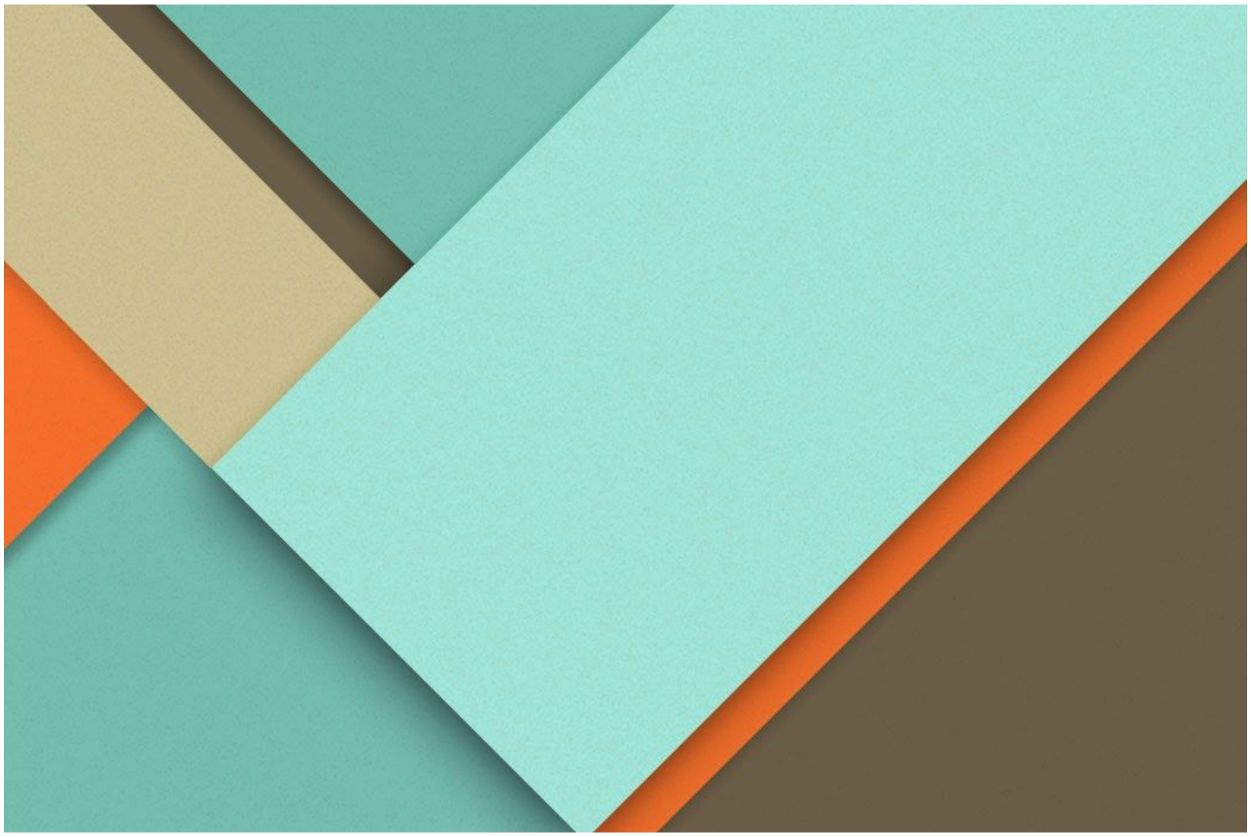

Image Captioning

Final-Term Project 19.12.2020

Jonathan Gruss
Sungkyungkwan University
2020318372

Introduction

When I first thought about the problem of creating captions for images, a few conceptions sprang to my mind. Firstly, this is a one to many problem that has an image as input and wants a caption(many words) as the output. Secondly, an image as input most likely means that a convolutional neural network will be good to capture these features. Thirdly, for the captions a recurrent neural network makes sense, an LSTM seems like a good choice there. This LSTM part should be a many-to-many sequential data type.

Dataset

The Dataset was the first thing to program in the project. I started off with reading the json file for the file_names and captions. After that, I tried reading the images with cv2.imgread() and storing them in an array, which I later noticed that this is too large for the RAM and not the right way to do it anyway. After researching for the correct way to do it, I ended up reading the image with PIL.Image.open(img_path).convert("RGB") in the __getItem__() method. This __getitem__() method gets called by the data loader to load the image and caption tuple during training and validation in batches.

Later on while I was implementing the decoder, I saw that my data was returning the captions as strings while I needed them tokenized and in uniform length. Thus, I had to tokenize the captions in the __getitem__() method. For that I created the tokenize_caption() in my Vocabulary class.

