COMS W4705 - Homework 3

Conditioned LSTM Language Model for Image Captioning

Daniel Bauer bauer@cs.columbia.edu

Follow the instructions in this notebook step-by step. Much of the code is provided (especially in part I, II, and III), but some sections are marked with **todo**. Make sure to complete all these sections.

Specifically, you will build the following components:

- Part I (14pts): Create encoded representations for the images in the flickr dataset using a pretrained image encoder(ResNet)
- Part II (14pts): Prepare the input caption data.
- Part III (24pts): Train an LSTM language model on the caption portion of the data and use it as a generator.
- Part IV (24pts): Modify the LSTM model to also pass a copy of the input image in each timestep.
- Part V (24pts): Implement beam search for the image caption generator.

As for homework 4, access to a GPU is required.

Getting Started

There are a few required packages.

```
import os
import PIL # Python Image Library

import torch
from torch import nn
from torch.utils.data import Dataset, DataLoader

import torchvision
from torchvision.models import ResNet18_Weights


if torch.cuda.is_available():
    DEVICE = 'cuda'
elif torch.mps.is_available():
    DEVICE = 'mps'
else:
    DEVICE = 'cpu'
    print("You won't be able to train the RNN decoder on a CPU, unfortunately.")
print(DEVICE)
```

```
cuda
```

Access to the flickr8k data

We will use the flickr8k data set, described here in more detail:

> M. Hodosh, P. Young and J. Hockenmaier (2013) "Framing Image Description as a Ranking Task: Data, Models and Evaluation Metrics", Journal of Artificial Intelligence Research, Volume 47, pages 853-899
> http://www.jair.org/papers/paper3994.html

If you are using Colab:

- The data is available on google drive. You can access the folder here: https://drive.google.com/drive/folders/1sXWOLkmhpA1KFjVR0VjxGUtzAlmIvU39?usp=sharing
- Sharing is only enabled for the lionmail domain. Please make sure you are logged into Google Drive using your Columbia UNI. I will not be able to respond to individual sharing requests from your personal account.
- Once you have opened the folder, click on "Shared With Me", then select the hw5data folder, and press shift+z. This will open the "add to drive" menu. Add the folder to your drive. (This will not create a copy, but just an additional entry point to the shared folder).

N.B.: Usage of this data is limited to this homework assignment. If you would like to experiment with the dataset beyond this course, I suggest that you submit your own download request here (it's free): https://forms.illinois.edu/sec/1713398

If you are running the code locally (or on a cloud VM): I also placed a copy in a Google cloud storage bucket here: https://storage.googleapis.com/4705-hw5-data/hw5data-20220809T182644Z-001.zip

```
# OPTIONAL (if not using Colab and the data in Google Drive): Download the data.
!wget https://storage.googleapis.com/4705_fa24_hw3/hw3data.zip
```

```
--2024-11-21 00:01:44--  https://storage.googleapis.com/4705_fa24_hw3/hw3data.zip
Resolving storage.googleapis.com (storage.googleapis.com)... 142.250.145.207, 74.125.128.207, 74.125.1
Connecting to storage.googleapis.com (storage.googleapis.com)|142.250.145.207|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 1115435617 (1.0G) [application/zip]
Saving to: ‘hw3data.zip’

hw3data.zip         100%[===================>]   1.04G  36.8MB/s    in 28s

2024-11-21 00:02:13 (37.4 MB/s) - ‘hw3data.zip’ saved [1115435617/1115435617]
```

```
#Then unzip the data
!unzip hw3data.zip
```

```
Streaming output truncated to the last 5000 lines.
  inflating: hw3data/Flickr8k_Dataset/2846785268_904c5fcf9f.jpg
  inflating: hw3data/Flickr8k_Dataset/2846843520_b0e6211478.jpg
  inflating: hw3data/Flickr8k_Dataset/2847514745_9a35493023.jpg
  inflating: hw3data/Flickr8k_Dataset/2847615962_c330bded6e.jpg
  inflating: hw3data/Flickr8k_Dataset/2847859796_4d9cb0d31f.jpg
  inflating: hw3data/Flickr8k_Dataset/2848266893_9693c66275.jpg
  inflating: hw3data/Flickr8k_Dataset/2848571082_26454cb981.jpg
  inflating: hw3data/Flickr8k_Dataset/2848895544_6d06210e9d.jpg
  inflating: hw3data/Flickr8k_Dataset/2848977044_446a31d86e.jpg
  inflating: hw3data/Flickr8k_Dataset/2849194983_2968c72832.jpg
  inflating: hw3data/Flickr8k_Dataset/2850719435_221f15e951.jpg
  inflating: hw3data/Flickr8k_Dataset/2851198725_37b6027625.jpg
  inflating: hw3data/Flickr8k_Dataset/2851304910_b5721199bc.jpg
  inflating: hw3data/Flickr8k_Dataset/2851931813_eaf8ed7be3.jpg
  inflating: hw3data/Flickr8k_Dataset/2852982055_8112d0964f.jpg
  inflating: hw3data/Flickr8k_Dataset/285306009_f6ddabe687.jpg
  inflating: hw3data/Flickr8k_Dataset/2853205396_4fbe8d7a73.jpg
  inflating: hw3data/Flickr8k_Dataset/2853407781_c9fea8eef4.jpg
  inflating: hw3data/Flickr8k_Dataset/2853743795_e90ebc669d.jpg
  inflating: hw3data/Flickr8k_Dataset/2853811730_fbb8ab0878.jpg
  inflating: hw3data/Flickr8k_Dataset/2854207034_1f00555703.jpg
  inflating: hw3data/Flickr8k_Dataset/2854234756_8c0e472f51.jpg
  inflating: hw3data/Flickr8k_Dataset/2854291706_d4c31dbf56.jpg
  inflating: hw3data/Flickr8k_Dataset/2854959952_3991a385ab.jpg
  inflating: hw3data/Flickr8k_Dataset/2855417531_521bf47b50.jpg
  inflating: hw3data/Flickr8k_Dataset/2855594918_1d1e6a6061.jpg
  inflating: hw3data/Flickr8k_Dataset/2855667597_bf6ceaef8e.jpg
```

```
  inflating: hw3data/Flickr8k_Dataset/2855695119_4342aae0a3.jpg
  inflating: hw3data/Flickr8k_Dataset/2855727603_e917ded363.jpg
  inflating: hw3data/Flickr8k_Dataset/285586547_c81f8905a1.jpg
  inflating: hw3data/Flickr8k_Dataset/2855910826_d075845288.jpg
  inflating: hw3data/Flickr8k_Dataset/2856080862_95d793fa9d.jpg
  inflating: hw3data/Flickr8k_Dataset/2856252334_1b1a230e70.jpg
  inflating: hw3data/Flickr8k_Dataset/2856456013_335297f587.jpg
  inflating: hw3data/Flickr8k_Dataset/2856524322_1d04452a21.jpg
  inflating: hw3data/Flickr8k_Dataset/2856699493_65edef80a1.jpg
  inflating: hw3data/Flickr8k_Dataset/2856700531_312528eea4.jpg
  inflating: hw3data/Flickr8k_Dataset/2856923934_6eb8832c9a.jpg
  inflating: hw3data/Flickr8k_Dataset/2857372127_d86639002c.jpg
  inflating: hw3data/Flickr8k_Dataset/2857473929_4f52662c30.jpg
  inflating: hw3data/Flickr8k_Dataset/2857558098_98e9249284.jpg
  inflating: hw3data/Flickr8k_Dataset/2857609295_16aaa85293.jpg
  inflating: hw3data/Flickr8k_Dataset/2858439751_daa3a30ab8.jpg
  inflating: hw3data/Flickr8k_Dataset/2858759108_6e697c5f3e.jpg
  inflating: hw3data/Flickr8k_Dataset/2858903676_6278f07ee3.jpg
  inflating: hw3data/Flickr8k_Dataset/2860035355_3fe7a5caa4.jpg
  inflating: hw3data/Flickr8k_Dataset/2860040276_eac0aca4fc.jpg
  inflating: hw3data/Flickr8k_Dataset/2860041212_797afd6ccf.jpg
  inflating: hw3data/Flickr8k_Dataset/2860202109_97b2b22652.jpg
  inflating: hw3data/Flickr8k_Dataset/2860372882_e0ef4131d4.jpg
  inflating: hw3data/Flickr8k_Dataset/2860400846_2c1026a573.jpg
  inflating: hw3data/Flickr8k_Dataset/2860667542_95abec3380.jpg
  inflating: hw3data/Flickr8k_Dataset/2860872588_f2c7b30e1a.jpg
  inflating: hw3data/Flickr8k_Dataset/2861100960_457ceda7fa.jpg
  inflating: hw3data/Flickr8k_Dataset/2861413434_f0e2a10179.jpg
  inflating: hw3data/Flickr8k_Dataset/2861932486_52befd8592.jpg
```

