Aim of the Project

To design a CNN-LSTM system that can perform image captioning based on Flickr 8K dataset.

About the Dataset

A benchmark collection for sentence-based image description and search, consisting of 8,000 images that are each paired with five different captions which provide clear descriptions of the salient entities and events. The images were chosen from six different Flickr groups, and tend not to contain any well-known people or locations, but were manually selected to depict a variety of scenes and situations.

Data Preprocessing

Image Processing Steps

Since we have used Resnet 50 CNN architecture to extract visual features from the images, hence we need to perform some basic transformations to the images before giving them as input to our architecture, like :

- Resize to 224×224
- We had tried to do color standardization as done in detection tasks but we didn't see any difference. So to increase the speed of training we removed this. (We have implemented all transformations and pre-processing in the data loader to prevent ram crashes)

Dataset Building

More or less, every image from the dataset has 5 captions associated with it. We have used a dataframe to store the filename of the image and the corresponding caption. This was replicated for all three train, test, and eval set given to us in the dataset.

Building Vocabulary

We build a dictionary for all the words present in the dataset for each train, test, and eval separately. We store 2 maps where each word is given an index and the reverse mapping corresponding to each index. This helps to create some sort of label encoding. However, in addition to the words present in the dataset we have introduced 4 different tokens:

- <**PAD**> Used for padding the embeddings
- <**SOS**> Start of string
- <**EOS**> End of string
- <**UNK**> Unknown token

Writing Custom Pytorch Dataset

As the data consist of 2 modalities: Vision and Language, traditional methods of handling datasets didn't work well with frequent crashing of the system due to ram issues. The Pytorch Dataset module applies the image transformations and language processing step of creating the vocabulary efficiently, preventing any crashes.

Approach

Our approach comprises 4 permutations:

1. CNN + LSTM Baseline with ResNet50 CNN and Label encoded text data.
2. CNN + LSTM with Attention with ResNet50 CNN and Label Encoded data
3. CNN + LSTM Baseline with ResNet50 CNN and Glove embedding text data.
4. CNN + LSTM with Attention with ResNet50 CNN and Glove embedding text data.

Architecture Details

In this section, we discuss the building blocks used to make the architectures stated above

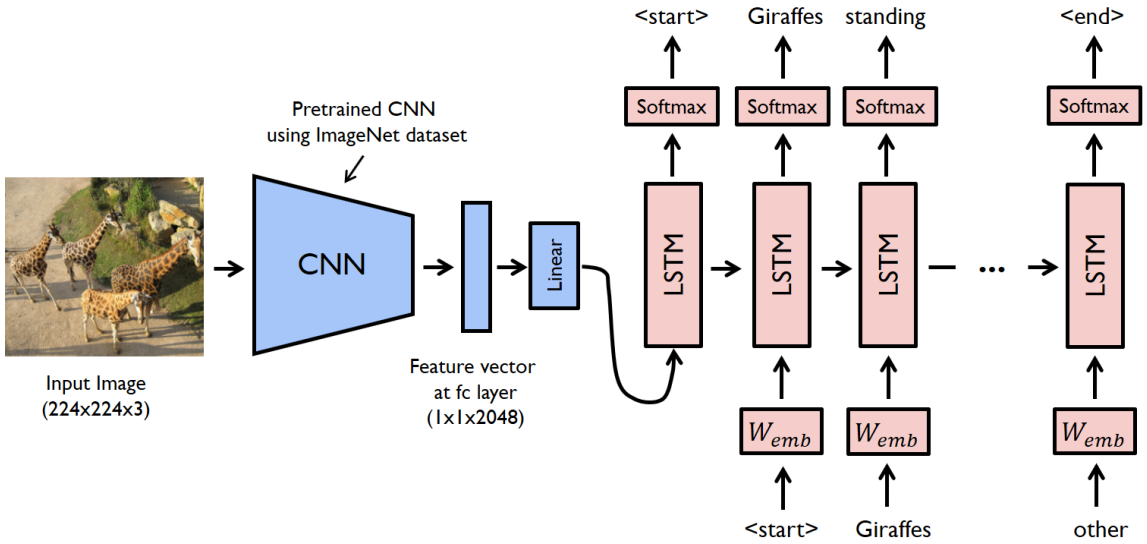

Figure 1: Representation of the model architecture

Encoder CNN

As a feature extractor, we have used ResNet50 which helps us to get the visual features for input images of size 224x224x3. We used Resnet 50 for the following reasons:

1. Depth: ResNet-50 is a deep neural network with 50 layers, which allows it to capture more complex features from images. This depth enables ResNet-50 to learn hierarchical representations of images.

2. Pre-trained weights: Using pre-trained weights as a feature extractor can help to improve the accuracy and efficiency of a model.

3. Performance: The forward pass per image using resnet is less than other networks like inceptionv3 and VGG and hence was a clear choice as we had limited time to train the model and couldn't allocate much time for the forward pass of the CNN.

Decoder LSTM architecture

At first, we initiate the embeddings using nn.Embedding to convert our label-encoded captions to Pytorch-compatible embeddings as shown in class.
This is followed by a torch.LSTM which takes in the concatenation of both the vision and language embeddings. After this, it is passed via a linear layer. We applied dropout but it deteriorated the performance and hence we didn't use it.

Attention Mechanism

The Attention module in code defines three linear layers: W, which maps the decoder hidden state to the attention vector, U, which maps the encoder output to the attention vector, and A, which maps the combined attention vector to a scalar score. This is similar to Query, Key and Value learnt in class. The algorithm is specifically called as the Bahdanau Attention and is as follows[1]:

1. The encoder generates a set of annotations, h_i, from the input sentence.

2. These annotations are fed to an alignment model and the previous hidden decoder state. The alignment model uses this information to generate the attention scores, $e_{t,i}$.

3. A softmax function is applied to the attention scores, effectively normalizing them into weight values, $\alpha_{t,i}$, in a range between 0 and 1.

4. Together with the previously computed annotations, these weights are used to generate a context vector, c_t , through a weighted sum of the annotations. That is:

$$c_t = \Sigma_{i=1}^{T} \alpha_{t,i} h_i$$

5. The context vector is fed to the decoder together with the previous hidden decoder state and the previous output to compute the final output, y_t.

6. Steps 2-6 are repeated until the end of the sequence.

Changes in Decoder LSTM for incorporating Attention

Since upon adding attention and passing it with hidden state information we need much more access to what goes inside the Pytorch LSTM module. Hence we had to switch from a Pytorch LSTM to Pytorch LSTM cell which allowed us to get the required states as well as set those states based on the attention computation. Now the lstm input is basically the context vector along with image embedding concatenated together. Earlier this was only the embeddings (both vision and language) put together but now context calculated by the Attention modules is also used.

Model Flow

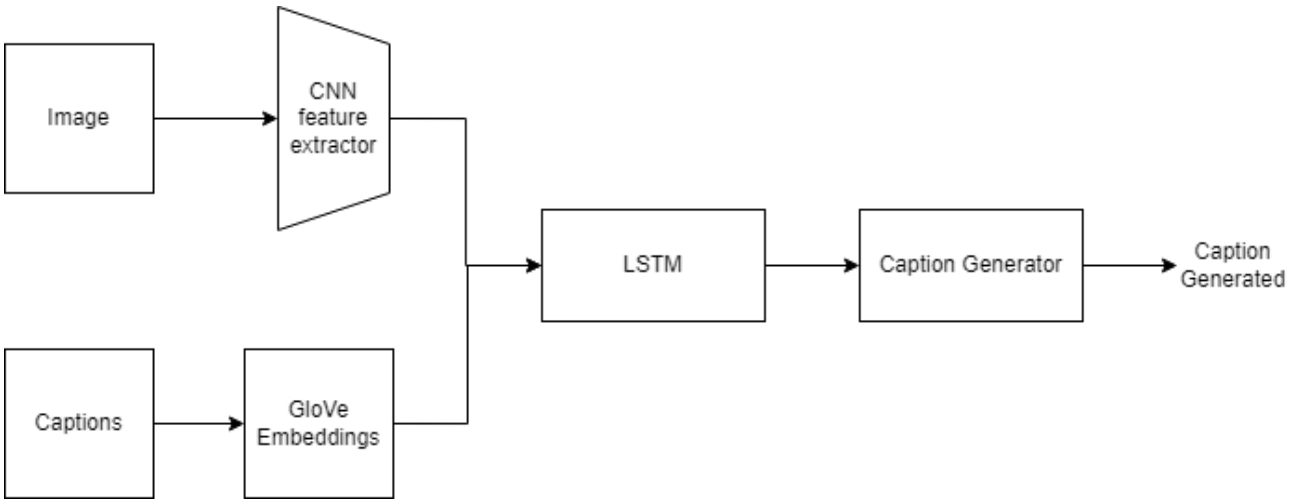

Figure 2: Representation of the data flow in the model for approaches 2 and 4

Our encoder architecture consists of ResNet as a feature vector which takes an image as an input of size 3 × 224 × 224 and passes the features thus extracted to a nn.Linear layer to obtain image embeddings, which is consequently passed to the decoder layer.

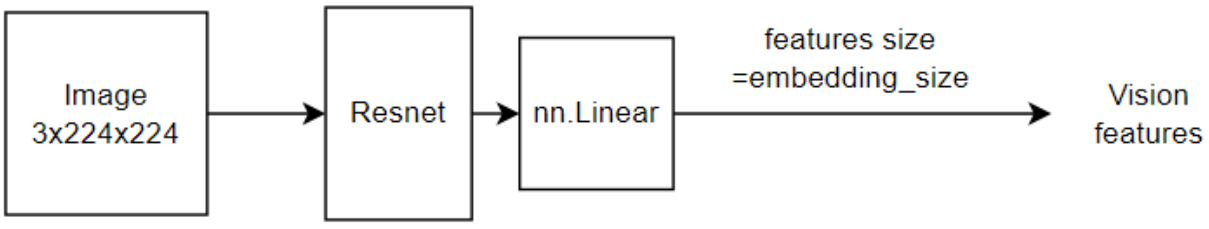

Figure 3: Representation of the Encoder layer

The Decoder layer consists of an LSTM which takes concatenation of the vision embeddings and the language embeddings, which is further passed through an nn.Linear layer.

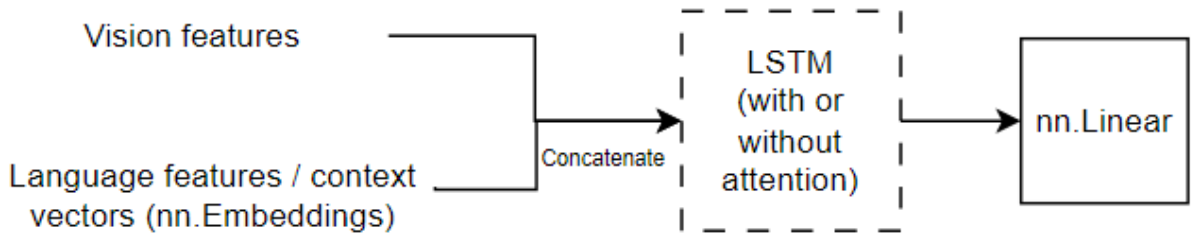

Figure 4: Representation of the Decoder layer

Hyper-parameters used for each approach

Approach 1 and 3

1. Size of text embeddings = 400 (in case of Approach 1, else it's 200 based on GloVe)
2. Hidden Layer Size = 512
3. Number of LSTM layers = 2
4. Optimizer = Adam
5. Learning rate = $1*10^{-4}$
6. Number of Epochs = 30

Approach 2 and 4

1. Size of text embeddings = 400 (in case of Approach 2, else it's 200 based on GloVe)
2. Attention Dimension = 256
3. Hidden Layer Size = 512
4. Number of LSTM cells = 1. This was done so to keep the train time comparable with the other approach. Using 2 attention layers increased the training time significantly and we timed out on Kaggle as well as Google Colab.
5. Optimizer = Adam
6. Learning rate = $1*10^{-4}$
7. Number of Epochs = 30

