

## 带有注意力机制的ResNet-101和LSTM图像描述模型



运行

更新于 2020-10-27 10:54:28

4

1

图像描述技术，是生成连贯流畅的语句，描述图像内容。对互联网中图像信息的检索、儿童的早期教育，与视障人士的生活辅助等方面都有重要的意义。因此，图像描述领域受到越来越多的关注。



a bird perched on a bird feeder



a fire hydrant in front of a building



a giraffe laying down on a dirt ground



a cat sitting on top of a wooden table

本案例采用的网络结构为：带有注意力机制的编码器-解码器结构，生成的语句能够较为准确地描述图像。

## 目录

[1. 数据集简介](#)[2. 模型介绍](#)[3. 模型结构](#)[3.1 编码器](#)[3.2 注意力机制](#)[3.3 解码器](#)[4. 构建模型](#)[5. 图像描述生成](#)[6. 总结](#)

## 1 数据集简介

本案例采用的模型，已经在MSCOCO 2014数据集上进行了预训练。MSCOCO 2014的训练集包含82783张图像，大小为13.5GB；验证集包含40504张图像，大小为6.6GB。

这一数据集收集并标注复杂的日常场景图像，常用于训练图像描述的模型。图像标注出80类物体，如：人、车辆类、动物类等，同时包含对图像的文本描述。

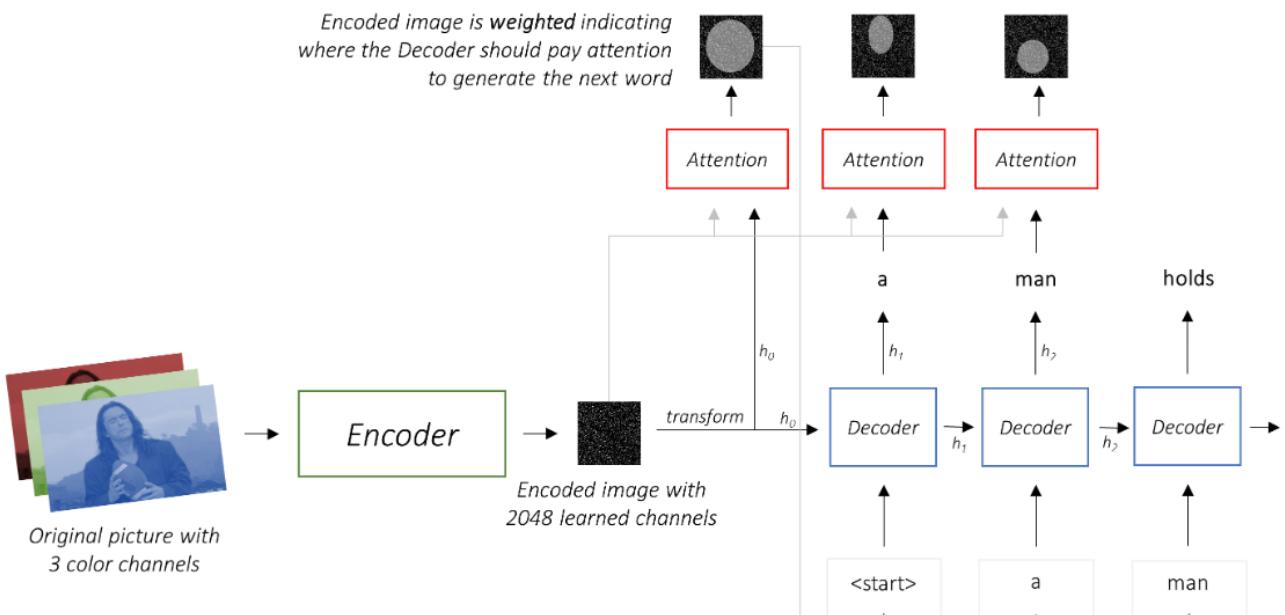


## 2 模型介绍

带有注意力机制的编码器-解码器结构：编码器能够将输入图像转为向量；解码器能够将向量转化为语句，描述输入图像。

引入注意力机制，解码器能够在逐字生成语句时，把注意力集中于，图像中与当前词最相关的部分。

