Image Captioning using Attention-based Encoder-Decoder Architecture

Undergraduate Machine Learning Final Project Report
Steven Jiang, Rowan University

1. Introduction

The task of image captioning sounds quite simple. Given an image, generate one or two sentences that depict the objects in the image and what those objects are doing. Image captioning combines the tough tasks of both computer vision and natural language processing. Not only do machine learning models need to accurately identify the objects in a given image, they also have to precisely construct and express the caption in a natural language readable by a human.

On the computer vision side, Convolutional Neural Networks (CNN) have become a standard for object detection. More and more architectures have been developed leading to better image analysis and object detection. On the natural language processing side, the evolutions of the sequential Recurrent Neural Networks (RNN) have also led to big accomplishments. Starting with the vanilla RNN's, moving onto the mainstream Long Short-Term Memory (LSTM) and Gated Recurrent Units (GRU), and finally to the recent addition of the Transformers. With all these recent breakthroughs and works of both sides, the quality of the image captioning output is significantly improved.

2. Proposed Approach

In this section, I will talk about my plan and proposed approach to solving the task of image captioning. The core ideas are illustrated in this paper [1], but there will be some alterations to provide a way of comparison between architecture and results.

2.1 High-Level Rundown

Figure 1 below shows the high level rundown of the entire project. The first step is to obtain our vocabulary of words from the training captions for our Recurrent Neural Network (RNN) to utilize. The next step is to prepare our encoder-decoder architecture. We will use a Convolutional Neural Network (CNN) to generate some feature vectors that represent the given image. This basically means that we convert the image to its numerical representation. We then feed the features vectors into our RNN, which will word by word generate a short caption for our image. In the following subsections, I will go more in-depth about each component.

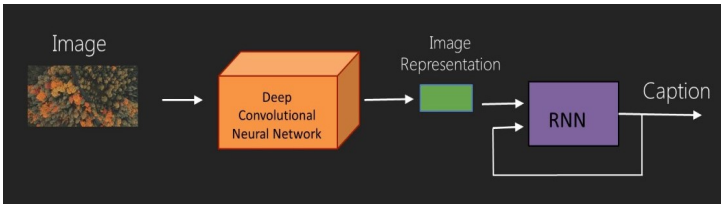

Figure 1. High level overview of the entire project.

2.2 Encoder-Decoder Architecture

The encoder-decoder architecture is a neural network design pattern consisting of an encoder and a decoder. The encoder takes an input and encodes it to a smaller dimension, capturing useful feature data. The decoder attempts to take the feature data and reconstruct it back to the original input. The CNN and RNN in our architecture will model the encoder and decoder parts, respectively. The major reason that image caption generation is well paired with this architecture is that this task is basically translating an image to a sentence(s).

2.3 Convolutional Neural Network

The CNN model I will be using is Inception V3 [3]. The model is 48 layers deep. It requires input vectors of size 299 x 299 x 3. Those vectors will be passed through several convolutional layers and concluded by a fully-connected network to reduce the dimensionality even more. The final dimension of the output data is of size 8 x 8 x 2048. Since Inception V3 is already pre-trained on the ImageNet dataset, we won't have to train

