Image Captioning Generator

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

Charles Brillo-Sonnino, Aidan Claffey, Darren Geng, Meenakshi Jhalani, Kevin Wilson
NYU Center for Data Science

DS-GA 3001 Advanced Python - May 2020

1 Introduction

Generating captions for images is a task at the intersection of two of the hottest fields currently in machine learning: computer vision and natural language processing. One of Google AI's projects is to apply image captioning to text-to-speech applications for aiding the visually impaired [1]. To perform the task a machine has to not only possess the ability to be able to see and interpret images, but it also has to be able to understand and describe those images in human language. As such, training a model on this task and interpreting its results involves many optimization problems usually unique to only computer vision tasks or natural language processing tasks.

Many real-world applications of image captioning rely on speed, such as captioning the scene around a self-driving car or generating real-time alerts for CCTV cameras when abnormal activities are caught on camera, and thus many optimizations have the potential to directly improve the performance of these systems as well.

For the purposes of our task, we will be using the Flickr30k dataset and training an encoder-decoder pair. This sequence to sequence model will take images as inputs and generate English captions for those images as outputs. Along the way, we will be monitoring any potential bottlenecks that may hamper our runtime and use methods from class to alleviate those bottlenecks.

2 Dataset: Flickr30k

The dataset we used to train and implement our image captioning model is called Flickr30k [2]. Compiled by the University of Illinois in 2012, Flick30k has become a standard benchmark dataset for sentence-based image description and caption retrieval. The dataset contains 31,783 `.jpg` images as well as 158,915 captions (5 per image) in a single tokenized data file. The content of the images largely focus on people and animals involved in everyday activities. Each caption includes 13.4 words, on average, with a standard deviation of 5.4 words. Refer to Appendix A.1 for an example image and associated captions.

3 Model Architecture and Overview

Our goal was to implement neural machine translation using the encoder-decoder paradigm [3] where an encoder network builds an encoded, featurized representation of the inputs and a decoder decodes that representation into the appropriate output form. In our case, the inputs are images so we use a convolutional neural network (CNN) as the encoder and our outputs are a sequence of text so we use a recurrent neural network (RNN) decoder. Specifically, we planned to use an encoder of the the ResNet [4] family and Long Short-Term Memory [5] (LSTM) as the decoder. In addition, we planned to use input attention over the image [6]. The idea is at every decode time-step, the decoder "compares" the hidden hidden state of the LSTM to the encoded representation of the original image. That output is then concatenated with the embedded incoming word vector before being fed into the LSTM as the input. In this implementation, the comparison is done using a multilayer perception.

4 Data Preprocessing Optimization

We encountered multiple opportunities to implement methods for optimization and efficiency in the preprocessing and processing of both images and word data. As explained below, implementing Python concurrency and parallelization techniques produced significant reduction in time for many of these tasks.

4.1 Image RGB Data Normalization

Prior to training the model, it was necessary to represent the images in a numerical data structure and to normalize these values. Using Python Image Library (PIL) and Numpy, we represented each image as an array of RGB color channel values, each on a range of [0, 255]. To accomplish this normalization as part of our model training pipeline, it was necessary to compute the mean and standard deviation for each color channel across all images in the dataset. Our first pass at this task involved a pair of serialized loops in which each image file would be individually opened and the relevant values computed. We quickly realized that this process provided an opportunity to implement Python concurrency methods to optimize.

We attempted both threading and multiprocessing, testing various values for number of threads and number of processes, respectively. We found that multithreading with 4 processes was the most efficient method, taking 104.2 seconds in total to compute the RGB means and standard deviations across all images in the dataset. Notably, this is just a fraction (about 22%) of the time taken without using concurrency to optimize the code, which totaled 472.9 seconds.

While threading also produced an improvement, it was not as fast as multiprocessing. This is likely due to the nature of the task being CPU-bound (rather than external I/O bound), as the images were pre-downloaded and exist on the local machine. Thus multiprocessing provides a more truly parallel technique as compared to threading.

