Computer Vision Nanodegree

Project: Image Captioning

In this notebook, you will learn how to load and pre-process data from the COCO dataset. You will also design a CNN-RNN model for automatically generating image captions.

Note that **any amendments that you make to this notebook will not be graded**. However, you will use the instructions provided in **Step 3** and **Step 4** to implement your own CNN encoder and RNN decoder by making amendments to the **models.py** file provided as part of this project. Your **models.py** file **will be graded**.

Feel free to use the links below to navigate the notebook:

Step 1: Explore the Data Loader

We have already written a data loader that you can use to load the COCO dataset in batches.

In the code cell below, you will initialize the data loader by using the `get_loader` function in **data_loader.py**.

> For this project, you are not permitted to change the **data_loader.py** file, which must be used as-is.

The `get_loader` function takes as input a number of arguments that can be explored in **data_loader.py**. Take the time to explore these arguments now by opening **data_loader.py** in a new window. Most of the arguments must be left at their default values, and you are only allowed to amend the values of the arguments below:

1. `transform` - an image transform specifying how to pre-process the images and convert them to PyTorch tensors before using them as input to the CNN encoder. For now, you are encouraged to keep the transform as provided in `transform_train`. You will have the opportunity later to choose your own image transform to pre-process the COCO images.
2. `mode` - one of `'train'` (loads the training data in batches) or `'test'` (for the test data). We will say that the data loader is in training or test mode, respectively. While following the instructions in this notebook, please keep the data loader in training mode by setting `mode='train'`.
3. `batch_size` - determines the batch size. When training the model, this is number of image-caption pairs used to amend the model weights in each training step.
4. `vocab_threshold` - the total number of times that a word must appear in the in the training captions before it is used as part of the vocabulary. Words that have fewer than `vocab_threshold` occurrences

in the training captions are considered unknown words.

5. `vocab_from_file` - a Boolean that decides whether to load the vocabulary from file.

We will describe the `vocab_threshold` and `vocab_from_file` arguments in more detail soon. For now, run the code cell below. Be patient - it may take a couple of minutes to run!

In [1]:
```
#!pip install pycocotools
```

In [4]:
```
import sys
sys.path.append('/opt/cocoapi/PythonAPI')
!pip install nltk
import nltk
nltk.download('punkt')
nltk.download('punkt_tab')
from data_loader import get_loader
from torchvision import transforms
from pycocotools.coco import COCO
```

```
Requirement already satisfied: nltk in ./icpy/lib/python3.10/site-packages (3.9.1)
Requirement already satisfied: click in ./icpy/lib/python3.10/site-packages (from nltk) (8.1.7)
Requirement already satisfied: joblib in ./icpy/lib/python3.10/site-packages (from nltk) (1.4.2)
Requirement already satisfied: regex>=2021.8.3 in ./icpy/lib/python3.10/site-packages (from nltk)
(2024.9.11)
Requirement already satisfied: tqdm in ./icpy/lib/python3.10/site-packages (from nltk) (4.66.5)
[nltk_data] Downloading package punkt to /home/johan/nltk_data...
[nltk_data]   Package punkt is already up-to-date!
[nltk_data] Downloading package punkt_tab to /home/johan/nltk_data...
[nltk_data]   Unzipping tokenizers/punkt_tab.zip.
```

In [5]:
```
# Define a transform to pre-process the training images.
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
    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))])

# Set the minimum word count threshold.
vocab_threshold = 5

# Specify the batch size.
batch_size = 10

# Obtain the data loader.
data_loader = get_loader(transform=transform_train,
                         mode='train',
                         batch_size=batch_size,
                         vocab_threshold=vocab_threshold,
                         vocab_from_file=False)
```

```
loading annotations into memory...
Done (t=0.40s)
creating index...
index created!
[0/414113] Tokenizing captions...
[100000/414113] Tokenizing captions...
[200000/414113] Tokenizing captions...
[300000/414113] Tokenizing captions...
[400000/414113] Tokenizing captions...
loading annotations into memory...
Done (t=0.25s)
creating index...
index created!
Obtaining caption lengths...
100%|██████████| 414113/414113 [00:16<00:00, 25337.12it/s]
```

In [7]:
```
# data_loader.dataset.caption_lengths
```

When you ran the code cell above, the data loader was stored in the variable `data_loader`.

You can access the corresponding dataset as `data_loader.dataset`. This dataset is an instance of the `CoCoDataset` class in **data_loader.py**. If you are unfamiliar with data loaders and datasets, you are encouraged to review this PyTorch tutorial.

Exploring the `__getitem__` Method

The `__getitem__` method in the `CoCoDataset` class determines how an image-caption pair is pre-processed before being incorporated into a batch. This is true for all `Dataset` classes in PyTorch; if this is unfamiliar to you, please review the tutorial linked above.

When the data loader is in training mode, this method begins by first obtaining the filename (`path`) of a training image and its corresponding caption (`caption`).

Image Pre-Processing

Image pre-processing is relatively straightforward (from the `__getitem__` method in the `CoCoDataset` class):

```
# Convert image to tensor and pre-process using transform
image = Image.open(os.path.join(self.img_folder, path)).convert('RGB')
image = self.transform(image)
```

After loading the image in the training folder with name `path`, the image is pre-processed using the same transform (`transform_train`) that was supplied when instantiating the data loader.

Caption Pre-Processing

The captions also need to be pre-processed and prepped for training. In this example, for generating captions, we are aiming to create a model that predicts the next token of a sentence from previous tokens, so we turn the caption associated with any image into a list of tokenized words, before casting it to a PyTorch tensor that we can use to train the network.

To understand in more detail how COCO captions are pre-processed, we'll first need to take a look at the `vocab` instance variable of the `CoCoDataset` class. The code snippet below is pulled from the `__init__` method of the `CoCoDataset` class:

```
def __init__(self, transform, mode, batch_size, vocab_threshold, vocab_file,
start_word,
        end_word, unk_word, annotations_file, vocab_from_file, img_folder):
        ...
        self.vocab = Vocabulary(vocab_threshold, vocab_file, start_word,
            end_word, unk_word, annotations_file, vocab_from_file)
        ...
```

From the code snippet above, you can see that `data_loader.dataset.vocab` is an instance of the `Vocabulary` class from **vocabulary.py**. Take the time now to verify this for yourself by looking at the full code in **data_loader.py**.

We use this instance to pre-process the COCO captions (from the `__getitem__` method in the `CoCoDataset` class):

```
# Convert caption to tensor of word ids.
tokens = nltk.tokenize.word_tokenize(str(caption).lower())    # line 1
caption = []                                                  # line 2
caption.append(self.vocab(self.vocab.start_word))            # line 3
caption.extend([self.vocab(token) for token in tokens])       # line 4
caption.append(self.vocab(self.vocab.end_word))               # line 5
caption = torch.Tensor(caption).long()                        # line 6
```

As you will see soon, this code converts any string-valued caption to a list of integers, before casting it to a PyTorch tensor. To see how this code works, we'll apply it to the sample caption in the next code cell.

In [8]:
```
sample_caption = 'A person doing a trick on a rail while riding a skateboard.'
```

In `line 1` of the code snippet, every letter in the caption is converted to lowercase, and the `nltk.tokenize.word_tokenize` function is used to obtain a list of string-valued tokens. Run the next code cell to visualize the effect on `sample_caption`.

In [9]:
```
import nltk

sample_tokens = nltk.tokenize.word_tokenize(str(sample_caption).lower())
print(sample_tokens)
```

```
['a', 'person', 'doing', 'a', 'trick', 'on', 'a', 'rail', 'while', 'riding', 'a', 'skateboard',
'.']
```

In `line 2` and `line 3` we initialize an empty list and append an integer to mark the start of a caption. The paper that you are encouraged to implement uses a special start word (and a special end word, which we'll examine below) to mark the beginning (and end) of a caption.

This special start word (`"<start>"`) is decided when instantiating the data loader and is passed as a parameter (`start_word`). You are **required** to keep this parameter at its default value (`start_word="<start>"`).

As you will see below, the integer `0` is always used to mark the start of a caption.

In [18]:
```
sample_caption = []

start_word = data_loader.dataset.vocab.start_word
print('Special start word:', start_word)
sample_caption.append(data_loader.dataset.vocab(start_word))
print(sample_caption)
```

```
Special start word: <start>
[0]
```

In `line 4`, we continue the list by adding integers that correspond to each of the tokens in the caption.

In [19]:
```
# a = 'I enjoy playing football and run around'
# b = nltk.tokenize.word_tokenize(str(a).lower())
# [data_loader.dataset.vocab(t) for t in b]
```

In [20]:
```
sample_caption.extend([data_loader.dataset.vocab(token) for token in sample_tokens])
print(sample_caption)
```

```
[0, 3, 98, 756, 3, 396, 39, 3, 1015, 207, 139, 3, 755, 18]
```

In `line 5`, we append a final integer to mark the end of the caption.

Identical to the case of the special start word (above), the special end word (`"<end>"`) is decided when instantiating the data loader and is passed as a parameter (`end_word`). You are **required** to keep this parameter at its default value (`end_word="<end>"`).

As you will see below, the integer `1` is always used to mark the end of a caption.

In [21]:
```
end_word = data_loader.dataset.vocab.end_word
print('Special end word:', end_word)

sample_caption.append(data_loader.dataset.vocab(end_word))
print(sample_caption)
```

```
Special end word: <end>
[0, 3, 98, 756, 3, 396, 39, 3, 1015, 207, 139, 3, 755, 18, 1]
```

Finally, in `line 6`, we convert the list of integers to a PyTorch tensor and cast it to long type. You can read more about the different types of PyTorch tensors on the website.

In [22]:
```
import torch

sample_caption = torch.Tensor(sample_caption).long()
print(sample_caption)
```

```
tensor([   0,    3,   98,  756,    3,  396,   39,    3, 1015,  207,  139,    3,
         755,   18,    1])
```

And that's it! In summary, any caption is converted to a list of tokens, with *special* start and end tokens marking the beginning and end of the sentence:

```
[<start>, 'a', 'person', 'doing', 'a', 'trick', 'while', 'riding', 'a',
'skateboard', '.', <end>]
```

This list of tokens is then turned into a list of integers, where every distinct word in the vocabulary has an associated integer value:

```
[0, 3, 98, 754, 3, 396, 207, 139, 3, 753, 18, 1]
```

Finally, this list is converted to a PyTorch tensor. All of the captions in the COCO dataset are pre-processed using this same procedure from `lines 1-6` described above.