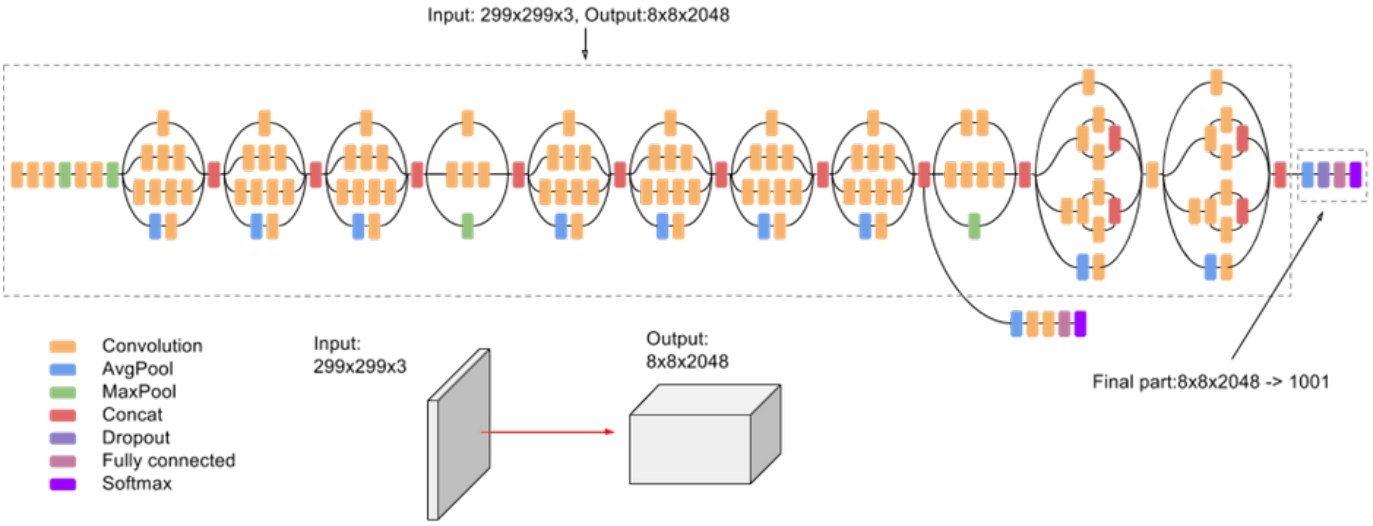

Figure 2. High level overview of the Inception V3 architecture.

it again, though accuracy results may vary since I am using the Microsoft COCO (MS-COCO) dataset.

This model was built mainly focusing on using less computational power. And this proved successful. In comparison to its previous versions and other CNN models, it has proved to be more computationally efficient. This is in terms of both the number of parameters generated by the model and the cost of memory and other computational resources. It is suggested that the below techniques have led to better optimize the network. Figure 2 above shows the general architecture.

1) Factorized Convolutions
   - Reduces the number of parameters involved in the network.
   - Keeps a lookout on the network efficiency.
2) Smaller Convolutions
   - Replaces bigger convolutions with smaller convolutions.
   - Example: A 5x5 filter has 25 (5*5) parameters, while two 3x3 filters has only 18 (3*3 + 3*3) parameters.
3) Asymmetric Convolutions
   - Also replaces bigger convolutions with smaller ones.
   - The idea that an n x n convolution is better replaced with a n x 1 convolution followed by a 1 x n convolution
4) Auxiliary Classifier
   - Small CNN added between layers during training.
   - Loss obtained there will be added to the main networks loss.
   - Allows for loss to be considered more during back-propagation.
5) Grid Size Reduction
   - Performs strides on the base feature maps.
   - Much faster and more efficient than pooling.

2.4 Recurrent Neural Network

The RNN I will be using is the Gated Recurrent Units (GRU). For my use case, GRU's should be sufficient, in terms of performance, because we only want to generate one to two sentences. GRUs and LSTMs are perfect when they don't have to remember a large context window. If this project had to generate a paragraph worth of caption, then I would use their stronger cousins, the Transformers [5].

The output of the CNN will become the input for the RNN. The output for the RNN at time step i will be the input for time step $i + 1$. During training, I will use teacher forcing to provide the model with a ground truth from a given time step as input for another time step. Essentially, if the first or second predicted word is way off from the actual word in the caption, we don't want the model to automatically get a horrible caption. More details about this will be provided when we get to the training process.

2.5 Attention Mechanism

The attention mechanism comes into play when the RNN is trying to predict a part of the output sequence. It enables the RNN to focus on a certain part of the input sequence. In our case, the input sequence is the vector form of an image in which we are trying to generate a caption for. This paper [2] provided a fundamental basis for the implementation of an attention mechanism which I will fine-tune for my project.

The basic idea of the attention mechanism is to avoid giving the decoder a single vector holding all the information about the image (it wants to generate a caption for). We don't want it to attempt to learn a single vector representation for each image. Instead, using attention, the decoder can focus on specific areas of the image while generating a word for that area. This is done with attention weights.

The specific attention mechanism I will be using in this project is the additive attention. Based on its name, we can infer that it performs a linear combination of the encoder and decoder states. Its score function is pictured below which gives each word a score, telling the decoder which words to focus on while going through the image.

$$\text{score}(\boldsymbol{s}_t, \boldsymbol{h}_i) = \mathbf{v}_a^\top \tanh(\mathbf{W}_a[\boldsymbol{s}_t; \boldsymbol{h}_i])$$

where v_a and W_a are matrices that represent the learned attention parameters and s_t and h_i are the hidden layer states.

3. Experiments/Data

As previously mentioned, I will be training and testing my models on the Microsoft COCO (MS-COCO) 2014 dataset [4]. The dataset contains over 82,000 images with each image having 5 captions. I will be using Google's Colab for running the program on GPUs to make execution faster. After training, I will test the accuracy of the trained models by using a combination of random images I find and the testing images provided by MS-COCO. The paper [2] used a different attention mechanism (multiplicative attention). Therefore, by additive attention for this project, I will be able to compare the results of using different attention mechanisms.

3.1 Thoughts to Keep in Mind

- Prediction accuracy will depend on the training instances. If I test the trained model and get a poor caption result, then it could be due to overfitting/underfitting.
- Experiment with different CNN/RNN models such as ResNet50 for CNNs and LSTM for RNNs.
- For our use case, using LSTM or GRU will be fine since our training captions are rather short in length. These two RNN architectures suffer from long term memory, meaning that for long sequences, they will have a hard time remembering earlier parts of the sequence.

4. Experimental Process

This section will go into detail the entire experimental, training, and testing process.

4.1 Data Preprocessing

The first part was data preprocessing which by far the most tedious but most important part of the entire project. I have downloaded all the data and stored them in a Google Drive folder since Google Colab doesn't store them on the cloud instance after disconnecting. This saves the hassle of having to download the data every time I connect to the instance. An example of an image along with its caption is shown below in Figure 2. In the next two subsections, I will talk about the preprocessing steps for the images and captions individually

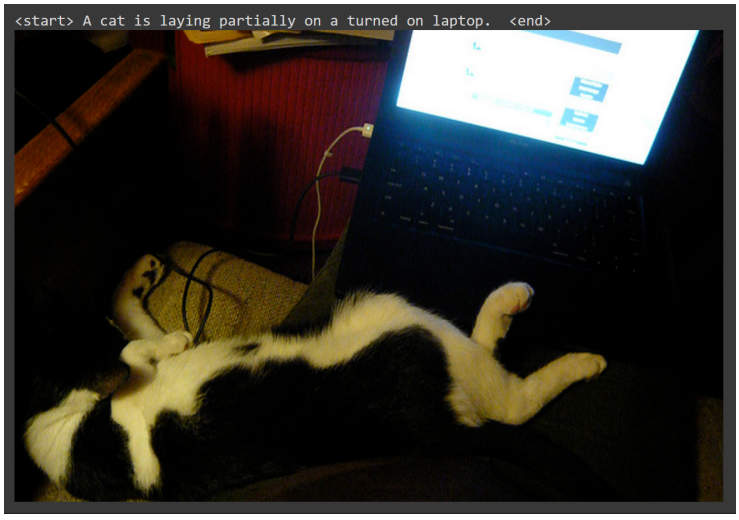

Figure 3. A sample image with one of its caption.

4.1.1 Image Preprocessing

Like previously mentioned, there are over 82,000 images in the 2014 MS-COCO dataset. Had I trained the network on the entire 82,000 images, I would've have little to no results. Therefore, the first step for preprocessing the image data is to heavily reduce the number of training images. I chose to reduce the number of training images down to about 5,000 to 10,000. As a result, this would decrease the training time.