A visual summary of the speed improvements is included in Appendix A.2. An overview about these techniques and detailed code are included in `Image Data Preprocessing with Concurrency.ipynb` in the project Google Drive. We also created a script to reproduce this operation on future data, with runtime argument options for various methods of Python concurrency. This can be found in the `ImageMeanStdev.py` file also in the project Google Drive. For our purposes, the results of this operation are arguments for the data normalization step, implemented with PyTorch in `train.py`.

4.2 Word tokenization

For the computer to be able to process and analyse large amount of captions, we need to build a vocabulary which will be used as features in training the model. This building of vocabulary progresses in the following manner : i) raw captions corpus ii) processed captions iii) tokenized captions into words iv) corpus vocabulary

Using nltk.tokenize module, we tokenized/split the captions into words and stored them in a dictionary called Vocabulary. We decided that our tokens are going to be limited to only words and numbers, and therefore while tokenizing we needed to be careful to not include punctuation marks like comma, colon, etc. We chose to only store words with count greater than four which helps reduce the complexity. We also added special tokens - unk (a token that does not exist in corpus or has been removed), pad (a token for padding), start (a token to present the beginning for a sentence) ,end (a token to present the end of a sentence), break (a token to represent separation of sentences).

This process is reliant on multiple for loops which clearly needed to be optimised for better performance. We decided to employ concurrency methods like multiprocessing and threading to help achieve better efficiency in processing captions. We found that the most efficient method is multiprocessing with N=4 processes. It took 12.16 seconds to generate the vocabulary as compared to the non-optimized version which took 26.2 seconds. However, threading did not show an improvement in performance. This can be explained by the fact that the task of generating a vocabulary involves more interactions with the CPU and hence multiprocessing is better equipped to handle this task.

A summary of the speed improvements is included in Appendix A.3. An overview about these techniques and detailed code are included in Build$_vocab.ipynbintheprojectGoogleDrive$

5 Training

In order to efficiently implement training this network, we needed to stack the images and and captions as tensors and feed them into a GPU. In order to do this, we had to make both inputs and outputs the same size. We achieved this by cropping our input images to 256 pixels which necessitated dropping 322 samples and by padding our outputs to the maximum sequence length of the batch size. We used a random crop on the training set (as well as a random horizontal flip) to add robustness to our model while only a central crop was used on the validation set.

Our model architecture ended up being particularly difficult to train. The loss function easily exploded with too high a learning rate. Our first model with a ResNet-50 encoder had very poor performance so we upgraded to a ResNet-152 that was pre-trained on ImageNet. We also tried to improve performance by initializing the embedding weights with Spacy pre-trained embedding but this also caused the loss function to eventually diverge. Lastly, we found that training both the encoder and decoder simultaneously caused the network to get stuck in poor local optima. Our best model resulted from training just the decoder network for 30 epochs with a learning rate starting at 0.001 and annealing by $\frac{1}{10}$ every 10 epochs, then switching over to training both the encoder and the decoder for another 6 epochs.

6 Validation

6.1 BLEU Score

BLEU (bilingual evaluation understudy) score was the primary metric used to evaluate our models on. Originally developed as a validation metric for machine translation tasks, nowadays it is used to evaluate the quality of text generated from a wide variety of tasks. BLEU score takes generated sentences and a corpus of reference sentences as input and outputs a modified precision score in $[0, 1]$.

BLEU score was an area that we looked to optimize, though `line_profiler` revealed that there were very few optimizations to be done as the vast majority of the time spent was in generating predictions via the encoder-decoder pair. The process could not be parallelized either, as BLEU score must be calculated all at once and thus splitting its inputs into chunks would not produce an accurate result.

We used BLEU-3 as our validation set metric which compares n-grams up to size 3 between the target and the prediction. The maximum BLEU-3 score we achieved was 21.3.

