

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

Harshal Lamba

Image Captioning with Keras

Teaching Computers to describe pictures



[Harshall Lamba](#)

Follow

Nov 3, 2018 · 21 min read

Table of Contents:

1. Introduction
2. Motivation
3. Prerequisites
4. Data collection
5. Understanding the data
6. Data Cleaning
7. Loading the training set
8. Data Preprocessing — Images
9. Data Preprocessing — Captions
10. Data Preparation using Generator Function
11. Word Embeddings
12. Model Architecture

13. Inference
14. Evaluation
15. Conclusion and Future work
16. References

1. Introduction

What do you see in the below picture?



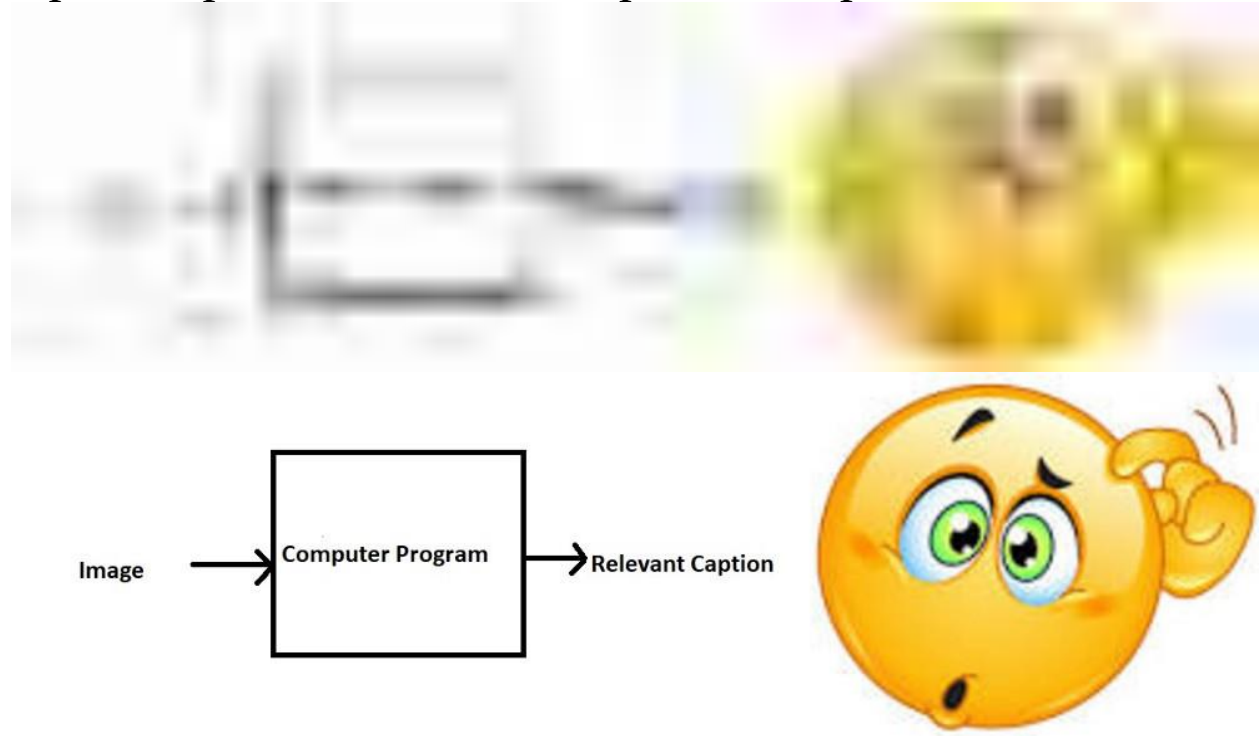
Can you write a caption?

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?



The Problem

Just prior to the recent development of Deep Neural Networks this problem was inconceivable even by the most advanced researchers in Computer Vision. But with the advent of Deep Learning this problem can be solved very easily if we have the required dataset.

This problem was well researched by [Andrej Karapathy](#) in his PhD thesis at Stanford [1], who is also now the **Director of AI at Tesla**.