下图展示了该网络结构的思路：





### 3 模型结构

接下来我们分别构建编码器和解码器网络。

#### 3.1 编码器

首先构建编码器（Encoder），本案例中的编码器为在MSCOCO 2014数据集上进行预训练的ResNet-101网络。

先加载需要使用的库：

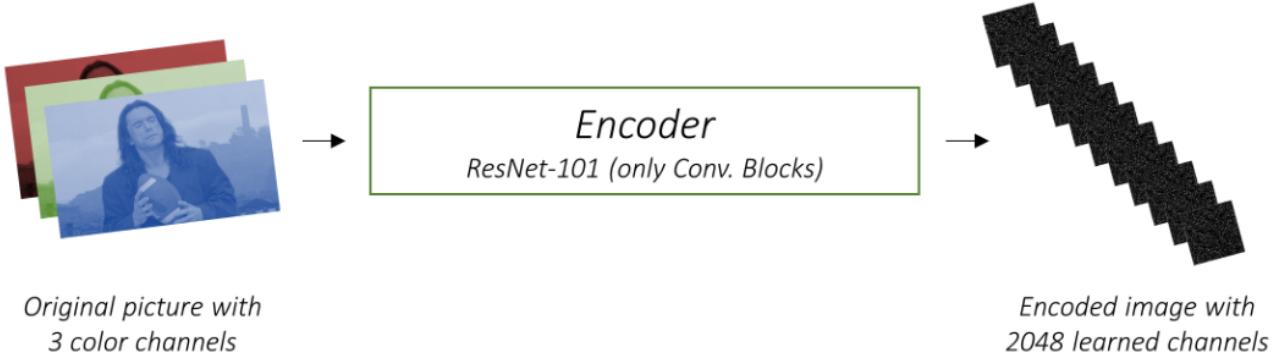
```
cp models.py ../
```

```
import models
import torch
from torch import nn
import torchvision
import torch.nn.functional as F
import numpy as np
import json
import os
import torchvision.transforms as transforms
import matplotlib.pyplot as plt
import matplotlib.cm as cm
import skimage.transform
from skimage import img_as_ubyte
from skimage.transform import resize
from imageio import imread
from PIL import Image

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
```

编码器使用预训练的模型，可以节省训练时间。预训练的ResNet-101结构中，最后一层池化层和全连接层用于图像分类，而编码过程不需要进行图像分类，因此我们删除这两层。

增加自适应池化层，将特征向量调整为一致大小，使模型可以接收任意像素大小的输入图像。此外，在网络结构内，添加对ResNet-101网络2-4层卷积层进行微调的模块。



