NYU

Image Captioning Generator

Analysis of Methods for Optimizing a Python-based Image Captioning Model

DS-GA 3001 Advanced Python

Charles Brillo-Sonnino   Aidan Claffey   Darren Geng
Meenakshi Jhalani   Kevin Wilson

May 5, 2020

Image Captioning Generator

- Dataset of 31,783 images and 158,915 sentence-based captions (5 per image)

- Task is to predict captions for previously unseen images, with many potential applications (e.g. accessibility, real-time video description)

- Many opportunities for optimization in data processing, model training, prediction and evaluation

Example image and associated captions

“An elderly man with light brown hair, wearing a gray sweater, blue shirt, brown pants and tan shoes, reads a book sitting at a bench, as two ducks feed themselves on the grass.”

“A man with parted hair and wearing glasses is seated outdoors on a bench where he is reading.”

“A man sits outside at a wooden table and reads a book while ducks eat in the foreground.”

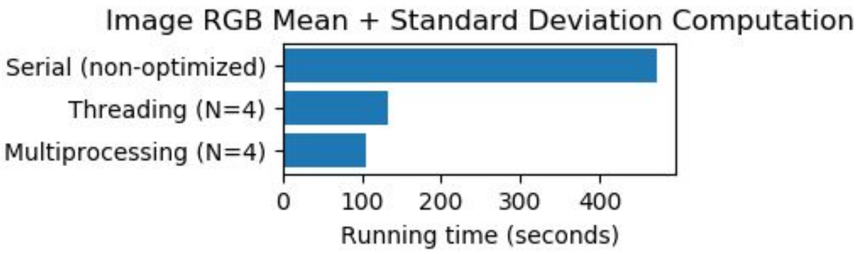
“An elderly man sitting on a bench while reading a book.”

“Man reads in a park while feeding the ducks.”

http://shannon.cs.illinois.edu/DenotationGraph/data/flickr30k.html

Using Python Concurrency to Optimize Data Preprocessing

- Computing mean and standard deviation of array representation of RGB color channels across all images in the dataset (to normalize the data)
- Before optimizing, this operation took 472.9 seconds to run
- Chunking the data and using multithreading with N=4 processes shortened the time to 104.2 seconds



RGB Means: Multiprocessing with N=4 processes

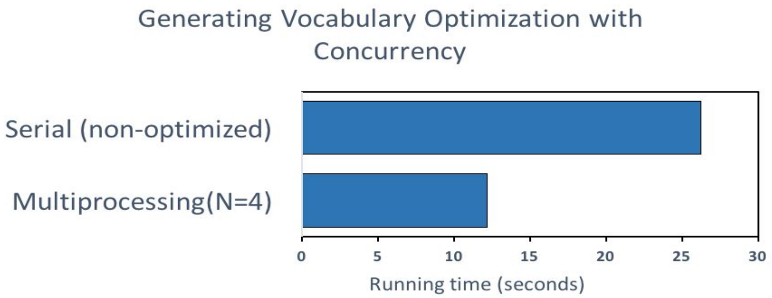
```
filepath = 'flickr30k-images'
img_files = [filename for filename in os.listdir(filepath)]
chunks = [(img_files[i:i+500]) for i in range(0, len(img_files), 500)]

def getRGB(chunk):
    r_channel_sum = 0
    g_channel_sum = 0
    b_channel_sum = 0
    count = 0
    for filename in chunk:
        if filename[-3:] == 'jpg':
            img = np.array(Image.open(os.path.join(filepath, filename)).convert('RGB'))
            r_channel_sum += np.sum(img[:,:,0])
            g_channel_sum += np.sum(img[:,:,1])
            b_channel_sum += np.sum(img[:,:,2])
            count += img.shape[0] * img.shape[1]
    return (r_channel_sum, g_channel_sum, b_channel_sum, count)

from multiprocessing.pool import Pool

start_time = time.time()
with Pool(4) as p:
    res = p.map(getRGB, chunks)

results = np.array(res).sum(axis=0)
r,g,b,c = results[0], results[1], results[2], results[3]

end_time = time.time()

print('R channel mean: {}'.format(r/c))
print('G channel mean: {}'.format(g/c))
print('B channel mean: {}'.format(b/c))
print("Time for Multiprocessing with N=4 processes: %ssecs" % (end_time - start_time))
```

```
R channel mean: 113.2971859326401
G channel mean: 107.42922106881713
B channel mean: 98.14465223794616
Time for Multiprocessing with N=4 processes: 47.823638916015625secs
```

