
 **kuubi3d-udacity** / **cv_image** Public


[Code](#) [Issues](#) [Pull requests](#) [Actions](#) [Projects](#) [Wiki](#) [Security](#) [Insights](#) [Settings](#)

 main


cv_image / 2_Training_v1.ipynb

Go to file

...

 **kuubi3d-udacity** trained_model

Latest commit 1490e59 13 minutes ago [History](#)

 1 contributor

478 lines (478 sloc) | 28.3 KB

[Code](#) [Raw](#) [Blame](#) [Edit](#) [Copy](#) [Share](#)

Computer Vision Nanodegree

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:

- [Step 1: Training Setup](#)
- [Step 2: Train your Model](#)
- [Step 3: \(Optional\) 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.**

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 `1`th epoch, the encoder and decoder weights will be saved in the `models/` folder as `encoder-1.pkl` and `decoder-1.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.

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:

- I'm using the pre-trained `ResNet-50` architecture for CNN which is provided by Udacity and described in the Preliminaries notebook.
- The RNN decoder follows the same architecture that was described in the paper referenced above ([arxiv:1411.4555](#)). The paper specifies values for several hyperparameters (`vocab_threshold=5`, `embed_size=hidden_size=512`) that I used.
- I also used a single hidden layer and no dropout and got reasonable results.

(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.

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 recommended image transform `transform_train` provided.

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())
```

Question 3

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

Answer:

- I used `list(decoder.parameters()) + list(encoder.embed.parameters())`. In the encoder section, we need to make the embedding layer learnable by containing important information about the image for the training step. Because the image captions are created by the RNN with values that contain meaningful information about the image in this layer.
- As image captions are generated by decoder all parameters in decoder should be learnable, LSTM weights must be influenced by all parameters of decoder during the training.

Task #4

Finally, you will select an `optimizer`.

Question 4

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

Answer:

- I decided to use the `Adam optimizer`. I had perviously used it in previous projects, it's easy to implement *much easier than other algorithms*, computationally efficient, with little memory requirements.

```
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 = 128          # batch size
vocab_threshold = 5       # minimum word count threshold
vocab_from_file = True    # if True, load existing vocab file
embed_size = 256         # dimensionality of image and word embeddings
```

```

embed_size = 256          # dimensionality of image and text 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 perplexity

# (Optional) TODO #2: Amend the image transform below.
transform_train = transforms.Compose([
    transforms.Resize(256),          # smaller edge of image resized to 256
    transforms.RandomCrop(224),      # get 224x224 crop from random location
    transforms.RandomHorizontalFlip(), # horizontally flip image with probability=0.5
    transforms.ToTensor(),           # convert the PIL Image to a tensor
    transforms.Normalize((0.485, 0.456, 0.406), # normalize image for pre-trained model
                        (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.

processor = ("cuda is available" if torch.cuda.is_available() else "cuda unavailable")
print(processor)

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

#device = torch.device("cpu")
print('device is', device)

encoder.to(device)
decoder.to(device)

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

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

# 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_sampler.batch_size)

#total_step = 5

```

```

Vocabulary successfully loaded from vocab.pkl file!
loading annotations into memory...
Done (t=1.43s)
creating index...
index created!
Obtaining caption lengths...

```

