Image Captioning Using Attention Mechanism

Sanjay Singh

san.singhsanjay@gmail.com

Introduction

- Automated Image Captioning (or simply Image Captioning) can be defined as generating a textual description for a given image.
- This problem was well researched by Andrej Karpathy in his PhD at Stanford University.
- Deep Learning has achieved state-of-art result in Image Captioning.
- In this project, the advanced solution (i.e., Attention Mechanism or to be more specific, Visual Attention Mechanism) for Image Captioning is implemented. A classical solution for Image Captioning is Encoder/Decoder based
- Attention Mechanism is also quite useful in Neural Machine Translation, i.e., translating text from one natural language to another.
- Following is an example of Image Captioning:

1. Children sit and watch the fish moving in the pond.
2. people stare at the orange fish.
3. Several people are standing near a fish pond.
4. Some children watching fish in a pool.
5. There are several people and children looking into water with a blue tiled floor and gold fish.

Introduction…

- Some other sample images are:






Introduction…

- Following are some of the applications of Image Captioning:
    1. Self Driving Cars
    2. Aid to blind people: It can guide blind people by generating text for the scene in front and speaking it by using TTS (Text to Speech) systems.
    3. CCTV cameras are everywhere, but along with viewing the world, it can generate relevant captions, then we can raise alarms as any malicious activity take place.
    4. Image Captioning can make Google Image Search better.

Dataset

- Flickr8k dataset is used here. Following is the link of this dataset:
  https://www.kaggle.com/adityajn105/flickr8k
- Flickr8k has 8,091 images with caption.txt file containing five captions for each image. All these five captions are written by different people. Thus,
  8,091 Images x 5 Captions = 40,455 Image-Captions
- Size of this dataset: 1.04 GB
- A training file and testing file containing name of images to be used in training and testing is downloaded from the Internet (source is missing).
- Other than Flickr8k dataset, some other datasets for Image Captioning are:
  1. Flickr30k: It contains 30,000 images
  2. MS-COCO: It contains 1,80,000 images. This is the largest dataset for Image Captioning.
- We will work on Flickr8k dataset as this is sufficient to learn the implementation of Automatic Image Captioning using Attention Mechanism.

Technology

- In this project, advanced solution (i.e., Attention Mechanism or Visual Attention Mechanism) for Image Captioning is implemented.
- A naive approach is also there, called as Classical Encoder/Decoder based approach. This advanced solution is also Encoder/Decoder based, however it has an additional layer for Attention Mechanism.
- Attention Mechanism is also quite useful for Neural Machine Translation (i.e., translating text from one natural language to another natural language). To be more specific, Attention Mechanism for Image Captioning is called as Visual Attention Mechanism.
- This project is at the intersection of two technologies:
    1. Computer Vision (CV): To understand the content of a given image.
    2. Natural Language Generation (NLG): NLG transforms data into plain English text.
- Applications of Natural Language Generation (NLG):
    1. Freeform text generation: User provides an input, like a phrase, sentence or paragraph and the NLG model generates continuation of this input as output. For instance, Google Smart Compose predicts a phrase following a word input in Gmail.
    2. Question Answering: This is a system that can answer questions posed by humans. These systems can be open-ended or close-ended (domain specific).

Technology…

3. Summarization: Summarization reduces the amount of information while capturing the most important details in a narrative. This is of two types:
    i. Extractive Summarization: It takes the most important phrases or sentences from the given text and stitches them together to form a summarized narrative.
    ii. Abstractive Summarization: This is equivalent of a human writing a summary in his / her own words. For instance, headline generation, abstract for journals / whitepaper / etc.
4. Image Captioning

- How NLG is different from NLP: NLP is focussed on deriving analytic insights from textual data. Whereas, NLG is used to synthesize textual content by combining analytic output with contextualized narratives. In short, NLP reads while NLG writes.

Attention Mechanism

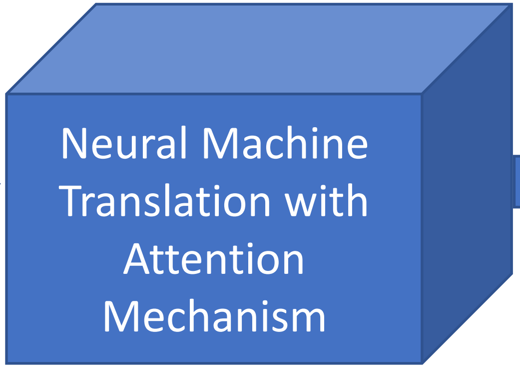
- Attention Mechanism is one of the most influential ideas in the Deep Learning.
- Initially, this idea was designed for Neural Machine Translation. However, today it can be used in various problems, like Image Captioning, etc.
- As we know, there is a classical Encoder / Decoder based solution for Image Captioning. The drawback with that classical approach is:

  Since each word of the caption would be defining a part of the image, thus by considering the whole representation of image to generate the next word of caption which will be describing a part of the image would not be efficient, especially for long captions or descriptions.
- Attention mechanism is a complex cognitive ability that human possess. When people receive information, they consciously ignore some of the secondary information. This ability of self selection is called Attention.
- Attention Mechanism allows the neural networks to have the ability to focus on its subset of inputs to select specific features.
- Neural Network architecture with Attention Mechanism for Image Captioning is also an Encoder / Decoder like classical Encoder / Decoder solution but with an additional layer, called as Attention Mechanism.

Attention Mechanism…

- Attention Mechanism was actually developed for Neural Machine Translation (i.e., translating text from one natural language to another). For this, Attention Mechanism gets text in one natural language and translate it into another natural language, as depicted below:

A dog is running fast over the beach.



एक कुत्ता समुद्र तट पर तेजी से दौड़ रहा है।

Thus, in case of Neural Machine Translation, input and output, both are text.

- Whereas, in case of Image Captioning, input is an image and output is text. That's why Attention Mechanism was modified to take an image as input. This modified version of Attention Mechanism is known as Visual Attention Mechanism because it takes an image as input.

A smiling man is sitting in his vegetable shop

Attention Mechanism…

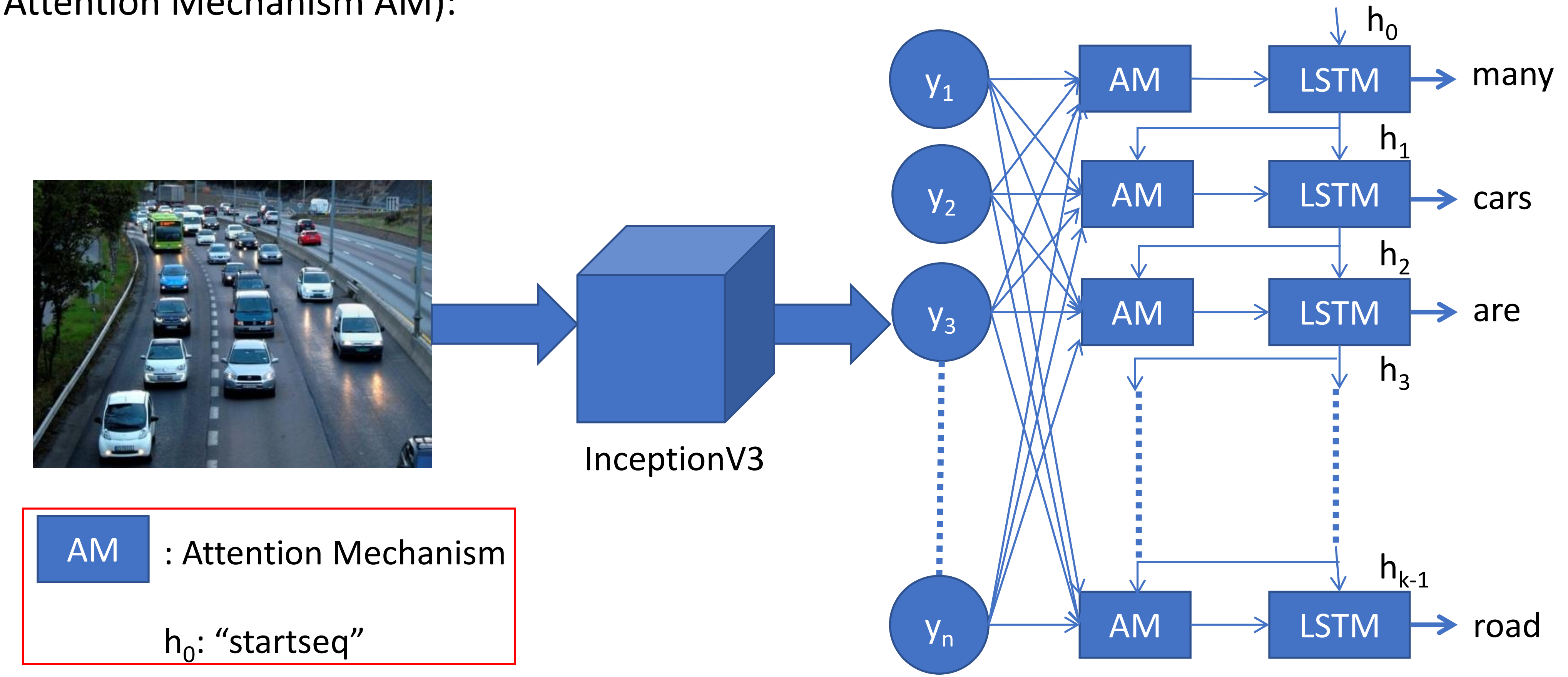
- With Attention Mechanism for Image Captioning, the image is first divided into n parts and we compute representation of each part (representation of each part is denoted by h_1, h_2, h_3, …, h_n) by a CNN (Convolutional Neural Network).

  When the RNN (Recurrent Neural Network) is generating a new word, the Attention Mechanism focuses on the relevant part of the image. So, the decoder uses the specific parts of input image while generating a new (or next) word. Following is the example:

Figure 3. Examples of attending to the correct object (*white* indicates the attended regions, *underlines* indicated the corresponding word)

Attention Mechanism…

- Following is the architecture of our Neural Network with Attention Mechanism for Image Captioning (similar to Classical Encoder / Decoder architecture but with an additional layer of Attention Mechanism AM):



Attention Mechanism…

- Following is how this advanced solution works:

  If network has predicted i words, then the hidden state of LSTM would be h_i. This h_i would be passed to the AM (i.e., Attention Mechanism) which will select the relevant part of the image by using h_i as context and pass this relevant part of image (say, z_i) to LSTM to predict the next word which will make the hidden state of LSTM as h_{i+1}.
- There are two types of Attention Mechanism:
  1. Global Attention Mechanism (aka Luong’s Attention): Attention is placed on all source position.
  2. Local Attention Mechanism (aka Bahdanu’s Attention): Attention is placed only on a few source positions.
- Both the types of Attention Mechanisms differ from the classical Encoder / Decoder architecture only in the decoding phase due to presence of AM (i.e., Attention Mechanism layer).
- Both Global and Local Attention Mechanism differ in the way that they compute context vector (aka thought vector, i.e., the output of Encoder), can be represented by c(t).

Attention Mechanism…

- Global Attention (i.e., Luong’s Attention) takes into consideration all encoder hidden states to derive the context vector c(t).

  This can be further simplified as in case of Neural Machine Translation, Global Attention focuses on all source side words to derive all target words. Similarly in case of Image Captioning, Global Attention considers all divided parts of input image to generate the caption (or textual description) for image.

  Thus, it is computationally very expensive and is impractical when translating for long sentences.
- Therefore, we will see implementation of only Local Attention (i.e., Bahdanu’s Attention).
- Local Attention (or Bahdanu’s Attention):

  Suppose, we have divided our input image into an input sequence of 5 parts. Now, before we start decoding, we first need to encode the input sequence into a set of internal states (h_1, h_2, h_3, h_4, h_5).

  Now, the next word in the output sequence is dependent on the current state of the decoder as well as on the hidden state of the encoder. Thus, at each time step, we consider these two things and follow the below steps:

Attention Mechanism…

We want our decoder to pay more attention to the states h_1 and h_2 (assume) while paying less attention to the remaining states of the encoder. For this reason, we train a feed forward neural network which will learn to identify relevant encoder states by generating a high score for the states for which attention is to be paid while low score for the states which are to be ignored.

Let s_1, s_2, s_3, s_4, s_5 be the scores generated for the states h_1, h_2, h_3, h_4, h_5. Since, we want to pay attention to h_1 and h_2 (assume) and ignore (h_3, h_4, h_5), thus s_1 and s_2 will be high while (s_3, s_4, s_5) are relatively low.

Once these scores are generated, we apply a softmax on these scores to produce the attention weights (e_1, e_2, e_3, e_4, e_5).

The advantage of applying softmax is as below:

1. All the weights lie between 0 and 1, i.e., $(e_1, e_2, e_3, e_4, e_5) \in [0, 1]$.
2. All the weights sum to 1, i.e., $e_1 + e_2 + e_3 + e_4 + e_5 = 1$.

Suppose, following are the values of attention weights:

$e_1 = 0.75$ $e_2 = 0.2$ $e_3 = 0.02$ $e_4 = 0.02$ $e_5 = 0.01$

Thus, due to large values of e_1 and e_2, attention will be on h_1 and h_2, others (i.e., h_3, h_4, h_5) will be ignored due to small values of (e_3, e_4, e_5).

Attention Mechanism…

Now, we will compute the context vector (or thought vector) which will be used by the decoder in order to predict (or generate) the next word in the sequence:

$$context\ vector, c(t) = e_1 * h_1 + e_2 * h_2 + e_3 * h_3 + e_4 * h_4 + e_5 * h_5$$

Due to high values of e_1 and e_2, context vector (or thought vector) will have more information from the states h_1 and h_2, and relatively less information from the states h_3, h_4 and h_5.

Finally, the decoder will use the below two inputs to generate the next word in the sequence:

1. The context vector (or thought vector), c(t).
2. The output word generated from the previous time step.

Note that for the first time step, since there is no output from the previous time step, thus we use a special <startseq> token for this purpose.

The decoder then generates the next word in the sequence and along with the output, the decoder will also generate an internal hidden state, let's call it as d_1.

In order to generate the next word, the decoder will repeat the same procedure.

At the end, the decoder will output <endseq> token and we stop the generation process.

Evaluation Metric

- Evaluating NLG system is a much more complicated task. There are following four evaluation metrics for evaluating a NLG system:
    1. Bilingual Evaluation Understudy (BLEU Score)
    2. Recall Oriented Understudy for Gisting Evaluation (ROUGE)
    3. Metric for Evaluation for Translation with Explicit Ordering (METEOR)
    4. Consensus based Image Descriptive Evaluation (CIDEr)
- Since above metrics differ mostly in terms of the way Precision and Recall (i.e., Sensitivity) calculated, thus we will first see how to calculate Precision and Recall (or Sensitivity) in NLG.
- In general,

$$Precision = \frac{True\ Positive}{True\ Positive + False\ Positive} = \frac{No.of\ correctly\ predicted\ positives}{Total\ no.of\ predicted\ positives}$$

$$Recall\ (or\ Sensitivity) = \frac{True\ Positive}{True\ Positive + False\ Negative} = \frac{No.of\ correctly\ predicted\ positives}{Total\ no.of\ actual\ positives}$$

Evaluation Metric…

- In NLG, predicted (or generated) text is called as Candidate text and the actual text is called as Reference text.
- Following is the definition of Precision and Recall (or Sensitivity) in NLG:

$$Precision = \frac{No. of\ words\ in\ Candidate\ matched\ with\ Reference}{Total\ no. of\ words\ in\ Candidate}$$

$$Recall\ (or\ Sensitivity) = \frac{No. of\ words\ in\ Candidate\ matched\ with\ Reference}{Total\ no. of\ words\ in\ Reference}$$

- Consider the following example:

  Reference: "I work on machine learning"

  Candidate A: "I work"

  Candidate B: "He works on machine learning"

$$Precision\ of\ Candidate\ A = \frac{2}{2} = 100\%$$

$$Recall\ (or\ Sensitivity) of\ Candidate\ A = \frac{2}{5} = 40\%$$

$$Precision\ of\ Candidate\ B = \frac{3}{5} = 60\%$$

$$Recall\ (or\ Sensitivity) of\ Candidate\ B = \frac{3}{5} = 60\%$$

Evaluation Metric…

- All previous calculations are done by using unigrams (i.e., no. of words, n = 1). These calculation can also be done by using bigrams (n = 2), trigrams (n = 3) and so on.
- Consider the following example:

  Reference: "I work on machine learning"

  Candidate A: "He works on machine learning"

  Candidate B: "He works on on machine machine learning learning"

  In case of unigram (i.e., n = 1):

$$Precision\ of\ Candidate\ A = \frac{3}{5} = 60\% \qquad Recall\ (or\ Sensitivity)\ of\ Candidate\ A = \frac{3}{5} = 60\%$$

$$Precision\ of\ Candidate\ B = \frac{6}{8} = 75\% \qquad Recall\ (or\ Sensitivity)\ of\ Candidate\ B = \frac{6}{5} = 120\%$$

- There is a modified n-gram scheme in which we match candidate's n-grams only as many times as they are present in any of reference text. Thus, "on", "machine" and "learning of Candidate B will get match only once in unigram (i.e., n = 1).

$$Precision\ of\ Candidate\ A = \frac{3}{5} = 60\% \qquad Recall\ (or\ Sensitivity)\ of\ Candidate\ A = \frac{3}{5} = 60\%$$

$$Precision\ of\ Candidate\ B = \frac{3}{8} = 37.5\% \qquad Recall\ (or\ Sensitivity)\ of\ Candidate\ B = \frac{3}{5} = 60\%$$

Evaluation Metric…

- To include all the n-gram precision scores (i.e., precision calculated by using unigram, bigram, trigram, etc.) in our final precision, we take their geometric mean. This is done because it has been found that precision decreases exponentially with n and we would require logarithmic averaging to represent all values fairly.

$$Precision = \exp\left(\sum_{n=1}^{N} w_n \log p_n\right), \qquad where, w_n = \frac{1}{n}$$

- Best Match Length: The problem with recall (or sensitivity) is that there may be many reference texts. So it is difficult to calculate the sensitivity of the candidate w.r.t a general reference. However, it is intuitive to think that a longer candidate text is more likely to contain a larger fraction of some reference than a shorter candidate.

  Therefore, we can introduce recall by just penalizing brevity (meaning: the state of being short or quick) in candidate texts. This is done by adding a multiplicative factor, called as Brevity Penalty (BP) with the modified n-gram precision as follows:

$$BP = \begin{cases} 1, & if\ c > r \\ \exp\left(1 - \frac{r}{c}\right), & otherwise \end{cases}$$

Evaluation Metric…

Where,

“c”: Total length of candidate translation corpus

“r”: The effective reference length of corpus, i.e., average length of all references

As the candidate length decreases, the ration $\frac{r}{c}$ increases, and the BP decreases exponentially.

Following is the formula of BLEU Score:

$$BLEU\ Score = BP.(Modified\ n - gram\ precision)$$

BLEU Score ϵ [0, 1].

BLEU is used for:

1. Neural Machine Translation (or simply Machine Translation)
2. Image Captioning
3. Text Summarization
4. Speech Recognition

BLEU Score can be directly used from the “nltk” library of Python:

```
import nltk.translate.bleu_score as bleu
bleu_sc = bleu.sentence_bleu(reference, candidate) # for one reference text
bleu_sc = bleu.corpus_bleu(reference, candidate) # for multiple reference text
```

Evaluation Metric…

- We have seen how to calculate modified n-gram precision for one reference. However, practically we have multiple references. Thus, let us see how to calculate it for multiple references:

  Candidate 1: It is a guide to action which ensures that the military always obeys the commands of the party.

  Reference 1: It is a guide to action that ensures that the military will forever heed party commands.

  Reference 2: It is the guiding principle which guarantees the military forces always being under the command of the party.

  Reference 3: It is the practical guide for the army always to heed the directions of the party.

- We will calculate following:
  1. Count: Count the maximum number of times a candidate n-gram occurs in the candidate.
  2. Ref1 Count, Ref2 Count and Ref3 Count: For each reference sentence, count the number of times a candidate n-gram occurs.
  3. Max Ref Count: Take the maximum number of n-grams occurrences in reference count.
  4. Count Clip: Take the minimum number of Count and Max Ref Count.

     Count Clip = min(Count, Max Ref Count)
  5. Divide the Clipped Count by the total unclipped number of candidate n-grams to get the modified precision score (p_n).

Evaluation Metric…

Candidate n-gram	Count	Ref1 Count	Ref2 Count	Ref3 Count	Max Ref Count	Count Clip
“It”	1	1	1	1	1	1
“is”	1	1	1	1	1	1
“a”	1	1	0	0	1	1
“guide”	1	1	0	1	1	1
“to”	1	1	0	1	1	1
“action”	1	1	0	0	1	1
“which”	1	0	1	0	1	1
“ensures”	1	1	0	0	1	1
“that”	2 (1)	2	0	0	2	2 (1)
“the”	3	1	4	4	4	3
“military”	1	1	1	0	1	1
“always”	1	0	1	1	1	1
“obeys”	0 (1)	0	0	0	0	0
“commands”	1	1	0	0	1	1
“of”	0 (1)	0	1	1	1	0 (1)
“party”	1	0	0 (1)	1	1	1
18						17

Evaluation Metric…

Applying step 5 (calculating Modified Precision Score, p_n):

$$p_n = \frac{17}{18}$$

Modified n-gram Precision Score (p_n) captures:

1. Adequacy: A candidate using the same words as in the references tends to satisfy adequacy.
2. Fluency: The long n-gram matches between candidate and reference account for fluency.

$$Brevity\ Penalty, BP = \begin{cases} 1, & if\ c > r \\ exp\left(1 - \frac{r}{c}\right), & otherwise \end{cases}$$

Where,

"r": Count of words in reference. "c": Count of words in candidate.

$$BLEU = BP.\exp\left(\sum_{n=1}^{N} w_n \log p_n\right)$$

Where,

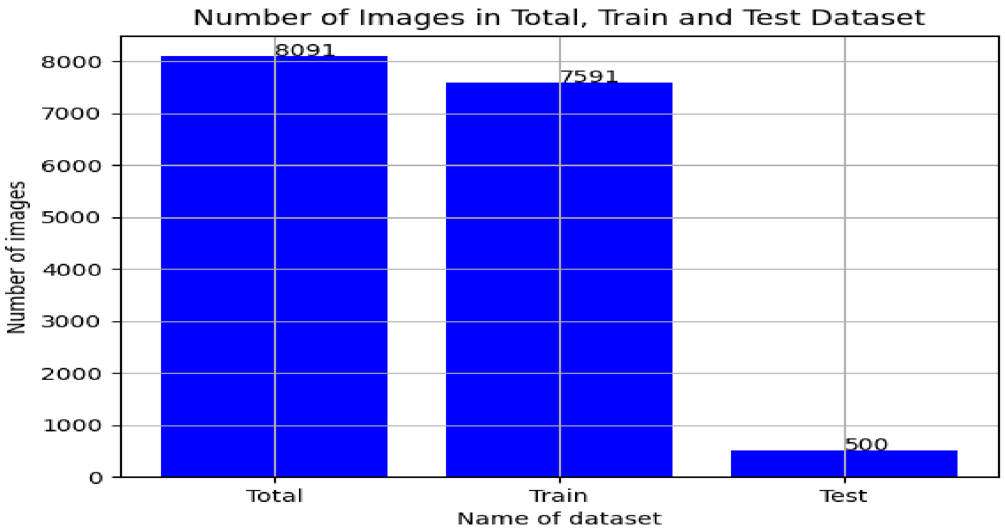
N: No. of n-grams, we usually use unigram, bigram, trigram, 4-gram.

$w_n = \frac{1}{N}, by\ default\ N = 4$ and p_n: Modified Precision Score

Scripts Execution Flow

1. Script “scripts/check_training_val_test.py” does following tasks:
   i. Verifies that the name of images given in training and testing .txt files (i.e., Flickr_8k.trainImages.txt and Flickr_8k.testImages.txt) are in captions.txt file or not.
   ii. It also creates a file Flickr8k.valImages.txt which has name of images that are not in training and not in testing .txt file (i.e., Flickr_8k.trainImages.txt and Flickr_8k.testImages.txt). These images can be used for validation purpose, as the name of file suggests.
   iii. At last, this script generates and saves following plot which summarizes the number of images in the entire dataset; training, validation and testing subsets.
2. Script “scripts/segregate_train_val_test.py” does following tasks:
   i. This script reads image filenames from Flickr_8.trainImages.txt, Flickr_8k.valImages.txt and Flickr_8k.testImages.txt; extracts these filenames and their captions (5 captions per file) from captions.txt;
   ii. Since we need more data to train Attention Mechanism in order to get acceptable performance, the complete training data along with complete validation and half of testing data is combined to use for training. Only remaining half of test data is used for testing.
   iii. Then, this script generated two csv files: train_image_caption.csv and test_image_caption.csv having two columns: ‘image’ and ‘caption’. ‘image’ column has image names and ‘caption’ columns has all 5 captions of corresponding image separated by ‘<>’.
   iv. At last, this script generated a plot showing below:

Scripts Execution Flow…



3. Script “scripts/preprocessing_train.py” does following tasks:
   i. It reads train_image_caption.csv file. Extracts image filenames and their five captions (joint by “<>”).
   ii. It takes out each caption of each image file and perform these operations: converts into lower case, removes all special characters, removes all single characters (like ‘a’, ‘s’, etc.) and removes numerical figures (such as ‘1’, ‘2’, etc.). This part is well known as Data Cleaning or Data Pre-processing.
   iii. Then, it puts “<startseq>” and “<endseq>” before and after each processed caption, join these processed captions by “#” and save it with its image filename in file train_image_caption_processed.csv. “<startseq>” and “<endseq>” are special tokens.
   iv. Along with this, it also saves following files: max_caption_length.txt, vocabulary.txt & WordFreq.csv.

Scripts Execution Flow…

4. Script “scripts/preprocessing_test.py” does following tasks:
    i. This script reads test_image_caption.csv file. Extracts image filenames & their five captions (joint by “<>”).
    ii. It takes out each caption of each image file and perform these operations: converts into lower case, removes all special characters, removes all single characters (like ‘a’, ‘s’, etc.) and removes numerical figures (such as ‘1’, ‘2’, etc.). This part is well known as Data Cleaning or Data Pre-processing. This entire processing will be done by referring vocabulary.txt file as words not in vocabulary will be deleted from captions.
    iii. Then, it joins processed captions by “#” and save them with their image filename in file test_image_caption_processed.csv. “<startseq>” and “<endseq>” special tokens are not required for test data because we have to directly match with generated captions which will not have “<startseq>” and “<endseq>” tokens.
5. Script “scripts/gen_image_features.csv” does following tasks:
    i. This script loads each image which is in train_image_caption_processed.csv, resize it for the pre-trained model to generate bottleneck features (here, we have used InceptionV3, discussed in later slides).
    ii. Then, it passes each of these images through our chosen pre-trained model (here it is InceptionV3) and generates bottleneck feature of dimension (8, 8, 2048). Then, script is reshaping it to (64, 2048) and saving it as a npy file. Script has done this task in batches of 64 images. In this manner, it has saved 7,591 npy files (since we have 7591 file names in train_image_caption_processed.csv file).

Scripts Execution Flow…

6. Script “scripts/training.py” or “scripts/training.ipynb” does following tasks:
   i. First, we have to manually upload following files on paperspace.com:

      i. train_image_caption_processed.csv
      ii. All 7591 npy files - bottleneck features
      iii. vocabulary.txt
      iv. max_caption_length.txt

      All files mentioned above are generated in previous steps.
   ii. First thing that this script does is reading above mentioned files. It will create dictionary type variable for i. and ii. with image file name as key; and captions and bottleneck features as values for quick access during training.
   iii. After reading “vocabulary.txt”, it creates a dictionary, “wordtoix” (i.e., word-to-index). This dictionary type variable has all words of our cleaned captions as key and their line indices (from 0) in vocabulary.txt as value. This variable (i.e., “wordtoix”) is helpful for training process as it provides a numerical representation of our textual captions (since our algorithm processes only numbers).
   iv. Similarly, we also create one more variable “ixtoword”. This variable is also of dictionary type but has line indices (from 0) as key and word as value. This variable will be helpful during inference.

Scripts Execution Flow…

vi. Script is also creating a variable “embedding_matrix”. It’s a numpy.ndarray type variable of dimension (vocab_size, EMBEDDING_DIMENSION), where vocab_size is the number of words in file vocabulary.txt and EMBEDDING_DIMENSION is 200 which is the dimension of embeddings (or bottleneck features) generated by GloVe model.
Thus, we pass each word of vocabulary.txt file to GloVe model, it will generate a vector of 200 dimensions that we will store in embedding_matrix and at last we will save this variable as a csv file. So, the 200 dimensional embedding (or vector) of i^{th} word in vocabulary.txt file is at i^{th} index in embedding_matrix variable and also in the saved csv file, “embedding_matrix.csv”.
We will use this variable (or values) in our encoder-decoder neural network architecture.

vii. Due to memory limitations, we cannot fed the entire data on encoder-decoder neural network architecture. Thus, a function called as “data_generator” is created which will prepare data and pass in batches to our encoder-decoder neural network architecture for training. Following tasks are performed by this “data_generator” function:

a. Pick an image name, extract its all five captions and bottleneck features from dictionary created in 5.ii. step.

b. Prepare training variable X (i.e., independent variable) and Y (i.e. dependent variable) as follows:

Scripts Execution Flow…

Suppose following are the five processed captions for i^{th} image:

1. "startseq man driving scooter endseq"
2. "startseq man on scooter endseq"
3. "startseq man wearing helmet driving scooter endseq"
4. "startseq man enjoying driving scooter endseq"
5. "startseq happy man driving scooter endseq"

For now, assume that this is the only image and these are the only five captions in our entire dataset. Thus, our vocabulary (and also wordtoix dictionary variable) will look like:

1. "startseq"
2. "man"
3. "driving"
4. "scooter"
5. "endseq"
6. "on"
7. "wearing"
8. "helmet"
9. "enjoying"
10. "happy"

Following are the lengths of above captions: 5, 5, 7, 6, 6. Thus, max caption length is 7 ("max_caption_length").
Now, the 2048 dimensional bottleneck feature generated by a pre-trained model (InceptionV3, here) of above image is: $[f_1, f_2, f_3, ..., f_{2048}]$.
Now, we will create numerical representation of above captions by using wordtoix dictionary variable and pad zeros in them to make length of each caption equal to max_caption_length (i.e., 7):

1. (1, 2, 3, 4, 5, 0, 0)
2. (1, 2, 6, 4, 5, 0, 0)
3. (1, 2, 7, 8, 3, 4, 5)
4. (1, 2, 9, 3, 4, 5, 0)
5. (1, 10, 2, 3, 4, 5, 0)

Zeros are padded to make length of each caption equal to max_caption_length because our neural network will take input of a fixed size.

Scripts Execution Flow…

Since, our model will generate captions word by word, thus we will train our image captioning model word by word, as shown below:

S.No.	Caption Input	X (Training Independent Variable)	Y (Dependent Variable)
1	"startseq"	$[f_1, f_2, f_3, ..., f_{2048}]$, [1, 0, 0, 0, 0, 0, 0]	"man" (2)
2	"startseq man"	$[f_1, f_2, f_3, ..., f_{2048}]$, [1, 2, 0, 0, 0, 0, 0]	"driving" (3)
3	"startseq man driving"	$[f_1, f_2, f_3, ..., f_{2048}]$, [1, 2, 3, 0, 0, 0, 0]	"scooter" (4)
4	"startseq man driving scooter"	$[f_1, f_2, f_3, ..., f_{2048}]$, [1, 2, 3, 4, 0, 0, 0]	"endseq" (5)
5	"startseq"	$[f_1, f_2, f_3, ..., f_{2048}]$, [1, 0, 0, 0, 0, 0, 0]	"man" (2)
6	"startseq man"	$[f_1, f_2, f_3, ..., f_{2048}]$, [1, 2, 0, 0, 0, 0, 0]	"on" (6)
7	"startseq man on"	$[f_1, f_2, f_3, ..., f_{2048}]$, [1, 2, 6, 0, 0, 0, 0]	"scooter" (4)
8	"startseq man on scooter"	$[f_1, f_2, f_3, ..., f_{2048}]$, [1, 2, 6, 4, 0, 0, 0]	"endseq" (5)
9	"startseq"	$[f_1, f_2, f_3, ..., f_{2048}]$, [1, 0, 0, 0, 0, 0, 0]	"man" (2)
10	"startseq man"	$[f_1, f_2, f_3, ..., f_{2048}]$, [1, 2, 0, 0, 0, 0, 0]	"wearing" (7)
11	"startseq man wearing"	$[f_1, f_2, f_3, ..., f_{2048}]$, [1, 2, 7, 0, 0, 0, 0]	"helmet" (8)
12	"startseq man wearing helmet"	$[f_1, f_2, f_3, ..., f_{2048}]$, [1, 2, 7, 8, 0, 0, 0]	"driving" (3)

Scripts Execution Flow…

S.No.	Caption Input	X (Training Independent Variable)	Y (Dep.Var.)
13	“startseq man wearing helmet driving”	[f_1, f_2, f_3, …., f_{2048}], [1, 2, 7, 8, 3, 0, 0]	“scooter” (4)
14	“startseq man wearing helmet driving scooter”	[f_1, f_2, f_3, …., f_{2048}], [1, 2, 7, 8, 3, 4, 0]	“endseq” (5)
15	“startseq”	[f_1, f_2, f_3, …., f_{2048}], [1, 0, 0, 0, 0, 0, 0]	“man” (2)
16	“startseq man”	[f_1, f_2, f_3, …., f_{2048}], [1, 2, 0, 0, 0, 0, 0]	“enjoying”(9)
17	“startseq man enjoying”	[f_1, f_2, f_3, …., f_{2048}], [1, 2, 9, 0, 0, 0, 0]	“driving” (3)
18	“startseq man enjoying driving”	[f_1, f_2, f_3, …., f_{2048}], [1, 2, 9, 3, 0, 0, 0]	“scooter” (4)
19	“startseq man enjoying driving scooter”	[f_1, f_2, f_3, …., f_{2048}], [1, 2, 9, 3, 4, 0, 0]	“endseq” (5)
20	“startseq”	[f_1, f_2, f_3, …., f_{2048}], [1, 0, 0, 0, 0, 0, 0]	“happy” (10)
21	“startseq happy”	[f_1, f_2, f_3, …., f_{2048}], [1, 10, 0, 0, 0, 0, 0]	“man” (2)
22	“startseq happy man”	[f_1, f_2, f_3, …., f_{2048}], [1, 10, 2, 0, 0, 0, 0]	“driving” (3)
23	“startseq happy man driving”	[f_1, f_2, f_3, …., f_{2048}], [1, 10, 2, 3, 0, 0, 0]	“scooter” (4)
24	“startseq happy man driving scooter”	[f_1, f_2, f_3, …., f_{2048}], [1, 10, 2, 3, 4, 0, 0]	“endseq” (5)

“data_generator_function” generates data for a particular number of images at a time (to prevent Memory Overflow error) in the above manner and pass it to the neural network for training.
After completion of training, script will save model in “output” directory.

Scripts Execution Flow

7. After successful completion of training, one can use the saved model by any of the following scripts:
   i. Script “scripts/inference_one_image_GColab.ipynb” to get caption of one image at a time. User has to keep the test image in a “single_test_image” directory, and enter its name when script ask.
   ii. Script “scripts/inference_test_images_GColab.ipynb” to get captions generated (or predicted) by trained model for all images saved under “test_image” directory. This script will save all those generated (or predicted) captions in a .txt file under “output/test_image_generated_captions/” directory.

Pre-Trained Models

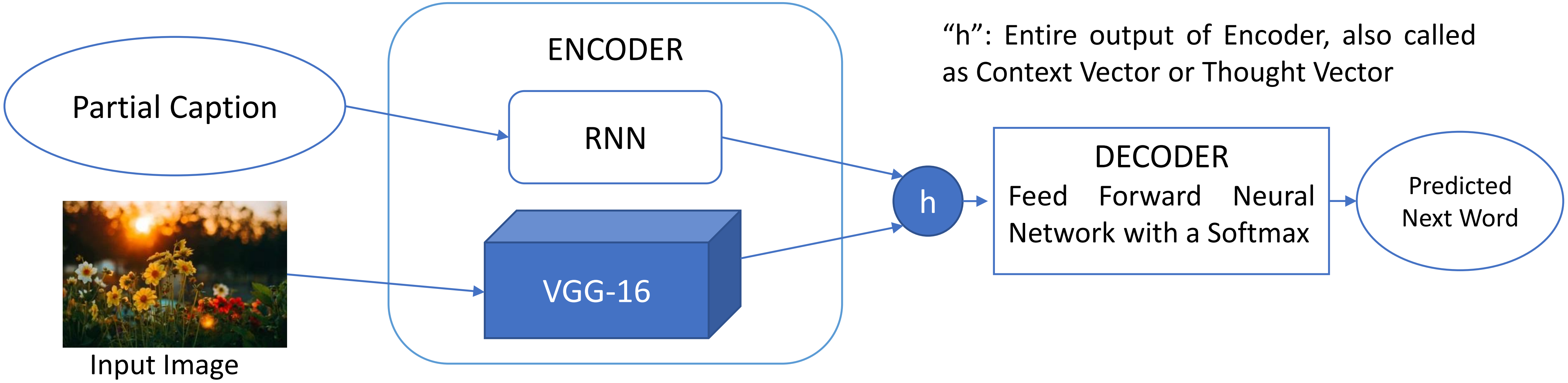
- We have used only one pre-trained model to generate bottleneck features of images. This model is InceptionV3:

Output of this layer is taken as bottleneck feature

type	patch size/stride or remarks	input size
conv	$3 \times 3/2$	$299 \times 299 \times 3$
conv	$3 \times 3/1$	$149 \times 149 \times 32$
conv padded	$3 \times 3/1$	$147 \times 147 \times 32$
pool	$3 \times 3/2$	$147 \times 147 \times 64$
conv	$3 \times 3/1$	$73 \times 73 \times 64$
conv	$3 \times 3/2$	$71 \times 71 \times 80$
conv	$3 \times 3/1$	$35 \times 35 \times 192$
$3 \times$Inception	As in figure 5	$35 \times 35 \times 288$
$5 \times$Inception	As in figure 6	$17 \times 17 \times 768$
$2 \times$Inception	As in figure 7	$8 \times 8 \times 1280$
pool	8×8	$8 \times 8 \times 2048$
linear	logits	$1 \times 1 \times 2048$
softmax	classifier	$1 \times 1 \times 1000$

Training & Neural Networks Specific

- Neural Network Framework: Keras (version: 2.4.3)
- Epchs: 20
- Number of images (with 5 captions) per batch: 3
- Loss Function: “categorical_crossentropy”
- Optimization Function: Adam
- Pre-trained model used: 1. InceptionV3
- It is clear from the previous tables (X and Y) that the data is time-series based where sequence matters a lot (as it builds context of captions, every next word is dependent on the current word), thus RNN with LSTM (Long Short Term Memory) cells is used here.
- As it is said earlier that the classical Encoder-Decoder solution is implemented here. Following is the architecture of this solution:



Training & Neural Networks Specific…

- Following is the code of neural network used here:

```
inputs1 = Input(shape=(2048,))
fe1 = Dropout(0.5)(inputs1)
fe2 = Dense(256, activation=‘relu’)(fe1)
inputs2 = Input(shape=(max_caption_length,))
se1 = Embedding(vocab_size, EMBEDDING_DIM, mask_zero=True)(inputs2) # EMBEDDING_DIM=200
se2 = Dropout(0.5)(se1)
se3 = LSTM(256)(se2)
decoder1 = add([fe2, se3])
decoder2 = Dense(256, activation=‘relu’)(decoder1)
outputs = Dense(vocab_size, activation=‘softmax’)(decoder2)
model = Model(inputs=[inputs1, inputs2], outputs=outputs)
```

- “embedding_matrix” created in 5.vi. under “Scripts Execution Flow” is used here as weights:

```
model.layers[2].set_weights([embedding_matrix])
model.layers[2].trainable = False
```

Training & Neural Networks Specific…

- Following is the summary of this model:

```
Layer (type)                    Output Shape         Param #     Connected to
==================================================================================================
input_1 (InputLayer)            [(None, 2048)]       0

dropout (Dropout)               (None, 2048)         0           input_1[0][0]

dense (Dense)                   (None, 256)          524544      dropout[0][0]

input_2 (InputLayer)            [(None, 28)]         0

embedding (Embedding)           (None, 28, 200)      330400      input_2[0][0]

dropout_1 (Dropout)             (None, 28, 200)      0           embedding[0][0]

lstm (LSTM)                     (None, 256)          467968      dropout_1[0][0]

add (Add)                       (None, 256)          0           dense[0][0]
                                                                 lstm[0][0]

dense_1 (Dense)                 (None, 256)          65792       add[0][0]

dense_2 (Dense)                 (None, 1652)         424564      dense_1[0][0]
==================================================================================================
Total params: 1,813,268
Trainable params: 1,813,268
```

Training & Neural Networks Specific…

- In the previous neural network code, there is an "Embedding" layer in which we have loaded the embedding_matrix which has GloVe bottleneck features (200 dimensional) of all words in our vocabulary.txt file. Following is the purpose of "Embedding" layer:
  - Embedding Layer is one of the available layers in Keras.
  - This layer is mainly useful in Natural Language Processing (NLP), and thus in Natural Language Generation (NLG).
  - In NLP (or NLG), one can use pre-trained word embeddings such as GloVe. Alternatively, one can also train our own embeddings using Keras embedding layer.
  - Word Embeddings can be thought of as an alternate to one-hot encoding along with dimensionality reduction.
  - As we know that while dealing with textual data, we need to convert it into numbers before feeding into any machine learning model. This can be simply done by considering each word as a class (or category) and transforming every word into one-hot vectors). Thus, if we have 10,000 words in our vocabulary (i.e., 10,000 unique words, then a matrix of 10,000 x 10,000 will form where each row will have only one "1" and rest are zero. Following are the two issues with this approach:
    1. This will require a lot of storage space.
    2. This will reduce model's efficiency as there will not be any mathematical justification for such representation

Training & Neural Networks Specific…

- Embedding layer enables us to convert each word into a fixed length vector of defined size (reduced dimension).
- The resultant vector have real values instead of just 0s and 1s.
- This way "Embedding" layer works like a lookup table. The words (or their indices) are the keys in this table while the dense word vectors are the values.

