Visual Recognition

Report on Image Captioning with Flickr8K Dataset

Team ID - 41

Team Members:-

1. **HUSSNAIN ASHRAF (MT2021055)**
2. **KUNAL (MT2021067)**

Q1].

A CNN-LSTM Architecture for image captioning :-

So first let’s understand what's actually Image Captioning is :-

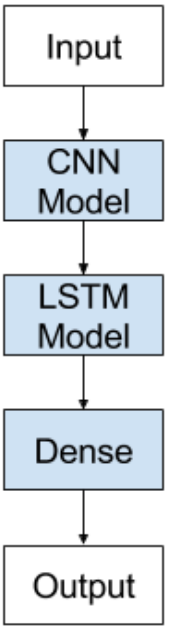
Image captioning is a task that involves computer vision and natural language processing concepts to recognize the context of an image and describe them in a natural language like English.

So, The goal of image captioning is to convert a given input image into a natural language description.

CNN-LSTM ARCHITECTURE:

The **CNN-LSTM architecture** involves using CNN layers for feature extraction on input data **combined with LSTMs to support sequence prediction**. This model is specifically designed for sequence prediction problems with spatial inputs, like images or videos. They are widely used in Activity Recognition, Image Description, Video Description and many more.

The general architecture CNN-LSTM are as follows:-

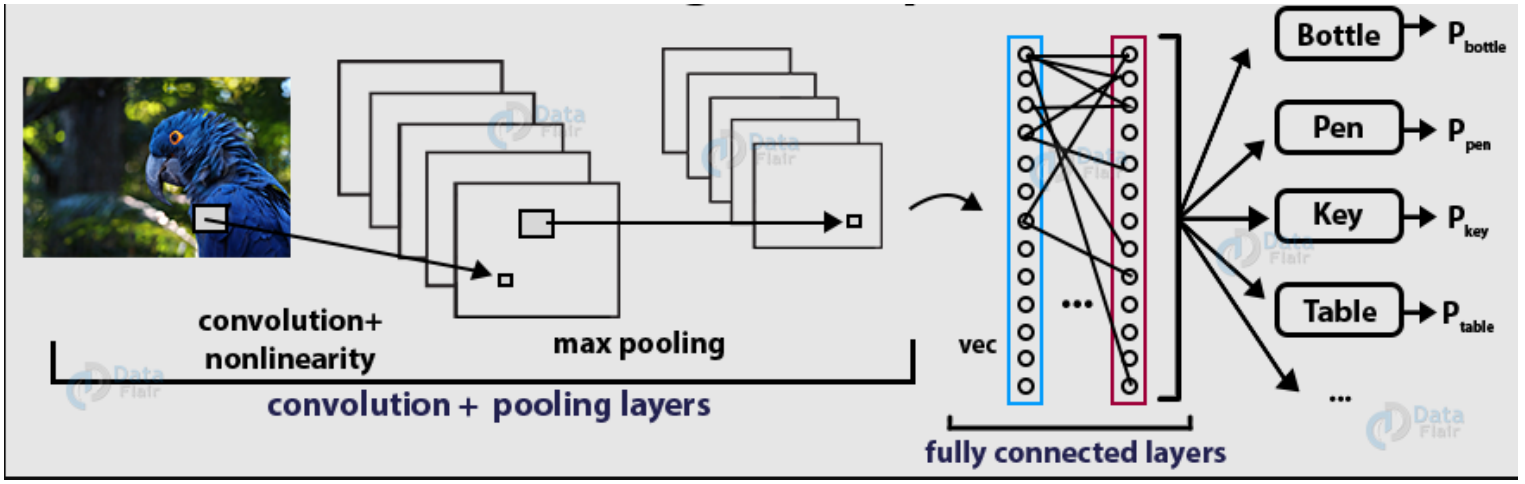


Let's understand what is CNN since i am using here CNN_LSTM model:-

Convolutional Neural networks(CNN) are specialised deep neural networks which can process data that has an input shape like a 2D matrix. Images are easily represented as a 2D matrix and CNN is very useful in working with images.

CNN is basically used for image classifications and identifying if an image is a bird, a plane or any images .

Working of Deep CNN

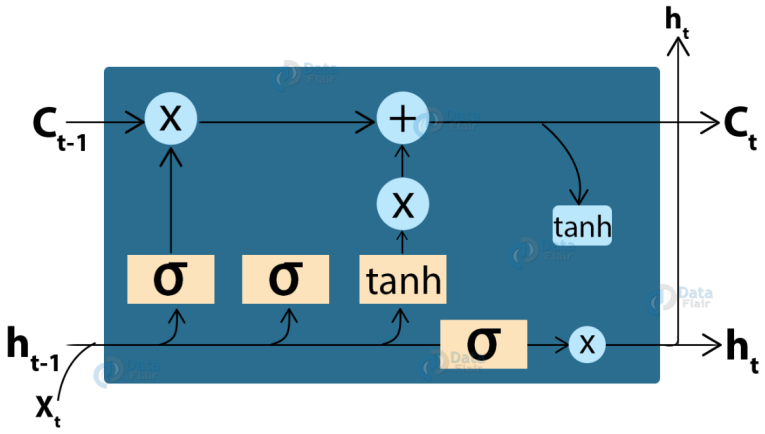


It scans images from left to right and top to bottom to pull out important features from the image and combines the features to classify images. It can handle the images that have been translated, rotated, scaled and changes in perspective.

Lets now understand LSTM :-

LSTM stands for Long short term memory, they are a type of RNN (recurrent neural network) which is well suited for sequence prediction problems. Based on the previous text, we can predict what the next word will be. **It has proven itself effective from the traditional RNN by overcoming the limitations of RNN which had short term memory.** LSTM can carry out relevant information throughout the processing of inputs and with a forget gate, it discards non-relevant information.

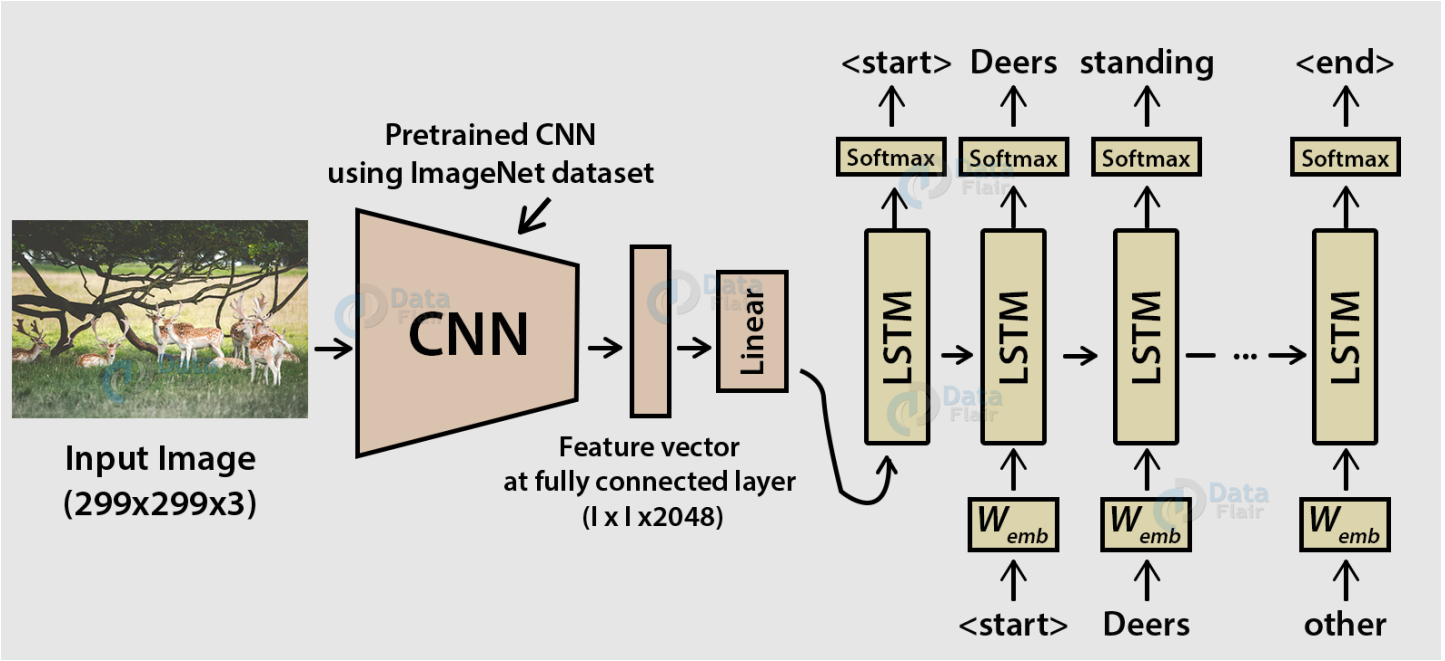
LSTM cell architecture :-



So, to make our image caption generator model, we will be merging these architectures. It is also called a CNN-RNN model.

- **CNN is used for extracting features from the image. We will use the pre-trained model Xception.**
- **LSTM will use the information from CNN to help generate a description of the image.**

Model of Image Caption Generator



The task of image captioning can be divided into two modules logically –

1. **Image based model** — Extracts the features of our image.
2. **Language based model** — which translates the features and objects extracted by our image based model to a natural sentence.

For our image based Model - we will rely on a Convolutional Neural Network model. And For language based models — we will rely on LSTM. The image above summarises the full approach.

Why the Flickr8k dataset ?

We usually take this dataset only because of the following features :-

1. It is small in size. So, the model can be trained easily on low-end laptops/desktops.
2. Data is properly labelled. For each image 5 captions are provided.
3. The dataset is available for free.

Explanation of dataset:-

We have basically two folders which are as follows:-

> **Flickr8k_Dataset**: **Contains a total of 8092 images in JPEG format with different shapes and sizes. Of which 6000 are used for training, 1000 for test and 1000 for development.**
>
> **Flickr8k_text : Contains text files describing train_set ,test_set. Flickr8k.token.txt contains 5 captions for each image i.e. total 40460 captions.**

We have mainly two types of data.

1. **Images**
2. **Captions (Text)**

Featurization of Images:-

There are already pre-trained models on standard Imagenet dataset provided in keras. Imagenet is a standard dataset used for classification. It contains more than 14 million images in the dataset, with little more than 21 thousand groups or classes.

We will be using InceptionV3 by google

We will **remove the softmax layer from inception** as we want to use it as a feature extractor. For a given input image, inception gives us a 2048 dimensional feature extracted vector.

```
# Importing necessary modules

from keras.applications.inception_v3 import InceptionV3,preprocess_input
from keras.layers import Dense,BatchNormalization,Dropout,Embedding,RepeatVector
from keras.preprocessing.image import load_img, img_to_array

from keras.models import Sequential
from keras.models import Model
```

```
# Since we are using this as feature extractor, the last softmax layer is not useful for us.
inception = InceptionV3(weights='imagenet')
```

```
Downloading data from https://storage.googleapis.com/tensorflow/keras-applications/inception_v3/inception_v3_weights_tf_dim_ordering_tf_kernels.h
96116736/96112376 [==============================] - 1s 0us/step
96124928/96112376 [==============================] - 1s 0us/step
```

```
inception.summary()
```

```
 average_pooling2d_8 (AveragePo  (None, 8, 8, 2048)  0           ['mixed9[0][0]']
 oling2D)

 conv2d_85 (Conv2D)             (None, 8, 8, 320)    655360      ['mixed9[0][0]']

 batch_normalization_87 (BatchN  (None, 8, 8, 384)   1152        ['conv2d_87[0][0]']
 ormalization)

 batch_normalization_88 (BatchN  (None, 8, 8, 384)   1152        ['conv2d_88[0][0]']
 ormalization)

 batch_normalization_91 (BatchN  (None, 8, 8, 384)   1152        ['conv2d_91[0][0]']
 ormalization)
```

So now here is the screenshot of how i am popping out the last layers and freezing the remaining layers :-

```
                                                                 'activation_93[0][0]']

 avg_pool (GlobalAveragePooling  (None, 2048)        0           ['mixed10[0][0]']
 2D)

 predictions (Dense)            (None, 1000)         2049000     ['avg_pool[0][0]']

==================================================================================================
Total params: 23,851,784
Trainable params: 23,817,352
Non-trainable params: 34,432
__________________________________________________________________________________________________
```

```
# pop the last softmax layer and freezing the remaining layers
inception.layers.pop()

for layer in inception.layers:
    layer.trainable = False
```

```
inception.layers[-2].output
```

```
<KerasTensor: shape=(None, 2048) dtype=float32 (created by layer 'avg_pool')>
```

Image == > {Inception + Embedding } == > we are extracting image from here and then building up the final model

```
# building the final model
final_model = Model( inception.input, inception.layers[-1].output)
```

```
[ ] final_model.summary()
```

```
Model: "model"
_________________________________________________________________________________________________
 Layer (type)                 Output Shape              Param #     Connected to
=================================================================================================
 input_1 (InputLayer)         [(None, 299, 299, 3       0           []
                              )]

 conv2d (Conv2D)              (None, 149, 149, 32       864         ['input_1[0][0]']
                              )

 batch_normalization (BatchNorm  (None, 149, 149, 32    96          ['conv2d[0][0]']
 alization)                   )

 activation (Activation)      (None, 149, 149, 32       0           ['batch_normalization[0][0]']
                              )

 conv2d_1 (Conv2D)            (None, 147, 147, 32       9216        ['activation[0][0]']
                              )
```

Now we are converting image to 300 dimensional in which first we are converting image into array and then we are adding one more dimension

```
# code for image imbedding i.e converting image to 300 dimentional

train_image_extracted = dict()
with open("/content/drive/MyDrive/Colab Notebooks/VR_Image_captioning_project/Flickr8k_text/Flickr_8k.trainImages.txt","r") as f:
    data = f.read()

try:
    for el in data.split("\n"):
        tokens = el.split(".")
        image_id = tokens[0]
        img = load_img(
            "/content/drive/MyDrive/Colab Notebooks/VR_Image_captioning_project/Flicker8k_Dataset/{}.jpg"
            .format(image_id),target_size=TARGET_SIZE)
        # Converting image to array
        img_array = img_to_array(img)
        nimage = preprocess_input(img_array)
        # Adding one more dimesion
        nimage = np.expand_dims(nimage, axis=0)
        fea_vec = final_model.predict(nimage)
        train_image_extracted[image_id] = np.reshape(fea_vec, fea_vec.shape[1]) # reshape from (1, 2048) to (2048, )


except Exception as e:
    print("Exception got :- \n",e)
```

For every training image, **we are resizing it to (299,299) and then passing it to Inception for feature extraction.** We will save the train_image_extracted Dictionary because it will **save a lot of time if you are fine-tuning the model.**

Caption Processing from Flickr8k Text files:
Each image in the dataset is provided with 5 captions which are depicted in the below screenshot :-

```
[136] image_id

'997722733_0cb5439472'

[137] image_desc

'A rock climber practices on a rock climbing wall .'

descriptions["1000268201_693b08cb0e"]

['A child in a pink dress is climbing up a set of stairs in an entry way .',
 'A girl going into a wooden building .',
 'A little girl climbing into a wooden playhouse .',
 'A little girl climbing the stairs to her playhouse .',
 'A little girl in a pink dress going into a wooden cabin .']
```

Captions are read from Flickr8k.token.txt file and stored in dictionary k:v where k = image id and value = [list of caption].

```
# opening text file
with open("/content/drive/MyDrive/Colab Notebooks/VR_Image_captioning_project/Flickr8k_text/Flickr8k.token.txt") as f:
    data = f.read()

# dictionary contining key as image_id and value as list of captions.
descriptions = dict()

try:
    for el in data.split("\n"):
        tokens = el.split()
        image_id , image_desc = tokens[0],tokens[1:]

        # dropping .jpg from image id
        image_id = image_id.split(".")[0]

        image_desc = " ".join(image_desc)

        # check if image_id is already present or not
        if image_id in descriptions:
            descriptions[image_id].append(image_desc)
        else:
            descriptions[image_id] = list()
            descriptions[image_id].append(image_desc)
except Exception as e:
    print("Exception got :- \n",e)

descriptions["1000268201_693b08cb0e"]
```

Detailed strategy used to asses the ‘language bias’ and justification for the same.

