**PROJECT DOCUMENTATION**

**ON**

**IMAGE CAPTIONING GENERATOR**

**Submitted in partial fulfillment of the requirements for the award of MCA 2021**

**MASTER OF COMPUTER APPLICATION**

****

**JAWAHARLAL NEHRU UNIVERSITY**

**SCHOOL OF COMPUTER AND SYSTEM SCIENCES**

**NEW DELHI 110067**

**DEC 2020**

**Submitted To:**

Prof. R.K Agrawal

**Submitted By**

Prerna Sharma

|  |  |
| --- | --- |
|  |  |

**Table of Contents**

1. Introduction to Sequence Models
   1. RNNs Architecture
   2. Problems with RNNs
   3. LSTM Architecture
2. Problem Statement
3. Motivation
4. Proposed Project
5. Prerequisites and System Requirements
6. Introduction to the ICG architecture
7. Dataset Used
8. Implementation of the project step by step
   1. Understanding Dataset
   2. Creating a dictionary of captions
   3. Cleaning the data
   4. Create a vocabulary
   5. Loading the training and testing set
   6. Data Pre-processing: Captions
   7. Data Pre-processing: Images
   8. Word Embeddings
   9. Data Preparation using Generator Function
   10. Defining Model
   11. Generating Captions Using Greedy Search
9. Scope of Improvements

**1. Introduction to sequence models**

**Sequence Modelling** is the task of predicting what word/letter comes next. Unlike the FNN (Feed forward Neural Network) and CNN (Convolutional Neural Network), in **sequence modelling**, the current output is dependent on the previous input and the length of the input is not fixed. In this section documentation, we will discuss practical application of **sequence modelling (Image Captioning Generator)**.

Traditional feed forward neural networks do not share features across different positions of the network. In other words, these models assume that all inputs (and outputs) are independent of each other. This model would not work in sequence prediction since the previous inputs are inherently important in predicting the next output. For example, if you were predicting the next word in a stream of text, you would want to know at least a couple of words before the target word.

Traditional neural networks require the input and output sequence lengths to be constant across all predictions. That is why we need a network that supports variable length input and output. Sequence models are widely used for **NLP** (Natural Language Processing) and **time-series**. Model that generates natural language sequences is called a Language Model.

**Language models** make predictions by estimating the probability of the next word given the words that precede it. After you’ve trained a language model, the conditional distributions you’ve estimated may be used to sample novel sequences.

Besides NLP, there are various other applications in other fields. In finance, for instance, you may use this type of model to generate sample stock paths. You could train the network on various 3 minute tick-by-tick intervals for a single name and then use the network to generate sample paths.

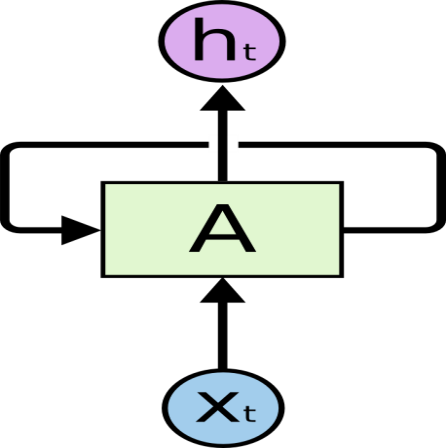
**1.1. RNNs Architecture**

**Recurrent neural network** is a sequence model that also contains short term memory.

Humans don’t start their thinking from scratch every second. As you read this essay, you understand each word based on your understanding of previous words. You don’t throw everything away and start thinking from scratch again. Your thoughts have persistence.

Traditional neural networks can’t do this, and it seems like a major shortcoming. For example, imagine you want to classify what kind of event is happening at every point in a movie. It’s unclear how a traditional neural network could use its reasoning about previous events in the film to inform later ones.

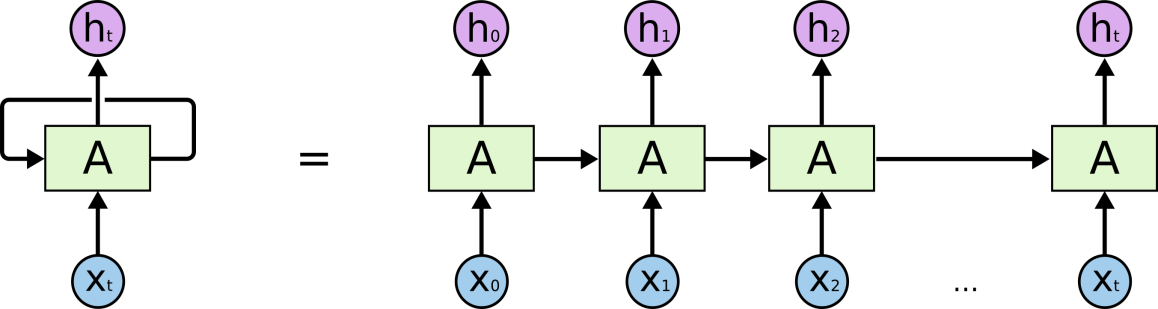
Recurrent neural networks address this issue. They are networks with loops in them, allowing information to persist.



**Recurrent Neural Networks have loops**

In the above diagram, a chunk of neural network, AA, looks at some input **xt**and outputs a value **ht**. A loop allows information to be passed from one step of the network to the next.

These loops make recurrent neural networks seem kind of mysterious. However, if you think a bit more, it turns out that they aren’t all that different than a normal neural network. A recurrent neural network can be thought of as multiple copies of the same network, each passing a message to a successor. Consider what happens if we unroll the loop:



**An unrolled recurrent neural network**

This chain-like nature reveals that recurrent neural networks are intimately related to sequences and lists. They’re the natural architecture of neural network to use for such data.

And they certainly are used! In the last few years, there have been incredible successes applying RNNs to a variety of problems: speech recognition, language modelling, translation, image captioning etc. I’ll leave discussion of the amazing feats one can achieve with RNNs to Andrej Karpathy’s excellent blog post, [The Unreasonable Effectiveness of Recurrent Neural Networks](http://karpathy.github.io/2015/05/21/rnn-effectiveness/). But they really are pretty amazing.

Essential to these successes is the use of “LSTMs,” a very special kind of recurrent neural network which works, for many tasks, much better than the standard version. Almost all exciting results based on recurrent neural networks are achieved with them. It’s these LSTMs that this essay will explore.

