2_Training

May 2, 2020

1 Computer Vision Nanodegree

1.1 Project: Image Captioning

In this notebook, you will train your CNN-RNN model.

You are welcome and encouraged to try out many different architectures and hyperparameters when searching for a good model.

This does have the potential to make the project quite messy! Before submitting your project, make sure that you clean up: - the code you write in this notebook. The notebook should describe how to train a single CNN-RNN architecture, corresponding to your final choice of hyperparameters. You should structure the notebook so that the reviewer can replicate your results by running the code in this notebook.
- the output of the code cell in **Step 2**. The output should show the output obtained when training the model from scratch.

This notebook **will be graded**.

Feel free to use the links below to navigate the notebook: - Section **??**: Training Setup - Section **??**: Train your Model - Section **??**: (Optional) Validate your Model

Step 1: Training Setup

In this step of the notebook, you will customize the training of your CNN-RNN model by specifying hyperparameters and setting other options that are important to the training procedure. The values you set now will be used when training your model in **Step 2** below.

You should only amend blocks of code that are preceded by a `TODO` statement. **Any code blocks that are not preceded by a `TODO` statement should not be modified**.

1.1.1 Task #1

Begin by setting the following variables: - `batch_size` - the batch size of each training batch. It is the number of image-caption pairs used to amend the model weights in each training step. - `vocab_threshold` - the minimum word count threshold. Note that a larger threshold will result in a smaller vocabulary, whereas a smaller threshold will include rarer words and result in a larger vocabulary.
- `vocab_from_file` - a Boolean that decides whether to load the vocabulary from file. - `embed_size` - the dimensionality of the image and word embeddings.
- `hidden_size` - the number of features in the hidden state of the RNN decoder.
- `num_epochs` - the number of epochs to train the model. We recommend that you set

`num_epochs=3`, but feel free to increase or decrease this number as you wish. This paper trained a captioning model on a single state-of-the-art GPU for 3 days, but you'll soon see that you can get reasonable results in a matter of a few hours! (*But of course, if you want your model to compete with current research, you will have to train for much longer.*) - `save_every` - determines how often to save the model weights. We recommend that you set `save_every=1`, to save the model weights after each epoch. This way, after the `i`th epoch, the encoder and decoder weights will be saved in the `models/` folder as `encoder-i.pkl` and `decoder-i.pkl`, respectively. - `print_every` - determines how often to print the batch loss to the Jupyter notebook while training. Note that you **will not** observe a monotonic decrease in the loss function while training - this is perfectly fine and completely expected! You are encouraged to keep this at its default value of `100` to avoid clogging the notebook, but feel free to change it. - `log_file` - the name of the text file containing - for every step - how the loss and perplexity evolved during training.

If you're not sure where to begin to set some of the values above, you can peruse this paper and this paper for useful guidance! **To avoid spending too long on this notebook**, you are encouraged to consult these suggested research papers to obtain a strong initial guess for which hyperparameters are likely to work best. Then, train a single model, and proceed to the next notebook (**3_Inference.ipynb**). If you are unhappy with your performance, you can return to this notebook to tweak the hyperparameters (and/or the architecture in **model.py**) and re-train your model.

1.1.2 Question 1

Question: Describe your CNN-RNN architecture in detail. With this architecture in mind, how did you select the values of the variables in Task 1? If you consulted a research paper detailing a successful implementation of an image captioning model, please provide the reference.

Answer: For CNN-RNN model i have used Encoder-Decoder architecture which is shown below:- For Encoder:- First i have used pre-trained ResNet50 model then i have remove the last fully-connected layer of ResNet architecture and transformed it in our own custom linear layer which will goes as an input to the Decoder.

For Decoder:- It has 1 embedding layer:- This transform our input image feature vectors and captions into word embedding. It has 1 LSTM RNN layer:- Word embedding then goes to this layer as an input It has 1 fully connected linear layer:- This transforms our outputs from lstm-rnn to vocabulary keys

The variables which i have choose is are as follows:-

batch_size = I have used 128 batch_size. Because our dataset is very big and by keeping our batch_size i don't want to trained my model even longer. So by choosing batch_size=128 i am able to trained our model with such big dataset faster.

vocab_threshold = I choose 4. Because if i choose bigger threshold then the vocabulary words will be less and so my model can't be able to give captions of different images properly. And with bigger vocab_threshold, the count of unknown words were also high so i decided to set my vocab_threshold to small.

embed_size = I have choose 256 because in hyperparameters lesson i learned that by for large dataset the recommended embed_size is from 128 to 640.

hidden_size = 512 looks good fit for such big dataset.

num_epochs = I used 3 epochs. For this size of dataset and by running my model for more than 10 hours i came to conclusion that this number of epochs is sufficent.