As you saw, in order to convert a token to its corresponding integer, we call `data_loader.dataset.vocab` as a function. The details of how this call works can be explored in the `__call__` method in the `Vocabulary` class in **vocabulary.py**.

```
def __call__(self, word):
    if not word in self.word2idx:
        return self.word2idx[self.unk_word]
    return self.word2idx[word]
```

The `word2idx` instance variable is a Python dictionary that is indexed by string-valued keys (mostly tokens obtained from training captions). For each key, the corresponding value is the integer that the token is mapped to in the pre-processing step.

Use the code cell below to view a subset of this dictionary.

In [23]:
```
# Preview the word2idx dictionary.
dict(list(data_loader.dataset.vocab.word2idx.items())[:10])
```

Out[23]:
```
{'<start>': 0,
 '<end>': 1,
 '<unk>': 2,
 'a': 3,
 'very': 4,
 'clean': 5,
 'and': 6,
 'well': 7,
 'decorated': 8,
 'empty': 9}
```

We also print the total number of keys.

In [24]:
```
# Print the total number of keys in the word2idx dictionary.
print('Total number of tokens in vocabulary:', len(data_loader.dataset.vocab))
```

```
Total number of tokens in vocabulary: 9947
```

As you will see if you examine the code in **vocabulary.py**, the `word2idx` dictionary is created by looping over the captions in the training dataset. If a token appears no less than `vocab_threshold` times in the training set, then it is added as a key to the dictionary and assigned a corresponding unique integer. You will have the option later to amend the `vocab_threshold` argument when instantiating your data loader. Note that in general, **smaller** values for `vocab_threshold` yield a **larger** number of tokens in the vocabulary. You are encouraged to check this for yourself in the next code cell by decreasing the value of `vocab_threshold` before creating a new data loader.

In [25]:
```
# Modify the minimum word count threshold.
vocab_threshold = 4

# Obtain the data loader.
data_loader = get_loader(transform=transform_train,
                         mode='train',
                         batch_size=batch_size,
                         vocab_threshold=vocab_threshold,
                         vocab_from_file=False)
```

```
loading annotations into memory...
Done (t=0.23s)
creating index...
index created!
[0/414113] Tokenizing captions...
[100000/414113] Tokenizing captions...
[200000/414113] Tokenizing captions...
[300000/414113] Tokenizing captions...
[400000/414113] Tokenizing captions...
loading annotations into memory...
Done (t=0.24s)
creating index...
index created!
Obtaining caption lengths...
100%|██████████| 414113/414113 [00:16<00:00, 25605.34it/s]
```

In [26]:
```
# Print the total number of keys in the word2idx dictionary.
print('Total number of tokens in vocabulary:', len(data_loader.dataset.vocab))
```

```
Total number of tokens in vocabulary: 9947
```

There are also a few special keys in the `word2idx` dictionary. You are already familiar with the special start word (`"<start>"`) and special end word (`"<end>"`). There is one more special token, corresponding to unknown words (`"<unk>"`). All tokens that don't appear anywhere in the `word2idx` dictionary are considered unknown words. In the pre-processing step, any unknown tokens are mapped to the integer `2` .

In [27]:
```
unk_word = data_loader.dataset.vocab.unk_word
print('Special unknown word:', unk_word)

print('All unknown words are mapped to this integer:', data_loader.dataset.vocab(unk_word))
```

```
Special unknown word: <unk>
All unknown words are mapped to this integer: 2
```

Check this for yourself below, by pre-processing the provided nonsense words that never appear in the training captions.

In [28]:
```
print(data_loader.dataset.vocab('jfkafejw'))
print(data_loader.dataset.vocab('ieowoqjf'))
```

```
2
2
```

The final thing to mention is the `vocab_from_file` argument that is supplied when creating a data loader. To understand this argument, note that when you create a new data loader, the vocabulary (`data_loader.dataset.vocab`) is saved as a pickle file in the project folder, with filename `vocab.pkl` .

If you are still tweaking the value of the `vocab_threshold` argument, you **must** set `vocab_from_file=False` to have your changes take effect.

But once you are happy with the value that you have chosen for the `vocab_threshold` argument, you need only run the data loader *one more time* with your chosen `vocab_threshold` to save the new vocabulary to file. Then, you can henceforth set `vocab_from_file=True` to load the vocabulary from file and speed the instantiation of the data loader. Note that building the vocabulary from scratch is the most time-consuming part of instantiating the data loader, and so you are strongly encouraged to set `vocab_from_file=True` as soon as you are able.

Note that if `vocab_from_file=True`, then any supplied argument for `vocab_threshold` when instantiating the data loader is completely ignored.

In [29]:
```
# Obtain the data loader (from file). Note that it runs much faster than before!
data_loader = get_loader(transform=transform_train,
                         mode='train',
                         batch_size=batch_size,
                         vocab_from_file=True)
```

```
Vocabulary successfully loaded from vocab.pkl file!
loading annotations into memory...
Done (t=0.21s)
creating index...
index created!
Obtaining caption lengths...
100%|██████████| 414113/414113 [00:17<00:00, 23469.29it/s]
```

In the next section, you will learn how to use the data loader to obtain batches of training data.

Step 2: Use the Data Loader to Obtain Batches

The captions in the dataset vary greatly in length. You can see this by examining `data_loader.dataset.caption_lengths`, a Python list with one entry for each training caption (where the value stores the length of the corresponding caption).

In the code cell below, we use this list to print the total number of captions in the training data with each length. As you will see below, the majority of captions have length 10. Likewise, very short and very long captions are quite rare.

