```python
import torch
import torch.nn as nn
import torchvision.models as models
from torch.nn.utils.rnn import pack_padded_sequence
import torch.nn.functional as F
import numpy as np
import pdb
from .densenet import *
from .resnet import *
from .vgg import *
import sys
thismodule = sys.modules[__name__]


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)

    def forward(self, x):
        bsize, c, ssize, _ = x.shape
        w = self.conv(x)
        w = F.softmax(w.view(bsize, 1, -1), 2)
        w = w.view(bsize, 1, ssize, ssize)
        x = (x*w).sum(3).sum(2)
        return x


```

```
60 def proc_densenet(model):
61     def hook(module, input, output):
62         model.feats[output.device.index] += [output]
63     model.features.transition3[-2].register_forward_hook(hook)
64     model.features.transition2[-2].register_forward_hook(hook)
65     # dilation
66     def remove_sequential(all_layers, network):
67         for layer in network.children():
68             if isinstance(layer, nn.Sequential):  # if sequential layer,
   apply recursively to layers in sequential layer
69                 remove_sequential(all_layers, layer)
70             if list(layer.children()) == []:  # if leaf node, add it to
   list
71                 all_layers.append(layer)
72     model.features.transition3[-1].kernel_size = 1
73     model.features.transition3[-1].stride = 1
74     all_layers = []
75     remove_sequential(all_layers, model.features.denseblock4)
76     for m in all_layers:
77         if isinstance(m, nn.Conv2d) and m.kernel_size==(3, 3):
78             m.dilation = (2, 2)
79             m.padding = (2, 2)
80     return model
81
82
83 procs = {
84     'densenet169': proc_densenet,
85     'densenet201': proc_densenet,
86          }
87
88
89 class EncoderCNN(nn.Module):
90     def __init__(self, embed_size, patt_size=512, base='densenet169'):
91         """Load the pretrained ResNet-152 and replace top fc layer."""
92         super(EncoderCNN, self).__init__()
93         if 'vgg' in base:
94             dims = dim_dict['vgg'][::-1]
95         else:
96             dims = dim_dict[base][::-1]
97         self.preds = nn.ModuleList([nn.Conv2d(d, 1, kernel_size=1) for d
   in dims])
98         self.upscales = nn.ModuleList([
99             nn.ConvTranspose2d(1, 1, 4, 2, 1),
100            nn.ConvTranspose2d(1, 1, 4, 2, 1),
101            nn.ConvTranspose2d(1, 1, 16, 8, 4),
102        ])
103        self.linear = nn.Linear(patt_size, embed_size)
104        self.reduce = nn.Conv2d(dims[0], patt_size, kernel_size=3,
   padding=1)
105        self.param_pool = ParamPool(patt_size)
106        for m in self.modules():
107            if isinstance(m, nn.Conv2d) or isinstance(m, nn.Linear):
108                m.weight.data.normal_(0.0, 0.02)
109                if m.bias is not None:
110                    m.bias.data.fill_(0)
111            if isinstance(m, nn.ConvTranspose2d):
112                assert m.kernel_size[0] == m.kernel_size[1]
113                initial_weight = get_upsampling_weight(
114                    m.in_channels, m.out_channels, m.kernel_size[0])
115                m.weight.data.copy_(initial_weight)
```

```
116
117         self.feature = getattr(thismodule, base)(pretrained=True)
118         self.feature.feats = {}
119         self.feature = procs[base](self.feature)
120         for m in self.feature.modules():
121             if isinstance(m, nn.BatchNorm2d):
122                 m.requires_grad=False
123
124     def forward(self, x):
125         """Extract feature vectors from input images."""
126         # pdb.set_trace()
127         # with torch.no_grad():
128         #     features = self.resnet(images)
129         # features = features.reshape(features.size(0), -1)
130         # features = self.bn(self.linear(features))
131         # return features
132         self.feature.feats[x.device.index] = []
133         x = self.feature(x)
134         feats = self.feature.feats[x.device.index]
135         feats += [x]
136         feats = feats[::-1]
137         msk = self.preds[0](feats[0])
138         big_msk = msk
139         # big_msk = self.upscales[0](msk)
140         # big_msk = F.upsample(msk, scale_factor=16)
141         msk = F.sigmoid(msk)
142         msk_feat = self.reduce(feats[0])*msk
143         # msk_feat = F.avg_pool2d(msk_feat, 16).squeeze(3).squeeze(2)
144         msk_feat = self.param_pool(msk_feat)
145         msk_feat = self.linear(msk_feat)
146         big_msk = F.upsample_bilinear(big_msk, scale_factor=16)
147         if self.training:
148             return big_msk, msk, msk_feat
149         else:
150             return big_msk
151
152
153 class DecoderRNN(nn.Module):
154     def __init__(self, embed_size, hidden_size, vocab_size, num_layers,
    max_seq_length=20):
155         """Set the hyper-parameters and build the layers."""
156         super(DecoderRNN, self).__init__()
157         self.embed = nn.Embedding(vocab_size, embed_size)
158         self.lstm = nn.LSTM(embed_size, hidden_size, num_layers,
    batch_first=True)
159         self.linear = nn.Linear(hidden_size, vocab_size)
160         self.max_seg_length = max_seq_length
161
162     def forward(self, features, captions, lengths):
163         """Decode image feature vectors and generates captions."""
164         embeddings = self.embed(captions)
165         embeddings = torch.cat((features.unsqueeze(1), embeddings), 1)
166         packed = pack_padded_sequence(embeddings, lengths,
    batch_first=True)
167         hiddens, _ = self.lstm(packed)
168         outputs = self.linear(hiddens[0])
169         return outputs
170
171     def sample(self, features, states=None):
172         """Generate captions for given image features using greedy
```

```
    search."""
173         sampled_ids = []
174         inputs = features.unsqueeze(1)
175         for i in range(self.max_seg_length):
176             hiddens, states = self.lstm(inputs, states)          #
    hiddens: (batch_size, 1, hidden_size)
177             outputs = self.linear(hiddens.squeeze(1))            #
    outputs:  (batch_size, vocab_size)
178             _, predicted = outputs.max(1)                        #
    predicted: (batch_size)
179             sampled_ids.append(predicted)
180             inputs = self.embed(predicted)                       #
    inputs: (batch_size, embed_size)
181             inputs = inputs.unsqueeze(1)                         #
    inputs: (batch_size, 1, embed_size)
182         sampled_ids = torch.stack(sampled_ids, 1)                #
    sampled_ids: (batch_size, max_seq_length)
183         return sampled_ids
184
185
186 class EncDec(nn.Module):
187     def __init__(self, vocab_size, embed_size=256, hidden_size=512,
    num_layers=1, max_seq_length=20):
188         super(EncDec, self).__init__()
189         self.encoder = EncoderCNN(embed_size)
190         self.decoder = DecoderRNN(embed_size, hidden_size, vocab_size,
    num_layers, max_seq_length)
191
192     def forward(self, images, captions=None, lengths=None):
193         if self.training:
194             if captions is not None:
195                 big_msk, msk, msk_feat = self.encoder(images)
196                 outputs = self.decoder(msk_feat, captions, lengths)
197                 return big_msk, msk, outputs
198             else:
199                 big_msk, msk, msk_feat = self.encoder(images)
200                 return big_msk, msk
201         else:
202             return self.encoder(images)
203
204
```