Assignment 4: Image Captioning in Italian

Team: Inception RSA

Ayushi Agarwal(2018ANZ8503)
Ramya Hebbalaguppe(2019ANZ8720)

September 8, 2020

1 Introduction

In this assignment, we will develop an Image-Captioning Machine Learning Model to generate captions in Italian for a given set of images. We will develop a model based on the Encoder-Decoder Architecture that can take an image and directly generate a sentence description of the image. The model that we have implemented is based on a combination of a Convolutional Neural Network (CNN - working as an image encoder) and a Recurrent Neural Network (RNN - that functions as a image decoder).

The encoder would sample the salient features or likely descriptions from images by a simple feed-forward process to cull out of clutter in an image. Subsequently, the decoder uses this information and is conditioned to understand the important descriptions of the image and tries to learn how to describe what the model "sees". It is interesting to visualize what the model "sees" - there are two mechanisms - hard and soft attention. Our work focuses on implementation of soft attention because and the preference is because it discredits irrelevant areas by multiply the corresponding features map with a low weight. That is instead of using the image as an input to the LSTM, we input weighted image features accounted for attention.

The key difference between a LSTM model and the one with attention is that "attention" pays attention to particular areas or objects rather than treating the whole image equally.

The assignment is broken down into two parts:

- **Non-Competitive part**: The model has to be developed from scratch and trained on the given dataset - Images and 5 captions per image.
  - **Encoder**: We will design a CNN based Encoder that handles variable sized images.
  - **Decoder**: We will design a LSTM based Decoder for generating captions given for the image input.
  - **Implementation Details**: We will use Cross Entropy Loss and teacher-forcing for training the decoder. For inference, we have implemented Greedy Search

(which generates the token with maximum probability at every step) and Beam Search (which uses a few top token with highest probabilities to construct a tree of possibilities and generates the most likely captions for the images. We will describe more about Beam Search.

  - **Evaluation**: For evaluation, we have used the SacreBleu score [1]. This scores is an aggregated score of all n-gram Bleu scores and also checks for brevity penalty.

- **Competitive Part:**

  - In this part, we will be using pretrained CNN models for Image classification. Instead of using the final Softmax layer for classification, we will directly take the output of the pooling layer of the ResNet that is image feature representations and we will feed into the Decoder Network.
  - We will use Attention Module for developing our Decoder.
  - We will generate one caption per image on the test data to test our model scores using **SacreBleu scores**.

2 Non Competitive Part

In the non-competitive part we implement both VGG-19 and ResNet-50 architectures implemented from first principle for the image encoder part. We observe that VGG-19 yields a better SacreBleu score in comparison to ResNet-50. Further, for the image decoder part, we employ LSTM architecture. We will discuss about the architecture in more details in this section.

2.1 Data Pre-processing

The performance of every model depends on the initial stage of data pre-processing. The dataset includes a set of images and for each image there are 5 golden captions that best describe the image. One train data point (1 image, 5 captions) in the dataset has been converted to 5 separate data points (5 replicated images, 5 captions). This will help train the decoder with every available golden caption. The dataset is creating by querying through the caption image dictionary. The data is loaded in batches by using the PyTorch DataLoader library.

Image Processing: In the Non-Competitive part, all the images in the dataset have been re-scaled to (112, 112) using random cropping – random cropping aid in regularissation and solved the memory issues, thus yielding gave better results. We have normalized the image to the standard ImageNet dataset convention thinking that the images would be similar to ImageNet given the huge dataset size, variability, and close to the wild setting. And to our expectation, normalized images performed better by 1 point on the SacreBleu score when compared to the same setting without normalisation. Normalisation would scale our input training vectors as the ranges of our training distributions of feature values would likely be different for each feature, and thus the learning rate would cause corrections in each dimension that would differ in range from one another.