Evaluation Metric

The two metrics mentioned to us were the BLEU score and METEOR Score

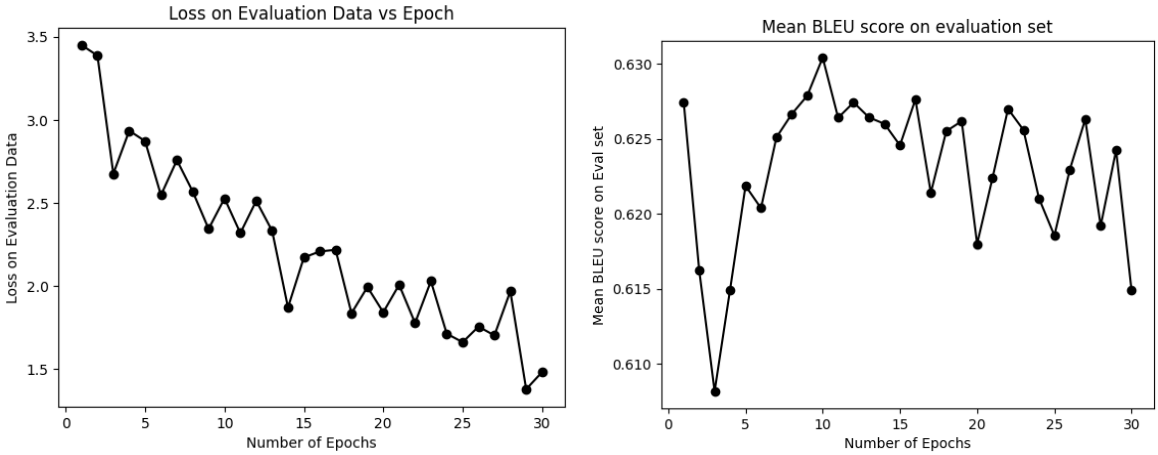

Figure 5: (Left) Loss on Evaluation Data with each epoch and (Right) Mean BLEU score on the evaluation set in each epoch for Approach 2

Bleu Score

BLEU stands for bilingual evaluation understudy in its full form. BLEU will generate a number between 0 and 1. The score represents how similar the given text is to the reference text, with values closer to 1 indicating more comparable texts. In actuality, a perfect score is impossible to get because a translation must precisely match the reference. Human translators are incapable of accomplishing this. The sentence-bleu() function in NLTK is used to compare a candidate sentence to one or more reference sentences.
However, on NLTK Sentence Bleu [4] by default gives BLEU-4 with weights = (0.25,0.25,0.25,0.25) and we have considered that to be our metric. There is also a corpus bleu available that rates the corpus but for our task, it didn't hold much relevance and we felt that sentence bleu can give a much more accurate view of how our model is performing. [5]

Meteor Score

The Meteor score [4] is based on a combination of precision, recall, and unigram matching between the machine-translated output and a set of reference translations. It also takes into account semantic and syntactic information by using WordNet-based synonymy and paraphrase matching. The Meteor score ranges from 0 to 1, with higher scores indicating better machine translation quality.
Since the meteor score uses unigram we can already foresee that even for a good BLEU-4 score (default for sentence BLEU), the Meteor score will be very low as compared to it and later we see that, that is exactly what we observed.

Results

Conditions	**METEOR**	**BLEU 1**	**BLEU 2**	**BLEU 3**	**BLEU 4**
Without attention + Label Encoding	0.0802	0.0923	0.2906	0.4337	0.5247
With attention + Label Encoding	0.1202	0.1014	0.3026	0.4443	0.5335
Without attention + GloVe	0.1196	0.0932	0.2906	0.4328	0.5231
With attention + GloVe	0.1203	0.1005	0.3029	0.4459	0.5359

Table 1: BLEU and METEOR scores for various approaches tried

From the above observation table, we can clearly see that approach 2 works better than approach 1, which makes sense, since attention helps the model to focus on more relevant features before making any prediction, which in turn improves the BLEU as well as METEOR metrics.

From the table we can also see that approach 3 is almost similar to approaches 1 and 2 (if not better), which implies that GloVe embeddings are better suited for our use case.

As from the above table, we can safely conclude that the architecture works best when attention is used and pre-trained GloVe embeddings are used.

