lstm_flickr_caption_generator_torch

November 5, 2025

1 COMS W4705 - Homework 3

1.1 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.

1.1.1 Getting Started

There are a few required packages.

[107]:
```
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
```

[108]:
```
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
```

1.1.2 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

[3]:
```
#Download the data.
!wget https://storage.googleapis.com/4705_sp25_hw3/hw3data.zip
```

```
--2025-11-01 14:52:46--
https://storage.googleapis.com/4705_sp25_hw3/hw3data.zip
Resolving storage.googleapis.com (storage.googleapis.com)…
2607:f8b0:4006:807::201b, 2607:f8b0:4006:809::201b, 2607:f8b0:4006:820::201b,
…
Connecting to storage.googleapis.com
(storage.googleapis.com)|2607:f8b0:4006:807::201b|:443… connected.
HTTP request sent, awaiting response… 200 OK
Length: 1115435617 (1.0G) [application/zip]
Saving to: 'hw3data.zip'

hw3data.zip         100%[===================>]   1.04G  35.5MB/s    in 30s

2025-11-01 14:53:16 (35.4 MB/s) - 'hw3data.zip' saved [1115435617/1115435617]
```

[ ]:
```
#Then unzip the data
!unzip hw3data.zip
```

```
Archive:  hw3data.zip
replace hw3data/Flickr8k.token.txt? [y]es, [n]o, [A]ll, [N]one, [r]ename:
```

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.

[8]:
```
#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"
```

1.2 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.

[109]:
```
def load_image_list(filename):
    with open(filename,'r') as image_list_f:
        return [line.strip() for line in image_list_f]
```

[110]:
```
FLICKR_PATH="hw3data/"
```

[111]:
```
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

[112]:
```
len(train_list), len(dev_list), len(test_list)
```

[112]:
```
(6000, 1000, 1000)
```

Each entry is an image filename.

[113]:
```
dev_list[20]
```

[113]:
```
'3693961165_9d6c333d5b.jpg'
```

The images are located in a subdirectory.

[114]:
```
IMG_PATH = os.path.join(FLICKR_PATH, "Flickr8k_Dataset")
```

We can use PIL to open and display the image:

[115]:
```
image = PIL.Image.open(os.path.join(IMG_PATH, dev_list[20]))
image
```

[115]:

1.2.1 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_p

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

[16]:
```
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).

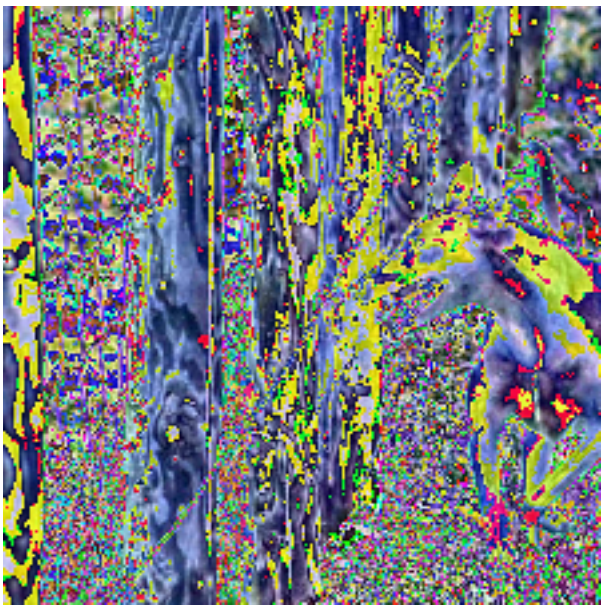
[17]:
```
processed_image = preprocess(image)
processed_image.shape
```

[17]:
```
torch.Size([3, 224, 224])
```

To the ResNet18 model, the images look like this:

[18]:
```
transforms.ToPILImage()(processed_image)
```

[18]:



1.2.2 Image Encoder

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

[19]:
```
img_encoder = torchvision.models.resnet18(weights=ResNet18_Weights.DEFAULT)
```

[20]:
```
img_encoder.eval()
```

[20]:
```
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)
      (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.

[64]:
```
lastremoved = list(img_encoder.children())[:-1]
img_encoder = torch.nn.Sequential(*lastremoved).to(DEVICE) # also send it to␣
 ↪GPU memory
img_encoder.eval()
```

[64]:
```
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)
      (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)
    )
  )
)
```

Let's try the encoder.

[65]:
```
def get_image(img_name):
    image = PIL.Image.open(os.path.join(IMG_PATH, img_name))
    return preprocess(image)
```

[66]:
```
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) becomes (1,3,224,224)
encoded.shape
```

[66]:
```
torch.Size([1, 512, 7, 7])
```

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:

[67]:
```
encoded = encoded[:,:,0,0] #this is our final image encoded
encoded.shape
```

[67]:
```
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.