The following variable should point to the location where the data is located.

```
#this is where you put the name of your data folder.
#Please make sure it's correct because it'll be used in many places later.
MY_DATA_DIR="hw3data"
```

Part I: Image Encodings (14 pts)

The files Flickr_8k.trainImages.txt Flickr_8k.devImages.txt Flickr_8k.testImages.txt, contain a list of training, development, and test images, respectively. Let's load these lists.

```
def load_image_list(filename):
    with open(filename,'r') as image_list_f:
        return [line.strip() for line in image_list_f]
```

```
FLICKR_PATH="hw3data/"
```

```
train_list = load_image_list(os.path.join(FLICKR_PATH, 'Flickr_8k.trainImages.txt'))
dev_list = load_image_list(os.path.join(FLICKR_PATH,'Flickr_8k.devImages.txt'))
test_list = load_image_list(os.path.join(FLICKR_PATH,'Flickr_8k.testImages.txt'))
```

Let's see how many images there are

```
len(train_list), len(dev_list), len(test_list)
```

```
(6000, 1000, 1000)
```

Each entry is an image filename.

```
dev_list[20]
```

```
'3693961165_9d6c333d5b.jpg'
```

The images are located in a subdirectory.

```
IMG_PATH = os.path.join(FLICKR_PATH, "Flickr8k_Dataset")
```

We can use PIL to open and display the image:

```
image = PIL.Image.open(os.path.join(IMG_PATH, dev_list[20]))
image
```

Preprocessing

We are going to use an off-the-shelf pre-trained image encoder, the ResNet-18 network. Here is more detail about this model (not required for this project):

> Kaiming He, Xiangyu Zhang, Shaoqing Ren, Jian Sun; Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2016, pp. 770-778
> https://openaccess.thecvf.com/content_cvpr_2016/papers/He_Deep_Residual_Learning_CVPR_2016_paper.pdf

The model was initially trained on an object recognition task over the ImageNet1k data. The task is to predict the correct class label for an image, from a set of 1000 possible classes.

To feed the flickr images to ResNet, we need to perform the same normalization that was applied to the training images. More details here: https://pytorch.org/vision/main/models/generated/torchvision.models.resnet18.html

```
from torchvision import transforms

preprocess = transforms.Compose([
    transforms.Resize(256),
    transforms.CenterCrop(224),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])
])
```

The resulting images, after preprocessing, are (3,224,244) tensors, where the first dimension represents the three color channels, R,G,B).

```
processed_image = preprocess(image)
processed_image.shape
```

```
torch.Size([3, 224, 224])
```

To the ResNet18 model, the images look like this:

```
transforms.ToPILImage()(processed_image)
```