1.1.3 (Optional) Task #2

Note that we have provided a recommended image transform `transform_train` for pre-processing the training images, but you are welcome (and encouraged!) to modify it as you wish. When modifying this transform, keep in mind that: - the images in the dataset have varying heights and widths, and - if using a pre-trained model, you must perform the corresponding appropriate normalization.

1.1.4 Question 2

Question: How did you select the transform in `transform_train`? If you left the transform at its provided value, why do you think that it is a good choice for your CNN architecture?

Answer: I have used the default values in transform function. I am doing resize,crop,horizontal flip,converting in tensor and normalization of our images. Here i am not data agumentation. Doing data augmentations will increase our model accuracy but here in this model key is not the only requirement. And by doing data augmentations the training process will take long time. I have notice that even without doing data augmentations the training time was about 10 hours and so in decided that i will not data augmentations

1.1.5 Task #3

Next, you will specify a Python list containing the learnable parameters of the model. For instance, if you decide to make all weights in the decoder trainable, but only want to train the weights in the embedding layer of the encoder, then you should set `params` to something like:

```
params = list(decoder.parameters()) + list(encoder.embed.parameters())
```

1.1.6 Question 3

Question: How did you select the trainable parameters of your architecture? Why do you think this is a good choice?

Answer: Here i used all the parameters of decoder because in our decoder there is less number of parameters and for encoder layer i have used only the embedding layer because my encoder is already train on ResNet50 and if i used all the parameters of encoder then the training time will take even longer.

1.1.7 Task #4

Finally, you will select an optimizer.

1.1.8 Question 4

Question: How did you select the optimizer used to train your model?

Answer: I have used Adam optimizer because it the one of the fastest optimizer. For such a big dataset it is recommended that to use Adam because it will converge faster as compared to other optimizer.

```
In [2]: import torch
        import torch.nn as nn
```

```python
from torchvision import transforms
import sys
sys.path.append('/opt/cocoapi/PythonAPI')
from pycocotools.coco import COCO
from data_loader import get_loader
from model import EncoderCNN, DecoderRNN
import math
from torch.optim import Adam


## TODO #1: Select appropriate values for the Python variables below.
batch_size = 64          # batch size
vocab_threshold = 4        # minimum word count threshold
vocab_from_file = True    # if True, load existing vocab file
embed_size = 256           # dimensionality of image and word embeddings
hidden_size = 512          # number of features in hidden state of the RNN decoder
num_epochs = 3             # number of training epochs
save_every = 1             # determines frequency of saving model weights
print_every = 100          # determines window for printing average loss
log_file = 'training_log.txt'       # name of file with saved training loss and perplexi

# (Optional) TODO #2: Amend the image transform below.
transform_train = transforms.Compose([
    transforms.Resize(256),                          # smaller edge of image resized to
    transforms.RandomCrop(224),                      # get 224x224 crop from random loca
    transforms.RandomHorizontalFlip(),               # horizontally flip image with prob
    transforms.ToTensor(),                           # convert the PIL Image to a tensor
    transforms.Normalize((0.485, 0.456, 0.406),      # normalize image for pre-trained m
                         (0.229, 0.224, 0.225))])

# Build data loader.
data_loader = get_loader(transform=transform_train,
                         mode='train',
                         batch_size=batch_size,
                         vocab_threshold=vocab_threshold,
                         vocab_from_file=vocab_from_file)

# The size of the vocabulary.
vocab_size = len(data_loader.dataset.vocab)

# Initialize the encoder and decoder.
encoder = EncoderCNN(embed_size)
decoder = DecoderRNN(embed_size, hidden_size, vocab_size)

# Move models to GPU if CUDA is available.
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
encoder.to(device)
decoder.to(device)
```

```
# Define the loss function.
criterion = nn.CrossEntropyLoss().cuda() if torch.cuda.is_available() else nn.CrossEntro

# TODO #3: Specify the learnable parameters of the model.
params = list(decoder.parameters()) + list(encoder.embed.parameters())

# TODO #4: Define the optimizer.
optimizer = Adam(params, lr = 0.001)

# Set the total number of training steps per epoch.
total_step = math.ceil(len(data_loader.dataset.caption_lengths) / data_loader.batch_samp
```

```
Vocabulary successfully loaded from vocab.pkl file!
loading annotations into memory...


100%|| 414113/414113 [01:34<00:00, 4367.45it/s]


Done (t=0.90s)
creating index...
index created!
Obtaining caption lengths...
```