In [30]:
```
from collections import Counter

# Tally the total number of training captions with each length.
counter = Counter(data_loader.dataset.caption_lengths)
lengths = sorted(counter.items(), key=lambda pair: pair[1], reverse=True)
for value, count in lengths:
    print('value: %2d --- count: %5d' % (value, count))
```

```
value: 10 --- count: 86302
value: 11 --- count: 79971
value:  9 --- count: 71920
value: 12 --- count: 57653
value: 13 --- count: 37668
value: 14 --- count: 22342
value:  8 --- count: 20742
value: 15 --- count: 12839
value: 16 --- count:  7736
value: 17 --- count:  4845
value: 18 --- count:  3101
value: 19 --- count:  2017
value:  7 --- count:  1594
value: 20 --- count:  1453
value: 21 --- count:   997
value: 22 --- count:   683
value: 23 --- count:   534
value: 24 --- count:   384
value: 25 --- count:   277
value: 26 --- count:   214
value: 27 --- count:   160
value: 28 --- count:   114
value: 29 --- count:    87
value: 30 --- count:    58
value: 31 --- count:    49
value: 32 --- count:    44
value: 34 --- count:    40
value: 37 --- count:    32
value: 35 --- count:    31
value: 33 --- count:    30
value: 36 --- count:    26
value: 38 --- count:    18
value: 39 --- count:    18
value: 43 --- count:    16
value: 44 --- count:    16
value: 48 --- count:    12
value: 45 --- count:    11
value: 42 --- count:    10
value: 40 --- count:     9
value: 49 --- count:     9
value: 46 --- count:     9
value: 47 --- count:     7
value: 50 --- count:     6
value: 51 --- count:     6
value: 41 --- count:     6
value: 52 --- count:     5
value: 54 --- count:     3
value: 56 --- count:     2
value:  6 --- count:     2
value: 53 --- count:     2
value: 55 --- count:     2
value: 57 --- count:     1
```

To generate batches of training data, we begin by first sampling a caption length (where the probability that any length is drawn is proportional to the number of captions with that length in the dataset). Then, we retrieve a batch of size `batch_size` of image-caption pairs, where all captions have the sampled length. This approach for assembling batches matches the procedure in this paper and has been shown to be computationally efficient without degrading performance.

Run the code cell below to generate a batch. The `get_train_indices` method in the `CoCoDataset` class first samples a caption length, and then samples `batch_size` indices corresponding to training data points with captions of that length. These indices are stored below in `indices`.

These indices are supplied to the data loader, which then is used to retrieve the corresponding data points. The pre-processed images and captions in the batch are stored in `images` and `captions`.