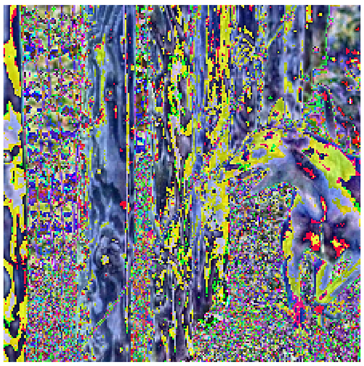


Image Encoder

Let's instantiate the ReseNet18 encoder. We are going to use the pretrained weights available in torchvision.

```
img_encoder = torchvision.models.resnet18(weights=ResNet18_Weights.DEFAULT)
```

```
Downloading: "https://download.pytorch.org/models/resnet18-f37072fd.pth" to /root/.cache/torch/hub/che
100%|██████████| 44.7M/44.7M [00:00<00:00, 182MB/s]
```

```
img_encoder.eval()
```

```
ResNet(
  (conv1): Conv2d(3, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3), bias=False)
  (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (relu): ReLU(inplace=True)
  (maxpool): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1, ceil_mode=False)
  (layer1): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
    (1): BasicBlock(
      (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
  )
  (layer2): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(64, 128, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
```

```
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (downsample): Sequential(
        (0): Conv2d(64, 128, kernel_size=(1, 1), stride=(2, 2), bias=False)
        (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      )
    )
    (1): BasicBlock(
      (conv1): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
  )
  (layer3): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(128, 256, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (downsample): Sequential(
        (0): Conv2d(128, 256, kernel_size=(1, 1), stride=(2, 2), bias=False)
        (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      )
    )
    (1): BasicBlock(
      (conv1): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
```

This is a prediction model,so the output is typically a softmax-activated vector representing 1000 possible object types. Because we are interested in an encoded representation of the image we are just going to use the second-to-last layer as a source of image encodings. Each image will be encoded as a vector of size 512.

We will use the following hack: remove the last layer, then reinstantiate a Squential model from the remaining layers.

```
lastremoved = list(img_encoder.children())[:-1]
img_encoder = torch.nn.Sequential(*lastremoved).to(DEVICE) # also send it to GPU memory
img_encoder.eval()
```

```
Sequential(
  (0): Conv2d(3, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3), bias=False)
  (1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (2): ReLU(inplace=True)
  (3): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1, ceil_mode=False)
  (4): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
    (1): BasicBlock(
      (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
  )
  (5): Sequential(
    (0): BasicBlock(
```

```
      (conv1): Conv2d(64, 128, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (downsample): Sequential(
        (0): Conv2d(64, 128, kernel_size=(1, 1), stride=(2, 2), bias=False)
        (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      )
    )
    (1): BasicBlock(
      (conv1): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
  )
  (6): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(128, 256, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (downsample): Sequential(
        (0): Conv2d(128, 256, kernel_size=(1, 1), stride=(2, 2), bias=False)
        (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      )
    )
    (1): BasicBlock(
      (conv1): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
```

Let's try the encoder.

```
def get_image(img_name):
    image = PIL.Image.open(os.path.join(IMG_PATH, img_name))
    return preprocess(image)
```

```
preprocessed_image = get_image(train_list[0])
encoded = img_encoder(preprocessed_image.unsqueeze(0).to(DEVICE)) # unsqueeze required to add batch dim (3
encoded.shape
```

```
torch.Size([1, 512, 1, 1])
```

The result isn't quite what we wanted: The final representation is actually a 1x1 "image" (the first dimension is the batch size). We can just grab this one pixel:

```
encoded = encoded[:,:,0,0] #this is our final image encoded
encoded.shape
```

```
torch.Size([1, 512])
```

TODO: Because we are just using the pretrained encoder, we can simply encode all the images in a preliminary step. We will store them in one big tensor (one for each dataset, train, dev, test). This will save some time when training the conditioned LSTM because we won't have to recompute the image encodings with each training epoch. We can also save the tensors to disk so that we never have to touch the bulky image data again.

Complete the following function that should take a list of image names and return a tensor of size [n_images, 512] (where each row represents one image).

For example `encode_imates(train_list)` should return a [6000,512] tensor.

```
def encode_images(image_list):
    encode_batch = torch.Tensor().to(DEVICE)
    for img_name in image_list:
        preprocessed_image = get_image(img_name)
        encoded = img_encoder(preprocessed_image.unsqueeze(0).to(DEVICE))[:, :, 0, 0]
        encode_batch = torch.concat([encode_batch, encoded])
    return encode_batch

with torch.no_grad():
    enc_images_train = encode_images(train_list)
enc_images_train.shape
```

```
torch.Size([6000, 512])
```

We can now save this to disk:

```
torch.save(enc_images_train, open('encoded_images_train.pt','wb'))
```

It's a good idea to save the resulting matrices, so we do not have to run the encoder again.

Part II Text (Caption) Data Preparation (14 pts)

Next, we need to load the image captions and generate training data for the language model. We will train a text-only model first.

Reading image descriptions

TODO: Write the following function that reads the image descriptions from the file `filename` and returns a dictionary in the following format. Take a look at the file `Flickr8k.token.txt` for the format of the input file. The keys of the dictionary should be image filenames. Each value should be a list of 5 captions. Each caption should be a list of tokens.

The captions in the file are already tokenized, so you can just split them at white spaces. You should convert each token to lower case. You should then pad each caption with a <START> token on the left and an <END> token on the right.

For example, a single caption might look like this: ['<START>', 'a', 'child', 'in', 'a', 'pink', 'dress', 'is', 'climbing', 'up', 'a', 'set', 'of', 'stairs', 'in', 'an', 'entry', 'way', '.', '<EOS>'],

```
def read_image_descriptions(filename):
    image_descriptions = {}

    with open(filename,'r') as in_file:
        for line in in_file:
            components = line.lower().split()
            img_name = components[0].split("#")[0]
            caption_tokens = ['<START>'] + components[1:] + ['<EOS>']
            if img_name in image_descriptions.keys():
                image_descriptions[img_name].append(caption_tokens)
            else:
```

```
            image_descriptions[img_name] = [caption_tokens]
    return image_descriptions
```

```
os.path.join(FLICKR_PATH, "Flickr8k.token.txt")
```

```
'hw3data/Flickr8k.token.txt'
```

```
descriptions = read_image_descriptions(os.path.join(FLICKR_PATH, "Flickr8k.token.txt"))
```

```
descriptions['1000268201_693b08cb0e.jpg']
```

```
[['<START>',
  'a',
  'child',
  'in',
  'a',
  'pink',
  'dress',
  'is',
  'climbing',
  'up',
  'a',
  'set',
  'of',
  'stairs',
  'in',
  'an',
  'entry',
  'way',
  '.',
  '<EOS>'],
 ['<START>',
  'a',
  'girl',
  'going',
  'into',
  'a',
  'wooden',
  'building',
  '.',
  '<EOS>'],
 ['<START>',
  'a',
  'little',
  'girl',
  'climbing',
  'into',
  'a',
  'wooden',
  'playhouse',
  '.',
  '<EOS>'],
 ['<START>',
  'a',
  'little',
  'girl',
  'climbing',
  'the',
  'stairs',
  'to',
  'her',
  'playhouse',
  '.',
  '<EOS>'],
 ['<START>',
  'a',
  'little',
```

```
        'girl',
        'in'
```

The previous line shoudl return

```
[['', 'a', 'child', 'in', 'a', 'pink', 'dress', 'is', 'climbing', 'up', 'a', 'set', 'of', 'stairs', 'in',
```

Creating Word Indices

Next, we need to create a lookup table from the **training** data mapping words to integer indices, so we can encode input and output sequences using numeric representations.

