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.

Access to a GPU is required for this assignment. If you have a recent mac, you can try using mps. Otherwise, I recommend renting a GPU instance through a service like vast.ai or lambdalabs. Google Colab can work in a pinch, but you would have to deal with quotas and it's somewhat easy to lose unsaved work.

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

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

The data is available in a Google Cloud storage bucket here: https://storage.googleapis.com/4705_sp25_hw3/hw3data.zip

```
#Download the data.
!wget https://storage.googleapis.com/4705_sp25_hw3/hw3data.zip
```

```
--2025-11-06 04:10:57--  https://storage.googleapis.com/4705_sp25_hw3/hw3data.zip
Resolving storage.googleapis.com (storage.googleapis.com)... 142.251.12.207, 64.233.170.207, 142.250.4.207, ...
Connecting to storage.googleapis.com (storage.googleapis.com)|142.251.12.207|:443... connected.
```

```
HTTP request sent, awaiting response... 200 OK
Length: 1115435617 (1.0G) [application/zip]
Saving to: ‘hw3data.zip.2’

hw3data.zip.2       100%[===================>]   1.04G  18.0MB/s    in 53s

2025-11-06 04:11:50 (20.3 MB/s) - ‘hw3data.zip.2’ saved [1115435617/1115435617]
```

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

```
  inflating: hw3data/Flickr8k_Dataset/947969010_f1ea572e89.jpg
  inflating: hw3data/Flickr8k_Dataset/948196883_e190a483b1.jpg
  inflating: hw3data/Flickr8k_Dataset/950273886_88c324e663.jpg
  inflating: hw3data/Flickr8k_Dataset/950411653_20d0335946.jpg
  inflating: hw3data/Flickr8k_Dataset/95151149_5ca6747df6.jpg
  inflating: hw3data/Flickr8k_Dataset/952171414_2db16f846f.jpg
  inflating: hw3data/Flickr8k_Dataset/953941506_5082c9160c.jpg
  inflating: hw3data/Flickr8k_Dataset/954987350_a0c608b467.jpg
  inflating: hw3data/Flickr8k_Dataset/956164675_9ee084364e.jpg
  inflating: hw3data/Flickr8k_Dataset/957230475_48f4285ffe.jpg
  inflating: hw3data/Flickr8k_Dataset/95728660_d47de66544.jpg
  inflating: hw3data/Flickr8k_Dataset/95728664_06c43b90f1.jpg
  inflating: hw3data/Flickr8k_Dataset/95734035_84732a92c1.jpg
  inflating: hw3data/Flickr8k_Dataset/95734036_bef6d1a871.jpg
  inflating: hw3data/Flickr8k_Dataset/95734038_2ab5783da7.jpg
  inflating: hw3data/Flickr8k_Dataset/957682378_46c3b07bcd.jpg
  inflating: hw3data/Flickr8k_Dataset/95783195_e1ba3f57ca.jpg
  inflating: hw3data/Flickr8k_Dataset/958326692_6210150354.jpg
  inflating: hw3data/Flickr8k_Dataset/961189263_0990f3bcb5.jpg
  inflating: hw3data/Flickr8k_Dataset/961611340_251081fcb8.jpg
  inflating: hw3data/Flickr8k_Dataset/963730324_0638534227.jpg
  inflating: hw3data/Flickr8k_Dataset/96399948_b86c61bfe6.jpg
  inflating: hw3data/Flickr8k_Dataset/964197865_0133acaeb4.jpg
  inflating: hw3data/Flickr8k_Dataset/96420612_feb18fc6c6.jpg
  inflating: hw3data/Flickr8k_Dataset/965444691_fe7e85bf0e.jpg
  inflating: hw3data/Flickr8k_Dataset/967719295_3257695095.jpg
  inflating: hw3data/Flickr8k_Dataset/968081289_cdba83ce2e.jpg
  inflating: hw3data/Flickr8k_Dataset/96973080_783e375945.jpg
  inflating: hw3data/Flickr8k_Dataset/96978713_775d66a18d.jpg
  inflating: hw3data/Flickr8k_Dataset/96985174_31d4c6f06d.jpg
  inflating: hw3data/Flickr8k_Dataset/970641406_9a20ee636a.jpg
  inflating: hw3data/Flickr8k_Dataset/97105139_fae46fe8ef.jpg
  inflating: hw3data/Flickr8k_Dataset/972381743_5677b420ab.jpg
  inflating: hw3data/Flickr8k_Dataset/973827791_467d83986e.jpg
  inflating: hw3data/Flickr8k_Dataset/97406261_5eea044056.jpg
  inflating: hw3data/Flickr8k_Dataset/974924582_10bed89b8d.jpg
  inflating: hw3data/Flickr8k_Dataset/975131015_9acd25db9c.jpg
  inflating: hw3data/Flickr8k_Dataset/97577988_65e2eae14a.jpg
  inflating: hw3data/Flickr8k_Dataset/976392326_082dafc3c5.jpg
  inflating: hw3data/Flickr8k_Dataset/97731718_eb7ba71fd3.jpg
  inflating: hw3data/Flickr8k_Dataset/977856234_0d9caee7b2.jpg
  inflating: hw3data/Flickr8k_Dataset/978580450_e862715aba.jpg
  inflating: hw3data/Flickr8k_Dataset/979201222_75b6456d34.jpg
  inflating: hw3data/Flickr8k_Dataset/979383193_0a542a059d.jpg
  inflating: hw3data/Flickr8k_Dataset/98377566_e4674d1ebd.jpg
  inflating: hw3data/Flickr8k_Dataset/985067019_705fe4a4cc.jpg
  inflating: hw3data/Flickr8k_Dataset/987907964_5a06a63609.jpg
  inflating: hw3data/Flickr8k_Dataset/989754491_7e53fb4586.jpg
  inflating: hw3data/Flickr8k_Dataset/989851184_9ef368e520.jpg
  inflating: hw3data/Flickr8k_Dataset/990890291_afc72be141.jpg
  inflating: hw3data/Flickr8k_Dataset/99171998_7cc800ceef.jpg
  inflating: hw3data/Flickr8k_Dataset/99679241_adc853a5c0.jpg
  inflating: hw3data/Flickr8k_Dataset/997338199_7343367d7f.jpg
  inflating: hw3data/Flickr8k_Dataset/997722733_0cb5439472.jpg
  inflating: hw3data/Flickr8k_text/CrowdFlowerAnnotations.txt
  inflating: hw3data/Flickr8k_text/ExpertAnnotations.txt
  inflating: hw3data/Flickr8k_text/Flickr8k.lemma.token.txt
  inflating: hw3data/Flickr8k_text/readme.txt
```

