```python
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 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

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': [256, 640, 1664],
    'densenet201': [256, 896, 1920],
    'vgg': [256, 512, 512]
}


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 ParamPool(nn.Module):
    def __init__(self, input_c):
        super(ParamPool, self).__init__()
        self.conv = nn.Conv2d(input_c, 1, kernel_size=1, bias=False)
        # self.conv2 = nn.Conv2d(input_c, input_c, kernel_size=1,
bias=False)

```

```
59      def forward(self, x):
60          bsize, c, ssize, _ = x.shape
61          w = self.conv(x)
62          # x = self.conv2(x)
63          w = F.softmax(w.view(bsize, 1, -1), 2)
64          w = w.view(bsize, 1, ssize, ssize)
65          x = (x*w).sum(3).sum(2)
66          return x
67
68
69  # def proc_densenet(model):
70  #     def hook(module, input, output):
71  #         model.feats[output.device.index] += [output]
72  #     model.features.transition3[-2].register_forward_hook(hook)
73  #     model.features.transition2[-2].register_forward_hook(hook)
74  #
75  #     model.features.transition3[-1].kernel_size=1
76  #     model.features.transition3[-1].stride=1
77  #     for m in model.features.denseblock4:
78  #         if isinstance(m, nn.Conv2d) and m.kernel_size==(3, 3):
79  #             m.dilation = (2, 2)
80  #     return model
81
82
83  def proc_densenet(model):
84      def hook(module, input, output):
85          model.feats[output.device.index] += [output]
86      model.features.transition3[-2].register_forward_hook(hook)
87      model.features.transition2[-2].register_forward_hook(hook)
88      # dilation
89      def remove_sequential(all_layers, network):
90          for layer in network.children():
91              if isinstance(layer, nn.Sequential):  # if sequential layer,
    apply recursively to layers in sequential layer
92                  remove_sequential(all_layers, layer)
93              if list(layer.children()) == []:  # if leaf node, add it to
    list
94                  all_layers.append(layer)
95      model.features.transition3[-1].kernel_size = 1
96      model.features.transition3[-1].stride = 1
97      # all_layers = []
98      # remove_sequential(all_layers, model.features.denseblock4)
99      # for m in all_layers:
100     #     if isinstance(m, nn.Conv2d) and m.kernel_size==(3, 3):
101     #         m.dilation = (2, 2)
102     #         m.padding = (2, 2)
103     return model
104
105
106 def proc_resnet(model):
107     def hook(module, input, output):
108         model.feats[output.device.index] += [output]
109     model.layer3[-1].register_forward_hook(hook)
110     model.layer2[-1].register_forward_hook(hook)
111     def remove_sequential(all_layers, network):
112         for layer in network.children():
113             if isinstance(layer, nn.Sequential):  # if sequential layer,
    apply recursively to layers in sequential layer
114                 remove_sequential(all_layers, layer)
115             if list(layer.children()) == []:  # if leaf node, add it to
```

```
list
                all_layers.append(layer)
    all_layers = []
    remove_sequential(all_layers, model.layer4)
    for m in all_layers:
        if isinstance(m, nn.Conv2d) and m.stride==(2, 2):
            m.stride = (1, 1)

    return model


procs = {
    'densenet169': proc_densenet,
    'densenet201': proc_densenet,
    'resnet152': proc_resnet,
    'resnet101': proc_resnet,
}


class FCN(nn.Module):
    def __init__(self, pretrained=True, c_input=3, n_class=200,
base='vgg16'):
        super(FCN, self).__init__()
        if 'vgg' in base:
            dims = dim_dict['vgg'][::-1]
        else:
            dims = dim_dict[base][::-1]
        self.preds = nn.ModuleList([nn.Conv2d(d, 1, kernel_size=1) for d
in dims])
        self.upscales = nn.ModuleList([
            nn.ConvTranspose2d(1, 1, 4, 2, 1),
            nn.ConvTranspose2d(1, 1, 4, 2, 1),
            nn.ConvTranspose2d(1, 1, 16, 8, 4),
        ])
        self.reduce = nn.Conv2d(dims[0], 512, kernel_size=3, padding=1)
        self.param_pool = ParamPool(512)
        self.classifier = nn.Linear(512, n_class)
        for m in self.modules():
            if isinstance(m, nn.Conv2d) or isinstance(m, nn.Linear):
                m.weight.data.normal_(0.0, 0.02)
                if m.bias is not None:
                    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.feats = {}
        self.feature = procs[base](self.feature)
        for m in self.modules():
            if isinstance(m, nn.BatchNorm2d):
                m.requires_grad=False

    def forward(self, x, boxes=None, ids=None):
        self.feature.feats[x.device.index] = []
        x = self.feature(x)
        feats = self.feature.feats[x.device.index]
        feats += [x]
        feats = feats[::-1]
```

```
173         msk = self.preds[0](feats[0])
174         big_msk = msk
175         msk = F.sigmoid(msk)
176         msk_feat = self.reduce(feats[0])*msk
177         # msk_feat = F.avg_pool2d(msk_feat, 16).squeeze(3).squeeze(2)
178         msk_feat = self.param_pool(msk_feat)
179         cls = self.classifier(msk_feat)
180         big_msk = F.upsample_bilinear(big_msk, scale_factor=16)
181         return big_msk, msk, cls
182
183
184
185 if __name__ == "__main__":
186     fcn = WSFCN2(base='densenet169').cuda()
187     x = torch.Tensor(2, 3, 256, 256).cuda()
188     sb = fcn(Variable(x))
189
```