[68]:
```
def encode_images(image_list):
    all_encodings = []
```

```
    for image_name in image_list:
        preprocessed_image = get_image(image_name)
        input_ready = preprocessed_image.unsqueeze(0).to(DEVICE)
        with torch.no_grad():
            encoded = img_encoder(input_ready)
            encoded = encoded[:, :, 0, 0]
        all_encodings.append(encoded.squeeze(0).cpu())
    return torch.stack(all_encodings)

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

[68]:
```
torch.Size([6000, 512])
```

We can now save this to disk:

[69]:
```
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.

1.3 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.

1.3.1 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>'],

[70]:
```
def read_image_descriptions(filename):
    image_descriptions = {}

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

            # Split image filename and caption text
```

```
            img_name, caption = line.split('\t', 1)
            img_name = img_name.split('#')[0]  # remove #0, #1, etc.

            # Tokenize and lowercase
            tokens = caption.lower().split()

            # Add <START> and <EOS> tokens
            tokens = [''] + tokens + ['']

            # Store in dictionary
            if img_name not in image_descriptions:
                image_descriptions[img_name] = []
            image_descriptions[img_name].append(tokens)


    return image_descriptions
```

[71]:
```
os.path.join(FLICKR_PATH, "Flickr8k.token.txt")
```

[71]:
```
'hw3data/Flickr8k.token.txt'
```

[72]:
```
descriptions = read_image_descriptions(os.path.join(FLICKR_PATH, "Flickr8k.
↪token.txt"))
```

[73]:
```
descriptions['1000268201_693b08cb0e.jpg']
```

[73]:
```
[['',
  'a',
  'child',
  'in',
  'a',
  'pink',
  'dress',
  'is',
  'climbing',
  'up',
  'a',
  'set',
  'of',
  'stairs',
  'in',
  'an',
  'entry',
  'way',
  '.',
  ''],
 ['', 'a', 'girl', 'going', 'into', 'a', 'wooden', 'building', '.', ''],
```

```
['',
 'a',
 'little',
 'girl',
 'climbing',
 'into',
 'a',
 'wooden',
 'playhouse',
 '.',
 ''],
['',
 'a',
 'little',
 'girl',
 'climbing',
 'the',
 'stairs',
 'to',
 'her',
 'playhouse',
 '.',
 ''],
['',
 'a',
 'little',
 'girl',
 'in',
 'a',
 'pink',
 'dress',
 'going',
 'into',
 'a',
 'wooden',
 'cabin',
 '.',
 '']]
```

The previous line shoudl return

1.3.2 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).

[74]:
```
id_to_word = {} #todo
id_to_word[0] = "<PAD>"
id_to_word[1] = "<START>"
id_to_word[2] = "<EOS>"
word_to_id = {} # todo
```

[75]:
```
def create_vocabulary(descriptions):
    all_tokens = set()
    for captions in descriptions.values():
        for caption in captions:
            all_tokens.update(caption)  # caption is a list of tokens (no␣
 ↪<START>, <EOS>)

    # Sort tokens so indices are consistent across runs
    all_tokens = sorted(all_tokens)

    # Initialize dictionaries with special tokens
    id_to_word = {
        0: "<PAD>",
        1: "<START>",

        2: "<EOS>"
    }

    word_to_id = {
        "<PAD>": 0,
        "<START>": 1,
        "<EOS>": 2
    }

    # Assign the rest of the words starting from index 3
    for idx, word in enumerate(all_tokens, start=3):
        id_to_word[idx] = word
        word_to_id[word] = idx

    return word_to_id, id_to_word
```

[76]:
```
word_to_id, id_to_word = create_vocabulary(descriptions)
```

[77]:
```
word_to_id['cat'] # should print an integer
```

[77]: 1350

[78]:
```
id_to_word[1350] # should print a token
```

[78]:
```
'cat'
```

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

1.4 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).

