Image Captioning Using Classical Encoder / Decoder Approach

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 classic solution (i.e., encoder / decoder based approach) for Image Captioning is implemented. An advanced solution for Image Captioning is Attention Mechanism which 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.
- 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 goldfish.

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 Flickr8k dataset as this is sufficient to learn the implementation of Automatic Image Captioning.

Technology

- In this project, classic solution (i.e., simple encoder-decoder based approach) for Image Captioning is implemented.
- An advanced approach is also there, called as Attention Mechanism which is 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.

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”

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

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

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

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

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.

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

- 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:

```
1  import nltk.translate.bleu_score as bleu
2  bleu_sc = bleu.sentence_bleu(reference, candidate)
```

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.

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

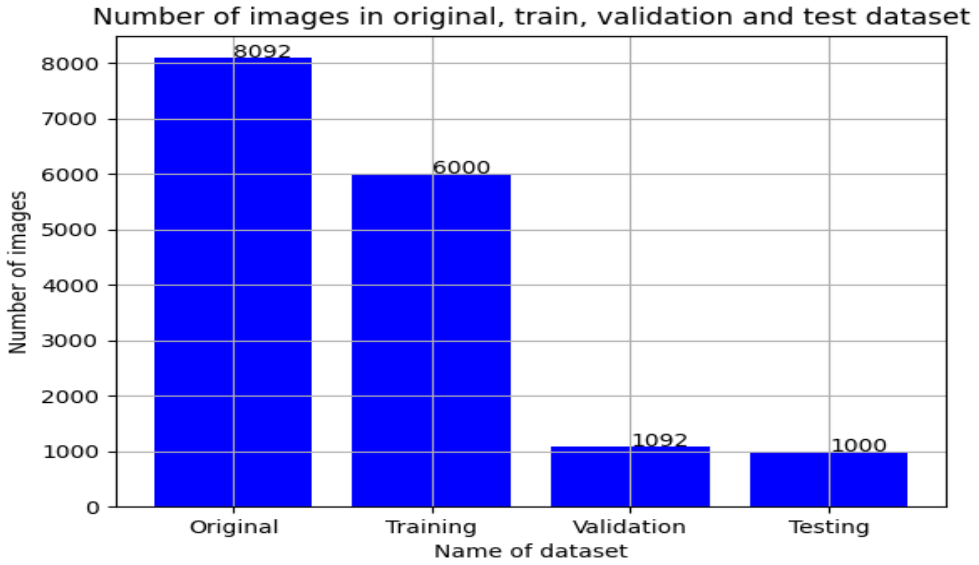
Where,

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.



Scripts Execution Flow…

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; and save it in train_image_caption.csv, val_image_caption.csv and test_image_caption.csv.
   ii. All these files (i.e., train_image_caption.csv, val_image_caption.csv and test_image_caption.csv) have two columns: “image” and “caption”.
   iii. All five caption of an image are in a single row corresponding to its image filename, separated by “<>”. These captions are yet not cleaned.
3. Script “scripts/preprocessing.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 put “startseq” and “endseq” before and after each processed caption, join them 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/gen_image_features.csv” does following tasks:
   i. This script loads each image which is in Flickr_8k.trainImages.txt, 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 2048. Script has done this task in chunks of size 1000, i.e., generated bottleneck feature and saved them with their image filenames for 1000 images. In this manner, it will save six csv files (because we have 6,000 file names in Flickr_8k.trainImages.txt file).
   iii. At last, we can run following code snippet to concatenate all six files generated in the last step:

```
import pandas as pd
data1 = pd.read_csv(“gen_image_vec_0_1000.csv”)
data2 = pd.read_csv(“gen_image_vec_1000_2000.csv”)
data3 = pd.read_csv(“gen_image_vec_2000_3000.csv”)
data4 = pd.read_csv(“gen_image_vec_3000_4000.csv”)
data5 = pd.read_csv(“gen_image_vec_4000_5000.csv”)
data6 = pd.read_csv(“gen_image_vec_5000_6000.csv”)
data = pd.concat([data1, data2, data3, data4, data5, data6]).reset_index(drop=True)
data.to_csv(“train_imagename_bottleneck_feat.csv”, index=False)
```

