EEE443 Project Report - Image Captioning

Okuyan A., Erdemoglu E., Donmez E., Eserol R., Topaloglu T., Akcin B.

***Abstract*— in this project, we have implemented two different deep learning structures for a given image captioning task. We have used a dataset that was given to us by our instructors and we were responsible of the preprocessing. In terms of general framework, we have used a CNN architecture to obtain image features and then an RNN architecture to constuct a caption. We have used transfer learning with Inception v3 to obtain the features and used a standard RNN structure with LSTM cells and a visual attention network for the second model, in which both will be discussed in the upcoming sections. We have tracked the batch losses and training losses after each epoch and used Bleu scores to elaborate on the validation accuracy.**

I. INTRODUCTION

In this project, we have performed image captioning, which can be described as, getting basic descriptions from images by extracting features from them, using neural network architectures. Image captioning is a hybrid Computer Vision and Natural Language Processing application that is useful for machines to learn from their surroundings. It is currently a field which is widely studied and has many articles that can be considered as a literature backbone for this project. In our case, we performed image captioning by firstly using a widely used state-of-the-art Convolutional Neural Network architecture given as Inception v3 for extracting features from the given image dataset, and then, we put the outputs from the Inception network to a specialized Recurrent Neural Networks for constructing captions for images. We used both Long Short-Term Memory Networks (LSTM) and Gated Recurrent Units (GRU) for this purpose. For image feature extraction, we used the Inception v3 model (built-in with Tensorflow), since it is a very deep and complex CNN architecture that is accurately able to extract important visual features, as seen by its introductory paper[1]. For captioning using these encodings by Inception v3 model, we followed a word embedding procedure by a simple fully-connected architecture and we put these embeddings to either an LSTM model or a GRU model, since both approaches seem to give similar performance generally [2]. For GRU, we also implemented a visual attention mechanism to improve captioning quality. From this methodology, we aim to get captions from the trained models that may not be on point with particular correct captions, but sensible enough to describe the respective images. As such, we present our methods and results for how we performed image captioning in this report.

As mentioned, there are many works and research done on the subject of image captioning, some of which can be considered similar to our project's work. Examples of articles explaining state-of-the-art image captioning techniques are far too abundant on the literature, so we have selected and explained a few articles that have overlapping tasks with our project. One exemplary work was done by Xu et al. in "Show, Attend and Tell: Neural Image Caption Generation with Visual Attention" [3], which describes how they achieved sensible image captioning by using LSTMs, Bahdanau attention mechanism (described by Bahdanau et al. [4]), convolutional network encoding by transfer learning and sentence difference metrics (primarily different n-gram BLEU scores, which are used in most image captioning works, described by Papineni et al. [5]). They present the mathematical background in their work, their training procedure, different datasets that they used (like Flickr and COCO), different models they used and their results by sentence difference metrics. In another paper called "Show and Tell: A Neural Image Caption Generator" by Vinyals et al. [6], they present a similar approach to image captioning, and in addition, they compare their results of different model results with a human's ability. They found that some of their models performed on par with a human, showing the success of their work. There is another paper written for the course CS231n at Stanford University by Rister et al. [7] called "Image Captioning with Attention", which shows a somewhat similar approach to our aforementioned proposed approach with LSTMs, transfer learning, attention and so on. It shows a detailed and visualized mathematical background and training procedures for all the tasks involved by using the ResNet-50 encoder model (described by He et al. in [8]), LSTMs, BLEU scores and the COCO dataset. It also shows some visualized results to present the success of the work done. Finally, there is the paper "Learning to Guide Decoding for Image Captioning" by Jiang et al. [9], which presents the most similar approach to our proposed methods by using the Inception-v3 encoder model, LSTMs and GRUs, BLEU sentence difference metrics (and some others like METEOR and CIDEr), similarly to our proposed approach and other previously mentioned approaches in this section. In this paper, there are also comparisons with the presented approach, as well as the approaches by Xu et al. [3], Vinyals et al. [6] and some other popular research done on this topic. This paper basically presents a decoding procedure for the popular Inception-v3 model for achieving sensible image captioning.From this review of the literature, we have proposed several methods and provided background information for the methods, models and metrics we have used in our task, which we will be integrating with our knowledge from the course and the dataset we have been provided with.

II. METHODS

Throughout this section, we will be analyzing the dataset we are given, the methods we have used to extract features

and generate captions. We will explain the chosen hyper-parameters and the optimization techniques. We will explain how we have manipulated the data to fit out needs.

A. Dataset

The data we were given contained 82783 image URLs taken from Flickr servers, their image ids, and for each image we were given approximately 5 captions. We have passed towards the URLs to obtain images and saved them to a directory. We have saved the images by their image indices which are given in the "imid" list of the data. While doing this, we have disguised our self as a browser by setting the user-agent parameter: "Mozilla/5.0". We have done this so that we are not treated as a robot. If we do not do this, some of the images are not allowed to download and we are getting the URL message "These are not the droids you are looking for." Instead of an actual image. While trying to download the images we still got some errors even though we got HTTP 200 OK message. We have captured the corrupted images by a try except block checking if the downloaded file is indeed an actual image.

Doing so, we have recognized that some of the images were corrupted and we have removed them. Also, we were not able to obtain some of the images, since Flickr tends to block the IP if it produces too much requests at a certain amount of time frame. As a result, we have worked with 73651 image samples. Also, we were given a dictionary for the words that is used in the captions which included 1004 words including the keywords to describe the beginning of the sequence (x_START_), the end (x_END_), the unknown words (x_UNK_) and the null (X_NULL_) words that come after the end. Hence, we have created a dictionary structure to map the integer representations of the words to the words themselves to easily convert the caption lists to readable caption strings.

Then, while working on the data, we have noticed that there were some irregular words that start with the letter "x" in the word dictionary given as "xWhile", "xFor", "xCatch" and "xCase" where they were for "while", "for", "catch" and "case". Hence, we have converted to these strings since we are using GLOVE in our project to obtain word embedding vectors.

B. LSTM Sentence Parser Network

This model is built as a baseline model for image classification task. For this model CNN Encoder was cut from average pooling of the 'mixed10' layer which outputs a vector of size 2048. The LSTM sentence parser also asks for the encodings with a input of the same size.

LSTM sentence parser constructs sentences with word by word basis. Therefore, the captions must be converted to n-grams in which the LSTM network can process sequence of words to predict the next word. Captions with respect to their encoded images by themselves are not compatible with word-by-word model therefore some subroutines are required to convert every caption to samples of n-grams.

Sequence generator does the following: For each available image encoding and its respective captions, the program constructs lists of time-series sequences using the captions. For each available captions, it removes the padding in the caption and create n-gram windows where n is from 0 to max length-1 of the caption without the zero-padding. The program extracts the generated n-grams and respective single word labels as lists.

The final dataset can be fit to RNN models. There are more than 2 million samples available for over 70000 images that this project has. A single sample has the following:

- Encoding of the image which the caption points to.
- Zero (pre)padded sequence which contains words.
- Label: The next word which should come after N-gram sequence.

For a caption of 6 words, the sequencer will generate 7 samples of data. Consider the example 'this is my neural network project' with image encoding named with 1x2048 vector named enc1. This will output the following list of sequences. Note that each bullet point is a sample and each colon divides encoding (e), sequence(s) and prediction(p) variables.

- e = enc1; s = [null,null,null,null,null,null,start-seq]; p=[this]
- e = enc1; s = [null,null,null,null,null,start-seq,this]; p=[is]
- e = enc1; s = [null,null,null,null,start-seq,this,is]; p=[my]
- e = enc1; s = [null,null,null,,start-seq,this,is,my]; p=[neural]
- e = enc1; s = [null,null,start-seq,this,is,my,neural]; p=[net]
- e = enc1; s = [null,start-seq,this,is,my,neural,net]; p=[proj]
- e = enc1; s =[start-seq,this,is,my,neural,net,proj]; p=[end-seq]

If there are any null characters left due to max-length, these will be left as pre-padded zeros. This process is done for every caption of every image.
Our LSTM model has the following structure:

- Input: Image Encoding, N-gram Sequence
- Output: Single word prediction

The image encoding is fed into a dense layer and dropout regularization is used. The encoding is reduced to 2048 to 512. 512 is selected as a convention, as the original dataset has the image encodings as a vector of length 512. N-gram input is fed into an embedding layer, which maps integer representations into dense vector representations. Layer, dropout regularization is used to prevent over fitting. This output is fed into one unidirectional then one bidirectional LSTM layer. Bidirectional connections are more advantageous while predicting future words given previous data, where unidirectional LSTM is advantageous in predicting single words from previous sequence inputs. The shape of an LSTM cell is provided with the implemented architecture.

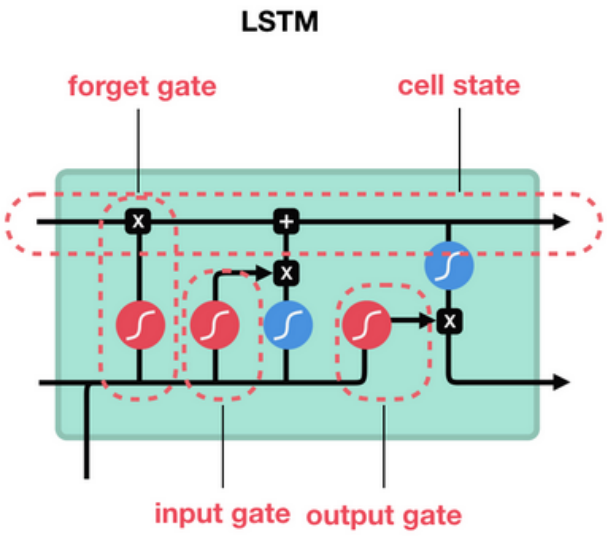


Fig. 1. LSTM Encoder Architecture

Model: "model"

Layer (type)	Output Shape	Param #	Connected to
input_2 (InputLayer)	[(None, 17)]	0	
embedding (Embedding)	(None, 17, 256)	257024	input_2[0][0]
input_1 (InputLayer)	[(None, 2048)]	0	
dropout_1 (Dropout)	(None, 17, 256)	0	embedding[0][0]
dropout (Dropout)	(None, 2048)	0	input_1[0][0]
bidirectional (Bidirectional)	(None, 17, 512)	1050624	dropout_1[0][0]
dense (Dense)	(None, 512)	1049088	dropout[0][0]
lstm_1 (LSTM)	(None, 512)	2099200	bidirectional[0][0]
add (Add)	(None, 512)	0	dense[0][0] lstm_1[0][0]
dense_1 (Dense)	(None, 256)	131328	add[0][0]
dense_2 (Dense)	(None, 1004)	258028	dense_1[0][0]

Total params: 4,845,292
Trainable params: 4,845,292
Non-trainable params: 0

Fig. 2. LSTM Encoder Architecture

The RNN decoder model is concatenated with the dense encoding of the images to construct the model. The model can be trained using model.fit method, provided the encoding, sequence and prediction lists (which are translated to numpy.ndarrays right before training) which is generated by caption sequencer.

This model is trained for 10 epochs. Training for less epochs cause introduce more xUNK parameters to be in the output predictions, where training for too many epochs cause generation of irrelevant captions with respect to the image. This is due to dense representation of the CNN image encodings. The next builds upon this model by using 8x8x2048 convolutional kernels as the image encodings to identify the scene better, therefore better predictions/captions can be generated.

1) Bidirectional vs. Unidirectional LSTM: In the project we have taken advantage of the idea of bidirectional LSTM while training, then in test decided the output by using it forward direction. The reason behind using both LSTMs is to improve the performance of our model. Bidirectional LSTM preserves information from both past and future and Unidirectional LSTM preserves information from past. Our algorithm uses bidirectional LSTM at the beginning in order to understand the context better because it performs more suitably on complicated tasks, leading to a better understanding in semantics. Bidirectional LSTM runs the inputs in two different methods, from past to future and from future to past by connecting two hidden layers of opposite directions to the same output. By taking inputs from past to future and from future to past, it analyzes the relation better. Additionally, in Bidirectional LSTM, inputs from past to future and from future to past are independent of each other which makes it easily applicable in terms of obtaining output. The reason behind that is Bidirectional LSTM is similar to Unidirectional LSTM which is iterated twice. Figure 2 shows the how both uni- and bidirectional LSTMs operate.

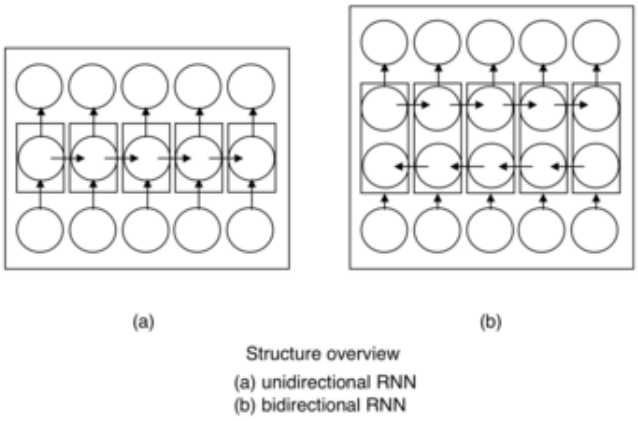


Fig. 3. LSTM Encoder Architecture

As it can be seen from Figure 2, output layer can get information from past to future and from future to past in Bidirectional LSTM, but output layer can only get information from past to future for Unidirectional LSTM. In the LSTM block we implemented uses Bidirectional LSTM in order to improve the performance of the model by analyzing information from past to future and from future to past. We have also used dropout to avoid overfitting.

C. Visual Attention Network

The second model that we have chosen to implement was the Visual Attention model. We have adopted this state-of-the-art architecture from the article “Show, Attend and Tell: Neural Image Caption Generation with Visual Attention ” by Xu et.al.[4]. The model as an overall architecture consists of two parts: A CNN Encoder and an RNN Decoder. Here, the reason we call this an Attention Network is that it uses an Attention Layer before the RNN in order to focus on relevant regions of the features while training.

1) CNN Feature Extraction: Leading up to the project, we knew that we were going to use a CNN structure to obtain the image features at a more abstract level, however, we have

considered two options in either training the CNN with the images or using the architecture with the weights pretrained on ImageNet. We have decided to go for the latter on since it would take a huge amount of time to train the CNN from scratch.

As the structure, we have chosen to use Google's Inception v3, which is a well-established network created for ILSVCR 2013. Before extracting features, we have used a input pipeline to automatically resize the image to (229,229,3) and run the normalization needed. For this process, we have used Keras with Tensorflow backend which contains both the network with the option to use pretrained weights and the preprocessing functions built-in.

Since we are extracting abstract features, we have used only up to a certain point of the architecture. For this network architecture, we needed to use a descriptive feature set, hence we have used the conv layer of mixed10 as the output layer, which gave features with (1,8,8,2048). We have extracted features for each of the images and saved them to a directory as numpy binaries. By doing this, we were able to retrieve them one-by-one to load them in batches. After building the data that we are going to use, we have created two data structures, one for the all the separate captions and one for their corresponding image ids. One important note is that, when loading the features for the Attention model, we reshape the feature to (1,64,2048).

2) CNN Encoder: CNN Encoder, since we use extracted features, consists of a fully connected layer that drops the dimension to the embedding dimension, which is the number of units in the word embedding layer which we will be discussing in the upcoming section. The architecture is given as follows.

```
Model: "cnn_encoder_14"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_44 (Dense)             multiple                  409800
=================================================================
Total params: 409,800
Trainable params: 409,800
Non-trainable params: 0
_________________________________________________________________
```

Fig. 4. CNN Encoder Architecture

3) RNN Decoder: The RNN Decoder consists of an attention, an embedding layer, and a recurrent layer. Here, we have chosen the GRU (Gated Recurrent Unit) as our choice of recurrent layer due to its computational efficiency. All of these layers and their working principles will be explained in their respective sections. The purpose of this section is to teach the algorithm the ability to generate the next word on an n-gram of words that represents parts of the sentence. The architecture of the decoder network is given below.

4) Attention Layer: Attention layer, being the reason behind the name of the architecture, is used to make the data generated from the caption more effective. The attention uses hidden state vectors to learn the mappings between the regions in the input sequence and the output sequence. In Xu et.al., we see two different options of attention being presented. In our project we have chosen to implement Deterministic (Soft) Attention, which is also called the Bahdanau Attention, first proposed by Bahdanau et.al.[5] in the paper "Neural Machine Translation by Jointly Learning to Align and Translate". The focus of this type of attention is to avoid resampling the attention location each time by simply taking the expected value of the context vector $\hat{z}_i$ which is as follows.

```
Model: "rnn_decoder_4"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_4 (Embedding)      multiple                  200800
_________________________________________________________________
gru_1 (GRU)                  multiple                  1403904
_________________________________________________________________
dense_10 (Dense)             multiple                  262656
_________________________________________________________________
dense_11 (Dense)             multiple                  515052
_________________________________________________________________
bahdanau_attention_1 (Bahdan multiple                  366081
=================================================================
Total params: 2,748,493
Trainable params: 2,748,493
Non-trainable params: 0
_________________________________________________________________
```

Fig. 5. RNN Decoder Architecture for Visual Attention Network

$$\mathbb{E}_{p(s_t|a)}[\hat{z}_t] = \sum_{i=1}^{L} \alpha_{t,i} a_i \tag{1}$$

In an attention layer, there exists attention weights, regarded as $\alpha_{t,i}$ which holds a value mapping corresponding to the probability of relevance of feature's regions in the given time. Practically, we input the extracted features and the hidden state, which gets updated in the recurrent network, and embed them to two separate fully-connected layers with number of units equal to the embedding dimension which is chosen to be 256 in this architecture. Then, the passed outputs are summed and passed through a tanh activation to extract relevancy information. Then this outputs is passed through a softmax activation which when multiplied with the feature and summed (which corresponds to taking expected value of the context vector with α probabilities). This corresponds to feeding a soft α weighted context vector[4] and makes it smooth and differentiable. Hence, making it easy to learn with standard backpropagation. An explanatory image that explains this flow is given below.

5) Gated Recurrent Unit (GRU): Gated Recurrent Units (GRUs), are other recurrent neural network structures with differently functioning gates, but similar to LSTM cells. A GRU architecture has the ability of understanding longer-term dependencies than standard RNN models because of this gated structure. This structure has two different gates, the update gate and the reset gate. The structure of one GRU cell is given.

This figure shows the structure of one GRU cell, along with the equations that is used in updating the inputs. The leftmost sigmoid activation is part of the reset gate, which

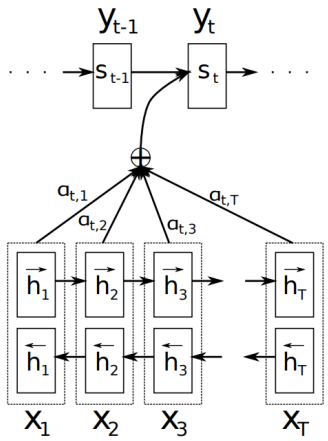


Fig. 6. Bahdanau Attention[6]

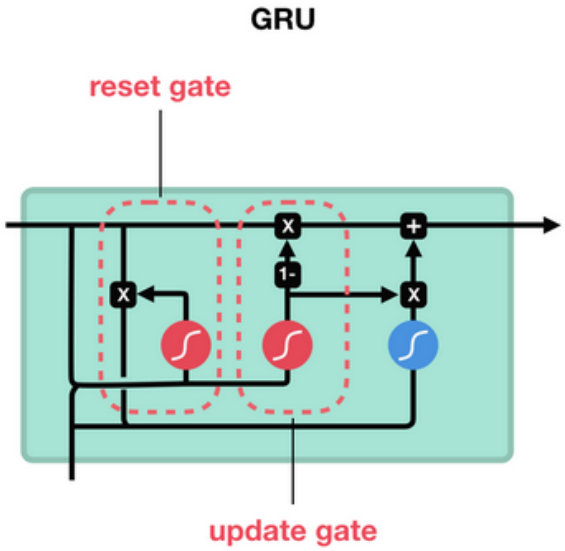


Fig. 7. GRU Cell Structure

performs the second operation in the right side for deciding how much of the past information to be forgotten (entries close to 0 are discarded because of the sigmoid function and the multiplication operation following it) [6]. The middle sigmoid cell is part of the update gate, which performs the first operation for learning the past information and passing this information to the "future". The other two operations are done at the rightmost section for memory content and passing this memory to the next cell. These operations also take the update gate's output as input. The operations are presented in mathematical form.Furthermore, we have chosen to use 512 hidden units inside the attention for the fully connected layers.

$$f_t = sigmoid([h_{t-1}, x_t]) \tag{2}$$

$$z_t = f_t * h_{t-1} \tag{3}$$

$$v_t = tanh([x_t, z_t]) * f_t \tag{4}$$

$$h_t = [h_{t-1} * (1 - f_t\ )] + v_t \tag{5}$$

6) Loss Function: In order to train the network, we are using categorical cross entropy (Sparse_Categorical_CrossEntropy in Keras) as in below.

$$Loss_{CE}(y_i, \hat{y_i}) = \sum_{i=1}^{C} I(y_i = 1) log(\hat{y_i}) \tag{6}$$

However, since we are running an RNN structure with teacher forcing, we modify the loss function in order to mask the losses that come from the contribution of the zero components in the caption as follows.

$$mask = !(y_i = 0) \tag{7}$$

$$loss = Loss_{CE}(y_i, \hat{y_i}) \odot mask \tag{8}$$

Where, mask is formed by using logical operations and $\odot$ represents the element-wise product.

7) Training: In order to train this network, we have worked with bare-bone Tensorflow 2.0 for its flexibility since the decoder uses a more complex and intricate training scheme. Encoder is a standard fully-connected network, hence we easily train it using backpropagation, however In the forward pass of the decoder we input the features and the hidden layer to the network and then pass them through the attention layer described above to obtain the context vector. Then, we used a self-trained embedding layer to draw word embedding vector from the context vector, which is concatenated with the context vector and then passed through the GRU to update its states. The output of the GRU is then passed through two fully connected layers to obtain the results with the size of vocabulary.

In the training loop, we use a method called "Teacher Forcing" to train the decoder. Starting from the first word, we obtain the loss by calculating the loss and expanding the sequence caption one word each time, meaning the algorithm uses the output of the previous time step as input to the current time step instead of using the output of the previous time step. This way, we force the algorithm to learn each time step according to the true caption, which increases stability and expedites the convergence.

Hence, in each training step, we forward pass the encoder, use teacher forcing to train the decoder and sum the loss over each time step. Then we extract the trainable variables for each of the networks in order and use tf.GradientTape.gradients function to calculate the gradient with respect to the weights and then update using optimizer.apply_gradients(), for which we use Adam optimizer.

After preprocessing the data as described in the Dataset section, we have used a training-validation two-way split on the data, on which we have decided on a 90-10% split for all captions. As a result, we have split 356052 captions and their corresponding images into two parts where they contained 320446-35606 captions each. Since we have run mini-batch gradient descent, we have chose the batch size as a integer

divisible to the number of training samples, hence chose it as 487.

D. RNN with Visual Attention (GLOVE)

This network is almost the same with the one described in the previous section, however with one major difference. In the other network, we have worked with a self-trained embedding layer that gives word embedding vectors for with dimensions (1,256). For this one, we have chosen to implement the Stanford NLP's pretrained GLOVE (Global Vectors for Word Representation) vectors for the initial embedding layer weights. We have moved to this approach with the hopes that it would enhance the performance by expediting the convergence since the vectors are pretrained on a general corpus. We have used the Glove6b where the corpus contained 6 billion words and 400 thousand words. From the ones there, we have applied the one which gives vectors in dimensions (1,200). We used that one since it was close to our previous embedding space which was 256. Hence, we have retrieved the vector representations for the words that were in our dictionary (1000 words) from the GLOVE vectors.

III. RESULTS

We have tested all of the networks that are mentioned above and discussed their results throughout this section.

A. LSTM Based Decoder Outputs

The baseline model proved to be somewhat successful at generating captions. Given the image encodings only have access to dense mapping of convolutional layers, which retain lower amount of information, rather than the convolutions themselves, the predictions of the RNN decoders were satisfactory. Very low or very high number of epochs tend to disturb the meaning of generated captions. After 10 epochs it is observed that predictions were somewhat acceptable, however the decoder frequently used the phrase 'standing on the table'. This might occurred because there could be too many standing and table words in the training dataset, or the encodings were not sufficient enough to provide variability in the image encodings. Nonetheless, having convolutional layers ('mixed10') as the encoding for the images with the attention mechanism provides with better predictions most of the time.

Due to being the first model, the BLEU scores were not calculated for this model as this feature was implemented later in the project iterations.


```
Prediction: x_START_ a train is x_UNK_ over a train station x_END_
original captions
Caption  1  textual:  x_START_ an old fashioned train traveling next to a stone
wall x_END_
Caption  2  textual:  x_START_ an old steam x_UNK_ x_UNK_ and about to x_UNK_ t
hrough an x_UNK_ x_END_
Caption  3  textual:  x_START_ a railroad train driving down the railroad track
s x_END_
Caption  4  textual:  x_START_ steam engine passing underneath an x_UNK_ on tra
cks x_END_
Caption  5  textual:  x_START_ an old train is coming down the tracks x_END_
```

Fig. 8. This prediction is very plausible as the output does not look like ground truths and it is in line with what image contains. There exists one unknown term, which could be a word like 'heading'. Heading is relatively a unique word that may not frequently found in the captions corpus. Given the dataset had already done preprocessing for us, this word may have been removed from frequently used words dictionary. For this particular example, LSTM model worked perfectly.

```
Prediction: x_START_ a pizza with a pizza full of food x_END_
original captions
Caption  1  textual:  x_START_ this is a x_UNK_ of x_UNK_ with x_UNK_ x_END_
Caption  2  textual:  x_START_ some x_UNK_ breakfast items and cup on a table x
_END_
Caption  3  textual:  x_START_ a close up of a plate of pastries and a bowl of
fruit x_END_
Caption  4  textual:  x_START_ a meal x_UNK_ of x_UNK_ pastry x_UNK_ and fruit
x_END_
```

Fig. 9. This was a particularly challenging scene. The decoder misidentified all objects in the scene, but was able to distinguish the scenery. It correctly found out that there is food in the picture, but mistook these food as pizza. If the encoding had more data to work with, more objects could be detected by the decoder so that the prediction would be better. This applies for all images, their encodings and predicted captions. LSTM decoder mostly detects most prominent object in the image encoding and parse a sentence around that object. Given that the 1x2048 encoding used here is taken from linear combination of convolutional kernels, the information retained in these convolutional kernels are greatly summarized. Using this encoding, it is very hard for the decoder to detect other objects which are less visible in the images. Due to this reason it is harder for the network to construct good captions for these images.

```
Prediction: x_START_ a group of people standing on a table with x_UNK_ x_END_
original captions
Caption  1  textual:  x_START_ several young boys are standing on top of a ramp
with skateboards x_END_
Caption  2  textual:  x_START_ a group of young men standing on top of a skateb
oard ramp x_END_
Caption  3  textual:  x_START_ some boys with skateboards getting ready to go d
own x_UNK_ x_END_
Caption  4  textual:  x_START_ a group of x_UNK_ prepares to skate around x_END
_
Caption  5  textual:  x_START_ a group of kids wearing x_UNK_ and holding skate
boards x_END_
```

Fig. 10. This prediction is also very plausible as it was able to understand that there is a group of people who are standing. It mistook people who are on skateboard ramp and assumed they are on a table. Given the shape of the ramp and the length of the encoding vector, it did a good job on captioning this image.

B. Visual Attention Networks

We have trained the network with and without GLOVE embeddings for 40 epochs. The training cross entropy errors over epochs are given for the training and validation data for the GLOVE and non-GLOVE networks in Figures 10 and 11.

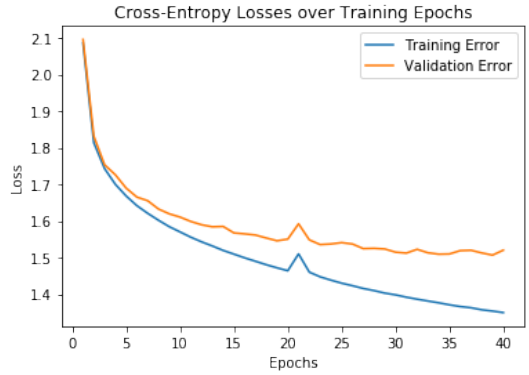


Fig. 11. Cross Entropy Losses over Training Epochs for Training and Validation with GLOVE Network

By looking at the graph, we can say that the algorithm converged at 40 epochs and the validation error fall down to 1.6 while the training error reached 1.38 for GLOVE network. However, the non-GLOVE network reached 1.25 in training and 1.35 in validation. We can see the effect of teacher forcing through the figures since there is stable convergence through the minimum at almost every step. The reason we observe a spike on the 21st epoch is that we have re-run the training sequence to continue after 20 epochs, we should

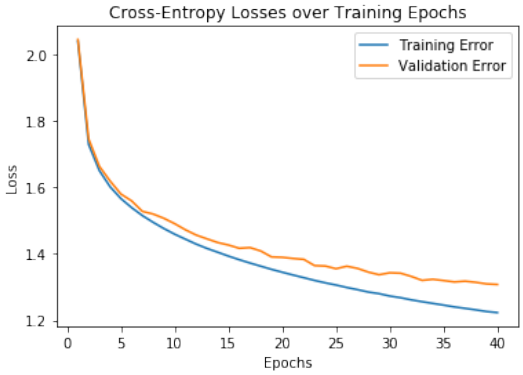


Fig. 12. Cross Entropy Losses over Training Epochs for Training and Validation with non-GLOVE Network

have retrained from scratch, however, it would take too long since one epoch lasts around 440 seconds (6.5 minutes). Below, we present ten validation samples that this algorithm generated with their sentence BLEU scores referenced with respect to all of the captions of that image for the GLOVE and non-GLOVE networks respectively.

```
------------------------------
Sample ID 25220
```


```
Captions:
a man is eating hot dogs at a restaurant
a man sitting at a table with hot dogs on a bun
a man holding a drink at a hot dog eating x_UNK_
a man seated on a table eating bread and a cup of coffee
a man with a plate of hot dogs drinking water

BLEU Score 0.28997844147152074

Prediction:a man holding a x_UNK_ x_UNK_ x_UNK_ x_UNK_ in a restaurant
------------------------------
```

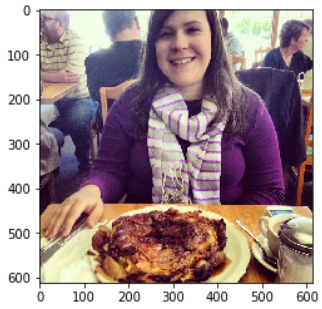
Fig. 13. Here, we observe a sample where the algorithm detects some attention regions, however was not able to determine what those regions converge into. This is probably due to the small dictionary that we have. This is a common case that we see where once the sequence gets disrupted by the unknown, it continues with that up to some point.

```
BLEU Score 7.208781414766327e-155

Prediction:a man in a x_UNK_ x_UNK_ eats a slice of pizza
```

Fig. 14. Here, we see that the algorithm performed worse although the model was better in terms of error. We think that there are too much pizza in the network that the food variants overfitted to network when trained with 40 epochs.

```
------------------------------
Sample ID 53234
```



```
Captions:
a woman in x_UNK_ sitting at a table with plate of x_UNK_
woman in purple shirt and x_UNK_ eating in a restaurant
a woman prepares to eat a large apple x_UNK_
a girl wearing a purple x_UNK_ and x_UNK_ in a restaurant

BLEU Score 0.4497332084013506

Prediction:a woman is sitting at a table with a large sandwich
```

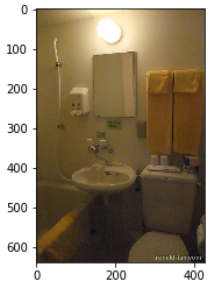
Fig. 15. Here, we observe a nearly perfect sample in terms of semantics. The woman in the image is identified clearly and the action of sitting is well-defined, however, the object was mistaken. The real captions are interested in the color of the shirt the woman is wearing however, out algorithm ignored that and instead focused on the sitting action instead of eating.

```
BLEU Score 0.4854917717073234

Prediction:a man sitting at a table with a pizza
```

Fig. 16. Here, we observe similar behavior we observed in the previous sample however, BLEU score is better. This is due to the word large in the previous caption lowering the score although being reasonable.

```
------------------------------
Sample ID 31856
```



```
Captions:
a white toilet sitting next to a white sink
a bathroom with a toilet mirror and sink
a bathroom is shown with yellow towels on display
small bathroom with tub sink and toilet in close x_UNK_
a bathroom with a white sink and yellow towels above a white toilet

BLEU Score 0.4671379777282001

Prediction:a bathroom with a sink and a toilet
------------------------------
```

Fig. 17. Here, we observe a good caption where the network was able to identify the main components of the image, being the sink and toilet in a bathroom setting. While some of the captions focused on color, again our generation ignored the color information completely.

```
BLEU Score 0.5623413251903491

Prediction:a bathroom with a toilet and a toilet
```

Fig. 18. Here, we see that the sink is removed from the caption and replaced with toilet, which was an unsuccessful attempt.

```
------------------------------
Sample ID 41662
```

```
Captions:
a man with a yellow surfboard in an ocean
a man riding a wave on top of a surfboard
a man x_UNK_ a wave on a surf board
a gray x_UNK_ man in a x_UNK_ takes a big wave
a man in a wet suit riding a surfboard on a wave

BLEU Score 0.5969491792019646

Prediction:a man in a x_UNK_ surfing on a wave
------------------------------
```

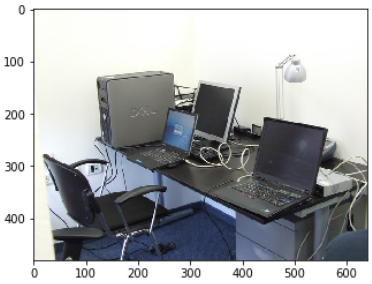
Fig. 19. In this caption, the algorithm was able to identify the man in the image, understandable since all of the captions contain the word “man” in the same region of the sentence. One very interesting point is that none of these captions were contained “surfing” as a word and our algorithm was able to correctly undertand the action being done can also be called “surfing” instead of “riding a surfboard”. The BLUE score hence was also high.

```
BLEU Score 1.0

Prediction:a man riding a wave on top of a surfboard
```

Fig. 20. Here, we see a BLEU score of 1, which is highly surprising since the generated caption is one-to-one with the original caption, which was pretty impressing.

```
------------------------------
Sample ID 69532
```



```
Captions:
the computer x_UNK_ is set up with three laptop computers
two lap x_UNK_ and a desktop computer sitting on an office desk
a desk with two laptops and a desktop computer on it
a computer x_UNK_ with x_UNK_ laptop computers and a desktop computer
three laptops and a desktop computer on a desk

BLEU Score 0.5972880661844315

Prediction:a desk with a laptop and a desktop computer on a desk
```

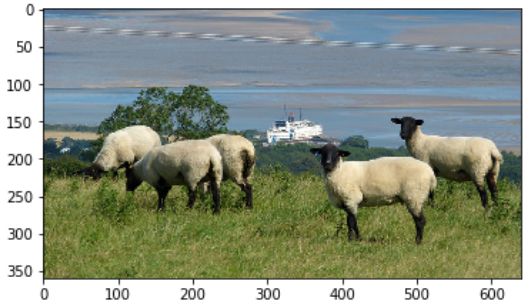
Fig. 21. Here, we observe a caption that is nearly perfect. In the figure, we observe a a desk with a laptop and a desktop computer on it. The only part the caption was problematic was repeating the desk twice. This is due to the fact that the algorithm doesn't have the ability to recognize if one information is mentioned earlier in the sequence. Divided to two sentences, "a desk with a laptop and a desktop computer" and "a laptop and a desktop computer on a desk" are both grammatically correct.

```
BLEU Score 0.392814650900513

Prediction:a desk with a laptop cellphone and a desktop computer
```

Fig. 22. The captions are pretty similar, the repetition seems to be gone, however, the laptop is written as laptop cellphone, which is inaccruate.

```
------------------------------
Sample ID 32964
```



```
Captions:
five sheep are grazing on a hill in the green grass
animals eating grass in a hill by the ocean
a herd of sheep grazing on a hill next to the ocean
five sheep on a hill eating grass overlooking the ocean
a field with sheep overlooking a x_UNK_ ship

BLEU Score 0.9036020036098449

Prediction:a herd of sheep grazing on a field
```

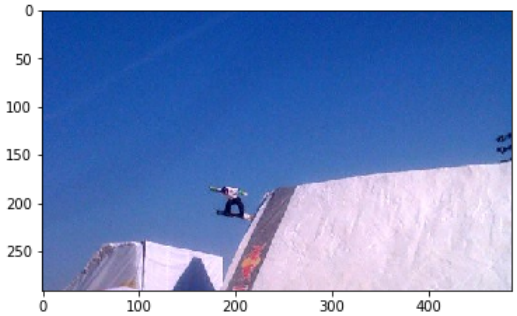
Fig. 23. This caption was one of the best that we have encountered with a BLEU score of 0.9. We believe the reason behind it is that both the sheep and the grass are pretty distinctive and the extracted features were highly informative. The algorithm was able to extract the information that the setting was a field correctly although it was not mentioned in any of the original captions.

```
BLEU Score 0.4032989116748133

Prediction:a herd of sheep are standing in the grass
```

Fig. 24. The BLEU score dropped significantly when the field turned into grass, however, this is still a reasonable option which also exists on the original captions.

```
------------------------------
Sample ID 59067
```



```
Captions:
a man flying through the air xWhile riding a snowboard
a skier is taking a x_UNK_ down a x_UNK_ man made x_UNK_
a man is x_UNK_ down the x_UNK_ hill with his skateboard
snowboarder jumping down a x_UNK_ snow hill
snow board rider x_UNK_ after x_UNK_ off ramp

BLEU Score 1.1231644147255971e-154

Prediction:a man on a snowboard in the air
```

Fig. 25. In this caption, we observe a BLEU score close to zero, which would normally be a horrible result, however, when we read the caption, we see that the caption generated is actually meaningful since the image represents a man on a snowboard in the air indeed. Here, we observe that BLEU score is a more restrictive type of metric than we first anticipated and only checks similarity and not quality of the captions. This is also plausible since the BLEU score is originally introduced in order to detect similarity in translations.

```
BLEU Score 8.15394372921771e-155

Prediction:a person flying a kite in the air
```

Fig. 26. This time the non-GLOVE network made a really bad mistake and somehow saw a kite in the image. We believe that the algorithm observed the man as a kite.

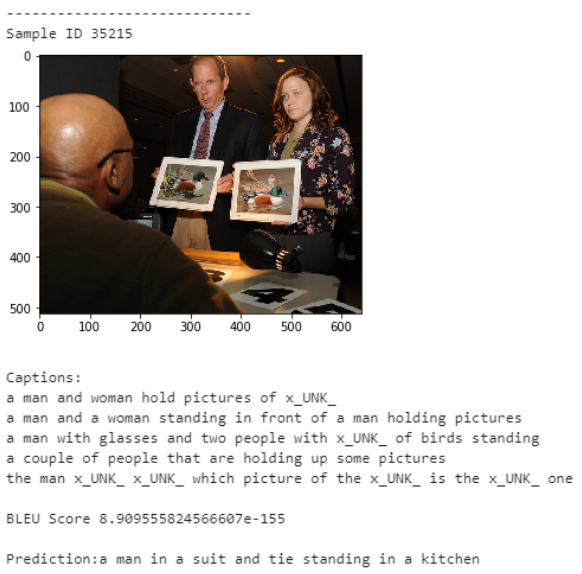


Fig. 27. In this image, we automatically observe that the caption is wrong since the setting is far from kitchen and the BLEU score is nearly zero, however, the algorithm managed to capture the man with the tie. We presume that the network mistook the papers with numbers in front of the jury on the table as plates and decided the setting is closed to a kitchen. This is a common pattern of error that we have observed. On an image with the window's reflection on the sky, the network mistook the reflection as a kite due to the rectangular shape. We have classified these types of errors under shape errors.

Fig. 28. This caption is completely irrelevant except the man part, which is the easiest word to guess.

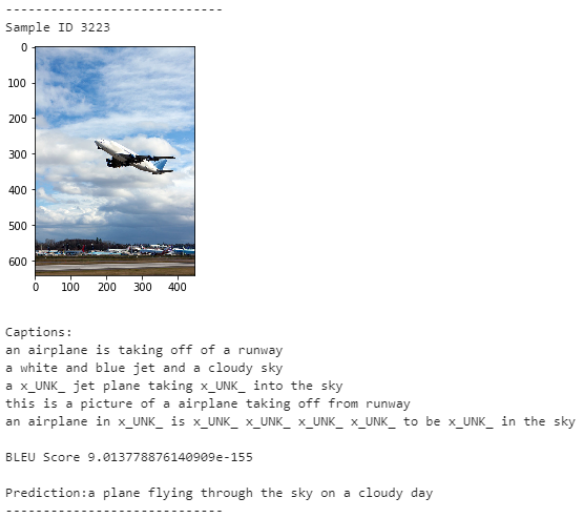


Fig. 29. Here, we observe another caption with a near-zero BLEU score. The captions say that this is a plane going off from the highway. Although our caption did not mention a takeoff, it was quite intelligently able to define the flying plane on a cloudy day, which was something even the original captions did not. From these examples, we observe that the algorithm we trained is more dependent on defining a setting for the picture, sometimes disregarding some action or color information for the setting information. Although the score was low, the caption was impressive.

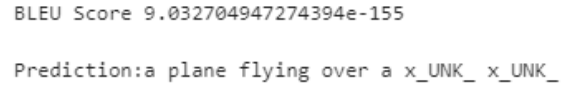


Fig. 30. Here, although the algorithm saw the plane and the action of flight, it was not able to produce a meaningful explanation as we saw in the GLOVE version of this.

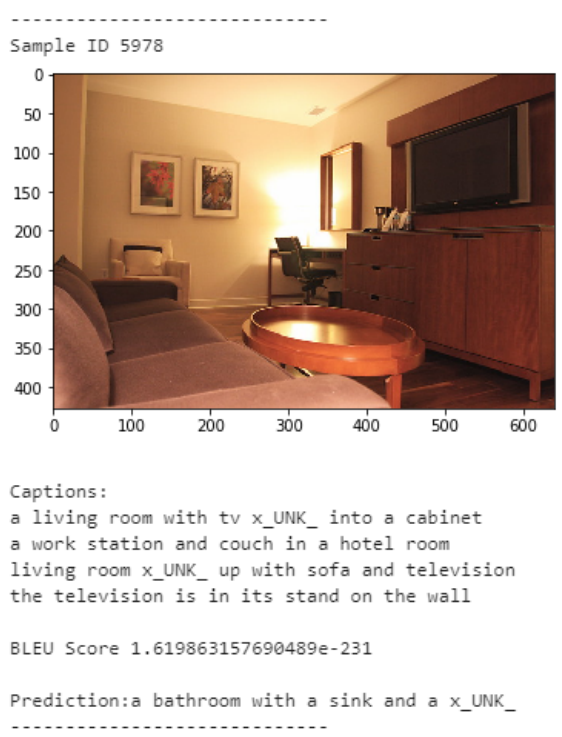


Fig. 31. This time, we see a completely irrelevant caption from the images and the captions altogether. Although the setting of the image was a living room, our decided that it was a bathroom with a sink and an unknown. However, we believe that there is a sound reason behind such deception. In nearly all of the bathroom instances that we trained on, there was a mirror and a sink in the bathroom. When we observe this picture, we see that there is a mirror on the wall with a table in front of it. We believe the algorithm mistook this combination of objects as a mirror and a sink given the dim lighting in the photo.

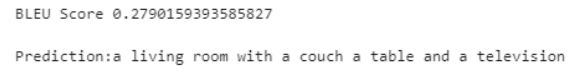


Fig. 32. This image was very interesting, since although the algorithm performed worse in every validation sample, it did significantly better on this one, correctly able to identify all three significant objects in the image in couch, table and television.

IV. DISCUSSION

The project consisted of two main parts, which constitute the overall frame of a image captioning algorithm. The first part, being the encoder, was relatively straightforward since we were able to extract features using transfer learning from a pretrained Inception v3 model. We have used two different layers as outputs to extract different features for different decoders. We were not able to optimize this part of the network since training the deep architecture would cost us a massive amount of time. However, one thing we can say for sure is that the information acquired from the conv layer of the network was substantially better when compared with the one taken from the average pool.

Throughout the project, we have worked on three different decoder architectures. First one was a base-standard LSTM networ. With this architecture, we were able to observe how Recurrent Neural Networks work in general and how LSTM states are updated through epochs, we have learned to use "teacher forcing", which we believe was a substantial improvement when compared with the standard RNN teaching algorithm where we feed the network the output of the previous time instance.

Hence, after realizing that we wouldn't be successful with a basic LSTM structure, we have started building a visual attention model and decided that we would try to optimize the algorithm by using pretrained GLOVE word embeddings.We have used Bahdanau attention in both of the networks. For the non-GLOVE network, we have used an embedding of size 256 which we chose the 200 sized version for the GLOVE network. In the beginning, we thought that the pretrained embeddings would change the performance greatly since the words are mapped better. However, when we trained on both we observed that the GLOVE network worked much better than the non-GLOVE network although the cross entropy errors suggested the opposite. While GLOVE network reached 1.6 in cross entropy error, the non-GLOVE network reached 1.35, and we were surprised once this was the case, however, when we observed the validation samples we saw that the GLOVE network was a lot more precise and descriptive than the non-GLOVE network. Overall, we believed that the visual attention would work better than the standard LSTM model and GLOVE to be better than non-GLOVE. Both of our hypotheses were true.

REFERENCES

[1] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, "Rethinking the Inception Architecture for Computer Vision," 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), pp. 2818–2826, 2016.

[2] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, "Rethinking the Inception Architecture for Computer Vision" 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), pp. 2818–2826, 2016.

[3] J. Chung, C. Gulcehre, K. H. Cho, and Y. Bengio,"NIPS 2014 Deep Learning and Representation Learning Workshop, Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling", Dec. 2014.

[4] K. Xu, J. Ba, R. Kiros, K. Cho, A. Courville, R. Salakhutdinov, R. Zemel, and Y. Bengio,"Show, Attend and Tell: Neural Image Caption Generation with Visual Attention," arXiv.org, 19-Apr-2016. [Online]. Available: https://arxiv.org/abs/1502.03044v3. [Accessed: 12-Jan-2020].

[5] D. Bahdanau, K. Cho, and Y. Bengio, "Neural Machine Translation by Jointly Learning to Align and Translate," arXiv.org, 19-May-2016. [Online]. Available: https://arxiv.org/abs/1409.0473. [Accessed: 12-Jan-2020].

[6] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu, " Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics," BLEU: a Method for Automatic Evaluation of Machine Translation, pp. 311–318, Jul. 2002.

[7] O. Vinyals, A. Toshev, S. Bengio, and D. Erhan, "Show and Tell: A Neural Image Caption Generator", arXiv.org, 20-Apr-2015. [Online]. Available: https://arxiv.org/abs/1411.4555. [Accessed: 12-Jan-2020].

[8] B. Rister and D. Lawson, "Image Captioning with Attention," CS231n: Convolutional Neural Networks for Visual Recognition, 23-Mar-2016. [Online]. Available: http://cs231n.stanford.edu/reports/2016/pdfs/362_Report.pdf. [Accessed: 12-Jan-2020].

[9] K. He, X. Zhang, S. Ren, and J. Sun, "Deep Residual Learning for Image Recognition," arXiv.org, 10-Dec-2015. [Online]. Available: https://arxiv.org/abs/1512.03385. [Accessed: 12-Jan-2020].

[10] W. Jiang, L. Ma, X. Chen, H. Zhang, and W. Liu, "Learning to Guide Decoding for Image Captioning," arXiv.org, 03-Apr-2018. [Online]. Available: https://arxiv.org/abs/1804.00887. [Accessed: 12-Jan-2020].

[11] G. Loye, "Attention Mechanism," FloydHub Blog, 17-Sep-2019. [Online]. Available: https://blog.floydhub.com/attention-mechanism/. [Accessed: 12-Jan-2020].

[12] G. Drakos, "What is a Recurrent NNs and Gated Recurrent Unit (GRUS)," Medium, 05-Feb-2019. [Online]. Available: https://medium.com/@george.drakos62/what-is-a-recurrent-nns-and-gated-recurrent-unit-grus-ea71d2a05a69. [Accessed: 12-Jan-2020].

[13] H. Lamba, "Image Captioning with Keras-'Teaching Computers to describe pictures,"' Medium, 17-Feb-2019. [Online]. Available: https://towardsdatascience.com/image-captioning-with-keras-teaching-computers-to-describe-pictures-c88a46a311b8. [Accessed: 12-Jan-2020].

[14] "Image captioning with visual attention : TensorFlow Core," TensorFlow. [Online]. Available: https://www.tensorflow.org/tutorials/text/image_captioning. [Accessed: 12-Jan-2020].

[15] M. Foord, "HOWTO Fetch Internet Resources Using The urllib Package," Python HOWTOs. [Online]. Available: https://docs.python.org/3/howto/urllib2.html. [Accessed: 12-Jan-2020].

[16] Y. Katariya, "Image Captioning using InceptionV3 and Beam Search," 13-Jul-2017. [Online]. Available: https://yashk2810.github.io/Image-Captioning-using-InceptionV3-and-Beam-Search/. [Accessed: 12-Jan-2020].

APPENDIX:

Code:

```
#################### IMAGE DOWNLOADER ####################

import os
# See the directories construct directories if needed:
root_dir = os.getcwd()
imgs_dir = root_dir + '\\images'
print(root_dir)

if not os.path.exists(imgs_dir):
    os.mkdir(imgs_dir)

import h5py
from tqdm import tqdm
import requests
from PIL import Image


def eee443_dataset_read(path):
    f = h5py.File(path + '\\eee443_project_dataset_train.h5', 'r')
    train_cap = f['train_cap']
    train_imid = f['train_imid']
    train_url = f['train_url']
    word_code = f['word_code']
    train_ims = None
    print('Size of URL list: ', train_url.shape[0])
    return train_imid, train_cap, train_url, word_code


def download_images(img_dir, train_url):
    os.chdir(img_dir)
    header = {'User-Agent': 'Mozilla/5.0'}  # fool the website to download easily
    print('Current working directory set as: ', os.getcwd())

    corrupt_idx = []
    for item in tqdm(range(1, len(train_url) + 1)):
        url = train_url[item - 1].decode('utf-8')

        # This option is more stable
        r = requests.get(url, allow_redirects=True, stream=True)

        # If this item exists, do not redo operation just pass to next item.
        if os.path.exists(img_dir + '/' + str(item)):
            continue

        # Log why certain images are gone
        # if not r.status_code == 200:
        # print('Image: ', item, ' Code: ', r.status_code)

        # When HTTP 200 is achieved write file: - Corrupted file stuck here
        if r.status_code == 200:
            with open(img_dir + '/' + str(item), 'wb') as f:
                for chunk in r.iter_content(1024):
                    f.write(chunk)

            # Validate image is correct:
            try:
                img = Image.open(img_dir + '/' + str(item))  # open the image file
```

```
                img.verify()  # verify that it is, in fact an image
            except (IOError, SyntaxError) as e:
                print(e)  # Print error encountered
                corrupt_idx.append(img_dir + '/' + str(item) + '  --- non image')
                # Might need to remove the image

        # if item % 500 == 0:
        # print('At image ' + str(item) + '. Continuing download.')

    os.chdir('..')
    print('Current working directory set as: ', os.getcwd())

    # At the end of output  the process report which indices are removed
    with open('removedIms.txt', 'w') as file_handle:
        for items in corrupt_idx:
            file_handle.write('%s\n' % items)
    return

# If already downloaded, do not attempt to re-download.
if not os.path.exists(imgs_dir):
    imid, cap, url, words = eee443_dataset_read(root_dir)
    download_images(imgs_dir,url)

##################### INCEPTION ENCODER ####################
# DEPENDING ON ATTENTION VS BASELINE CHANGE FINAL LAYER TO 'mixed10'
# VS 'avg_pool' respectively.

import os
#os.environ["CUDA_VISIBLE_DEVICES"]="-1" # Disable GPU
# See the directories construct directories if needed:
root_dir = os.getcwd()
imgs_dir = root_dir + '\\images'
exports_dir = root_dir + '\\exports'

print(root_dir)

if not os.path.exists(exports_dir):
    os.mkdir(exports_dir)

import tensorflow as tf
from tensorflow.keras.applications.inception_v3 import InceptionV3, preprocess_input
from tensorflow.keras.preprocessing import image
from tensorflow.python.keras import backend as K

def inception_load_image(path):
    img = image.load_img(path, target_size=(299, 299, 3))
    imgar = image.img_to_array(img)
    imgar = np.expand_dims(imgar, axis=0)
    imgar = preprocess_input(imgar)
    return imgar

def inception_transfer_model():
    tf.keras.backend.clear_session() # clears previous session if this code is run multiple
time
    # This is necessary, as re-running these segments may stack up models

    inception_model = InceptionV3(include_top=True, weights='imagenet',
input_shape=(299,299,3))
    inception_model.trainable = False

    # Check layers via inception_model.summary()
```

```
    #inception_model.summary()

    inception_tx_layer = inception_model.get_layer('avg_pool') # mixed10 is the final layer
with notop layout.

    new_input = inception_model.input
    x = inception_tx_layer.output

    inception_tx_model = tf.keras.Model(outputs=x, inputs=new_input) # directly make a model
from it.
    #inception_tx_model.summary()

    inception_img_size = K.int_shape(inception_tx_model.input)[1:3]
    print('Image size: ', inception_img_size)

    inception_tx_values_size = K.int_shape(inception_tx_layer.output)
    print('Vector size of transfer values: ', inception_tx_values_size)
    return inception_tx_model

def inception_encode_image(image_dir, img_id,model):
    image = inception_load_image(image_dir+img_id)
    image = image.reshape((1, image.shape[1], image.shape[2], image.shape[3]))
    image = preprocess_input(image)
    encoding = model.predict(image)
    #encoding = np.reshape(encoding, encoding.shape[1])
    return encoding

inception_tx_model = inception_transfer_model()

from tqdm import tqdm
import numpy as np

os.chdir(imgs_dir)  # change to training directory
inception_v3_exp_dir = exports_dir + '\\inception_v3_encodings'

if not os.path.exists(inception_v3_exp_dir):
    os.mkdir(inception_v3_exp_dir)

for img in tqdm(os.listdir()):
    if os.path.exists(inception_v3_exp_dir + '\\' + img + '.npy'):
        continue
    else:
        tmp = inception_encode_image(os.getcwd() + '\\', img, inception_tx_model)
        np.save(inception_v3_exp_dir + '\\' + img + '.npy', tmp)

#################### LSTM DECODER ####################
import tensorflow as tf # tensorflow gpu capabilities available
import os
ROOT_DIR = os.getcwd()
EXP_DIR = os.path.join(ROOT_DIR, 'exports')

import h5py
import pickle
import numpy as np


def eee443_dataset_read(path):
    f = h5py.File(path + '\\eee443_project_dataset_train.h5', 'r')
    train_cap = f['train_cap']
    train_imid = f['train_imid']
    train_url = f['train_url']
```

```
    word_code = f['word_code']
    train_ims = None
    return train_imid, train_cap, train_url, word_code, f


def extract_word_dict(dataset):
    words_struct = dataset.get('word_code')[()]
    word_list = list(words_struct.dtype.names)  # returns the key
    w_all_idx = []
    for word in word_list:
        w_idx = int(words_struct[word])
        w_all_idx.append(w_idx)

    word_dict = dict(zip(w_all_idx, word_list))
    return word_dict


def unpickle_data(path, filename):
    file = pickle.load(open(path + filename, 'rb'), encoding='utf8')
    return file

# Dataset reading: - Export word_dict too.
_, _, _, _, f = eee443_dataset_read(ROOT_DIR) # from previous notebook.
word_dict = extract_word_dict(f)

# Print if everything is alright or not.
print('Size of words dictionary: ', len(word_dict.keys()))
print(word_dict[627])

import numpy as np


# Check the text data for images:
# Extracting data from dataset:
def fetch_captions(image_id, dataset, word_dict):
    # Query this on train_imid to extract which indices hold the captions for this image:
    imid = np.array(dataset.get('train_imid')[()])
    indices = np.where(imid == int(image_id))[0]  # since everything is string, must be cast to
int manually.
    # indices is a tuple of array

    # Extract the list of integer captions for the given image
    all_caps = np.array(dataset.get('train_cap')[()])
    # print('Overall shape of the captions: ', all_caps.shape)

    count = 1
    caps = []
    for idx in indices:
        cap = all_caps[idx][:]
        # print('Caption ', str(count), ': ', str(cap))
        caps.append(cap)
        count += 1

    # print('')
    # print('Captions: ', str(caps))  # Final look at the captions

    # Now do conversion:
    text_cap = []
    count = 0
    for item in caps:
        temp = []
```

```
        count += 1
        for word in item:
            # print(item)
            # print(str(word)  , (word_dict[word]))
            temp.append(word_dict[word])
        text_cap.append(temp)
        # print('Caption ', count, ' textual: ', (' '.join(map(str,
temp))).split('x_NULL_')[0]) # list comprehension
        # temp.remove()

    # print(type(text_cap))
    # Return captions
    caps = [list(c) for c in caps]  # list comprehension to make arrays list
    return indices, text_cap, caps


# Helper function to print captions of an image:
def print_captions(caps, word_dict):
    # Only for displaying
    # print('Indices of captions for this image:', test_id)
    # print('\n')

    i = 0
    text_cap = []
    for item in caps:
        i += 1
        temp = []
        # print('Caption ', i, ': ',  item)
        for word in item:
            # print(item)
            # print(str(word)  , (word_dict[word]))
            temp.append(word_dict[word])
        text_cap.append(temp)
        # x_NULL_ strings are only ignored, not erased from the captions
        print('Caption ', i, ' textual: ', (' '.join(map(str, temp))).split('x_NULL_')[0])
    return

indices, text_cap, caps = fetch_captions(10, f, word_dict)

# Validation using image file: '10.jpg'
# Printing the image itself or its encoded output doesn't matter.
print('Indices: ', indices)
print('\n')

print('Type of captions: ', type(caps))
print('Length of captions list: ', len(caps))
print('Type of one of the tokenized captions: ', type(caps[0]))
print('\n')

for c in text_cap:
    print(c)
    print(type(c))

# Inception Data Unpickling:
enc_inception = unpickle_data(EXP_DIR, '\\enc_inception.pkl')
enc_inception_idx = unpickle_data(EXP_DIR, '\\inception_enc_idx.pkl')

# Verify sizes and all sorts of stuff:
print(type(enc_inception))
print(np.shape(enc_inception))
enc_inception = np.squeeze(enc_inception)
```

```
print('Encoded images final shape: ', np.shape(enc_inception))
print('\n')

print(type(enc_inception_idx))
print(np.shape(enc_inception_idx))
enc_inception_idx = np.squeeze(enc_inception_idx)
print('Encoded image indices final shape: ', np.shape(enc_inception_idx))
print(len(enc_inception_idx[300:360]))
print('\n')

enc_inception_dict = dict(zip(enc_inception_idx, enc_inception))
print('Inception encoding dictionary with query 10: ', enc_inception_dict['10']) # 10 is the
10.jpg here.
print('Size of inception encoding dictionary with query 10: ', len(enc_inception_dict['10'])) #
10 is the 10.jpg here.

print(enc_inception_dict['5'])

import random
import math

def divide_into_two(percentage, list_to_divide):
    indices = random.sample(range(1, len(list_to_divide)), len(list_to_divide)-math.ceil((1-
percentage)*len(list_to_divide)))
    samples = []
    indices.sort(reverse=True) # reverse to do not alter original indices when removed
    for idx in indices:
            samples.append(list_to_divide[idx])
            list_to_divide.remove(list_to_divide[idx])
    # return back samples to original format
    samples.reverse()
    return samples, list_to_divide

TRAIN_PERCENTAGE = 0.8 # 80% train and validation data
key_list = list(enc_inception_dict.keys())

train, test = divide_into_two(TRAIN_PERCENTAGE, key_list)

print('Train size:', len(train))
print('Test size:',len(test))
print(train[100:105])

from tensorflow.keras.utils import to_categorical
from tqdm import tqdm


def img_cap_sequence_generator(img_name, h5file, word_dict, enc_dict, max_length=17):
    enc, seq, prd = list(), list(), list()

    _, _, captions = fetch_captions(img_name, h5file, word_dict)

    for idx in range(0, len(captions)):
        enc_temp, seq_temp, prd_temp = caption_sequencer(img_name, enc_dict, captions[idx])
        enc.append(enc_temp)
        seq.append(seq_temp)
        prd.append(prd_temp)

    return enc, seq, prd


def caption_sequencer(img_name, enc_dict, cap_idx, max_length=17):
```

```
    encoding, sequence, prediction = list(), list(), list()

    temp = []
    count = 0
    for i in range(0, max_length - 2):
        encoding.append(enc_dict[img_name])
        temp.append(cap_idx[:i + 1])

        captions.append(temp)
        if cap_idx[i + 1] == 0:  # np other labeling required
            temp.pop()
            encoding.pop()
            break
        prediction.append(cap_idx[i + 1])
    sequence.append(temp)

    # For each sequence input must be normalized to max_length pre padding
    for seq in sequence:
        for item in seq:
            while len(item) < max_length:
                item.insert(0, 0)
            item = np.array(item)
            item = np.squeeze(sequence)

    encoding = np.array(encoding)

    sequence = np.array(sequence)
    sequence = np.squeeze(sequence)

    prediction = np.array(prediction)
    return encoding, sequence, prediction


def stack_lists(enc, caps, prd):
    return np.vstack(enc), np.vstack(caps), np.hstack(prd)


def list_sequence_gen(ims, f, word_dict, enc_inception_dict, classes=1004):
    e, s, p = list(), list(), list()
    for item in tqdm(ims):
        enc, seq, prd = img_cap_sequence_generator(item, f, word_dict, enc_inception_dict)
        enc, seq, prd = stack_lists(enc, seq, prd)
        e.append(enc)
        s.append(seq)
        p.append(prd)
    e, s, p = stack_lists(e, s, p)
    p = to_categorical(p, num_classes=classes)
    return e, s, p

_, _, captions = fetch_captions('10', f, word_dict)
print('cap ln ', len(captions[0]))
print('cap ', (captions[0]))
print('\n')

enc,caps,preds = caption_sequencer('10', enc_inception_dict, captions[0])

print('enc: ', (enc).shape)
print('caps ', type(caps[0]))
print('caps ', (caps[6]))
print('caps ', (caps).shape)
print('preds: ', (preds))
```

```
enc, seq, prd = img_cap_sequence_generator('10', f, word_dict, enc_inception_dict)

#enc = np.array(enc)
enc,seq,prd = stack_lists(enc, seq, prd)

print(type(enc))
print(len(enc))
print((enc).shape)
#print((enc))
print('\n')

print(type(seq))
print(len(seq))
print((seq).shape)
#print((seq))
print('\n')

print(type(prd))
print(len(prd))
print((prd).shape)
#print((prd))
print('\n')

# Test Case:
print((enc[0]))
print((enc[5]))
print((seq[5]))
print((prd[5]))

from sys import getsizeof

ims = train
print(len(ims))

e,s,p = list_sequence_gen(ims, f, word_dict, enc_inception_dict)

print(e.shape)
print(s.shape)
print(p.shape)

print(e[1].shape)

E_DIR = os.path.join(EXP_DIR, 'e')
S_DIR = os.path.join(EXP_DIR, 's')
P_DIR = os.path.join(EXP_DIR, 'p')

