In [12]:

```
%cd /home/mw/project/
```

```
/home/mw/project
```

In [13]:

```
import torch
import torch.nn.functional as F
import torchvision.transforms as transforms
import torch.backends.cudnn as cudnn
import numpy as np
from nltk.translate.bleu_score import corpus_bleu
from tqdm import tqdm
from datasets import *
from utils import *
```

In [14]:

```
# Parameters
data_folder = '/home/mw/work/project/coco2014'  # folder with data files saved
data_name = 'coco_5_cap_per_img_5_min_word_freq'  # base name shared by data fi
checkpoint = '/home/mw/project/BEST_checkpoint_coco_5_cap_per_img_5_min_word_fr
word_map_file = '/home/mw/work/project/coco2014/WORDMAP_coco_5_cap_per_img_5_mi
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
cudnn.benchmark = True
beam_size = 1
attention = True
```

In [15]:

```python
# Load model
checkpoint = torch.load(checkpoint)
encoder = checkpoint['encoder']
encoder = encoder.to(device)
encoder.eval()
decoder = checkpoint['decoder']
decoder = decoder.to(device)
decoder.eval()
```

```
DecoderWithAttention(
  (attention): Attention(
    (encoder_att): Linear(in_features=2048, out_features=512, bias=True)
    (decoder_att): Linear(in_features=512, out_features=512, bias=True)
    (att): Linear(in_features=512, out_features=1, bias=True)
    (relu): ReLU()
    (softmax): Softmax(dim=1)
  )
  (embedding): Embedding(9490, 512)
  (decode_step): LSTMCell(2560, 512)
  (init_h): Linear(in_features=2048, out_features=512, bias=True)
  (init_c): Linear(in_features=2048, out_features=512, bias=True)
  (beta): Linear(in_features=512, out_features=1, bias=True)
  (fc): Linear(in_features=512, out_features=9490, bias=True)
  (dropout_layer): Dropout(p=0.5, inplace=False)
  (bn): BatchNorm1d(512, eps=1e-05, momentum=0.01, affine=True, track_running_st
)
```

In [16]:

```python
# Load word map (word2ix)
with open(word_map_file, 'r') as j:
    word_map = json.load(j)
rev_word_map = {v: k for k, v in word_map.items()}
vocab_size = len(word_map)
```

In [17]:

```python
# Normalization transform
normalize = transforms.Normalize(mean=[0.485, 0.456, 0.406],
                                 std=[0.229, 0.224, 0.225])
```

In [18]:

```python
# DataLoader
loader = torch.utils.data.DataLoader(
    CaptionDataset(data_folder, data_name, 'TEST', transform=transforms.Compose
    batch_size=1, shuffle=False, num_workers=1, pin_memory=True)
```

In [19]:

```
# Lists to store references (true captions), and hypothesis (prediction) for ea
# If for n images, we have n hypotheses, and references a, b, c... for each ima
# references = [[ref1a, ref1b, ref1c], [ref2a, ref2b], ...], hypotheses = [hyp1
references = list()
hypotheses = list()
```

