Image Caption with Attention

Work by: VAHID FARJOOD CHAFI

Contents

1. Introduction

Automatic caption generator for a given image is a fundamental problem in the field of artificial intelligence which connects computer vision and natural language processing together. In this project, I provide a model which is based on Recurrent Neural Network architecture that uses CNN and LSTM where it can be used to produce a natural explanation of an image. This model will be trained in order to maximize the likelihood of the target sentence given its training image but it will also learn where to look when generation the related words.

1.1 Problem Analysis

A “classic” image captioning model would normally encode the given image, using a computer vision models like CNN as an ENCODER which would produce a hidden state h. Then, it will try to decode this hidden state by using a natural language processing models like LSTM and then generate recursively each word of the caption. The problem with this model is that, using the whole representation of the image h to condition the generation of each word cannot efficiently produce different words for different parts of the image.
I try to demonstrate a single joint model in which it takes an input image I, and will be trained to maximize the likelihood $p(W \mid I)$ of generating a target sentence of words $W = \{w_1, w_2, \ldots\}$ where each word in W_t comes from a given vocabulary, which can describe the image properly. But as you produce a caption, word by word, the model look moving over the image respectively. This is possible just because of its Attention system, which allows it to focus on some specific parts of the image which are most relevant to the word it is going to generate next.

1.2 Dataset

We will use Flickr8k dataset which is compose of 8,000 images that are each paired with five different captions. Each image is repeated 5 times with corresponding caption in the dataset during training and validation procedures. The dataset will be divided into a training set, for the purpose of learning the model parameters, a validation set, for comparing the models, and a test set, for the final quality of the selected model.

1.3 Model

The main idea of this project comes from the way that machine translation mechanism works, where the procedure is to transform a source sentence S, into its translated destination sentence D, by maximizing $p(D \mid S)$. In this project, I follow the same mechanism, by replacing the RNN(encoder) with a convolution neural network (CNN) in order to produce a good representation of the input image on the last hidden layer and embedding this fixed-length vector with the input caption, such that this representation can be feed as an input to the RNN decoder in order to generate next word but in a way that it will only focuses on those parts of the picture which is most related to the word that it is going to be generated. We can call it "Image Caption with Attention".

1.4 Performance

The most straightforward way to check the performance of this type of models would be to choose the word with the highest score. Although it is not a good way to measure this model (there are some better methods like BLEU) but I used this simple method for my project. We will discuss more about measuring the performance on training the model section (*2.3.7 Training - page 8*).

2. Implementation

I would explain the implementation of this project according to the order of each classes that are written in the code.

2.1 TextManager

The first class that I am going to explain you is the so called *TextManager()*. This class will give you the ability of doing some operations related to the text manipulation. For instance you can tokenize a text, you can build a vocabulary, or counting the words frequency and some other functionalities which I am going to explain them next.

2.1.1 Building a Vocabulary

First thing to do is to build a vocabulary from all the words which we have in our captions file. We need this vocabulary in order to train our model and also to generate captions. We also need to add some special words in our vocabulary for some important purpose. Those special words are "PAD", "SOS", "EOS", "UNK" which means Padding, Start Of Sentence, End Of Sentence, and Unknown words, respectively. Firstly, we need to pad sequences with different length in order to feed into the network and also for having more efficiency. Secondly, we have to indicate in some way the start and the end of a sentence so the network will realizes where the sentence started or where it ends. Finally, we add UNK for those words that are not present in our vocabulary (in the case where we choose a higher threshold frequency from 1). The method that we used to build it is called *build_vocabulary( )*.