[79]:
```
MAX_LEN = max(len(description) for image_id in train_list for description in␣
 ↪descriptions[image_id])
MAX_LEN
```

[79]:
```
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.

[80]:
```
from torch import nn

vocab_size = len(word_to_id)
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)

1.4.1 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.

[81]:
```
import torch

PAD, START, EOS = "<PAD>", "<START>", "<EOS>"

def make_example(tokens, word_to_id, max_len):
    # full sequence with boundaries
    seq = [START] + tokens + [EOS]
    # cap to length max_len+1 so input/target can be max_len each
    seq = seq[:max_len + 1]

    inp  = seq[:-1]            # shifted right
    tgt  = seq[1:]             # next-token labels

    # pad to max_len
    pad_n = max_len - len(inp)
```

```
    inp_ids = [word_to_id[w] for w in inp] + [word_to_id[PAD]] * pad_n
    tgt_ids = [word_to_id[w] for w in tgt] + [word_to_id[PAD]] * pad_n

    return torch.tensor(inp_ids, dtype=torch.long), torch.tensor(tgt_ids,␣
 ↪dtype=torch.long)
```

[82]:
```
MAX_LEN = 40

class CaptionDataset(Dataset):

    def __init__(self, img_list):

        self.data = []
        for img in img_list:
            caps = descriptions.get(img, [])
            for tokens in caps:
                inp_ids, tgt_ids = make_example(tokens, word_to_id, MAX_LEN)
                self.data.append((inp_ids, tgt_ids))

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

    def __getitem__(self,k):
        input_enc, output_enc = self.data[k]
        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:

for the output:

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.

[83]:
```
data = CaptionDataset(train_list)
```

[84]:
```
i, o = data[0]
i
```

[84]:
```
tensor([   1,    3,   73,  804, 2311, 4014, 6487,  169,   73, 8685, 2311, 3921,
        7921, 7124,   18,    3,    0,    0,    0,    0,    0,    0,    0,    0,
           0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,
           0,    0,    0,    0])
```

[85]:
```
o
```

[85]:
```
tensor([   3,   73,  804, 2311, 4014, 6487,  169,   73, 8685, 2311, 3921, 7921,
        7124,   18,    3,    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])
```

Let's try the model:

[86]:
```
model = GeneratorModel().to(DEVICE)
```

[87]:
```
model(i.to(DEVICE)).shape   # should return a [40, vocab_size]  tensor.
```

[87]:
```
torch.Size([40, 8922])
```

1.4.2 Training the Model

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

[88]:
```
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.

[89]:
```
train()
```

```
Current average loss: 9.097975730895996
Current average loss: 4.33622131961407
Current average loss: 3.8792753278912597
Current average loss: 3.672295331954956
Current average loss: 3.532153718786644
Current average loss: 3.427486072757287
Current average loss: 3.354467778356619
```

```
Current average loss: 3.29535346228454
Current average loss: 3.2444443393140547
Current average loss: 3.203642227276581
Current average loss: 3.1676326414921903
Current average loss: 3.1367932556976954
Current average loss: 3.104756240542981
Current average loss: 3.0778445776016503
Current average loss: 3.0528300125032217
Current average loss: 3.0328982100337445
Current average loss: 3.013224076212085
Current average loss: 2.9956107766119473
Current average loss: 2.97847047717885
Training loss epoch: 2.9668203063964844
Average accuracy epoch: 0.46
```

[90]:
```
train()
```

```
Current average loss: 2.4822943210601807
Current average loss: 2.4493799705316524
Current average loss: 2.4311595556154773
Current average loss: 2.429334636146444
Current average loss: 2.437436081524799
Current average loss: 2.4385576098265047
Current average loss: 2.435493794534845
Current average loss: 2.4326553610014
Current average loss: 2.433630380291171
Current average loss: 2.4313297550897883
Current average loss: 2.43083993955092
Current average loss: 2.434513266902962
Current average loss: 2.432672770096797
Current average loss: 2.4283033900770383
Current average loss: 2.4266850986623663
Current average loss: 2.4277185495498577
Current average loss: 2.4276583261001417
Current average loss: 2.4277474816443987
Current average loss: 2.4251114783321466
Training loss epoch: 2.42389175084432
Average accuracy epoch: 0.50
```

1.4.3 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.