In [43]:
```
import numpy as np
import torch.utils.data as data

# Randomly sample a caption length, and sample indices with that length.
indices = data_loader.dataset.get_train_indices()
print('sampled indices:', 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))

print('images.shape:', images.shape)
print('captions.shape:', captions.shape)

# (Optional) Uncomment the lines of code below to print the pre-processed images and captions.
print('images:', images)
print('captions:', captions)
```

```
sampled indices: [np.int64(161568), np.int64(236516), np.int64(328201), np.int64(12314), np.int64
(359672), np.int64(228996), np.int64(96767), np.int64(8054), np.int64(257607), np.int64(246097)]
images.shape: torch.Size([10, 3, 224, 224])
captions.shape: torch.Size([10, 16])
images: tensor([[[[ 0.8618,  0.9988,  1.1187,  ...,  1.2557,  1.2557,  1.2385],
          [ 0.8104,  0.9132,  1.0159,  ...,  1.2728,  1.2728,  1.2385],
          [ 0.8104,  0.8961,  0.9646,  ...,  1.2728,  1.2557,  1.2214],
          ...,
          [ 2.2318,  2.2147,  2.1975,  ...,  1.3413,  1.3242,  1.2557],
          [ 2.1975,  2.1633,  2.1633,  ...,  1.3070,  1.2899,  1.2214],
          [ 2.0092,  1.9749,  1.9920,  ...,  1.2557,  1.2214,  1.1872]],

         [[ 0.9580,  1.1155,  1.2731,  ...,  1.3782,  1.3782,  1.3782],
          [ 0.8704,  1.0455,  1.1681,  ...,  1.3957,  1.3957,  1.3957],
          [ 0.8354,  0.9755,  1.0805,  ...,  1.4132,  1.4132,  1.4132],
          ...,
          [ 2.4111,  2.3761,  2.3585,  ...,  1.3606,  1.3256,  1.2381],
          [ 2.3410,  2.3060,  2.3060,  ...,  1.2906,  1.2556,  1.1856],
          [ 2.1310,  2.1134,  2.1134,  ...,  1.2206,  1.1856,  1.0980]],

         [[ 1.0888,  1.2805,  1.4897,  ...,  1.3851,  1.3851,  1.3851],
          [ 0.9842,  1.1585,  1.3677,  ...,  1.4025,  1.4025,  1.4025],
          [ 0.8971,  1.0714,  1.2457,  ...,  1.4200,  1.4200,  1.4025],
          ...,
          [ 2.6400,  2.6226,  2.6226,  ...,  1.5594,  1.5071,  1.4374],
          [ 2.6226,  2.6051,  2.6051,  ...,  1.4897,  1.4374,  1.3677],
          [ 2.4657,  2.4308,  2.4308,  ...,  1.4025,  1.3677,  1.2805]]],


        [[[ 0.2453,  0.2453,  0.2453,  ...,  0.2111,  0.2111,  0.2111],
          [ 0.2453,  0.2453,  0.2453,  ...,  0.1939,  0.2111,  0.2282],
          [ 0.2796,  0.2624,  0.2796,  ...,  0.2111,  0.2111,  0.2796],
          ...,
          [ 0.0398,  0.0227,  0.0227,  ..., -1.3302, -1.3302, -1.3302],
          [ 0.0227,  0.0398,  0.0569,  ..., -1.2445, -1.2617, -1.3130],
          [ 0.0569,  0.0398,  0.0569,  ..., -1.2445, -1.2445, -1.2103]],

         [[ 0.5378,  0.5203,  0.5203,  ...,  0.4853,  0.4678,  0.4503],
          [ 0.5203,  0.5203,  0.5378,  ...,  0.4328,  0.4678,  0.4853],
          [ 0.5378,  0.5553,  0.5903,  ...,  0.4503,  0.4678,  0.5378],
          ...,
          [-0.6176, -0.6352, -0.6352,  ..., -1.7381, -1.7206, -1.7031],
          [-0.6176, -0.6001, -0.6001,  ..., -1.7031, -1.6856, -1.7206],
          [-0.5826, -0.6001, -0.6001,  ..., -1.7556, -1.7206, -1.6681]],

         [[ 0.9668,  0.9494,  0.9319,  ...,  0.8797,  0.8622,  0.8448],
          [ 0.9842,  0.9668,  0.9494,  ...,  0.8622,  0.8622,  0.8797],
          [ 1.0365,  1.0017,  0.9842,  ...,  0.8797,  0.8797,  0.9145],
          ...,
          [-1.1247, -1.1421, -1.1770,  ..., -1.7870, -1.7870, -1.7173],
          [-1.1596, -1.1421, -1.1421,  ..., -1.7696, -1.7696, -1.7522],
          [-1.1596, -1.1944, -1.1596,  ..., -1.7870, -1.7522, -1.7522]]],


        [[[-2.1008, -1.9980, -1.3302,  ...,  0.4508,  0.4337,  0.4508],
          [-2.1179, -2.0152, -1.4500,  ...,  0.3994,  0.4508,  0.3138],
          [-2.1179, -2.0323, -1.6042,  ...,  0.5022,  0.4679,  0.0912],
          ...,
          [-2.1179, -2.1179, -2.1179,  ..., -2.1179, -2.1179, -2.1179],
          [-2.1179, -2.1179, -2.1179,  ..., -2.1179, -2.1179, -2.1179],
          [-2.1179, -2.1179, -2.1179,  ..., -2.1179, -2.1179, -2.1179]],
```

```
 [[-2.0182, -1.9832, -1.4230,  ...,  0.6779,  0.6254,  0.6078],
  [-2.0357, -1.9832, -1.5455,  ...,  0.5903,  0.6254,  0.4503],
  [-2.0182, -2.0007, -1.7031,  ...,  0.6604,  0.6254,  0.2402],
  ...,
  [-2.0357, -2.0357, -2.0357,  ..., -2.0357, -2.0357, -2.0357],
  [-2.0357, -2.0357, -2.0357,  ..., -2.0357, -2.0357, -2.0357],
  [-2.0357, -2.0357, -2.0357,  ..., -2.0357, -2.0357, -2.0357]],

 [[-1.7696, -1.6824, -1.0724,  ...,  0.5834,  0.5485,  0.6008],
  [-1.7696, -1.6999, -1.1770,  ...,  0.5136,  0.5659,  0.4614],
  [-1.7696, -1.7173, -1.3339,  ...,  0.5659,  0.5659,  0.2173],
  ...,
  [-1.8044, -1.8044, -1.8044,  ..., -1.8044, -1.8044, -1.8044],
  [-1.8044, -1.8044, -1.8044,  ..., -1.8044, -1.8044, -1.8044],
  [-1.8044, -1.8044, -1.8044,  ..., -1.8044, -1.8044, -1.8044]]],


...,


[[[ 1.7009,  1.7009,  1.4783,  ...,  0.6221,  0.5707,  0.5536],