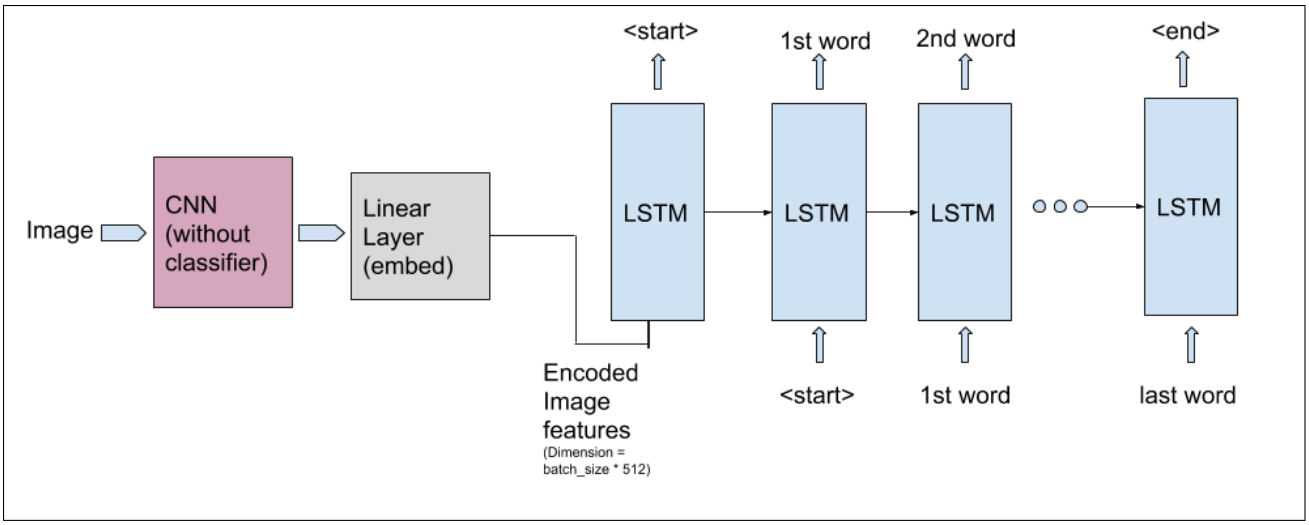

Figure 1: Non-competitive part: Proposed Image Captioning Pipeline

Caption Processing: We have not performed caption processing in our final results. **The removal of punctuation and caption processing gives better bleu score on public test data with the golden captions also pre-processed. But since it gives lower bleu score in general, we used whole vocabulary.** In an image captioning problem (variable length captions setting), the RNN has to output sentences of variable lengths so it needs a *start* and *end* token to recognize the starting and ending of the caption. So these special tokens are appended to each training caption at first and end respectively. Every caption in a batch is padded to the length of the caption with maximum length only in that batch because the LSTM only needs to learn batch-wise.

2.2 Proposed Architecture

Figure 1 summarises the approach we have implemented for image captioning that comprises of image encoder and caption decoder. The input is an image, and the output is a sentence describing the content of the image. It uses a convolutional neural network to extract visual features from the image, and uses a LSTM recurrent neural network to decode these features into a sentence. The CNN is used without the classifier layer, and instead of classifying we send the output feature vector from the CNN to the decoder. We encode the output into the caption embedding dimension. Now given a caption, we create two vectors target and train. Target is same as the given caption, where as we remove the *end* token from the training caption. This is because the RNN does not have to learn to predict the next token after the *end* token.

Encoder implementation using CNN's: We have implemented two CNN's : ResNet-50 and VGG19 from scratch. The model used is taken from literature and is the standard architecture. VGG performs better in terms of performance and is also faster to train. So we will only report results on VGG-19.
Input to CNN is of the size 224*224*3 which is cropped randomly to downscale to 112*112*3 and the **embedding size is 256** which is empirically determined to be fed to LSTM. This

was also done take into account GPU memory issues. **To train the whole encoder from scratch, we could not experiment with higher embedding dimensions.**

Decoder implementation using LSTMs: All the caption words are converted to tokens and fed to the embedding layer which converts to a fixed length vector to be fed into RNN (LSTM). We have used PyTorch library to pack the sequences in a batch to send it together to the LSTM layer. The LSTM layer takes care of teacher-forcing and sends shifted inputs to the cells accordingly. Teacher forcing helps quickly and efficiently train recurrent neural networks, LSTM in our case that uses the ground truth from a prior time step as input. In addition, teacher training helps to address slow convergence and instability when training[1]. We have passed the LSTM output through a linear layer for the final prediction. The linear converts the output of the LSTM into a set of probabilities. It generates a vector of size [caption length * vocab size]. So the output is basically the probability distribution of how likely a word is in the current caption.