Step 2: Train your Model

Once you have executed the code cell in **Step 1**, the training procedure below should run without issue.

It is completely fine to leave the code cell below as-is without modifications to train your model. However, if you would like to modify the code used to train the model below, you must ensure that your changes are easily parsed by your reviewer. In other words, make sure to provide appropriate comments to describe how your code works!

You may find it useful to load saved weights to resume training. In that case, note the names of the files containing the encoder and decoder weights that you'd like to load (`encoder_file` and `decoder_file`). Then you can load the weights by using the lines below:

```
# Load pre-trained weights before resuming training.
encoder.load_state_dict(torch.load(os.path.join('./models', encoder_file)))
decoder.load_state_dict(torch.load(os.path.join('./models', decoder_file)))
```

While trying out parameters, make sure to take extensive notes and record the settings that you used in your various training runs. In particular, you don't want to encounter a situation where you've trained a model for several hours but can't remember what settings you used :).

1.1.9 A Note on Tuning Hyperparameters

To figure out how well your model is doing, you can look at how the training loss and perplexity evolve during training - and for the purposes of this project, you are encouraged to amend the hyperparameters based on this information.

However, this will not tell you if your model is overfitting to the training data, and, unfortunately, overfitting is a problem that is commonly encountered when training image captioning models.

For this project, you need not worry about overfitting. **This project does not have strict requirements regarding the performance of your model**, and you just need to demonstrate that your model has learned ***something*** when you generate captions on the test data. For now, we strongly encourage you to train your model for the suggested 3 epochs without worrying about performance; then, you should immediately transition to the next notebook in the sequence (**3_Inference.ipynb**) to see how your model performs on the test data. If your model needs to be changed, you can come back to this notebook, amend hyperparameters (if necessary), and re-train the model.

That said, if you would like to go above and beyond in this project, you can read about some approaches to minimizing overfitting in section 4.3.1 of this paper. In the next (optional) step of this notebook, we provide some guidance for assessing the performance on the validation dataset.