[91]:
```
def decoder():
    START = word_to_id["<START>"]
```

```
    PAD   = word_to_id["<PAD>"]
    EOS   = word_to_id["<EOS>"]

    seq = torch.full((1, MAX_LEN), PAD, dtype=torch.long, device=DEVICE)
    seq[0, 0] = START
    cur_len = 1

    with torch.no_grad():
        while cur_len < MAX_LEN:
            logits = model(seq)                # (1, max_len, vocab)
            step_logits = logits[0, cur_len-1] # predict next token at this step
            next_id = int(step_logits.argmax(dim=-1))

            if next_id == EOS:
                break

            seq[0, cur_len] = next_id
            cur_len += 1

    out_ids = seq[0, 1:cur_len].tolist()
    return [id_to_word[i] for i in out_ids]
```

[92]:
```
decoder()
```

[92]:
```
['',
 'a',
 'man',
 'in',
 'a',
 'red',
 'shirt',
 'is',
 'standing',
 'in',
 'front',
 'of',
 'a',
 'crowd',
 '.',
 '']
```

[93]:
```
import numpy as np
import torch.nn.functional as F


def sample_decoder():
    START = word_to_id["<START>"]
```

```
    PAD   = word_to_id["<PAD>"]
    EOS   = word_to_id["<EOS>"]

    seq = torch.full((1, MAX_LEN), PAD, dtype=torch.long, device=DEVICE)
    seq[0, 0] = START
    cur_len = 1

    with torch.no_grad():
        while cur_len < MAX_LEN:
            logits = model(seq)
            step_logits = logits[0, cur_len - 1]

            # Convert logits → probabilities
            probs = F.softmax(step_logits, dim=-1).cpu().numpy()

            # Randomly sample next word ID
            next_id = np.random.choice(len(probs), p=probs)

            # Stop if EOS is generated
            if next_id == EOS:
                break

            seq[0, cur_len] = next_id
            cur_len += 1

    out_ids = seq[0, 1:cur_len].tolist()
    return [id_to_word[i] for i in out_ids]

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

```
['', 'a', 'man', 'in', 'white', 'snoopy', 'along', 'a', 'river', 'in', 'front',
'of', 'a', 'fountain', '.', '']
['', 'instruments', 'four', 'guys', 'under', 'an', 'umbrella', '.', '']
['', 'a', 'golden', 'retriever', 'with', 'brown', 'hair', 'is', 'running', 'in',
'a', 'colorful', 'yellow', 'rails', '.', '']
['', 'two', 'dogs', 'are', 'playing', 'with', 'tall', 'toys', '.', '']
['', 'the', 'dog', 'is', 'running', 'along', 'the', 'snow', 'mound', '.', '']
```

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

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.

1.5 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 em-

bedding. 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.

[94]:
```
MAX_LEN = 40

class CaptionAndImage(Dataset):

    def __init__(self, img_list):

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

        self.data = []
        for name in img_list:
            for tokens in descriptions.get(name, []):
                inp_ids, tgt_ids = make_example(tokens, word_to_id, MAX_LEN)
                img_idx = self.img_name_to_id[name]
                self.data.append((img_idx, inp_ids, tgt_ids))

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

    def __getitem__(self,k):
        img_idx, inp_ids, tgt_ids = self.data[k]
        img_vec = self.img_data[img_idx]
        return img_vec, inp_ids, tgt_ids
```

[95]:
```
joint_data = CaptionAndImage(train_list)
img, i, o = joint_data[0]
img.shape  # expect torch.Size([512])
loader = DataLoader(joint_data, batch_size = 16, shuffle = True)
```

```
/var/folders/k5/4bbvpgbd7rb8t2gjc26njsn00000gn/T/ipykernel_5393/1591797335.py:7:
FutureWarning: You are using `torch.load` with `weights_only=False` (the current
default value), which uses the default pickle module implicitly. It is possible
to construct malicious pickle data which will execute arbitrary code during
unpickling (See
https://github.com/pytorch/pytorch/blob/main/SECURITY.md#untrusted-models for
more details). In a future release, the default value for `weights_only` will be
flipped to `True`. This limits the functions that could be executed during
```

```
unpickling. Arbitrary objects will no longer be allowed to be loaded via this
mode unless they are explicitly allowlisted by the user via
`torch.serialization.add_safe_globals`. We recommend you start setting
`weights_only=True` for any use case where you don't have full control of the
loaded file. Please open an issue on GitHub for any issues related to this
experimental feature.
  self.img_data = torch.load(open("encoded_images_train.pt",'rb'))
```

[96]:
```
i.shape # should return torch.Size([40])
```

[96]:
```
torch.Size([40])
```

[97]:
```
o.shape # should return torch.Size([40])
```

[97]:
```
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.

