# 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)
```

```
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

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
```

```
--2025-10-24 15:48:38--  https://storage.googleapis.com/4705_sp25_hw3/hw3data.zip
Resolving storage.googleapis.com (storage.googleapis.com)... 142.250.65.251, 142.251.32.123, 142.250.80.91, ...
Connecting to storage.googleapis.com (storage.googleapis.com)|142.250.65.251|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 1115435617 (1.0G) [application/zip]
Saving to: ‘hw3data.zip’

hw3data.zip         100%[===================>]   1.04G  7.22MB/s    in 2m 22s

2025-10-24 15:51:00 (7.51 MB/s) - ‘hw3data.zip’ saved [1115435617/1115435617]
```

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

```
Archive:  hw3data.zip
   creating: hw3data
  inflating: hw3data/Flickr8k.token.txt
   creating: hw3data/Flickr8k_Dataset
   creating: hw3data/Flickr8k_text
  inflating: hw3data/Flickr_8k.devImages.txt
  inflating: hw3data/Flickr_8k.testImages.txt
  inflating: hw3data/Flickr_8k.trainImages.txt
```

```
  inflating: hw3data/Flickr8k_Dataset/1000268201_693b08cb0e.jpg
  inflating: hw3data/Flickr8k_Dataset/1001773457_577c3a7d70.jpg
  inflating: hw3data/Flickr8k_Dataset/1002674143_1b742ab4b8.jpg
  inflating: hw3data/Flickr8k_Dataset/1003163366_44323f5815.jpg
  inflating: hw3data/Flickr8k_Dataset/1007129816_e794419615.jpg
  inflating: hw3data/Flickr8k_Dataset/1007320043_627395c3d8.jpg
  inflating: hw3data/Flickr8k_Dataset/1009434119_febe49276a.jpg
  inflating: hw3data/Flickr8k_Dataset/1012212859_01547e3f17.jpg
  inflating: hw3data/Flickr8k_Dataset/1015118661_980735411b.jpg
  inflating: hw3data/Flickr8k_Dataset/1015584366_dfcec3c85a.jpg
  inflating: hw3data/Flickr8k_Dataset/101654506_8eb26cfb60.jpg
  inflating: hw3data/Flickr8k_Dataset/101669240_b2d3e7f17b.jpg
  inflating: hw3data/Flickr8k_Dataset/1016887272_03199f49c4.jpg
  inflating: hw3data/Flickr8k_Dataset/1019077836_6fc9b15408.jpg
  inflating: hw3data/Flickr8k_Dataset/1019604187_d087bf9a5f.jpg
  inflating: hw3data/Flickr8k_Dataset/1020651753_06077ec457.jpg
  inflating: hw3data/Flickr8k_Dataset/1022454332_6af2c1449a.jpg
  inflating: hw3data/Flickr8k_Dataset/1022454428_b6b660a67b.jpg
  inflating: hw3data/Flickr8k_Dataset/1022975728_75515238d8.jpg
  inflating: hw3data/Flickr8k_Dataset/102351840_323e3de834.jpg
  inflating: hw3data/Flickr8k_Dataset/1024138940_f1fefbdce1.jpg
  inflating: hw3data/Flickr8k_Dataset/102455176_5f8ead62d5.jpg
  inflating: hw3data/Flickr8k_Dataset/1026685415_0431cbf574.jpg
  inflating: hw3data/Flickr8k_Dataset/1028205764_7e8df9a2ea.jpg
  inflating: hw3data/Flickr8k_Dataset/1030985833_b0902ea560.jpg
  inflating: hw3data/Flickr8k_Dataset/103106960_e8a41d64f8.jpg
  inflating: hw3data/Flickr8k_Dataset/103195344_5d2dc613a3.jpg
  inflating: hw3data/Flickr8k_Dataset/103205630_682ca7285b.jpg
  inflating: hw3data/Flickr8k_Dataset/1032122270_ea6f0beedb.jpg
  inflating: hw3data/Flickr8k_Dataset/1032460886_4a598ed535.jpg
  inflating: hw3data/Flickr8k_Dataset/1034276567_49bb87c51c.jpg
  inflating: hw3data/Flickr8k_Dataset/104136873_5b5d41be75.jpg
  inflating: hw3data/Flickr8k_Dataset/1042020065_fb3d3ba5ba.jpg
  inflating: hw3data/Flickr8k_Dataset/1042590306_95dea0916c.jpg
  inflating: hw3data/Flickr8k_Dataset/1045521051_108ebc19be.jpg
  inflating: hw3data/Flickr8k_Dataset/1048710776_bb5b0a5c7c.jpg
  inflating: hw3data/Flickr8k_Dataset/1052358063_eae6744153.jpg
  inflating: hw3data/Flickr8k_Dataset/105342180_4d4a40b47f.jpg
  inflating: hw3data/Flickr8k_Dataset/1053804096_ad278b25f1.jpg
  inflating: hw3data/Flickr8k_Dataset/1055623002_8195a43714.jpg
  inflating: hw3data/Flickr8k_Dataset/1055753357_4fa3d8d693.jpg
  inflating: hw3data/Flickr8k_Dataset/1056249424_ef2a2e041c.jpg
  inflating: hw3data/Flickr8k_Dataset/1056338697_4f7d7ce270.jpg
  inflating: hw3data/Flickr8k_Dataset/1056359656_662cee0814.jpg
  inflating: hw3data/Flickr8k_Dataset/1056873310_49c665eb22.jpg
  inflating: hw3data/Flickr8k_Dataset/1057089366_ca83da0877.jpg
  inflating: hw3data/Flickr8k_Dataset/1057210460_09c6f4c6c1.jpg
  inflating: hw3data/Flickr8k_Dataset/1057251835_6ded4ada9c.jpg
  inflating: hw3data/Flickr8k_Dataset/106490881_5a2dd9b7bd.jpg
  inflating: hw3data/Flickr8k_Dataset/106514190_bae200f463.jpg
```

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])
])
```