  [ 1.7009,  1.7009,  1.4612,  ...,  0.8104,  0.7933,  0.7419],
  [ 1.6838,  1.6838,  1.4098,  ...,  0.8276,  0.7762,  0.6221],
  ...,
  [ 1.6667,  1.6667,  1.2214,  ..., -0.1999, -0.4054, -0.4911],
  [ 1.6324,  1.6324,  1.2043,  ..., -0.0629, -0.2171, -0.3027],
  [ 1.6153,  1.6153,  1.2043,  ...,  0.0056,  0.0398,  0.0056]],

 [[ 1.7633,  1.7808,  1.5707,  ...,  0.6954,  0.6429,  0.6254],
  [ 1.7808,  1.7983,  1.5532,  ...,  0.8880,  0.8704,  0.8179],
  [ 1.7983,  1.7983,  1.5182,  ...,  0.9055,  0.8529,  0.6954],
  ...,
  [ 1.7808,  1.7633,  1.3431,  ..., -0.0224, -0.2325, -0.3200],
  [ 1.7458,  1.7283,  1.3256,  ...,  0.1176, -0.0399, -0.1275],
  [ 1.7283,  1.7283,  1.3081,  ...,  0.1877,  0.2227,  0.1877]],

 [[ 1.2805,  1.3328,  1.1237,  ...,  0.4265,  0.3742,  0.3742],
  [ 1.2805,  1.3328,  1.1062,  ...,  0.6356,  0.6182,  0.5659],
  [ 1.2805,  1.3328,  1.0888,  ...,  0.6356,  0.6008,  0.4614],
  ...,
  [ 1.2631,  1.2805,  0.8971,  ..., -0.1312, -0.3404, -0.4275],
  [ 1.2282,  1.2457,  0.8797,  ...,  0.0082, -0.1487, -0.2358],
  [ 1.1934,  1.2282,  0.8797,  ...,  0.0605,  0.0953,  0.0605]]],


[[[-1.0733, -1.0904, -1.1075,  ..., -1.6384, -1.6042, -1.5699],
  [-1.1418, -1.1589, -1.1760,  ..., -1.5870, -1.5870, -1.6042],
  [-1.1760, -1.1932, -1.1932,  ..., -1.6555, -1.6384, -1.6384],
  ...,
  [ 1.8722,  1.8722,  1.8722,  ..., -1.3815, -1.4158, -1.4500],
  [ 1.8550,  1.8379,  1.8722,  ..., -1.4158, -1.4329, -1.4672],
  [ 1.8208,  1.8037,  1.8379,  ..., -1.5014, -1.4672, -1.4500]],

 [[-1.2304, -1.2479, -1.2654,  ..., -1.7031, -1.7031, -1.7031],
  [-1.3004, -1.3179, -1.3004,  ..., -1.7206, -1.7206, -1.7031],
  [-1.3354, -1.3704, -1.3529,  ..., -1.7731, -1.7381, -1.7031],
  ...,
  [ 1.7633,  1.7808,  1.8158,  ..., -1.5980, -1.5980, -1.5805],
  [ 1.7458,  1.7283,  1.7633,  ..., -1.5980, -1.5805, -1.6155],
  [ 1.7283,  1.7108,  1.7283,  ..., -1.6331, -1.5980, -1.6155]],

 [[-1.4384, -1.4210, -1.4384,  ..., -1.6476, -1.6302, -1.6650],
```

```
          [-1.4210, -1.4210, -1.4384,  ..., -1.6476, -1.6476, -1.6650],
          [-1.4210, -1.4210, -1.4384,  ..., -1.6999, -1.6824, -1.6650],
          ...,
          [ 1.5420,  1.5420,  1.5420,  ..., -1.5256, -1.5430, -1.5430],
          [ 1.4897,  1.4897,  1.5245,  ..., -1.5256, -1.5604, -1.5604],
          [ 1.4200,  1.4374,  1.4897,  ..., -1.5604, -1.5779, -1.5604]]],


        [[[ 1.1872,  1.1872,  1.1872,  ...,  1.2899,  1.2899,  1.2899],
          [ 1.1872,  1.2043,  1.2043,  ...,  1.2899,  1.2899,  1.2899],
          [ 1.2043,  1.2043,  1.2043,  ...,  1.2899,  1.2899,  1.2899],
          ...,
          [ 1.4612,  1.4440,  1.4612,  ...,  1.3755,  1.3755,  1.4440],
          [ 1.2899,  1.3755,  1.3927,  ...,  1.4098,  1.3242,  1.3413],
          [ 1.2214,  1.2214,  1.2385,  ...,  1.4269,  1.3927,  1.3413]],

         [[ 1.5007,  1.5182,  1.5007,  ...,  1.6232,  1.6232,  1.6232],
          [ 1.5007,  1.5182,  1.5182,  ...,  1.6057,  1.6057,  1.6232],
          [ 1.5007,  1.5182,  1.5182,  ...,  1.6232,  1.6232,  1.6232],
          ...,
          [ 1.4832,  1.4832,  1.4657,  ...,  1.4657,  1.4482,  1.4657],
          [ 1.1506,  1.2381,  1.2556,  ...,  1.4657,  1.3957,  1.4307],
          [ 0.8354,  0.8529,  0.9405,  ...,  1.4482,  1.4132,  1.3782]],

         [[ 1.9777,  1.9951,  1.9777,  ...,  2.0474,  2.0474,  2.0300],
          [ 1.9951,  1.9951,  1.9951,  ...,  2.0823,  2.0823,  2.0474],
          [ 1.9777,  1.9951,  1.9951,  ...,  2.0474,  2.0474,  2.0474],
          ...,
          [ 1.5594,  1.5245,  1.5420,  ...,  1.5594,  1.5420,  1.5768],
          [ 1.3328,  1.3851,  1.3851,  ...,  1.5768,  1.5071,  1.5594],
          [ 1.0888,  1.0714,  1.1585,  ...,  1.4897,  1.4722,  1.4548]]]])
captions: tensor([[   0,    3,  926,   13,  577,  130,   39,    3,   20,  822,   21,    3,
          194, 3167,   18,    1],
        [   0,    3,   92,  224,   39,    3,  715, 1062,    3, 1845,   77,    3,
          862,   28,   18,    1],
        [   0,  366, 1373,   77,    3,  681,   86,  110,  105,    3, 3747, 2234,
           39,   46,   18,    1],
        [   0,    3,   91,  170,   77,  121,   13,    3,  334,   21, 3824,   77,
          263,  792,   18,    1],
        [   0,    3,  174,  224,  111,    3,   29,  114,   39,    3,   40,   21,
          409,  279,   18,    1],
        [   0,  285,  417,    3, 5655,   39,  119,  111,    3,   29,   30,    6,
         8756,   40,   30,    1],
        [   0,    3,    8,  822,   21,    3,  608,   13, 9681, 3475,    6,    3,
         3167,   39,   46,    1],
        [   0,   47,  239,  402,   26, 1195,    3,    2,  901,  101,    3,  134,
            6, 1767,   18,    1],
        [   0,    3,  169,   39,    3,   84,  134,  130, 9225,  101,  524,  169,
            6,   91,   18,    1],
        [   0,  169,  756,    3,  377,  272,   39,  257,   13,    3,  755,  101,
           32,   55,   18,    1]])
```