In [20]:

```
# For each image
for i, (image, caps, caplens, allcaps) in enumerate(tqdm(loader, desc="EVALUATI
    k = beam_size

    # Tensor to store top k previous words at each step; now they're just <star
    k_prev_words = torch.LongTensor([[word_map['<start>']]] * k).to(device)   #

    # Tensor to store top k sequences; now they're just <start>
    seqs = k_prev_words  # (k, 1)

    # Tensor to store top k sequences' scores; now they're just 0
    top_k_scores = torch.zeros(k, 1).to(device)  # (k, 1)

    # Lists to store completed sequences and scores
    complete_seqs = list()
    complete_seqs_scores = list()

    # Move to GPU device, if available
    image = image.to(device)  # (1, 3, 256, 256)

    # Encode
    encoder_out = encoder(image)  # (1, enc_image_size, enc_image_size, encoder

    # Flatten encoding
    # We'll treat the problem as having a batch size of k
    if attention:
        encoder_dim = encoder_out.size(3)
        encoder_out = encoder_out.view(1, -1, encoder_dim)  # (1, num_pixels, e
        num_pixels = encoder_out.size(1)
        encoder_out = encoder_out.expand(k, num_pixels, encoder_dim)  # (k, num
    else:
        encoder_out = encoder_out.reshape(1, -1)
        encoder_dim = encoder_out.size(1)
        encoder_out = encoder_out.expand(k, encoder_dim)

    # Start decoding
    step = 1
    if attention:
        mean_encoder_out = encoder_out.mean(dim=1)
        h = decoder.init_h(mean_encoder_out)  # (1, decoder_dim)
        c = decoder.init_c(mean_encoder_out)
    else:
        init_input = decoder.bn(decoder.init(encoder_out))
        h, c = decoder.decode_step(init_input)  # (batch_size_t, decoder_dim)

    smoth_wrong = False

    # s is a number less than or equal to k, because sequences are removed fron
```

```
    while True:
        embeddings = decoder.embedding(k_prev_words).squeeze(1)  # (s, embed_di
        if attention:
            scores, _, h, c = decoder.one_step(embeddings, encoder_out, h, c)
        else:
            scores, h, c = decoder.one_step(embeddings, h, c)
        scores = F.log_softmax(scores, dim=1)
        scores = top_k_scores.expand_as(scores) + scores  # (s, vocab_size)
        # For the first step, all k points will have the same scores (since sam
        if step == 1:
            top_k_scores, top_k_words = scores[0].topk(k, 0, True, True)  # (s)
        else:
            # Unroll and find top scores, and their unrolled indices
            top_k_scores, top_k_words = scores.view(-1).topk(k, 0, True, True)
        # Convert unrolled indices to actual indices of scores
        prev_word_inds = top_k_words // vocab_size  # (s)
        next_word_inds = top_k_words % vocab_size  # (s)
        # Add new words to sequences
        seqs = torch.cat([seqs[prev_word_inds], next_word_inds.unsqueeze(1)], d
        # Which sequences are incomplete (didn't reach <end>)?
        incomplete_inds = [ind for ind, next_word in enumerate(next_word_inds)
                           next_word != word_map['<end>']]
        complete_inds = list(set(range(len(next_word_inds))) - set(incomplete_i
        # Set aside complete sequences
        if len(complete_inds) > 0:
            complete_seqs.extend(seqs[complete_inds].tolist())
            complete_seqs_scores.extend(top_k_scores[complete_inds])
        k -= len(complete_inds)  # reduce beam length accordingly
        # Proceed with incomplete sequences
        if k == 0:
            break
        seqs = seqs[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)
        # Break if things have been going on too long
        if step > 50:
            smoth_wrong = True
            break
        step += 1
    if not smoth_wrong:
        i = complete_seqs_scores.index(max(complete_seqs_scores))
        seq = complete_seqs[i]
    else:
        seq = seqs[0][:20]
    # References
    img_caps = allcaps[0].tolist()
```

```
    img_captions = list(
        map(lambda c: [w for w in c if w not in {word_map['<start>'], word_map[
            img_caps))  # remove <start> and pads
    references.append(img_captions)
    # Hypotheses
    hypotheses.append([w for w in seq if w not in {word_map['<start>'], word_ma
    assert len(references) == len(hypotheses)
```

```
EVALUATING AT BEAM SIZE 1:   0%|          | 0/25000 [00:00<?, ?it/s]/opt/conda/l
EVALUATING AT BEAM SIZE 1: 100%|██████████| 25000/25000 [14:08<00:00, 29.46it/s]
```

In [21]:

```
# Calculate BLEU-4 scores
bleu4 = corpus_bleu(references, hypotheses)
print(bleu4)
```

```
0.2607194479996266
```

In [ ]: