Visual Recognition
Mini-Project-2
Team ID 16

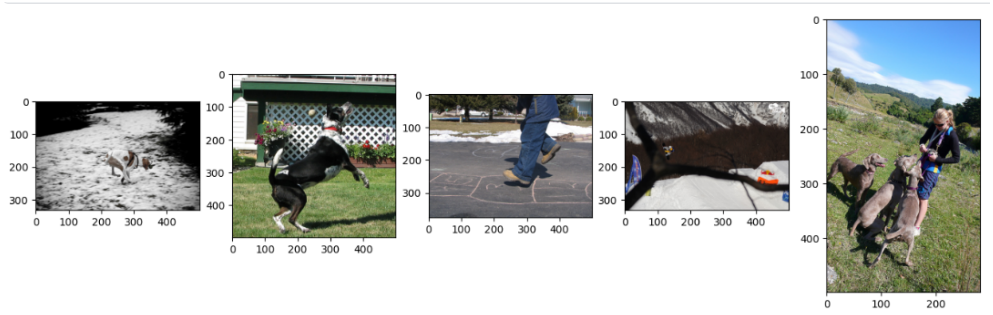
- Abhinav Mahajan, IMT2020553,
- Hardik Khandelwal, IMT2020509,
- Chaitanya Manas Cheedela, IMT2020053,
- Yash Koushik, IMT2020033

Image Captioning: -

Image Captioning, the stepping stone towards multimodal learning, incorporates both the concepts of Computer Vision as well as Natural Language Processing. In this report, we would go over the detailed summary of how we approached making a simple Baseline LSTM model which takes in Vision Embeddings and caption embeddings as input and generates captions, and then a modified model using Attention between the vision regions.

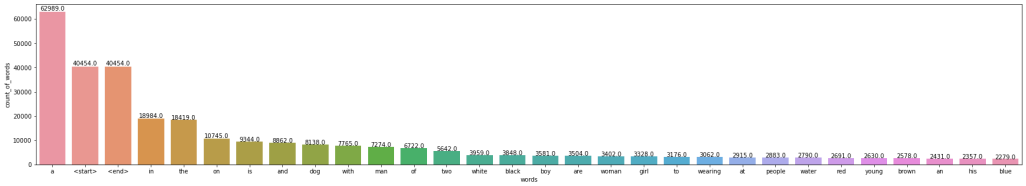
Approach: -

- For this project, we were given the Flickr8K dataset. It is small enough to be run on local machines but rich enough to cover various image-captioning scenarios.
- First step, as in any Machine Learning related project, was to perform EDA, Exploratory Data Analysis on it.
- The dataset consisted of a file with all the images, one file which consists of approximately 5 captions per image ID, and 3 files distributing the respective images for training, testing and validation.
- There was one file with lemmatised tokens, or captions as well, but it was not used as we got worse results with the lemmatized captions.
- During preliminary EDA, we notice a few things. First there are 5 image IDs in the tokens.txt file that do not exist. Secondly there is a null entry in the tokens.txt file. We are left with 8091 correct, valid photos, mapped to 40454, non-null captions.
- The caption and image lengths are imbalance, and varying. Care needs to be taken when encoding them to a vector. Sample pictures and captions, jumbled: -



```
1000268201_693b08cb0e.jpg#1     A girl going into a wooden building .
1000268201_693b08cb0e.jpg#2     A little girl climbing into a wooden playhouse .
1000268201_693b08cb0e.jpg#3     A little girl climbing the stairs to her playhouse .
1000268201_693b08cb0e.jpg#4     A little girl in a pink dress going into a woode
```

- Now, we first try to build a vocabulary on the entire corpus of data.
- We append all the captions into one list called annotations.
- We use the re library to split the lines to get the words to build the vocabulary.
- Since we are using GPUs, which have low RAM, care must be taken for vocabulary length.
- Therefore, we limit the vocabulary to the top 5000 words. And these are the top 30 words: -



- Now we perform pre-processing by using tf.keras.preprocessing, and we convert the text to lower case, remove special characters, our delimiter is ' ', introduce the 'unk' and 'pad' token for unknown and reserved tokens respectively and then perform tokenization immediately after vocabulary building.
- And as we said before, we want the same tokenized length of caption embeddings. Therefore, we calculate the max_length and then pad the remaining captions and store the tokenized vector in cap_vector. Here is an example of how we have stored our captions: -

```
    # Checking if the above process worked correctly or not.
    print(cap_vector[0])
    print(annotations[0])
```