2.3 Evaluation of Generated Captions

BLEU Scores: We have used SacreBleu scores to evaluate our models. To generate captions, there are two of the following methods:

2.3.1 Greedy Search

In Greedy Search, the algorithm selects the word from the vocabulary generated from the LSTM with the highest probability. We use the Equation 1 and 2 for finding the caption. Instead of maximizing the product of probability, we maximize the sum of log probability to overcome the problem of underflow when the captions generated are of long length. This can sometimes leads to poor results.

$$\underset{y}{\text{argmax}} \prod_{t=1}^{T} P(y^{<t>}|x, y^{<1>}, ..., y^{<t-1>}) \tag{1}$$

$$\underset{y}{\text{argmax}} \sum_{t=1}^{T} \log(P(y^{<t>}|x, y^{<1>}, ..., y^{<t-1>})) \tag{2}$$

2.3.2 Beam Search Algorithm

This section will discuss our implementation of the Beam Search Algorithm for evaluating the generated captions with the golden captions. In beam search, we have recursively taken the top few (according to beam width) words at every input steps and generated a most probable captions tree. This algorithm works better than greedy search and improves the bleu score by at least a few points. We observed that the bleu score increases by at least 2 points in our results by using beam search algorithm with a beam width of 5. We empirically found that optimal beam width to be 5 when we vary the beam-width from from $1 - 10$. At low beam width the performance is poor and at higher beam widths we observe diminishing returns.

[1] https://machinelearningmastery.com/teacher-forcing-for-recurrent-neural-networks/

Encoder Model	Hyper-parameter	Epochs	SacreBleu Score
VGG-19	Adaptive LR = 0.1(En)	5	6.23
VGG-19	LR = 0.1(En)	10	5.5
ResNet-50	LR = 0.1	10	4.2

Table 1: SacreBleu scores obtained by hyperparameter tuning on model trained from scratch using same decoder (lr fixed at 0.0004). Adaptive Learning Rate for Encoder: Starting LR is shown in the table, step-size=2 and step=0.5 in our experiments

2.4 Hyper-parameter Optimizations

The core of all learning algorithms lies in the hyperparameter tuning of the model for better convergence and performance. This section discusses some of the many optimizations we have performed to tune the hyperparameters of the model. Given that these hyperparameters don't work independently and have a correlated impact on the performance of the model, the discussed optimizations are not exhaustive although cover most of the very important hyperparameter optimizations needed for a good model. Table 1 shows the summary of the BLEU scores we obtained by varying different hyperparameters of the model.

- **Optimizer:** We first experimented with Adam optimizer since Image captioning literature points [2] that the Adam optimizer works well. But we found that on training from scratch, our encoder was not learning very well using this optimizer. So we separated the optimizer for encoder and decoder. We used SGD optimizer with a learning rate of 0.1, weight decay of 0.0005 and momentum of 0.9. These values were empirically found after a lot of experimentation by noticing if the gradients are changing at the first layer of our encoder. We used Adam optimizer for Decoder with a learning rate of 0.0004 (again empirical found that it works best).

- **Learning Rate:** We have experimented with Adaptive Learning Rate for the encoder. The decoder works well at $lr = 0.0004$ so we have not touched that. We use a step LR scheduler for our encoder which gives the best performance with step size 2, initial lr 0.1 and step factor 0.5.

- **Dropout:** Our encoder model has a dropout layer between the last two linear layers. We added it to avoid overfitting since we are training a very deep model on a small dataset. In the Decoder model, we first added dropout after the embedding layer. It seems to work better than the dropout after the LSTM output (contrary to what we will see with pretrained models). We froze out model to this configuration of dropout (one dropout layer in encoder and one layer after embedding in the decoder).

- **Embedding and Hidden State Dimensions:** We could not experiment much in this part for the dimensions because of memory issues on Google Colab. We were somehow just able to make 256 dimension work.

3 Competitive Part

In the competitive part the major difference is that we have used a pretrained model for the Encoder available in torchvision library.

3.1 Data Preprocessing

The caption preprocessing remains the same as the non-competitive part. Since, we are running on pretrained encoder models, we were able to use the whole image for the run without any memory issues. So in this part, we have not used done the random cropping of the images to scale them down. All the images in the dataset have been re-scaled to the size of (224,224) since we have used pretrained models (mostly trained on ImageNet dataset). All the other transforms remain same as the non-competitive part.

3.2 Competative part: Proposed Architecture

3.2.1 Encoder

Empirically, it was found that ResNet-50 works best on the current dataset provided (other networks tried for encoder are: GoogleNet, VGG-19). Hence, ResNet-50 has been used for all the experiments that will be reported in this section. We have taken a pretrained model, removed its last layer (Softmax) and then added a Linear with ReLU activation and Batch Normalization Layer. The output of this layer goes to the decoder. We have only trained the last layers that we added and the other layers are kept fixed.