```

100%|██████████| 414113/414113 [00:40<00:00, 10233.00it/s]
/home/kuubi3d/.local/lib/python3.10/site-packages/torchvision/models/_utils.py:208: UserWarning: The parameter 'pretrained' is deprecated since 0.13 and may be removed in the future, please use 'weights' instead.
  warnings.warn(
/home/kuubi3d/.local/lib/python3.10/site-packages/torchvision/models/_utils.py:223: UserWarning: Arguments other than a weight enum or `None` for 'weights' are deprecated since 0.13 and may be removed in the future. The current behavior is equivalent to passing `weights=ResNet50_Weights.IMAGENET1K_V1`. You can also use `weights=ResNet50_Weights.DEFAULT` to get the most up-to-date weights.
  warnings.warn(msg)
cuda is available
device is cuda

```

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 :).

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()

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

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

        # 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: %.4f' % (epoch, num_epochs, i_step, total_step, loss.item(), np.exp
```

```
# 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' % epoch))
    torch.save(encoder.state_dict(), os.path.join('./models', 'encoder-%d.pkl' % epoch))

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

Epoch [1/3], Step [100/3236], Loss: 3.8114, Perplexity: 45.21427
Epoch [1/3], Step [200/3236], Loss: 3.3977, Perplexity: 29.8953
Epoch [1/3], Step [300/3236], Loss: 3.2841, Perplexity: 26.6861
Epoch [1/3], Step [400/3236], Loss: 3.2198, Perplexity: 25.0225
Epoch [1/3], Step [500/3236], Loss: 2.8024, Perplexity: 16.4834
Epoch [1/3], Step [600/3236], Loss: 2.9308, Perplexity: 18.7426
Epoch [1/3], Step [700/3236], Loss: 2.7849, Perplexity: 16.1985
Epoch [1/3], Step [800/3236], Loss: 2.7979, Perplexity: 16.4101
Epoch [1/3], Step [900/3236], Loss: 2.7649, Perplexity: 15.8771
Epoch [1/3], Step [1000/3236], Loss: 2.5896, Perplexity: 13.3244
Epoch [1/3], Step [1100/3236], Loss: 2.6126, Perplexity: 13.6342
Epoch [1/3], Step [1200/3236], Loss: 2.5511, Perplexity: 12.8214
Epoch [1/3], Step [1300/3236], Loss: 2.5776, Perplexity: 13.16605
Epoch [1/3], Step [1400/3236], Loss: 2.3809, Perplexity: 10.8148
Epoch [1/3], Step [1500/3236], Loss: 2.5840, Perplexity: 13.2506
Epoch [1/3], Step [1600/3236], Loss: 2.7795, Perplexity: 16.1114
Epoch [1/3], Step [1700/3236], Loss: 2.3545, Perplexity: 10.5328
Epoch [1/3], Step [1800/3236], Loss: 3.1796, Perplexity: 24.0368
Epoch [1/3], Step [1900/3236], Loss: 2.2126, Perplexity: 9.13948
Epoch [1/3], Step [2000/3236], Loss: 2.2359, Perplexity: 9.35538
Epoch [1/3], Step [2100/3236], Loss: 2.2405, Perplexity: 9.39834
Epoch [1/3], Step [2200/3236], Loss: 2.4759, Perplexity: 11.8923
Epoch [1/3], Step [2300/3236], Loss: 2.3041, Perplexity: 10.0154
Epoch [1/3], Step [2400/3236], Loss: 2.4747, Perplexity: 11.8785
Epoch [1/3], Step [2500/3236], Loss: 2.5797, Perplexity: 13.1933
Epoch [1/3], Step [2600/3236], Loss: 2.3522, Perplexity: 10.5091
Epoch [1/3], Step [2700/3236], Loss: 2.1930, Perplexity: 8.96225
Epoch [1/3], Step [2800/3236], Loss: 2.2473, Perplexity: 9.46231
Epoch [1/3], Step [2900/3236], Loss: 2.2250, Perplexity: 9.25381
Epoch [1/3], Step [3000/3236], Loss: 2.4310, Perplexity: 11.3705
Epoch [1/3], Step [3100/3236], Loss: 2.1155, Perplexity: 8.29356
Epoch [1/3], Step [3200/3236], Loss: 2.0380, Perplexity: 7.67522
Epoch [2/3], Step [100/3236], Loss: 2.6199, Perplexity: 13.73377
Epoch [2/3], Step [200/3236], Loss: 2.1491, Perplexity: 8.57705
Epoch [2/3], Step [300/3236], Loss: 2.2251, Perplexity: 9.25493
Epoch [2/3], Step [400/3236], Loss: 2.1621, Perplexity: 8.68904
Epoch [2/3], Step [500/3236], Loss: 2.1943, Perplexity: 8.97336
Epoch [2/3], Step [600/3236], Loss: 2.1697, Perplexity: 8.75583
Epoch [2/3], Step [700/3236], Loss: 2.1735, Perplexity: 8.78905