7 Prediction

7.1 Algorithmic optimization

To predict captions based on the trained model, we compared two different algorithms for selecting ordered tokens.

1. Greedy search

The simpler algorithm we implemented is a greedy search that takes the highest scoring token at each sequence iteration and stops after `max_sequence_length` iterations. Since the algorithm operates on a sequential basis with only one token choice per image, the only optimization we can make to improve computation is to parallelize the greedy search. Since we are using CUDA to train and validate, CUDA will outperform any CPU multiprocessing or multithreading.

2. Beam search

   Beam search, while still considered a 'greedy algorithm', looks through multiple trees, or beams, to find the best token sequence. At each iteration, it selects the top k beams based on the score of the network output for each token based on the previous token sequence, and then it continues searching until, like *greedy search*, it has reached `max_sequence_length` iterations. Then the top sequence is chosen based on the softmax of the network output for the remaining beams. Refer to Appendix A.4 for a graphic on beam-search. We tried two scoring methods for selecting token sequences. The first used an additive method of scoring, which summed the direct network outputs of the top tokens to each sequence in each iteration. The second took a softmax of the network output scores for each token and multiplied those softmax scores iteratively. The first gave better results, so we kept the first implementation.

   Since beam-search searches through multiple token sequences in each iteration, it can be optimized through list comprehension and reducing function call overhead. In our initial implementation, we use several for-loops to iterate through each image set to add each token sequence and remove a sequence if it had the worst score. This implementation was slow due to the many for-loops, and using `line_profiler` we found the bottlenecks in the search and improved them through list comprehension and reducing function call overhead. Specifically, instead of adding one token sequence per image per iteration and removing sequences during the for-loop, we append all sequences at one time per iteration and remove after all sequences have been added.

   According to a test run with an image batch of size 16, this change led to a speed increase of approximately 100% for this part of the calculation, from around 0.1 seconds to 0.05 seconds. So, for a full validation set of 3000 images, this would lead to an approximate increase of around 9 seconds for a process that would take around 220 seconds. As mentioned before, the main bottleneck in the beam-search algorithm is the encoding of each image, which cannot be improved based on improvements from course material. The `line_profiler` output for the efficient and inefficient can be found in full in the project Google Drive, and the full code (along with the commented out inefficient code, can be found in `beam.py` and the `beam_sample` function of the `DecoderRNNwithAttention` model of the `model.py` file.

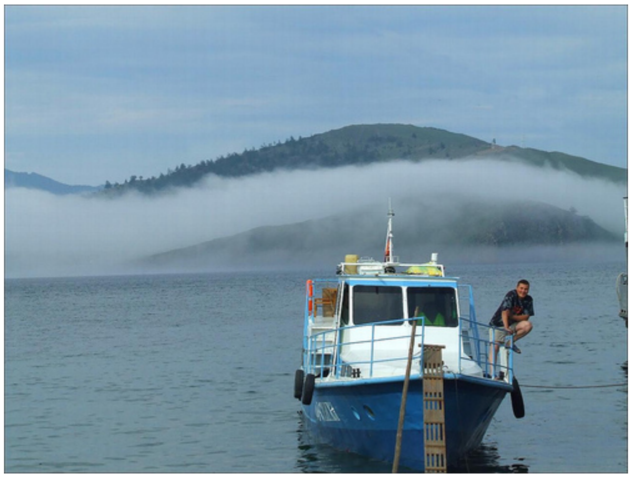
8 Conclusion

Overall we are quite pleased with the results of this exercise. While our image captioning model itself is not novel (and the dataset and methods used are used widely to address the task of image captioning), the techniques for optimization and efficiency we implemented demonstrate significant improvements over existing implementations. We have shown that tools such as `line_profiler` enable the identification of bottlenecks or problem areas in code and direct the programmer's attention to areas for improvement throughout the machine learning workflow. Efficient programming techniques such as list comprehension in place of for-loops and reduction of function call overhead can significantly reduce running time, as evidenced in our beam search optimizations. Concurrency methods such as threading, and particularly in our case multiprocessing, offered drastic improvements on CPU-dependent tasks such as image data computations and word tokenization.

