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.

NOTE: I developed this on my local (VSCODE), and uploaded to colab for PDF Conversion. Hence, the cell number while running doesn't appear here

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)
```

```
mps
```

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/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).

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_sp25_hw3/hw3data.zip

```
# OPTIONAL (if not using Colab and the data in Google Drive): Download the data
!wget https://storage.googleapis.com/4705_sp25_hw3/hw3data.zip
```

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

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.t
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)
```

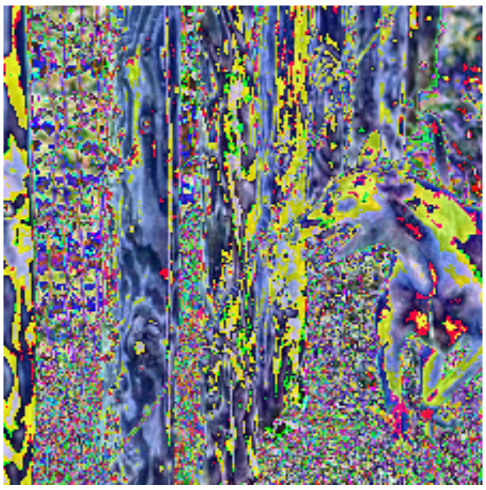


Image Encoder

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()
```

```
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)
```

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 GP
img_encoder.eval()
```

```
    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)
```

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 r
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):
    #TODO....
    with torch.no_grad():

        prep_images = [get_image(im).unsqueeze(0).to(DEVICE) for im in image_li
        encoded = [img_encoder(pre_im) for pre_im in prep_images]
        return torch.stack([im[0,:,0,0] for im in encoded])

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:
        # todo
        for line in in_file.read().split('\n'):
                line = line.split('\t')
                if(len(line)!=2): continue
                file = line[0][:-2] # neglect '#number' where number ranges fro
                sentence = ["<START>"] + line[1].lower().split(" ") + ["<EOS>"]
                image_descriptions[file] = image_descriptions.get(file,[])+[ser

    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.toke
```

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

```
  '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',
  'a',
  'pink',
  'dress',
  'going',
  'into',
  'a',
  'wooden',
  'cabin',
  '.',
```

The previous line shoudl return

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

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).

```
all_word = set(word for val in descriptions.values() for sentence in val for wo
all_word-={"PAD","<START>","<EOS>"}
all_word = list(all_word)
all_word.sort()

id_to_word = {} #todo
word_to_id = {} # todo

id_to_word[0] = "<PAD>"
id_to_word[1] = "<START>"
id_to_word[2] = "<EOS>"

for k,v in id_to_word.items():
    word_to_id[v] = k

for id in range(len(all_word)):
    id_to_word[id+3] = all_word[id]
    word_to_id[all_word[id]] = id+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)

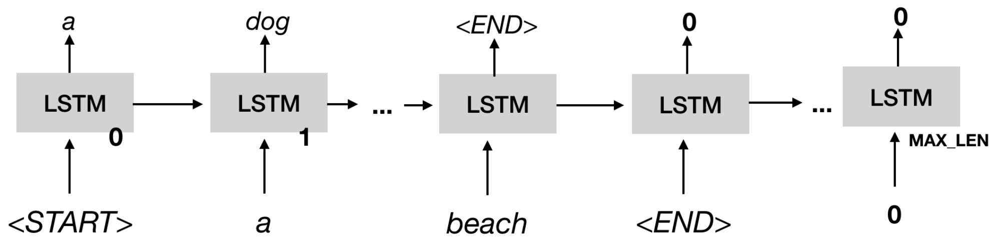
For now, we will just train a model for text generation without conditioning the generator on the image input.