if not os.path.exists(E_DIR):
    os.mkdir(E_DIR)

if not os.path.exists(S_DIR):
    os.mkdir(S_DIR)

if not os.path.exists(P_DIR):
    os.mkdir(P_DIR)

from tqdm import tqdm

for i in tqdm(range(len(e))):
    np.save(E_DIR + '\\' + str(i) + '.npy', e[i])
    np.save(S_DIR + '\\' + str(i) + '.npy', s[i])
```

```
    np.save(P_DIR + '\\' + str(i) + '.npy', p[i])

# np.save(EXP_DIR + '\\' 'e.npy', e)
# np.save(EXP_DIR + '\\' 's.npy', s)
# np.save(EXP_DIR + '\\' 'p.npy', p)

import numpy as np
e = np.load(os.path.join(EXP_DIR,'e.npy'))
s = np.load(os.path.join(EXP_DIR,'s.npy'))
p = np.load(os.path.join(EXP_DIR,'p.npy'))

train_dataset = tf.data.Dataset.from_tensor_slices(([e,s], p))

BATCH_SIZE = 64
SHUFFLE_BUFFER_SIZE = 100

train_dataset = train_dataset.shuffle(SHUFFLE_BUFFER_SIZE).batch(BATCH_SIZE)

from tensorflow.keras.layers import Input, Dense, Dropout, LSTM, Embedding, GRU, add,
Bidirectional


def lstm_block(enc_size, vocab_size, max_length, unit_size, drop_rate):  # max_length: max
length of captions
    # feature extractor model - Already calculated reduce size.
    tin = Input(shape=(enc_size,))
    f1 = Dropout(drop_rate)(tin)
    f2 = Dense(unit_size * 2, activation='relu')(f1)

    # sequence model - this part is for the captions:
    cin = Input(shape=(max_length,))
    s1 = Embedding(vocab_size, unit_size, mask_zero=True)(cin)
    s2 = Dropout(drop_rate)(s1)
    s3 = Bidirectional(LSTM(unit_size, return_sequences=True))(s2)
    s4 = (LSTM(unit_size * 2))(s3)
    # decoder model
    decoder1 = add([f2, s4])
    decoder2 = Dense(unit_size, activation='relu')(decoder1)
    outputs = Dense(vocab_size, activation='softmax')(decoder2)

    # tie it together [image, seq] [word]
    model = tf.keras.Model(inputs=[tin, cin], outputs=outputs)
    model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])

    # summarize model
    # print(model.summary())
    return model


def caption_generator(imgname, enc_dict, word_dict, model, max_length=17):
    START = 1
    END = 2

    enc_in = enc_dict[imgname]
    # print(type(enc_in))
    # print(enc_in.shape)
    # print(enc_in)

    in_text = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1]
    in_text = np.array(in_text)
```

```
    # print(type((in_text)))

    # print(type(in_text))
    # print(len(in_text))
    # print(in_text.shape)

    for i in range(max_length - 1):
        # print(i)
        pred = model.predict(x=[enc_in.reshape((1, 2048)), in_text.reshape((1, 17))],
verbose=0)
        pred = np.argmax(pred)

        in_text = np.delete(in_text, 0)
        in_text = np.append(in_text, pred)
        # print(in_text)

        if in_text[-1] == END:
            break
    return in_text

import math
tf.keras.backend.clear_session()  # For easy reset of notebook state.
model = lstm_block(2048, 1004, 17, 256, 0.3)
#model.fit([e[:5000], s[:5000]], p[:5000], epochs=10, verbose=1)
model.fit([e[:(e)], s[:(e)]], p[:e], epochs=5,
          verbose=1, validation_split=0.1)

from PIL import Image
def show_image(path, image):
    name = os.path.join(path, image)
    im = Image.open(name)
    display(im)
    return


ims = ['8335', '9474', '50342', '20584', '999', '22196', '37463', '6445', '123']

IMS_DIR = os.path.join(ROOT_DIR, 'images')

for ix in ims:
    in_text = caption_generator(ix, enc_inception_dict, word_dict, model)
    textual = ''
    for word in in_text:
        if word_dict[word] != 'x_NULL_':
            textual += word_dict[word] + ' '

    show_image(IMS_DIR, ix)
    print('Prediction:', textual)

    print('original captions')
    indices, text_cap, caps = fetch_captions(int(ix), f, word_dict)
    print_captions(caps, word_dict)
    # for c in text_cap:
    # print(c)
    # print(type(c))

indices, text_cap, caps = fetch_captions(10, f, word_dict)

# Validation using image file: '10.jpg'
# Printing the image itself or its encoded output doesn't matter.
print('Indices: ', indices)
```

```
print('\n')

print('Type of captions: ', type(caps))
print('Length of captions list: ', len(caps))
print('Type of one of the tokenized captions: ', type(caps[0]))
print('\n')

for c in text_cap:
    print(c)
    print(type(c))

model.summary()

#################### INCEPTION ENCODER FOR VISUAL ATTENTION ####################
import os
os.environ["CUDA_VISIBLE_DEVICES"]="-1" # Disable GPU
# See the directories construct directories if needed:
root_dir = os.getcwd()
imgs_dir = root_dir + '\\images'
exports_dir = root_dir + '\\exports'

print(root_dir)

if not os.path.exists(exports_dir):
    os.mkdir(exports_dir)

    import tensorflow as tf
    from tensorflow.keras.applications.inception_v3 import InceptionV3, preprocess_input
    from tensorflow.keras.preprocessing import image
    from tensorflow.python.keras import backend as K


    def inception_load_image(path):
        img = image.load_img(path, target_size=(299, 299, 3))
        imgar = image.img_to_array(img)
        imgar = np.expand_dims(imgar, axis=0)
        imgar = preprocess_input(imgar)
        return imgar


    def inception_transfer_model():
        tf.keras.backend.clear_session()  # clears previous session if this code is run
multiple time
        # This is necessary, as re-running these segments may stack up models

        inception_model = InceptionV3(include_top=True, weights='imagenet', input_shape=(299,
299, 3))
        inception_model.trainable = False

        # Check layers via inception_model.summary()
        # inception_model.summary()

        inception_tx_layer = inception_model.get_layer('mixed10')  # mixed10 is the final layer
with notop layout.

        new_input = inception_model.input
        x = inception_tx_layer.output

        inception_tx_model = tf.keras.Model(outputs=x, inputs=new_input)  # directly make a
model from it.
        # inception_tx_model.summary()
```

```
        inception_img_size = K.int_shape(inception_tx_model.input)[1:3]
        print('Image size: ', inception_img_size)

        inception_tx_values_size = K.int_shape(inception_tx_layer.output)
        print('Vector size of transfer values: ', inception_tx_values_size)
        return inception_tx_model


    def inception_encode_image(image_dir, img_id, model):
        image = inception_load_image(os.path.join(image_dir, img_id))
        image = image.reshape((1, image.shape[1], image.shape[2], image.shape[3]))
        image = preprocess_input(image)
        encoding = model.predict(image)
        # encoding = np.reshape(encoding, encoding.shape[1])
        return encoding

inception_tx_model = inception_transfer_model()

from tqdm import tqdm
import numpy as np

inception_v3_exp_dir = os.path.join(root_dir,'attention_embed')

if not os.path.exists(inception_v3_exp_dir):
    os.mkdir(inception_v3_exp_dir)

for img in tqdm(os.listdir(imgs_dir)):
    tmp = inception_encode_image(os.path.join(root_dir, "images"), img, inception_tx_model)
    np.save(inception_v3_exp_dir + '\\' + img +'.npy', tmp)

#################### ATTENTION V2 ####################
import pickle
import numpy as np
import os
import tensorflow as tf
import time
import h5py
import matplotlib.pyplot as plt
import PIL
from tqdm import tqdm
from nltk.translate.bleu_score import sentence_bleu
from sklearn.model_selection import train_test_split
from sklearn.utils import shuffle

#Load indices and the itoword dictionary
word_dict = open("word_dict.pkl", "rb")
word_dict = pickle.load(word_dict)

print(len(word_dict))
print(word_dict[1])
print(word_dict[2])

ROOT_DIR = os.getcwd()
print('ROOT:', ROOT_DIR)
EMBED_DIR = os.path.join(ROOT_DIR, 'attention_embed')
print('embeddings:', EMBED_DIR)
CAPTION_DIR = os.path.join(ROOT_DIR, 'attention_captions')
print('captions:', CAPTION_DIR)
IMAGE_DIR = os.path.join(ROOT_DIR, 'images')
print('images:', IMAGE_DIR)
```

```
MODEL_DIR = os.path.join(ROOT_DIR, 'attention_models')
print('models:', MODEL_DIR)

sampleList = os.listdir(EMBED_DIR)
print(len(sampleList))
print(sampleList[0:10])

validImageList = [(int(s.replace('.npy', ''))) for s in sampleList]
print(validImageList[0:10])

trial = np.load(os.path.join(EMBED_DIR, sampleList[0]))
print('Shape of one embedding:', trial.shape)

f = h5py.File(ROOT_DIR + '\\eee443_project_dataset_train.h5', 'r')
print(f.keys())
trainImid = f['train_imid'].value
print(trainImid)

trainCap = f['train_cap'].value
print(trainCap[0])

#extract captions for existing images
def retreive_capt_id(image_id ,imid_list, cap_list):
    capList = []
    idList = []
    caps = np.where(imid_list == image_id)[0]
    caps = cap_list[caps]
    for cap in caps:
        capList.append(cap)
        idList.append(image_id)
    return capList, idList

def retreive_capt_batch(image_id_list ,imid_list, cap_list):
    idList = []
    captionsList = []
    for image_id in image_id_list:
        caps, ids = retreive_capt_id(image_id,imid_list,cap_list)
        captionsList.extend(caps)
        idList.extend(ids)
    return np.asarray(captionsList), np.asarray(idList)

captionsList, captionIDList = retreive_capt_batch(validImageList,trainImid, trainCap)
print(captionsList.shape)
print(captionIDList.shape)

print(captionsList[0:15,:])
print(captionIDList[0:15])

if not os.path.exists(CAPTION_DIR):
    os.mkdir(CAPTION_DIR)
    np.save(os.path.join(CAPTION_DIR, 'captionsAttention.npy'), captionsList)
    np.save(os.path.join(CAPTION_DIR, 'captionsIDAttention.npy'), captionIDList)

#Load cached caption-id matched lists
captionList = np.load(os.path.join(CAPTION_DIR,'captionsAttention.npy'))
captionIDList = np.load(os.path.join(CAPTION_DIR,'captionsIDAttention.npy'))
print(captionList.shape, captionIDList.shape)

#split train and test
img_name_train, img_name_val, cap_train, cap_val = train_test_split(captionIDList,
captionList,test_size=0.1, random_state=0)
```

```
print('Train ImIDs and Captions', img_name_train.shape, cap_train.shape)
print('Val ImIDs and Captions', img_name_val.shape, cap_val.shape)

#Hyperparameters
BATCH_SIZE = 487
#for shuffling
BUFFER_SIZE = 1000
#using glove6b/200
embedding_dim = 256
#unit num for attention
units = 512

vocab_size = len(word_dict)
batch_num = int(len(img_name_train)/BATCH_SIZE)

dataset = tf.data.Dataset.from_tensor_slices((img_name_train, cap_train))

def map_func(img_name, cap):
    #load feature
    embed_tensor = np.load(os.path.join(EMBED_DIR, str(img_name)+'.npy'))
    #reshape the 1x8x8x2048 tensor to 64x2048
    n,x,y,z = embed_tensor.shape
    embed_tensor = embed_tensor.reshape((x*y,z))
    return embed_tensor, cap

dataset = dataset.map(lambda item1, item2: tf.numpy_function(
          map_func, [item1, item2], [tf.float32, tf.int32]),
          num_parallel_calls=tf.data.experimental.AUTOTUNE)

dataset = dataset.shuffle(BUFFER_SIZE).batch(BATCH_SIZE)
dataset = dataset.prefetch(buffer_size=tf.data.experimental.AUTOTUNE)


class SoftAttention(tf.keras.Model):
    def __init__(self, fcDim):
        super(SoftAttention, self).__init__()
        self.featLayer = tf.keras.layers.Dense(fcDim)
        self.hidLayer = tf.keras.layers.Dense(fcDim)
        self.sumLayer = tf.keras.layers.Dense(1)

    # explained in report
    def call(self, features, hidden):
        hidden_exp = tf.expand_dims(hidden, 1)

        sumLayers = self.featLayer(features) + self.hidLayer(hidden_exp)
        relevanceScore = tf.nn.tanh(sumLayers)

        attWeights = tf.nn.softmax(self.sumLayer(relevanceScore), axis=1)
        # ottend over features by estimating expectation
        contextVec = attWeights * features
        contextVec = tf.reduce_sum(contextVec, axis=1)

        return contextVec, attWeights

class Encoder(tf.keras.Model):
    def __init__(self, embedding_dim):
        super(Encoder, self).__init__()
        self.outLayer = tf.keras.layers.Dense(embedding_dim)
    def call(self, x):
        x = self.outLayer(x)
        x = tf.nn.relu(x)
```

```
        return x

class Decoder(tf.keras.Model):
    def __init__(self, embedding_dim, units, vocab_size):
        super(Decoder, self).__init__()
        self.units = units
        self.attention = SoftAttention(self.units)
        #use glove vectors as weights
        self.embedding = tf.keras.layers.Embedding(vocab_size, embedding_dim)
        self.gru = tf.keras.layers.GRU(self.units, recurrent_initializer='glorot_uniform',
return_sequences=True, return_state=True)
        self.fcHid = tf.keras.layers.Dense(self.units)
        self.fcOut = tf.keras.layers.Dense(vocab_size)


    def call(self, x, features, hidden):
        contextVec, attWeights = self.attention(features, hidden)
        # extract embedding
        x = self.embedding(x)
        temp = tf.expand_dims(contextVec, 1)
        #concat the enbedding with context vector
        x = tf.concat([temp, x], axis=-1)
        # pass the concatenated vector gru
        output, state = self.gru(x)
        x = self.fcHid(output)
        x = tf.reshape(x, (-1, x.shape[2]))
        x = self.fcOut(x)
        return x, state, attWeights

    def reset_state(self, batch_size):
        return tf.zeros((batch_size, self.units))

encoder = Encoder(embedding_dim)
decoder = Decoder(embedding_dim, units, vocab_size)

lossFcn = tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True)

def customLoss(real, pred):
    lossVal = lossFcn(real, pred)
    mask = tf.cast(tf.math.logical_not(tf.math.equal(real, 0)),  dtype=lossVal.dtype)
    lossVal *= mask
    return tf.reduce_mean(lossVal)


lossList = []
optim = tf.keras.optimizers.Adam()


@tf.function
def trainingStep(img_embed, caption):
    loss = 0
    # reset hidden state
    hidden = decoder.reset_state(batch_size=caption.shape[0])

    # In our data, this corresponds to x_START_
    START_INDEX = 1
    dec_input = tf.expand_dims([START_INDEX] * caption.shape[0], 1)

    with tf.GradientTape() as tape:
        # pass through encoder
        features = encoder(img_embed)
```

```
        # train rnn with teacher forcing (recursive)
        for i in range(1, caption.shape[1]):
            # pass through decoder
            pred, hidden, _ = decoder(dec_input, features, hidden)
            loss += customLoss(caption[:, i], pred)
            dec_input = tf.expand_dims(caption[:, i], 1)

    total_loss = (loss / int(caption.shape[1]))
    trainable_variables = encoder.trainable_variables + decoder.trainable_variables