Analysis and some insights on the result

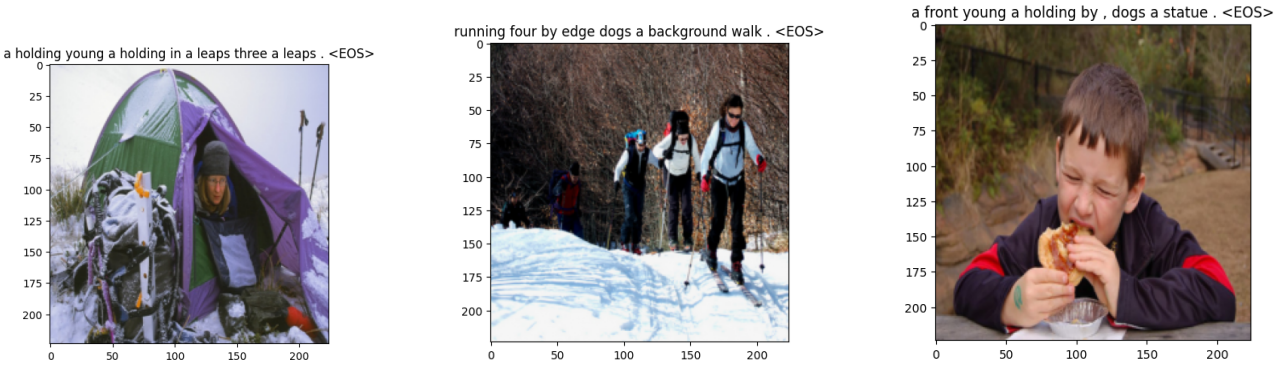

Figure 6: Example outputs from our model

Why are the captions generated by our CNN-LSTM model not perfect?

There can be a few possible reasons for this:

1. Small dataset: It may be the case that 6,000 training samples are not enough for our model to learn the associations between the image and captions. We can try a more extensive dataset to avoid this problem.

2. Fewer epochs: We notice in the loss plot above itself that the loss is still decreasing on the eval data when we stop it in 30 epochs. Therefore it may be the case that 30 epochs are not enough for the model to learn the relationships between images and captions. Maybe, we need to train the model for more epochs like 70 or 100.

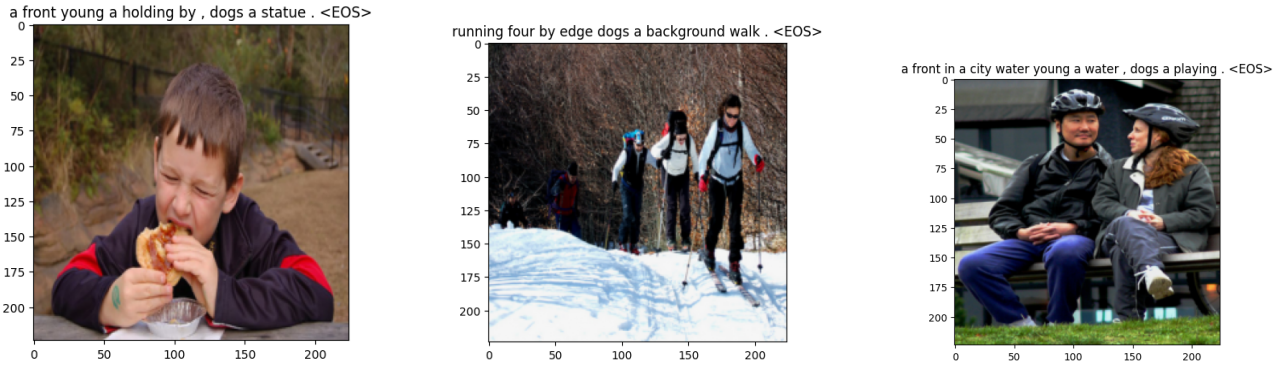

Figure 7: We notice that our model is wrongly predicting the word 'dog' for many images

Why is our model wrongly predicting the word 'dog' for many images? The notion of language bias and "mode collapse"

The phenomenon when our model keeps predicting the same word for most images is called "mode collapse"[2]. It occurs when the model outputs the same word regardless of what input is given to it. This issue can arise due to various factors such as inadequate training data, limitations in the model architecture, or insufficient diversity in the training examples.
It may be the case that the dataset contains too many captions containing the word 'dog' and thus our model thinks this word is too common and thus predicts it for most inputs. There are certain ways in which we can overcome this problem:

- Increasing training data: Expanding the dataset used for training can provide the model with a broader range of examples, helping to capture more diverse image-caption relationships.
- Data augmentation: Applying techniques such as image transformations, cropping, or adding noise to the training data can introduce additional variations, which may help the model generate more diverse captions.
- Architecture modifications: Using more advanced models like Transformer-based architectures, can potentially improve the model's ability to generate diverse and contextually relevant captions.
- Regularization techniques: Applying regularization techniques like dropout, batch normalization, or adding noise to the model's parameters during training can help reduce overfitting and encourage the model to explore different captioning options.