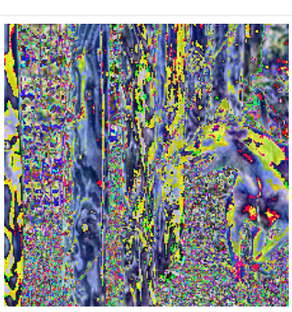
The resulting images, after preprocessing, are (3,224,244) tensors, where the first dimension represents the three color channels, R,G,B).

```
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)
```

```
1.7%Downloading: "https://download.pytorch.org/models/resnet18-f37072fd.pth" to /Users/soumilbaldota/.cache/torch/hu
100.0%
```

```
img_encoder.eval()
```

```
ResNet(
  (conv1): Conv2d(3, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3), bias=False)
  (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (relu): ReLU(inplace=True)
  (maxpool): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1, ceil_mode=False)
  (layer1): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
    (1): BasicBlock(
      (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
  )
  (layer2): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(64, 128, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (downsample): Sequential(
        (0): Conv2d(64, 128, kernel_size=(1, 1), stride=(2, 2), bias=False)
        (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      )
    )
    (1): BasicBlock(
      (conv1): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
  )
  (layer3): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(128, 256, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (downsample): Sequential(
        (0): Conv2d(128, 256, kernel_size=(1, 1), stride=(2, 2), bias=False)
        (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      )
    )
    (1): BasicBlock(
      (conv1): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
```

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

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

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

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

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

Let's try the encoder.

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

```
preprocessed_image = get_image(train_list[0])
encoded = img_encoder(preprocessed_image.unsqueeze(0).to(DEVICE)) # unsqueeze required to add batch dim (3,224,224)
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.

```
import numpy as np
def encode_images(image_list):
    encodings = []
    img_encoder.eval()
    with torch.no_grad():
        for img_name in image_list:
            preprocessed_image = get_image(img_name).unsqueeze(0).to(DEVICE)
            encoded = img_encoder(preprocessed_image)[:, :, 0, 0]
            encodings.append(encoded.squeeze(0).cpu())
    return torch.stack(encodings) if encodings else torch.empty((0, 512))

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>'],

```
from collections import defaultdict

def read_image_descriptions(filename):
    image_descriptions = defaultdict(list)

    with open(filename,'r') as in_file:
       for line in in_file:
            image = line[:line.index('#')]
            caption = line[line.index('#') + 3:]
            image_descriptions[image] += [['<START>'] + caption.split() + ['<EOS>']]

    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).

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

# ...existing code...
id_to_word = {}
word_to_id = {}

# Reserve 0, 1, 2 for special tokens
special_tokens = ["<PAD>", "<START>", "<EOS>"]
for idx, token in enumerate(special_tokens):
    id_to_word[idx] = token
    word_to_id[token] = idx

# Collect all tokens from training captions
vocab = set()
for image_id in train_list:
    for caption in descriptions[image_id]:
        for token in caption:
            if token not in special_tokens:
                vocab.add(token.lower())

# Sort for reproducibility
vocab = sorted(vocab)
for idx, token in enumerate(vocab, start=len(special_tokens)):
    id_to_word[idx] = token
    word_to_id[token] = idx
```

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

```
1163
```

```
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)

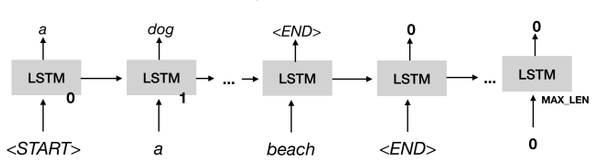
For now, we will just train a model for text generation without conditioning the generator on the image input.

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

↳ 9 cells hidden

## Training the Model

↳ 4 cells hidden

## 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.

↳ 6 cells hidden

# 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'))  # [n_images, 512]
        self.img_name_to_id = dict([(i, j) for (j, i) in enumerate(img_list)])
        self.data = []
        # For each image, add (img_name, caption) for all captions
        for img_name in img_list:
            for caption in descriptions[img_name]:
                self.data.append((img_name, caption))

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

    def __getitem__(self, k):
        img_name, caption = self.data[k]
        img_idx = self.img_name_to_id[img_name]
        img_encoding = self.img_data[img_idx]  # shape: [512]
```

```
        # Token to id conversion (same as before)
        def tok2id(token):
            if token in word_to_id:
                return word_to_id[token]
            elif token.lower() in word_to_id:
                return word_to_id[token.lower()]
            else:
                raise KeyError(token)
        input_ids = [tok2id(token) for token in caption[:-1]]
        output_ids = [tok2id(token) for token in caption[1:]]
        input_enc = input_ids + [0] * (MAX_LEN - len(input_ids))
        output_enc = output_ids + [0] * (MAX_LEN - len(output_ids))
        return img_encoding, torch.tensor(input_enc), torch.tensor(output_enc)
```

```
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__()
        self.embedding = nn.Embedding(vocab_size, 512)
        self.lstm = nn.LSTM(1024, 512, num_layers=1, bidirectional=False, batch_first=True)
        self.output = nn.Linear(512, vocab_size)

    def forward(self, img, input_seq):
        # img: (batch_size, 512)
        # input_seq: (batch_size, MAX_LEN)
        batch_size = input_seq.size(0)
        seq_len = input_seq.size(1)
        embedded = self.embedding(input_seq)  # (batch_size, MAX_LEN, 512)
        # Expand image encoding to (batch_size, MAX_LEN, 512)
        img_expanded = img.unsqueeze(1).expand(-1, seq_len, -1)
        # Concatenate along the last dimension
        lstm_input = torch.cat([embedded, img_expanded], dim=2)  # (batch_size, MAX_LEN, 1024)
        hidden = self.lstm(lstm_input)
        out = self.output(hidden[0])  # (batch_size, MAX_LEN, vocab_size)
        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, 7709])
```

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}")
```

```
train()
```

```
Current average loss: 8.939224243164062
Current average loss: 4.9177945462783965
Current average loss: 4.506590422113144
Current average loss: 4.278944828185528
Current average loss: 4.117303017666215
Current average loss: 3.9987654362372056
Current average loss: 3.896688973050744
Current average loss: 3.8186213823936126
Current average loss: 3.7584758998451755
Current average loss: 3.7018388074457844
Current average loss: 3.6521069529530528
Current average loss: 3.611836300234054
Current average loss: 3.5732902257666797
Current average loss: 3.53953224507595
Current average loss: 3.508025643146523
Current average loss: 3.477741012884568
Current average loss: 3.45168501968908
Current average loss: 3.428249423746359
Current average loss: 3.4047819187348582
Training loss epoch: 3.387144566090902
Average accuracy epoch: 0.38
```

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

```
import numpy as np

def greedy_decoder(img):
    # img: (512,) or (1, 512) torch tensor
    if img.dim() == 1:
        img = img.unsqueeze(0)
    input_ids = [word_to_id["<START>"]] + [word_to_id["<PAD>"]] * (MAX_LEN - 1)
    input_tensor = torch.tensor(input_ids).unsqueeze(0).to(DEVICE)  # shape: [1, MAX_LEN]
    result = ["<START>"]
    for t in range(1, MAX_LEN):
        with torch.no_grad():
            logits = model(img.to(DEVICE), input_tensor)
            probs = torch.softmax(logits[0, t-1], dim=0).cpu().numpy()
            next_token_id = np.random.choice(len(probs), p=probs)
        next_token = id_to_word[next_token_id]
        result.append(next_token)
        if next_token == "<EOS>":
            break
```

```
            input_tensor[0, t] = next_token_id  # update input for next step
        return result
```

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>', 'a', 'young', 'boy', 'with', 'her', 'nose', 'coloring', 'sweatshirt', 'in', 'a', 'chased', '<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.