*Original picture with  
3 color channels*

*Encoded image with  
2048 learned channels*

```
# 编码器
class Encoder(nn.Module):
    def __init__(self, encoded_image_size=14): # encoded_image_size: 生成的特征图的大小
        super(Encoder, self).__init__()
        self.enc_image_size = encoded_image_size

    # 加载预训练的ResNet-101模型
    resnet = torchvision.models.resnet101(pretrained=True)

    # 删去最后一层池化层和全连接层（该层用于分类）
    modules = list(resnet.children())[:-2]
    self.resnet = nn.Sequential(*modules)

    # 调整特征向量大小
    self.adaptive_pool = nn.AdaptiveAvgPool2d((encoded_image_size, encoded_image_size))

    self.fine_tune()

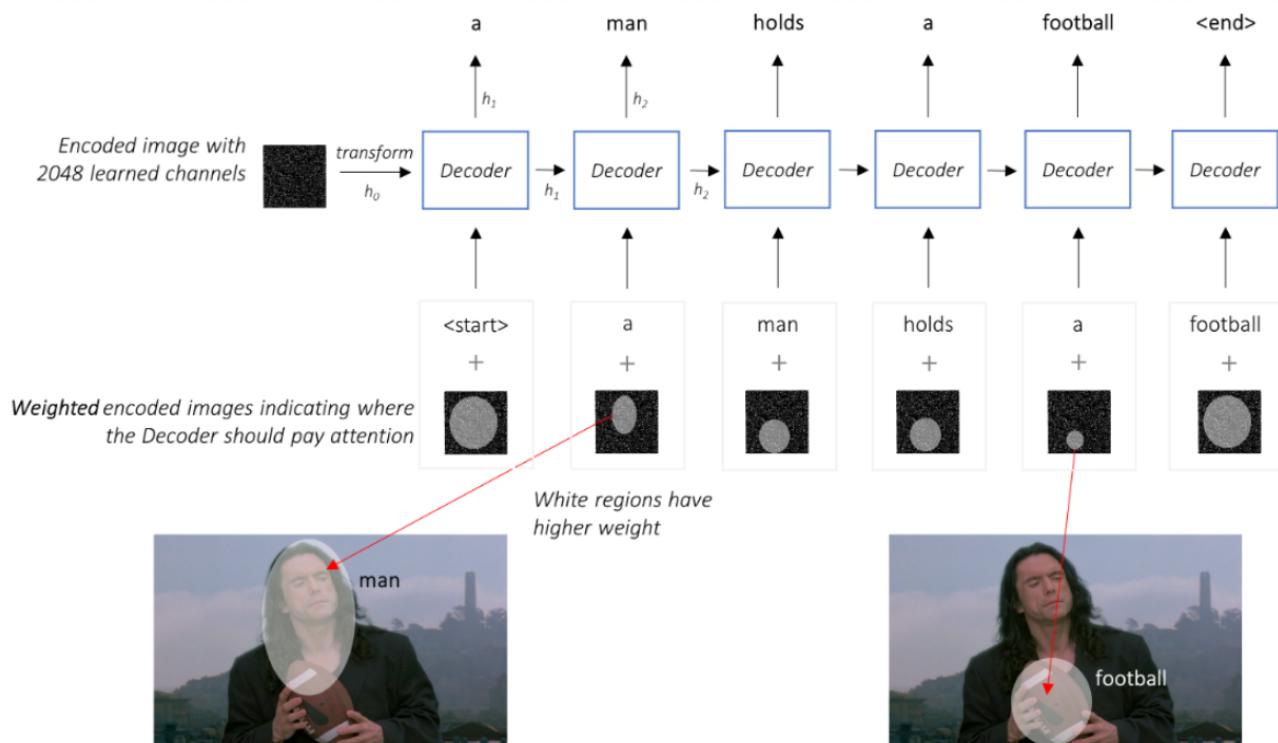
    # 前向传播 传入图片，提取特征
    def forward(self, images):
        # 输出特征大小: (2048, image_size/32, image_size/32)
        out = self.resnet(images)
        # 转化为一致大小: (2048, encoded_image_size, encoded_image_size)
        out = self.adaptive_pool(out)
        # 调换位置 (encoded_image_size, encoded_image_size, 2048)
        out = out.permute(0, 2, 3, 1)

        # 输出图像经过编码后，得到的向量
        return out

    # 微调模型
    def fine_tune(self, fine_tune=True):
        for p in self.resnet.parameters():
            p.requires_grad = False
        # 对2-4层卷积层进行微调
        for c in list(self.resnet.children())[5:]:
            for p in c.parameters(): # 微调某层参数
                p.requires_grad = fine_tune
```

## 3.2 注意力机制

在构建解码器前，需要先定义注意力机制（Attention）。注意力机制返回加权的特征向量。利用加权的特征向量，与上一个预测的单词，结合起来预测下一个单词，解码器能够生成更加准确的语句。下图展示了该机制的结构：