2.1.2 Tokenization and Numericalization

Before starting to build a vocabulary we need a way to tokenize our sentences into words and then put each of these words into vocabulary separately. Tokenization is the first step in text processing task. Tokenization is not only to breaking the text into components, pieces like words, punctuation etc known as tokens, But also we can do some more advanced tokenizing which internally could identifies whether a "." is punctuation and separate it into token or it could be a part of an abbreviation like "U.S." and do not separate it. So instead of writing my own regular expression for tokenization I used "spaCy" library which is designed specifically for advance tokenization use. We accomplish it by a method called *tokenize( )*.
We also must convert our words into a numerical representation in order to feed them into our network. In other word, we need a method to get the index number of a word from our vocabulary. We called this method *numericalize( )*.
Another useful method that I used here is called *get_key( )* which is used when you have the indices of a word and you need to convert it back to its corresponding key or the word(used during generating the captions).

2.1.3 Save and load

During working with this type of project it might be a good idea to save the vocabulary that you have just built and use it in the future. For instance you might want to evaluate only your model with some different small dataset which in that case you must already have your vocabulary. So in that case you need to load it from your disk. There is two method called *Save( )* and *Load( )* in TextManager specially for this purpose.

2.1.4 Load Embedding

There is a method called *load_embedding_file()* which is used to load a pre-trained words embedding matrix like gloVe. Inside the method you will find out that I initialized the matrix with a uniform distribution and the reason is that for those special reserved token(PAD, SOS, EOS, UNK), we do need some random numbers for their embed vector since we don't have them in our embedding file and also the embed matrix will be freeze during training(so no learning for that). But for those words that are exists in our vocabulary we take their values from embedding file and overwrite them in our new embedding matrix, so it is not random anymore.

2.2 Dataset

The data that we are going to use in the project is called *Flikr8k* and it is compose of a folder with all the images and another file called caption.txt which contains all images names with their corresponding caption. If you look inside of this file then you will realize that each image name is repeated 5 times with different caption. In other way, each image has 5 line with different descriptions (figure-1). We will use a customize dataset class called *CapDataset()* which is a child of *torch.utils.data.dataset* class in order to build our dataset object.

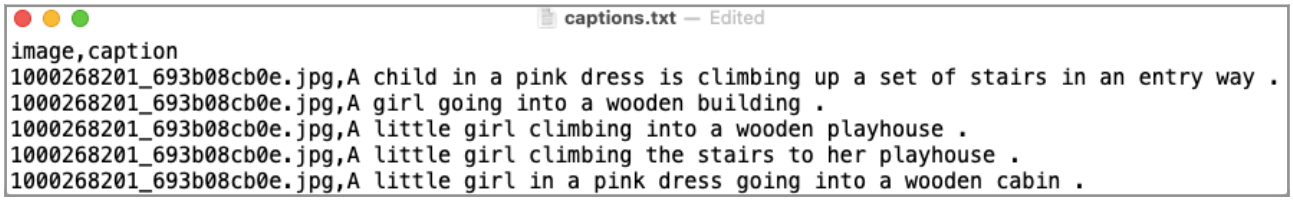


Figure - 1

2.2.1 Initialization

In order to feed the input data into the network we will use Pytorch DataLoader facility but before that we need to clean and prepare our data and then it would be possible to use data loader. When you create an object of CapDataset for the first time, it will be initialized by some attributes and also it will check the dataset folder that you provided just to see if it is a correct path or not. Then it tries to set the preprocessing method which we need in order to use images as an input for our encoder ResNet50. If you chose the *custom* preprocess then there will be some small changes in each images like resizing or rotation otherwise it will use a default preprocessing to prepare the image for the encoder.

There is also another important thing that will be check according to the need of creating the dataset in terms of training, evaluating, or testing purposes. The most obvious difference is whether to build a vocabulary or not. For the purpose of training, for sure we need to build a vocabulary but in the case of evaluating or testing our model we would use our predefined vocabulary and we just need to load it. There is also a small difference in case of test dataset which is to add some dummy caption just to make the dataset to be in its standard way. (Normally we don't have any caption for test images).

2.2.2 Randomization

In most machine learning task we would normally first randomize our dataset in order to prevent our model to face some technical issues related to the dataset like not to be biased. Of course we also expect our program to do randomization by default but I tried to make it choosable for some technical reasons which is related to the lack of having a good hardware resources. So in that case you might not wish your data to be randomize and you want the same data as previous used data to

evaluate your new edited model. You will see that during splitting the data, we can also choose a small portion of the dataset and use them for training, validating and testing.