TODO create the dictionaries id_to_word and word_to_id, which should map tokens to numeric ids and numeric ids to tokens. Hint: Create a set of tokens in the training data first, then convert the set into a list and sort it. This way if you run the code multiple times, you will always get the same dictionaries. This is similar to the word indices you created for homework 3 and 4.

Make sure you create word indices for the three special tokens <PAD>, <START>, and <EOS> (end of sentence).

```
id_to_word = {} #todo
id_to_word[0] = "<PAD>"
id_to_word[1] = "<START>"
id_to_word[2] = "<EOS>"
word_to_id = {} # todo


word_to_id["<PAD>"] = 0
word_to_id["<START>"] = 1
word_to_id["<EOS>"] = 2
vocab = set()
for caption_arr in descriptions.values():
    for caption_tokens in caption_arr:
        for token in caption_tokens[1:-1]:
            vocab.add(token)

id = 3
for word in sorted(list(vocab)):
    word_to_id[word] = id
    id_to_word[id] = word
    id += 1
```

```
len(word_to_id)
```

```
8921
```

```
word_to_id['cat'] # should print an integer
```

```
1349
```

```
id_to_word[1] # should print a token
```

```
'<START>'
```

Note that we do not need an UNK word token because we will only use the model as a generator, once trained.

Part III Basic Decoder Model (24 pts)

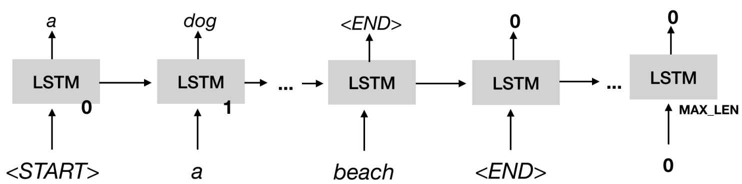
For now, we will just train a model for text generation without conditioning the generator on the image input.

We will use the LSTM implementation provided by PyTorch. The core idea here is that the recurrent layers (including LSTM) create an "unrolled" RNN. Each time-step is represented as a different position, but the weights for these positions are shared. We are going to use the constant MAX_LEN to refer to the maximum length of a sequence, which turns out to be 40 words in this data set (including START and END).

```
MAX_LEN = max(len(description) for image_id in train_list for description in descriptions[image_id])
MAX_LEN
```

```
40
```

In class, we discussed LSTM generators as transducers that map each word in the input sequence to the next word.



To train the model, we will convert each description into a set of input output pairs as follows. For example, consider the sequence

```
['<START>', 'a', 'black', 'dog', '<EOS>']
```

We would train the model using the following input/output pairs (note both sequences are padded to the right up to MAX_LEN)

i	input	output
0	[<START> , <PAD> , <PAD> , <PAD> ,...]	[a , <PAD> , <PAD> , <PAD> ,...
1	[<START> , a , <PAD> , <PAD> ,...]	[a , black , <PAD> , <PAD> ,...
2	[<START> , a , black , <PAD> ,...]	[a , black , dog , <PAD> ,...
3	[<START> , a , back , dog ,...]	[a , black , dog , <EOS> ,...

Here is the lange model in pytorch. We will choose input embeddings of dimensionality 512 (for simplicitly, we are not initializing these with pre-trained embeddings here). We will also use 512 for the hidden state vector and the output.

```
from torch import nn

vocab_size = len(word_to_id)+1
class GeneratorModel(nn.Module):

    def __init__(self):
        super(GeneratorModel, self).__init__()
        self.embedding = nn.Embedding(vocab_size, 512)
        self.lstm = nn.LSTM(512, 512, num_layers = 1, bidirectional=False, batch_first=True)
        self.output = nn.Linear(512,vocab_size)

    def forward(self, input_seq):
        hidden = self.lstm(self.embedding(input_seq))
        out = self.output(hidden[0])
        return out
```

The input sequence is an integer tensor of size `[batch_size, MAX_LEN]`. Each row is a vector of size MAX_LEN in which each entry is an integer representing a word (according to the `word_to_id` dictionary). If the input sequence is shorter than MAX_LEN, the remaining entries should be padded with ''.

For each input example, the model returns a distribution over possible output words. The model output is a tensor of size `[batch_size, MAX_LEN, vocab_size]`. vocab_size is the number of vocabulary words, i.e. len(word_to_id)

Creating a Dataset for the text training data

TODO: Write a Dataset class for the text training data. The **getitem** method should return an (input_encoding, output_encoding) pair for a single item. Both input_encoding and output_encoding should be tensors of size `[MAX_LEN]`, encoding the padded input/output sequence as illustrated above.

I recommend to first read in all captions in the **init** method and store them in a list. Above, we used the get_image_descriptions function to load the image descriptions into a dictionary. Iterate through the images in img_list, then access the corresponding captions in the `descriptions` dictionary.