In the case of training the model with label encoding, we also see the instance of this with the case of Woman repeatedly occurring. This is due to the high frequency of this word in the dictionary.

Attention maps

We have plotted attention maps for our models to gain a better understanding of the working of our model. The plots for approaches 2 and 4 are as follows:

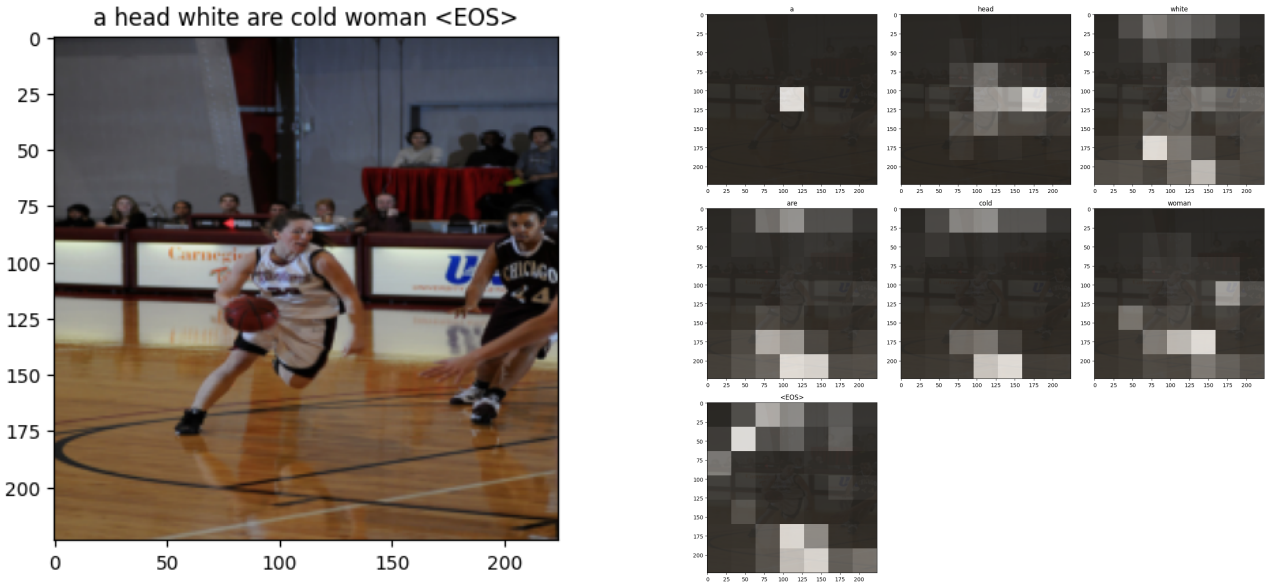

Figure 8: Attention map over an example image with Label Encoding

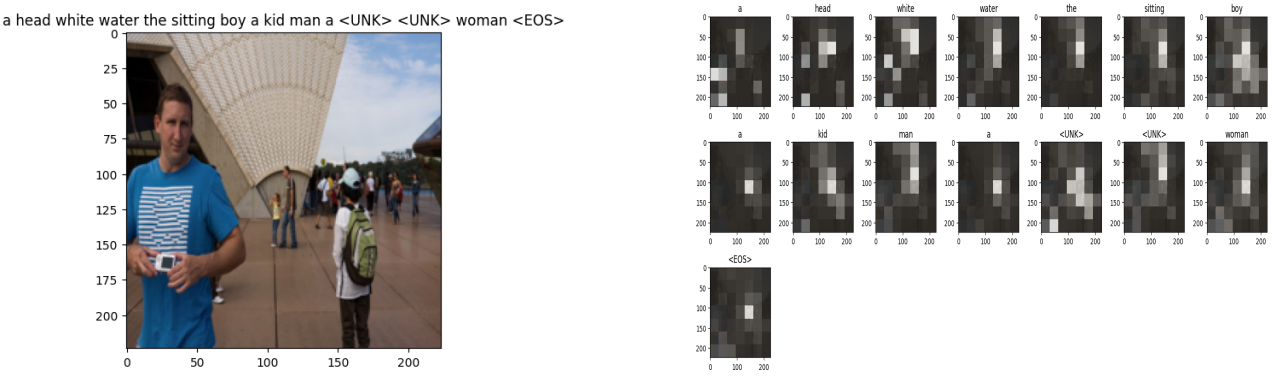

Figure 9: Attention map over an example image with Label Encoding

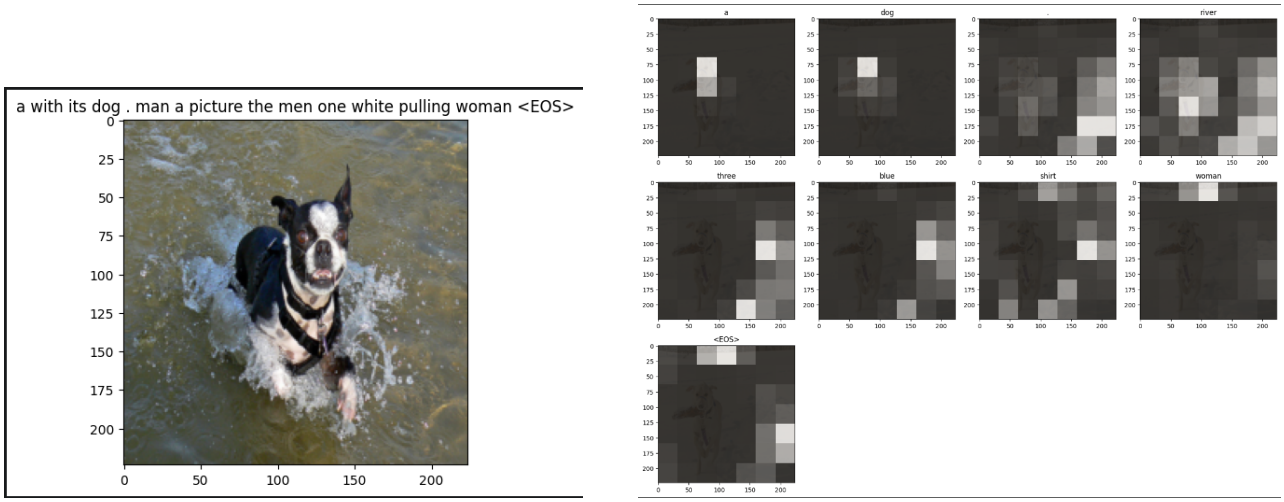