Each time you run the code cell above, a different caption length is sampled, and a different batch of training data is returned. Run the code cell multiple times to check this out!

You will train your model in the next notebook in this sequence (**2_Training.ipynb**). This code for generating training batches will be provided to you.

> Before moving to the next notebook in the sequence (**2_Training.ipynb**), you are strongly encouraged to take the time to become very familiar with the code in **data_loader.py** and

> **vocabulary.py**. **Step 1** and **Step 2** of this notebook are designed to help facilitate a basic introduction and guide your understanding. However, our description is not exhaustive, and it is up to you (as part of the project) to learn how to best utilize these files to complete the project. **You should NOT amend any of the code in either *data_loader.py* or *vocabulary.py*.**

In the next steps, we focus on learning how to specify a CNN-RNN architecture in PyTorch, towards the goal of image captioning.

Step 3: Experiment with the CNN Encoder

Run the code cell below to import `EncoderCNN` and `DecoderRNN` from **model.py**.

In [45]:
```
# Watch for any changes in model.py, and re-load it automatically.
# % load_ext autoreload
# % autoreload 2

# Import EncoderCNN and DecoderRNN.
from model import EncoderCNN, DecoderRNN
```

In the next code cell we define a `device` that you will use move PyTorch tensors to GPU (if CUDA is available). Run this code cell before continuing.

In [46]:
```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
```

In [47]:
```
device
```

Out[47]:
```
device(type='cuda')
```

Run the code cell below to instantiate the CNN encoder in `encoder`.

The pre-processed images from the batch in **Step 2** of this notebook are then passed through the encoder, and the output is stored in `features`.

In [48]:
```
images[0].shape
```

Out[48]:
```
torch.Size([3, 224, 224])
```

In [49]:
```
# Specify the dimensionality of the image embedding.
embed_size = 256

#-#-#-# Do NOT modify the code below this line. #-#-#-#

# Initialize the encoder. (Optional: Add additional arguments if necessary.)
encoder = EncoderCNN(embed_size)

# Move the encoder to GPU if CUDA is available.
encoder.to(device)

# Move last batch of images (from Step 2) to GPU if CUDA is available.
images = images.to(device)

# Pass the images through the encoder.
features = encoder(images)
```

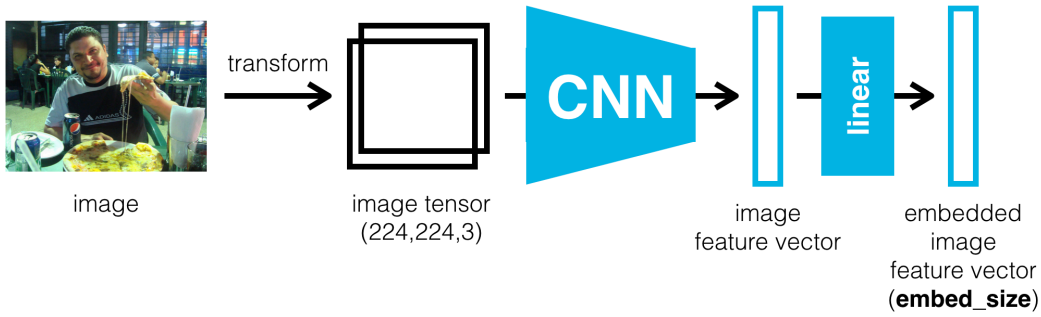
```
print('type(features):', type(features))
print('features.shape:', features.shape)

# Check that your encoder satisfies some requirements of the project! :D
assert type(features)==torch.Tensor, "Encoder output needs to be a PyTorch Tensor."
assert (features.shape[0]==batch_size) & (features.shape[1]==embed_size), "The shape of the enco
```

```
/home/johan/content/opt/image-captioning-pytorch/icpy/lib/python3.10/site-packages/torchvision/mo
dels/_utils.py:208: UserWarning: The parameter 'pretrained' is deprecated since 0.13 and may be r
emoved in the future, please use 'weights' instead.
  warnings.warn(
/home/johan/content/opt/image-captioning-pytorch/icpy/lib/python3.10/site-packages/torchvision/mo
dels/_utils.py:223: UserWarning: Arguments other than a weight enum or `None` for 'weights' are d
eprecated since 0.13 and may be removed in the future. The current behavior is equivalent to pass
ing `weights=ResNet50_Weights.IMAGENET1K_V1`. You can also use `weights=ResNet50_Weights.DEFAULT`
to get the most up-to-date weights.
  warnings.warn(msg)
Downloading: "https://download.pytorch.org/models/resnet50-0676ba61.pth" to /home/johan/.cache/to
rch/hub/checkpoints/resnet50-0676ba61.pth
100%|██████████| 97.8M/97.8M [00:06<00:00, 15.0MB/s]
type(features): <class 'torch.Tensor'>
features.shape: torch.Size([10, 256])
```

In [50]:
```
features[0].shape
```

Out[50]:
```
torch.Size([256])
```

The encoder that we provide to you uses the pre-trained ResNet-50 architecture (with the final fully-connected layer removed) to extract features from a batch of pre-processed images. The output is then flattened to a vector, before being passed through a `Linear` layer to transform the feature vector to have the same size as the word embedding.