The thing I struggled the most with throughout the project was loading the images both at an acceptable speed and reliably. At the beginning, I would get errors from the dataloader about non-existing paths although some did exist, when I loaded them individually. I ended up checking if the path exists when loading the file_name. To increase the speed I tried many different libraries but nothing ended up being an improvement over pillow. The reason for the slow speed was not the loading function. I assume that Colab itself was really slow in finding the images and loading them from the drive. The speed and reliability increased significantly after the images had been loaded once in the past.

DataLoader

I programmed a function that both splits the dataset into a validation and training set and returns a DataLoader for both sets. I randomly split the dataset with the random_split function.

Preprocess

The preprocessing of the images is very minimal. Since, the model should caption various aspects of an image, I did only resize it to be uniform, added a random horizontal flip, converted the PIL.Image to a tensor and normalized this tensor with values used for the imgnet dataset.

As already mentioned, the captions had to be tokenized for the lstm model. For the tokenization I used NLKT. I after counted the occurrences of the words and removed the ones under a certain threshold, since they won't be useful for training. I added a start and end token to identify the start and end of a caption. Furthermore, the length of captions are varying but the model requires it to be a fixed size. For that reason I set a max_len for the captions and either cut them or added a pad token if the length was not correct.

Vocabulary

To organize the word tokens I created a vocabulary class. The main attributes of the class are a word2idx and idx2word dictionary. This way I can get the indices for the model to train on and later convert the models caption output back to words. I have added start and end tokens to identify the start and end of the captions. A pad token to add padding to short captions and a unk token to identify words that are not in the vocabulary. These correspond to the indices 0 to 4 in the idx2word dict. For example, a raw sentence "I love deep learning" will be tokenized by NLTK into ['i', 'love', 'deep', 'learning'] and eventually become [0, 16, 32, 23, 1, 2, 2, 2, 2, 2, ...].

Encoder

The encoder gets the images as input to create a smaller summary representation of the useful attributes in the original image. For that I used the pre-trained resnext101_32x8d an improved version of the normal resnet with 101 layers (the best performing model available on torchvision.models). Since, the resnet is already pre-trained I fixed the weights to save training time. Because I need to extract the features and not classify the image I remove the last layer. Furthermore, I added a linear and a batchnorm layer, the first to connect the feature vector to the lstm model and the batchnorm to normalize it.

Decoder

The decoder's job is to generate a caption word by word from the feature vector given by the encoder. To increase the representation ability of my model, I start the decoder with a word embedding layer that turns the tokenized captions into a vector. I then added the LSTM model that gets the features concatenated with the embeddings as input. At the end I map the LSTM output to the vocabulary size with a linear layer. This layer will decide which token will be chosen next.

To test an image I created a sample method. This function creates a caption in tokenized form based on the image features the encoder gave us and the previous token prediction (state). The model's prediction gets generated by the LSTM and classified by the linear layer. The embedding layer then uses the prediction to generate the input for the prediction of the next token together with the states output of the LSTM's last round. It loops until the given max caption length is reached.

Training

The training process is done very ordinary. I use the cross entropy loss since I practically want to classify the generated tokens with the caption tokens. My optimizer is Adam with a learning rate of 0.001. I train and validate the models for ten epochs, which I found to be a good number with little overfitting and a validation loss that stagnates only at the very end.

Testing

To have images for testing I gave my dataset the option to exclude a given amount of images from the main data and use those as the testing images. This is only for my testing. Since I do not know how many images will be tested. I decided to create a simple dataset for the test images and load the data with a data loader. I first load the vocabulary file with pickle. Create the dataset with basic image preprocessing. Initialize the encoder and decoder and load the weights for both of these models. I then loop over every image in the data loader and the batch while generating the captions with the encoder and the sample method from the decoder. Afterwards, I convert the tokenized captions back to sentences. I then store these sentences together with the file name of the image in a json file.

Further improvements

While I could not finish any further improvements, I also worked on introducing an attention model and a beam search algorithm. The attention model adds weights to a specific part of a picture where it thinks it should focus on. These weights also have to be aware of the sequence that has been generated so far. So, after each token is generated, the next token will, additionally to the previous token sequence, also be based on the attention weights. The difficulty in the implementation was that you cannot use the full LSTM model anymore, but have to use LSTMCells and handle the in and output for each of them in a loop.

Another possible improvement I was working on was the beam search algorithm. Currently, the sample method in the decoder chooses the word with the highest score and uses it to predict the next word. If the first word choice was not optimal, the next choices will likely be even worse, the error propagates. Beam search does not decide for a word or sequence fully until a given step size and then chooses the sequence with the best overall score.