In [3]:
```
import torch.utils.data as data
import numpy as np
import os
import requests
import time

# Open the training log file.
f = open(log_file, 'w')

old_time = time.time()
response = requests.request("GET",
                            "http://metadata.google.internal/computeMetadata/v1/instance
                            headers={"Metadata-Flavor":"Google"})

for epoch in range(1, num_epochs+1):

    for i_step in range(1, total_step+1):

        if time.time() - old_time > 60:
            old_time = time.time()
            requests.request("POST",
                             "https://nebula.udacity.com/api/v1/remote/keep-alive",
                             headers={'Authorization': "STAR " + response.text})

        # Randomly sample a caption length, and sample indices with that length.
        indices = data_loader.dataset.get_train_indices()
        # Create and assign a batch sampler to retrieve a batch with the sampled indices
        new_sampler = data.sampler.SubsetRandomSampler(indices=indices)
        data_loader.batch_sampler.sampler = new_sampler

        # Obtain the batch.
        images, captions = next(iter(data_loader))
```

```
            # Move batch of images and captions to GPU if CUDA is available.
            images = images.to(device)
            captions = captions.to(device)

            # Zero the gradients.
            decoder.zero_grad()
            encoder.zero_grad()

            # Pass the inputs through the CNN-RNN model.
            features = encoder(images)
            outputs = decoder(features, captions)

            # Calculate the batch loss.
            loss = criterion(outputs.view(-1, vocab_size), captions.view(-1))

            # Backward pass.
            loss.backward()

            # Update the parameters in the optimizer.
            optimizer.step()

            # Get training statistics.
            stats = 'Epoch [%d/%d], Step [%d/%d], Loss: %.4f, Perplexity: %5.4f' % (epoch, n

            # Print training statistics (on same line).
            print('\r' + stats, end="")
            sys.stdout.flush()

            # Print training statistics to file.
            f.write(stats + '\n')
            f.flush()

            # Print training statistics (on different line).
            if i_step % print_every == 0:
                print('\r' + stats)

        # Save the weights.
        if epoch % save_every == 0:
            torch.save(decoder.state_dict(), os.path.join('./models', 'decoder-%d.pkl' % epo
            torch.save(encoder.state_dict(), os.path.join('./models', 'encoder-%d.pkl' % epo

    # Close the training log file.
    f.close()
```

```
Epoch [1/3], Step [100/6471], Loss: 4.2886, Perplexity: 72.8622
Epoch [1/3], Step [200/6471], Loss: 3.4867, Perplexity: 32.6794
Epoch [1/3], Step [300/6471], Loss: 3.2075, Perplexity: 24.71721
Epoch [1/3], Step [400/6471], Loss: 3.1726, Perplexity: 23.86953
```

```
Epoch [1/3], Step [500/6471], Loss: 3.7156, Perplexity: 41.0842
Epoch [1/3], Step [600/6471], Loss: 2.7441, Perplexity: 15.5504
Epoch [1/3], Step [700/6471], Loss: 2.9943, Perplexity: 19.9707
Epoch [1/3], Step [800/6471], Loss: 2.7940, Perplexity: 16.3456
Epoch [1/3], Step [900/6471], Loss: 2.7401, Perplexity: 15.4887
Epoch [1/3], Step [1000/6471], Loss: 2.7781, Perplexity: 16.0887
Epoch [1/3], Step [1100/6471], Loss: 2.5946, Perplexity: 13.3906