3.2.2 Decoder

Figure 1 depicts the same model as our network trained from scratch for the LSTM. We feed the image feature vector to the LSTM as input and predict $\langle start \rangle$ using it.
Experiments: Done using Bleu-1 scores.

- **Non-linearity in the decoder.** Since LSTM module of PyTorch already does a sigmoid function already, so any non-linearity after LSTM is detrimental to the performance.
- **Dropout:** We tried several variations of dropouts in the network. Dropout added after LSTM layer seems to perform better than the dropout after the embeddings (Bleu-1 score=0.47). We think this is because it reduces the overfitting of the linear layer after the LSTM to only a few words in the embeddings. Dropout after both layers performs much worse. **So, we froze dropout after LSTM layer in our model (Bleu-1 score = 0.50).**
- **Vocab Length:** We experimented a little with the vocabulary generated for the captions in the training dataset. We saw improvement in the performance on the public dataset by a few points in the Bleu-1 score (0.01 : Not very significant too). But due to the leader board requirements, we did not take in these changes.

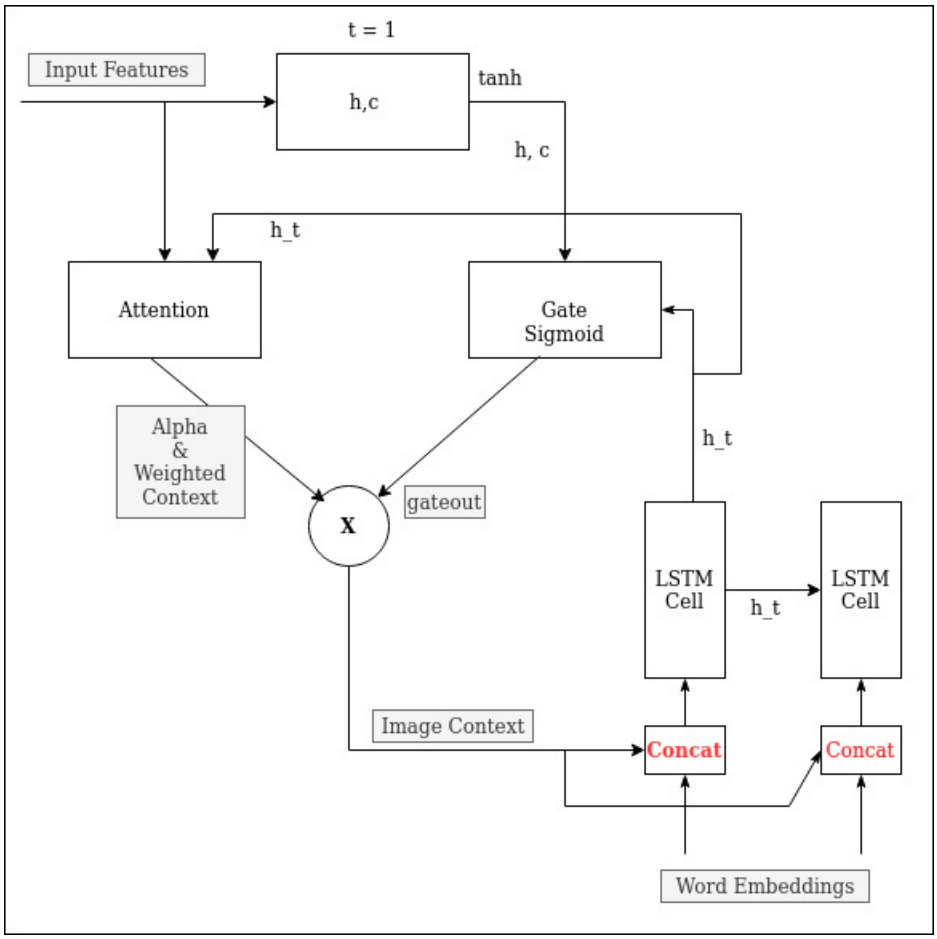

Figure 2: Depiction of our implementation of Attention Module

- **Using Pretrained Italian Embedding:** We used pretrained Italian embedding generated by [3]. We initialized the weights of our embedding layer with the pretrained model weights. However, we found that many words were missing from this repository so we had to initialize random weights for those words. Hence, we had to fine-tune the pretrained weights and train the initialized weights. Somehow this mixture did not perform very well. We got similar scores and hence discarded the pretrained embeddings.