[98]:
```
vocab_size = len(word_to_id)

class CaptionGeneratorModel(nn.Module):

    def __init__(self):
        super(CaptionGeneratorModel, self).__init__()
        self.embedding = nn.Embedding(vocab_size, 512)
        self.lstm = nn.LSTM(
            input_size=1024,
            hidden_size=512,
            num_layers=1,
            batch_first=True
        )
        self.output = nn.Linear(512, vocab_size)

    def forward(self, img, input_seq):
        emb = self.embedding(input_seq)
        img = img.unsqueeze(1).repeat(1, emb.size(1), 1)
        x = torch.cat((img, emb), dim=-1)
```

```
        lstm_out, _ = self.lstm(x)
        out = self.output(lstm_out)
        return out
```

Let's try this new model on one item:

[99]:
```
model = CaptionGeneratorModel().to(DEVICE)
```

[100]:
```
item = joint_data[0]
img, input_seq, output_seq = item
```

[101]:
```
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)
```

[101]:
```
torch.Size([1, 40, 8922])
```

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

[102]:
```
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)

def train1():
    """
    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}")
```

[103]:
```
train1()
```

```
Current average loss: 9.101882934570312
Current average loss: 4.369837890757193
Current average loss: 3.9367501047713245
Current average loss: 3.716358721058234
Current average loss: 3.5691699553606218
Current average loss: 3.471749705468823
Current average loss: 3.4005243564009073
Current average loss: 3.335627857866709
Current average loss: 3.2836950125914535
Current average loss: 3.2378404696164993
Current average loss: 3.2054930014329237
Current average loss: 3.1680207495035417
Current average loss: 3.1380224287460288
Current average loss: 3.111188436635726
Current average loss: 3.082816266060556
Current average loss: 3.0567397304092703
```

```
Current average loss: 3.0348481117822765
Current average loss: 3.0139787949792503
Current average loss: 2.995617513190634
Training loss epoch: 2.9843530604044597
Average accuracy epoch: 0.46
```

[104]:
```
train1()
```

```
Current average loss: 2.416506767272949
Current average loss: 2.4134783414330814
Current average loss: 2.408363963241008
Current average loss: 2.4106322641784566
Current average loss: 2.419328722275998
Current average loss: 2.4275489448311323
Current average loss: 2.4276998241411865
Current average loss: 2.4230203567319863
Current average loss: 2.4175461701834844
Current average loss: 2.416086265170217
Current average loss: 2.4111546748406165
Current average loss: 2.4080397794725243
Current average loss: 2.405065215100456
Current average loss: 2.400051030079096
Current average loss: 2.3974128428056867
Current average loss: 2.3956121473134475
Current average loss: 2.3917904871839943
Current average loss: 2.388406707636403
Current average loss: 2.387893607787196
Training loss epoch: 2.386043938446045
Average accuracy epoch: 0.50
```

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

[117]:
```
def greedy_decoder(img):
    model.eval()
    START = word_to_id["<START>"]
    PAD = word_to_id["<PAD>"]
    EOS = word_to_id["<EOS>"]

    seq = torch.full((1, MAX_LEN), PAD, dtype=torch.long, device=DEVICE)
    seq[0, 0] = START
    cur_len = 1

    with torch.no_grad():
        while cur_len < MAX_LEN:
            # forward pass through the model
            logits = model(img.unsqueeze(0).to(DEVICE), seq[:, :cur_len])
```

```
            step_logits = logits[0, cur_len - 1]  # last predicted token's
↪logits

            # pick the most likely token
            next_id = int(step_logits.argmax(dim=-1))

            if next_id == EOS:
                break

            seq[0, cur_len] = next_id
            cur_len += 1

    out_ids = seq[0, 1:cur_len].tolist()
    return [id_to_word[i] for i in out_ids]
```

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

[119]:
```
raw_img = PIL.Image.open(os.path.join(IMG_PATH, dev_list[199]))
preprocessed_img = preprocess(raw_img).to(DEVICE)
with torch.no_grad():
    fmap = img_encoder(preprocessed_img.unsqueeze(0))
    encoded_img = fmap.mean(dim=(2, 3)).squeeze(0)
caption = greedy_decoder(encoded_img)
print(caption)
raw_img
```

```
['', 'a', 'young', 'boy', 'in', 'a', 'blue', 'shirt', 'is', 'jumping', 'into',
'a', 'pool', '.', '']
```

[119]:

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

1.6 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.