Epoch [1/3], Step [1200/6471], Loss: 2.6392, Perplexity: 14.0019
Epoch [1/3], Step [1300/6471], Loss: 2.7423, Perplexity: 15.5233
Epoch [1/3], Step [1400/6471], Loss: 3.1606, Perplexity: 23.5843
Epoch [1/3], Step [1500/6471], Loss: 2.5362, Perplexity: 12.6312
Epoch [1/3], Step [1600/6471], Loss: 2.8463, Perplexity: 17.2247
Epoch [1/3], Step [1700/6471], Loss: 2.6214, Perplexity: 13.7556
Epoch [1/3], Step [1800/6471], Loss: 2.4355, Perplexity: 11.4213
Epoch [1/3], Step [1900/6471], Loss: 2.3856, Perplexity: 10.8661
Epoch [1/3], Step [2000/6471], Loss: 2.4358, Perplexity: 11.4253
Epoch [1/3], Step [2100/6471], Loss: 2.3739, Perplexity: 10.7388
Epoch [1/3], Step [2200/6471], Loss: 2.4157, Perplexity: 11.1974
Epoch [1/3], Step [2300/6471], Loss: 2.5042, Perplexity: 12.2332
Epoch [1/3], Step [2400/6471], Loss: 2.8316, Perplexity: 16.9728
Epoch [1/3], Step [2500/6471], Loss: 2.5931, Perplexity: 13.3718
Epoch [1/3], Step [2600/6471], Loss: 2.5389, Perplexity: 12.6659
Epoch [1/3], Step [2700/6471], Loss: 2.4288, Perplexity: 11.3454
Epoch [1/3], Step [2800/6471], Loss: 2.5064, Perplexity: 12.2613
Epoch [1/3], Step [2900/6471], Loss: 2.5895, Perplexity: 13.3228
Epoch [1/3], Step [3000/6471], Loss: 2.6047, Perplexity: 13.5266
Epoch [1/3], Step [3100/6471], Loss: 2.3116, Perplexity: 10.0902
Epoch [1/3], Step [3200/6471], Loss: 2.2621, Perplexity: 9.60328
Epoch [1/3], Step [3300/6471], Loss: 2.6697, Perplexity: 14.4359
Epoch [1/3], Step [3400/6471], Loss: 2.6615, Perplexity: 14.3174
Epoch [1/3], Step [3500/6471], Loss: 2.2428, Perplexity: 9.42011
Epoch [1/3], Step [3600/6471], Loss: 2.3162, Perplexity: 10.1373
Epoch [1/3], Step [3700/6471], Loss: 2.8349, Perplexity: 17.0280
Epoch [1/3], Step [3800/6471], Loss: 2.3781, Perplexity: 10.7842
Epoch [1/3], Step [3900/6471], Loss: 2.7532, Perplexity: 15.6929
Epoch [1/3], Step [4000/6471], Loss: 2.4169, Perplexity: 11.2108
Epoch [1/3], Step [4100/6471], Loss: 2.2699, Perplexity: 9.67815
Epoch [1/3], Step [4200/6471], Loss: 2.3660, Perplexity: 10.6547
Epoch [1/3], Step [4300/6471], Loss: 2.2192, Perplexity: 9.19988
Epoch [1/3], Step [4400/6471], Loss: 2.5564, Perplexity: 12.8897
Epoch [1/3], Step [4500/6471], Loss: 2.2419, Perplexity: 9.41130
Epoch [1/3], Step [4600/6471], Loss: 2.2543, Perplexity: 9.52901
Epoch [1/3], Step [4700/6471], Loss: 2.2395, Perplexity: 9.38883
Epoch [1/3], Step [4800/6471], Loss: 2.2071, Perplexity: 9.08912
Epoch [1/3], Step [4900/6471], Loss: 2.6513, Perplexity: 14.1723
Epoch [1/3], Step [5000/6471], Loss: 1.9954, Perplexity: 7.35529
Epoch [1/3], Step [5100/6471], Loss: 2.2501, Perplexity: 9.48869
Epoch [1/3], Step [5200/6471], Loss: 2.2545, Perplexity: 9.53044
```

```
Epoch [1/3], Step [5300/6471], Loss: 2.2558, Perplexity: 9.54293
Epoch [1/3], Step [5400/6471], Loss: 2.0788, Perplexity: 7.99480
Epoch [1/3], Step [5500/6471], Loss: 2.3058, Perplexity: 10.0317
Epoch [1/3], Step [5600/6471], Loss: 2.4937, Perplexity: 12.1054
Epoch [1/3], Step [5700/6471], Loss: 2.4445, Perplexity: 11.5249
Epoch [1/3], Step [5800/6471], Loss: 2.1044, Perplexity: 8.20226
Epoch [1/3], Step [5900/6471], Loss: 2.2239, Perplexity: 9.24349
Epoch [1/3], Step [6000/6471], Loss: 2.2264, Perplexity: 9.26650
Epoch [1/3], Step [6100/6471], Loss: 2.0506, Perplexity: 7.772621
Epoch [1/3], Step [6200/6471], Loss: 2.4309, Perplexity: 11.3694