You are welcome (and encouraged) to amend the encoder in **model.py**, to experiment with other architectures. In particular, consider using a different pre-trained model architecture. You may also like to add batch normalization.

> You are **not** required to change anything about the encoder.

For this project, you **must** incorporate a pre-trained CNN into your encoder. Your `EncoderCNN` class must take `embed_size` as an input argument, which will also correspond to the dimensionality of the input to the RNN decoder that you will implement in Step 4. When you train your model in the next notebook in this sequence (**2_Training.ipynb**), you are welcome to tweak the value of `embed_size`.

If you decide to modify the `EncoderCNN` class, save **model.py** and re-execute the code cell above. If the code cell returns an assertion error, then please follow the instructions to modify your code before

proceeding. The assert statements ensure that `features` is a PyTorch tensor with shape `[batch_size, embed_size]`.

Step 4: Implement the RNN Decoder

Before executing the next code cell, you must write `__init__` and `forward` methods in the `DecoderRNN` class in **model.py**. (Do **not** write the `sample` method yet - you will work with this method when you reach **3_Inference.ipynb**.)

> The `__init__` and `forward` methods in the `DecoderRNN` class are the only things that you **need** to modify as part of this notebook. You will write more implementations in the notebooks that appear later in the sequence.

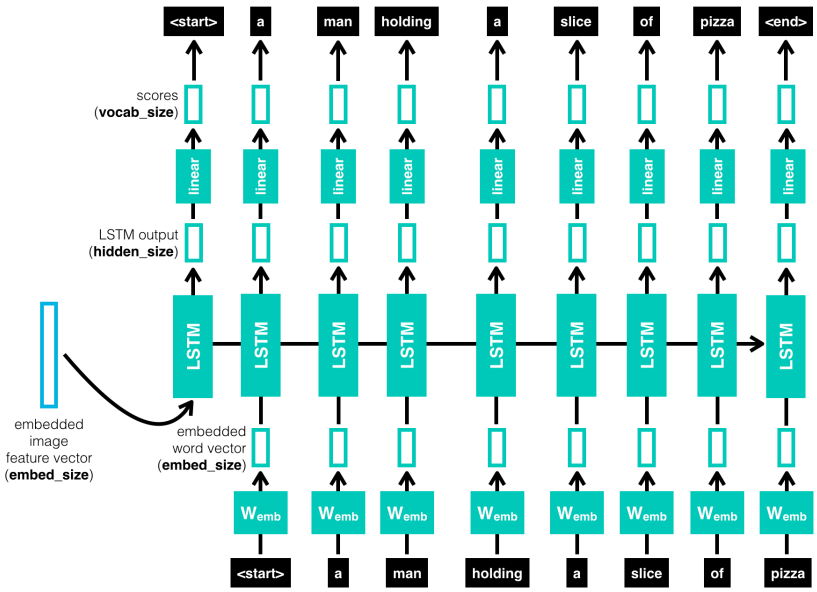
Your decoder will be an instance of the `DecoderRNN` class and must accept as input:

- the PyTorch tensor `features` containing the embedded image features (outputted in Step 3, when the last batch of images from Step 2 was passed through `encoder`), along with
- a PyTorch tensor corresponding to the last batch of captions (`captions`) from Step 2.

Note that the way we have written the data loader should simplify your code a bit. In particular, every training batch will contain pre-processed captions where all have the same length (`captions.shape[1]`), so **you do not need to worry about padding**.

> While you are encouraged to implement the decoder described in this paper, you are welcome to implement any architecture of your choosing, as long as it uses at least one RNN layer, with hidden dimension `hidden_size`.

Although you will test the decoder using the last batch that is currently stored in the notebook, your decoder should be written to accept an arbitrary batch (of embedded image features and pre-processed captions [where all captions have the same length]) as input.



In the code cell below, `outputs` should be a PyTorch tensor with size `[batch_size, captions.shape[1], vocab_size]`. Your output should be designed such that `outputs[i,j,k]` contains the model's predicted score, indicating how likely the `j`-th token in the `i`-th caption in the batch is the `k`-th token in the vocabulary. In the next notebook of the sequence (**2_Training.ipynb**), we provide code to supply these scores to the `torch.nn.CrossEntropyLoss` optimizer in PyTorch.

In [51]:
```
from model import EncoderCNN, DecoderRNN
```

In [52]:
```
# Specify the number of features in the hidden state of the RNN decoder.
hidden_size = 512

#-#-#-# Do NOT modify the code below this line. #-#-#-#

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

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

# Move the decoder to GPU if CUDA is available.
decoder.to(device)

# Move last batch of captions (from Step 1) to GPU if CUDA is available
captions = captions.to(device)

# Pass the encoder output and captions through the decoder.
outputs = decoder(features, captions)

print('type(outputs):', type(outputs))
print('outputs.shape:', outputs.shape)

# Check that your decoder satisfies some requirements of the project! :D
assert type(outputs)==torch.Tensor, "Decoder output needs to be a PyTorch Tensor."
assert (outputs.shape[0]==batch_size) & (outputs.shape[1]==captions.shape[1]) & (outputs.shape[2
```

```
type(outputs): <class 'torch.Tensor'>
outputs.shape: torch.Size([10, 16, 9947])
```

When you train your model in the next notebook in this sequence (**2_Training.ipynb**), you are welcome to tweak the value of `hidden_size`.