Since there are 5 captions for each image and we have preprocessed and encoded them in below format

“startseq(start sequence) “ + caption + “ endseq(end sequence)”

The reason behind startseq and endseq is,

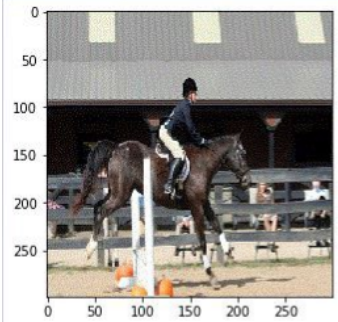
startseq : Will act as our first word when a feature extracted image vector is fed to the decoder. It will kick-start the caption generation process.

enseq : This will tell the decoder when to stop. We will stop predicting words as soon as endseq appears or we have predicted all words from the train dictionary, whichever comes first.

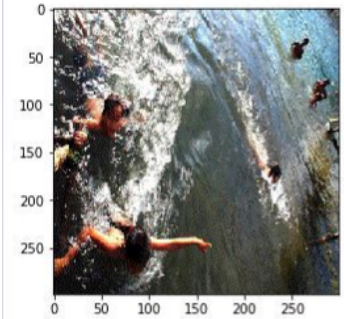
```
for k in descriptions.keys():
    value = descriptions[k]
    caption_list = []
    for ec in value:

        # replaces specific and general phrases
        sent = decontracted(ec)
        sent = sent.replace('\\r', ' ')
        sent = sent.replace('\\"', ' ')
        sent = sent.replace('\\n', ' ')
        sent = re.sub('[^A-Za-z0-9]+', ' ', sent)

        # startseq is for kick starting the partial sequence generation and endseq is to stop while predicting.

        image_cap = 'startseq ' + sent.lower() + ' endseq'
        caption_list.append(image_cap)
    descriptions[k] = caption_list
```

```
[141] descriptions["1000268201_693b08cb0e"]
```

```
['startseq a child in a pink dress is climbing up a set of stairs in an entry way  endseq',
 'startseq a girl going into a wooden building  endseq',
 'startseq a little girl climbing into a wooden playhouse  endseq',
 'startseq a little girl climbing the stairs to her playhouse  endseq',
 'startseq a little girl in a pink dress going into a wooden cabin  endseq']
```



['startseq a jockey on a black horse jumps over a hurdle endseq',
'startseq an equestrian and a horse are jumping over an obstacle endseq',
'startseq a person wearing a navy jacket and black hat jumping over a small partition on a horse endseq',
'startseq a show jumper is making a brown horse jump over a white fence endseq',
'startseq a woman on a horse jumps an obstacle endseq']



['startseq a bunch of people swimming in water endseq',
'startseq a group of children in the ocean endseq',
'startseq a group of youngsters swim in lake water endseq',
'startseq many children are playing and swimming in the water endseq',
'startseq several people swim in a body of water endseq']

We save all the bottleneck train features in a Python dictionary and save it on the disk using Pickle file, namely "**encoded_train_images.pkl**" whose keys are image names and values are corresponding 2048 length feature vector.

Similarly we encode all the test images and save them in the file "Encoded_test_images.pkl".

Data preprocessing – captions:-

We must note that captions are something that we want to predict. So during the training period, captions will be the target variables (Y) that the model is learning to predict.

But the prediction of the entire caption, given the image does not happen at once. We will predict the caption **word by word**. Thus, we need to encode each word into a fixed sized vector. However, this part will be seen later when we look at the model design, but for now we will create two Python Dictionaries namely "wordtoix" (word to index) and "ixtoword" (index to word).

Stating simply, we will represent every unique word in the vocabulary by an integer (index). As seen above, we have 1652 unique words in the corpus and thus each word will be represented by an integer index between 1 to 1652.

Data Preparation using Generator Function:-

This is one of the most important steps in this case study. Here we will understand how to prepare the data in a manner which will be convenient to be given as input to the deep learning model. Hereafter, I will try to explain the remaining steps by taking a sample example as follows:

Consider we have 3 images and their 3 corresponding captions as follows:

(Train image 1) Caption -> The black cat sat on grass

(Train image 2) Caption -> The white cat is walking on road

(Test image) Caption -> The black cat is walking on grass

Now, let's say we use the **first two images** and their captions to **train** the model and the **third image** to **test** our model.

Now the questions that will be answered are: how do we frame this as a supervised learning problem?, what does the data matrix look like? how many data points do we have?

First we need to convert both the images to their corresponding 2048 length feature vector as discussed above. Let "**Image_1**" and "**Image_2**" be the feature vectors of the first two images respectively

Secondly, let's build the vocabulary for the first two (train) captions by adding the two tokens "startseq" and "endseq" in both of them: (Assume we have already performed the basic cleaning steps)

Caption_1 -> **"startseq the black cat sat on grass endseq"**

Caption_2 -> **"startseq the white cat is walking on road endseq"**

vocab = {black, cat, endseq, grass, is, on, road, sat, startseq, the, walking, white}

Let's give an index to each word in the vocabulary:

black -1, cat -2, endseq -3, grass -4, is -5, on -6, road -7, sat -8, startseq -9, the -10, walking -11, white -12

Now let's try to frame it as a **supervised learning problem** where we have a set of data points D = {Xi, Yi}, where Xi is the feature vector of data point 'i' and Yi is the corresponding target variable.

Let's take the first image vector **Image_1** and its corresponding caption "**startseq the black cat sat on grass endseq**". Recall that, Image vector is the input and the caption is what we need to predict. But the way we predict the caption is as follows:

For the first time, we provide the image vector and the first word as input and try to predict the second word, i.e.:

Input = Image_1 + 'startseq'; Output = 'the'

Then we provide image vector and the first two words as input and try to predict the third word, i.e.:

Input = Image_1 + 'startseq the'; Output = 'cat'

Thus, we can summarize the data matrix for one image and its corresponding caption as follows:

	Xi		Yi
i	Image feature vector	Partial Caption	Target word
1	Image_1	startseq	the
2	Image_1	startseq the	black
3	Image_1	startseq the black	cat
4	Image_1	startseq the black cat	sat
5	Image_1	startseq the black cat sat	on
6	Image_1	startseq the black cat sat on	grass
7	Image_1	startseq the black cat sat on grass	endseq

So the most important point :-

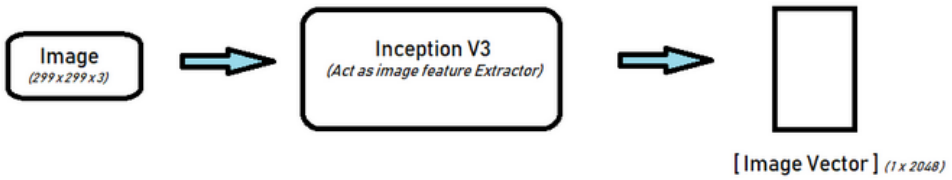
We must now understand that in every data point, it's not just the image which goes as input to the system, but also, a partial caption which helps to predict the next word in the sequence.

Since we are processing sequences, we will employ a Recurrent Neural Network to read these partial captions

```python
def data_generator(descriptions, photos, MAX_LENGTH,VOCAB_SIZE, num_photos_per_batch):
    X1, X2, y = list(), list(), list()
    n=0
    # loop for ever over images
    while 1:
        for key, desc_list in train_descriptions.items():
            n+=1
            # retrieve the photo feature
            photo = photos[key]
            for desc in desc_list:
                seq = token.texts_to_sequences([desc])
                seq = seq[0]
                for i in range(1,len(seq)):
                    in_seq , op_seq = seq[:i],seq[i]
                    #converting input sequence to fix length
                    in_seq = pad_sequences([in_seq],maxlen=MAX_LENGTH,padding="post")[0]
                    # converting op_seq to vocabulary size
                    op_seq = to_categorical([op_seq],num_classes=VOCAB_SIZE)[0]
                    X1.append(photo)
                    X2.append(in_seq)
                    y.append(op_seq)
            # yield the batch data
            if n==num_photos_per_batch:
                yield [[array(X1), array(X2)], array(y)]
                X1, X2, y = list(), list(), list()
                n=0
```

Sequential Data preparation:-

For this we will first feed the image to inception and get feature extracted 2048 dimensional vector.



convert the sequence to numerical with the help of vocabulary. Below is the source code for dataset preparation

```python
class Vocabulary:
    def __init__(self, freq_threshold):

        self.itos = {0: "<PAD>", 1: "<SOS>", 2: "<EOS>", 3: "<UNK>"}
        self.stoi = {"<PAD>": 0, "<SOS>": 1, "<EOS>": 2, "<UNK>": 3}

        self.freq_threshold = freq_threshold

    def __len__(self):
        return len(self.itos)

    @staticmethod
    def tokenizer_eng(text):
        return [tok.text.lower() for tok in spacy_eng.tokenizer(text)]

    def build_vocabulary(self,sentences):
        idx = 4
        frequency = {}

        for sentence in sentences:
            for word in self.tokenizer_eng(sentence):
                if word not in frequency:
                    frequency[word] = 1
                else:
                    frequency[word] += 1

                if (frequency[word] > self.freq_threshold-1):
                    self.itos[idx] = word
                    self.stoi[word] = idx
                    idx += 1

    def numericalize(self,sentence):
        tokenized_text = self.tokenizer_eng(sentence)
```

Architecture details of System 1:-

Encoder function we have defined for LSTM :-

```
class encoderCNN(nn.Module):
    def __init__(self, embed_size, should_train=False):
        super(encoderCNN, self).__init__()
        self.should_train = should_train
        self.inception = models.inception_v3(pretrained=True, aux_logits=False)
        self.inception.fc = nn.Linear(self.inception.fc.in_features, embed_size)
        self.dropout= nn.Dropout(0.5)
        self.relu = nn.ReLU()

    def forward(self, x):
        features = self.inception(x)

        for name, param in self.inception.named_parameters():
            param.requires_grad = False

#         for name, param in self.inception.named_parameters():
#             if "fc.weight" in name or "fc.bias" in name:
#                 param.requires_grad = True
#             else:
#                 param.required_grad = self.should_train


        return self.dropout(self.relu(features))
```

```python
class CNN2RNN(nn.Module):
    def __init__(self, embed_size, vocab_size, hidden_size, num_layers):
        super(CNN2RNN, self).__init__()
        self.encoderCNN = encoderCNN(embed_size)
        self.decoderRNN = decoderRNN(embed_size, vocab_size, hidden_size, num_layers)

    def forward(self, images, caption):
        x = self.encoderCNN(images)
        x = self.decoderRNN(x, caption)
        return x

    def captionImage(self, image, vocabulary, maxlength=50):
        result_caption = []

        with torch.no_grad():
            x = self.encoderCNN(image).unsqueeze(0)
            states = None

            for _ in range(maxlength):
                hiddens, states = self.decoderRNN.lstm(x, states)
                output = self.decoderRNN.linear(hiddens.squeeze(0))
                predicted = output.argmax(1)
                print(predicted.shape)
                result_caption.append(predicted.item())
                x = self.decoderRNN.embedding(output).unsqueeze(0)

                if vocabulary.itos[predicted.item()] == "<EOS>":
                    break
        return [vocabulary.itos[i] for i in result_caption]
```

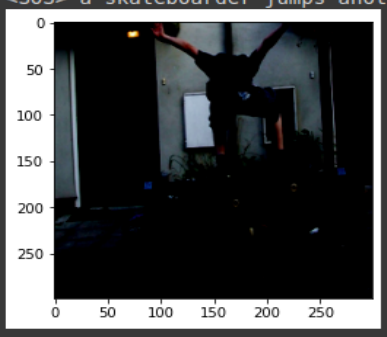
```python
class decoderRNN(nn.Module):
    def __init__(self, embed_size,vocab_size, hidden_size, num_layers):
        super(decoderRNN, self).__init__()
        self.embedding = nn.Embedding(vocab_size, embed_size)
        self.lstm = nn.LSTM(embed_size, hidden_size, num_layers)
        self.linear = nn.Linear(hidden_size, vocab_size)
        self.dropout = nn.Dropout(0.5)

    def forward(self, features, caption):
        embeddings = self.dropout(self.embedding(caption))
        embeddings = torch.cat((features.unsqueeze(0),embeddings), dim=0)
        hiddens, _ = self.lstm(embeddings)
        outputs = self.linear(hiddens)
        return outputs
```

(parameter) self: decoderRNN

Random example showing with caption :-

```
import matplotlib.pyplot as plt

plt.imshow(x.permute(1,2,0))
print(y)

print(dataset.vocab.itos[1])

for i in y:
    print(dataset.vocab.itos[int(i)],end=" ")
```

```
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integers).
tensor([     1, 108926, 102607, 108623, 108151, 102645, 108925,      2])
<SOS>
<SOS> a skateboarder jumps another skateboard . <EOS>
```



Training the Model and defining the Hyperparamaeters:-
Embed_size = 256
Hidden_aize = 256
num_layers(number of layers) =5
Num_epochs = 5
Learnin_rate = 3e-4

```
#Training the model
```

```
from tqdm import tqdm
# from torchvision.utils.tensorboard import SummaryWriter
```

```
torch.backends.cudnn.benchmark = True
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
load_model = False
save_model=False
train_CNN = False
```

```
# Hyperparameters
import torch.optim as optim

step = 0
embed_size = 256
hidden_size = 256
num_layers = 5
num_epochs = 5
learning_rate = 3e-4
vocab_size = len(dataset.vocab)
```

```
model = CNN2RNN(embed_size=embed_size, hidden_size=hidden_size,vocab_size=vocab_size, num_layers=num_layers).to(device=device)
```

Decoder Description :-

```
class decoderRNN(nn.Module):
    def __init__(self, embed_size,vocab_size, hidden_size, num_layers):
        super(decoderRNN, self).__init__()
        self.embedding = nn.Embedding(vocab_size, embed_size)
        self.lstm = nn.LSTM(embed_size, hidden_size, num_layers)
        self.linear = nn.Linear(hidden_size, vocab_size)
        self.dropout = nn.Dropout(0.5)

    def forward(self, features, caption):
        embeddings = self.dropout(self.embedding(caption))
        embeddings = torch.cat((features.unsqueeze(0),embeddings), dim=0)
        hiddens, _ = self.lstm(embeddings)
        outputs = self.linear(hiddens)
        return outputs
```

```
model.decoderRNN
```

```
decoderRNN(
  (embedding): Embedding(108931, 256)
  (lstm): LSTM(256, 256, num_layers=5)
  (linear): Linear(in_features=256, out_features=108931, bias=True)
  (dropout): Dropout(p=0.5, inplace=False)
)
```

Model training and then saving the weights or checkpoints after every epochs :-

```
model.train()

for epoch in range(num_epochs):
    if save_model:
        checkpoint = {
            "state_dict": model.state_dict(),
            "optimizer": model.state_dict(),
            "step": step,
        }
        save_checkpoint(checkpoint)

#     for idx, (imgs, captions) in tqdm(
#         enumerate(loader), total=len(loader), leave=False
#     ):
    for idx, (imgs, captions) in enumerate(loader):
        imgs = imgs.to(device)
        captions = captions.to(device)

        score = model(imgs, captions[:-1])

#         print(score.shape, captions.shape)
#         print(score.reshape(-1, score.shape[2]).shape, captions.reshape(-1).shape)
#         print("why are we reshaping it here?")
        optimizer.zero_grad()
        loss = loss_criterion(score.reshape(-1, score.shape[2]), captions.reshape(-1))


        step += 1

        loss.backward()
        optimizer.step()
    print(f"Loss for epoch {epoch}: {loss}")
```

objective and subjective results with System 1 and its analysis

BLEU :-

BLEU stands for Bilingual Evaluation Understudy.

It is an algorithm, which has been used for evaluating the quality of machine translated text. We can use BLEU to check the quality of our generated caption.

- BLEU is language independent

- Easy to understand
- It is easy to compute.
- It lies between [0,1]. Higher the score better the quality of caption

How to calculate BLEU score ?

predicted caption= “the weather is good”

references:

1. the sky is clear
2. the weather is extremely good

First, convert the predicted caption and references to unigram/bigrams.

$$\text{modified ngram precision} = \frac{\text{max number of times ngram occurs in referance}}{\text{total number of ngrams in hypothesis}}$$

BLEU tells how good is our predicted caption as compared to the provided 5 reference captions.