Finally, the use of CUDA/GPUs was crucial to efficiently train and validate the model.

Appendix A

A.1 Flickr30k example image and associated captions



1. A man in shorts and a Hawaiian shirt leans over the rail of a pilot boat, with fog and mountains in the background.

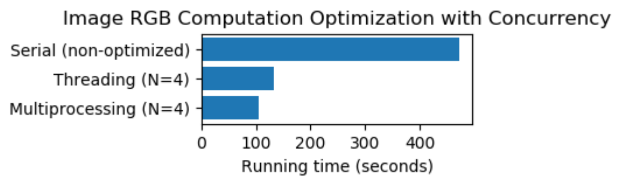
2. A young man hanging over the side of a boat, which is in a like with fog rolling over a hill behind it.

3. A man is leaning off of the side of a blue and white boat as it sits in a body of water.

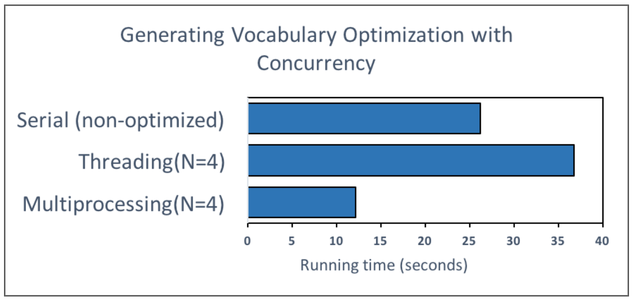
4. A man riding a small boat in a harbor, with fog and mountains in the background.

5. A man on a moored blue and white boat with hills and mist in the background.

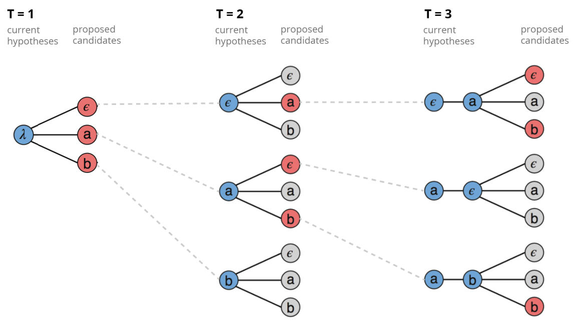
A.2 Python concurrency methods to optimize image data preprocessing



A.3 Python concurrency methods to optimize generation of vocabulary



A.4 Beam Search example



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

References

[1] Google. *Google's Conceptual Captions*. 2019. URL: `https://ai.google.com/research/ConceptualCaptions`.

[2] Peter Young et al. 'From image descriptions to visual denotations: New similarity metrics for semantic inference over event descriptions'. In: *Transactions of the Association for Computational Linguistics* 2 (2014), pp. 67–78. DOI: `10.1162/tacl_a_00166`. URL: `https://www.aclweb.org/anthology/Q14-1006`.

[3] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. *Neural Machine Translation by Jointly Learning to Align and Translate*. 2014. arXiv: `1409.0473 [cs.CL]`.

[4] K. He et al. 'Deep Residual Learning for Image Recognition'. In: *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2016, pp. 770–778.

[5] Sepp Hochreiter and Jürgen Schmidhuber. 'Long Short-term Memory'. In: *Neural computation* 9 (Dec. 1997), pp. 1735–80. DOI: 10.1162/neco.1997.9.8.1735.

[6] Kelvin Xu et al. *Show, Attend and Tell: Neural Image Caption Generation with Visual Attention*. 2015. arXiv: 1502.03044 [cs.LG].