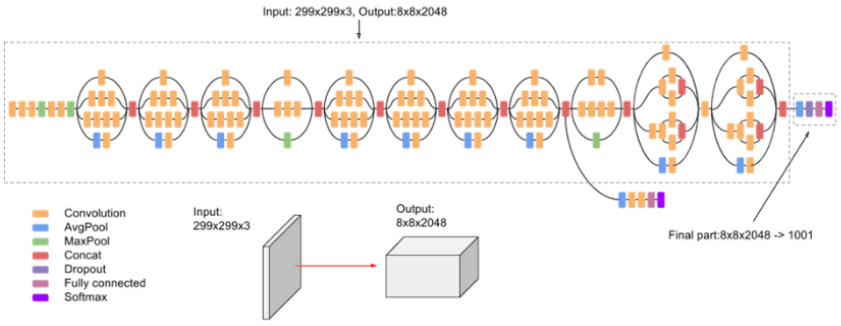
```
[  3   2  20 317  65   2 197 118   4   0   0   0   0   0   0   0   0   0
   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
   0   0   0]
<start> A girl going into a wooden building . <end>
```

- Now we move on to the images

- As shown before, our images are of varying height and width. Also all Vision Backbones use fixed size inputs.
- We have chosen to user InceptionV3 as our Vision Embedding pre-trained model, and we will justify its usage later but for the time being, it requires an input image of 299*299*3 dimensions, and keeping the pixel aspect ratio in mind, we resize all our images to the same.
- Also, another specification of InceptionV3 model is that the RGB inputs are normalized to the range of [-1, 1], therefore we took care of that as well.
- So far we have seen all the common aspects of the Baseline and improved model. This is purposely done so that we will have minimal differences in code between the two models.

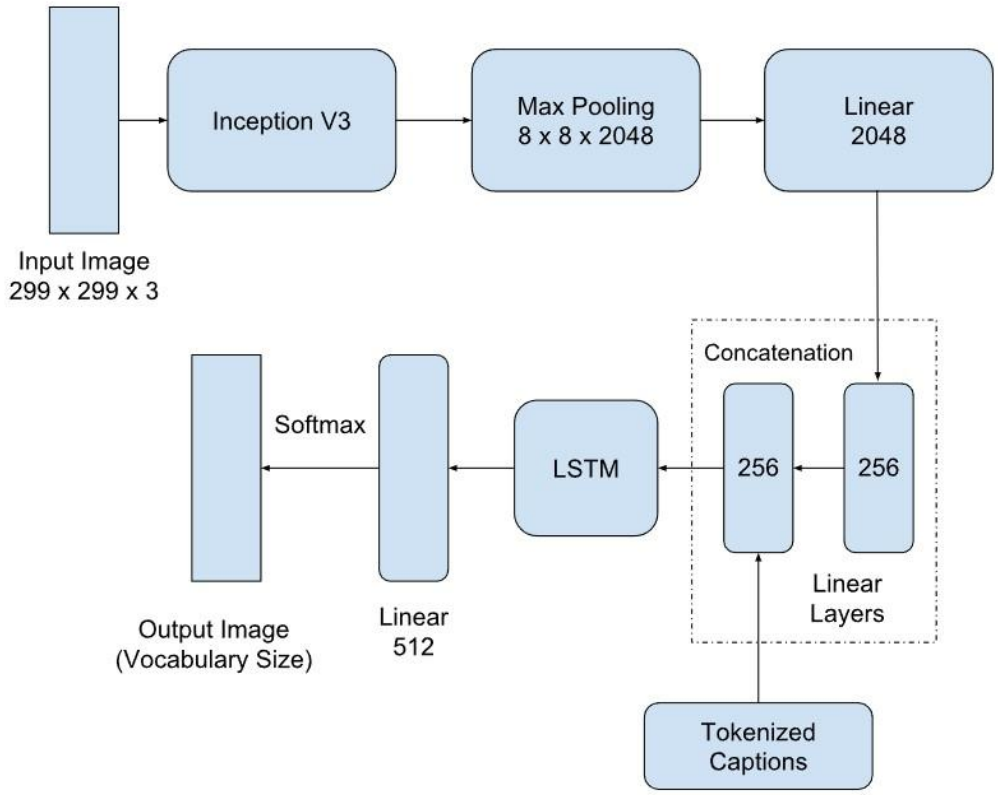
Baseline Model: -

- Firstly let us answer why we are using InceptionV3, it is noteworthy that VGG16, VGG19, ResNet would have given equally good encodings, but the inception model is lighter due to the addition of inception layers, which by-pass the dense heavy connections but provide just as rich a representation.
- Anyway, we now make use of batches, and in batches we send in chunks of images (batch size has been set as 64), and they get converted to a (64, 2048) representation. Where 64 is the batch size and 2048 is second last fully connected layer of InceptionV3.