Predicted captions with the real one captions which is not that much improved captions as real one since we have trained only on 30 epochs as it was taking huge amount of time even using GPU on colab and also we are restricted to use GPU on colab :

```
/usr/local/lib/python3.7/dist-packages/torch/utils/data/dataloader.py:490: UserWarning: This DataLoader will create 8 wor
  cpuset_checked))
Image name: 106490881_5a2dd9b7bd.jpg
Real caption:  ['<SOS>', 'a', 'boy', 'in', 'his', 'blue', 'swim', 'shorts', 'at', 'the', 'beach', '.', '<EOS>']
Predicted caption:  ['<SOS>', 'a', 'young', 'boy', 'a', 'a', 'a', 'is', 'the', 'the', 'the', 'the', 'the', '<EOS>']


Image name: 1107246521_d16a476380.jpg
Real caption:  ['<SOS>', 'a', 'black', 'dog', 'jumping', 'to', 'catch', 'a', 'rope', 'toy', '<EOS>']
Predicted caption:  ['<SOS>', 'a', 'black', 'dog', 'is', 'a', 'a', '.', '<EOS>']


Image name: 1122944218_8eb3607403.jpg
Real caption:  ['<SOS>', 'a', 'baby', 'in', 'a', 'white', 'garment', 'holds', 'a', 'flag', 'with', '<UNK>', 'moon', 'and'
Predicted caption:  ['<SOS>', 'a', 'young', 'boy', 'a', 'a', 'a', 'a', 'and', 'his', 'his', '.', '<EOS>']


Image name: 1131800850_89c7ffd477.jpg
Real caption:  ['<SOS>', 'a', 'brown', 'and', 'white', 'dog', 'stands', 'outside', 'while', 'it', 'snows', '.', '<EOS>']
Predicted caption:  ['<SOS>', 'a', 'dogs', 'and', 'white', 'a', 'a', '.', 'in', 'the', '.', '<EOS>']


Image name: 1131932671_c8d17751b3.jpg
Real caption:  ['<SOS>', 'a', 'boy', 'is', 'jumping', 'on', 'a', 'bed', '.', '<EOS>']
Predicted caption:  ['<SOS>', 'two', 'children', 'in', 'a', 'a', 'a', 'a', 'in', 'the', '<EOS>']
```

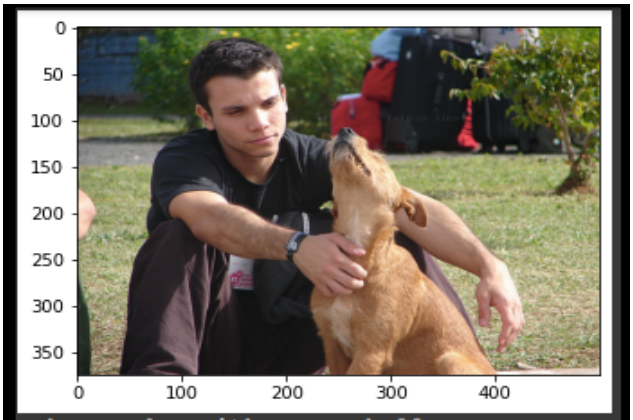
```python
base_path = '/content/drive/MyDrive/test/'
def showAndCaptionImage(img, model):

    img = Image.open(base_path + img).convert("RGB")
    plt.imshow(img)
    plt.show()
    img = transform(img)
    caption = model.caption_image(img.unsqueeze(0).to(device), dataset.vocab)[1:-1]
    captionStr = ""
    for e in caption:
        captionStr += e + " "
    print(captionStr)

subjective_images = ['test_1.jpg','test_2.jpg','test_3.jpg','test_4.jpg','test_5.jpg']
for image in subjective_images:
    showAndCaptionImage(image, model)
```

```
a child boy in a a and and and
```

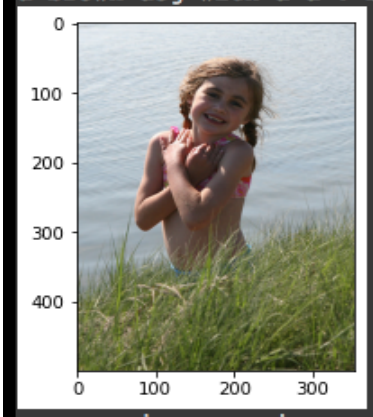
✓ 4s completed at 19:44

```
a man is a a a a a
```

```
a dog dog with a a beach beach the beach beach
```



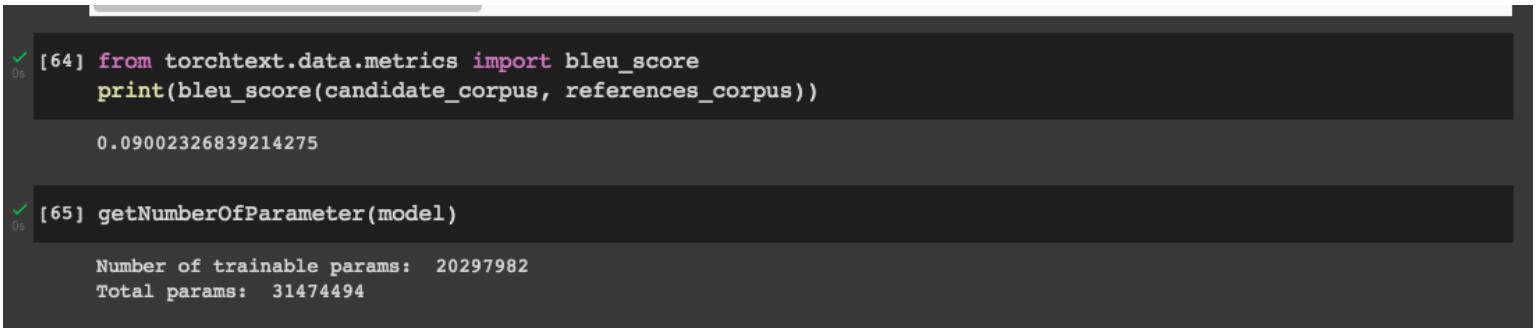
```
a brown dog with a a . ball a a . .
```



```
a woman in a a a in a a a on
```

Realisation of BLEU score :-

Its coming too low which is 0.09 but i have also tried by changing hyperparameters which improved somewhat BLEU score but not that too much like of i got also 0.11 and also 0.23 but i have tried also on multiple pretrained model weights and calculated BLEU score which i give as chart below:-



Observations :-

The following table summarizes our results for our training experiments with 10 epochs.

We also experimented with different resnet models.

- **Layers column- number of layers in the lstm cell**
- **Input column- input size to lstm model**
- **Hidden column- hidden layer size**

CNN	Input	Hidden	Layers	BLEU
RESNET-18	512	512	1	0.064
RESNET-50	128	256	1	0.071
RESNET-50	512	512	1	0.070
RESNET-152	256	1024	1	0.065
RESNET-152	512	512	2	0.091

We were able to reach a bleu score of 0.11 using RESNET-50, LSTM model with input size 128 and hidden size 256 and 1 layer in lstm. The model was trained for 40 epochs on training set size-7091 with mini batch size of 64 and test set size-1000 with batch size 32 was used for evaluation.

Important points to note & conclusions :-

We must understand that the images used for testing must be semantically related to those used for training the model. For example, if we train our model on the images of cats, dogs, etc. we must not test it on images of air planes, waterfalls, etc. This is an example where the distribution of the train and test sets will be very different and in such cases no Machine Learning model in the world will give good performance.

I have just simply used the CNN-LSTM network approach for image captioning but due to huge datasets and LSTM network we are unable to run the training of our models on larger no of epochs due to which my BLEU score is coming less and also loss per epoch is behaving inconsistent so a and a lot of modifications can be made to improve this solution like:

1. Using a **larger** dataset.
2. Changing the model architecture, like include an **attention** module.
3. Doing more **hyper parameter tuning** (learning rate, batch size, number of layers, number of units, dropout rate, batch normalization etc.).
4. Use the cross validation set to understand **overfitting**.
5. Using **Beam Search** instead of Greedy Search during Inference.

2nd Part

So firstly the **architectural changes which I have tried before the sir's clarification on Question -2.**

I have added the Attention Mechanism on CNN-LSTM Memory_Constrained architecture of image captioning which is one of the architectural changes.

We have explored and read many articles from where we got to know about the Attention Mechanism addition which led to architectural changes in the CNN-LSTM architecture of image captioning.

We observed it was having one of the great performances on Image captioning.

Attention mechanisms we have used and studied we will discuss later .

So first now we will discuss according to what sir said that without any change in architecture of system-1(that is CNN-LSTM architecture) what can we do to improve our CNN-LSTM model accuracy

So here i am using Image Captioning using InceptionV3 and Beam Search(addition in our system-1 i.e. CNN-LSTM architecture)

- In Image Captioning, a **CNN is used to extract the features from an ima**ge which is then along with the captions is fed into an RNN.

- To extract the features, we use a model trained on Imagenet. I tried out VGG-16, Resnet-50 and InceptionV3. Vgg16 has almost 134 million parameters and its top-5 error on Imagenet is 7.3%.
- InceptionV3 has 21 million parameters and its top-5 error on Imagenet is 3.46%. Human top-5 error on Imagenet is 5.1%.

We have used VGG-16 as my first model for extracting the features. It **took a long time to extract features from 6000** training images on google collab . This is very slow.

Finally, we have used InceptionV3. Since it has very few parameters as compared to VGG-16, it took less time for InceptionV3 to extract features from 6000 images.

Datasets and how to obtain into train images after removing newline :-

```
img[:5]
```

```
['Flickr8k_Dataset/Flicker8k_Dataset/17273391_55cfc7d3d4.jpg',
 'Flickr8k_Dataset/Flicker8k_Dataset/2890075175_4bd32b201a.jpg',
 'Flickr8k_Dataset/Flicker8k_Dataset/3356642567_f1d92cb81b.jpg',
 'Flickr8k_Dataset/Flicker8k_Dataset/186890605_ddff5b694e.jpg',
 'Flickr8k_Dataset/Flicker8k_Dataset/2773682293_3b712e47ff.jpg']
```

```
train_images_file = 'Flickr8k_text/Flickr_8k.trainImages.txt'
```

```
train_images = set(open(train_images_file, 'r').read().strip().split('\n'))
```

```
def split_data(l):
    temp = []
    for i in img:
        if i[len(images):] in l:
            temp.append(i)
    return temp
```

Getting the training ,validation and test images of all the images that is 6000 training images, 1000 validation images and 1000 test images :-

```
# Getting the training images from all the images
train_img = split_data(train_images)
len(train_img)
```

```
6000
```

```
val_images_file = 'Flickr8k_text/Flickr_8k.devImages.txt'
val_images = set(open(val_images_file, 'r').read().strip().split('\n'))
```

```
# Getting the validation images from all the images
val_img = split_data(val_images)
len(val_img)
```

```
1000
```

```
test_images_file = 'Flickr8k_text/Flickr_8k.testImages.txt'
test_images = set(open(test_images_file, 'r').read().strip().split('\n'))
```

```
# Getting the testing images from all the images
test_img = split_data(test_images)
len(test_img)
```

```
1000
```

We will feed these images to VGG-16 to get the encoded images. Hence we need to preprocess the images .

The last layer of VGG-16 is the softmax classifier(FC layer with 1000 hidden neurons) which returns the probability of a class.

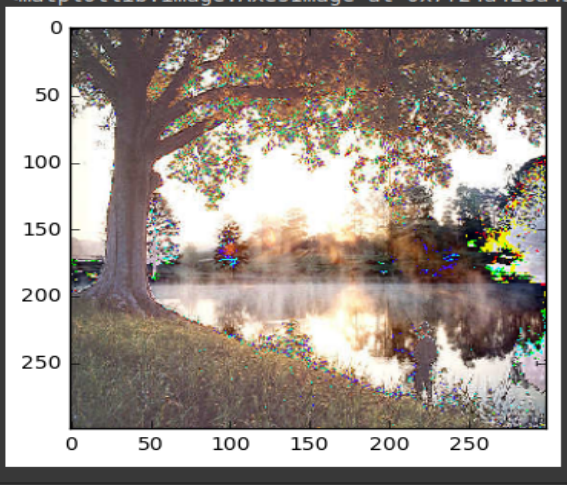
This layer should be removed so as to get a feature representation of an image. We will use the **last Dense layer(4096 hidden neurons)** after popping the classifier layer. Hence the shape of the encoded image will be (1, 4096)

```
[ ] def preprocess_input(x):
        x /= 255.
        x -= 0.5
        x *= 2.
        return x
```

```
[ ] def preprocess(image_path):
        img = image.load_img(image_path, target_size=(299, 299))
        x = image.img_to_array(img)
        x = np.expand_dims(x, axis=0)

        x = preprocess_input(x)
        return x
```

```
plt.imshow(np.squeeze(preprocess(train_img[0])))
```

```
<matplotlib.image.AxesImage at 0x7f24a426a438>
```




```
[ ] model = InceptionV3(weights='imagenet')
```

```
[ ] from keras.models import Model

    new_input = model.input
    hidden_layer = model.layers[-2].output

    model_new = Model(new_input, hidden_layer)
```

```
[ ]
```

```
tryi = model_new.predict(preprocess(train_img[0]))
```

```
[ ] tryi.shape
```

```
(1, 2048)
```

Calculating the unique words in the vocabulary:-Calculating the unique words in the vocabulary:-

```
[ ] caps = []
    for key, val in train_d.items():
        for i in val:
            caps.append('<start> ' + i + ' <end>')
```

```
[ ] words = [i.split() for i in caps]
```

```
[ ] unique = []
    for i in words:
        unique.extend(i)
```

```
[ ] unique = list(set(unique))
```

```
[ ] # with open("unique.p", "wb") as pickle_d:
    #     pickle.dump(unique, pickle_d)
```

```
    unique = pickle.load(open('unique.p', 'rb'))
```

```
[ ] len(unique)
```

```
8256
```

Mapping the unique words to indices and vice-versa and the calculating the maximum length among all the captions :-

Mapping the unique words to indices and vice-versa

```
[ ] word2idx = {val:index for index, val in enumerate(unique)}
```

```
[ ] word2idx['<start>']
```

```
5553
```

```
[ ] idx2word = {index:val for index, val in enumerate(unique)}
```

```
idx2word[5553]
```

```
'<start>'
```

Calculating the maximum length among all the captions

```
[ ] max_len = 0
    for c in caps:
        c = c.split()
        if len(c) > max_len:
            max_len = len(c)
    max_len
```

```
40
```

```
[ ] len(unique), max_len
```

```
(8256, 40)
```

Adding and to all the captions to indicate the starting and ending of a sentence. This will be used while we predict the caption of an image

Example

```
[ ] a = c[-1]
    a, imgs[-1]

    ('<start> Two rafts overturn in a river . <end>', '3421928157_69a325366f.jpg')

    for i in a.split():
        print (i, "=>", word2idx[i])

    <start> => 5553
    Two => 2666
    rafts => 4606
    overturn => 3779
    in => 8156
    a => 32
    river => 1816
    . => 7023
    <end> => 5232
```

Generator

We will use the encoding of an image and use a start word to predict the next word.

After that, we will again use the same image and use the predicted word to predict the next word.