Epoch [1/3], Step [6300/6471], Loss: 2.1967, Perplexity: 8.99532
Epoch [1/3], Step [6400/6471], Loss: 2.0490, Perplexity: 7.75991
Epoch [2/3], Step [100/6471], Loss: 2.1254, Perplexity: 8.376294
Epoch [2/3], Step [200/6471], Loss: 2.1364, Perplexity: 8.46886
Epoch [2/3], Step [300/6471], Loss: 2.1227, Perplexity: 8.35358
Epoch [2/3], Step [400/6471], Loss: 2.0573, Perplexity: 7.82467
Epoch [2/3], Step [500/6471], Loss: 2.0507, Perplexity: 7.77343
Epoch [2/3], Step [600/6471], Loss: 2.1826, Perplexity: 8.86927
Epoch [2/3], Step [700/6471], Loss: 2.1205, Perplexity: 8.33531
Epoch [2/3], Step [800/6471], Loss: 2.0469, Perplexity: 7.74354
Epoch [2/3], Step [900/6471], Loss: 2.1220, Perplexity: 8.34808
Epoch [2/3], Step [1000/6471], Loss: 2.0015, Perplexity: 7.4003
Epoch [2/3], Step [1100/6471], Loss: 1.9195, Perplexity: 6.81771
Epoch [2/3], Step [1200/6471], Loss: 1.9206, Perplexity: 6.82518
Epoch [2/3], Step [1300/6471], Loss: 2.0886, Perplexity: 8.07381
Epoch [2/3], Step [1400/6471], Loss: 2.0472, Perplexity: 7.74627
Epoch [2/3], Step [1500/6471], Loss: 2.1289, Perplexity: 8.40597
Epoch [2/3], Step [1600/6471], Loss: 2.1977, Perplexity: 9.00400
Epoch [2/3], Step [1700/6471], Loss: 1.9950, Perplexity: 7.35235
Epoch [2/3], Step [1800/6471], Loss: 2.0694, Perplexity: 7.92016
Epoch [2/3], Step [1900/6471], Loss: 2.0724, Perplexity: 7.94385
Epoch [2/3], Step [2000/6471], Loss: 2.2489, Perplexity: 9.47747
Epoch [2/3], Step [2100/6471], Loss: 2.2764, Perplexity: 9.74172
Epoch [2/3], Step [2200/6471], Loss: 2.2077, Perplexity: 9.09446
Epoch [2/3], Step [2300/6471], Loss: 2.1966, Perplexity: 8.99407
Epoch [2/3], Step [2400/6471], Loss: 1.9986, Perplexity: 7.37883
Epoch [2/3], Step [2500/6471], Loss: 1.9550, Perplexity: 7.06417
Epoch [2/3], Step [2600/6471], Loss: 2.2062, Perplexity: 9.081053
Epoch [2/3], Step [2700/6471], Loss: 1.9438, Perplexity: 6.98498
Epoch [2/3], Step [2800/6471], Loss: 1.9964, Perplexity: 7.36274
Epoch [2/3], Step [2900/6471], Loss: 1.9754, Perplexity: 7.20951
Epoch [2/3], Step [3000/6471], Loss: 2.2228, Perplexity: 9.23325
Epoch [2/3], Step [3100/6471], Loss: 2.1692, Perplexity: 8.75141
Epoch [2/3], Step [3200/6471], Loss: 1.8942, Perplexity: 6.64749
Epoch [2/3], Step [3300/6471], Loss: 1.9821, Perplexity: 7.25795
Epoch [2/3], Step [3400/6471], Loss: 2.0777, Perplexity: 7.98624
Epoch [2/3], Step [3500/6471], Loss: 2.0004, Perplexity: 7.39194
Epoch [2/3], Step [3600/6471], Loss: 2.1593, Perplexity: 8.66541
```

```
Epoch [2/3], Step [3700/6471], Loss: 2.1445, Perplexity: 8.53756
Epoch [2/3], Step [3800/6471], Loss: 2.0050, Perplexity: 7.42596
Epoch [2/3], Step [3900/6471], Loss: 2.0372, Perplexity: 7.66939
Epoch [2/3], Step [4000/6471], Loss: 2.0336, Perplexity: 7.64149
Epoch [2/3], Step [4100/6471], Loss: 3.1058, Perplexity: 22.3270
Epoch [2/3], Step [4200/6471], Loss: 2.2521, Perplexity: 9.50730
Epoch [2/3], Step [4300/6471], Loss: 1.9717, Perplexity: 7.18319
Epoch [2/3], Step [4400/6471], Loss: 2.2120, Perplexity: 9.13384
Epoch [2/3], Step [4500/6471], Loss: 1.9522, Perplexity: 7.04404
Epoch [2/3], Step [4600/6471], Loss: 2.0377, Perplexity: 7.67303
Epoch [2/3], Step [4700/6471], Loss: 2.0620, Perplexity: 7.86147
Epoch [2/3], Step [4800/6471], Loss: 2.8517, Perplexity: 17.3174
Epoch [2/3], Step [4900/6471], Loss: 1.9229, Perplexity: 6.84064