The next step is to reshape the images to a size of 299 pixels by 299 pixels by n, where n represents the number of color channels in the images. For MS-COCO, each image is RGB (red, green, blue), so there are 3 color channels. Remember, the input images passed to Inception V3 must be of shape 299 by 299 by n, as shown in Figure 4 below. Had the images been grayscale images, n would have been 1 and there would have been a reduction in computation complexity. Since color images contain more information than black and white images, complexity is added and more space in memory is taken up.

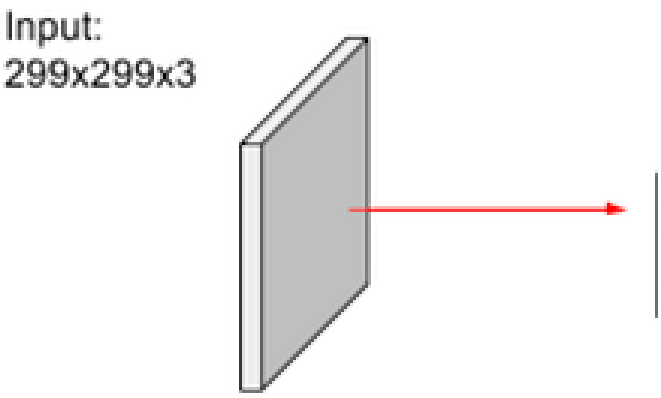

Figure 4. Reshaping images to required shape.

The last step is to normalize each pixel values to be between -1 and 1 for faster convergence, although 0 to 1 would also work if we weren't using Inception V3. Normalization makes computations much more efficient and brings the image pixels into a range of values that is closer to the normal distribution. Not much surprise here as these are some of the standard procedures for preprocessing image data.

4.1.2 Caption Preprocessing

For preprocessing captions, more steps had to be taken before moving on to the training process. Let's walk through them.

1) Bag of Vocabulary

   The first step is to obtain our bag of vocabulary. This means that we have to go through all of the words in the captions and add each unique word to our collection. We will choose the first n words. This can be done by splitting a caption on spaces. For example, let's say we have the following sentence, "A boy is playing basketball in the park". By splitting on a space, our bag of vocabulary would get the following words added on: "A", "boy", "is", "playing", "basketball", "in", "the", "park". Once we have our n words, all other words will still get added to our bag but they get replaced with the <end> token. This is since the RNN network may predict a value that doesn't correspond to any of the words in our bag.

2) Add Tokens

   The next step is go through each caption in our training set and prepend a <start> token to the beginning and append an <end> token to the end. Let's say one of our original training caption is "a living room with a television and a brown couch". After the two tokens are added on, we get "<start> a living room with a television and a brown couch <end>". This is so when the captions reach the RNN, it will know, surprise, when to start and end.

3) Convert Words to Numbers

   Now that our captions have their appropriate tokens added, we can move on to converting each word and token to their numerical representation. This process is similar to one-hot encoding where we convert a categorical value to a numerical value. How would we accomplish this? Well, all we do is associate each word's position in the bag and use that value. So if the word "basketball" was the seventh word added into the bag, then it's corresponding numerical value would be 7. We do this for all of the training captions.

4) Pad Trailing Zeros

The final step in the caption preprocessing process is to append zeros at the end of each numerical vector. How many zeros do we add? We keep padding zeros until the length of the vector is equal to the length of the longest caption vector. This is because I use batch processing later on in the training process, and it's helpful for each caption to be of the same length.

The below figures illustrate and summarize this entire preprocessing procedure for captions. Figure 5 shows the two tokens added in their respective positions. Figure 6 shows the numerical representation of the sentence in Figure 5. Figure 7 shows the padded zeros at the end of the vector.

```
<start> a living room with a television and a brown couch <end>
```

Figure 5. Respective tokens appended and prepended.

```
[3, 2, 127, 46, 9, 2, 312, 10, 2, 116, 159, 4]
```

Figure 6. Words converted to numerical representation.

```
[3, 2, 127, 46, 9, 2, 312, 10, 2, 116, 159, 4,
 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

Figure 7. Trailing zeros padded at the end.

4.2 The Training Loop

With all the pieces in place, it is finally time to train our models and the network. Figure 7 provides a pseudo-code representation of the actual training function.

We see that it accepts two parameters as arguments: the image tensor and its corresponding target caption tensor. Our loss value begins at zero. As mentioned before, the starting input to the decoder will be the <start> token. We pass the image tensor to our CNN encoder. Recall that Inception V3 outputs tensors with a shape of 8 by 8 by 2048. We actually run those tensors through one last fully-connected layer to output our feature vectors of shape 64 by 2048. The hidden state will initially be zero and reset on every epoch.

Moving on to the loop itself. This will run through each word of the caption. This means it will go from 1 to the length of the longest caption. If we pass the three variables previously described to the decoder, it returns back a prediction tensor along with the new hidden state. We use our loss function, which is the sparse categorical crossentropy, on the target and prediction to calculate the loss for that round.

The final thing to prepare the decoder for the next round is the input. We use teacher forcing which is the technique where the target word, for the current round, is passed to the decoder as the next input. Clearly, this is still the training step and is the basis for supervised learning. Lastly, though not displayed in Figure 7, is calculating the gradients and applying them to the optimizer and backpropagate.

```
def train(image, target):
    loss = 0

    input = <start> token
    features = encoder(image)
    hidden = initial hidden state

    for i in range(1, target.shape):
        predictions, hidden = decoder(input, features, hidden)
        loss += loss_function(target, predictions)

        input = target word for this iteration

    return loss
```

Figure 8. Pseudocode of the train function.

4.2.1 Loss Plots

To clarify again, the loss function that I used in the training loop was sparse categorical crossentropy. This is because the caption labels are provided as integers and are not one-hot encoded. Had they been one-hot encoded, then I would have used just a regular categorical crossentropy loss function.

Figure 8 below shows three "different" loss plots for varying n number of images trained on. I trained the network on 5,000 images, 7,000 images, and 10,000 images. Though the three plots all look the same, there is a small difference between the final values. The right-most plot had the lowest loss and therefore led to generating the slightly better captions in the end.

One thing I wanted to point out was the fact that I only ran the training loop for 10 epochs. Just from looking at the loss plots, this was clearly not enough as the plots didn't even converge yet. Due to the time it took for one epoch, I had to limit the training epochs.

Figure 9. Loss plots with varying n different number of images trained on.

4.3 The Evaluation Loop

After the models are trained, we are ready to move on to the evaluation stage. In other words, we can begin testing our models and begin generating captions for images! Figure 9 provides a pseudo-code representation of the actual evaluation function.

Unlike the training function, which takes in two parameters, the evaluation function just needs one parameter which is the image tensor in which we want to generate a caption for. In fact, the evaluate function is very similar to the training function, except we don't use teacher forcing in this function. The first thing is to reshape the input image data to the shape that Inception V3 likes (299 by 299 by n). Keep in mind that we only conducted image preprocessing on the training dataset. We never touched the testing images. Lines 4, 5, and 6 are the exact same as the ones in the training function. The only difference is, again, using the reshaped input image and not the original input image. There is also a results list to store the final caption for the image, which gets word by word appended onto it.

The loop itself will iterate however many times the length of the longest caption is. Same as the training function, we pass the three variables into the decoder which also returns a prediction tensor and the hidden state for that iteration. The predictions tensor isn't directly associated with one word, since the RNN could have predicted multiple candidate words for the current iteration. Using the categorical function from TensorFlow, it draws a sample using a categorical distribution. That function returns an id for the final selected and predicted word for the current iteration. We use that id to grab the word from our bag of words and append that to our results list. Finally, we say that the input to the next iteration of the decoder is its previous predictions along with the hidden state for the current iteration. The encoder feature vectors will always stay the same. We keep iterating until the model predicts the <end> token and we return the predicted caption list.