**1.2. Problems with RNNs**

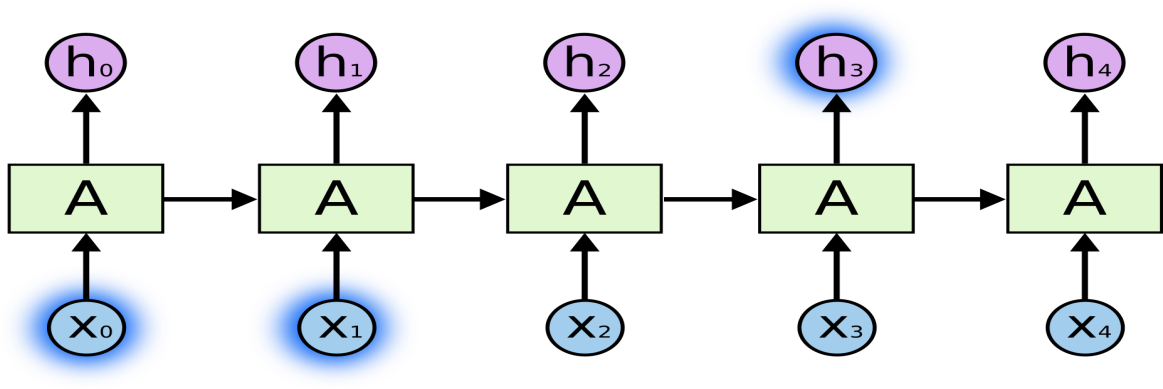
**The Problem of Vanishing Gradients:**

RNNs suffer from the problem of vanishing gradients, which hampers learning of long data sequences. The gradients carry information used in the RNN parameter update and when the gradient becomes smaller and smaller, the parameter updates become insignificant which means no real learning is done. In machine learning, the **vanishing gradient problem** is encountered when training artificial neural networks with gradient based learning method and back propagation. In such methods, each of the neural network's weights receives an update proportional to the partial derivative of the error function with respect to the current weight in each iteration of training. The problem is that in some cases, the gradient will be vanishingly small, effectively preventing the weight from changing its value. In the worst case, this may completely stop the neural network from further training.

**The Problem of Long-Term Dependencies:**

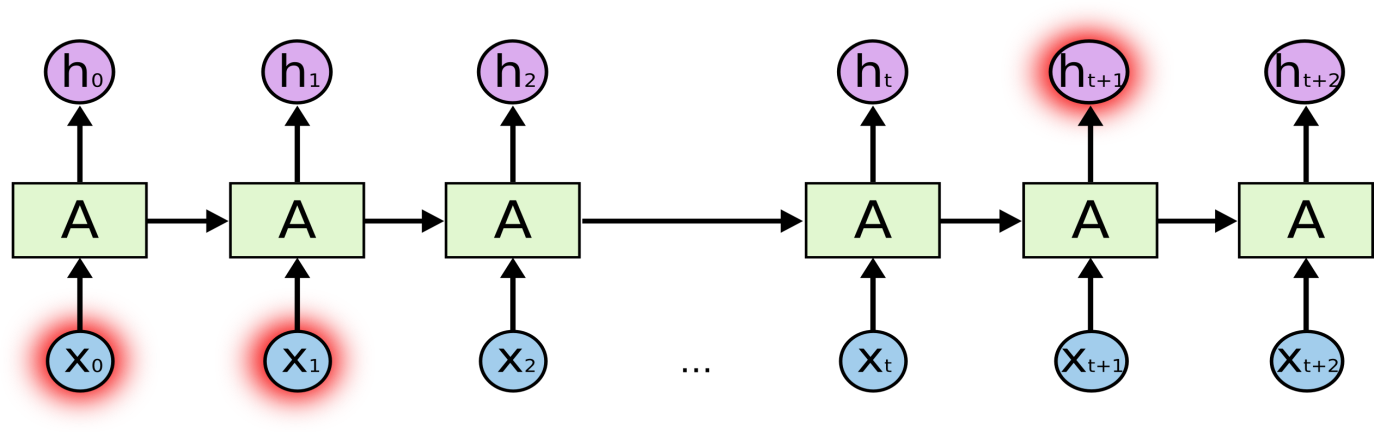
One of the appeals of RNNs is the idea that they might be able to connect previous information to the present task, such as using previous video frames might inform the understanding of the present frame. If RNNs could do this, they’d be extremely useful. But can they? It depends.

Sometimes, we only need to look at recent information to perform the present task. For example, consider a language model trying to predict the next word based on the previous ones. If we are trying to predict the last word in “the clouds are in the *sky*,” we don’t need any further context – it’s pretty obvious the next word is going to be sky. In such cases, where the gap between the relevant information and the place that it’s needed is small, RNNs can learn to use the past information.



But there are also cases where we need more contexts. Consider trying to predict the last word in the text “I grew up in France… I speak fluent *French*.” Recent information suggests that the next word is probably the name of a language, but if we want to narrow down which language, we need the context of France, from further back. It’s entirely possible for the gap between the relevant information and the point where it is needed to become very large.

Unfortunately, as that gap grows, RNNs become unable to learn to connect the information.



In theory, RNNs are absolutely capable of handling such “long-term dependencies.” A human could carefully pick parameters for them to solve toy problems of this form. Sadly, in practice, RNNs don’t seem to be able to learn them. The problem was explored in depth by [Hochreiter (1991) [German]](http://people.idsia.ch/~juergen/SeppHochreiter1991ThesisAdvisorSchmidhuber.pdf) and [Bengio, et al. (1994)](http://www-dsi.ing.unifi.it/~paolo/ps/tnn-94-gradient.pdf), who found some pretty fundamental reasons why it might be difficult.

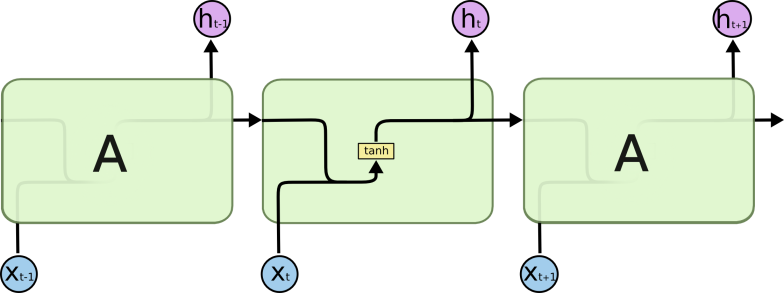
Thankfully, LSTMs don’t have this problem!