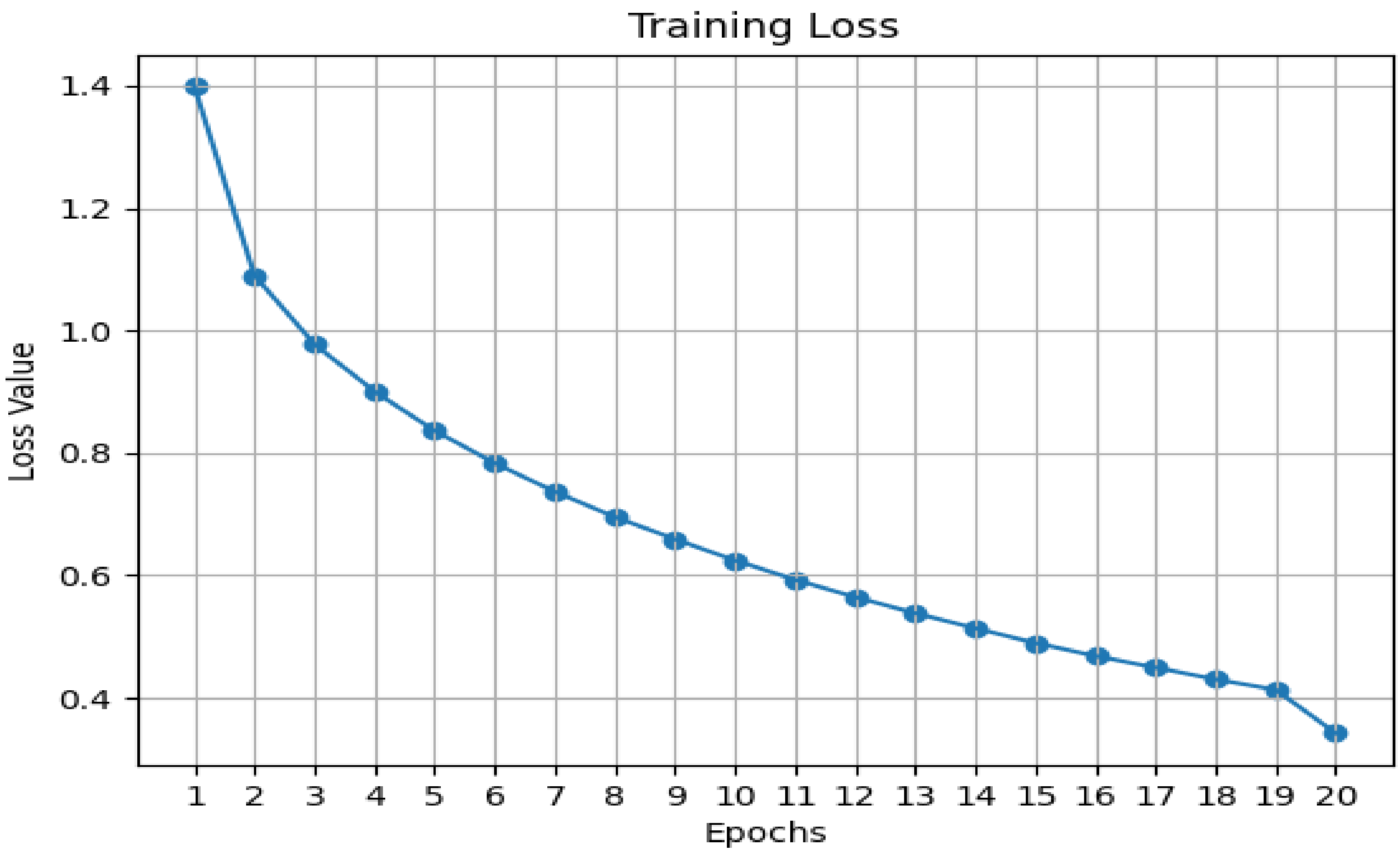
➢ Model is trained on the following platform:

On an AMD E-series (E2-7110) CPU, it was taking enormously long time.
Thus, a popular GPU cloud service was used for training model: paperspace.com
It is a paid service that charge USD 8 / month and provides 200 GB storage for a month.
On top of that, paid GPU was used that charges USD 0.51 / hour. Following are the offerings of this paid GPU:

Nvidia Quadro P4000 GPU – 8 GB GPU Memory
8 vCPU – 30 GB RAM

Training & Neural Networks Specific…

- Following is the plot of training loss:



Result

- Following are some results obtained from model:

six children sitting at their picture

father watching baby on the man

Result…

boy is sitting on the water

man with backpack and another on the railing in the background

Result…

- Trained model is tested on half of test images, i.e., 500 test images. Each of these 500 test images has 5 captions written by different people.
- Model generated a caption for each image.
- Since, it has become a kind of document where each reference has five texts, thus corpus bleu is used to measure the performance.
- Obtained Corpus BLEU Score is:

0.029157590163286912

Conclusion

- Model is predicting (or generating) “woman” many times because “0” was padded in each train caption to make its length equal to “max_caption_length”. We added “0” with thought that it means nothing, however in our vocabulary (vocabulary.txt), indexing has started from zero and the word at 0^{th} index is “woman”. Thus, for all those images where model is not getting anything, it is generating 0 as output and since “woman” is at 0^{th} index, therefore we are getting “woman” several times in some predictions.
- Since model is trained for less number of epochs and also on insufficient data, thus the BLEU Score is quite less on validation images. However, if we see model’s performance on a single image, then it seems that the model has learnt something during training and it verifies the technique used here for Image Captioning.

Future Work

- Remove the issue of many “woman” words in captions. This can be done by adding <UNK> (“unknown”) at 0^{th} index of vocabulary.
- Training on huge dataset, such MSCOCO, the largest open source dataset for Image Captioning, in order to improve BLEU Score of the model.

Template

- Template

Thank You