Scripts Execution Flow…

5. Script “scripts/training_GColab.py” does following tasks:
   i. Upload following files on Google Drive:

      i. train_image_caption_processed.csv

      ii. train_imagename_bottleneck_feat.csv

      iii. vocabulary.txt

      iv. max_caption_length.txt

      v. glove.6B.200d.txt

      All files mentioned above are generated in previous steps except “glove.6B.200d.txt”. It is a pre-trained model from NLP (Natural Language Processing) that we will discuss later.

   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.

   v. Script will save “wordtoix” and “ixtoword” as csv file for later use (when you will make inference without running this training script).

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.

Pre-Trained Models

- We are using two pre-trained models in this project:
  1. InceptionV3: To generate bottleneck features for images.
  2. GloVe: To generate word embeddings for captions.
- Inception V3:

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

Pre-Trained Models…

- GloVe (Global Vectors):
  - GloVe is a word vector technique and an Unsupervised Learning algorithm.
  - Word Vectors put words to a nice vector space where similar words cluster together and different words repet.
  - The advantage of GloVe is that unlike Word2Vec, GloVe doesn't rely just on local statistics (i.e., local context information of words), but it incorporates global statistics (word co-occurrence) to obtain word vectors.
  - Thus, GloVe captures both, global and local statistics of a corpus in order to come up with word vectors.
  - GloVe method is built on an important method: Co-occurrence Matrix.
  - Co-occurrence Matrix is very helpful to derive semantic relationships. Following is an example:

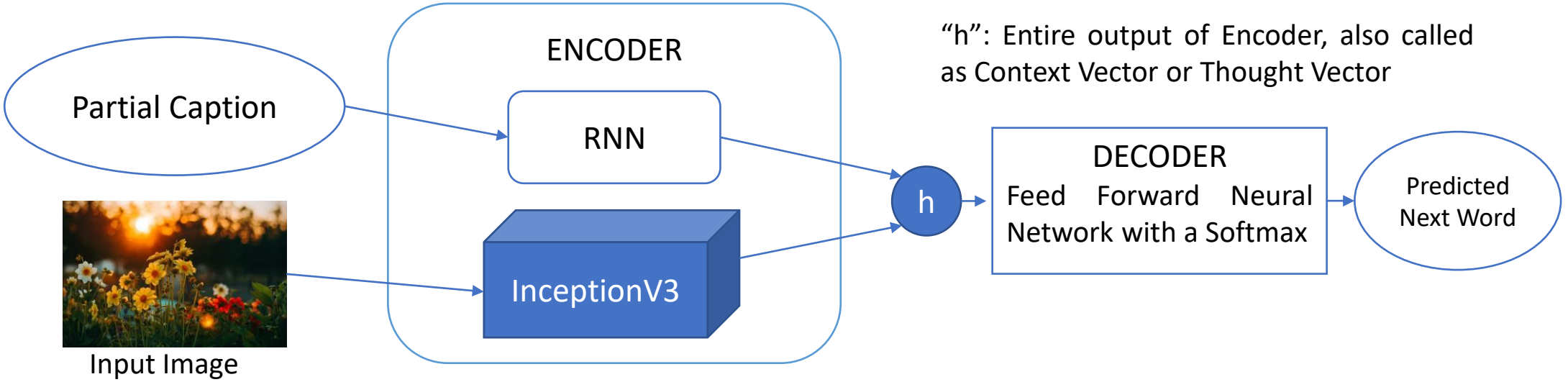
    "the cat sat on the mat"

    Co-occurrence Matrix is:

	"the"	"cat"	"sat"	"on"	"mat"
"the"	0	1	0	1	1
"cat"	1	0	1	0	0
"sat"	0	1	0	1	0
"on"	1	0	1	0	0
"mat"	1	0	0	0	0

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:
- Optimization Function: Adam
- Pre-trained models used: 1. InceptionV3 2. GloVe
- 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.

Template

- Template

Thank You