2.2.3 Splitting

You can easily split your data in to two or three portions with some arbitrary size as you would like. I said with some arbitrary size because as we discuss in the previous paragraph due to the lack of having enough hardwares like GPUs, I provide this facility in order to apart the data into some small sizes. So there is no limit on the sum of all fractions to be one in terms of splitting data size.

2.2.4 Collate Function

In order to fetch a minibatch of data and collating them into batched samples with padding sequential data to max length of a batch, we need to use a custom collate function for our dataloader and it is called *CapCollate( )*.

2.3 Model

Now we are ready to build our model and start to learn its parameters from data. You will also see the procedures of selecting the best model with the use of validation dataset. Then I will explain how I implement the performance measure for this specific neural model.

2.3.1 Architecture

The model uses an encoder-decoder like architecture. The encoder is a pre-trained ResNet50 network used as a feature extractor. The decoder is a recurrent neural network(LSTM) that takes the extracted features as an input and produces a caption. The decoder incorporates with an attention system that allows the decoder to focus on some parts of the encoded input when producing the caption. You can see the architecture of the model in below(figure-2).

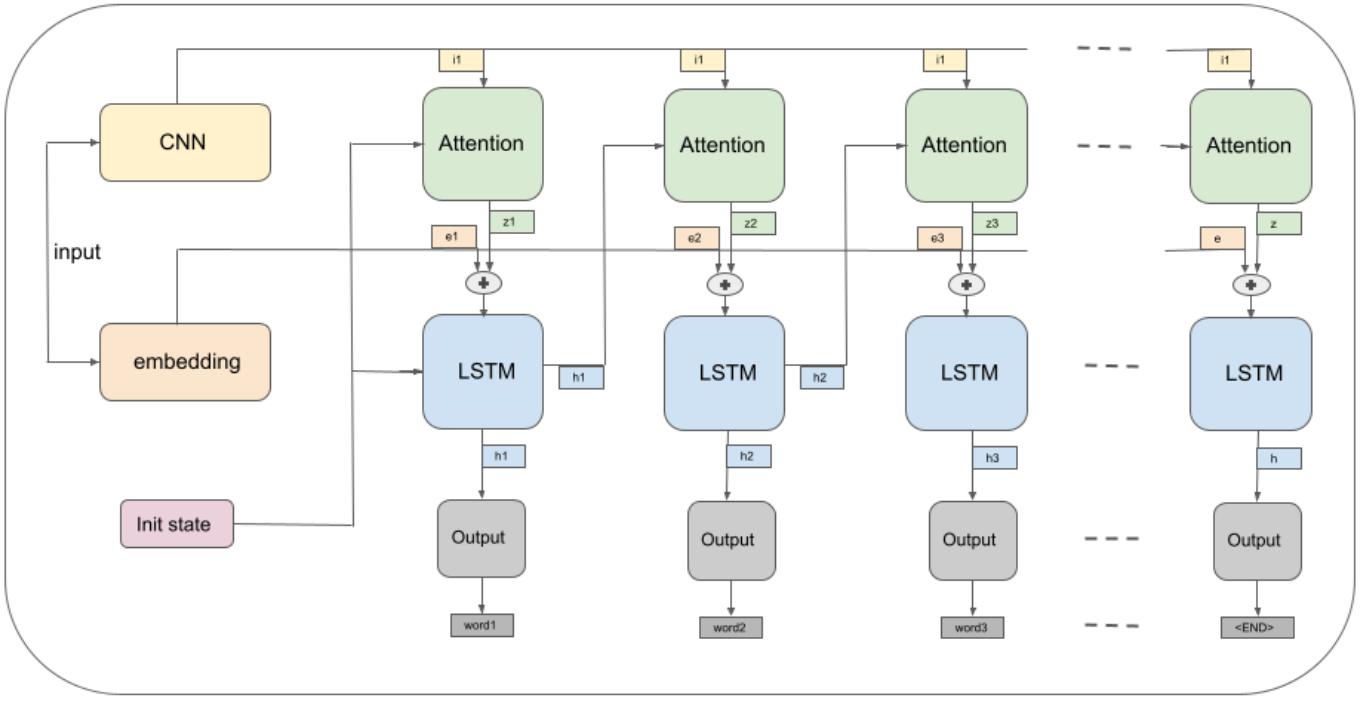


Figure - 2

2.3.2 EncoderDecoder

At the first time when you create an object of *EncoderDecoder()* class, it will be initialized by some default attributes such as *self.encoder()* and *self.decoder()* and also the type of criterion function which I chose *nn.CrossEntropyLoss()* function since we will generate a sequence of words.

2.3.3 Encoder

The encoder will be initialized with ResNet50 and we call it *Encoder_CNN()* class. The pre-trained network reset50 can classify images into 1000 object categories but for the purpose of our project we don't need the last layer since our task is not classifying images. So we will discard the last layer of the network which is Linear(in=2048, out=1000) plus *AdaptiveAvgPool2d* layer. By removing these two last layer what we get is 7x7x2048 feature maps. This 7x7 pixel locations corresponds to certain portion in image and we can treat it as 49 locations where each of them having 2048 dimension representation which in turn will produce the hidden state h for our decoder. The model will then learn how to attention over these locations.

We would also set the requires grad to be False since we don't want to train this huge network again! Although we could add another linear layer on the last layer and set the requires-grad of this layer to be True in order to fine-tune our model but I prefer not to do it since it will affect on the performance.

2.3.4 Attention