```
import torch

def img_beam_decoder(n, img):
    # img: (512,) or (1, 512) torch tensor
    if img.dim() == 1:
        img = img.unsqueeze(0)
    # Each beam entry: (log_prob, [token_ids])
    beams = [(0.0, [word_to_id["<START>"]])]
    for t in range(1, MAX_LEN):
        candidates = []
        for log_prob, seq in beams:
            # Prepare input tensor for this sequence
            input_ids = seq + [word_to_id["<PAD>"]] * (MAX_LEN - len(seq))
            input_tensor = torch.tensor(input_ids).unsqueeze(0).to(DEVICE)
            with torch.no_grad():
                logits = model(img.to(DEVICE), input_tensor)
                probs = torch.softmax(logits[0, t-1], dim=0).cpu().numpy()
            # Get top n next tokens
            top_indices = probs.argsort()[-n:][::-1]
            for idx in top_indices:
                new_seq = seq + [idx]
                new_log_prob = log_prob + float(np.log(probs[idx] + 1e-12))  # add epsilon for safety
                candidates.append((new_log_prob, new_seq))
        # Prune to top n beams
        beams = sorted(candidates, key=lambda x: x[0], reverse=True)[:n]
    # Return the sequence (as tokens) with the highest log-probability
    best_seq = beams[0][1]
    # Convert to words, stop at <EOS> if present
    result = []
    for idx in best_seq:
```

```
            token = id_to_word[idx]
            result.append(token)
            if token == "<EOS>":
                break
        return result
```

**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 PIL import Image, ImageShow

dev_indices = [10, 50, 199]

for idx in dev_indices:
    img_name = dev_list[idx]
    raw_img = Image.open(os.path.join(IMG_PATH, img_name))
    preprocessed_img = preprocess(raw_img).to(DEVICE)
    encoded_img = img_encoder(preprocessed_img.unsqueeze(0)).reshape((512))

    print(f"Image: {img_name}")
    raw_img.show()  # Use Pillow's show method

    print("Greedy:", " ".join(greedy_decoder(encoded_img)))
    print("Beam search (n=3):", " ".join(img_beam_decoder(3, encoded_img)))
    print("Beam search (n=5):", " ".join(img_beam_decoder(5, encoded_img)))
    print("-" * 80)
```

```
Image: 2735792721_b8fe85e803.jpg
Greedy: <START> three two hikers running in a field balloon at some waves wading into the sandy . <EOS>
Beam search (n=3): <START> a black and black dog jumps over a red ball . <EOS>
Beam search (n=5): <START> a black and black dog jumps over a red ball . <EOS>
--------------------------------------------------------------------------------
Image: 3294952558_96bb8c8cf3.jpg
Greedy: <START> the horse is jumping off another dog in a brown area . <EOS>
Beam search (n=3): <START> a brown dog is running through a grassy field . <EOS>
Beam search (n=5): <START> two brown dogs play in the grass . <EOS>
--------------------------------------------------------------------------------
Image: 3730011701_5352e02286.jpg
Greedy: <START> a young boy holding his neck hands over the rock above a stick and playground . <EOS>
Beam search (n=3): <START> a little girl in a blue shirt and a blue shirt is standing in the water . <EOS>
Beam search (n=5): <START> a little girl in a blue shirt is standing in the water . <EOS>
--------------------------------------------------------------------------------
```

```
from IPython.display import Image as IPyImage, display

dev_indices = [10, 50, 199]

for idx in dev_indices:
    img_name = dev_list[idx]
    img_path = os.path.join(IMG_PATH, img_name)
    raw_img = PIL.Image.open(img_path)
    preprocessed_img = preprocess(raw_img).to(DEVICE)
    encoded_img = img_encoder(preprocessed_img.unsqueeze(0)).reshape((512))

    print(f"Image: {img_name}")
    display(IPyImage(filename=img_path))  # Show image inline in notebook

    print("Greedy:", " ".join(greedy_decoder(encoded_img)))
    print("Beam search (n=3):", " ".join(img_beam_decoder(3, encoded_img)))
    print("Beam search (n=5):", " ".join(img_beam_decoder(5, encoded_img)))
    print("-" * 80)
```

Image: 2735792721_b8fe85e803.jpg

Greedy: <START> two men tops dogs play in the sand . <EOS>
Beam search (n=3): <START> a black and black dog jumps over a red ball . <EOS>
Beam search (n=5): <START> a black and black dog jumps over a red ball . <EOS>
---
Image: 3294952558_96bb8c8cf3.jpg

Greedy: <START> a brown dog carries a yellow puppy playing in a running in the green grass . <EOS>
Beam search (n=3): <START> a brown dog is running through a grassy field . <EOS>
Beam search (n=5): <START> two brown dogs play in the grass . <EOS>
---
Image: 3730011701_5352e02286.jpg