Alternative option if you are using Colab (though using wget, as shown above, works on Colab as well):

- The data is available on google drive. You can access the folder here: https://drive.google.com/drive/folders/1sXWOLkmhpA1KFjVR0VjxGUtzAImIvU39?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).

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)
```

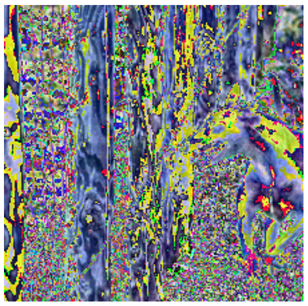


Image Encoder

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)
```

```
img_encoder.eval()
```

```
    (0): BasicBlock(
    (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)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
  )
  (layer4): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(256, 512, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (downsample): Sequential(
        (0): Conv2d(256, 512, kernel_size=(1, 1), stride=(2, 2), bias=False)
        (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      )
    )
    (1): BasicBlock(
      (conv1): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
  )
  (avgpool): AdaptiveAvgPool2d(output_size=(1, 1))
  (fc): Linear(in_features=512, out_features=1000, bias=True)
)
```

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()
```

```
      (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(
      (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)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
  )
  (7): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(256, 512, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (downsample): Sequential(
        (0): Conv2d(256, 512, kernel_size=(1, 1), stride=(2, 2), bias=False)
        (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      )
    )
    (1): BasicBlock(
      (conv1): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
  )
  (8): AdaptiveAvgPool2d(output_size=(1, 1))
)
```

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 (3,224,224)
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):
  img_encoder.eval()
  all_encodings = []

  with torch.no_grad():
    for i, img_name in enumerate(image_list):
      if i % 500 == 0:
          print(f"Encoding image {i}/{len(image_list)}...")

      img = get_image(img_name).unsqueeze(0).to(DEVICE)
      encoded = img_encoder(img)        # [1, 512, 1, 1]
      encoded = encoded[:, :, 0, 0]     # [1, 512]

      all_encodings.append(encoded.squeeze(0))
      all_encodings = torch.stack(all_encodings)
      return all_encodings


    enc_images_train = encode_images(train_list)
    enc_images_train.shape
```

```
    Encoding image 0/6000...
    Encoding image 500/6000...
    Encoding image 1000/6000...
    Encoding image 1500/6000...
    Encoding image 2000/6000...
    Encoding image 2500/6000...
    Encoding image 3000/6000...
    Encoding image 3500/6000...
    Encoding image 4000/6000...
    Encoding image 4500/6000...
    Encoding image 5000/6000...
    Encoding image 5500/6000...
    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):
      #return dictionary image name to 5 captions

      image_descriptions = {}

      with open(filename,'r') as in_file:
        for line in in_file:
          line = line.strip()
          if not line:
            continue

          # splitting to name and caption
          img_id, caption = line.split('\t') #tab
          img_name = img_id.split('#')[0]  # removing #n

          # splitting to tokens
          tokens = caption.lower().split()

          # Adding start and eos tokens - Using EOS as the example is EOS. I do understand that it has to be END,
          # but it seems that EOS is keep getting used throughout the course of assignment.

          # *** if we needed the END token as this part, I would have used  <END> instead of <EOS> on the following line