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 de
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, batc
        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):

        # TODO

        self.data = []

        for img in img_list:
            for desc in descriptions[img]:
                input_enc = torch.tensor([word_to_id[word] for word in desc]+[0
                self.data+=[(input_enc,torch.cat([input_enc[1:], torch.tensor(

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

    def __getitem__(self,k):

        #TODO

        # input_enc = self.data[k][0]
        # output_enc = self.data[k][1]

        return self.data[k]
```

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, 792
         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])
```

for the output:

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

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

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

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

```
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])
```

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 !=
    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()
# 5th Epoch: Average accuracy epoch: 0.52
```

```
Current average loss: 2.438460574185759
Current average loss: 2.4433533645675567
Current average loss: 2.4459573841729694
Current average loss: 2.4466831044701807
Current average loss: 2.4540460574046503
Current average loss: 2.457086763159681
Current average loss: 2.458725645111038
Current average loss: 2.4630158573361984
Current average loss: 2.4648673395431606
Current average loss: 2.467718762493793
Current average loss: 2.471306114176356
Current average loss: 2.4742407134816617
Current average loss: 2.476180921562905
Current average loss: 2.478848748456584
Current average loss: 2.4811507481326665
```

```
Training loss epoch: 2.4822630285898843
Average accuracy epoch: 0.45
Current average loss: 2.092463731765747
Current average loss: 2.1030757557047477
Current average loss: 2.1215660720322265
Current average loss: 2.139678180019721
Current average loss: 2.1507043065572913
Current average loss: 2.1551567901394324
Current average loss: 2.16087742534136
Current average loss: 2.165034841028668
Current average loss: 2.1714852700072726
Current average loss: 2.1776029669352033
Current average loss: 2.1828307126547313
Current average loss: 2.1875738521579393
Current average loss: 2.192057931552223
Current average loss: 2.196483998368283
Current average loss: 2.201342027243506
Current average loss: 2.204642669587513
Current average loss: 2.2087018907554143
Current average loss: 2.2136073803495195
Current average loss: 2.2188274039618507
Training loss epoch: 2.221678070449829
Average accuracy epoch: 0.48
Current average loss: 1.883617639541626
Current average loss: 1.9109050318746283
Current average loss: 1.912457422237491
Current average loss: 1.9214032480487
Current average loss: 1.9291081039091
Current average loss: 1.93571649649424
Current average loss: 1.9392599030064663
Current average loss: 1.9438337694731316
Current average loss: 1.952475671911061
Current average loss: 1.9599794470906655
Current average loss: 1.9632454123292173
Current average loss: 1.9691189894559273
Current average loss: 1.9729319625254178
Current average loss: 1.9773645536611852
Current average loss: 1.9816946927178851
Current average loss: 1.9858733899270908
Current average loss: 1.9918944195313129
Current average loss: 1.9963420477423648
Current average loss: 2.0007738079513726
Training loss epoch: 2.0037379722595214
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():
    # TODO
    index = 0
    sentence = torch.tensor([[1]+[0]*(MAX_LEN-1)]).to(DEVICE)

    while(index+1<MAX_LEN):

        output = model(sentence)
        next_token_logits = output[0, index]

        next_token_id = torch.argmax(next_token_logits).item()
        sentence[0,index+1] = next_token_id # replace the next index with the p

        if next_token_id == 2: break # <EOS>
        index+=1

    return [id_to_word[i] for i in sentence[0].tolist() if i!=0]


decoder()
```

```
['<START>',
 'a',
 'man',
 'in',
 'a',
 'blue',
 'shirt',
 'is',
 'operating',
 'a',
 'fiery',
 'furnace',
 '.',
 '<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():
    # TODO

    index = 0
    sentence = torch.tensor([[1]+[0]*(MAX_LEN-1)]).to(DEVICE)

    while(index+1<MAX_LEN):

        output = model(sentence)
        prob = torch.softmax(output[0, index],dim=0).cpu().detach().numpy()

        next_token_id = np.random.choice(len(prob), p=prob) #random prob instea
        sentence[0,index+1] = next_token_id

        if next_token_id == 2: break
        index+=1

    return [id_to_word[i] for i in sentence[0].tolist() if i!=0]

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

```
['<START>', 'a', 'child', 'is', 'sitting', 'in', 'a', 'hospital', 'bed', '.
['<START>', 'a', 'girl', 'in', 'water-wear', 'has', 'skipped', 'a', 'tattoo
['<START>', 'two', 'children', 'play', 'with', 'adults', 'and', 'covered',
['<START>', 'four', 'boys', 'that', 'perform', 'on', 'a', 'stage', 'in', 'a
['<START>', 'a', 'man', 'takes', 'a', 'picture', 'of', 'a', 'funny', 'face'
```

