :, :] = filt
    return torch.from_numpy(weight).float()


class DeepLab(nn.Module):
    def __init__(self, pretrained=True, c_output=21, c_input=3, base='vgg16'):
        super(DeepLab, self).__init__()
        if 'vgg' in base:
            dims = dim_dict['vgg'][::-1]
        else:
            dims = dim_dict[base][::-1]
        # self.pred = nn.Conv2d(dims[0], c_output, kernel_size=3, dilation=8, padding=8)
        self.preds = nn.ModuleList([nn.Conv2d(dims[0], c_output, kernel_size=3, dilation=dl, padding=dl)
                                    for dl in [6, 12, 18, 24]])
        self.upscale = nn.ConvTranspose2d(c_output, c_output, 16, 8, 4)
        # self.upscale = nn.ConvTranspose2d(c_output, c_output, 8, 4, 2)
        for m in self.modules():
            if isinstance(m, nn.Conv2d) or isinstance(m, nn.Linear):
                m.weight.data.normal_(0.0, 0.01)
                m.bias.data.fill_(0)
            if isinstance(m, nn.ConvTranspose2d):
                assert m.kernel_size[0] == m.kernel_size[1]
                initial_weight = get_upsampling_weight(
                    m.in_channels, m.out_channels, m.kernel_size[0])
                m.weight.data.copy_(initial_weight)
        self.feature = getattr(thismodule, base)(pretrained=pretrained)
        self.feature = procs[base](self.feature)
        for m in self.modules():
            if isinstance(m, nn.BatchNorm2d):
                m.requires_grad=False

    def forward(self, x):
        x = self.feature(x)
        # x = self.pred(x)
        x = sum([f(x) for f in self.preds])
        x = self.upscale(x)
        return x, None, None

    def forward_mscale(self, xs):
        outputs = []
        for x in xs:
            x = self.feature(x)
            # x = self.pred(x)
            x = sum([f(x) for f in self.preds])
            x = self.upscale(x)
            outputs += [x]
        merge = torch.max(outputs[0], F.upsample(outputs[1], size=img_size, mode='bilinear'))
        merge = torch.max(merge, F.upsample(outputs[2], size=img_size, mode='bilinear'))
        outputs += [merge]
        return outputs



if __name__ == "__main__":
    fcn = WSFCN2(base='densenet169').cuda()
    x = torch.Tensor(2, 3, 256, 256).cuda()
    sb = fcn(Variable(x))

```