You can just refer back to the variables you have defined above, including `descriptions`, `train_list`, `vocab_size`, etc.

```
MAX_LEN = 40

class CaptionDataset(Dataset):

    def __init__(self, img_list):

        self.data = []
        for img_name in img_list:
            for caption in descriptions[img_name]:
                for sentence_len in range(2, len(caption)+1):
                    # caption_enc = [word_to_id[word] for word in caption[:sentence_len]]
                    # caption_enc += [word_to_id["<PAD>"]] * (MAX_LEN - len(caption_enc))
                    self.data.append(caption[:sentence_len])

    def __len__(self):
        return len(self.data)

    def __getitem__(self,k):

        #TODO COMPLETE THIS METHOD
        # print(self.data[k][:-1], self.data[k][1:])
        padding = MAX_LEN - len(self.data[k][:-1])

        input_enc = torch.tensor([word_to_id[word] for word in self.data[k][:-1]] + [0] * padding)
        output_enc = torch.tensor([word_to_id[word] for word in self.data[k][1:]] + [0] * padding)
        return input_enc, output_enc
```

Let's instantiate the caption dataset and get the first item. You want to see something like this:

for the input:

```
tensor([   1,   74,  805, 2312, 4015, 6488,  170,   74, 8686, 2312, 3922, 7922,
        7125,   17,    2,    0,    0,    0,    0,    0,    0,    0,    0,    0,
           0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,
           0,    0,    0,    0])
```

for the output:

```
tensor([  74,  805, 2312, 4015, 6488,  170,   74, 8686, 2312, 3922, 7922, 7125,
          17,    2,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,
           0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,
           0,    0,    0,    0])
```

```
data = CaptionDataset(train_list)
```

```
i, o = data[0]
i
```

```
tensor([1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0])
```

```
o
```

```
tensor([72,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
         0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
         0,  0,  0,  0])
```

Let's try the model:

```
model = GeneratorModel().to(DEVICE)
```

```
model(i.to(DEVICE)).shape   # should return a [40, vocab_size]  tensor.
```

```
torch.Size([40, 8922])
```

Training the Model

The training function is identical to what you saw in homework 3 and 4.

```
from torch.nn import CrossEntropyLoss
loss_function = CrossEntropyLoss(ignore_index = 0, reduction='mean')

LEARNING_RATE = 1e-03
optimizer = torch.optim.AdamW(params=model.parameters(), lr=LEARNING_RATE)

loader = DataLoader(data, batch_size = 16, shuffle = True)

def train():
    """
    Train the model for one epoch.
    """
    tr_loss = 0
    nb_tr_examples, nb_tr_steps = 0, 0
    total_correct, total_predictions = 0, 0
    tr_preds, tr_labels = [], []
    # put model in training mode
    model.train()

    for idx, batch in enumerate(loader):

        inputs,targets = batch
        inputs = inputs.to(DEVICE)
        targets = targets.to(DEVICE)
```

```
        # Run the forward pass of the model
        logits = model(inputs)
        loss = loss_function(logits.transpose(2,1), targets)
        tr_loss += loss.item()
        #print("Batch loss: ", loss.item()) # can comment out if too verbose.
        nb_tr_steps += 1
        nb_tr_examples += targets.size(0)

        # Calculate accuracy
        predictions = torch.argmax(logits, dim=2)  # Predicted token labels
        not_pads = targets != 0  # Mask for non-PAD tokens
        correct = torch.sum((predictions == targets) & not_pads)
        total_correct += correct.item()
        total_predictions += not_pads.sum().item()

        if idx % 100==0:
            #torch.cuda.empty_cache() # can help if you run into memory issues
            curr_avg_loss = tr_loss/nb_tr_steps
            print(f"Current average loss: {curr_avg_loss}")

        # Run the backward pass to update parameters
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        # Compute accuracy for this batch
        # matching = torch.sum(torch.argmax(logits,dim=2) == targets)
        # predictions = torch.sum(torch.where(targets==-100,0,1))

    epoch_loss = tr_loss / nb_tr_steps
    epoch_accuracy = total_correct / total_predictions if total_predictions != 0 else 0  # Avoid division
    print(f"Training loss epoch: {epoch_loss}")
    print(f"Average accuracy epoch: {epoch_accuracy:.2f}")
```

Run the training until the accuracy reaches about 0.5 (this would be high for a language model on open-domain text, but the image caption dataset is comparatively small and closed-domain). This will take about 5 epochs.

```
epoch = 5
for _ in range(epoch):
    train()
```

```
Current average loss: 9.1083984375
Current average loss: 5.133734410352046
Current average loss: 4.65307370821635
Current average loss: 4.418297103077074
Current average loss: 4.275063166891845
Current average loss: 4.16414109389939
Current average loss: 4.068567214115289
Current average loss: 3.9900835937848274
Current average loss: 3.9298001463791254
Current average loss: 3.882667666137814
Current average loss: 3.8359282371642944
Current average loss: 3.7960014793680106
Current average loss: 3.7561903940847174
Current average loss: 3.7243730906428603
Current average loss: 3.69176983169621
Current average loss: 3.6654873329508235
Current average loss: 3.641632707769762
Current average loss: 3.6191181975347306
Current average loss: 3.596282205602846
Current average loss: 3.5714095269674253
Current average loss: 3.551613004728295
Current average loss: 3.5335704821850333
Current average loss: 3.512849487970223
```

```
Current average loss: 3.4955377705145896
Current average loss: 3.477521126094534
Current average loss: 3.463693593035503
Current average loss: 3.4495205012984753
Current average loss: 3.4359759277257598
Current average loss: 3.421704642307073
Current average loss: 3.4069585177208874
Current average loss: 3.3931127477192713
Current average loss: 3.379267887929223
Current average loss: 3.366313863642251
Current average loss: 3.3550810001359426
Current average loss: 3.3429034600991145
Current average loss: 3.331448895833861
Current average loss: 3.321172056049812
Current average loss: 3.3118682712260146
Current average loss: 3.301722309960092
Current average loss: 3.289553123570442
Current average loss: 3.2787839647115273
Current average loss: 3.268868661979093
Current average loss: 3.259097246577756
Current average loss: 3.2501368125187358
Current average loss: 3.239509465991841
Current average loss: 3.230818651803624
Current average loss: 3.2220086450811003
Current average loss: 3.212926053108243
Current average loss: 3.204096788268515
Current average loss: 3.1959601504733137
Current average loss: 3.187322572144812
Current average loss: 3.17923027972244
Current average loss: 3.1711933039353686
Current average loss: 3.163875693308815
Current average loss: 3.155931967140414
Current average loss: 3.1469382448773713
Current average loss: 3.138982484690144
```

