2_Training

May 19, 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 ith 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: Referred this link. Added a batch normalization to the CNN layer as it shows better results. Batch_size = 64, to balance the computation time and efficiency Embed_size = 512, allows a larger vector storage capacity Hidden_size = 1024, intuition based

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: The mentioned transforms are pretty standard, and gave satisfactory results.

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: The link mentioned in Qu.1 recommends to add bn layer to the trainable parameters, thus added list(encoder.bn.parameters()) to the list of params

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: Adams is shown to have a better performance, thus selected Adams over SGD

```
In [1]: import torch
        import torch.nn as nn
        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


        ## TODO #1: Select appropriate values for the Python variables below.
        batch_size = 64          # batch size
        vocab_threshold = 5        # minimum word count threshold
        vocab_from_file = True    # if True, load existing vocab file
        embed_size = 512           # dimensionality of image and word embeddings
        hidden_size = 1024          # 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()) + list(encoder.bn

# TODO #4: Define the optimizer.
optimizer = torch.optim.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...
Done (t=0.89s)
creating index...


  0%|          | 774/414113 [00:00<01:52, 3690.36it/s]

index created!
Obtaining caption lengths...


100%|| 414113/414113 [01:31<00:00, 4514.92it/s]
```

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 [2]: 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()
```

```python
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: 3.4496, Perplexity: 31.4869
Epoch [1/3], Step [200/6471], Loss: 3.2413, Perplexity: 25.5671
Epoch [1/3], Step [300/6471], Loss: 2.9972, Perplexity: 20.0290
Epoch [1/3], Step [400/6471], Loss: 2.8860, Perplexity: 17.9213
Epoch [1/3], Step [500/6471], Loss: 2.9215, Perplexity: 18.5687
Epoch [1/3], Step [600/6471], Loss: 2.6081, Perplexity: 13.5729
Epoch [1/3], Step [700/6471], Loss: 2.9384, Perplexity: 18.8853
Epoch [1/3], Step [800/6471], Loss: 2.8055, Perplexity: 16.5346
Epoch [1/3], Step [900/6471], Loss: 2.6160, Perplexity: 13.6812
Epoch [1/3], Step [1000/6471], Loss: 2.5839, Perplexity: 13.2483
Epoch [1/3], Step [1100/6471], Loss: 2.6617, Perplexity: 14.32022
Epoch [1/3], Step [1200/6471], Loss: 2.5574, Perplexity: 12.9028
Epoch [1/3], Step [1300/6471], Loss: 2.3604, Perplexity: 10.5954
Epoch [1/3], Step [1400/6471], Loss: 2.2124, Perplexity: 9.13767
Epoch [1/3], Step [1500/6471], Loss: 2.3872, Perplexity: 10.8830
Epoch [1/3], Step [1600/6471], Loss: 2.7700, Perplexity: 15.9589
Epoch [1/3], Step [1700/6471], Loss: 2.0472, Perplexity: 7.746433
Epoch [1/3], Step [1800/6471], Loss: 2.3630, Perplexity: 10.6228
Epoch [1/3], Step [1900/6471], Loss: 2.2482, Perplexity: 9.47099
Epoch [1/3], Step [2000/6471], Loss: 2.5096, Perplexity: 12.3001
Epoch [1/3], Step [2100/6471], Loss: 2.5194, Perplexity: 12.4209
Epoch [1/3], Step [2200/6471], Loss: 2.3181, Perplexity: 10.1562
Epoch [1/3], Step [2300/6471], Loss: 2.0874, Perplexity: 8.06406
Epoch [1/3], Step [2400/6471], Loss: 2.3880, Perplexity: 10.8911
Epoch [1/3], Step [2500/6471], Loss: 2.2386, Perplexity: 9.38025
Epoch [1/3], Step [2600/6471], Loss: 2.2875, Perplexity: 9.85063
Epoch [1/3], Step [2700/6471], Loss: 2.3726, Perplexity: 10.7252
Epoch [1/3], Step [2800/6471], Loss: 2.0853, Perplexity: 8.04690
Epoch [1/3], Step [2900/6471], Loss: 2.1083, Perplexity: 8.23439
```

```
Epoch [1/3], Step [3000/6471], Loss: 2.0777, Perplexity: 7.98597
Epoch [1/3], Step [3100/6471], Loss: 2.4919, Perplexity: 12.0845
Epoch [1/3], Step [3200/6471], Loss: 2.5515, Perplexity: 12.8264
Epoch [1/3], Step [3300/6471], Loss: 2.1462, Perplexity: 8.55228
Epoch [1/3], Step [3400/6471], Loss: 2.3275, Perplexity: 10.2522
Epoch [1/3], Step [3500/6471], Loss: 2.5076, Perplexity: 12.2753
Epoch [1/3], Step [3600/6471], Loss: 2.6803, Perplexity: 14.5894
Epoch [1/3], Step [3700/6471], Loss: 2.1618, Perplexity: 8.68659
Epoch [1/3], Step [3800/6471], Loss: 2.2131, Perplexity: 9.14419
Epoch [1/3], Step [3900/6471], Loss: 2.1700, Perplexity: 8.75828
Epoch [1/3], Step [4000/6471], Loss: 2.1830, Perplexity: 8.87266
Epoch [1/3], Step [4100/6471], Loss: 2.5401, Perplexity: 12.6812
Epoch [1/3], Step [4200/6471], Loss: 2.3128, Perplexity: 10.1029
Epoch [1/3], Step [4300/6471], Loss: 2.0512, Perplexity: 7.77733
Epoch [1/3], Step [4400/6471], Loss: 1.8835, Perplexity: 6.57628
Epoch [1/3], Step [4500/6471], Loss: 2.6251, Perplexity: 13.8065
Epoch [1/3], Step [4600/6471], Loss: 2.1047, Perplexity: 8.20441
Epoch [1/3], Step [4700/6471], Loss: 1.9718, Perplexity: 7.18380
Epoch [1/3], Step [4800/6471], Loss: 2.2821, Perplexity: 9.79725
Epoch [1/3], Step [4900/6471], Loss: 2.0171, Perplexity: 7.51658
Epoch [1/3], Step [5000/6471], Loss: 2.1596, Perplexity: 8.66760
Epoch [1/3], Step [5100/6471], Loss: 1.9498, Perplexity: 7.02708
Epoch [1/3], Step [5200/6471], Loss: 2.1205, Perplexity: 8.33541
Epoch [1/3], Step [5300/6471], Loss: 2.1505, Perplexity: 8.58938
Epoch [1/3], Step [5400/6471], Loss: 2.0783, Perplexity: 7.99111
Epoch [1/3], Step [5500/6471], Loss: 1.9695, Perplexity: 7.16733
Epoch [1/3], Step [5600/6471], Loss: 2.0544, Perplexity: 7.80181
Epoch [1/3], Step [5700/6471], Loss: 2.2629, Perplexity: 9.61077
Epoch [1/3], Step [5800/6471], Loss: 2.3906, Perplexity: 10.9204
Epoch [1/3], Step [5900/6471], Loss: 2.0939, Perplexity: 8.11658
Epoch [1/3], Step [6000/6471], Loss: 2.3690, Perplexity: 10.6870
Epoch [1/3], Step [6100/6471], Loss: 2.3881, Perplexity: 10.8929
Epoch [1/3], Step [6200/6471], Loss: 1.9355, Perplexity: 6.92764
Epoch [1/3], Step [6300/6471], Loss: 2.3930, Perplexity: 10.9464
Epoch [1/3], Step [6400/6471], Loss: 2.2833, Perplexity: 9.80947
Epoch [2/3], Step [100/6471], Loss: 1.9616, Perplexity: 7.110880
Epoch [2/3], Step [200/6471], Loss: 2.1354, Perplexity: 8.46053
Epoch [2/3], Step [300/6471], Loss: 2.1007, Perplexity: 8.17225
Epoch [2/3], Step [400/6471], Loss: 2.5567, Perplexity: 12.8934
Epoch [2/3], Step [500/6471], Loss: 1.9581, Perplexity: 7.08589
Epoch [2/3], Step [600/6471], Loss: 2.0653, Perplexity: 7.88782
Epoch [2/3], Step [700/6471], Loss: 2.1548, Perplexity: 8.62595
Epoch [2/3], Step [800/6471], Loss: 1.9250, Perplexity: 6.85497
Epoch [2/3], Step [900/6471], Loss: 2.2020, Perplexity: 9.04340
Epoch [2/3], Step [1000/6471], Loss: 1.9626, Perplexity: 7.1179
Epoch [2/3], Step [1100/6471], Loss: 1.8412, Perplexity: 6.30432
Epoch [2/3], Step [1200/6471], Loss: 2.0473, Perplexity: 7.74730
Epoch [2/3], Step [1300/6471], Loss: 2.0749, Perplexity: 7.96413
```

```
Epoch [2/3], Step [1400/6471], Loss: 1.9225, Perplexity: 6.83775
Epoch [2/3], Step [1500/6471], Loss: 2.1455, Perplexity: 8.54636
Epoch [2/3], Step [1600/6471], Loss: 1.7313, Perplexity: 5.64780
Epoch [2/3], Step [1700/6471], Loss: 1.9483, Perplexity: 7.01653
Epoch [2/3], Step [1800/6471], Loss: 2.0001, Perplexity: 7.38962
Epoch [2/3], Step [1900/6471], Loss: 2.1342, Perplexity: 8.45062
Epoch [2/3], Step [2000/6471], Loss: 2.2910, Perplexity: 9.88482
Epoch [2/3], Step [2100/6471], Loss: 1.7362, Perplexity: 5.67574
Epoch [2/3], Step [2200/6471], Loss: 1.8120, Perplexity: 6.12291
Epoch [2/3], Step [2300/6471], Loss: 2.0049, Perplexity: 7.42524
Epoch [2/3], Step [2400/6471], Loss: 2.0890, Perplexity: 8.07707
Epoch [2/3], Step [2500/6471], Loss: 2.1624, Perplexity: 8.69178
Epoch [2/3], Step [2600/6471], Loss: 1.9142, Perplexity: 6.78135
Epoch [2/3], Step [2700/6471], Loss: 1.9021, Perplexity: 6.69980
Epoch [2/3], Step [2800/6471], Loss: 2.1834, Perplexity: 8.87638
Epoch [2/3], Step [2900/6471], Loss: 2.0149, Perplexity: 7.49976
Epoch [2/3], Step [3000/6471], Loss: 1.8535, Perplexity: 6.38200
Epoch [2/3], Step [3100/6471], Loss: 2.0112, Perplexity: 7.47227
Epoch [2/3], Step [3200/6471], Loss: 1.8104, Perplexity: 6.11307
Epoch [2/3], Step [3300/6471], Loss: 1.9370, Perplexity: 6.93799
Epoch [2/3], Step [3400/6471], Loss: 2.1749, Perplexity: 8.80149
Epoch [2/3], Step [3500/6471], Loss: 1.8806, Perplexity: 6.55727
Epoch [2/3], Step [3600/6471], Loss: 2.8469, Perplexity: 17.2343
Epoch [2/3], Step [3700/6471], Loss: 2.1512, Perplexity: 8.59534
Epoch [2/3], Step [3800/6471], Loss: 2.0924, Perplexity: 8.10453
Epoch [2/3], Step [3900/6471], Loss: 1.9388, Perplexity: 6.95033
Epoch [2/3], Step [4000/6471], Loss: 2.2226, Perplexity: 9.23141
Epoch [2/3], Step [4100/6471], Loss: 1.9835, Perplexity: 7.26828
Epoch [2/3], Step [4200/6471], Loss: 1.9802, Perplexity: 7.24441
Epoch [2/3], Step [4300/6471], Loss: 2.2400, Perplexity: 9.39295
Epoch [2/3], Step [4400/6471], Loss: 1.8699, Perplexity: 6.48803
Epoch [2/3], Step [4500/6471], Loss: 2.0358, Perplexity: 7.65812
Epoch [2/3], Step [4600/6471], Loss: 2.0359, Perplexity: 7.65887
Epoch [2/3], Step [4700/6471], Loss: 1.9803, Perplexity: 7.24528
Epoch [2/3], Step [4800/6471], Loss: 1.9107, Perplexity: 6.75802
Epoch [2/3], Step [4900/6471], Loss: 1.9006, Perplexity: 6.68993
Epoch [2/3], Step [5000/6471], Loss: 2.5329, Perplexity: 12.5900
Epoch [2/3], Step [5100/6471], Loss: 2.1703, Perplexity: 8.76073
Epoch [2/3], Step [5200/6471], Loss: 1.8093, Perplexity: 6.10636
Epoch [2/3], Step [5300/6471], Loss: 1.9636, Perplexity: 7.12504
Epoch [2/3], Step [5400/6471], Loss: 1.9124, Perplexity: 6.76960
Epoch [2/3], Step [5500/6471], Loss: 2.0067, Perplexity: 7.43841
Epoch [2/3], Step [5600/6471], Loss: 1.9142, Perplexity: 6.78187
Epoch [2/3], Step [5700/6471], Loss: 2.2020, Perplexity: 9.04293
Epoch [2/3], Step [5800/6471], Loss: 1.9516, Perplexity: 7.03994
Epoch [2/3], Step [5900/6471], Loss: 1.8537, Perplexity: 6.38318
Epoch [2/3], Step [6000/6471], Loss: 1.7325, Perplexity: 5.65484
Epoch [2/3], Step [6100/6471], Loss: 1.8350, Perplexity: 6.26522
```

```
Epoch [2/3], Step [6200/6471], Loss: 1.8229, Perplexity: 6.19001
Epoch [2/3], Step [6300/6471], Loss: 2.1229, Perplexity: 8.35520
Epoch [2/3], Step [6400/6471], Loss: 1.9615, Perplexity: 7.10987
Epoch [3/3], Step [100/6471], Loss: 2.5852, Perplexity: 13.26594
Epoch [3/3], Step [200/6471], Loss: 1.7203, Perplexity: 5.58610
Epoch [3/3], Step [300/6471], Loss: 1.7847, Perplexity: 5.95780
Epoch [3/3], Step [400/6471], Loss: 1.9129, Perplexity: 6.77240
Epoch [3/3], Step [500/6471], Loss: 1.8573, Perplexity: 6.40670
Epoch [3/3], Step [600/6471], Loss: 1.7937, Perplexity: 6.011890
Epoch [3/3], Step [700/6471], Loss: 2.1968, Perplexity: 8.99622
Epoch [3/3], Step [800/6471], Loss: 1.8785, Perplexity: 6.54382
Epoch [3/3], Step [900/6471], Loss: 2.0551, Perplexity: 7.80791
Epoch [3/3], Step [1000/6471], Loss: 2.0490, Perplexity: 7.7598
Epoch [3/3], Step [1100/6471], Loss: 1.6858, Perplexity: 5.39677
Epoch [3/3], Step [1200/6471], Loss: 2.0042, Perplexity: 7.42040
Epoch [3/3], Step [1300/6471], Loss: 2.1700, Perplexity: 8.75820
Epoch [3/3], Step [1400/6471], Loss: 1.9718, Perplexity: 7.18351
Epoch [3/3], Step [1500/6471], Loss: 1.7969, Perplexity: 6.03122
Epoch [3/3], Step [1600/6471], Loss: 1.7627, Perplexity: 5.82821
Epoch [3/3], Step [1700/6471], Loss: 1.9831, Perplexity: 7.26539
Epoch [3/3], Step [1800/6471], Loss: 1.8221, Perplexity: 6.18498
Epoch [3/3], Step [1900/6471], Loss: 1.7947, Perplexity: 6.01782
Epoch [3/3], Step [2000/6471], Loss: 2.0236, Perplexity: 7.56572
Epoch [3/3], Step [2100/6471], Loss: 1.9903, Perplexity: 7.31817
Epoch [3/3], Step [2200/6471], Loss: 1.7837, Perplexity: 5.95201
Epoch [3/3], Step [2300/6471], Loss: 1.7516, Perplexity: 5.76382
Epoch [3/3], Step [2400/6471], Loss: 1.8785, Perplexity: 6.54393
Epoch [3/3], Step [2500/6471], Loss: 1.9477, Perplexity: 7.01226
Epoch [3/3], Step [2600/6471], Loss: 1.8712, Perplexity: 6.49622
Epoch [3/3], Step [2700/6471], Loss: 1.8515, Perplexity: 6.36935
Epoch [3/3], Step [2800/6471], Loss: 1.9215, Perplexity: 6.83128
Epoch [3/3], Step [2900/6471], Loss: 1.8353, Perplexity: 6.26687
Epoch [3/3], Step [3000/6471], Loss: 1.9081, Perplexity: 6.74048
Epoch [3/3], Step [3100/6471], Loss: 1.7713, Perplexity: 5.87874
Epoch [3/3], Step [3200/6471], Loss: 1.9226, Perplexity: 6.83848
Epoch [3/3], Step [3300/6471], Loss: 1.7704, Perplexity: 5.87300
Epoch [3/3], Step [3400/6471], Loss: 1.6906, Perplexity: 5.42280
Epoch [3/3], Step [3500/6471], Loss: 1.7148, Perplexity: 5.55560
Epoch [3/3], Step [3600/6471], Loss: 1.9869, Perplexity: 7.29288
Epoch [3/3], Step [3700/6471], Loss: 1.8099, Perplexity: 6.10983
Epoch [3/3], Step [3800/6471], Loss: 1.6326, Perplexity: 5.11736
Epoch [3/3], Step [3900/6471], Loss: 1.8602, Perplexity: 6.42494
Epoch [3/3], Step [4000/6471], Loss: 1.8446, Perplexity: 6.32577
Epoch [3/3], Step [4100/6471], Loss: 1.7756, Perplexity: 5.90389
Epoch [3/3], Step [4200/6471], Loss: 1.8321, Perplexity: 6.24683
Epoch [3/3], Step [4300/6471], Loss: 1.9077, Perplexity: 6.73788
Epoch [3/3], Step [4400/6471], Loss: 2.0595, Perplexity: 7.84240
Epoch [3/3], Step [4500/6471], Loss: 1.7138, Perplexity: 5.55001
```

```
Epoch [3/3], Step [4600/6471], Loss: 1.8632, Perplexity: 6.44427
Epoch [3/3], Step [4700/6471], Loss: 1.9136, Perplexity: 6.77772
Epoch [3/3], Step [4800/6471], Loss: 2.8823, Perplexity: 17.8550
Epoch [3/3], Step [4900/6471], Loss: 1.8191, Perplexity: 6.16638
Epoch [3/3], Step [5000/6471], Loss: 1.8808, Perplexity: 6.55849
Epoch [3/3], Step [5100/6471], Loss: 1.9360, Perplexity: 6.93084
Epoch [3/3], Step [5200/6471], Loss: 1.8982, Perplexity: 6.67405
Epoch [3/3], Step [5300/6471], Loss: 1.9644, Perplexity: 7.13094
Epoch [3/3], Step [5400/6471], Loss: 1.5988, Perplexity: 4.94711
Epoch [3/3], Step [5500/6471], Loss: 2.1045, Perplexity: 8.20309
Epoch [3/3], Step [5600/6471], Loss: 1.9374, Perplexity: 6.940746
Epoch [3/3], Step [5700/6471], Loss: 2.0188, Perplexity: 7.52916
Epoch [3/3], Step [5800/6471], Loss: 1.9901, Perplexity: 7.31618
Epoch [3/3], Step [5900/6471], Loss: 1.8714, Perplexity: 6.49765
Epoch [3/3], Step [6000/6471], Loss: 1.9586, Perplexity: 7.08973
Epoch [3/3], Step [6100/6471], Loss: 1.5651, Perplexity: 4.78333
Epoch [3/3], Step [6200/6471], Loss: 1.7587, Perplexity: 5.80529
Epoch [3/3], Step [6300/6471], Loss: 1.7124, Perplexity: 5.54231
Epoch [3/3], Step [6400/6471], Loss: 2.1947, Perplexity: 8.97749
Epoch [3/3], Step [6471/6471], Loss: 1.7991, Perplexity: 6.04444
```

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