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
['', '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')) # sugg
        self.img_name_to_id = dict([(i,j) for (j,i) in enumerate(img_list)])

        self.data = []


        # TODO
        for img in img_list: # img is file name
            for desc in descriptions[img]: # desc is each description from #0 t
                input_enc = torch.tensor([word_to_id[word] for word in desc]+[0
                self.data+=[(self.img_data[self.img_name_to_id[img]],input_enc,


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

    def __getitem__(self,k):
        # TODO COMPLETE THIS METHOD
        # img_data = self.data[k][0]
        # input_enc = self.data[k][1]
        # output_enc = self.data[k][2]

        # return img_data, input_enc, output_enc
        return self.data[k]


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

```
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, bat
        self.output = nn.Linear(512,vocab_size)

    def forward(self, img, input_seq):

        # TODO COMPLETE THIS METHOD
        emb = self.embedding(input_seq)
        concat = torch.cat((img.unsqueeze(1).expand(-1,emb.size(1),-1), self.en

        hidden = self.lstm(concat)
        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 !=
    print(f"Training loss epoch: {epoch_loss}")
    print(f"Average accuracy epoch: {epoch_accuracy:.2f}")


for i in range(5):
    train()
# 5th Epoch: Average accuracy epoch: 0.51
```

```
Current average loss: 2.4145027763423728
Current average loss: 2.4157518742982287
Current average loss: 2.410931154163536
Current average loss: 2.4089833712220785
Current average loss: 2.410865110439513
Current average loss: 2.4142357417557867
Current average loss: 2.4078657095228526
Current average loss: 2.4054418384254754
Current average loss: 2.405208136253634
Current average loss: 2.4071007679145997
Current average loss: 2.4072377052241154
Current average loss: 2.4067217679129254
Current average loss: 2.4057191169555785
Current average loss: 2.4037923409296975
Current average loss: 2.4037319448399304
Current average loss: 2.404001814997375
Training loss epoch: 2.4039426300048827
Average accuracy epoch: 0.46
```

```
Current average loss: 2.156339168548584
Current average loss: 2.1424163591743697
Current average loss: 2.141384867886406
Current average loss: 2.142565762481816
Current average loss: 2.1531079957015495
Current average loss: 2.1601160215046593
Current average loss: 2.159141581387766
Current average loss: 2.1595054043512714
Current average loss: 2.159418835473269
Current average loss: 2.159124686371341
Current average loss: 2.162611331258501
Current average loss: 2.161437443989607
Current average loss: 2.161784289877778
Current average loss: 2.1659488216168508
Current average loss: 2.1671965565535105
Current average loss: 2.1676564265059914
Current average loss: 2.1673785949185222
Current average loss: 2.168056263494744
Current average loss: 2.167906323649498
Training loss epoch: 2.1682262396494547
Average accuracy epoch: 0.49
Current average loss: 1.943197250366211
Current average loss: 1.91210138089586
Current average loss: 1.9231564334375941
Current average loss: 1.9387202698526984
Current average loss: 1.9395250417942418
Current average loss: 1.9398913297824516
Current average loss: 1.942644141676422
Current average loss: 1.947590694617953
Current average loss: 1.953921649786417
Current average loss: 1.9568125172275286
Current average loss: 1.9577241369060703
Current average loss: 1.9612382127413632
Current average loss: 1.9646544453504182
Current average loss: 1.966092595863489
Current average loss: 1.96924557437393
Current average loss: 1.972521802649984
Current average loss: 1.974568586957075
Current average loss: 1.9791186448758242
Current average loss: 1.9810537560524906
Training loss epoch: 1.9831674019495646
```

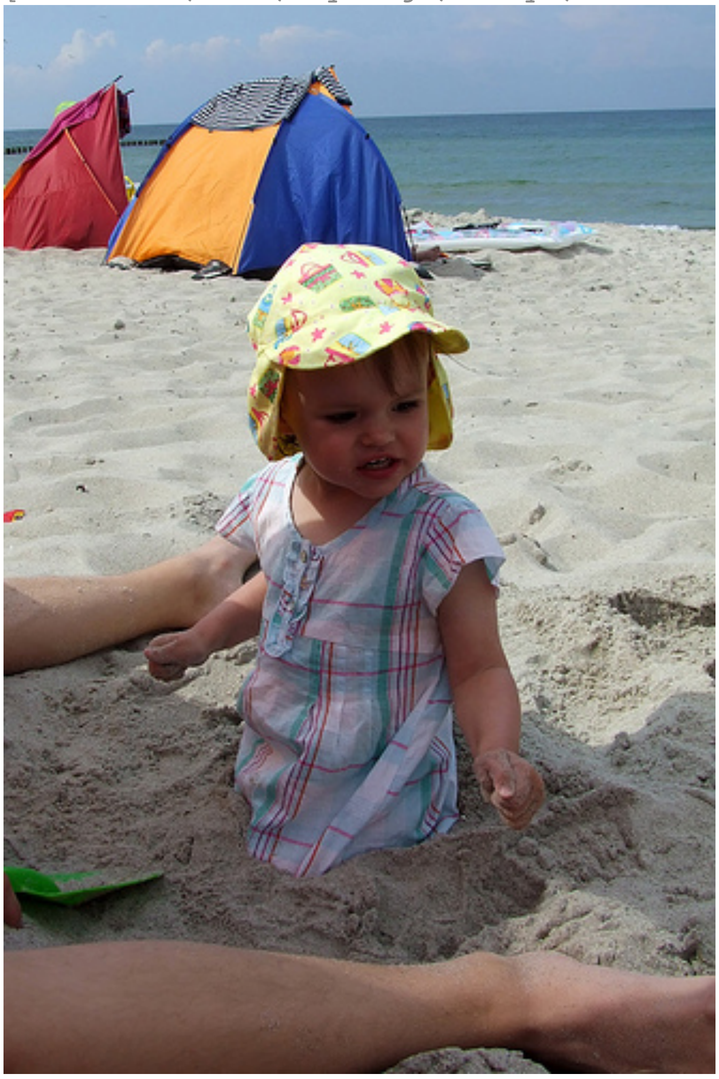
TODO: Testing the model: Rewrite the greedy decoder from above to take an encoded image representation as input.

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

    index = 0
    sentence = torch.tensor([[1]+[0]*(MAX_LEN-1)])
    sentence = sentence.to(DEVICE)
    while(index+1<MAX_LEN):
        output = model(img.unsqueeze(0),sentence)
        next_token_logits = output[0, index]

        next_token_id = torch.argmax(next_token_logits).item()
        sentence[0,index+1] = next_token_id

        if next_token_id == 2: break
        index+=1

    return [id_to_word[i] for i in sentence[0].tolist() if i!=0]
```

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
```

```
['<START>', 'a', 'young', 'boy', 'wearing', 'a', 'blue', 'shirt', 'and', 'b
```



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.

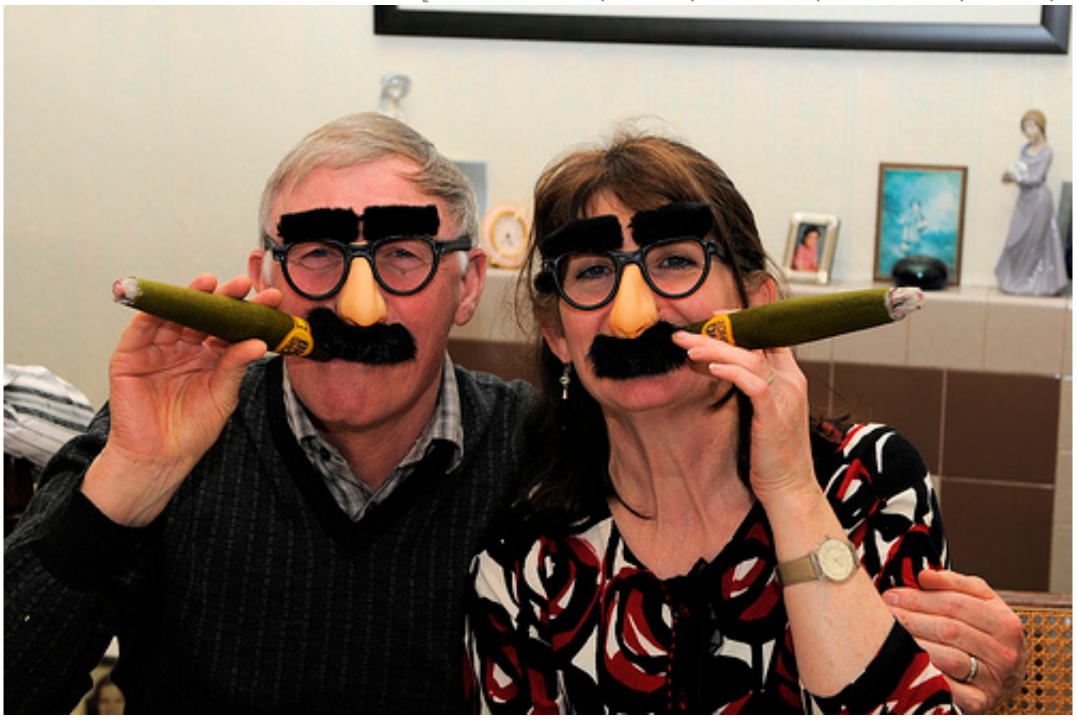
Once `MAX_LEN` has been reached, return the most likely sequence out of the current n.

```
def img_beam_decoder(n, img):

    # TODO: Complete this method

    index = 0
    initial_sentence = torch.tensor([[1]+[0]*(MAX_LEN-1)])
    initial_sentence = initial_sentence.to(DEVICE)
    probseq = [(1,initial_sentence)]

    while(index+1<MAX_LEN):
        nowseq = []
        for oriprob,sentence in probseq:
            output = model(img.unsqueeze(0),sentence)
            prob = torch.softmax(output[0, index],dim=0)

            val, ind = torch.topk(prob, k=n)

            for i in range(n):
                next_token_id = ind[i].item()
                newsentence = sentence.clone()
                newsentence[0,index+1] = next_token_id
                nowseq += [(oriprob * val[i].item(), newsentence)]

        nowseq.sort(reverse=True)
        probseq = nowseq[:n] # keep top n
        index+=1

    probseq.sort(reverse=True)

    final = []
    for i in probseq[0][1][0].tolist(): # best sequence
        if i==0: continue # skip <PAD>
        final+=[id_to_word[i]]
        if i==2: break # stop after <EOS>
    return final
```

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.