Greedy Decoder

TODO Next, you will write a decoder. The decoder should start with the sequence `["<START>", "<PAD>","<PAD>"...]`, use the model to predict the most likely word in the next position. Append the word to the input sequence and then continue until `"<EOS>"` is predicted or the sequence reaches `MAX_LEN` words.

```
def decoder():
    """
    Decode a single image using the trained model.
    """
    model.eval()
    with torch.no_grad():
        input_enc = [word_to_id[token] for token in ['<START>'] + ['<PAD>'] * (MAX_LEN - 1)]  # (1, MAX_LE
        input_enc = torch.tensor(input_enc).unsqueeze(0).to(DEVICE)
        decoded = [1]
        for i in range(MAX_LEN):
            logits = model(input_enc)  # (1, MAX_LEN, VOCAB_SIZE)
            next_word = torch.argmax(logits, dim=2)[0, i]
            decoded.append(next_word.item())
            if i < MAX_LEN - 1:
                input_enc[0][i+1] = next_word
            if decoded[-1] == word_to_id['<EOS>']:
                break
        decoded = [id_to_word[id] for id in decoded]
        return decoded


decoder()
```

```
['a',
 'man',
 'in',
 'a',
 'white',
 'shirt',
 'and',
 'tie',
 'sings',
 'into',
 'a',
 'microphone',
 'while',
 'a',
 'woman',
 'wearing',
 'a',
 'white',
 'shirt',
 'and',
 'tie',
 'plays',
 'the',
 'guitar',
 'at',
 'a',
 'concert',
 '.',
 '<EOS>']
```

this will return something like ['a', 'man', 'in', 'a', 'white', 'shirt', 'and', 'a', 'woman', 'in', 'a', 'white', 'dress', 'walks', 'by', 'a', 'small', 'white', 'building', '.', '']

This simple decoder will of course always predict the same sequence (and it's not necessarily a good one).

TODO: Modify the decoder as follows. Instead of choosing the most likely word in each step, sample the next word from the distribution (i.e. the softmax activated output) returned by the model. Make sure to apply torch.softmax() to convert the output activations into a distribution.

To sample fromt he distribution, I recommend you take a look at [np.random.choice](https://docs.scipy.org/doc/numpy/reference/generated/numpy.random.choice.html), which takes the distribution as a parameter p.

```
import numpy as np

def sample_decoder():
    """
    Decode a single image using the trained model.
    """
    model.eval()
    with torch.no_grad():
        # Get the encoded image
        input_enc = [word_to_id[token] for token in ['<START>'] + ['<PAD>'] * (MAX_LEN - 1)]
        input_enc = torch.tensor(input_enc).unsqueeze(0).to(DEVICE)
        decoded = [1]
        for i in range(MAX_LEN):
            logits = model(input_enc)
            next_word = torch.multinomial(torch.nn.functional.softmax(logits, dim=2).squeeze(), 1)[i]
            decoded.append(next_word.item())
            if i < MAX_LEN - 1:
                input_enc[0][i+1] = next_word
            if decoded[-1] == word_to_id['<EOS>']:
                break
        decoded = [id_to_word[id] for id in decoded]
        return decoded
```

```
for i in range(5):
    print(sample_decoder())
```

```
['a', 'young', 'boy', 'jumping', 'to', 'catch', 'something', 'in', 'a', 'pool', '.', '<EOS>']
['a', 'young', 'girl', 'wearing', 'a', 'bulky', 'red', 'life', 'jacket', 'in', 'a', 'playground', '.',
['a', 'crowd', 'of', 'people', 'standing', 'in', 'front', 'of', 'statues', '.', '<EOS>']
['the', 'rider', 'of', 'the', 'brown', 'horse', 'is', 'skiing', 'down', 'a', 'blue', 'soccer', '.', '<I
['a', 'black', 'dog', 'is', 'in', 'action', 'on', 'a', 'green', 'grass', '.', '<EOS>']
```

Some example outputs (it's stochastic, so your results will vary

```
['', 'people', 'on', 'rocky', 'ground', 'swinging', 'basketball', '']
['', 'the', 'two', 'hikers', 'take', 'a', 'tandem', 'leap', 'while', 'another', 'is', 'involving', 'watch:
['', 'a', 'man', 'attached', 'to', 'a', 'bicycle', 'rides', 'a', 'motorcycle', '.', '']
['', 'a', 'surfer', 'is', 'riding', 'a', 'wave', 'in', 'the', 'ocean', '.', '']
['', 'a', 'child', 'plays', 'in', 'a', 'round', 'fountain', '.', '']
```

You should now be able to see some interesting output that looks a lot like flickr8k image captions -- only that the captions are generated randomly without any image input.

Part III - Conditioning on the Image (24 pts)

We will now extend the model to condition the next word not only on the partial sequence, but also on the encoded image.

We will concatenate the 512-dimensional image representation to each 512-dimensional token embedding. The LSTM will therefore see input representations of size 1024.

TODO: Write a new Dataset class for the combined image captioning data set. Each call to **getitem** should return a triple (image_encoding, input_encoding, output_encoding) for a single item. Both input_encoding and output_encoding should be tensors of size [MAX_LEN], encoding the padded input/output sequence as illustrated above. The image_encoding is the size [512] tensor we pre-computed in part I.

Note: One tricky issue here is that each image corresponds to 5 captions, so you have to find the correct image for each caption. You can create a mapping from image names to row indices in the image encoding tensor. This way you will be able to find each image by it's name.

```
MAX_LEN = 40

class CaptionAndImage(Dataset):

    def __init__(self, img_list):

        self.img_data = torch.load(open("encoded_images_train.pt",'rb')) # suggested
        self.img_name_to_id = dict([(i,j) for (j,i) in enumerate(img_list)])

        self.data = []
        # TODO COMPLETE THIS METHOD
        for img_name in img_list:
            for caption in descriptions[img_name]:
                # caption_enc = [word_to_id[word] for word in caption]
                # caption_enc += [word_to_id["<PAD>"]] * (MAX_LEN - len(caption_enc))
                img_id = self.img_name_to_id[img_name]
                for sentence_len in range(2, len(caption)+1):
                    # caption_enc = [word_to_id[word] for word in caption[:sentence_len]]
```

```
                # caption_enc += [word_to_id["<PAD>"]] * (MAX_LEN - len(caption_enc))
                self.data.append((img_id, caption[:sentence_len]))

    def __len__(self):
        return len(self.data)

    def __getitem__(self,k):
        # TODO COMPLETE THIS METHOD
        img_id = self.data[k][0]
        img_data = self.img_data[img_id]

        caption_data = self.data[k][1]
        padding = MAX_LEN - len(caption_data[:-1])
        input_enc = torch.tensor([word_to_id[word] for word in caption_data[:-1]] + [0] * padding)
        output_enc = torch.tensor([word_to_id[word] for word in caption_data[1:]] + [0] * padding)

        return img_data, input_enc, output_enc


torch.cuda.empty_cache()
joint_data = CaptionAndImage(train_list)
img, i, o = joint_data[0]
img.shape # should return torch.Size([512])
```

```
<ipython-input-150-22bc268c1ac8>:7: FutureWarning: You are using `torch.load` with `weights_only=False`
  self.img_data = torch.load(open("encoded_images_train.pt",'rb')) # suggested
torch.Size([512])
```

```
i.shape # should return torch.Size([40])
```

```
torch.Size([40])
```

```
o.shape # should return torch.Size([40])
```

```
torch.Size([40])
```