## # 注意力机制

```
class Attention(nn.Module):
    def __init__(self, encoder_dim, decoder_dim, attention_dim):
        # encoder_dim: 编码图像的特征维度；decoder_dim: decoder维度；attention_dim: 注意力机制层数
        super(Attention, self).__init__()
        # 两个注意力机制模块，分别针对编码和解码
        self.encoder_att = nn.Linear(encoder_dim, attention_dim) # 全连接层转换编码的特征向量
        self.decoder_att = nn.Linear(decoder_dim, attention_dim) # 全连接层转换解码器的输出
        self.full_att = nn.Linear(attention_dim, 1)
        self.relu = nn.ReLU()
        self.softmax = nn.Softmax(dim=1) # softmax层计算权重

    # 前向传播
    def forward(self, encoder_out, decoder_hidden):
        att1 = self.encoder_att(encoder_out) # 将图像特征传入att1
        att2 = self.decoder_att(decoder_hidden) # batch_size * attention_dim
        att = self.full_att(self.relu(att1 + att2.unsqueeze(1))).squeeze(2) # 每个像素的权重
        alpha = self.softmax(att) # softmax计算权重
        attention_weighted_encoding = (encoder_out * alpha.unsqueeze(2)).sum(dim=1)

    # 反向每个时刻加权的图像特征向量
```

# 必归母子的类加权的函数在这里  
return attention\_weighted\_encoding, alpha

### 3.3 解码器

最后构建带有注意力机制的解码器（Decoder）。本案例采用LSTM作为解码器，逐字生成语句。因为注意力机制，解码器在生成不同单词的时候，会关注图像不同部分。比如，生成“a man holds a football”中的“football”时，解码器会关注图像中的“足球”。

解码器保留LSTM得到的中间输出语句，并在生成新的单词时，先用注意力机制，得到特征向量的权重。通过之前预测的语句和新的权重，共同预测下一个单词。

本案例中，带有注意力机制的LSTM网络结构包含：注意力机制、嵌入层、dropout层、通过前向传播解码。

```
class DecoderWithAttention(nn.Module):
    def __init__(self, attention_dim, embed_dim, decoder_dim, vocab_size, encoder_dim=2048, dropout=0.5):
        # attention_dim: 注意力机制的层数; embed_dim: 嵌入层; decoder_dim: 解码器维度
        # vocab_size: 单词表 word map; encoder_dim: 编码器维度
        super(DecoderWithAttention, self).__init__()

        self.encoder_dim = encoder_dim
        self.attention_dim = attention_dim
        self.embed_dim = embed_dim
        self.decoder_dim = decoder_dim
        self.vocab_size = vocab_size
        self.dropout = dropout

        self.attention = Attention(encoder_dim, decoder_dim, attention_dim) # 注意力机制

        self.embedding = nn.Embedding(vocab_size, embed_dim) # 嵌入层 数据降维、转换为稠密向量
        self.dropout = nn.Dropout(p=self.dropout)
        self.decode_step = nn.LSTMCell(embed_dim + encoder_dim, decoder_dim, bias=True)

        # LSTM细胞
        self.init_h = nn.Linear(encoder_dim, decoder_dim) # 初始隐藏状态
        self.init_c = nn.Linear(encoder_dim, decoder_dim) # 初始细胞状态
        self.f_beta = nn.Linear(decoder_dim, encoder_dim)
        self.sigmoid = nn.Sigmoid() # sigmoid型激活门
        self.fc = nn.Linear(decoder_dim, vocab_size) # 为单词打分
        self.init_weights() # 初始化权重为均匀分布

    # 将权重初始化为均匀分布
    def init_weights(self):
```

```

    self.embedding.weight.data.uniform_(-0.1, 0.1)
    self.fc.bias.data.fill_(0)
    self.fc.weight.data.uniform_(-0.1, 0.1)

# 加载预训练的嵌入层
def load_pretrained_embeddings(self, embeddings):
    self.embedding.weight = nn.Parameter(embeddings)

# 微调嵌入层
def fine_tune_embeddings(self, fine_tune=True):
    for p in self.embedding.parameters():
        p.requires_grad = fine_tune

# 隐藏层初始状态
def init_hidden_state(self, encoder_out):
    mean_encoder_out = encoder_out.mean(dim=1) # 对第二维求平均 如果已经展开 相当于求每层所有像素的平均值
    h = self.init_h(mean_encoder_out) # 大小为 (batch_size, decoder_dim)
    c = self.init_c(mean_encoder_out)
    return h, c

# 前向传播
def forward(self, encoder_out, encoded_captions, caption_lengths):
    batch_size = encoder_out.size(0)
    encoder_dim = encoder_out.size(-1)
    vocab_size = self.vocab_size

    # 编码图像
    encoder_out = encoder_out.view(batch_size, -1, encoder_dim) # 输出向量: (batch_size, num_pixels, encoder_dim)
    num_pixels = encoder_out.size(1)

    # 对句子长度进行排序
    caption_lengths, sort_ind = caption_lengths.squeeze(1).sort(dim=0, descending=True)
    encoder_out = encoder_out[sort_ind]
    encoded_captions = encoded_captions[sort_ind]

    # 嵌入层 将每个词用512维的向量表示
    embeddings = self.embedding(encoded_captions) # 嵌入层
    h, c = self.init_hidden_state(encoder_out) # LSTM初始状态
    decode_lengths = (caption_lengths - 1).tolist() # 在<end>部分停止解码

    # max(decode_lengths): 最长词语数量; vocab_size: 单词表
    predictions = torch.zeros(batch_size, max(decode_lengths), vocab_size)
    .to(device)
    # num_pixels: 像素总数
    alphas = torch.zeros(batch_size, max(decode_lengths), num_pixels).to(device)

    # 用之前的单词和注意力机制得到的权重, 解码新的单词
    for t in range(max(decode_lengths)): # max(decode_lengths): 句子最长
        length
        batch_size_t = sum([l > t for l in decode_lengths])
        attention_weighted_encoding, alpha = self.attention(encoder_out[:b