An Attention model is a neural network in which its input is compose of a pair of vectors. First one is the state of the language generation and the second one is the feature vector of input image that you want to consider. Basically this network will compare each portion of input image with the state of the language generation and it would understand how a portion of image is important given what we said up to this point.

The importance of each portion of image is provided by a coefficient value called 'alpha' and the attention model should learn how to compute this "alpha" at each time step.

Lets say our language generation(we will discuss about it in the next section) has an internal representation "h_t" from which it decides what is next word to say. however, what we want to do here is to devise a neural model that the way it generate next word is not only because of internal state of the language generator but it should be also because of different portion of the image that we are observing. Lets see how it works on formula:

$$Wt = Gen(h_t , z_t) \qquad (1)$$

Where Wt is our language generator which depends on the hidden state 'h_t' and context-vector 'z_t'. For the context vector we have:

$$z_t = \sum a_i . v_i \qquad (2)$$

$$\sum a_i = 1 \qquad where \ \ i = 1 \ldots m \qquad (3)$$

$$a_t = [a_{t,1} , a_{t,2} , \ldots a_{t,m}] \qquad (4)$$

Where 'm' is the number of image locations. And 'a_t' is the output of the neural network with softmax activation in the output layer:

$$a_t = \text{softmax}(f_{ATT}(h_t,v_1), \ldots f_{ATT}(h_t,v_m)) \qquad (5)$$

So we are going to have a special neural model that is only use to understand how each input portion is important with respect what we said so far in order to decide what to say next.

In the code if you look at *Attention()* class, then you immediately will realize that we follow exactly the same thing that we just discussed. First we will get our feature vector and previously computed hidden state and then apply a linear transformation to each of them separately in order to prepare them as an input for our attention network.

```
#linear transformation from encoder(CNN) and decoder(LSTM) dimension to attention dimension
LinearEncoder_outputs = self.LinearEncoder(features)
LinearDecoder_outputs = self.LinearDecoder(hidden_state)
```

Then we add them together and apply a non-linear transformation (relu or tanh) on top of that.

```
#combine encoder and decoder outputs together (we can also use tanh())
combined_output = self.relu(LinearEncoder_outputs + LinearDecoder_outputs.unsqueeze(1))
```

Then right after that we apply another linear transformation and a softmax on top of it in order to get our alphas.

```
#compute the outpout of attention network
attention_outputs = self.LinearAttention(combined_output)
attention_scores = attention_outputs.squeeze(2)
alphas = F.softmax(attention_scores, dim=1) #the sum of all alphas should be equal to one.
```