    gradients = tape.gradient(loss, trainable_variables)
    optim.apply_gradients(zip(gradients, trainable_variables))

    return loss, total_loss

EPOCHS = 40

for epoch in range(EPOCHS):
    start_time = time.time()
    total_loss = 0

    for (batch, (embed_tensor, caption)) in enumerate(dataset):
        running_loss, total_loss_t = trainingStep(embed_tensor, caption)
        total_loss += total_loss_t

        if batch % 100 == 0:
            print ('Epoch {} Batch {} Loss {:.8f}'.format( epoch + 1, batch,
running_loss.numpy() / int(caption.shape[1])))
    lossList.append(total_loss / batch_num)
    dur = time.time() - start_time
    #report epoch err and time
    print ('Epoch {} CE Error {:.8f}'.format(epoch + 1, total_loss/batch_num))
    print ('Duration {} seconds\n'.format(dur))


def print_pred(imgID):
    print('-----------------------------\nSample ID', imgID)
    img = PIL.Image.open(os.path.join(IMAGE_DIR, str(imgID)))
    plt.imshow(img)
    plt.show()

    print('\nCaptions:')
    captions = retreive_capt_id(imgID, trainImid, trainCap)[0]
    captionsStr = []
    for caption in captions:
        capStr = ''
        cap = []
        for word in caption:
            if (word != 2 and word != 1 and word != 0):
                capStr += word_dict[word] + ' '
                cap.append(word_dict[word])
        print(capStr)
        captionsStr.append(cap)

    embed = np.load(os.path.join(EMBED_DIR, str(imgID) + '.npy'))
    pred = evaluate(embed)
    predLst = []
    capStr = '\nPrediction:'
    for word in pred:
        if (word != 2 and word != 1 and word != 0):
            capStr += word_dict[word] + ' '
            predLst.append(word_dict[word])
```

```
    print('\nBLEU Score', sentence_bleu(captionsStr, predLst))
    print(capStr)


def evaluate(embedding):
    hidden = decoder.reset_state(batch_size=1)
    embedding = tf.reshape(embedding, (embedding.shape[0], -1, embedding.shape[3]))
    # pass through encoder
    features = encoder(embedding)

    START_INDEX = 1
    END_INDEX = 2
    MAX_LEN = 17
    dec_input = tf.expand_dims([START_INDEX], 0)
    result = []

    # don't use teacher forcing this time
    for i in range(MAX_LEN):
        preds, hidden, _ = decoder(dec_input, features, hidden)
        predictedWord = tf.argmax(preds, axis=1).numpy()
        result.append(predictedWord[0])
        if predictedWord == 2:
            return result
        dec_input = tf.expand_dims([predictedWord[0]], 0)

    return result

no_samples = 20
np.random.seed(22)
test_indices = np.random.permutation(len(img_name_val))[0:no_samples]
for ind in test_indices:
    print_pred(img_name_val[ind])

if not os.path.exists(MODEL_DIR):
    os.mkdir(MODEL_DIR)

encoder.save(os.path.join(MODEL_DIR, 'encoder_attention_glove'))
#decoder.save_weights(os.path.join(MODEL_DIR, 'decoder_attention_glove'), )

encoderlol = Encoder(embedding_dim)
encoderlol.build(input_shape=(64,2048))
print(encoderlol.summary())

print(decoder.summary())

##################### ATTENTION GLOVE ####################
import pickle
import numpy as np
import os
import tensorflow as tf
import time
import h5py
import matplotlib.pyplot as plt
import PIL
from tqdm import tqdm
from nltk.translate.bleu_score import sentence_bleu
from sklearn.model_selection import train_test_split
from sklearn.utils import shuffle

#Load indices and the itoword dictionary
```

```
word_dict = open("word_dict.pkl", "rb")
word_dict = pickle.load(word_dict)

print(len(word_dict))
print(word_dict[1])
print(word_dict[2])

ROOT_DIR = os.getcwd()
print('ROOT:', ROOT_DIR)
EMBED_DIR = os.path.join(ROOT_DIR, 'attention_embed')
print('embeddings:', EMBED_DIR)
CAPTION_DIR = os.path.join(ROOT_DIR, 'attention_captions')
print('captions:', CAPTION_DIR)
IMAGE_DIR = os.path.join(ROOT_DIR, 'images')
print('images:', IMAGE_DIR)
MODEL_DIR = os.path.join(ROOT_DIR, 'attention_models')
print('models:', MODEL_DIR)
GLOVE_DIR = os.path.join(ROOT_DIR, 'glove.6B.200d.txt')
print('glove:', GLOVE_DIR)

sampleList = os.listdir(EMBED_DIR)
print(len(sampleList))
print(sampleList[0:10])

validImageList = [(int(s.replace('.npy', ''))) for s in sampleList]
print(validImageList[0:10])

trial = np.load(os.path.join(EMBED_DIR, sampleList[0]))
print('Shape of one embedding:', trial.shape)

f = h5py.File(ROOT_DIR + '\\eee443_project_dataset_train.h5', 'r')
print(f.keys())
trainImid = f['train_imid'].value
print(trainImid)

trainCap = f['train_cap'].value
print(trainCap[0])

#extract captions for existing images
def retreive_capt_id(image_id ,imid_list, cap_list):
    capList = []
    idList = []
    caps = np.where(imid_list == image_id)[0]
    caps = cap_list[caps]
    for cap in caps:
        capList.append(cap)
        idList.append(image_id)
    return capList, idList

def retreive_capt_batch(image_id_list ,imid_list, cap_list):
    idList = []
    captionsList = []
    for image_id in image_id_list:
        caps, ids = retreive_capt_id(image_id,imid_list,cap_list)
        captionsList.extend(caps)
        idList.extend(ids)
    return np.asarray(captionsList), np.asarray(idList)

captionsList, captionIDList = retreive_capt_batch(validImageList,trainImid, trainCap)
print(captionsList.shape)
print(captionIDList.shape)
```

```
print(captionsList[0:15,:])
print(captionIDList[0:15])

if not os.path.exists(CAPTION_DIR):
    os.mkdir(CAPTION_DIR)
    np.save(os.path.join(CAPTION_DIR, 'captionsAttention.npy'), captionsList)
    np.save(os.path.join(CAPTION_DIR, 'captionsIDAttention.npy'), captionIDList)

#Load cached caption-id matched lists
captionList = np.load(os.path.join(CAPTION_DIR,'captionsAttention.npy'))
captionIDList = np.load(os.path.join(CAPTION_DIR,'captionsIDAttention.npy'))
print(captionList.shape, captionIDList.shape)

#split train and test
img_name_train, img_name_val, cap_train, cap_val = train_test_split(captionIDList,
captionList,test_size=0.1, random_state=0)
print('Train ImIDs and Captions', img_name_train.shape, cap_train.shape)
print('Val ImIDs and Captions', img_name_val.shape, cap_val.shape)

#Hyperparameters
BATCH_SIZE = 487
#for shuffling
BUFFER_SIZE = 1000
#using glove6b/200
embedding_dim = 200
#unit num for attention
units = 512

vocab_size = len(word_dict)
batch_num = int(len(img_name_train)/BATCH_SIZE)

dataset = tf.data.Dataset.from_tensor_slices((img_name_train, cap_train))

def map_func(img_name, cap):
    #load feature
    embed_tensor = np.load(os.path.join(EMBED_DIR, str(img_name)+'.npy'))
    #reshape the 1x8x8x2048 tensor to 64x2048
    n,x,y,z = embed_tensor.shape
    embed_tensor = embed_tensor.reshape((x*y,z))
    return embed_tensor, cap

dataset = dataset.map(lambda item1, item2: tf.numpy_function(
          map_func, [item1, item2], [tf.float32, tf.int32]),
          num_parallel_calls=tf.data.experimental.AUTOTUNE)

dataset = dataset.shuffle(BUFFER_SIZE).batch(BATCH_SIZE)
dataset = dataset.prefetch(buffer_size=tf.data.experimental.AUTOTUNE)

# Load Glove vectors
embeddings_index = {} # empty dictionary
f = open(GLOVE_DIR, encoding="utf-8")
for line in tqdm(f):
    values = line.split()
    word = values[0]
    coefs = np.asarray(values[1:], dtype='float32')
    embeddings_index[word] = coefs
f.close()

embedding_matrix = np.zeros((vocab_size, embedding_dim))
for i, word in word_dict.items():
```

```
    embedding_vector = embeddings_index.get(word)
    if embedding_vector is not None:
        embedding_matrix[i] = embedding_vector
    else:
        word_x = str.lower(word.replace('x',''))
        embedding_vector = embeddings_index.get(word_x)
        if embedding_vector is None:
            print(word_x)
            embedding_vector = np.zeros((1,embedding_dim))
        embedding_matrix[i] = embedding_vector

print(embedding_matrix.shape)
print(embedding_matrix)


class SoftAttention(tf.keras.Model):
    def __init__(self, fcDim):
        super(SoftAttention, self).__init__()
        self.featLayer = tf.keras.layers.Dense(fcDim)
        self.hidLayer = tf.keras.layers.Dense(fcDim)
        self.sumLayer = tf.keras.layers.Dense(1)

    # explained in report
    def call(self, features, hidden):
        hidden_exp = tf.expand_dims(hidden, 1)

        sumLayers = self.featLayer(features) + self.hidLayer(hidden_exp)
        relevanceScore = tf.nn.tanh(sumLayers)

        attWeights = tf.nn.softmax(self.sumLayer(relevanceScore), axis=1)
        # ottend over features by estimating expectation
        contextVec = attWeights * features
        contextVec = tf.reduce_sum(contextVec, axis=1)

        return contextVec, attWeights

class Encoder(tf.keras.Model):
    def __init__(self, embedding_dim):
        super(Encoder, self).__init__()
        self.outLayer = tf.keras.layers.Dense(embedding_dim)
    def call(self, x):
        x = self.outLayer(x)
        x = tf.nn.relu(x)
        return x

class Decoder(tf.keras.Model):
    def __init__(self, embedding_dim, units, vocab_size):
        super(Decoder, self).__init__()
        self.units = units
        self.attention = SoftAttention(self.units)
        #use glove vectors as weights
        self.embedding = tf.keras.layers.Embedding(vocab_size, embedding_dim,
weights=[embedding_matrix])
        self.gru = tf.keras.layers.GRU(self.units, recurrent_initializer='glorot_uniform',
return_sequences=True, return_state=True)
        self.fcHid = tf.keras.layers.Dense(self.units)
        self.fcOut = tf.keras.layers.Dense(vocab_size)


    def call(self, x, features, hidden):
        contextVec, attWeights = self.attention(features, hidden)
```

```
        # extract embedding
        x = self.embedding(x)
        temp = tf.expand_dims(contextVec, 1)
        #concat the enbedding with context vector
        x = tf.concat([temp, x], axis=-1)
        # pass the concatenated vector gru
        output, state = self.gru(x)
        x = self.fcHid(output)
        x = tf.reshape(x, (-1, x.shape[2]))
        x = self.fcOut(x)
        return x, state, attWeights

    def reset_state(self, batch_size):
        return tf.zeros((batch_size, self.units))

encoder = Encoder(embedding_dim)
decoder = Decoder(embedding_dim, units, vocab_size)

lossFcn = tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True)

def customLoss(real, pred):
    lossVal = lossFcn(real, pred)
    mask = tf.cast(tf.math.logical_not(tf.math.equal(real, 0)),  dtype=lossVal.dtype)
    lossVal *= mask
    return tf.reduce_mean(lossVal)


lossList = []
optim = tf.keras.optimizers.Adam()


@tf.function
def trainingStep(img_embed, caption):
    loss = 0
    # reset hidden state
    hidden = decoder.reset_state(batch_size=caption.shape[0])

    # In our data, this corresponds to x_START_
    START_INDEX = 1
    dec_input = tf.expand_dims([START_INDEX] * caption.shape[0], 1)

    with tf.GradientTape() as tape:
        # pass through encoder
        features = encoder(img_embed)
        # train rnn with teacher forcing (recursive)
        for i in range(1, caption.shape[1]):
            # pass through decoder
            pred, hidden, _ = decoder(dec_input, features, hidden)
            loss += customLoss(caption[:, i], pred)
            dec_input = tf.expand_dims(caption[:, i], 1)

    total_loss = (loss / int(caption.shape[1]))
    trainable_variables = encoder.trainable_variables + decoder.trainable_variables
    gradients = tape.gradient(loss, trainable_variables)
    optim.apply_gradients(zip(gradients, trainable_variables))

    return loss, total_loss

EPOCHS = 40

for epoch in range(EPOCHS):
```

```
    start_time = time.time()
    total_loss = 0

    for (batch, (embed_tensor, caption)) in enumerate(dataset):
        running_loss, total_loss_t = trainingStep(embed_tensor, caption)
        total_loss += total_loss_t

        if batch % 100 == 0:
            print ('Epoch {} Batch {} Loss {:.8f}'.format( epoch + 1, batch,
running_loss.numpy() / int(caption.shape[1])))
    lossList.append(total_loss / batch_num)
    dur = time.time() - start_time
    #report epoch err and time
    print ('Epoch {} CE Error {:.8f}'.format(epoch + 1, total_loss/batch_num))
    print ('Duration {} seconds\n'.format(dur))


def print_pred(imgID):
    print('----------------------------\nSample ID', imgID)
    img = PIL.Image.open(os.path.join(IMAGE_DIR, str(imgID)))
    plt.imshow(img)
    plt.show()

    print('\nCaptions:')
    captions = retreive_capt_id(imgID, trainImid, trainCap)[0]
    captionsStr = []
    for caption in captions:
        capStr = ''
        cap = []
        for word in caption:
            if (word != 2 and word != 1 and word != 0):
                capStr += word_dict[word] + ' '
                cap.append(word_dict[word])
        print(capStr)
        captionsStr.append(cap)

    embed = np.load(os.path.join(EMBED_DIR, str(imgID) + '.npy'))
    pred = evaluate(embed)
    predLst = []
    capStr = '\nPrediction:'
    for word in pred:
        if (word != 2 and word != 1 and word != 0):
            capStr += word_dict[word] + ' '
            predLst.append(word_dict[word])

    print('\nBLEU Score', sentence_bleu(captionsStr, predLst))
    print(capStr)

def evaluate(embedding):
    hidden = decoder.reset_state(batch_size=1)
    embedding = tf.reshape(embedding, (embedding.shape[0], -1, embedding.shape[3]))
    # pass through encoder
    features = encoder(embedding)

    START_INDEX = 1
    END_INDEX = 2
    MAX_LEN = 17
    dec_input = tf.expand_dims([START_INDEX], 0)
    result = []

    # don't use teacher forcing this time
```

```
    for i in range(MAX_LEN):
        preds, hidden, _ = decoder(dec_input, features, hidden)
        predictedWord = tf.argmax(preds, axis=1).numpy()
        result.append(predictedWord[0])
        if predictedWord == 2:
            return result
        dec_input = tf.expand_dims([predictedWord[0]], 0)

    return result

no_samples = 20
np.random.seed(22)
test_indices = np.random.permutation(len(img_name_val))[0:no_samples]
for ind in test_indices:
    print_pred(img_name_val[ind])

if not os.path.exists(MODEL_DIR):
    os.mkdir(MODEL_DIR)

encoder.save(os.path.join(MODEL_DIR, 'encoder_attention_glove'))
#decoder.save_weights(os.path.join(MODEL_DIR, 'decoder_attention_glove'), )

encoderlol = Encoder(embedding_dim)
encoderlol.build(input_shape=(64,2048))
print(encoderlol.summary())

print(decoder.summary())
```