Epoch [2/3], Step [5000/6471], Loss: 2.0230, Perplexity: 7.56077
Epoch [2/3], Step [5100/6471], Loss: 2.0016, Perplexity: 7.40098
Epoch [2/3], Step [5200/6471], Loss: 2.6352, Perplexity: 13.9456
Epoch [2/3], Step [5300/6471], Loss: 2.3402, Perplexity: 10.3835
Epoch [2/3], Step [5400/6471], Loss: 2.1860, Perplexity: 8.89985
Epoch [2/3], Step [5500/6471], Loss: 2.1107, Perplexity: 8.25405
Epoch [2/3], Step [5600/6471], Loss: 1.9740, Perplexity: 7.19927
Epoch [2/3], Step [5700/6471], Loss: 2.2783, Perplexity: 9.76023
Epoch [2/3], Step [5800/6471], Loss: 1.9558, Perplexity: 7.06952
Epoch [2/3], Step [5900/6471], Loss: 1.9030, Perplexity: 6.70629
Epoch [2/3], Step [6000/6471], Loss: 2.3091, Perplexity: 10.0651
Epoch [2/3], Step [6100/6471], Loss: 2.0124, Perplexity: 7.48094
Epoch [2/3], Step [6200/6471], Loss: 2.0242, Perplexity: 7.56975
Epoch [2/3], Step [6300/6471], Loss: 2.1163, Perplexity: 8.30013
Epoch [2/3], Step [6400/6471], Loss: 2.5960, Perplexity: 13.4098
Epoch [3/3], Step [100/6471], Loss: 2.3987, Perplexity: 11.00929
Epoch [3/3], Step [200/6471], Loss: 1.8495, Perplexity: 6.35644
Epoch [3/3], Step [300/6471], Loss: 2.2262, Perplexity: 9.26443
Epoch [3/3], Step [400/6471], Loss: 1.9155, Perplexity: 6.79015
Epoch [3/3], Step [500/6471], Loss: 2.1346, Perplexity: 8.45383
Epoch [3/3], Step [600/6471], Loss: 1.8576, Perplexity: 6.40831
Epoch [3/3], Step [700/6471], Loss: 2.1442, Perplexity: 8.53497
Epoch [3/3], Step [800/6471], Loss: 2.1889, Perplexity: 8.92573
Epoch [3/3], Step [900/6471], Loss: 1.9085, Perplexity: 6.74307
Epoch [3/3], Step [1000/6471], Loss: 2.0333, Perplexity: 7.6391
Epoch [3/3], Step [1100/6471], Loss: 1.8304, Perplexity: 6.23611
Epoch [3/3], Step [1200/6471], Loss: 2.0623, Perplexity: 7.86361
Epoch [3/3], Step [1300/6471], Loss: 1.8422, Perplexity: 6.31049
Epoch [3/3], Step [1400/6471], Loss: 2.0546, Perplexity: 7.80352
Epoch [3/3], Step [1500/6471], Loss: 1.8926, Perplexity: 6.636740
Epoch [3/3], Step [1600/6471], Loss: 2.0801, Perplexity: 8.00526
Epoch [3/3], Step [1700/6471], Loss: 1.9666, Perplexity: 7.14644
Epoch [3/3], Step [1800/6471], Loss: 1.9712, Perplexity: 7.17918
Epoch [3/3], Step [1900/6471], Loss: 1.9185, Perplexity: 6.81087
Epoch [3/3], Step [2000/6471], Loss: 1.9382, Perplexity: 6.94636
```

```
Epoch [3/3], Step [2100/6471], Loss: 2.0318, Perplexity: 7.62773
Epoch [3/3], Step [2200/6471], Loss: 1.9210, Perplexity: 6.82797
Epoch [3/3], Step [2300/6471], Loss: 1.7989, Perplexity: 6.04334
Epoch [3/3], Step [2400/6471], Loss: 1.9430, Perplexity: 6.98007
Epoch [3/3], Step [2500/6471], Loss: 1.8499, Perplexity: 6.35936
Epoch [3/3], Step [2600/6471], Loss: 2.0787, Perplexity: 7.99425
Epoch [3/3], Step [2700/6471], Loss: 1.9080, Perplexity: 6.73944
Epoch [3/3], Step [2800/6471], Loss: 1.8390, Perplexity: 6.29028
Epoch [3/3], Step [2900/6471], Loss: 1.8147, Perplexity: 6.13939
Epoch [3/3], Step [3000/6471], Loss: 2.0982, Perplexity: 8.15170
Epoch [3/3], Step [3100/6471], Loss: 2.1538, Perplexity: 8.61732
Epoch [3/3], Step [3200/6471], Loss: 1.8499, Perplexity: 6.35946
Epoch [3/3], Step [3300/6471], Loss: 1.8313, Perplexity: 6.24228
Epoch [3/3], Step [3400/6471], Loss: 1.6785, Perplexity: 5.35750
Epoch [3/3], Step [3500/6471], Loss: 1.9427, Perplexity: 6.97793
Epoch [3/3], Step [3600/6471], Loss: 1.9375, Perplexity: 6.94137
Epoch [3/3], Step [3700/6471], Loss: 2.1000, Perplexity: 8.16625