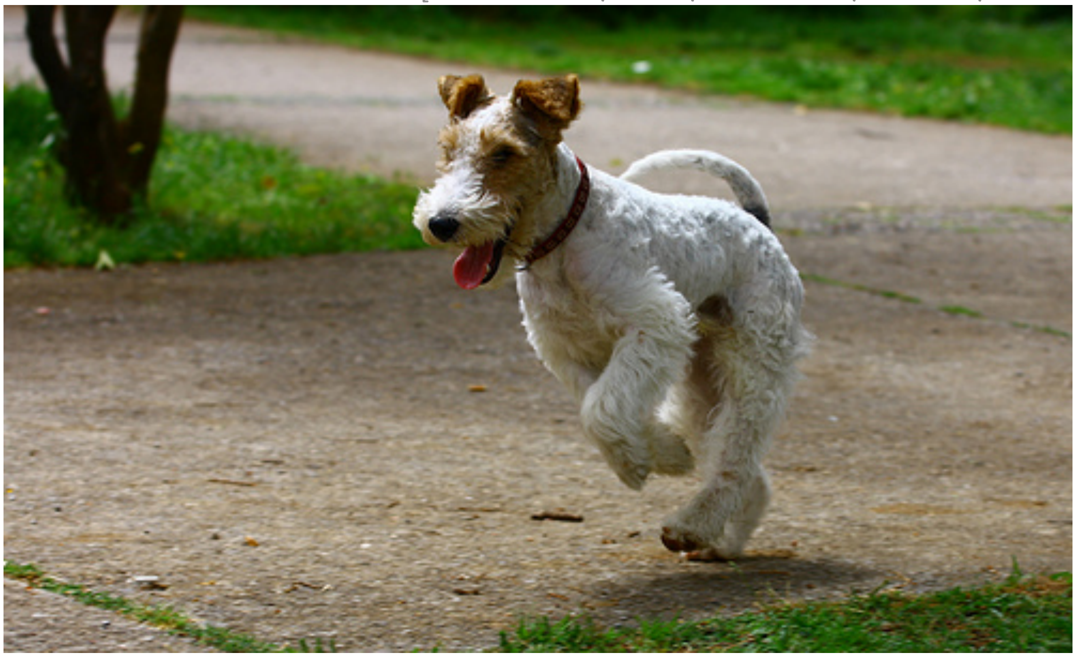
```
import random

img_list = []
def decoder_comparison():
    raw_img = PIL.Image.open(os.path.join(IMG_PATH, dev_list[random.randint(0,1
    preprocessed_img = preprocess(raw_img).to(DEVICE)
    encoded_img = img_encoder(preprocessed_img.unsqueeze(0)).reshape((512))


    print("Greedy decoder: ",greedy_decoder(img))
    print("Beam search at n = 3: ",img_beam_decoder(3,encoded_img))
    print("Beam search at n = 5: ",img_beam_decoder(5,encoded_img))
    return raw_img


decoder_comparison()
```

```
Greedy decoder:  ['<START>', 'two', 'dogs', 'are', 'playing', 'on', 'a', 's
Beam search at n = 3:  ['<START>', 'a', 'man', 'with', 'a', 'beard', 'and',
Beam search at n = 5:  ['<START>', 'a', 'man', 'with', 'a', 'beard', 'and',
```



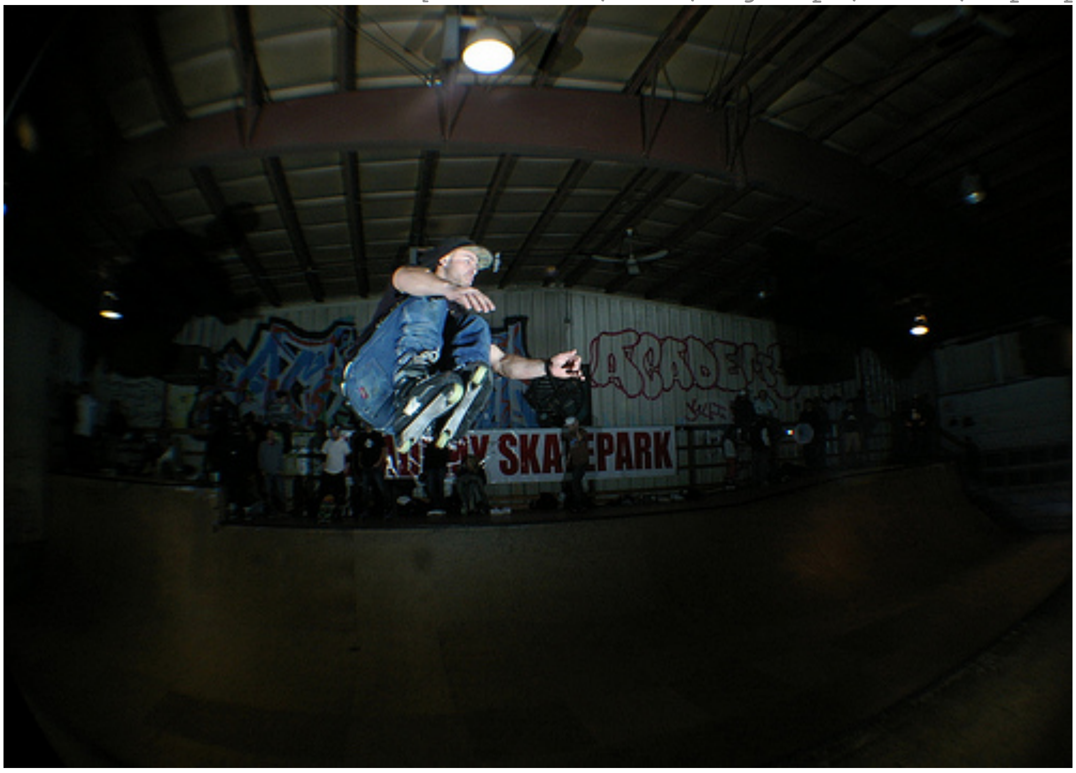
```
decoder_comparison()
```

```
Greedy decoder:  ['<START>', 'two', 'dogs', 'are', 'playing', 'on', 'a', 's
Beam search at n = 3:  ['<START>', 'a', 'white', 'dog', 'is', 'running', 'o
Beam search at n = 5:  ['<START>', 'a', 'brown', 'and', 'white', 'dog', 'is
```



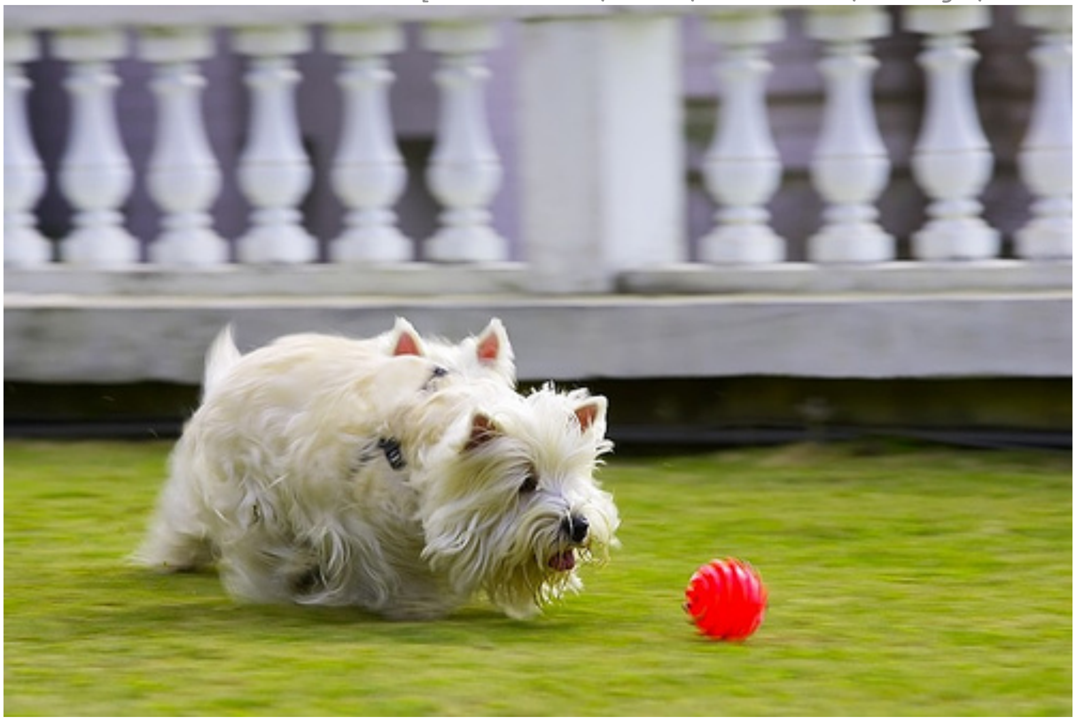
```
decoder_comparison()
```

```
Greedy decoder:  ['<START>', 'two', 'dogs', 'are', 'playing', 'on', 'a', 's
Beam search at n = 3:  ['<START>', 'a', 'group', 'of', 'people', 'in', 'the
Beam search at n = 5:  ['<START>', 'a', 'group', 'of', 'people', 'on', 'a',
```



```
decoder_comparison()
```

```
Greedy decoder:  ['<START>', 'two', 'dogs', 'are', 'playing', 'on', 'a', 's
Beam search at n = 3:  ['<START>', 'a', 'small', 'white', 'dog', 'is', 'jum
Beam search at n = 5:  ['<START>', 'a', 'white', 'dog', 'and', 'a', 'white'
```



Start coding or generate with AI.