```

```

batch_size_t], h[:batch_size_t]) # 每次循环都重新生成注意力权重
    gate = self.sigmoid(self.f_beta(h[:batch_size_t])) # (batch_size_t, encoder_dim)
        attention_weighted_encoding = gate * attention_weighted_encoding # 每层的注意力权重
    h, c = self.decode_step(torch.cat([embeddings[:batch_size_t, t, :], attention_weighted_encoding], dim=1),
                           (h[:batch_size_t], c[:batch_size_t])) # (batch_size_t, decode_r_dim)
    preds = self.fc(self.dropout(h)) # (batch_size_t, vocab_size)
    predictions[:batch_size_t, t, :] = preds # 生成预测
    alphas[:batch_size_t, t, :] = alpha # 每个像素的重点区域

    return predictions, encoded_captions, decode_lengths, alphas, sort_in
d

```

## 4 构建模型

构建好上述编码器、解码器、注意力机制后，需要定义实现图像描述的函数

`caption_image_beam_search`，该函数连接编码器、注意力机制、解码器。即将图像传入编码器后，计算得到特征向量；将特征向量传入带有注意力机制的解码器，通过LSTM算法得到单词表内生成语句对应的数字。

在搜索最优生成语句时，我们使用beam search（束搜索）算法，即每次挑选出所有生成语句中条件概率最大的k个，作为该时间步长下的候选输出序列。始终保持k个候选语句，最后从k个候选中挑出最优的生成语句。

```

def caption_image_beam_search(encoder, decoder, image_path, word_map, beam_size=3):
    k = beam_size # 每个解码步骤中，选择概率最大的k个词
    vocab_size = len(word_map) # 单词表个数

    # 读取图像
    img = imread(image_path)
    if len(img.shape) == 2:
        img = img[:, :, np.newaxis]
        img = np.concatenate([img, img, img], axis=2)
    img = img_as_ubyte(resize(img, (256, 256)))
    img = img.transpose(2, 0, 1)
    img = img / 255.
    img = torch.FloatTensor(img).to(device)
    normalize = transforms.Normalize(mean=[0.485, 0.456, 0.406],
                                    std=[0.229, 0.224, 0.225])
    transform = transforms.Compose([normalize])
    image = transform(img) # (3, 256, 256)

    # 编码器
    image = image.unsqueeze(0) # (1, 3, 256, 256)