Using Python Concurrency to Optimize Generation of Vocabulary

- Function build_vocab contains for loops which can be optimized for better performance

- Implementation of Multiprocessing with N= 4 processes decreased the time from 26.2 seconds to 12.16 seconds



Multiprocessing with N=4 processes

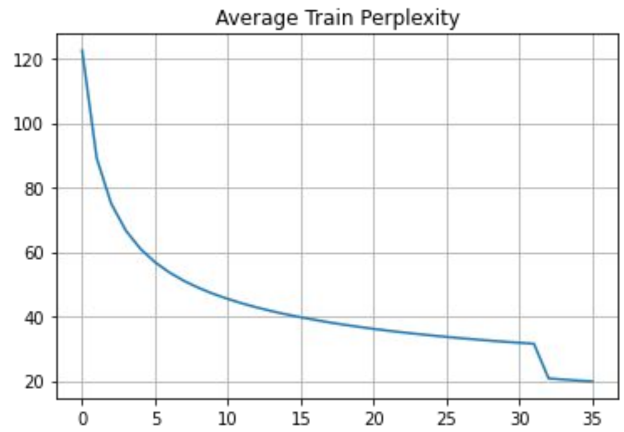
```
def build_vocab(ann_file = '../flickr30k/results_20130124.token', threshold = 4):
    """Build a simple vocabulary wrapper."""
    punc_set = set([',',';',':','.','?','!','(',')'])
    counter = Counter()
    caption_list = []
    split = pickle.load(open('train_set.p', 'rb'))
    ann_file = os.path.expanduser(ann_file)
    with open(ann_file) as fh:
        for line in fh:
            img, caption = line.strip().split('\t')
            if img[:-2] in split:
                caption_list.append(caption)

    pool = mp.Pool(4)
    tokens = pool.map(nltk.tokenize.word_tokenize, [caption.lower() for caption in tqdm(caption_li
st)])
    pool.close()
    tokens = [item for elem in tokens for item in elem]
    tokens = [elem for elem in tokens if elem not in punc_set]
    counter = Counter(tokens)

    # If the word frequency is less than 'threshold', then the word is discarded.
    words = [word for word, cnt in counter.items() if cnt >= threshold]

    # Create a vocab wrapper and add some special tokens.
    vocab = Vocabulary()
    vocab.add_word('<pad>')
    vocab.add_word('<start>')
    vocab.add_word('<end>')
    vocab.add_word('<unk>')
    vocab.add_word('<break>')

    # Add the words to the vocabulary.
    for i, word in enumerate(words):
        vocab.add_word(word)
    return vocab
```

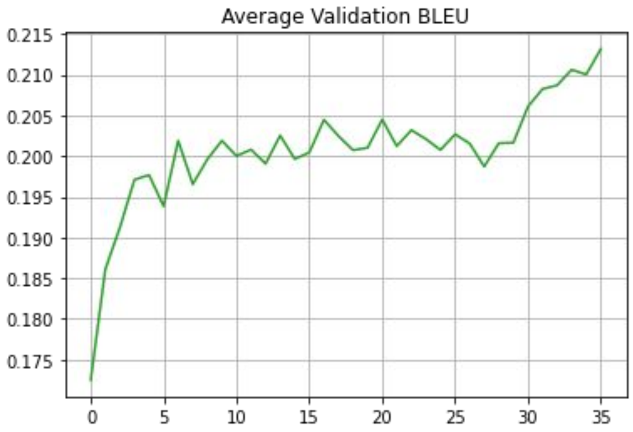
```
start_time = time()

build_vocab()

end_time = time()

print("Time for Multiprocessing with N=4 processes: %ssecs" % (end_time - start_time))
```

```
100%|██████████| 141960/141960 [00:00<00:00, 875697.33it/s]
Time for Multiprocessing with N=4 processes: 12.161671161651611secs
```