* 1. **LSTM Architecture**

Long Short Term Memory networks – usually just called “**LSTMs**” – are a special kind of RNN, capable of learning long-term dependencies. They were introduced by [Hochreiter & Schmidhuber (1997)](http://www.bioinf.jku.at/publications/older/2604.pdf), and were refined and popularized by many people in following work.[1](https://colah.github.io/posts/2015-08-Understanding-LSTMs/#fn1) They work tremendously well on a large variety of problems, and are now widely used.

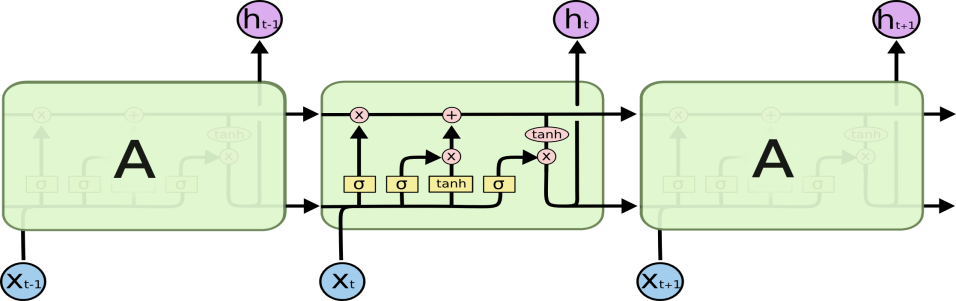
LSTMs are explicitly designed to avoid the long-term dependency problem. Remembering information for long periods of time is practically their default behaviour, not something they struggle to learn!

All recurrent neural networks have the form of a chain of repeating modules of neural network. In standard RNNs, this repeating module will have a very simple structure, such as a single **tanh** layer.



**The repeating module in a standard RNN contains a single layer**

LSTMs also have this chain like structure, but the repeating module has a different structure. Instead of having a single neural network layer, there are four, interacting in a very special way.

* 1. 

**The repeating module in an LSTM contains four interacting layers**

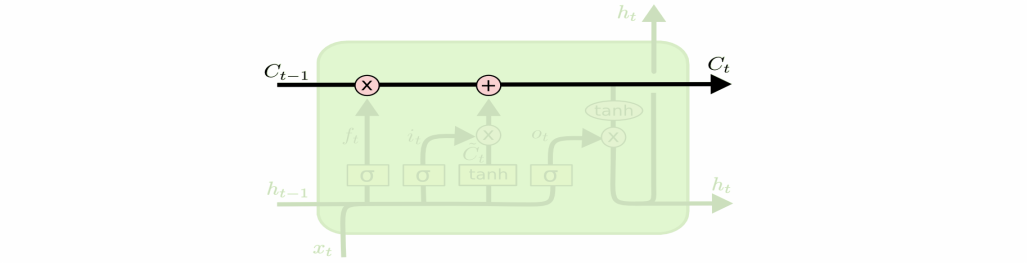
Don’t worry about the details of what’s going on. We’ll walk through the LSTM diagram step by step later. For now, let’s just try to get comfortable with the notation we’ll be using.



In the above diagram, each line carries an entire vector, from the output of one node to the inputs of others. The pink circles represent point wise operations, like vector addition, while the yellow boxes are learned neural network layers. Lines merging denote concatenation, while a line forking denote its content being copied and the copies going to different locations.

**The Core Idea behind LSTMs**: The key to LSTMs is the cell state, the horizontal line running through the top of the diagram.

The cell state is kind of like a conveyor belt. It runs straight down the entire chain, with only some minor linear interactions. It’s very easy for information to just flow along it unchanged.



The LSTM does have the ability to remove or add information to the cell state, carefully regulated by structures called gates.

Gates are a way to optionally let information through. They are composed out of a sigmoid neural net layer and a point wise multiplication operation.



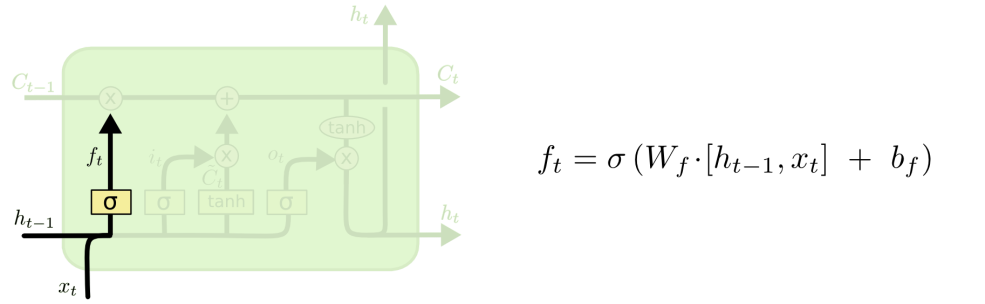
The sigmoid layer outputs numbers between zero and one, describing how much of each component should be let through. A value of zero means “let nothing through,” while a value of one means “let everything through!”

An LSTM has three of these gates, to protect and control the cell state.

**Step-by-Step LSTM Walk Through**

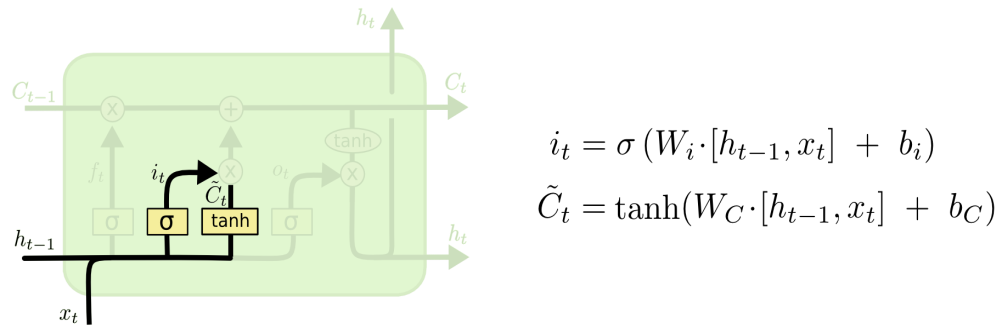
The first step in our LSTM is to decide what information we’re going to throw away from the cell state. This decision is made by a sigmoid layer called the “forget gate layer.” It looks at ht−1 and xt, and outputs a number between 0 and 1 for each number in the cell state Ct−1. A 1 represents “completely keep this” while a 0 represents “completely get rid of this.”

Let’s go back to our example of a language model trying to predict the next word based on all the previous ones. In such a problem, the cell state might include the gender of the present subject, so that the correct pronouns can be used. When we see a new subject, we want to forget the gender of the old subject.



The next step is to decide what new information we’re going to store in the cell state. This has two parts. First, a sigmoid layer called the “input gate layer” decides which values we’ll update. Next, a tanh layer creates a vector of new candidate values, , that could be added to the state. In the next step, we’ll combine these two to create an update to the state.

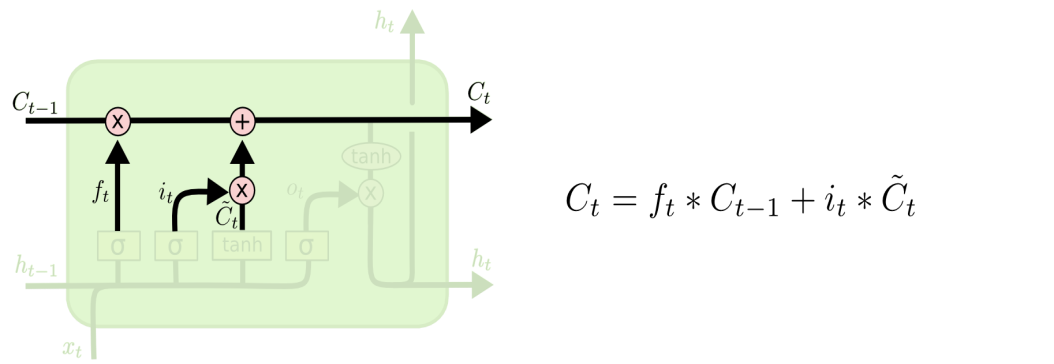
In the example of our language model, we’d want to add the gender of the new subject to the cell state, to replace the old one we’re forgetting.



It’s now time to update the old cell state, Ct−1, into the new cell state Ct .The previous steps already decided what to do, and we just need to actually do it.

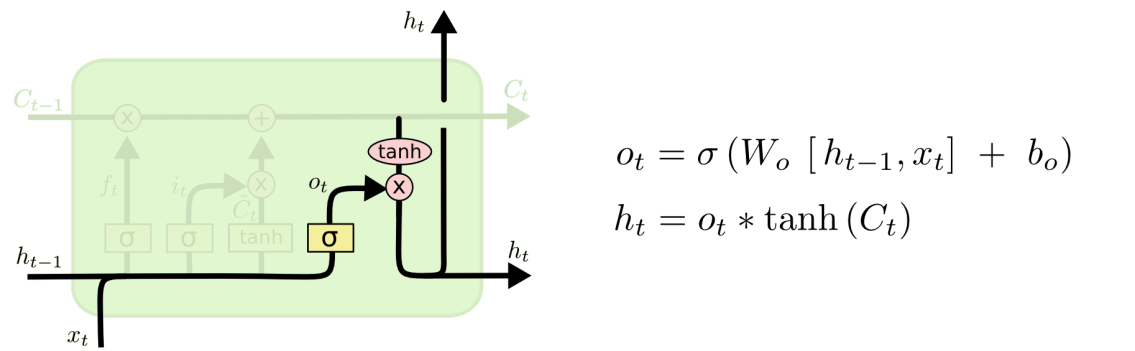
We multiply the old state by ft , forgetting the things we decided to forget earlier. Then we add . This is the new candidate values, scaled by how much we decided to update each state value.

In the case of the language model, this is where we’d actually drop the information about the old subject’s gender and add the new information, as we decided in the previous steps.



Finally, we need to decide what we’re going to output. This output will be based on our cell state, but will be a filtered version. First, we run a sigmoid layer which decides what parts of the cell state we’re going to output. Then, we put the cell state through tanh (to push the values to be between −1and 1) and multiply it by the output of the sigmoid gate, so that we only output the parts we decided to.

For the language model example, since it just saw a subject, it might want to output information relevant to a verb, in case that’s what is coming next. For example, it might output whether the subject is singular or plural, so that we know what form a verb should be conjugated into if that’s what follows next.



**2. Problem Statement**

**What do you see in the given picture?**

Well some of you might say “**A white dog in a grassy area**”, some may say “**White dog with brown spots**” and yet some others might say “**A dog on grass and some pink flowers**”. Definitely all of these captions are relevant for this image and there may be some others also. But the point I want to make is; it’s so easy for us, as human beings, to just have a glance at a picture and describe it in an appropriate language. Even a 5 year old could do this with utmost ease. But, can you write a computer program that takes an image as input and produces a relevant caption as output?

This is the most discussed research problem now days.

**3. Motivation**

We must first understand how important this problem is to real world scenarios. Let’s see few applications where a solution to this problem can be very useful.

* Self-driving cars: Automatic driving is one of the biggest challenges and if we can properly caption the scene around the car, it can give a boost to the self-driving system.
* Aid to the blind: We can create a product for the blind which will guide them travelling on the roads without the support of anyone else. We can do this by first converting the scene into text and then the text to voice. Both are now famous applications of Deep Learning. Refer this [**link**](https://www.youtube.com/watch?v=rLyF4XQLwr0)where it’s shown how NVidia research is trying to create such a product.
* CCTV cameras are everywhere today, but along with viewing the world, if we can also generate relevant captions, then we can raise alarms as soon as there is some malicious activity going on somewhere. This could probably help reduce some crime and/or accidents.
* Automatic Captioning can help; make Google Image Search as good as Google Search, as then every image could be first converted into a caption and then search can be performed based on the caption.

**4. Proposed Project**

I have used deep learning to tackle this problem. Since, it is a combination of two typical problems; Language modeling and Image classification hence this problem can be automated using two neural network architectures; CNN and LSTM. CNN is used for Image classification, and LSTM is used for language modeling. Name of the project is Image captioning Generator or **ICG** in short.

# 5. Prerequisites and system requirements

**Prerequisites:** Basic Deep Learning concepts like Multi-layered Perceptrons, Convolution Neural Networks, Recurrent Neural Networks, Long-short term memory, Transfer Learning, Gradient Descent, Backpropagation, Overfitting, Probability, Text Processing, Python syntax and data structures, Keras library, etc.

**Requirements:** Neural Networks Training is time consuming task as it requires multiple CPU and GPU support hence it is always a good choice to use Google COLAB for Model training and testing.

**6. Introduction to ICG architecture**

Since the input consists of two parts, an image vector and a partial caption, we cannot use the Sequential API provided by the Keras library. For this reason, we use the Functional API which allows us to create Merge Models.First, let’s look at the brief architecture which contains the high level sub-modules:



**Predicted word, next in the sequence of partial caption**

**Input 1**

**Input 2**

**High level architecture**

**7. Dataset used**

There are many open source datasets available for this problem, like Flickr 8k (containing8k images), Flickr 30k (containing 30k images), MS COCO (containing 180k images), etc. But for the purpose of this case study, I have used the Flickr 8k dataset which you can download from this link: https://drive.google.com/drive/folders/1j-LPG-gnziaKQGajOo9L-99rKC3do9zo?usp=sharing

Also training a model with large number of images may not be feasible on a system which is not a very high end PC/Laptop.This dataset contains 8000 images each with 5 captions (as we have already seen in the Introduction section that an image can have multiple captions, all being relevant simultaneously).

These images are bifurcated as follows:

* Training Set — 6000 images
* Test Set — 2000 images

# 8. Implementation of the project step by step

# We will implement the complete project step by step. Please follow the below steps in order to achieve desired result.

* **Frame work:** Tensor flow, Keras
* **Platform:** Google Colab
* **Dataset:** Flickr8k
* **Transfer learning:** InceptionV3
* **Word Embeddings:** GloVe.6B.200d

**8.1 Understanding the data**

If you have downloaded the data from the link that I have provided, then, along with images, you will also get some text files related to the images. One of the files is “Flickr8k.token.txt” which contains the name of each image along with its 5 captions. We can read this file as follows:

If the dataset is located in your Google drive and Platform is Google COLAB then ***"/content/drive/My Drive/Flickr8k/Flickr8k.token.txt"*** reads file from the drive folder

token\_path = "/content/drive/My Drive/Flickr8k/Flickr8k.token.txt"

train\_images\_path = '/content/drive/My Drive/Flickr8k/Flickr8k.trainImages.txt'

test\_images\_path = '/content/drive/My Drive/Flickr8k/Flickr8k.testImages.txt'

images\_path = '/content/drive/My Drive/Flickr8k/Images/'

glove\_path = '/content/drive/My Drive/Flickr8k/glove.6B.200d.txt'

with open(token\_path,'r') as f:

  desc = f.read()

print(desc[:410])

1000268201\_693b08cb0e.jpg A child in a pink dress is climbing up a set of stairs in an entry way .

1000268201\_693b08cb0e.jpg A girl going into a wooden building .

1000268201\_693b08cb0e.jpg A little girl climbing into a wooden playhouse .

1000268201\_693b08cb0e.jpg A little girl climbing the stairs to her playhouse .

1000268201\_693b08cb0e.jpg A little girl in a pink dress going into a wooden cabin .

# 8.2 Creating a dictionary of captions

Now we will create a dictionary that contains all the captions with keys as image name.

#it's a dictionry having image id's as keys and list of all captions of that key as values

captions\_dict = dict()

for l in desc.split('\n'):

        tokens = l.split()

        if len(l) > 2:

       image\_name = tokens[0].split('.')[0]

          image\_cap = ' '.join(tokens[1:])

          if image\_name not in captions\_dict:

              captions\_dict[image\_name] = list()

          captions\_dict[image\_name].append(image\_cap)

# 8.3 Cleaning the data

When we deal with text, we generally perform some basic cleaning like lower-casing all the words (otherwise “hello” and “Hello” will be regarded as two separate words), removing special tokens (like ‘%’, ‘$’, ‘#’, etc.), eliminating words which contain numbers (like ‘hey199’, etc.).

#it creates the mapping of to be replaced characters with respective desired characters

#here fisrt parameter contains the string of characters that has to be replaced

#second parameter contains the string of respective chars that are the replacements of first #parameter's chars

#third parameter is to map characters to None (completely remove that from the string)

map\_table = str.maketrans('', '', string.punctuation)

for image\_name, cap\_list in captions\_dict.items():

    for i in range(len(cap\_list)):

        cap = cap\_list[i]

        cap = cap.split()

        cap = [word.lower() for word in cap]

        #translate does rest of the work of map table by finish mapping

        cap = [w.translate(map\_table) for w in cap]

        cap\_list[i] =  ' '.join(cap)

    captions\_dict[image\_name] = cap\_list

print(captions\_dict["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']

# 8.4 Create a vocabulary

Create a vocabulary of all the unique words present across all the 8000\*5 (i.e. 40000) image captions (**corpus**) in the data set:

vocabulary = set()

for image\_name in captions\_dict.keys():

        [vocabulary.update(c.split()) for c in captions\_dict[image\_name]]

print('Original Vocabulary Size: %d' % len(vocabulary))

Original Vocabulary Size: 8828

This means we have 8828 unique words across all the 40000 image captions. We write all these captions along with their image names in a new file namely, “new\_captions.txt” and save it on the disk. However, if we think about it, many of these words will occur very few times, say 1, 2 or 3 times. Since we are creating a predictive model, we would not like to have all the words present in our vocabulary but the words which are more likely to occur or which are common. This helps the model become more **robust to outliers** and make fewer mistakes. Hence we consider only those words which **occur at least 10 times** in the entire corpus. The code for this is below:

all\_train\_captions = []

count = 0

for name,caps in train\_captions.items():

    for cap in caps:

      if cap not in all\_train\_captions:

        all\_train\_captions.append(cap)

print(all\_train\_captions[:5])

print(len(all\_train\_captions))

threshold = 10

word\_counts = dict()

for cap in all\_train\_captions:

    for word in cap.split(' '):

        word\_counts[word] = word\_counts.get(word, 0) + 1

#it creates the dictionary of word in which each word occurs atleast 10 times

vocab = [word for word in word\_counts if word\_counts[word] >= threshold]

print('Vocabulary = %d' % (len(vocab)))

Vocabulary = 1675: So now we have only 1675 unique words in our vocabulary. However, we will append 0’s (zero padding explained later) and thus total words = 1675+1 = **1676**(one index for the 0) so that masking can be performed.

# 8.5 Loading the training and testing set

The text file “Flickr\_8k.trainImages.txt” contains the names of the images that belong to the training set, and “Flickr\_8k.testImages.txt” contains the names of the images that belong to the testing set. So we load these names into a list “train” and then create two lists of paths to test images and train images.

with open(train\_images\_path,'r') as f:

  train\_images = f.read().strip().split("\n")

with open(test\_images\_path,'r') as f:

  test\_images = f.read().strip().split("\n")

dataset = list()

for img in train\_images:

    if len(img) > 1:

      image\_name = img.split('.')[0]

      dataset.append(image\_name)

#set of all distinct training images names

train\_data = set(dataset)

#glob gets complete paths or let's say global of the files of particular directory

img = glob.glob(images\_path + '\*.jpg')

print(len(img))

#list of all training images that contains complete path of the images unlike train\_images

train\_img = list()

test\_img = list()

for i in img:

    if i[len(images\_path):] in train\_images:

        train\_img.append(i)

    if i[len(images\_path):] in test\_images:

        test\_img.append(i)

Now, we load the captions of these images from “new\_captions.txt” (saved on the hard disk) in the Python dictionary “**train\_captions**”. However, when we load them, we will add two tokens in every caption as follows (significance explained later.

‘**START**’: This is a start sequence token which will be added at the start of every caption.

‘**END**’: This is an end sequence token which will be added at the end of every caption.

train\_captions = dict()

for line in new\_captions.split('\n'):

    tokens = line.split()

    image\_name, image\_cap = tokens[0], tokens[1:]

    if image\_name in train\_data:

        if image\_name not in train\_captions:

            train\_captions[image\_name] = list()

        cap = 'START ' + ' '.join(image\_cap) + ' END'

        train\_captions[image\_name].append(cap)

print(train\_captions["1000268201\_693b08cb0e"])

'START a child in a pink dress is climbing up a set of stairs in an entry way END', 'START a girl going into a wooden building END', 'START a little girl climbing into a wooden playhouse END', 'START a little girl climbing the stairs to her playhouse END', 'START a little girl in a pink dress going into a wooden cabin END'

# 8.6 Data Pre-processing: 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” (pronounced: word to index) and “ixtoword” (pronounced: index to word).

Stating simply, we will represent every unique word in the vocabulary by an integer (index). As seen above, we have 1676 unique words in the corpus and thus each word will be represented by an integer index ranging 1 to 1676. There is one more parameter that we need to calculate, i.e., the maximum length of a caption.

**These two Python dictionaries can be used as follows:**

* **wordtoix[‘abc’]:-** returns index of the word ‘abc’
* **ixtoword[k]:-** returns the word whose index is ‘k’

The code used is as below:

ixtoword = {}

wordtoix = {}

ix = 1

for w in vocab:

    wordtoix[w] = ix

    ixtoword[ix] = w

    ix += 1

vocab\_size = len(ixtoword) + 1

max\_length = max([len(cap.split()) for cap in all\_train\_captions])

print('capription Length: %d' % max\_length)

**Output:** 38

# 8.7 Data Pre-processing: Images

Images are nothing but input (X) to our model. As you may already know that any input to a model must be given in the form of a vector.

We need to convert every image into a fixed sized vector which can then be fed as input to the neural network. For this purpose, we opt for **transfer learning** by using the InceptionV3 model (Convolutional Neural Network) created by Google Research.

This model was trained on “**Imagenet**” dataset to perform image classification on 1000 different classes of images. However, our purpose here is not to classify the image but just get fixed-length informative vector for each image. This process is called **automatic feature engineering.**

Hence, we just remove the last Softmax layer from the model and extract a 2048 length vector (**bottleneck features**) for every image as follows:

#use transfer learning

model = InceptionV3(weights='imagenet')

model\_new = Model(model.input, model.layers[-2].output)

Now, we pass every image to this model after pre-processing to get the corresponding 2048 length feature vector as follows:

def preprocess(image\_path):

    img = image.load\_img(image\_path, target\_size=(299, 299))

    #flattens the 3 dimensional array of RGB image

    x = image.img\_to\_array(img)

    #expands the dimension along given axis

    x = np.expand\_dims(x, axis=0)

    #preprocess input performs 3 operations:

# noramlization of vector(/255) , change the range of data from (0 to 1) to (-2 to 0)

    x = preprocess\_input(x)

    return x

def encode(image):

    image = preprocess(image)

    fea\_vec = model\_new.predict(image)

    #changes the dimesnion (n,1) or (1,n) to (n,)

    fea\_vec = np.reshape(fea\_vec, fea\_vec.shape[1])

    return fea\_vec

encoding\_train = {}

for img in train\_img:

    encoding\_train[img[len(images\_path):]] = encode(img)

train\_features = encoding\_train

encoding\_test = {}

for img in test\_img:

    encoding\_test[img[len(images\_path):]] = encode(img)

test\_features = encoding\_test

We save all the bottleneck train features in a Python dictionary and save it on the disk using hdf5 file with hierarchical structure, namely “**train\_features.hdf5”** whose keys are “testing\_data” and “training\_data”. The former contains “train\_features” and the latter contains “test\_features” which further has image vectors, each of size 2048.

**NOTE: This process might take an hour or two if you do not have a high end PC/laptop.**

# 8.8 Word Embeddings

As already stated above, we will map the every word (index) to a 200-long vector and for this purpose, we will use a pre-trained GLOVE Model:

Now, for all the 1676 unique words in our vocabulary, we create an embedding matrix which will be loaded into the model before training. To understand more about word Embeddings, please refer this [**link**](https://www.analyticsvidhya.com/blog/2017/06/word-embeddings-count-word2veec/)

embeddings\_index = {}

with open(glove\_path,"r",encoding="utf-8") as f:

  for line in f:

      values = line.split()

      word = values[0]

      coefs = np.asarray(values[1:], dtype='float32')

      embeddings\_index[word] = coefs

embedding\_dim = 200

embedding\_matrix = np.zeros((vocab\_size, embedding\_dim))

for word, i in wordtoix.items():

    embedding\_vector = embeddings\_index.get(word)

    if embedding\_vector is not None:

        embedding\_matrix[i] = embedding\_vector

print(embedding\_matrix.shape)

**Output:** (1676, 200)

# ****8.9**** 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:



**(Train image 1)** **(Train image 2)**

**Caption:** The black cat sat on grass **Caption:** The white cat is walking on road

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

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? etc.

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 “START” and “END” in both of them: (Assume we have already performed the basic cleaning steps)

Caption\_1: “START the black cat sat on grass END”

Caption\_2 -> “START the white cat is walking on road END”

Vocabulary = {black, cat, END, grass, is, on, road, sat, START, the, walking, white}

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

black -1, cat -2, END -3, grass -4, is -5, on -6, road -7, sat -8, START -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 “**STARTS the black cat sat on grass END**”. 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 + ‘START’; 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 + ‘START the’; Output = ‘cat’

And so on…

Thus, we can summarize the data matrix for one image and its corresponding caption as follows: It must be noted that, one image + caption is **not a single data point** but are multiple data points depending on the length of the caption. Similarly if we consider both the images and their captions, our data matrix will then look as follows:

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 (more on this later).

However, we have already discussed that we are not going to pass the actual English text of the caption; rather we are going to pass the sequence of indices where each index represents a unique word.

Since we have already created an index for each word, let’s now replace the words with their indices and understand how the data matrix will look like:

Data matrix after replacing the words by their indices

Since we would be doing **batch processing**(explained later), we need to make sure that each sequence is of **equal length**. Hence we need to **append 0’s** (zero padding) at the end of each sequence. But **how many** zeros should we append in each sequence?

Well, this is the reason we had calculated the maximum length of a caption, which is 34 (if you remember). So we will append those many number of zeros which will lead to every sequence having a length of 34.

The data matrix will then look as follows: **Need for a Data Generator:**

I hope this gives you a good sense as to how we can prepare the dataset for this problem. However, there is a big catch in this.

In the above example, I have only considered 2 images and captions which have led to 15 data points.

However, in our actual training dataset we have 6000 images, each having 5 captions. This makes a total of **30000** images and captions.

Even if we assume that each caption on an average is just 7 words long, it will lead to a total of 30000\*7 i.e. **210000**data points.

**Compute the size of the data matrix:**

Size of the data matrix = n\*m

Where n: number of data points (assumed as 210000)

And m: length of each data point

Clearly m= Length of image vector (2048) + Length of partial caption(x).

m = 2048 + x

But what is the value of x?

Well you might think it is 38, but no wait, it’s wrong.

Every word (or index) will be mapped (embedded) to higher dimensional space through one of the word embedding techniques.

Later, during the model building stage, we will see that each word/index is mapped to a 200-long vector using a pre-trained GLOVE word embedding model.

Now each sequence contains 38 indices, where each index is a vector of length 200. Therefore x = 38\*200 = 7600

Hence, m = 2048 + 7600 = 9048

Finally, size of data matrix= 210000 \* 9048 = 1900080000 blocks.

Now even if we assume that one block takes 2 byte, then, to store this data matrix, we will require more than 3 GB of main memory.

This is pretty huge requirement and even if we are able to manage to load this much data into the RAM, it will make the system very slow.

For this reason we use data generators a lot in Deep Learning. Data Generators are a functionality which is natively implemented in Python. The Image Data Generator class provided by the Keras API is nothing but an implementation of generator function in Python.

**So how does using a generator function solve this problem?**

If you know the basics of Deep Learning, then you must know that to train a model on a particular dataset, we use some version of Stochastic Gradient Descent (SGD) like Adam, **Rmsprop, Adagrad**, etc.

With **SGD**, we do not calculate the loss on the entire data set to update the gradients. Rather in every iteration, we calculate the loss on a **batch** of data points (typically 64, 128, 256, etc.) to update the gradients.

This means that we do not require storing the entire dataset in the memory at once. Even if we have the current batch of points in the memory, it is sufficient for our purpose.

A generator function in Python is used exactly for this purpose. It’s like an iterator which resumes the functionality from the point it left the last time it was called.

To understand more about Generators, please read [**here**](https://wiki.python.org/moin/Generators). The code for data generator is as follows:

def data\_generator(captions, photos, wordtoix, max\_length, batch\_size):

    X1, X2, y = list(), list(), list()

    n=0

    # loop for ever over images until yeild doesn't return data

    while 1:

        for key, cap\_list in captions.items():

            n+=1

            # retrieve the photo feature

            photo = photos[key+'.jpg']

            for cap in cap\_list:

                # encode the sequence

                seq = [wordtoix[word] for word in cap.split(' ') if word in wordtoix]

                # split one sequence into multiple X, y pairs

                for i in range(1, len(seq)):

                    # split into input and output pair

                    in\_seq, out\_seq = seq[:i], seq[i]

                    # pad input sequences with leading zeros

                    in\_seq = pad\_sequences([in\_seq], maxlen=max\_length)[0]

                    # encode output sequence to

                    out\_seq = to\_categorical([out\_seq], num\_classes=vocab\_size)[0]

                    # store

                    X1.append(photo)

                    X2.append(in\_seq)

                    y.append(out\_seq)

            if n==batch\_size:

                yield ([array(X1), array(X2)], array(y))

                X1, X2, y = list(), list(), list()

                n=0

# ****8.10 Defining Model****

#creates input layer with no. of neurons = 2048 in it, for the cnn model

inp\_1 = Input(shape=(2048,))

#regularize the neurons activation by reducing it to 50%

reg\_1 = Dropout(0.5)(inp\_1)

#creates dense layer with no. of neurons = 256 in it, this layer is connected to inp layer defined above

out\_1 = Dense(256, activation='relu')(reg\_1)

#creates input layer with size max\_length of caption, for lstm

inp\_2 = Input(shape=(max\_length,))

#200 dimensional vector to each input word (vocab). This statement only creates a #mapping table that maps word to

#a vector in n dimension, in our case 200, and then whenever a input word comes, it replaces with the corresponding vector

#note here,embedding layer is always connected to the first layer or the input layer #of the network

#mask\_zero identifies the special zero padding and ignores it to continue variable

# size computing

hidden\_1 = Embedding(vocab\_size, embedding\_dim, mask\_zero=True)(inp\_2)

reg\_2 = Dropout(0.5)(hidden\_1)

#create lstm unit having output size 256 or let's say neuorns. It is connceted to the #input layer

out\_2 = LSTM(256)(reg\_2)

#merging by adding the outpts of lstm and cnn having same dimesion 256

#and return single layer with same dimension

decoder1 = add([out\_1, out\_2])

#dense layer (or let's say fully connected layer to above layer)

decoder2 = Dense(256, activation='relu')(decoder1)

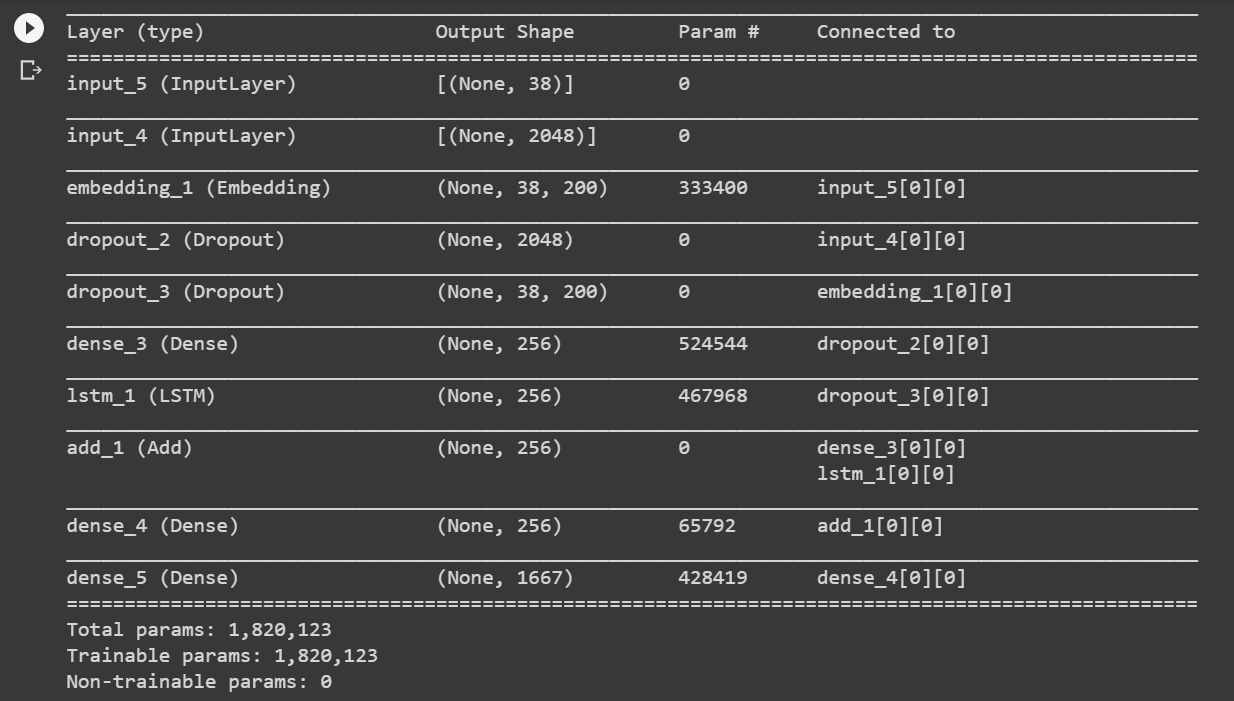
#final probabilistic output

output = Dense(vocab\_size, activation='softmax')(decoder2)

#creating a model finaly

model = Model(inputs=[inp\_1, inp\_2], outputs=output)

model.summary()



The **LSTM (Long Short Term Memory)** layer is nothing but a specialized Recurrent Neural Network to process the sequence input (partial captions in our case). To read more about LSTM, click [**here**](http://colah.github.io/posts/2015-08-Understanding-LSTMs/).

If you have followed the previous section, I think reading these comments should help you to understand the model architecture in a straight forward manner.

Recall that we had created an embedding matrix from a pre-trained Glove model which we need to include in the model before starting the training. Notice that since we are using a pre-trained embedding layer, we need to **freeze**it (trainable = False), before training the model, so that it does not get updated during the back propagation.

Finally we compile the model using the Adam optimizer

#set weights the first hidden layer and set trainable = False as the weights are alrea#dy trained

model.layers[2].set\_weights([embedding\_matrix])

model.layers[2].trainable = False

#adam optimizer is the combination of two most useful algorithms : AdaGrad, RMS#prop

#categorial\_crossentropy: refined version of binary cross entrop(only for 2 classes)

model.compile(loss='categorical\_crossentropy', optimizer='adam')

epochs = 30

batch\_size = 3

steps = len(train\_captions)//batch\_size

#data\_generator returns the list (yielded)

generator = data\_generator(train\_captions, train\_features, wordtoix, max\_length, batch\_size)

#verbose = 0: show progress bar steady along with loss

#verbose = 1: show progress bar dynamic along with loss

#verbose = 2: no progress bar, only loss

print("\n================== MODEL TRAINING BEGAN ==================\n")

model.fit(generator, epochs=epochs, steps\_per\_epoch=steps, verbose=1)

Finally the weights of the model will be updated through back propagation algorithm and the model will learn to output a word, given an image feature vector and a partial caption. So in summary, we have:

**Input\_1:** Partial Caption

**Input\_2:** Image feature vector

**Output:** An appropriate word, next in the sequence of partial caption provided in the input\_1 (or in probability terms we say **conditioned**on image vector and the partial caption)

**Hyper parameters during training:**

The model was then trained for 30 epochs and 3 pictures per batch (batch size.

**Time Taken:** I used **Google COLAB Notebook** (weak network in my case) hence it took me approximately **6 hours to train the model**. However, if you train it with strong internet speed then it would surely take less time, and on a PC without GPU, it could take anywhere from **8 to 16 hours** depending on the configuration of your system.

**8.11 Generating Captions using Greedy Search**

This method chooses the word with the highest probability for the next prediction. This is called as **Maximum Likelihood Estimation (MLE)** i.e. we select that word which is most likely according to the model for the given input. And sometimes this method is also called as **Greedy Search**, as we greedily select the word with maximum probability.

So till now, we have seen how to prepare the data and build the model. In the final step of this series, we will understand how do we test (infer) our model by passing in new images, i.e. how can we generate a caption for a new test image.

def greedySearch(photo):

    caption = 'START'

    for i in range(max\_length):

        sequence = [wordtoix[w] for w in caption.split() if w in wordtoix]

        sequence = pad\_sequences([sequence], maxlen=max\_length)

        yhat = model.predict([photo,sequence], verbose=0)

        #select the one with the geatest probability

        yhat = np.argmax(yhat)

        word = ixtoword[yhat]

        caption += ' ' + word

        if word == 'END':

            break

    #remove star and end tags

    caption.replace("START","")

    caption.replace("END","")

    return caption

pic = list(encoding\_test.keys())[0]

image = encoding\_test[pic].reshape((1,2048))

x=plt.imread(images\_path+pic)

plt.imshow(x)

plt.show()

print("Greedy Search:",greedySearch(image))



**a little girl climbing into a wooden playhouse**

**Important Points:**

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.

**9. Scope of Improvements**

Of course this is just a first-cut solution and a lot of modifications can be made to improve this solution like:

* Using a **larger**dataset.
* Changing the model architecture, e.g. include an **attention**module.
* Doing more **hyper parameter tuning** (learning rate, batch size, number of layers, number of units, dropout rate, batch normalization etc.).
* Use the cross validation set to understand **over fitting**.
* Using **Beam Search** instead of Greedy Search during Inference.
* Using **BLEU Score** to evaluate and measure the performance of the model.
* Writing the code in a proper object oriented way so that it becomes easier for others to replicate.

**================= Thanking You ================**