```

```

encoder_out = encoder(image) # (1, enc_image_size, enc_image_size, encoder_dim)
enc_image_size = encoder_out.size(1)
encoder_dim = encoder_out.size(3)

# 编码得到的特征向量
encoder_out = encoder_out.view(1, -1, encoder_dim) # (1, num_pixels, encoder_dim)
num_pixels = encoder_out.size(1)

# batch size = k
encoder_out = encoder_out.expand(k, num_pixels, encoder_dim) # (k, num_pixels, encoder_dim)

# 每次储存前k个单词，初始状态为<start>
k_prev_words = torch.LongTensor([[word_map['<start>']]]) * k).to(device) # (k, 1)
seqs = k_prev_words # (k, 1)
top_k_scores = torch.zeros(k, 1).to(device) # (k, 1)

# 初始权重
seqs_alpha = torch.ones(k, 1, enc_image_size, enc_image_size).to(device)
complete_seqs = []
complete_seqs_alpha = []
complete_seqs_scores = []

# 解码
step = 1
h, c = decoder.init_hidden_state(encoder_out)

# s <= k, 到达<end>时停止解码
while True:
    embeddings = decoder.embedding(k_prev_words).squeeze(1) # 维度: (s, embed_dim)
    awe, alpha = decoder.attention(encoder_out, h) # (s, encoder_dim), (s, num_pixels)
    alpha = alpha.view(-1, enc_image_size, enc_image_size) # (s, enc_image_size, enc_image_size)
    gate = decoder.sigmoid(decoder.f_beta(h)) # gating scalar, (s, encoder_dim)
    awe = gate * awe

    h, c = decoder.decode_step(torch.cat([embeddings, awe], dim=1), (h, c))
    # (s, decoder_dim)
    scores = decoder.fc(h) # (s, vocab_size)
    scores = F.log_softmax(scores, dim=1)
    scores = top_k_scores.expand_as(scores) + scores # (s, vocab_size)

    # 初始状态下，分数相同
    if step == 1:
        top_k_scores, top_k_words = scores[0].topk(k, 0, True, True) #
        (s)
    else:
        # 找到最高得分的单词
        top_k_scores, top_k_words = scores.view(-1).topk(k, 0, True, True)

```

```

) # (s)

    prev_word_inds = top_k_words // vocab_size # (s)
    next_word_inds = top_k_words % vocab_size # (s)

    # 输出新单词
    seqs = torch.cat([seqs[prev_word_inds], next_word_inds.unsqueeze(1)],
dim=1) # (s, step+1)
    seqs_alpha = torch.cat([seqs_alpha[prev_word_inds], alpha[prev_word_in
ds].unsqueeze(1)],
dim=1) # (s, step+1, enc_image_size, enc_image
_size)

    # 判断是否可以停止解码
    incomplete_inds = [ind for ind, next_word in enumerate(next_word_inds
) if
        next_word != word_map['<end>']]
    complete_inds = list(set(range(len(next_word_inds))) - set(incomplete_
inds))

    # 完整语句
    if len(complete_inds) > 0:
        complete_seqs.extend(seqs[complete_inds].tolist())
        complete_seqs_alpha.extend(seqs_alpha[complete_inds].tolist())
        complete_seqs_scores.extend(top_k_scores[complete_inds])
        k -= len(complete_inds)

    # 处理不完整的语句
    if k == 0:
        break
    seqs = seqs[incomplete_inds]
    seqs_alpha = seqs_alpha[incomplete_inds]
    h = h[prev_word_inds[incomplete_inds]]
    c = c[prev_word_inds[incomplete_inds]]
    encoder_out = encoder_out[prev_word_inds[incomplete_inds]]
    top_k_scores = top_k_scores[incomplete_inds].unsqueeze(1)
    k_prev_words = next_word_inds[incomplete_inds].unsqueeze(1)

    # 步骤过长则终止
    if step > 50:
        break
    step += 1
    i = complete_seqs_scores.index(max(complete_seqs_scores))
    seq = complete_seqs[i]
    alphas = complete_seqs_alpha[i]

return seq, alphas