```
1  def evaluate(image):
2      reshaped_image = image.reshape
3
4      input = <start> token
5      features = encoder(reshaped_image)
6      hidden = initial hidden state
7
8      results = []
9
10     for i in range(length of longest caption):
11         prediction, hidden = decoder(input, features, hidden)
12         predicted_id = categorical(prediction)
13         results.append(bag_of_words[predicted_id])
14
15         if bag_of_word[predicted_id] == <end> token:
16             return results
17
18         input = prediction
19
20     return results
```

Figure 10. Pseudocode of the evaluate function.

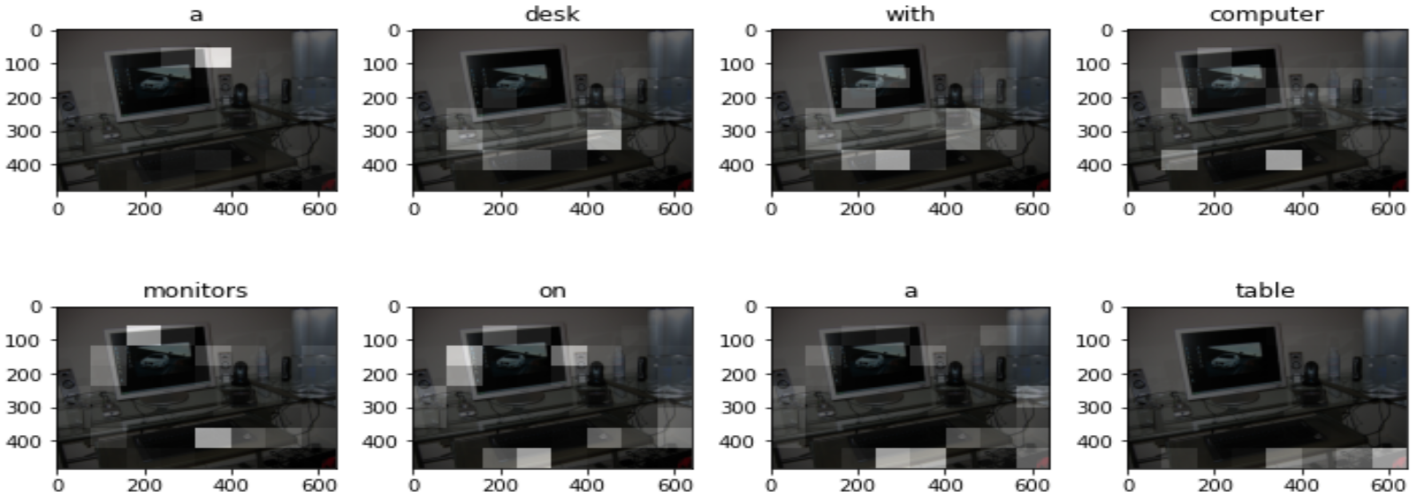

Figure 11. Attention Plot.

5. Attention Visualization

In this section, I will talk more about how the attention mechanism can be visualized and how it allows our decoder (the RNN) to retain context to a specific area in the image. I will be referencing Figure 10 for the remainder of this paragraph. Taking a look at the second image first, we can see that there are brighter squares around the desk area. For this current iteration in the RNN, it has more attention and context on those brighter areas. When the iteration concludes, we see that the model has predicted the word "desk" which does seem to depict the object that the brighter squares are visible around. The fourth and fifth images also seem to correspond pretty closely to "computer" and "monitors" respectively.

But how is all of this possible? Well, I actually forgot to mention one small, yet important, piece in the training function in Figure 7 above. The attention mechanism residing in the decoder computes the attention weights. The attention weights are computed by passing the score matrix (algorithm pictured in Section 2.5) through a softmax layer. Using the attention weights and the image feature vectors passed in, the attention layer computes a context vector. It is this vector which gets combined with the caption data that gets sent to the RNN.

6. Example Outputs

Real Caption:
<start> a brown cow with white spots walking across the field <end>

Predicted Caption:
a cow stands next to each other in the grass <end>

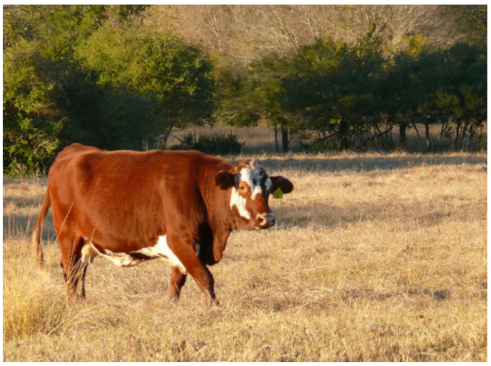

Figure 12. Example output: trained on 5,000 images.

Real Caption:
<start> blackened chicken sandwich with lettuce tomato mayo and fries <end>

Predicted Caption:
a plate of food is laying on a white plate <end>

Figure 13. Example output: trained on 7,000 images.

7. Data Issues and Limitations

The entire project definitely had a few flaws that prevented the models from generating better captions. In this section, I will list a couple of those flaws that relate to data issues and training limitations.

- Dataset Distribution