NYU

Model Results : Training

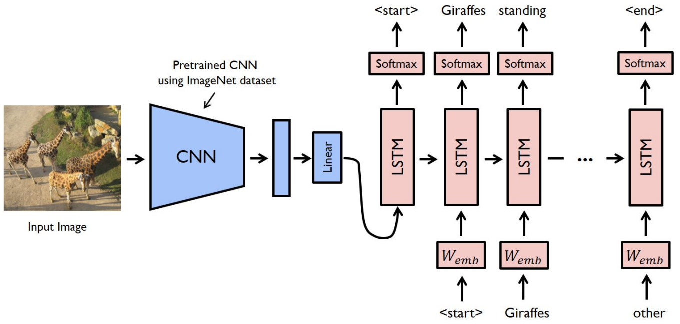


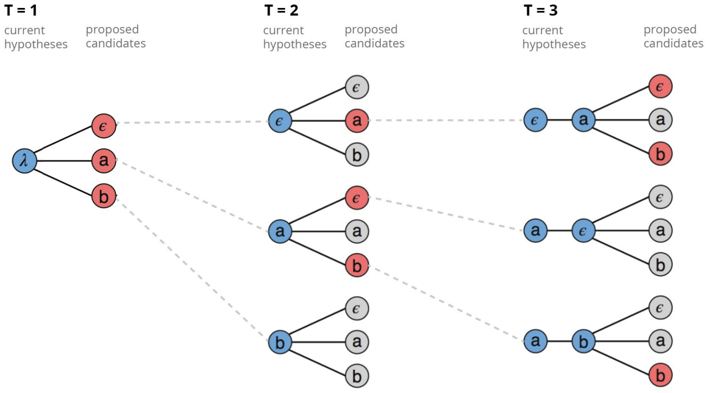


- *Encoder* : ResNet - 152 Pre-Trained on ImageNet
- *Decoder* : Single-layer LSTM with 512 hidden dimension and attention over image
- Training:
  - 30 epochs training just *Decoder*
  - 6 epoch training *Encoder* and *Decoder*
- Validation: Maximum BLEU-3 of **21.3**

NYU

Generating Text





Standard beam search algorithm with an output alphabet {ε,a,b} and a beam size of three.

- Decoder outputs the top-n best results at each timestep, along with their associated scores
- Greedy approach:
  - Always select the token with the best score
- Beam search:
  - Keep the best k results at each timestep, discard the rest

NYU

Beam Search Algorithmic Optimization

```
Timer unit: 1e-06 s

Total time: 1.30777 s

Function: beam_sample at line 200

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
   200                                           def beam_sample(self, features, targets=None, imgs=None, beam_size=3, max_seq_length=20, return_attention = False):

   *** REMOVING IRRELEVANT CODE ***
   201                                               """Beam Search"""
   236
   237       232        768.0      3.3      0.1                  for k in range(beam_size):
   238      2958       6736.0      2.3      0.5                      for j in range(batch_size):
   239      5568      12557.0      2.3      1.0                          next_candidates[j].append(
   240      5568      33011.0      5.9      2.5                              (beam.scores[j].item() + topv[j][k].item(),
   241      2784      18173.0      6.5      1.4                               topi[j][k].item(),
   242      2784      20555.0      7.4      1.6                               beam.seq[j] + [str(topi[j][k].item())])
   243                                                                   )
   244
   245
   246      2784       6841.0      2.5      0.5                          if len(next_candidates[j]) > beam_size:
   247      1824       5831.0      3.2      0.4                              next_candidates[j].remove(min(next_candidates[j])) # only the top `beam_size` candidates are needed
```

Inefficient implementation

- Our first implementation continued several nested for loops and inefficient function call overhead
- Using line_profiler, we found that this part of the algorithm took about 0.1 seconds for 16 images (The main bottleneck of beam_search is in the encoding, which we could not optimize further)

Beam Search Algorithmic Optimization

```
Timer unit: 1e-06 s

Total time: 1.26072 s

Function: beam_sample at line 200

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
   200                                           def beam_sample(self, features, targets=None, imgs=None, beam_size=3, max_seq_length=20, return_attention = False):
   201                                               """Beam Search"""

  *** REMOVING IRRELEVANT CODE ***

   256       986       2960.0      3.0      0.2                  for j in range(batch_size):
   257      1856      42397.0     22.8      3.4                      next_candidates[j] += [(beam.scores[j].item()+topv[j][k],
   258       928       2725.0      2.9      0.2                                             topi[j][k], beam.seq[j] + [str(topi[j][k])]) for k in range(beam_size)]
   259
   260        20       1253.0     62.6      0.1              next_candidates = [sorted(next_cand)[-beam_size:] for next_cand in next_candidates]

   *** REMOVING IRRELEVANT CODE ***
```