          tokens = ['<START>'] + tokens + ['<EOS>']

          # adding names if none exists, and if it exists, add captions to dictionary
          if img_name not in image_descriptions:
              image_descriptions[img_name] = []
        image_descriptions[img_name].append(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']
```

```
  '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',
  'a',
  'pink',
  'dress',
  'going',
  'into',
  'a',
  'wooden',
  'cabin',
  '.',
  '<EOS>']]
```

The previous line shoudl return

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

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).

```
#set of tokens
vocab = set()
for captions in descriptions.values():
  for cap in captions:
      vocab.update(cap)

# converting set to sorted list and remove repetitive PAD, START, EOS tokens
vocab = sorted(vocab)
for tok in ["<PAD>", "<START>", "<EOS>"]:
  if tok in vocab:
    vocab.remove(tok)

# early dictionary with special tokens
id_to_word = {0: "<PAD>", 1: "<START>", 2: "<EOS>"}
word_to_id = {"<PAD>": 0, "<START>": 1, "<EOS>": 2}

# fill dictionary to final ver.
for i, word in enumerate(vocab):
  id_to_word[i+3] = word
  word_to_id[word] = i+3
```

```
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)

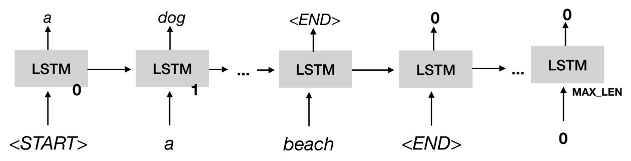
For now, we will just train a model for text generation without conditioning the generator on the image input.

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 an input output pair as follows. For example, consider the sequence

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

We would train the model using the following input/output pair (note both sequences are padded to the right up to MAX_LEN). That is, the output is simply the input shifted left (and with an extra on the righ).

output	**a**	**black**	**dog**	**<EOS>**	**<PAD>**	**<PAD>**	**...**
input	<START>	a	black	dog	<EOS>	<PAD>	...

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 in img_list:
      for caption in descriptions[img]:
        #inital
        input_seq = caption[:-1] # take out <END>
        output_seq = caption[1:] # take out <START>
        input_ids = [word_to_id.get(w, word_to_id['<PAD>']) for w in input_seq]
        output_ids = [word_to_id.get(w, word_to_id['<PAD>']) for w in output_seq]

        # Fitting MAX_LEN
        if len(input_ids) < MAX_LEN:
          input_ids += [word_to_id['<PAD>']] * (MAX_LEN - len(input_ids))
        else:
          input_ids = input_ids[:MAX_LEN]

        if len(output_ids) < MAX_LEN:
          output_ids += [word_to_id['<PAD>']] * (MAX_LEN - len(output_ids))
        else:
          output_ids = output_ids[:MAX_LEN]

        # save
        self.data.append((input_ids, output_ids))

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

  def __getitem__(self,k):
    input_ids, output_ids = self.data[k]
    input_enc = torch.tensor(input_ids, dtype=torch.long)
    output_enc = torch.tensor(output_ids, dtype=torch.long)
    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,   72,  803, 2310, 4013, 6486,  168,   72, 8684, 2310, 3920, 7920,
        7123,   17,    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,  803, 2310, 4013, 6486,  168,   72, 8684, 2310, 3920, 7920, 7123,
          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])
```

```
print(' '.join([id_to_word[x.item()] for x in i]))
print(' '.join([id_to_word[x.item()] for x in o]))
```

```
<START> a black dog is running after a white dog in the snow . <PAD> <PAD> <PAD> <PAD> <PAD> <PAD> <PAD> <PAD> <PAD>
a black dog is running after a white dog in the snow . <EOS> <PAD> <PAD> <PAD> <PAD> <PAD> <PAD> <PAD> <PAD> <PAD> <P
```