TODO: Updating the model Update the language model code above to include a copy of the image for each position. The forward function of the new model should take two inputs:

1. a `(batch_size, 2048)` ndarray of image encodings.
2. a `(batch_size, MAX_LEN)` ndarray of partial input sequences.

And one output as before: a `(batch_size, vocab_size)` ndarray of predicted word distributions.

The LSTM will take input dimension 1024 instead of 512 (because we are concatenating the 512-dim image encoding).

In the forward function, take the image and the embedded input sequence (i.e. AFTER the embedding was applied), and concatenate the image to each input. This requires some tensor manipulation. I recommend taking a look at torch.Tensor.expand and torch.Tensor.cat.

```
vocab_size = len(word_to_id)+1

class CaptionGeneratorModel(nn.Module):

    def __init__(self):
        super(CaptionGeneratorModel, self).__init__()
        # TODO COMPLETE THIS METHOD
        self.embedding = nn.Embedding(vocab_size, 512)
        self.lstm = nn.LSTM(1024, 512, num_layers = 1, bidirectional=False, batch_first=True)
        self.output = nn.Linear(512, vocab_size)
```

```
    def forward(self, img, input_seq):

        # TODO COMPLETE THIS METHOD
        hidden = self.lstm(torch.cat([img.unsqueeze(1).expand(-1, MAX_LEN, -1), self.embedding(input_seq)]
        out = self.output(hidden[0])
        return out
```

Let's try this new model on one item:

```
model = CaptionGeneratorModel().to(DEVICE)
```

```
item = joint_data[0]
img, input_seq, output_seq = item
```

```
logits = model(img.unsqueeze(0).to(DEVICE), input_seq.unsqueeze(0).to(DEVICE))

logits.shape # should return (1,40,8922) = (batch_size, MAX_LEN, vocab_size)
```

```
torch.Size([1, 40, 8922])
```

The training function is, again, mostly unchanged. Keep training until the accuracy exceeds 0.5.

```
from torch.nn import CrossEntropyLoss
loss_function = CrossEntropyLoss(ignore_index = 0, reduction='mean')

LEARNING_RATE = 1e-03
optimizer = torch.optim.AdamW(params=model.parameters(), lr=LEARNING_RATE)

loader = DataLoader(joint_data, batch_size = 16, shuffle = True)

def train():
    """
    Train the model for one epoch.
    """
    tr_loss = 0
    nb_tr_examples, nb_tr_steps = 0, 0
    total_correct, total_predictions = 0, 0
    tr_preds, tr_labels = [], []
    # put model in training mode
    model.train()

    for idx, batch in enumerate(loader):

        img, inputs,targets = batch
        img = img.to(DEVICE)
        inputs = inputs.to(DEVICE)
        targets = targets.to(DEVICE)

        # Run the forward pass of the model
        logits = model(img, inputs)
        loss = loss_function(logits.transpose(2,1), targets)
        tr_loss += loss.item()
        #print("Batch loss: ", loss.item()) # can comment out if too verbose.
        nb_tr_steps += 1
        nb_tr_examples += targets.size(0)

        # Calculate accuracy
        predictions = torch.argmax(logits, dim=2)  # Predicted token labels
        not_pads = targets != 0  # Mask for non-PAD tokens
```

```
        correct = torch.sum((predictions == targets) & not_pads)
        total_correct += correct.item()
        total_predictions += not_pads.sum().item()

        if idx % 100==0:
            #torch.cuda.empty_cache() # can help if you run into memory issues
            curr_avg_loss = tr_loss/nb_tr_steps
            print(f"Current average loss: {curr_avg_loss}")

        # Run the backward pass to update parameters
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        # Compute accuracy for this batch
        # matching = torch.sum(torch.argmax(logits,dim=2) == targets)
        # predictions = torch.sum(torch.where(targets==-100,0,1))

    epoch_loss = tr_loss / nb_tr_steps
    epoch_accuracy = total_correct / total_predictions if total_predictions != 0 else 0  # Avoid division
    print(f"Training loss epoch: {epoch_loss}")
    print(f"Average accuracy epoch: {epoch_accuracy:.2f}")


epoch = 4
for _ in range(epoch):
    train()
```

```
Current average loss: 9.090627670288086
Current average loss: 5.242588723059928
Current average loss: 4.777101095636093
Current average loss: 4.527749517035247
Current average loss: 4.359527883981529
Current average loss: 4.227676679512222
Current average loss: 4.1295447647075685
Current average loss: 4.053660031222073
Current average loss: 3.98924126607202
Current average loss: 3.9283752219129218
Current average loss: 3.8778664279770063
Current average loss: 3.8343586313193976
Current average loss: 3.7917928256956763
Current average loss: 3.7533637857913607
Current average loss: 3.7199325590453602
Current average loss: 3.6901337676331014
Current average loss: 3.658158825011197
Current average loss: 3.6316811968339744
Current average loss: 3.6093962140906726
Current average loss: 3.5844243836992353
Current average loss: 3.561179779220497
Current average loss: 3.537114596718666
Current average loss: 3.5164597864207328
Current average loss: 3.496237804971535
Current average loss: 3.477109297570066
Current average loss: 3.457187472224855
Current average loss: 3.4395654603913397
Current average loss: 3.4238696955434045
Current average loss: 3.406202951102374
Current average loss: 3.391639757468675
Current average loss: 3.3749389407317745
Current average loss: 3.3598482689908225
Current average loss: 3.347999346289773
Current average loss: 3.33403835506809
Current average loss: 3.319886178663288
Current average loss: 3.308000650483518
Current average loss: 3.2946976742390888
Current average loss: 3.282902825332854
Current average loss: 3.2708474057688584
Current average loss: 3.2600255786562786
```

```
Current average loss: 3.24793171733655
Current average loss: 3.2372161016264243
Current average loss: 3.2268596257632245
Current average loss: 3.217037686052169
Current average loss: 3.2064668082616676
Current average loss: 3.1959035008039987
Current average loss: 3.186429471794457
Current average loss: 3.1767069527352776
Current average loss: 3.1668397125614804
Current average loss: 3.157634356366788
Current average loss: 3.1487075230570416
Current average loss: 3.1392696491995644
Current average loss: 3.1301940688259946
Current average loss: 3.122582152730405
Current average loss: 3.114115038551107
Current average loss: 3.1056486208251206
Current average loss: 3.097383779428874
Current average loss: 3.0894510128940955
```