- **Fine-Tune Encoder:** We also tried to fine-tune the last few layers of our ResNet model using the dataset provided. But it also performed worse because it needed more training epochs to run well. Due to resource constraints we did not explore much in this direction.

3.2.3 Using Attention:

Attention implementation is inspired by the work in [4]. We have used the concept called soft attention described in this paper. Instead of doing hard attention, where we would have

to sample the exact attention location in the image feature at every time step, we do soft attention. We determine a weighted context vector at every time sample that determine how important is a part of the image in that time step. Using this weighted context we can tell each LSTM cell how important that feature was in determining the current word in the caption. This context vector keeps changing with every hidden state and every time step. A more detailed representation of our implementation is shown in Figure 2. We have also implemented a small penalty as mentioned in [4] called *Doubly Stochastic Attention* where we add a penalty to the weights of the context to the loss function. This acts as a regularizer for the overall model. Since the weights are found after softmax, their sum is liable to be 1, but this penalty allows the sum to be approximately close to 1. We have not experimented on this and adapted what the paper has shown [5]. For using attention, we do not pass the encoder output through a linear layer. We take the raw features from the ResNet-50 and feed it to the Attention module.

We did not have time to finetune the Attention code alot since we did a lot of experiments with the LSTM layer and we are sure that it can lead to significant improvements.

Decoder Model	**Epochs**	**SacreBleu Score**
LSTM with LR = 0.0004	10	9.48
LSTM with adaptive LR	10	11.67
Attention with LR = 0.0001	10	14.64
Attention with Adaptive LR	10	14.22
Attention with Dropout=0.5, LR=0.0004	10	15.02
Attention with Dropout=0.4, LR=0.0004	7	14.07

Table 2: Comparison of Decoder Model with the same Encoder. SacreBleu Scores reported are on Public Dataset. Hidden size = 512, Embed Dim = 512, in all cases; Vocab used is full with no cleaning of captions. Adaptive LR: Starting LR=0.0009, step-size=2 and step=0.8

3.2.4 Hyper-parameter Optimizations

This section discusses some of the many optimizations we have performed to tune the hyperparameters of the model.

- **Optimizer:** We have used the Adam Optimizer for both the encoder and the decoder of our model since we are training the encoder and taking only raw features from it.
- **Learning Rate:** We have experimented with Adaptive Learning Rate for the decoder as well thinking it might lead to benefits. The LSTM decoder works well at $lr = 0.0004$. But by using Adaptive LR, we could increase the Bleu score by 2 points using LSTM model. We use a step LR scheduler for our decoder which gives the best performance with step size 2, initial lr 0.0009 and step factor 0.8. Table 2 shows a detailed analysis

of the learning rate tuning that we have done. We get maximum performance for Attention Model with fixed learning rate of 0.0004.

- **Dropout:** Neural networks are often over-parameterised and dropping a few neurons provides an effect of regularisation to overcome overfitting. In the light of optimising the neural networks, with the dropout rate of 0.5, the SacreBleu score increased to 15.03 and without dropout, the performance was 14.64. The reason we give is dropout acts as a regulariser and at the same time, it would reduce the number of parameters in the model aiding neural network compression.

- **Embedding and Hidden State Dimensions:** As we found that an embedding size and hidden dimension of 512 works best for the decoder, we used the same for the Attention Model also. However, since there are many more parameters in this model, we wanted to try lower dimensions also. But due to lack of resources and time, we could not run this.

- **alpha_c param for regularization**: This parameter is used in the regularization of the model. This is a hyperparameter that we have not looked into at all. We have used the values pointed out in [5].

4 Conclusions

The report presents a qualitative and empirical analysis of the Image Captioning Machine Learning Models and the results that are obtained by using different CNN models as Encoder of the image and different RNN models as the Decoder.

References

[1] https://github.com/mjpost/sacrebleu

[2] https://cs.stanford.edu/people/karpathy/main.pdf

[3] https://github.com/jvparidon/subs2vec

[4] Kelvin Xu and Jimmy Ba and Ryan Kiros and Kyunghyun Cho and Aaron Courville and Ruslan Salakhutdinov and Richard Zemel and Yoshua Bengio, "Show, Attend and Tell: Neural Image Caption Generation with Visual Attention"

[5] https://github.com/AaronCCWong/Show-Attend-and-Tell