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 by zero
    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.

```
for i in range(5):
  train()
```

```
Current average loss: 2.4850564901154297
Current average loss: 2.4859564991154297
Current average loss: 2.4883399917839766
Training loss epoch: 2.4911125136057537
Average accuracy epoch: 0.45
Current average loss: 2.108229875564575
Current average loss: 2.137659011500897
Current average loss: 2.142777024216913
Current average loss: 2.151253263815693
Current average loss: 2.16300842946307
Current average loss: 2.1672180626920596
Current average loss: 2.1727881292733495
Current average loss: 2.180001128247053
Current average loss: 2.185204339086935
Current average loss: 2.1921073002503
Current average loss: 2.194753564321078
Current average loss: 2.2009171143539596
Current average loss: 2.203845213394578
Current average loss: 2.20811114743707
Current average loss: 2.2114898790894535
Current average loss: 2.2156177893390185
Current average loss: 2.219902794782554
Current average loss: 2.2238864616952454
Current average loss: 2.2260051462664863
Training loss epoch: 2.229163643201192
Average accuracy epoch: 0.48
Current average loss: 1.8637607097625732
Current average loss: 1.91989532437655
Current average loss: 1.9281688519378206
Current average loss: 1.9366073220275168
Current average loss: 1.938081006754069
Current average loss: 1.9418832763226446
Current average loss: 1.9479295229554772
Current average loss: 1.9560809975513889
Current average loss: 1.961537620696831
Current average loss: 1.9657674212831504
Current average loss: 1.9708279649932663
Current average loss: 1.978048012323752
Current average loss: 1.9829062664141563
Current average loss: 1.988043038135854
Current average loss: 1.9925324798566286
Current average loss: 1.996210097790082
Current average loss: 1.999333088953446
Current average loss: 2.0037383959617143
Current average loss: 2.009222252841528
Training loss epoch: 2.012354603258769
Average accuracy epoch: 0.52
```

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():
  model.eval()

  with torch.no_grad():
    #has to save in id for model
    input_seq = torch.full((1, MAX_LEN), word_to_id["<PAD>"], dtype=torch.long).to(DEVICE)# <PAD> <PAD> <PAD> ...
    input_seq[0, 0] = word_to_id["<START>"] # <START> <PAD> <PAD> ...
    #print(' '.join([id_to_word[x.item()] for x in input_seq[0]]))
    for t in range(1, MAX_LEN):
      #predicting
      outputs = model(input_seq)