Efficient implementation

- Updating the previous code with list-comprehension and fewer function calls allowed the program to run faster, taking up only approximately 0.05 seconds per 16 images, or an speed increase of 100%
- For a full validation set of 3000 images, this means a decrease in computation time of around 9 seconds

Model Results : Inference

Greedy Search

a man in a black shirt and white hat is singing into a microphone while another man plays the drums

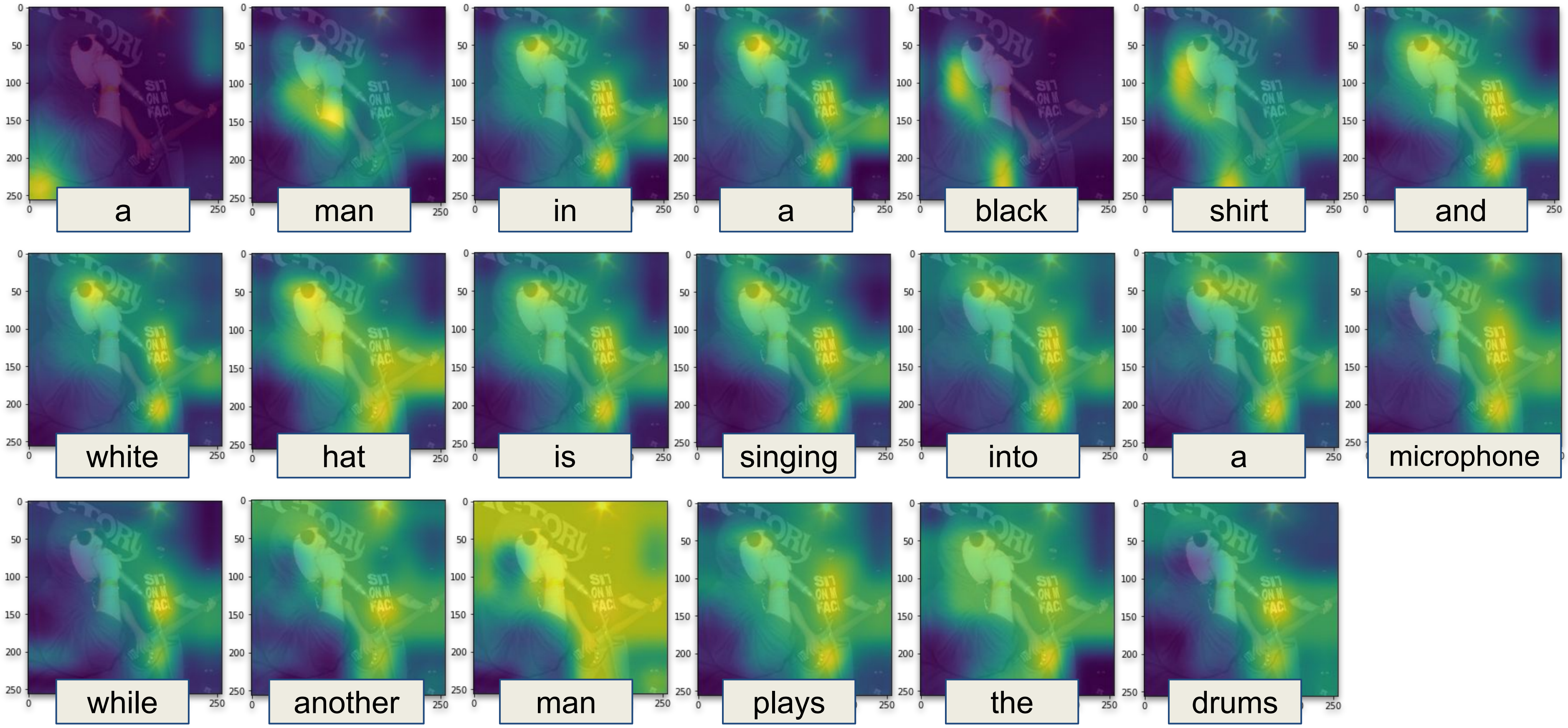
Beam = 3

man in white shirts and playing a flute with a man in a tattoo in the same with a beard

Beam = 5

three people are standing in the crowd behind them sandwich and the room with the sun shining a v belt

Model Results : Attention



NYU

Questions?