  How the dataset is distributed into a training set and a testing set doesn't sound like a contributing factor to poor captioning results. Though it can. By only pulling the first *n* unique words from the captions, we have the possibility of leaving out words that some images might need in order to generate a semi-accurate and readable caption. What if our bag of vocabulary just contains mostly animal words such as "giraffe" or "swarm"? When we move to the testing phase, the captions for images such of buildings and vehicles will include a lot of <unknown> tokens and/or be very off from the actual caption.
- Training Epochs

  The other issue is not enough training epochs which I mentioned earlier. I had only ran the training function for 10 epochs as shown in Figure 8. This clearly wasn't enough as convergence was not reached yet. I definitely would have expected better caption results had I ran the function for 20 or 30 epochs. Depending on the size of the training images, each epoch took around 10-20 minutes to complete, so with the 10 epochs, the entire training function took anywhere from an hour to three hours. Had I gone up to 20 epochs, this would have meant that I had to wait about three to eight hours. The problem was that Google Colab would keep disconnecting if it detected inactivity, even though the training function was still training.

Both of these issues are part of the memory-accuracy tradeoff. By only grabbing the first *n* words and not all unique words from the caption dataset, we gave up accuracy for more memory space and faster training time. Similarly, we also gave up the number of training epochs for faster training time. The results in the end will be reflected. I should have also spent more time looking at fine-tuning the hyperparameters to see which set of numbers gave the lowest loss and eventually, the better captions.

8. Conclusion

I learned a lot of new amazing machine learning techniques and also brushed up on previously learned concepts in this project. The data preprocessing was definitely the most tedious part of this entire project because if this step wasn't set up properly, the models wouldn't be able to run at all. The attention mechanism was just a fairly new concept implemented not too long ago. It's addition to the machine learning world, specifically to the natural language processing side, allowed higher achievements that previously weren't doable. If I were to redo the project from the beginning, I would definitely try to get more epochs into the training function and attempt to separate the images into different categories to prevent outliers of a particular class.

References

[1] Kelvin Xu et al. *Show, Attend and Tell: Neural Image Caption Generation with Visual Attention*. 2015. https://arxiv.org/pdf/1502.03044.pdf

[2] Dzmitry Bahdanau et al. *Neural Machine Translation by Jointly Learning to Align and Translate*. 2014. https://arxiv.org/pdf/1409.0473.pdf

[3] *Advanced Guide to Inception v3 on Cloud TPU*. https://cloud.google.com/tpu/docs/inception-v3-advanced

[4] *Microsoft COCO*. https://cocodataset.org/#home

[5] *Transformers*. https://huggingface.co/transformers/