Epoch [2/3], Step [800/3236], Loss: 2.6933, Perplexity: 14.7811
Epoch [2/3], Step [900/3236], Loss: 2.1095, Perplexity: 8.24439
Epoch [2/3], Step [1000/3236], Loss: 2.2943, Perplexity: 9.9175
Epoch [2/3], Step [1100/3236], Loss: 2.1295, Perplexity: 8.41096
Epoch [2/3], Step [1200/3236], Loss: 2.1065, Perplexity: 8.219880
Epoch [2/3], Step [1300/3236], Loss: 2.1314, Perplexity: 8.42633
Epoch [2/3], Step [1400/3236], Loss: 2.1864, Perplexity: 8.90280
Epoch [2/3], Step [1500/3236], Loss: 2.0219, Perplexity: 7.55254
Epoch [2/3], Step [1600/3236], Loss: 2.0516, Perplexity: 7.78075
Epoch [2/3], Step [1700/3236], Loss: 2.7572, Perplexity: 15.7562
Epoch [2/3], Step [1800/3236], Loss: 2.1129, Perplexity: 8.27208
Epoch [2/3], Step [1900/3236], Loss: 2.2242, Perplexity: 9.24648
Epoch [2/3], Step [2000/3236], Loss: 2.0226, Perplexity: 7.55823
Epoch [2/3], Step [2100/3236], Loss: 2.1502, Perplexity: 8.58633
Epoch [2/3], Step [2200/3236], Loss: 2.0441, Perplexity: 7.72252
Epoch [2/3], Step [2300/3236], Loss: 2.0163, Perplexity: 7.51072
Epoch [2/3], Step [2400/3236], Loss: 2.1010, Perplexity: 8.17435
Epoch [2/3], Step [2500/3236], Loss: 2.0968, Perplexity: 8.14027
Epoch [2/3], Step [2600/3236], Loss: 1.9994, Perplexity: 7.38474
Epoch [2/3], Step [2700/3236], Loss: 1.9209, Perplexity: 6.82716
Epoch [2/3], Step [2800/3236], Loss: 2.0912, Perplexity: 8.09457
Epoch [2/3], Step [2900/3236], Loss: 2.2003, Perplexity: 9.02803
Epoch [2/3], Step [3000/3236], Loss: 2.7045, Perplexity: 14.9469
Epoch [2/3], Step [3100/3236], Loss: 2.1232, Perplexity: 8.35803
Epoch [2/3], Step [3200/3236], Loss: 1.9146, Perplexity: 6.78408
```

```
Epoch [3/3], Step [100/3236], Loss: 2.0653, Perplexity: 7.887812
Epoch [3/3], Step [200/3236], Loss: 1.9423, Perplexity: 6.97499
Epoch [3/3], Step [300/3236], Loss: 2.3257, Perplexity: 10.2334
Epoch [3/3], Step [400/3236], Loss: 1.9286, Perplexity: 6.88012
Epoch [3/3], Step [500/3236], Loss: 2.2820, Perplexity: 9.79642
Epoch [3/3], Step [600/3236], Loss: 1.9484, Perplexity: 7.01720
Epoch [3/3], Step [700/3236], Loss: 1.9987, Perplexity: 7.37931
Epoch [3/3], Step [800/3236], Loss: 2.1535, Perplexity: 8.61468
Epoch [3/3], Step [900/3236], Loss: 2.6893, Perplexity: 14.7214
Epoch [3/3], Step [1000/3236], Loss: 1.8966, Perplexity: 6.6634
Epoch [3/3], Step [1100/3236], Loss: 2.1365, Perplexity: 8.46968
Epoch [3/3], Step [1200/3236], Loss: 2.4165, Perplexity: 11.2070
Epoch [3/3], Step [1300/3236], Loss: 1.9006, Perplexity: 6.68981
Epoch [3/3], Step [1400/3236], Loss: 1.8728, Perplexity: 6.50629
Epoch [3/3], Step [1500/3236], Loss: 1.9913, Perplexity: 7.32487
Epoch [3/3], Step [1600/3236], Loss: 2.0310, Perplexity: 7.62154
Epoch [3/3], Step [1700/3236], Loss: 1.9380, Perplexity: 6.94503
Epoch [3/3], Step [1800/3236], Loss: 2.1100, Perplexity: 8.24834
Epoch [3/3], Step [1900/3236], Loss: 2.0248, Perplexity: 7.57427
Epoch [3/3], Step [2000/3236], Loss: 2.0444, Perplexity: 7.72442
Epoch [3/3], Step [2100/3236], Loss: 2.0081, Perplexity: 7.44938
Epoch [3/3], Step [2200/3236], Loss: 1.9107, Perplexity: 6.75785
Epoch [3/3], Step [2300/3236], Loss: 1.9749, Perplexity: 7.20629
Epoch [3/3], Step [2400/3236], Loss: 2.0552, Perplexity: 7.80858
Epoch [3/3], Step [2500/3236], Loss: 1.8976, Perplexity: 6.66998
Epoch [3/3], Step [2600/3236], Loss: 1.8700, Perplexity: 6.48860
Epoch [3/3], Step [2700/3236], Loss: 1.8238, Perplexity: 6.19566
Epoch [3/3], Step [2800/3236], Loss: 1.9544, Perplexity: 7.05962
Epoch [3/3], Step [2900/3236], Loss: 1.9418, Perplexity: 6.97103
Epoch [3/3], Step [3000/3236], Loss: 2.0168, Perplexity: 7.514462
Epoch [3/3], Step [3100/3236], Loss: 1.8555, Perplexity: 6.39521
Epoch [3/3], Step [3200/3236], Loss: 1.8514, Perplexity: 6.36897
Epoch [3/3], Step [3236/3236], Loss: 1.8936, Perplexity: 6.64345
```

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.
```