Figure 10: Attention map over an example image with GloVe encoding

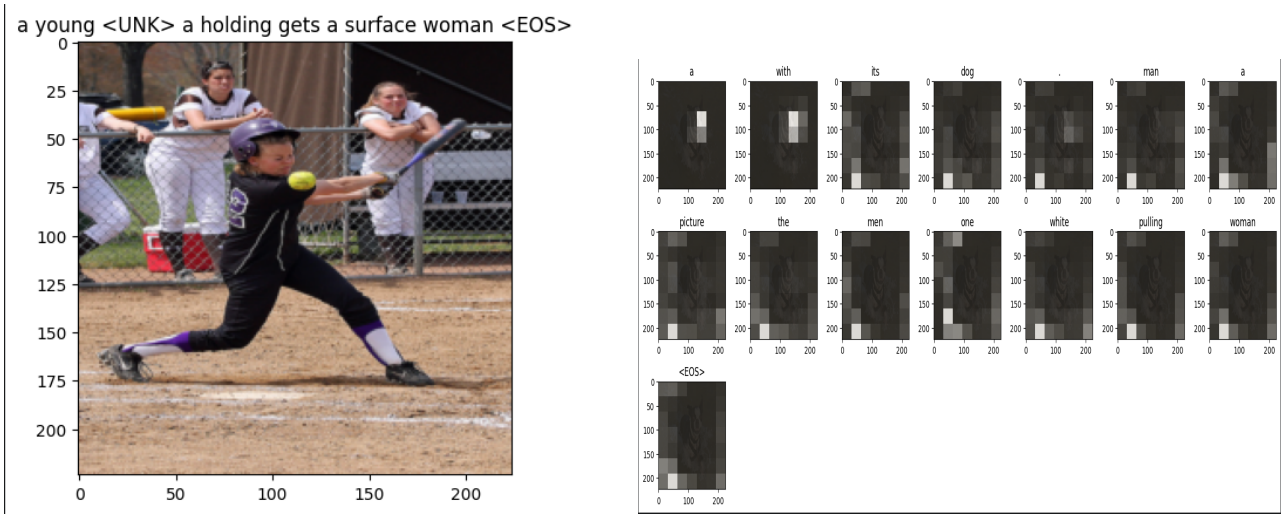

Figure 11: Attention map over an example image with GloVe encoding

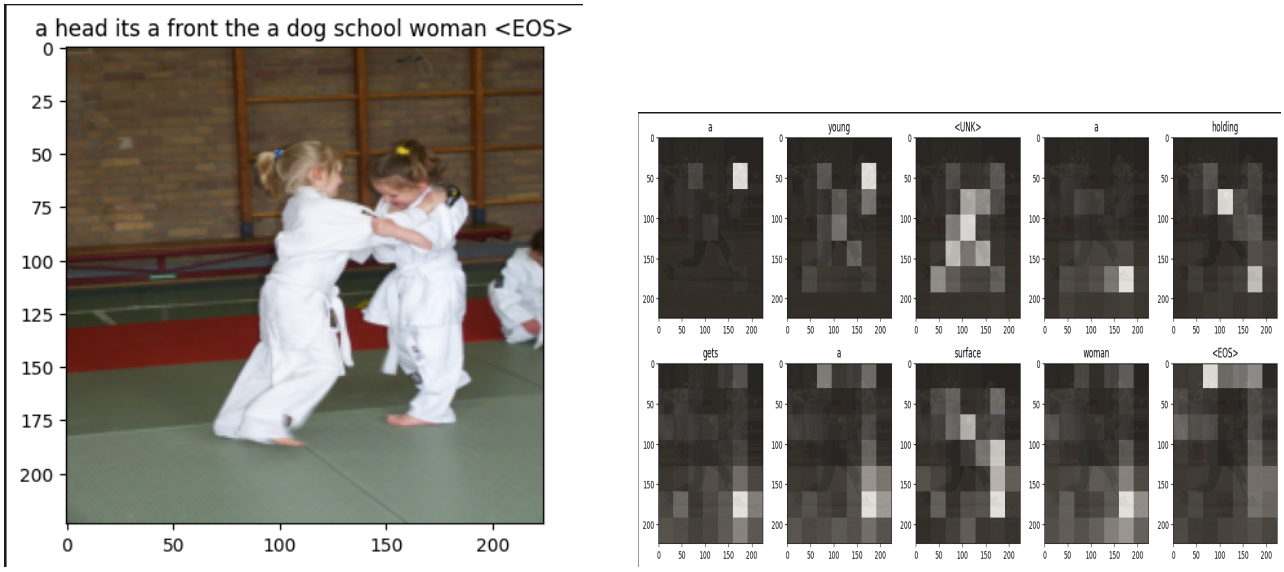

Figure 12: Attention map over an example image with GloVe encoding

Further improvements

Following can be steps to further improve our model:

1. We can train the model for higher epochs
2. Use other neural networks for visual feature extraction, like VGGNet, Inception net, etc., and compare with respect to ResNet in terms of training time as well as a metric value.
3. Try Word2Vec, and BERT to generate word embeddings and see how the performance improves.
4. In the Glove embedding model we replace the weights in nn.Embedding directly and using it as it is. We can also make those ready-made weights as trainable parameters (by setting their requires_grad as true) so that they are also part of the compute graph during backpropagation and be fine-tuned based on the dataset.

Our **Kaggle notebook** is public and can be found here

References

Attention Mechanism from here

Useful article on Mode Collapse

A useful Medium article which helped us to get started:

NLTK docs on Sentence Bleu and Meteor Score

Stackoverflow link to difference between sentence and corpus bleu.