```

为了更好地展示输出结果，我们定义 `visualize_att` 函数，将图像注意力集中的位置和对应的输出单词，直观展示出来。

```
def visualize_att(image_path, seq, alphas, rev_word_map, smooth=True):
```

```

image = Image.open(image_path)      # 读入图像
image = image.resize([14 * 24, 14 * 24], Image.LANCZOS) # 调整图像大小
words = [rev_word_map[ind] for ind in seq]
plt.figure(figsize=(18, 9))

for t in range(len(words)):
    if t > 50:
        break
    plt.subplot(np.ceil(len(words) / 5.), 5, t + 1)

    # 将注意力集中的图像部分高亮
    plt.text(0, 1, '%s' % (words[t]), color='black', backgroundcolor='white', fontsize=12)
    plt.imshow(image)
    current_alpha = alphas[t, :]
    if smooth:
        alpha = skimage.transform.pyramid_expand(current_alpha.numpy(), upscale=24, sigma=8)
    else:
        alpha = skimage.transform.resize(current_alpha.numpy(), [14 * 24, 14 * 24])
    if t == 0:
        plt.imshow(alpha, alpha=0)
    else:
        plt.imshow(alpha, alpha=0.8)
    plt.set_cmap(cm.Greys_r)
    plt.axis('off')
plt.show()

```

## 5 图像描述生成

现在我们可以通过调用上述定义的函数，对输入的图像，生成对应的语句描述图像。考虑到训练时间过长，本案例加载训练好的编码器与解码器。

```

from glob import glob

checkpoint = '/content/BEST_checkpoint_coco_5_cap_per_img_5_min_word_freq.pth.tar' # 预训练参数
word_map_file = '/content/WORDMAP_coco_5_cap_per_img_5_min_word_freq.json' # 单词表
beam_size = 5
smooth = True

# 加载模型
checkpoint = torch.load(checkpoint, map_location=device)
decoder = checkpoint['decoder']
decoder = decoder.to(device)
decoder.eval()
encoder = checkpoint['encoder']
encoder = encoder.to(device)
encoder.eval()

```

加载单词表，即每个数字对应的单词。

```
# 加载单词表
with open(word_map_file, 'r') as j:
    word_map = json.load(j)
# 转换格式
rev_word_map = {v: k for k, v in word_map.items()}
# 展示单词表
for i in range(5):
    t = random.choice(list(rev_word_map))
    print(t, ":", rev_word_map[t])
```

```
333 : chairs
505 : plant
1575 : snowboarder
1859 : indian
3049 : build
```

接下来，我们可以向模型输入一张图像，观察模型运行结果。

```
Image.open("/content/surf.png") # 打开图片
```



下图展示了输出结果。通过可视化注意力机制，我们看到在生成"people"这一单词时，模型关注图像中的“两个人”；生成"surfboards"时，则关注人们手中的“冲浪板”。

生成的语句为：“海里有两个人拿着冲浪板”，语句通顺，且准确地描述出了图像的内容，包含了“冲浪板”、“两个人”、“海”这些关键词语。

```

my_files = np.array(glob("/content/surf.png"))
for img in my_files:

    # 生成图像描述
    seq, alphas = caption_image_beam_search(encoder, decoder, img, word_map, beam_size)
    alphas = torch.FloatTensor(alphas)

    for i in seq:
        print(rev_word_map[i], end=' ')
    # 展示图像描述结果
    visualize_att(img, seq, alphas, rev_word_map, smooth)

```

<start> a couple of people with surfboards in the water <end>



## 6 总结

编码器-解码器结构的模型得到广泛应用，本文分析了编码器、注意力机制、解码器的结构，详细介绍了代码编写思路，构建了带有注意力机制的ResNet-101和LSTM的模型，实现图像描述。其中ResNet-101模型已经在MSCOCO 2014上进行了预训练，节省了模型训练的时间。在寻找最优语句时，使用beam search算法，实现搜索最优语句的任务。

观察生成结果，可知模型的训练结果较好，生成的语句能够准确地描述图像内容。

发表您的讨论

© 2018 CookData 京ICP备1705652

竞赛

关于

产品

服务

帮助

联系

3号-1 (<http://www.beian.miit.gov.cn/>)

平台

[我们 \(/](#)

[介绍 \(/](#)

[条款 \(/](#)

[中心 \(/](#)

[我们 \(/](#)

(<http://>

foote

foote

foote

foote

foote

cookd

r?sub

r?sub

r?sub

r?sub

r?sub

ata.c

=0)

=0)

=1)

=2)

=3)

n/com

petitio

n/)