Epoch [3/3], Step [3800/6471], Loss: 1.9315, Perplexity: 6.89988
Epoch [3/3], Step [3900/6471], Loss: 1.7446, Perplexity: 5.72388
Epoch [3/3], Step [4000/6471], Loss: 1.7892, Perplexity: 5.98480
Epoch [3/3], Step [4100/6471], Loss: 2.0860, Perplexity: 8.05265
Epoch [3/3], Step [4200/6471], Loss: 1.8918, Perplexity: 6.63159
Epoch [3/3], Step [4300/6471], Loss: 1.7105, Perplexity: 5.53151
Epoch [3/3], Step [4400/6471], Loss: 2.1300, Perplexity: 8.41530
Epoch [3/3], Step [4500/6471], Loss: 1.9628, Perplexity: 7.11950
Epoch [3/3], Step [4600/6471], Loss: 2.0803, Perplexity: 8.00662
Epoch [3/3], Step [4700/6471], Loss: 1.9536, Perplexity: 7.05413
Epoch [3/3], Step [4800/6471], Loss: 2.3566, Perplexity: 10.5547
Epoch [3/3], Step [4900/6471], Loss: 1.7959, Perplexity: 6.02478
Epoch [3/3], Step [5000/6471], Loss: 2.0553, Perplexity: 7.80951
Epoch [3/3], Step [5100/6471], Loss: 1.8084, Perplexity: 6.10052
Epoch [3/3], Step [5200/6471], Loss: 1.8611, Perplexity: 6.43075
Epoch [3/3], Step [5300/6471], Loss: 1.8553, Perplexity: 6.39393
Epoch [3/3], Step [5400/6471], Loss: 1.8719, Perplexity: 6.50056
Epoch [3/3], Step [5500/6471], Loss: 1.7134, Perplexity: 5.54781
Epoch [3/3], Step [5600/6471], Loss: 1.9495, Perplexity: 7.02508
Epoch [3/3], Step [5700/6471], Loss: 1.8084, Perplexity: 6.10093
Epoch [3/3], Step [5800/6471], Loss: 2.0376, Perplexity: 7.67258
Epoch [3/3], Step [5900/6471], Loss: 1.8882, Perplexity: 6.60771
Epoch [3/3], Step [6000/6471], Loss: 2.4541, Perplexity: 11.6358
Epoch [3/3], Step [6100/6471], Loss: 1.8122, Perplexity: 6.12383
Epoch [3/3], Step [6200/6471], Loss: 2.0369, Perplexity: 7.66670
Epoch [3/3], Step [6300/6471], Loss: 1.8107, Perplexity: 6.11461
Epoch [3/3], Step [6400/6471], Loss: 1.8466, Perplexity: 6.33858
Epoch [3/3], Step [6471/6471], Loss: 2.1778, Perplexity: 8.82671
```

Step 3: (Optional) Validate your Model
To assess potential overfitting, one approach is to assess performance on a validation set. If

you decide to do this **optional** task, you are required to first complete all of the steps in the next notebook in the sequence (**3_Inference.ipynb**); as part of that notebook, you will write and test code (specifically, the `sample` method in the `DecoderRNN` class) that uses your RNN decoder to generate captions. That code will prove incredibly useful here.

If you decide to validate your model, please do not edit the data loader in **data_loader.py**. Instead, create a new file named **data_loader_val.py** containing the code for obtaining the data loader for the validation data. You can access: - the validation images at filepath `'/opt/cocoapi/images/train2014/'`, and - the validation image caption annotation file at filepath `'/opt/cocoapi/annotations/captions_val2014.json'`.

The suggested approach to validating your model involves creating a json file such as this one containing your model's predicted captions for the validation images. Then, you can write your own script or use one that you find online to calculate the BLEU score of your model. You can read more about the BLEU score, along with other evaluation metrics (such as TEOR and Cider) in section 4.1 of this paper. For more information about how to use the annotation file, check out the website for the COCO dataset.

```
In [ ]: # (Optional) TODO: Validate your model.
```