- Also, we make use of a dictionary/pickle file which stores the embeddings, so that we don't have to get the embeddings again and again during training. This also helps in preventing RAM exhaustion.

Architecture: -



- As we can see, the encoder is nothing but a fully connected layer, (2048, 256). The input image is mapped to its Inception Encoding, then sent to it.
- The Decoder is also very simple. We concatenate the 256 length Vision Embedding with the 256 length image caption embedding. (we embed the tokenized caption)
- Also it is noteworthy that we have tried to concatenate the 5 captions per image and use it as one input. But we had better results when we treated each image-caption pair separately, without concatenating.
- We also tried using GRUs instead of LSTMs, and both gave equivalent results. This is not surprising as GRUs are also considered when we have small datasets.

Training Details: -

- The loss function used was Cross entropy loss, after sending an image and its caption, we are able to generate an output caption, and each word corresponds to each output generated by the LSTM at each time instant.
- During evaluation, we don't send the image caption, but just the image, and later again, we generate the predicted output using the GREEDY algorithm and then perform the BLEU alignment or METEOR single reference single hypothesis alignment and get the score.
- For evaluation we do this over the entire test set defined and average out the BLEU AND METEOR scores.

- We can perform inference in the wild by this method by this method as well.

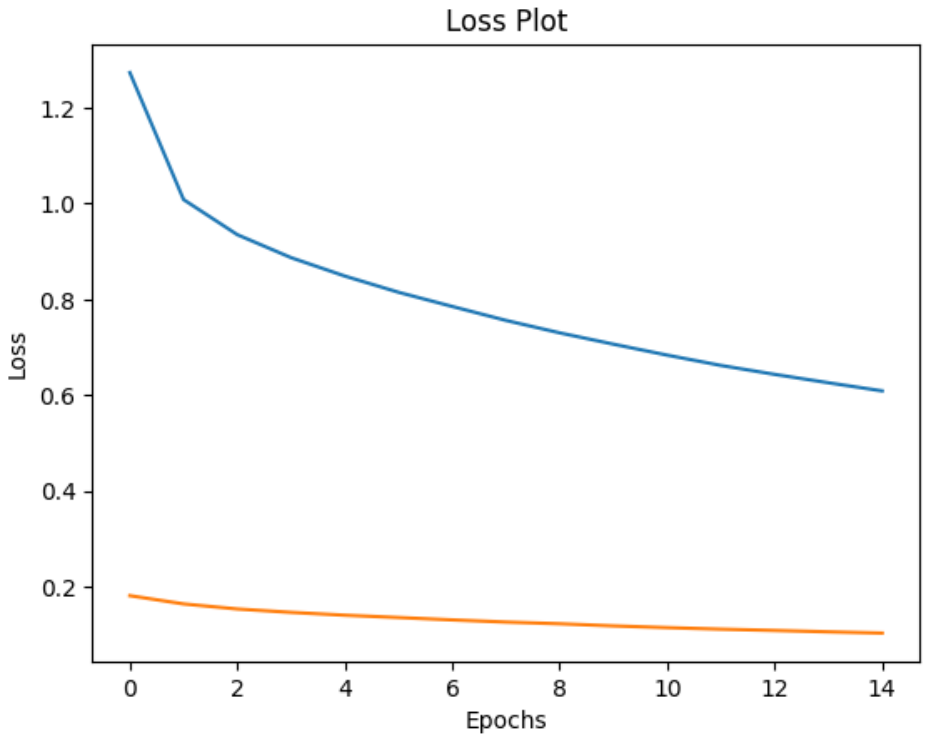
Results: -

- We received an average testing BLEU score of 20.2 and average testing SINGLE-SHOT-METEOR score of 23.6.

```
100%|██████████| 5000/5000 [24:04<00:00,  3.46it/s]
```

```
[60]
    avg_bleu = avg_bleu / num_test_cases
    print("Average BLEU score:", avg_bleu * 100)
    avg_meteor = avg_meteor / num_test_cases
    print("Average METEOR score:", avg_meteor*100)
```

```
...  Average BLEU score: 20.257878142459678
     Average METEOR score: 23.62070699678795
```



- Above is the train-test loss curve during training.