      # Use the last predicted tokens logits
      next_token_logits = outputs[:, t-1, :]
      next_token_id = next_token_logits.argmax(dim=-1).item()
      input_seq[0, t] = next_token_id

      # EOS detection
      if id_to_word[next_token_id] == "<EOS>":
        break

    # Convert ids to words
    #print(' '.join([id_to_word[x.item()] for x in input_seq[0]]))
    generated = [id_to_word[x.item()] for x in input_seq[0] if id_to_word[x.item()] != "<PAD>"]

  return generated
```

```
decoder()
```

```
['<START>',
 'a',
 'man',
 'in',
 'a',
 'blue',
 'shirt',
 'and',
 'tie',
 'sings',
 'into',
 'a',
 'microphone',
 'while',
 'a',
 'woman',
 'stands',
 'next',
 'to',
 'him',
 '.',
 '<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, which takes the distribution as a parameter p.

```
import numpy as np

def sample_decoder():
  model.eval()
  with torch.no_grad():
    input_seq = torch.full((1, MAX_LEN), word_to_id["<PAD>"], dtype=torch.long).to(DEVICE)# <PAD> <PAD> <PAD> ...
    input_seq[0, 0] = word_to_id["<START>"] # <START> <PAD> <PAD> ...

    for i in range(1, MAX_LEN):
      #predicting
      outputs = model(input_seq)

      # Use the last predicted tokens logits, but now with softmax
      next_token_logits = outputs[:, i-1, :]
      probs = torch.softmax(next_token_logits, dim=-1).squeeze(0).cpu().numpy()
      next_token_id = np.random.choice(len(probs), p=probs)
      input_seq[0, i] = next_token_id

      # EOS detection
      if id_to_word[next_token_id] == "<EOS>":
        break

    # Convert ids to words
    #print(' '.join([id_to_word[x.item()] for x in input_seq[0]]))
    generated = [id_to_word[idx.item()] for idx in input_seq[0] if idx.item() != word_to_id["<PAD>"]]

    return generated

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

```
['<START>', 'a', 'man', 'and', 'a', 'woman', 'looking', 'at', 'the', 'camera', '.', '<EOS>']
['<START>', 'girls', 'wear', 'little', 'dive', 'ride', '.', '<EOS>']
['<START>', 'skateboarder', 'on', 'the', 'dirt', 'ramp', '.', '<EOS>']
['<START>', 'a', 'pitbull', 'running', '.', '<EOS>']
['<START>', 'people', 'with', 'dog', 'looking', 'at', 'something', 'in', 'the', 'distance', '.', '<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', 'watching', '.', '
['', '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):
    #setups
    self.img_data = torch.load(open("encoded_images_train.pt", 'rb'))  # [num_images, 512]
    self.img_name_to_id = {img_name: idx for idx, img_name in enumerate(img_list)}
    self.data = []

    # for all images
    for img_name in img_list:
      img_idx = self.img_name_to_id[img_name]
      captions = descriptions[img_name]
      # for all captions
      for cap in captions:
        # Encode input sequence: <START> + caption[:-1] (without <EOS>)
        input_tokens = cap[:-1]  # takes out <EOS>
        input_ids = [word_to_id[token] for token in input_tokens]

        # Fitting MAX_LEN
        if len(input_ids) < MAX_LEN:
          input_ids += [word_to_id["<PAD>"]] * (MAX_LEN - len(input_ids))
        else:
          input_ids = input_ids[:MAX_LEN]

        # Encoding caption (including <EOS>)
        output_tokens = cap[1:]
        output_ids = [word_to_id[token] for token in output_tokens]
        if len(output_ids) < MAX_LEN:
          output_ids += [word_to_id["<PAD>"]] * (MAX_LEN - len(output_ids))
        else:
          output_ids = output_ids[:MAX_LEN]