So, the image will be used at every iteration for the entire caption. This is how we will generate the caption for an image. Hence, we need to create a custom generator for that.

```
def data_generator(batch_size = 32):
        partial_caps = []
        next_words = []
        images = []

        df = pd.read_csv('flickr8k_training_dataset.txt', delimiter='\t')
        df = df.sample(frac=1)
        iter = df.iterrows()
        c = []
        imgs = []
        for i in range(df.shape[0]):
            x = next(iter)
            c.append(x[1][1])
            imgs.append(x[1][0])


        count = 0
        while True:
            for j, text in enumerate(c):
                current_image = encoding_train[imgs[j]]
                for i in range(len(text.split())-1):
                    count+=1

                    partial = [word2idx[txt] for txt in text.split()[:i+1]]
                    partial_caps.append(partial)

                    # Initializing with zeros to create a one-hot encoding matrix
                    # This is what we have to predict
                    # Hence initializing it with vocab_size length
                    n = np.zeros(vocab_size)
```

Let's create the model now :-

```
[ ] embedding_size = 300
```

Input dimension is 4096 since we will feed it the encoded version of the image.

```
image_model = Sequential([
        Dense(embedding_size, input_shape=(2048,), activation='relu'),
        RepeatVector(max_len)
    ])
```

Since we are going to predict the next word using the previous words(length of previous words changes with every iteration over the caption), we have to set return_sequences = True.

```
[ ] caption_model = Sequential([
        Embedding(vocab_size, embedding_size, input_length=max_len),
        LSTM(256, return_sequences=True),
        TimeDistributed(Dense(300))
    ])
```

Merging the models and creating a softmax classifier

```
[ ] final_model = Sequential([
        Merge([image_model, caption_model], mode='concat', concat_axis=1),
        Bidirectional(LSTM(256, return_sequences=False)),
        Dense(vocab_size),
        Activation('softmax')
    ])
```

Architecture details of System 1 Modified

Training and Hyperparameters

For creating the model, the captions have to be put in an embedding. We wanted to try Word2Vec to get the pre-trained embedding weights of my vocabulary, but it didn't pan out. So, I took some ideas from it by setting the embedding size to 300.

Following is the image we have used in the model.

- The optimizer used was RMSprop and the batch size was set to 128.

- I trained the model using the VGG-16 extracted features for about 50 epochs and got a loss value of **2.77**
- We tried learning rate annealing, changing the optimizer, changing the model architecture, the embedding size, number of LSTM units and almost every other hyperparameter.

We tried changing **BATCH SIZE from 128 to 256** and the loss was significantly dropped.

- The reason changing the batch size worked was because if the batch size is small, the **gradients are an approximation of the real gradients**. So, it will take longer to find a good solution.

- Moreover, increasing my batch size decreased by training time. First at batch size of 128 it took approximately 1000 seconds for an epoch. With a batch size of 2048, it took me 343 seconds per epoch.

Final model summary :-

```
[ ] final_model.compile(loss='categorical_crossentropy', optimizer=RMSprop(), metrics=['accuracy'])

final_model.summary()

_________________________________________________________________________________________________
Layer (type)                 Output Shape              Param #     Connected to
=================================================================================================
dense_1 (Dense)              (None, 300)               614700
_________________________________________________________________________________________________
repeatvector_1 (RepeatVector)(None, 40, 300)           0
_________________________________________________________________________________________________
embedding_1 (Embedding)      (None, 40, 300)           2476800
_________________________________________________________________________________________________
lstm_1 (LSTM)                (None, 40, 256)           570368
_________________________________________________________________________________________________
timedistributed_1 (TimeDistribut (None, 40, 300)       77100
_________________________________________________________________________________________________
bidirectional_1 (Bidirectional)  (None, 512)           1140736     merge_1[0][0]
_________________________________________________________________________________________________
dense_3 (Dense)              (None, 8256)              4235328     bidirectional_1[0][0]
_________________________________________________________________________________________________
activation_1 (Activation)    (None, 8256)              0           dense_3[0][0]
=================================================================================================
Total params: 9,115,032
Trainable params: 9,115,032
Non-trainable params: 0
_________________________________________________________________________________________________
```

Saving the weights of model after applying fit generator :-

```
[ ] final_model.save_weights('time_inceptionV3_3.15_loss.h5')

final_model.fit_generator(data_generator(batch_size=128), samples_per_epoch=samples_per_epoch, nb_epoch=1,
                          verbose=2)

Epoch 1/1
992s - loss: 3.1449 - acc: 0.4643
/usr/local/lib/python3.5/site-packages/keras/engine/training.py:1573: UserWarning: Epoch comprised more than `samples_per_epoch` samples, which might affect lear
  warnings.warn('Epoch comprised more than '
<keras.callbacks.History at 0x7f7632732a90>
```

Now we are creating the function to apply beam search prediction during inference :-

```
def beam_search_predictions(image, beam_index = 3):
    start = [word2idx["<start>"]]

    start_word = [[start, 0.0]]

    while len(start_word[0][0]) < max_len:
        temp = []
        for s in start_word:
            par_caps = sequence.pad_sequences([s[0]], maxlen=max_len, padding='post')
            e = encoding_test[image[len(images):]]
            preds = final_model.predict([np.array([e]), np.array(par_caps)])

            word_preds = np.argsort(preds[0])[-beam_index:]

            # Getting the top <beam_index>(n) predictions and creating a
            # new list so as to put them via the model again
            for w in word_preds:
                next_cap, prob = s[0][:], s[1]
                next_cap.append(w)
                prob += preds[0][w]
                temp.append([next_cap, prob])

        start_word = temp
        # Sorting according to the probabilities
        start_word = sorted(start_word, reverse=False, key=lambda l: l[1])
        # Getting the top words
        start_word = start_word[-beam_index:]
```

Predictions

We have used 2 methods for predicting the captions.

- **Argmax Search** is where the maximum value index(argmax) in the 8256 long predicted vector is extracted and appended to the result. This is done until we hit `<end>` or the maximum length of the caption.
- **Beam Search** is where **we take top k predictions,** f**eed them again in the model and then sort them using the probabilities returned by the model.** So, the list will always contain the top **k** predictions. In the end, we take the one with the highest probability and go through it till we encounter `<end>` or reach the maximum caption length.

Finally, here are some results that we got. The rest of the results are in the python notebook.

objective and subjective results on System 1 Modified and their analysis:-

Result on some of the test image we are getting :-

```
[ ] try_image2 = test_img[7]
    Image.open(try_image2)
```

```
[ ] print ('Normal Max search:', predict_captions(try_image2))
    print ('Beam Search, k=3:', beam_search_predictions(try_image2, beam_index=3))
    print ('Beam Search, k=5:', beam_search_predictions(try_image2, beam_index=5))
    print ('Beam Search, k=7:', beam_search_predictions(try_image2, beam_index=7))
```

```
Normal Max search: A snowboarder flies through the air after midair from a mountain .
Beam Search, k=3: A skier is performing a trick high in the air over a snowy area .
Beam Search, k=5: A downhill skier races near trees .
Beam Search, k=7: This person is snowboarding off a ramp .
```

```
im = 'Flickr8k_Dataset/Flicker8k_Dataset/3316725440_9ccd9b5417.jpg'
print ('Normal Max search:', predict_captions(im))
print ('Beam Search, k=3:', beam_search_predictions(im, beam_index=3))
print ('Beam Search, k=5:', beam_search_predictions(im, beam_index=5))
print ('Beam Search, k=7:', beam_search_predictions(im, beam_index=7))
Image.open(im)
```

```
Normal Max search: A man rides a bicycle on a trail down a river .
Beam Search, k=3: A man is riding a bicycle on a trail through some trees .
Beam Search, k=5: A man rides a mountain bike down a slope in the woods .
Beam Search, k=7: A man rides a bicycle on a trail down a river .
```

```
[ ] im = 'Flickr8k_Dataset/Flicker8k_Dataset/2542662402_d781dd7f7c.jpg'
    print ('Normal Max search:', predict_captions(im))
    print ('Beam Search, k=3:', beam_search_predictions(im, beam_index=3))
    print ('Beam Search, k=5:', beam_search_predictions(im, beam_index=5))
    print ('Beam Search, k=7:', beam_search_predictions(im, beam_index=7))
    Image.open(im)
```

```
Normal Max search: A small dog jumping an obstacle in a grassy field .
Beam Search, k=3: A small dog jumping an obstacle in a grassy field .
Beam Search, k=5: A small dog jumping an obstacle in a grassy field .
Beam Search, k=7: A small dog jumping an obstacle in a grassy field .
```

Next Steps(which we have tried for self learning)

Attention has been proven to be very effective in a plethora of tasks including image captioning.

Implementing attention in my model has lead to an improvement:-

MODEL

We will be dividing the problem into below modules.

• **Encoder using resnet18:** We will encode the **images into 512x14x14 tensor. We chose this model because it has a small size (44 Mb) and approximately 11 million parameters.** This leaves room for lstm to have 14 million parameters. We are constrained by the size of the model i.e it should be below 100 Mb in size which amounts to 25 million parameters assuming float data type.

• **Attention mechanism:** Attention is way for a model to choose only those parts of the encoding that it thinks is relevant to the task at hand. We use activation function as ReLU and softmax function with dimension 1.

• **Decoder using LSTM cell:** We will use a single LSTMCell instead of the LSTM from pytorch to have more control on the behaviour modelling.

• **Teacher forcer mechanism:** Teacher forcing is a method for quickly and efficiently training recurrent neural network models that use the ground truth from a prior time step

```python
class Attention(nn.Module):
    def __init__(self, encoder_dim, decoder_dim, attention_dim):
        super(Attention, self).__init__()
        self.encoder_att = nn.Linear(encoder_dim, attention_dim)
        self.decoder_att = nn.Linear(decoder_dim, attention_dim)
        self.full_att = nn.Linear(attention_dim, 1)  # linear layer to calculate values to be softmax-ed
        self.relu = nn.ReLU()
        self.softmax = nn.Softmax(dim=1)

    def forward(self, encoder_out, decoder_hidden):
        att1 = self.encoder_att(encoder_out)
        att2 = self.decoder_att(decoder_hidden)
        att = self.full_att(self.relu(att1 + att2.unsqueeze(1))).squeeze(2)  # (batch_size, num_pixels)
        alpha = self.softmax(att)  # (batch_size, num_pixels)
        attention_weighted_encoding = (encoder_out * alpha.unsqueeze(2)).sum(dim=1)  # (batch_size, encoder_dim)

        return attention_weighted_encoding, alpha
```

How does Attention enhance the Image Caption performance.

The earliest Image Caption architectures included the other three components without Attention. Let's first go over how that model works, and then see what is different with Attention.

At each timestep, the Decoder takes the hidden state from the previous timestep and the current input word to produce the output word for this timestep. The hidden state carries some representation of the encoded image features.

In the absence of Attention, the Decoder treats all parts of the image equally while generating the output word.

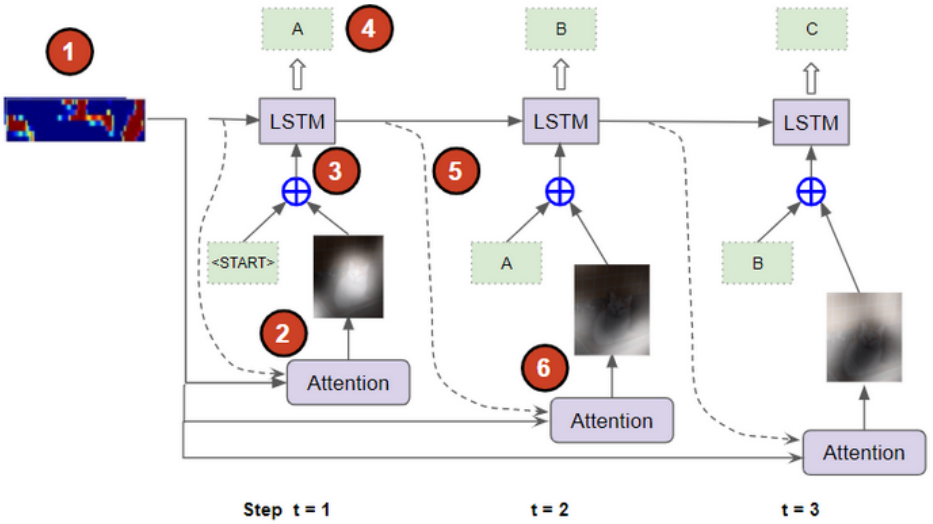
So how does **Attention** behave differently?

- **At each timestep, the Attention module takes the encoded image as input along with the Decoder's hidden state for the previous timestep.**
- **It produces an Attention Score that assigns a weight** to each pixel of the encoded image. The higher the weight for a pixel, the more relevant it is for the word to be output at the next timestep.
- For example, if the target output sequence is "A girl is eating an apple", the girl's pixels in the photo are highlighted when generating

the word “girl”, while the apple’s pixels are highlighted for the word “apple”.

- This **Score is then concatenated with the input word for that timestep, and fed to the Decoder.** This helps the Decoder to focus on the most relevant parts of the picture and generate the appropriate output word.

A TYPICAL REPRESENTATION OF ATTENTION :-



Adding Attention into class Decoder_RNN:-

```
for t in range(decode_length):
    if teacher:
        inputEmbeddings = embeddings[:, t, :]
    else:
        inputEmbeddings = self.embedding(preds)[:, 0, :]

    attention_weighted_encoding, alpha = self.attention(encoder_out, h)
    gate = self.sigmoid(self.f_beta(h))  # gating scalar, (batch_size_t, encoder_dim)
    attention_weighted_encoding = gate * attention_weighted_encoding

    h, c = self.decode_step(torch.cat([inputEmbeddings, attention_weighted_encoding], dim=1), (h, c))
    preds = self.fc(self.dropout(h))  # (batch_size_t, vocab_size)
    predictions[:, t, :] = preds
    preds = preds.argmax(1).unsqueeze(1)
    alphas[:, t, :] = alpha
```

Process sequence over multiple timesteps:-

Next, we iterate through each element of the input sequence over multiple timesteps. Let's look at the flow for one sequence from the batch:

- The Attention Module takes the encoded image from the Encoder, and **the hidden state from the Sequence Decoder** and computes the weighted Attention Score.
- The input sequence is passed through the Embedding layer and then combined with the Attention Score.
- The combined input sequence is fed to the Sequence Decoder, which produces an output sequence along with a new hidden state.
- The Sentence Generator processes the output sequence and generates its predicted word probabilities.
- **We now repeat this cycle for the next timestep. The Decoder's new hidden state from this time step is used for the next timestep**. We continue doing this until an 'End' token is predicted or we reach the maximum length of the sequence.

Hyperparameters:-

```python
# Hyperparameters
attention_dim = 512
embed_dim = 512
decoder_dim = 512
dropout = 0.5
vocab_size = len(dataset.vocab)
learning_rate = 1e-03
num_epochs = 20
load_model = True
save_model = True
train_CNN = False
alpha_c = 1
teacher = False

def train():

    #for tensorboard:
    # writer = SummaryWriter("runs/flickr")
    step = 0

    # initialize model, loss etc
    model = CNNtoRNN(attention_dim, embed_dim, decoder_dim, vocab_size,
                     train_CNN=train_CNN, dropout=dropout, teacher=teacher).to(device)
    criterion = nn.CrossEntropyLoss(ignore_index=dataset.vocab.stoi["<PAD>"])
    optimizer = optim.Adam(model.parameters(), lr=learning_rate)
```

Generating Loss after every epoch in the figure below:-

```python
            #Add doubly stochastic attention regularization
            loss += alpha_c * ((1. - alphas.sum(dim=1)) ** 2).mean()

            optimizer.step()

        print('Epoch {} completed with loss {}'.format(epoch+1, loss))


#train()
```

```
Epoch 36 completed with loss 3.4731693267822266
=> Saving checkpoint
Epoch 37 completed with loss 4.567559719085693
=> Saving checkpoint
Epoch 38 completed with loss 3.5936696529388428
=> Saving checkpoint
Epoch 39 completed with loss 3.9755353927612305
=> Saving checkpoint
Epoch 40 completed with loss 4.354511737823486
=> Saving checkpoint
Epoch 41 completed with loss 3.6624350547790527
=> Saving checkpoint
Epoch 42 completed with loss 4.159901142120361
=> Saving checkpoint
Epoch 43 completed with loss 3.413750410079956
=> Saving checkpoint
```

With Attention

We were able to achieve a bleu score of 0.211

References :-

https://towardsdatascience.com/image-captions-with-attention-in-tensorflow-step-by-step-927dad3569fa

https://towardsdatascience.com/image-captioning-with-keras-teaching-computers-to-describe-pictures-c88a46a311b8

https://youtu.be/y2BaTt1fxJU

https://towardsdatascience.com/intuitive-understanding-of-attention-mechanism-in-deep-learning-6c9482aecf4f