Finally, having our computed alphas, now we can multiply it to our feature vector in order to get the weighted sum combination of input vector which is called context vector.

```
#compoute the context vector
context_vector = features * alphas.unsqueeze(2)
context_vector = context_vector.sum(dim=1)
```

You can also see the flow diagram of the procedure in below figure-3:

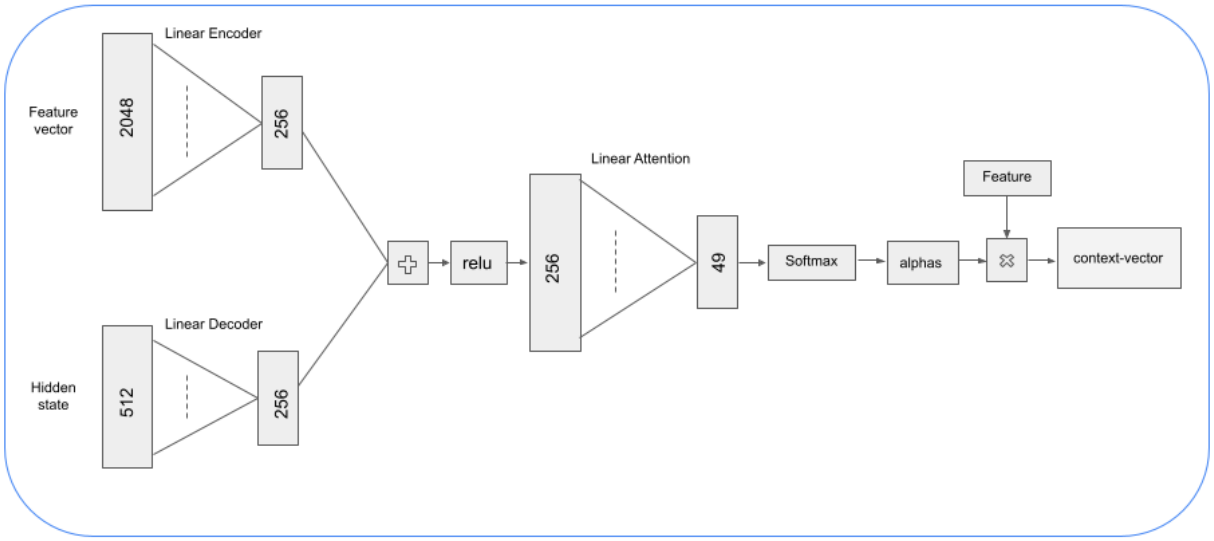


Figure - 3

2.3.5 Decoder

Basically, for generating the caption, we have a decoder class in order to predict the next word and we call it *Decoder_LSTM()*. First thing we do here is to check whether we want to use pre-trained word embedding or not. Next, we convert our caption vector from sequence of tokens into vector of embeddings by *self.embedding(captions)*. Then we initialize the hidden and cell state of the LSTM using the encoded image with *self.init_hidden_state(features)* method, which uses two separate linear layers.

At the beginning, we sorted the N images and captions by decreasing caption lengths. This is needed when we want to process only valid time steps, and not processing the <PAD>. Now in order to produce the hidden state over each time step, we will iterate over the sequence lengths.

This iteration is performed manually in a ***for*** loop with *Pytorch LSTMCell*. I didn't use *Pytorch LSTM* and the reason is that we need to execute the Attention mechanism between each decode step. An *LSTMCell* operations as a single time step, conversely an LSTM iterates over multiple time steps continuously and will give you all outputs at once.
Inside the loop, at each time step we will compute the context-vector and the alphas.

```
#get context vector with the encoder features and previous hidden state
alpha, context = self.attention(features, h) # context=[32, 2048]
```

Then we concatenate this context-vector with the embedding of the previous word, and run the LSTMCell to generate the new hidden state(this technique is called Teacher Forcing).

```
#combine embedding vector of the word with context vector and feed it to lstmcell
lstm_input = torch.cat([embeds[:, word], context], dim=1) # [32, 2348]
h, c = self.lstm_cell(lstm_input, (h, c))
```

Then a linear layer transforms this new hidden state into scores for each word in the vocabulary, so the output dimension is equal to the size of vocabulary. Also we used Dropout() in order to help prevent overfitting.

```
#get the logits of the decoder (also we used dropout for regularization purpose)
logits = self.decoder_out(self.dropout(h))
```

Finally, it will collect all generated word for all time step for the corresponding minibatch and return it as an output tensor to the training procedure plus the alphas(we need alphas for loss!)

2.3.6 Caption Generating

This specific method is used only for the purpose of generating a new caption for a given image. It follows the same procedure as we had in the previous section (*2.3.5 Decoder*) but with some small differences. This method is call with only one single image whenever you want to evaluate the model, either during training, evaluation or testing.

2.3.7 Training

Training procedure is a function of the model which is called ***Train_model()***. You will pass some arguments (like training_set and validation_set, …) and at the very outset, it will initialize the optimizer of the model as Adam() which is an adaptive optimizer. Next, it will check whether you want to resume the state of your previous model or not. So you can easily reload your previous model that you might were stoped in the middle for any reasons but then by using this facility it will help you to continue your training procedure from last state of the model(from last epoch).
Inside of *epoch* loop, first you will initialize some attribute in order to keep track of some performance measuring values and then you will have minibatch loop. Going inside the minibatch loop, we first have our minibatch samples including (*image, captions, caption_len)* and we move them to the proper device. We need to set the gradients to zero before starting to do backpropagation because PyTorch accumulates the gradients on subsequent backward passes.
Next we will start to FeedForward our minibatch data into our model in order to get the outputs:

```
# Feed forward the data to the main model
outputs, alphas, captions, seq_length = model(image, captions, caption_len)
targets = captions[:, 1:] #skip the start token (<SOS>)
```

Then we use this output in order to compute the loss and accuracy but before that we first use another PyTorch facility which is called *pack_padded_sequence()* and this will help us to discard the padded sequence. So we would only care to the real sequence length not padded sequences:

```
#skip the padded sequences
outputs = pack_padded_sequence(outputs, seq_length.cpu().numpy(), batch_first=True)
targets= pack_padded_sequence(targets, seq_length.cpu().numpy(), batch_first=True)
```

```
#compute the accuracy of the model
acc = model.__performance(outputs, targets)
#compute the loss
loss = model.__loss(outputs, targets)
loss += 1. * ((1. - alphas.sum(dim=1)) ** 2).mean()
```

I added another line for the loss and the reason is that as we already know the sum of all the alphas is 1 at a given time step, but it would be more effective if the sum of alphas at a single location 'p' be also 1 over all time steps. It means, we are going to minimize the differences between 1 and the sum of a pixel's location weights over all time steps:

$$\sum a_{p,t} \approx 1 \qquad \textit{where the } \textbf{\textit{sum}} \textit{ is over } \textbf{\textit{T}} \textit{ and } \textbf{\textit{t}} \textit{ is the time steps} \qquad (6)$$

For the purpose of the accuracy what I did is to take the index of the highest value of the last dimension of the *Output* (size of vocabulary) and compare it with the *Target* and count how many indices were predicted correctly according to the words of the corresponding target. But this is not optimal and the reason is that if you think what happens inside the model when predicting each word, you will realize that the rest of the sequence depends on that first word you choose. So if that choice is not the best one, then everything that follows would be sub-optimal. And this is not only for the first word, also each word in the sequence has this consequence. It might be a better choice if you choose the second best or third one which eventually might give you a better description of the image. So it would be better if you could select not only the first highest one but also select up to k number and then evaluate them(it is called Beam Search).

Although there are some better ways to check the accuracy like BLEU which will simply take the fraction of n-grams in the predicted sentence that appears in the ground-truth, but even though it is still a controversial topic and very challenging.

Finally, at the end of each epoch we will try to evaluate our model with the validation set and right after computing the accuracy of the model on the validation set, we check the best performed model that we got on validation set, and save that best performed model to the disk.

2.3.8 Evaluating

We have the same procedure of the evaluation method as the training procedure with having only one difference that here we are evaluating the model with validation set and we are not going to update the model so at the beginning of the method we make sure to set the model on evaluating state with *model.eval()* in order to prevent our model from being updated.

2.3.9 Visualization

You can use *Test_and_Plot()* method of the model for visualization purposes. This method will use two other functions to visualize an image which are *plot_attention()* that is used to show the attention mechanism and *show_image()* which is used to show an image when you generate a caption. We will also use a function called *save_acc_graph()* for saving the accuracy graph into the disk.

2.5 Input Interface

With the use of Argparse library you can easily tune the project without changing the code. You just need to choose your customized arguments and then pass those arguments to the program. In order to see what arguments you can choose, then try to get help by running this command:

```
> python image_captioning_with_attention.py -h
```

3. Inference

3.1 Conclusion

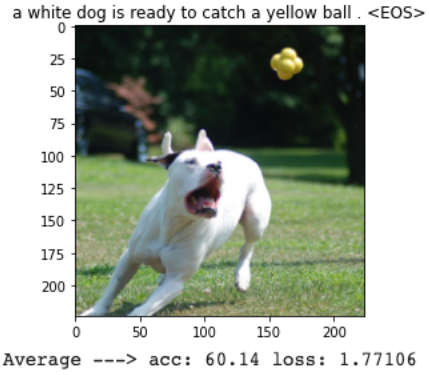
1. It is very important to know that the images used for testing must be conceptually related to those used for training the model. In other word, the distribution of test data should be same as distribution of training data, otherwise you will not get an appropriate result. As an example, if we train our model on the images of elephant, dogs, rat, etc. we must not test it on images of air planes, cars, bike etc.
2. I used a technique called Teacher Forcing during training and evaluating which is a way to speed-up training procedure by providing the real target as the input at each decode-step but as I understand it is common to do that during training but not for evaluating since during evaluating it must mimic real inference conditions as much as possible.
3. For computing the accuracy it is not fear enough to use only the first predicted word at each time steps and eventually compare that generated caption with the real caption since there might be many different sentences that could describe the image correctly in different way. We need to select top k predicted word at each time steps and then compare the accuracy of those k different generated captions which probably will gives us a better result.
4. Do we also need more target captions as a reference to compare each generated sentence with them? For instance having 20 different captions for each image would improve accuracy since now we have more chance of generating a caption in a different way? But having more captions for each image will force us to repeat each image with respect to each caption just because of the number of captions it has which probably will help the model to understand more about the images but wouldn't it make the model confuse in a way that it should update itself each time with a new caption…?!
5. In order to represent the words as an input to the decoder LSTM, I use word embedding vectors instead of using one-hot encoding or bags of words representation, which will give us the advantage of being independent of the size of the vocabulary.
6. The main problem with overfitting normally would be the distribution of the data but apart from that I also tried several techniques to deal with. I tried to initialized randomly some of the weights of the model but since PyTorch by default will initialize all the wights and biases of a Linear layer then there was no need to do that, so I just tried to initialized randomly the wights of embedding. Another technique that I used is Dropout at the last layer of the decoder in order to turning off some number of hidden units. We could also try with a smaller size of the network in order to prevent our network being overfit.
7. Setting a higher frequency threshold for building the vocabulary will help the performance of the network in term of better prediction since with smaller size of vocabulary we will have smaller number of output neurons in the model and the language generator has more chance of producing those words that are more frequent.
8. We could also add another linear layer on the top of the encoder in order to fine-tune it for our specific task which could probably give us a better result.
9. In this specific architecture that we used here, what if the encoder fails to identify the most important objects and their attributes, then I guess, there would be no valid description to be generated. So it would be paramount to choose a prefect encoder model to detect objects very well. I used resnet50 but we could also try with resent101 or VGGNet.

3.2 Results

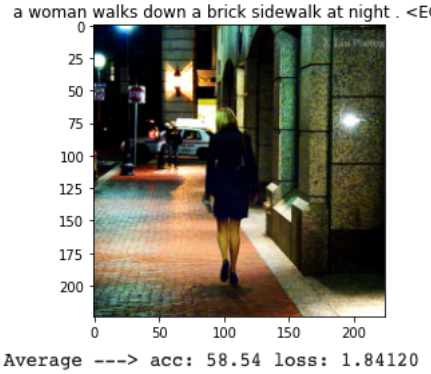
You can see some examples of the model with the below setting:

```
epochs=50                       attention_dim=256
batch_size=32                   encoder_dim=2048
learning_rate=0.0003            decoder_dim=512
Training size: 1618             embed_size=256
Validation size: 323            freq_threshold=3
```

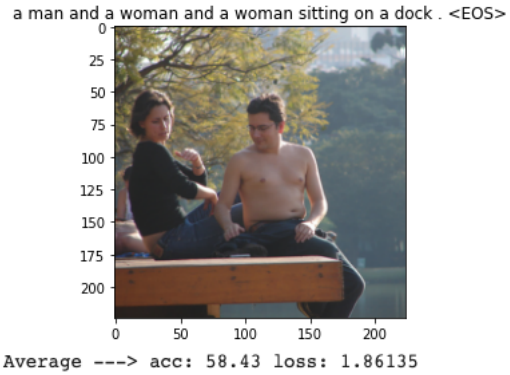
Results on Training_set:




```
Average ---> acc: 60.14 loss: 1.77106
```

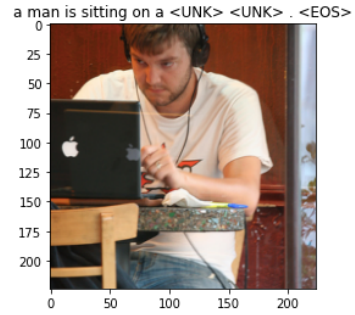



```
Average ---> acc: 58.54 loss: 1.84120
```

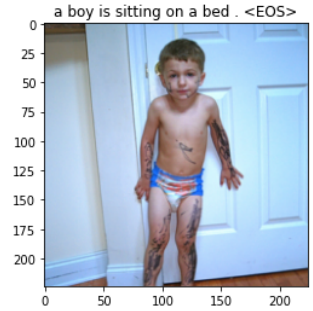



```
Average ---> acc: 58.43 loss: 1.86135
```

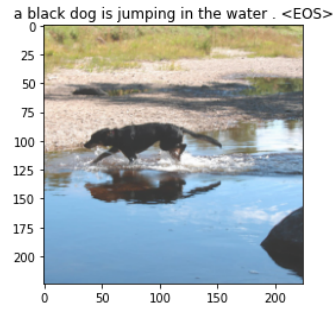
Results on Validation_set:



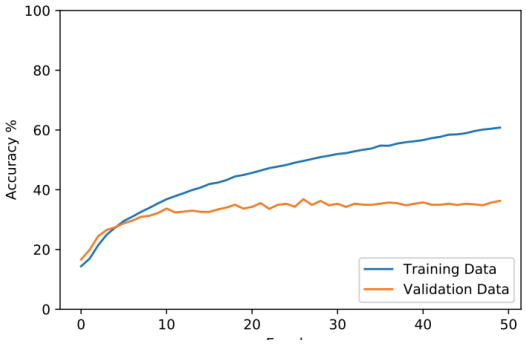

```
Average ---> acc: 34.89 loss: 4.27821
```




```
Average ---> acc: 36.29 loss: 4.21903
```




```
Average ---> acc: 34.98 loss: 4.28394
```



Epoch from 0 to 50

It shows that training accuracy goes up as long as the number of epoch goes up but validation accuracy stoped after 10 epochs which means that with small dataset the model will suffer from overfitting.