        # save
        self.data.append((img_idx, input_ids, output_ids))

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

  def __getitem__(self, k):
    img_idx, input_ids, output_ids = self.data[k]
    img_encoding = self.img_data[img_idx]  # [512]
    input_enc = torch.tensor(input_ids, dtype=torch.long)
    output_enc = torch.tensor(output_ids, dtype=torch.long)
    return img_encoding, input_enc, output_enc
```

```
joint_data = CaptionAndImage(train_list)
img, i, o = joint_data[3]
img.shape # should return torch.Size([512])
```

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

```
i.shape # should return torch.Size([40])
print(' '.join([id_to_word[x.item()] for x in i]))
```

```
<START> two dogs play together in the snow . <PAD> <PAD> <PAD> <PAD> <PAD> <PAD> <PAD> <PAD> <PAD> <PAD> <PAD> <PAD>
```

```
o.shape # should return torch.Size([40])
print(' '.join([id_to_word[x.item()] for x in o]))
```

```
two dogs play together in the snow . <EOS> <PAD> <PAD> <PAD> <PAD> <PAD> <PAD> <PAD> <PAD> <PAD> <PAD> <PAD> <PAD> <P
```

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
EMBED_DIM = 512
HIDDEN_DIM = 512
MAX_LEN = 40

class CaptionGeneratorModel(nn.Module):
  def __init__(self):
    super(CaptionGeneratorModel, self).__init__()
    self.embedding = nn.Embedding(vocab_size, EMBED_DIM)
    self.lstm = nn.LSTM(input_size=EMBED_DIM + HIDDEN_DIM, hidden_size=HIDDEN_DIM, num_layers=1, batch_first=True)
    self.output = nn.Linear(HIDDEN_DIM, vocab_size)

  def forward(self, img, input_seq):
    embedded = self.embedding(input_seq)
    img_expanded = img.unsqueeze(1).expand(-1, embedded.size(1), -1)
    hidden_lstminp = torch.cat([embedded, img_expanded], dim=2)
    hidden_lstmout, _ = self.lstm(hidden_lstminp)  # hidden_lstmout = [batch_size, MAX_LEN, HIDDEN_DIM]
    out = self.output(hidden_lstmout)
    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 by zero
    print(f"Training loss epoch: {epoch_loss}")
    print(f"Average accuracy epoch: {epoch_accuracy:.2f}")
```

```
for i in range(5):
  train()
```

```
Current average loss: 2.0972721007332873
```

```
    Current average loss: 2.0972721007332873
    Current average loss: 2.10528711464714
    Current average loss: 2.1049060937472412
    Current average loss: 2.119456465610725
    Current average loss: 2.1237434323337827
    Current average loss: 2.1235651106024944
    Current average loss: 2.1279107681969727
    Current average loss: 2.1335030047927925
    Current average loss: 2.1392337250542806
    Current average loss: 2.1380334987519114
    Current average loss: 2.1419264209161293
    Current average loss: 2.1440183050718606
    Current average loss: 2.145570336793849
    Current average loss: 2.1470345946489218
    Current average loss: 2.1490379999310876
    Current average loss: 2.1501765699683744
    Current average loss: 2.151785957819882
    Training loss epoch: 2.152059738032023
    Average accuracy epoch: 0.49
    Current average loss: 2.0039050579071045
    Current average loss: 1.9018872634019002
    Current average loss: 1.9158400024347637
    Current average loss: 1.9198654581145986
    Current average loss: 1.9202609109759627
    Current average loss: 1.9240174621878983
    Current average loss: 1.9298773605295902
    Current average loss: 1.932319298960513
    Current average loss: 1.9349953720483293
    Current average loss: 1.940287183180501
    Current average loss: 1.9443795214166175
    Current average loss: 1.9475878204246957
    Current average loss: 1.9497524526692946
    Current average loss: 1.9534378349551964
    Current average loss: 1.9559640502351083
    Current average loss: 1.9591004730938752
    Current average loss: 1.960740864984249
    Current average loss: 1.9614090374958368
    Current average loss: 1.963777265768459
    Training loss epoch: 1.9652198223114015
    Average accuracy epoch: 0.52
```

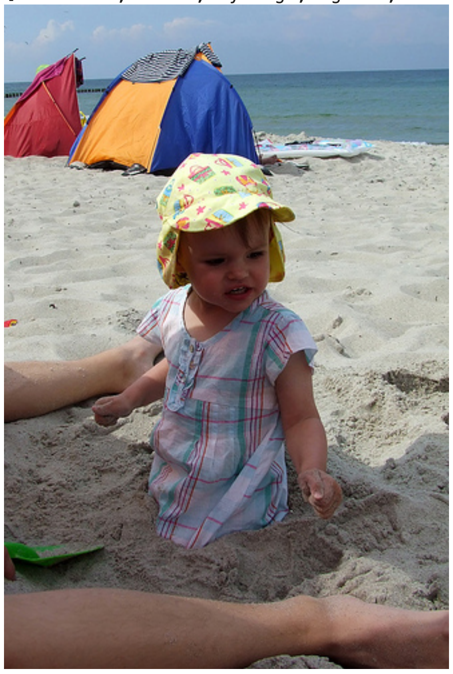
TODO: Testing the model: Rewrite the greedy decoder from above to take an encoded image representation as input.

```
def sample_decoder(img):
  model.eval()

  #check for dimension
  if img.dim() == 1:
    img = img.unsqueeze(0).to(DEVICE)
  else:
    img = img.to(DEVICE)

  with torch.no_grad():
    # changing from <START> <PAD> <PAD> ... to just <START> and appending new values due to different max len size
    input_seq = torch.tensor([[word_to_id["<START>"]]], dtype=torch.long).to(DEVICE)

    for _ in range(MAX_LEN):
      # prediction
      outputs = model(img, input_seq)  # [1, seq_len, vocab_size]

      # logits
      next_token_logits = outputs[:, -1, :]  # [1, vocab_size]
      probs = torch.softmax(next_token_logits, dim=-1)
      probs = probs / probs.sum()  # normalizing
      probs_np = probs.squeeze(0).cpu().numpy()

      next_token_id = np.random.choice(len(probs_np), p=probs_np)
      next_word = id_to_word[next_token_id]

      # Append predicted token to input sequence
      input_seq = torch.cat([input_seq, torch.tensor([[next_token_id]], device=DEVICE)], dim=1)

      # Stop if the last token saved is <EOS>
      if next_word == "<EOS>":
        break

    generated = [id_to_word[idx.item()] for idx in input_seq[0] if idx.item() != word_to_id["<PAD>"]]


  return generated
```

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 = sample_decoder(encoded_img)
print(caption)
raw_img
```

```
['<START>', 'the', 'young', 'girl', 'swing', 'near', 'the', 'ocean', '.', '<EOS>']
```



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.

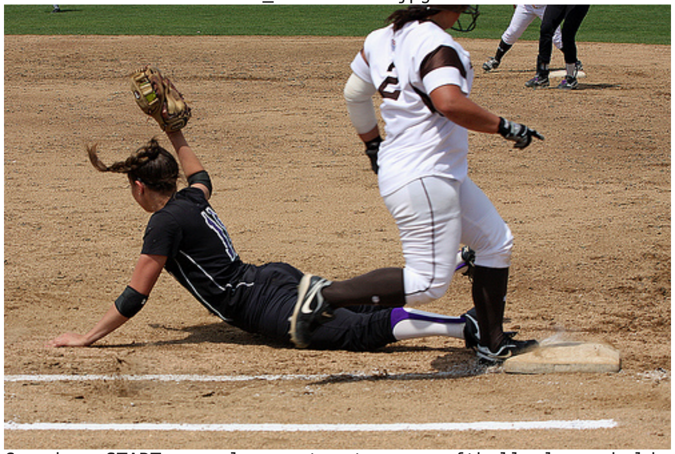
```
def img_beam_decoder(n, img):
    """
    Beam search decoder for image captioning.
    n: beam width (e.g., 3 or 5)
    img: torch.Tensor [1, 512] image encoding
    """
    model.eval()
    with torch.no_grad():
        # Initialize beam with start token
        start_id = word_to_id["<START>"]

        beam = [(1.0, [start_id])]  # (probability, sequence)

        for _ in range(MAX_LEN):
            candidates = []

            for prob, seq in beam:
                # Convert current partial sequence to tensor
                input_seq = torch.tensor([seq], dtype=torch.long).to(DEVICE)
                outputs = model(img, input_seq)
                next_token_logits = outputs[:, -1, :]  # last token
                probs = torch.softmax(next_token_logits, dim=-1).squeeze(0).cpu().numpy()

                # top n next tokens
                top_indices = np.argsort(probs)[-n:][::-1]

                # Create n new candidates (prob * next_prob, seq + [next_word])
                for idx in top_indices:
                    new_prob = prob * probs[idx]
                    new_seq = seq + [idx]
                    candidates.append((new_prob, new_seq))

            # Sort all candidates by probability and keep top-n
            beam = sorted(candidates, key=lambda x: x[0], reverse=True)[:n]

        # Return the most probable sequence
        best_seq = beam[0][1]

        # caption post process to end at first EOS
        # Generation is done, therefore this should be fine based on the instructions? for producing best perfect li
        caption = [id_to_word[idx] for idx in best_seq]
        if "<EOS>" in caption:
          caption = caption[:caption.index("<EOS>")+1]

        return caption
```

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.

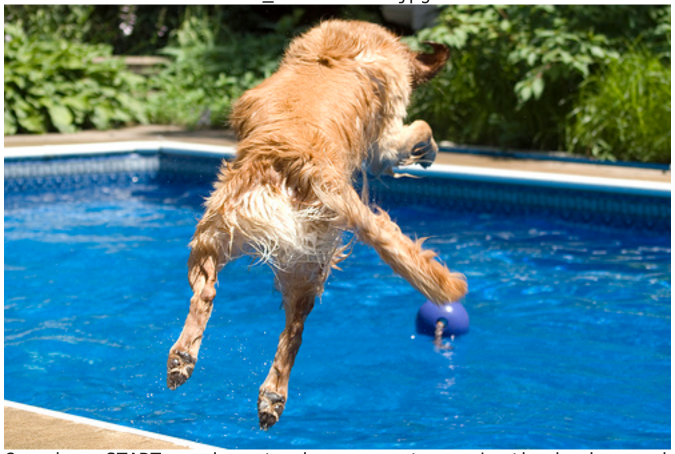
```
from IPython.display import display

#test images randomly selected
```

```
#test images randomly selected
test_imgs = [dev_list[11], dev_list[27], dev_list[99]]

#encoding
encode_test_imgs = encode_images(test_imgs)

#loop for 3 development images
for i, img_name in enumerate(test_imgs):
  #showing the image itself
  print("")
  print(f"=== IMAGE {i+1}: {img_name} ===")
  image = PIL.Image.open(os.path.join(IMG_PATH, test_imgs[i]))
  display(image)

  #
  enc_img = encode_test_imgs[i].unsqueeze(0).to(DEVICE)

  # Greedy decoding
  greedy = sample_decoder(enc_img)
  print("Greedy:", " ".join(greedy))

  # Beam search n=3
  beam3 = img_beam_decoder(3, enc_img)
  print("Beam search (n=3):", " ".join(beam3))

  # Beam search n=5
  beam5 = img_beam_decoder(5, enc_img)
  print("Beam search (n=5):", " ".join(beam5))
```

```
Encoding image 0/3...

=== IMAGE 1: 3484841598_e26ee96aab.jpg ===
```



```
Greedy: <START> people on street on a softball player holds the ball as people watch . <EOS>
Beam search (n=3): <START> a man in a blue shirt is riding a baseball during a softball game . <EOS>
Beam search (n=5): <START> a man in a blue shirt is riding a horse during a softball game . <EOS>

=== IMAGE 2: 1178705300_c224d9a4f1.jpg ===
```



```
Greedy: <START> a dog stands near water . in the background . <EOS>
```