- Above is a sample test case to which the predicted and actual labels are: -

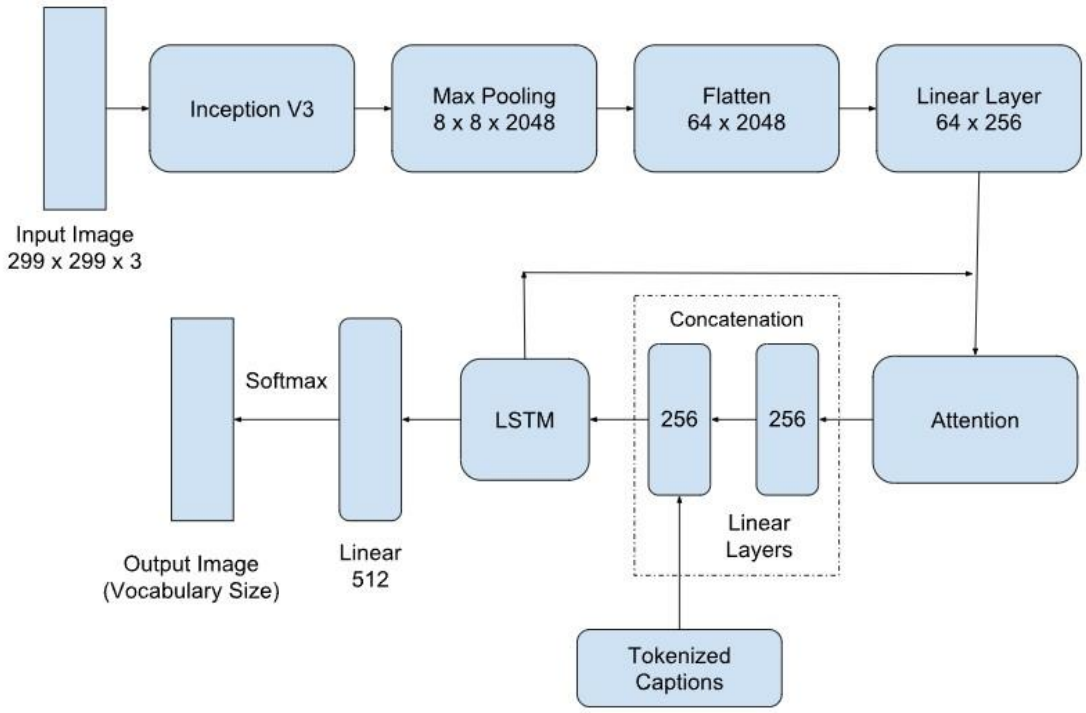
```
BELU score: 18.887560283756187
METEOR score:  39.25925925925926
Real Caption: a man struggles to climb a rock wall
Prediction Caption: a rock wall
```

- Bear in mind the above BLEU and METEOR score correspond just to the 1 sample test image predicted caption.

Modified (Attention): -

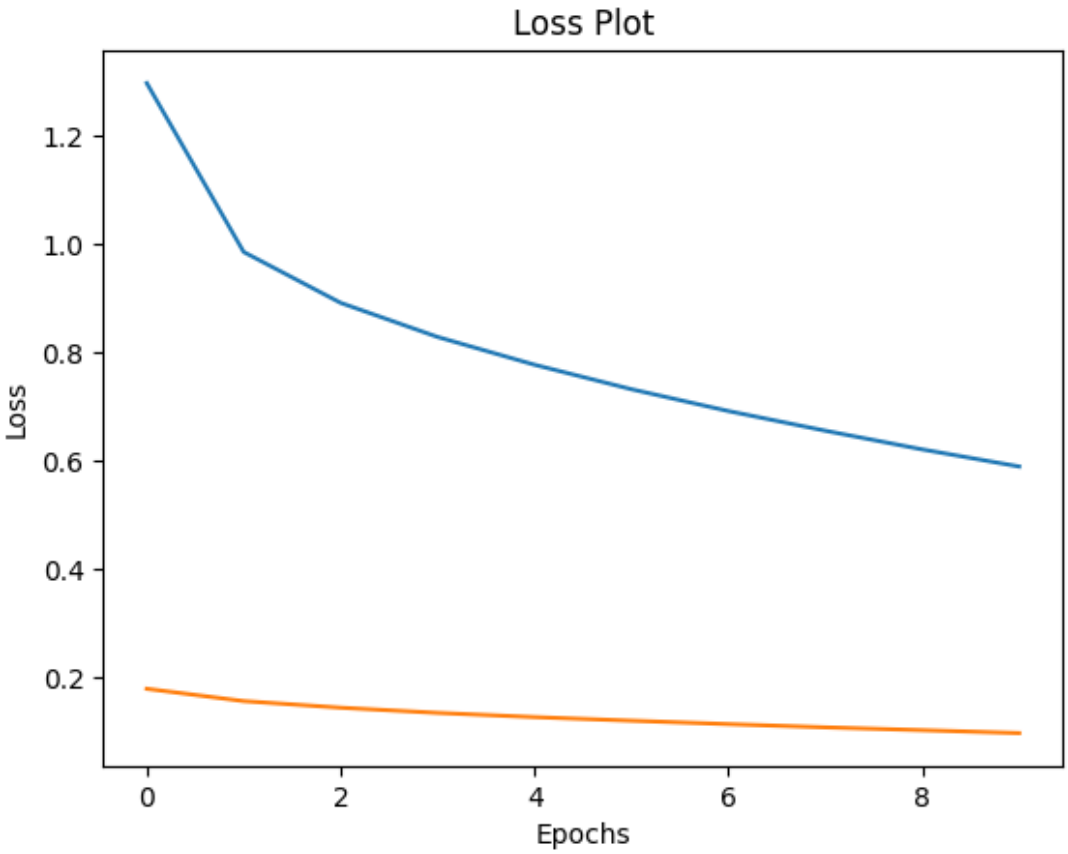
- Most of the code and approach remains the same, I will just mention the changes. The point was to make minimal changes in not just the architecture, but the code as well to see the impact of attention. All the other hyperparameters other than the ones mentioned remain the same.

Modified Architecture: -



- As you can see, our encoder architecture changes slightly, now our encodings will be slightly different. This time we don't use the fully connected layer of InceptionV3, we stop at the (8,8,2048) embeddings.
- The above is evaluated and stored in a pickle file. You can find it in my google_drive_link here: - https://drive.google.com/file/d/1YwbKvkGDqaETihd2bwqoVA3K0Pt_HN29/view?usp=share_link
- We flatten it to (64, 2048) where 64 represents the different pixel areas, and we will perform attention across these pixel level representations only.
- Bear in mind, when we send this in batches, it will get scaled correspondingly in that dimension.
- We project each of those sixty four, 2048 "feature vectors" into an embedding dimension of 256.
- Our effective embeddings would be (batch_size, 64, 256)
- Then we perform attention on it with the hidden state vector, we get the attention weights and the context vector is nothing but the weighted sum of the attention weights and the 64 pixel locations.
- Therefore, we will end up with a (batch_size, 256) embedding, which is then combined with the language embeddings as in the previous case and the remaining decoding LSTM is the same.

Results: -



- The average testing BLEU score was 25.4 and the average testing METEOR score was 25.6. In a lesser number of epochs, we are able to outperform our baseline comfortably. The only reason we had to limit our epochs was because of RAM exhaustion.

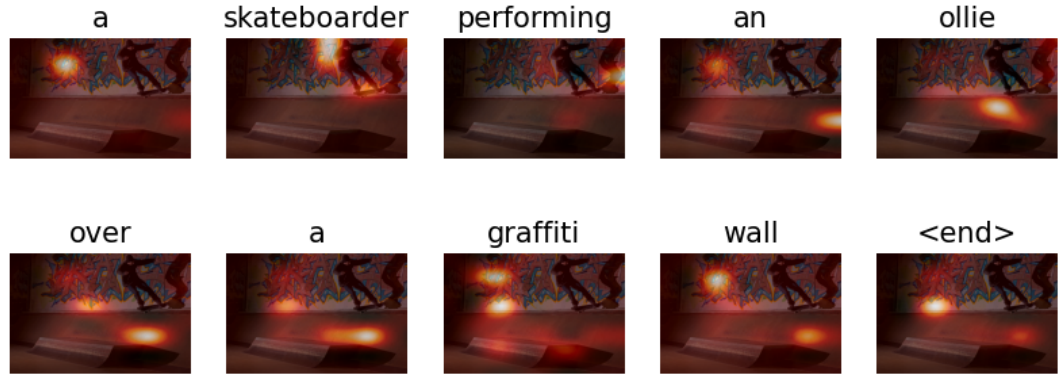
```
    avg_bleu = avg_bleu / num_test_cases
    print("Average BLEU score:", avg_bleu * 100)
    avg_meteor = avg_meteor / num_test_cases
    print("Average METEOR score:", avg_meteor*100)
```

[65]

```
Average BLEU score: 25.41718546822364
Average METEOR score: 25.67805187846878
```

- In addition to sampling in the wild, we can also visualize the attention locations: -

```
BELU score: 31.875190040968487
METEOR score:  33.037037037037045
Real Caption: a skateboarder is jumping through the air in front of a graffiti covered wall
Prediction Caption: a skateboarder performing an ollie over a graffiti wall
```



- Again, keep in mind the picture shows only the BLEU and METEOR score of that particular inference image.