TODO: Testing the model: Rewrite the greedy decoder from above to take an encoded image representation as input.

```
def greedy_decoder(img):
    #TODO: Complete this method

    model.eval()
    img = img.unsqueeze(0)
    with torch.no_grad():
        input_enc = [word_to_id[token] for token in ['<START>'] + ['<PAD>'] * (MAX_LEN - 1)]
        input_enc = torch.tensor(input_enc).unsqueeze(0).to(DEVICE)
        decoded = [1]
        for i in range(MAX_LEN):
            logits = model(img, input_enc)
            next_word = torch.argmax(logits, dim=2)[0, i]
            decoded.append(next_word.item())
            if i < MAX_LEN - 1:
                input_enc[0][i+1] = next_word
            if decoded[-1] == word_to_id['<EOS>']:
                break
        decoded = [id_to_word[id] for id in decoded]
        return decoded
```

Now we can load one of the dev images, pass it through the preprocessor and the image encoder, and then into the decoder!

```
raw_img = PIL.Image.open(os.path.join(IMG_PATH, dev_list[199]))
preprocessed_img = preprocess(raw_img).to(DEVICE)
encoded_img = img_encoder(preprocessed_img.unsqueeze(0)).reshape((512))
caption = greedy_decoder(encoded_img)
print(caption)
raw_img
```

```
['a', 'woman', 'is', 'pointing', 'at', 'a', 'white', 'snow', 'and', 'pointing', 'to', 'the', 'woman',
```

The result should look pretty good for most images, but the model is prone to hallucinations.

Part IV - Beam Search Decoder (24 pts)

TODO Modify the simple greedy decoder for the caption generator to use beam search. Instead of always selecting the most probable word, use a *beam*, which contains the n highest-scoring sequences so far and their total probability (i.e. the product of all word probabilities). I recommend that you use a list of `(probability, sequence)` tuples. After each time-step, prune the list to include only the n most probable sequences.

Then, for each sequence, compute the n most likely successor words. Append the word to produce n new sequences and compute their score. This way, you create a new list of n*n candidates.

Prune this list to the best n as before and continue until `MAX_LEN` words have been generated.

Note that you cannot use the occurence of the `"<EOS>"` tag to terminate generation, because the tag may occur in different positions for different entries in the beam.

Once `MAX_LEN` has been reached, return the most likely sequence out of the current n.

```
def img_beam_decoder(n, img):

    # TODO: Complete this method
    model.eval()
    img = img.unsqueeze(0)
    with torch.no_grad():
        input_enc = [word_to_id[token] for token in ['<START>'] + ['<PAD>'] * (MAX_LEN - 1)]
        input_enc = torch.tensor(input_enc).unsqueeze(0).to(DEVICE)
        beam = [(1, input_enc)]
```

```
        for i in range(MAX_LEN):
            search_range = []
            for prob, seq in beam:
                logits = model(img, seq)
                probs = torch.nn.functional.softmax(logits, dim=2)
                next_words = torch.argsort(probs[0, i], descending=True)[:n]
                for next_word in next_words:
                    new_seq = seq.clone()
                    if i < MAX_LEN - 1:
                        new_seq[0][i+1] = next_word
                    new_prob = prob * probs[0, i, next_word].item()
                    search_range.append((new_prob, new_seq))
            search_range.sort(reverse=True, key=lambda x: x[0])
            beam = search_range[:n]
    return [id_to_word[id.item()] for id in beam[0][1][0, :]]
```

TODO Finally, before you submit this assignment, please show 3 development images, each with 1) their greedy output, 2) beam search at n=3 3) beam search at n=5.

```
test_imgs = dev_list[:3]
for img_name in test_imgs:
    raw_img = PIL.Image.open(os.path.join(IMG_PATH, img_name))
    preprocessed_img = preprocess(raw_img).to(DEVICE)
    encoded_img = img_encoder(preprocessed_img.unsqueeze(0)).reshape((512))
    greedy_caption = greedy_decoder(encoded_img)
    caption1 = img_beam_decoder(3, encoded_img)
    caption2 = img_beam_decoder(5, encoded_img)
    print(f"Caption for {img_name} with greedy search:\n {greedy_caption} \n Caption for {img_name} with b
    display(raw_img)
```

```
Caption for 2090545563_a4e66ec76b.jpg with greedy search:
 ['a', 'young', 'blonde', 'boy', 'in', 'a', 'red', 'shirt', 'grinds', 'a', 'skateboard', 'along', 'a',
 Caption for 2090545563_a4e66ec76b.jpg with beam size 3:
['<START>', 'there', 'is', 'a', 'person', 'in', 'red', 'shirt', 'on', 'a', 'skateboard', 'going', 'dow
 Caption for 2090545563_a4e66ec76b.jpg with beam size 5:
['<START>', 'there', 'is', 'a', 'person', 'in', 'red', 'shirt', 'on', 'a', 'skateboard', 'going', 'dow
```