[120]:
```
import torch
import math

def img_beam_decoder(n, img, length_penalty=0.0):
    model.eval()
    START = word_to_id["<START>"]
    PAD   = word_to_id["<PAD>"]
    EOS   = word_to_id["<EOS>"]

    img = img.to(DEVICE)

    # Each beam item is (logprob, ids_tensor) where ids is 1D Long tensor␣
↪(current length)
    beams = [(0.0, torch.tensor([START], dtype=torch.long, device=DEVICE))]

    with torch.no_grad():
        for t in range(1, MAX_LEN):  # we already placed <START>
            candidates = []

            for logp, seq_ids in beams:
                # Run the model on the *current* prefix
                logits = model(img.unsqueeze(0), seq_ids.unsqueeze(0))      #␣
↪(1, t, V)
                step_logits = logits[0, -1]                                 #␣
↪(V,)
                step_logprobs = torch.log_softmax(step_logits, dim=-1)      #␣
↪log P(next|prefix,img)

                # Take top-k next words for this prefix
                topk_logp, topk_idx = torch.topk(step_logprobs, k=n)

                for add_logp, wid in zip(topk_logp.tolist(), topk_idx.tolist()):
                    new_seq = torch.cat([seq_ids, torch.tensor([wid],␣
↪device=DEVICE)])
                    new_logp = logp + add_logp
                    # Optional length normalization for ranking/pruning
```

```
                    norm = (len(new_seq)) ** length_penalty if length_penalty >␣
↪0 else 1.0
                    candidates.append((new_logp / norm, new_seq))

            # Keep the best n candidates
            candidates.sort(key=lambda x: x[0], reverse=True)
            beams = candidates[:n]

    # Choose the best finished sequence after MAX_LEN steps
    best_logp, best_ids = max(beams, key=lambda x: x[0])

    # Convert ids -> tokens; drop <START>, cut at first <EOS>, strip trailing␣
↪<PAD>
    out = []
    for w in best_ids.tolist()[1:]:
        if w == EOS:
            break
        if w == PAD:
            continue
        out.append(id_to_word[w])
    return out
```

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.

[121]:
```
import os
import random
from PIL import Image
import matplotlib.pyplot as plt

# pick three distinct example indices (replace with your chosen dev indices)
example_indices = [199, 200, 201]

def show_example(idx):
    # 1) fetch image name, raw image (for display), and its encoded vector for␣
↪decoding
    img_name = dev_list[idx]
    pil_img  = Image.open(os.path.join(IMG_PATH, img_name)).convert("RGB")
    img_vec, _, _ = joint_data[idx]

    # 2) run decoders
    greedy_cap = greedy_decoder(img_vec)
    beam3_cap  = img_beam_decoder(3, img_vec)
    beam5_cap  = img_beam_decoder(5, img_vec)

    # 3) display
    plt.figure(figsize=(7,5))
```

```
    plt.imshow(pil_img)
    plt.axis("off")
    plt.title(img_name)
    plt.show()

    print("Greedy: ", " ".join(greedy_cap))
    print("Beam-3:", " ".join(beam3_cap))
    print("Beam-5:", " ".join(beam5_cap))
    print("-"*80)

for idx in example_indices:
    show_example(idx)
```


```
Greedy:   a man in a black shirt is standing in front of a building with a large
crowd .
Beam-3:  a group of people are sitting on the sidewalk .
Beam-5:  a group of people are sitting on a bench in front of a building .
--------------------------------------------------------------------------------
```

3341961913_9a9b362f15.jpg


```
Greedy:   a boy in a red shirt is jumping on a swing .
Beam-3:  a little girl in a pink shirt is running through a field of flowers .
Beam-5:  a little girl in a pink dress is running through a field of flowers .
--------------------------------------------------------------------------------
```

2198511848_311d8a8c2f.jpg

```
Greedy:   a boy in a red shirt is jumping on a swing .
Beam-3:  a little girl in a pink shirt is running through a field of flowers .
Beam-5:  a little girl in a pink dress is running through a field of flowers .
--------------------------------------------------------------------------------
```

[ ]: