2_Training

May 17, 2021

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

2

2.1 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**.

2.1.1 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.

2.1.2 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:

CNN architecture: ResNet-50 was already provided. ResNet (residual networks) fixes the vanishing gradient problem by using skip connections. This allows for deeper network architectures.

RNN architecture: Similar to the architecture described in the suggested paper (1411.4555.pdf). I used the numbers from the article: 512 dimensions for the embeddings and the size of the LSTM memory. Kept all words that appeared at least 5 times in the training set.

2.1.3 (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.

2.1.4 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 used the provided value. The provided transform includes resizing, random cropping, horizontal flipping, and normalization. These are widely used techniques so I did not see the need to modify them.

2.1.5 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())
```

2.1.6 Question 3

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

Answer:

As mentioned above, I used `params = list(decoder.parameters()) + list(encoder.embed.parameters())` The decoder weights are trainable because that contains our RNN. In the encoder, the CNN is pre-trained so we don't need to include that. However the embedding layer of the encoder is the linear layer we added and we need to train that. It needed to be included. I was not sure whether I should have added `list(encoder.batchNorm1d.parameters())`. If I had more time, I would have tried it as well.

I used the numbers from the article: 512 dimensions for the embeddings and the size of the LSTM memory. Kept all words that appeared at least 5 times in the training set.

I used 3 epochs as it was recommended. Kept my batch size at 128. Maybe I could have used a smaller number.

2.1.7 Task #4

Finally, you will select an optimizer.

2.1.8 Question 4

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

Answer:

I chose the Adam optimizer. It was used in the other recommended paper (1502.03044.pdf). I also know that it is a very popular optimizer that produces good results efficiently.

In [3]:
```
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
```

```python
import math


## TODO #1: Select appropriate values for the Python variables below.
batch_size = 128          # 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 = 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())
```

```python
    # TODO #4: Define the optimizer.
    optimizer = torch.optim.Adam(params, lr=0.01, betas=(0.9, 0.999), eps=1e-08)
    # optimizer = torch.optim.RMSprop()

    # 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...


  0%|          | 932/414113 [00:00<01:30, 4556.92it/s]

Done (t=0.85s)
creating index...
index created!
Obtaining caption lengths...


100%|| 414113/414113 [01:23<00:00, 4980.29it/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:

```python
# 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 :).

2.1.9 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 [4]: import torch.utils.data as data
        import numpy as np
        import os
        import requests
        import time
        from workspace_utils import keep_awake

        # 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 i in keep_awake(range(5)):
        #for epoch in range(1, num_epochs+1):
        for epoch in keep_awake(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 [100/3236], Loss: 3.0339, Perplexity: 20.7781
Epoch [1/3], Step [200/3236], Loss: 2.8270, Perplexity: 16.8947
Epoch [1/3], Step [300/3236], Loss: 2.6725, Perplexity: 14.4754
```

```
Epoch [1/3], Step [400/3236], Loss: 2.8468, Perplexity: 17.2331
Epoch [1/3], Step [500/3236], Loss: 2.6792, Perplexity: 14.5739
Epoch [1/3], Step [600/3236], Loss: 2.6330, Perplexity: 13.9161
Epoch [1/3], Step [700/3236], Loss: 2.5876, Perplexity: 13.2981
Epoch [1/3], Step [800/3236], Loss: 2.5862, Perplexity: 13.2798
Epoch [1/3], Step [900/3236], Loss: 2.5392, Perplexity: 12.6701
Epoch [1/3], Step [1000/3236], Loss: 2.5358, Perplexity: 12.6261
Epoch [1/3], Step [1100/3236], Loss: 2.3731, Perplexity: 10.7304
Epoch [1/3], Step [1200/3236], Loss: 2.5037, Perplexity: 12.2281
Epoch [1/3], Step [1300/3236], Loss: 2.2853, Perplexity: 9.82881
Epoch [1/3], Step [1400/3236], Loss: 3.2270, Perplexity: 25.2033
Epoch [1/3], Step [1500/3236], Loss: 2.4753, Perplexity: 11.8853
Epoch [1/3], Step [1600/3236], Loss: 2.5056, Perplexity: 12.2507
Epoch [1/3], Step [1700/3236], Loss: 2.7823, Perplexity: 16.1566
Epoch [1/3], Step [1800/3236], Loss: 2.4047, Perplexity: 11.07572
Epoch [1/3], Step [1900/3236], Loss: 2.2074, Perplexity: 9.09216
Epoch [1/3], Step [2000/3236], Loss: 2.9011, Perplexity: 18.1935
Epoch [1/3], Step [2100/3236], Loss: 2.5184, Perplexity: 12.4082
Epoch [1/3], Step [2200/3236], Loss: 2.3938, Perplexity: 10.9549
Epoch [1/3], Step [2300/3236], Loss: 2.2122, Perplexity: 9.13544
Epoch [1/3], Step [2400/3236], Loss: 2.6340, Perplexity: 13.9289
Epoch [1/3], Step [2500/3236], Loss: 2.8338, Perplexity: 17.0104
Epoch [1/3], Step [2600/3236], Loss: 2.3017, Perplexity: 9.99136
Epoch [1/3], Step [2700/3236], Loss: 2.4887, Perplexity: 12.0460
Epoch [1/3], Step [2800/3236], Loss: 2.4049, Perplexity: 11.0771
Epoch [1/3], Step [2900/3236], Loss: 3.0470, Perplexity: 21.0529
Epoch [1/3], Step [3000/3236], Loss: 2.3758, Perplexity: 10.7600
Epoch [1/3], Step [3100/3236], Loss: 2.4878, Perplexity: 12.0349
Epoch [1/3], Step [3200/3236], Loss: 2.2859, Perplexity: 9.83465
Epoch [2/3], Step [100/3236], Loss: 2.3331, Perplexity: 10.30988
Epoch [2/3], Step [200/3236], Loss: 2.4746, Perplexity: 11.8775
Epoch [2/3], Step [300/3236], Loss: 2.3124, Perplexity: 10.0984
Epoch [2/3], Step [400/3236], Loss: 2.1978, Perplexity: 9.00528
Epoch [2/3], Step [500/3236], Loss: 2.5261, Perplexity: 12.5051
Epoch [2/3], Step [600/3236], Loss: 2.6314, Perplexity: 13.8936
Epoch [2/3], Step [700/3236], Loss: 2.2759, Perplexity: 9.73673
Epoch [2/3], Step [800/3236], Loss: 2.4400, Perplexity: 11.4735
Epoch [2/3], Step [900/3236], Loss: 2.7004, Perplexity: 14.8851
Epoch [2/3], Step [1000/3236], Loss: 2.4294, Perplexity: 11.3515
Epoch [2/3], Step [1100/3236], Loss: 2.4034, Perplexity: 11.0604
Epoch [2/3], Step [1200/3236], Loss: 2.4000, Perplexity: 11.0237
Epoch [2/3], Step [1300/3236], Loss: 2.3213, Perplexity: 10.1887
Epoch [2/3], Step [1400/3236], Loss: 2.2360, Perplexity: 9.35628
Epoch [2/3], Step [1500/3236], Loss: 2.6913, Perplexity: 14.7511
Epoch [2/3], Step [1600/3236], Loss: 2.5976, Perplexity: 13.4314
Epoch [2/3], Step [1700/3236], Loss: 2.3713, Perplexity: 10.7115
Epoch [2/3], Step [1800/3236], Loss: 2.5958, Perplexity: 13.4067
Epoch [2/3], Step [1900/3236], Loss: 2.6086, Perplexity: 13.5801
```

```
Epoch [2/3], Step [2000/3236], Loss: 2.4824, Perplexity: 11.9695
Epoch [2/3], Step [2100/3236], Loss: 2.3317, Perplexity: 10.2951
Epoch [2/3], Step [2200/3236], Loss: 2.5850, Perplexity: 13.2629
Epoch [2/3], Step [2300/3236], Loss: 2.4264, Perplexity: 11.3185
Epoch [2/3], Step [2400/3236], Loss: 2.4957, Perplexity: 12.1297
Epoch [2/3], Step [2500/3236], Loss: 3.4738, Perplexity: 32.2598
Epoch [2/3], Step [2600/3236], Loss: 2.4096, Perplexity: 11.1295
Epoch [2/3], Step [2700/3236], Loss: 2.3018, Perplexity: 9.992234
Epoch [2/3], Step [2800/3236], Loss: 2.6544, Perplexity: 14.2159
Epoch [2/3], Step [2900/3236], Loss: 2.3239, Perplexity: 10.2158
Epoch [2/3], Step [3000/3236], Loss: 2.4904, Perplexity: 12.0661
Epoch [2/3], Step [3100/3236], Loss: 2.1791, Perplexity: 8.83868
Epoch [2/3], Step [3200/3236], Loss: 2.2922, Perplexity: 9.89660
Epoch [3/3], Step [100/3236], Loss: 2.3674, Perplexity: 10.66936
Epoch [3/3], Step [200/3236], Loss: 2.6425, Perplexity: 14.0483
Epoch [3/3], Step [300/3236], Loss: 2.3667, Perplexity: 10.6616
Epoch [3/3], Step [400/3236], Loss: 2.8628, Perplexity: 17.5098
Epoch [3/3], Step [500/3236], Loss: 2.3614, Perplexity: 10.6054
Epoch [3/3], Step [600/3236], Loss: 2.2569, Perplexity: 9.55324
Epoch [3/3], Step [700/3236], Loss: 2.2689, Perplexity: 9.66884
Epoch [3/3], Step [800/3236], Loss: 2.2044, Perplexity: 9.06449
Epoch [3/3], Step [900/3236], Loss: 2.1759, Perplexity: 8.81059
Epoch [3/3], Step [1000/3236], Loss: 2.3838, Perplexity: 10.8465
Epoch [3/3], Step [1100/3236], Loss: 2.2011, Perplexity: 9.03526
Epoch [3/3], Step [1200/3236], Loss: 2.6362, Perplexity: 13.9599
Epoch [3/3], Step [1300/3236], Loss: 2.3275, Perplexity: 10.2525
Epoch [3/3], Step [1400/3236], Loss: 2.3084, Perplexity: 10.0578
Epoch [3/3], Step [1500/3236], Loss: 2.8751, Perplexity: 17.7277
Epoch [3/3], Step [1600/3236], Loss: 2.2476, Perplexity: 9.46520
Epoch [3/3], Step [1700/3236], Loss: 2.4107, Perplexity: 11.1417
Epoch [3/3], Step [1800/3236], Loss: 2.3353, Perplexity: 10.3330
Epoch [3/3], Step [1900/3236], Loss: 2.2303, Perplexity: 9.30310
Epoch [3/3], Step [2000/3236], Loss: 2.3643, Perplexity: 10.6364
Epoch [3/3], Step [2100/3236], Loss: 2.2641, Perplexity: 9.62245
Epoch [3/3], Step [2200/3236], Loss: 2.2221, Perplexity: 9.22705
Epoch [3/3], Step [2300/3236], Loss: 2.3581, Perplexity: 10.5712
Epoch [3/3], Step [2400/3236], Loss: 2.3424, Perplexity: 10.4060
Epoch [3/3], Step [2500/3236], Loss: 2.3053, Perplexity: 10.0270
Epoch [3/3], Step [2600/3236], Loss: 2.2252, Perplexity: 9.25536
Epoch [3/3], Step [2700/3236], Loss: 2.3265, Perplexity: 10.2420
Epoch [3/3], Step [2800/3236], Loss: 2.2236, Perplexity: 9.24068
Epoch [3/3], Step [2900/3236], Loss: 2.2126, Perplexity: 9.139455
Epoch [3/3], Step [3000/3236], Loss: 2.4471, Perplexity: 11.5551
Epoch [3/3], Step [3100/3236], Loss: 2.2153, Perplexity: 9.16375
Epoch [3/3], Step [3200/3236], Loss: 2.2270, Perplexity: 9.27232
Epoch [3/3], Step [3236/3236], Loss: 2.4623, Perplexity: 11.7321
```

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.
```