The purpose of this blog post is to explain (in as simple words as possible) that how Deep Learning can be used to solve this

problem of generating a caption for a given image, hence the name **Image Captioning**.

To get a better feel of this problem, I strongly recommend to use this state-of-the-art system created by Microsoft called as **Caption Bot**. Just go to this link and try uploading any picture you want; this system will generate a caption for it.

2. 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](#) where its 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.

3. Prerequisites

This post assumes familiarity with basic Deep Learning concepts like Multi-layered Perceptrons, Convolution Neural Networks, Recurrent Neural Networks, Transfer Learning, Gradient Descent, Backpropagation, Overfitting, Probability, Text Processing, Python syntax and data structures, Keras library, etc.

4. Data Collection

There are many open source datasets available for this problem, like Flickr 8k (containing 8k 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 by filling this [form](#) provided by the University of Illinois at Urbana-Champaign. 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
- Dev Set — 1000 images

- Test Set — 1000 images

5. 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:

```
# Below is the path for the file "Flickr8k.token.txt" on your disk
filename = "/dataset/TextFiles/Flickr8k.token.txt"
file = open(filename, 'r')
doc = file.read()
```

The text file looks as follows:

Sample Text File

Thus every line contains the <image name>#i <caption>, where $0 \leq i \leq 4$

i.e. the name of the image, caption number (0 to 4) and the actual caption.

Now, we create a dictionary named “descriptions” which contains the name of the image (without the .jpg extension) as keys and a list of the 5 captions for the corresponding image as values.

For example with reference to the above screenshot the dictionary will look as follows:

```
descriptions['101654506_8eb26cfb60'] = ['A brown and white dog is running through the snow .', 'A dog is running in the snow', 'A dog running through snow .', 'a white and brown dog is running through a snow covered field .', 'The white and brown dog is running over the surface of the snow .']
```

6. Data Cleaning

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.).

The below code does these basic cleaning steps:

Code to perform Data Cleaning

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 key in descriptions.keys():
    [vocabulary.update(d.split()) for d in descriptions[key]]
print('Original Vocabulary Size: %d' % len(vocabulary))
Original Vocabulary Size: 8763
```

This means we have 8763 unique words across all the 40000 image captions. We write all these captions along with their image names in a new file namely, “*descriptions.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 less mistakes.

Hence we consider only those words which **occur at least 10 times** in the entire corpus. The code for this is below:

Code to retain only those words which occur at least 10 times in the corpus

So now we have only 1651 unique words in our vocabulary. However, we will append 0's (zero padding explained later) and thus total words = $1651+1 = \mathbf{1652}$ (one index for the 0).

7. Loading the training set

The text file “Flickr_8k.trainImages.txt” contains the names of the images that belong to the training set. So we load these names into a list “train”.

```
filename = 'dataset/TextFiles/Flickr_8k.trainImages.txt'
doc = load_doc(filename)
train = list()
for line in doc.split('\n'):
    identifier = line.split('.')[0]
    train.append(identifier)
print('Dataset: %d' % len(train))
Dataset: 6000
```

Thus we have separated the 6000 training images in the list named “train”.

Now, we load the descriptions of these images from “descriptions.txt” (saved on the hard disk) in the Python dictionary “train_descriptions”.

However, when we load them, we will add two tokens in every caption as follows (significance explained later):

‘**startseq**’ -> This is a start sequence token which will be added at the start of every caption.

‘**endseq**’ -> This is an end sequence token which will be added at the end of every caption.

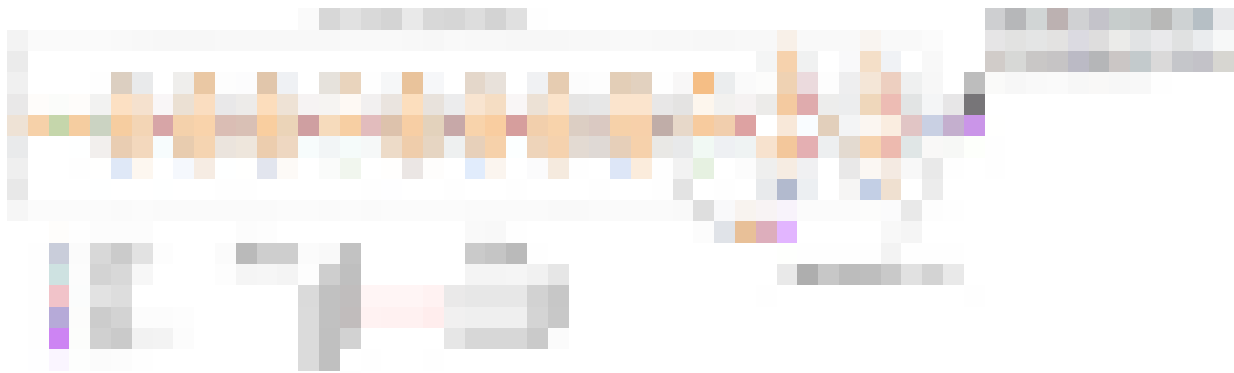
8. Data Preprocessing — Images

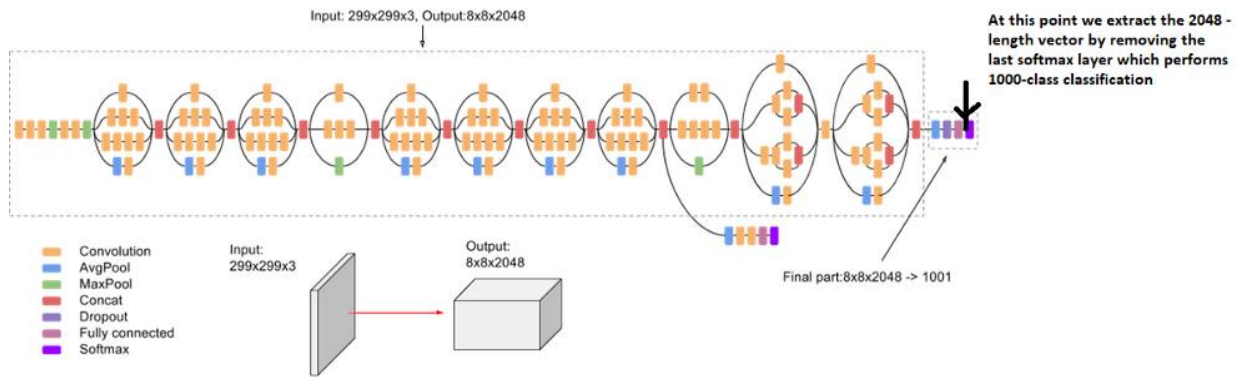
Images are nothing but input (X) to our model. As you may already know that any input to a model must be given in the form of a vector.

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:





Feature Vector Extraction (Feature Engineering) from InceptionV3

The code for this is as follows:

```
# Get the InceptionV3 model trained on imagenet data
model = InceptionV3(weights='imagenet')
# Remove the last layer (output softmax layer) from the inception
v3
model_new = Model(model.input, model.layers[-2].output)
```

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

```
# Convert all the images to size 299x299 as expected by the
# inception v3 model
img = image.load_img(image_path, target_size=(299, 299))
# Convert PIL image to numpy array of 3-dimensions
x = image.img_to_array(img)
# Add one more dimension
x = np.expand_dims(x, axis=0)
# preprocess images using preprocess_input() from inception module
x = preprocess_input(x)
# reshape from (1, 2048) to (2048, )
x = np.reshape(x, x.shape[1])
```

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.

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

Similarly we encode all the test images and save them in the file “**encoded_test_images.pkl**”.

9. 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” (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 1652 unique words in the corpus and thus each word will be represented by an integer index between 1 to 1652.

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

```

There is one more parameter that we need to calculate, i.e., the maximum length of a caption and we do it as below:

```

# convert a dictionary of clean descriptions to a list of
descriptions
def to_lines(descriptions):
    all_desc = list()
    for key in descriptions.keys():
        [all_desc.append(d) for d in descriptions[key]]
    return all_desc# calculate the length of the description with the
most words
def max_length(descriptions):
    lines = to_lines(descriptions)
    return max(len(d.split()) for d in lines)# determine the maximum
sequence length
max_length = max_length(train_descriptions)
print('Max Description Length: %d' % max_length)
Max Description Length: 34

```

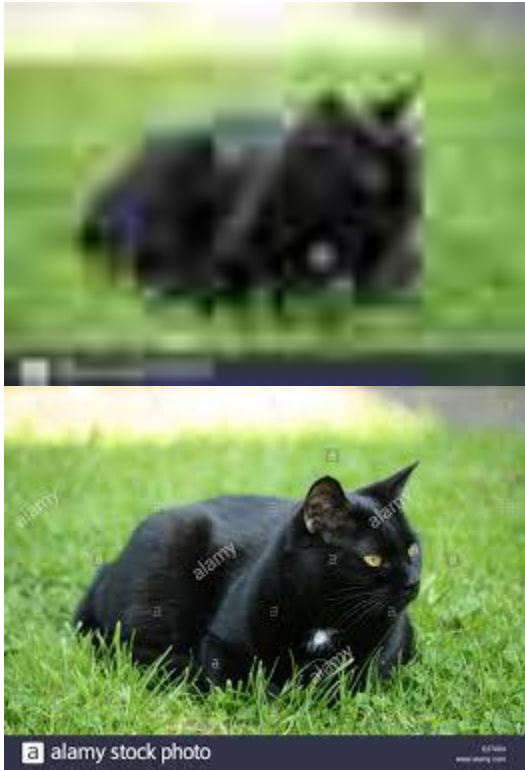
So the maximum length of any caption is 34.

10. Data Preparation using Generator Function

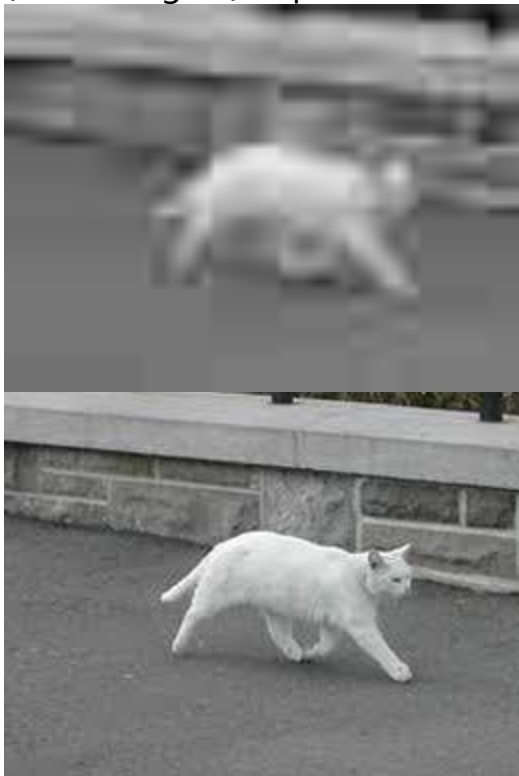
This is one of the most important steps in this case study. Here we will understand how to prepare the data in a manner which will be convenient to be given as input to the deep learning model.

Hereafter, I will try to explain the remaining steps by taking a sample example as follows:

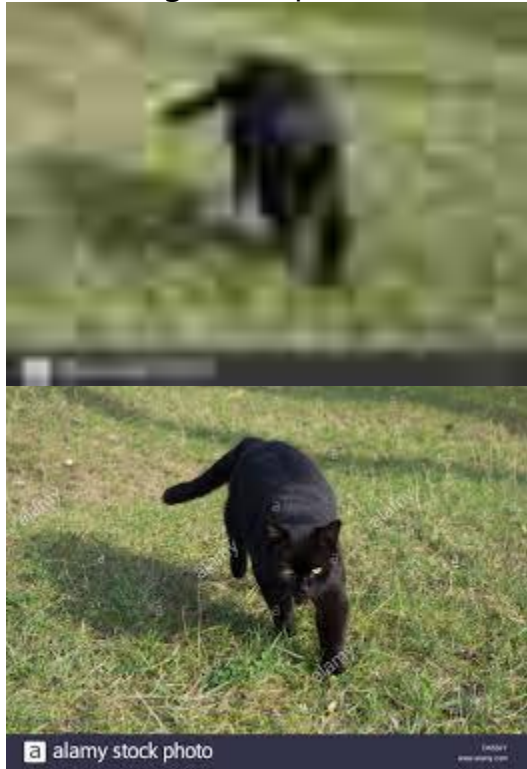
Consider we have 3 images and their 3 corresponding captions as follows:



(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?, 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 “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 $D = \{X_i, Y_i\}$, where X_i is the feature vector of data point 'i' and Y_i 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.:

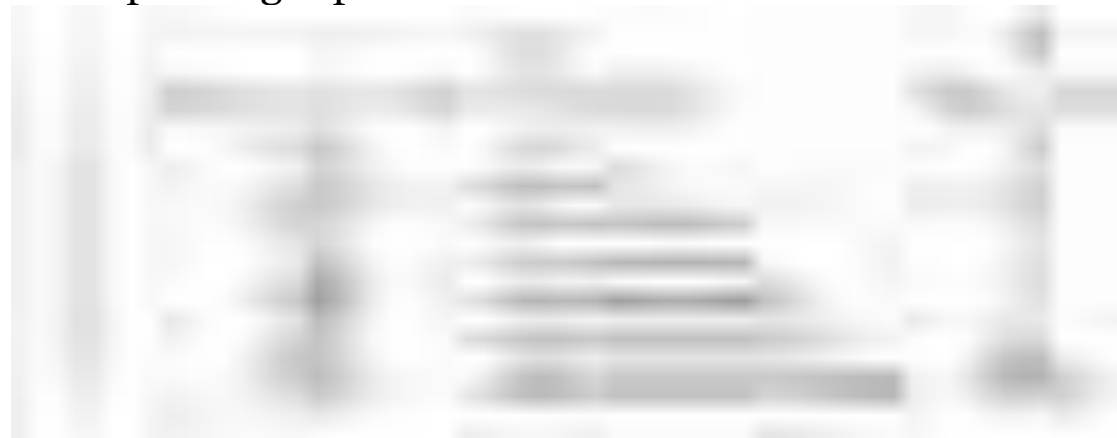
Input = Image_1 + 'startseq'; Output = 'the'

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'

And so on...

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

Data points corresponding to one image and its caption

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:



	Xi		Yi	
i	Image feature vector	Partial Caption	Target word	
1	Image_1	startseq	the	data points corresponding to image 1 and its caption
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	
8	Image_2	startseq	the	data points corresponding to image 2 and its caption
9	Image_2	startseq the	white	
10	Image_2	startseq the white	cat	
11	Image_2	startseq the white cat	is	
12	Image_2	startseq the white cat is	walking	
13	Image_2	startseq the white cat is walking	on	
14	Image_2	startseq the white cat is walking on	road	
15	Image_2	startseq the white cat is walking on road	endseq	

Data Matrix for both the images and captions

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:



		X_i	Y_i
i	Image feature vector	Partial Caption	Target word
1	Image_1	[9]	10
2	Image_1	[9, 10]	1
3	Image_1	[9, 10, 1]	2
4	Image_1	[9, 10, 1, 2]	8
5	Image_1	[9, 10, 1, 2, 8]	6
6	Image_1	[9, 10, 1, 2, 8, 6]	4
7	Image_1	[9, 10, 1, 2, 8, 6, 4]	3
8	Image_2	[9]	10
9	Image_2	[9, 10]	12
10	Image_2	[9, 10, 12]	2
11	Image_2	[9, 10, 12, 2]	5
12	Image_2	[9, 10, 12, 2, 5]	11
13	Image_2	[9, 10, 12, 2, 5, 11]	6
14	Image_2	[9, 10, 12, 2, 5, 11, 6]	7
15	Image_2	[9, 10, 12, 2, 5, 11, 6, 7]	3

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 o'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:



	Xi		Yi
i	Image feature vector	Partial Caption	Target word
1	Image_1	[9, 0, 0 ..., 0]	10
2	Image_1	[9, 10, 0, 0 ..., 0]	1
3	Image_1	[9, 10, 1, 0, 0 ..., 0]	2
4	Image_1	[9, 10, 1, 2, 0, 0 ..., 0]	8
5	Image_1	[9, 10, 1, 2, 8, 0, 0 ..., 0]	6
6	Image_1	[9, 10, 1, 2, 8, 6, 0, 0 ..., 0]	4
7	Image_1	[9, 10, 1, 2, 8, 6, 4, 0, 0 ..., 0]	3
8	Image_2	[9, 0, 0 ..., 0]	10
9	Image_2	[9, 10, 0, 0 ..., 0]	12
10	Image_2	[9, 10, 12, 0, 0 ..., 0]	2
11	Image_2	[9, 10, 12, 2, 0, 0 ..., 0]	5
12	Image_2	[9, 10, 12, 2, 5, 0, 0 ..., 0]	11
13	Image_2	[9, 10, 12, 2, 5, 11, 0, 0 ..., 0]	6
14	Image_2	[9, 10, 12, 2, 5, 11, 6, 0, 0 ..., 0]	7
15	Image_2	[9, 10, 12, 2, 5, 11, 6, 7, 0, 0 ..., 0]	3

Appending zeros to each sequence to make them all of same length 34

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 lead 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:



Data Matrix

Image vector (len 2048)	Partial caption (length 34)

$n \times m$

Data Matrix

Size of the data matrix = $n \times m$

Where $n \rightarrow$ number of data points (assumed as 210000)

And $m \rightarrow$ length of each data point

Clearly $m = \text{Length of image vector}(2048) + \text{Length of partial caption}(x)$.

$$m = 2048 + x$$

But what is the value of x ?

Well you might think it is 34, 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 34 indices, where each index is a vector of length 200. Therefore $x = 34 * 200 = 6800$

Hence, $m = 2048 + 6800 = 8848$.

Finally, size of data matrix = $210000 * 8848 = 1858080000$ 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 ImageDataGenerator 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 to store 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](#).

The code for data generator is as follows:

Code to load data in batches

11. 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:

```
# Load Glove vectors
glove_dir = 'dataset/glove'
embeddings_index = {} # empty dictionary
f = open(os.path.join(glove_dir, 'glove.6B.200d.txt'),
encoding="utf-8")
for line in f:
    values = line.split()
    word = values[0]
    coefs = np.asarray(values[1:], dtype='float32')
    embeddings_index[word] = coefs
f.close()
```


Now, for all the 1652 unique words in our vocabulary, we create an embedding matrix which will be loaded into the model before training.

```
embedding_dim = 200# Get 200-dim dense vector for each of the 10000 words in our vocabulary
embedding_matrix = np.zeros((vocab_size, embedding_dim))for word, i in wordtoix.items():
    #if i < max_words:
    embedding_vector = embeddings_index.get(word)
    if embedding_vector is not None:
        # Words not found in the embedding index will be all zeros
        embedding_matrix[i] = embedding_vector
```

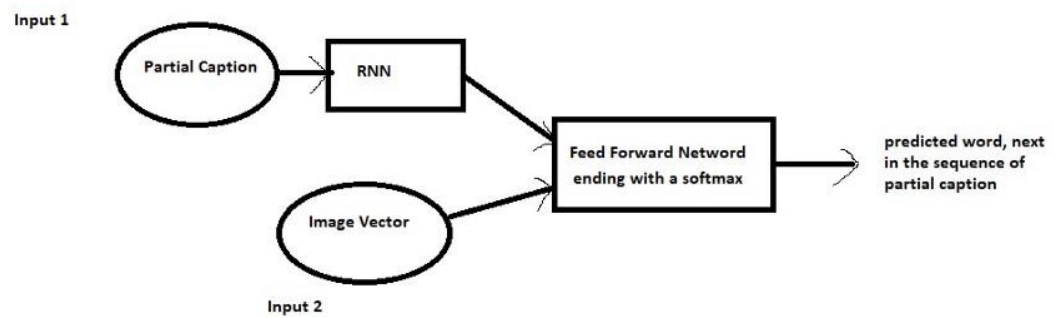
To understand more about word embeddings, please refer [this link](#)

12. Model 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:





High level architecture

We define the model as follows:

Code to define the Model

Let's look at the model summary:

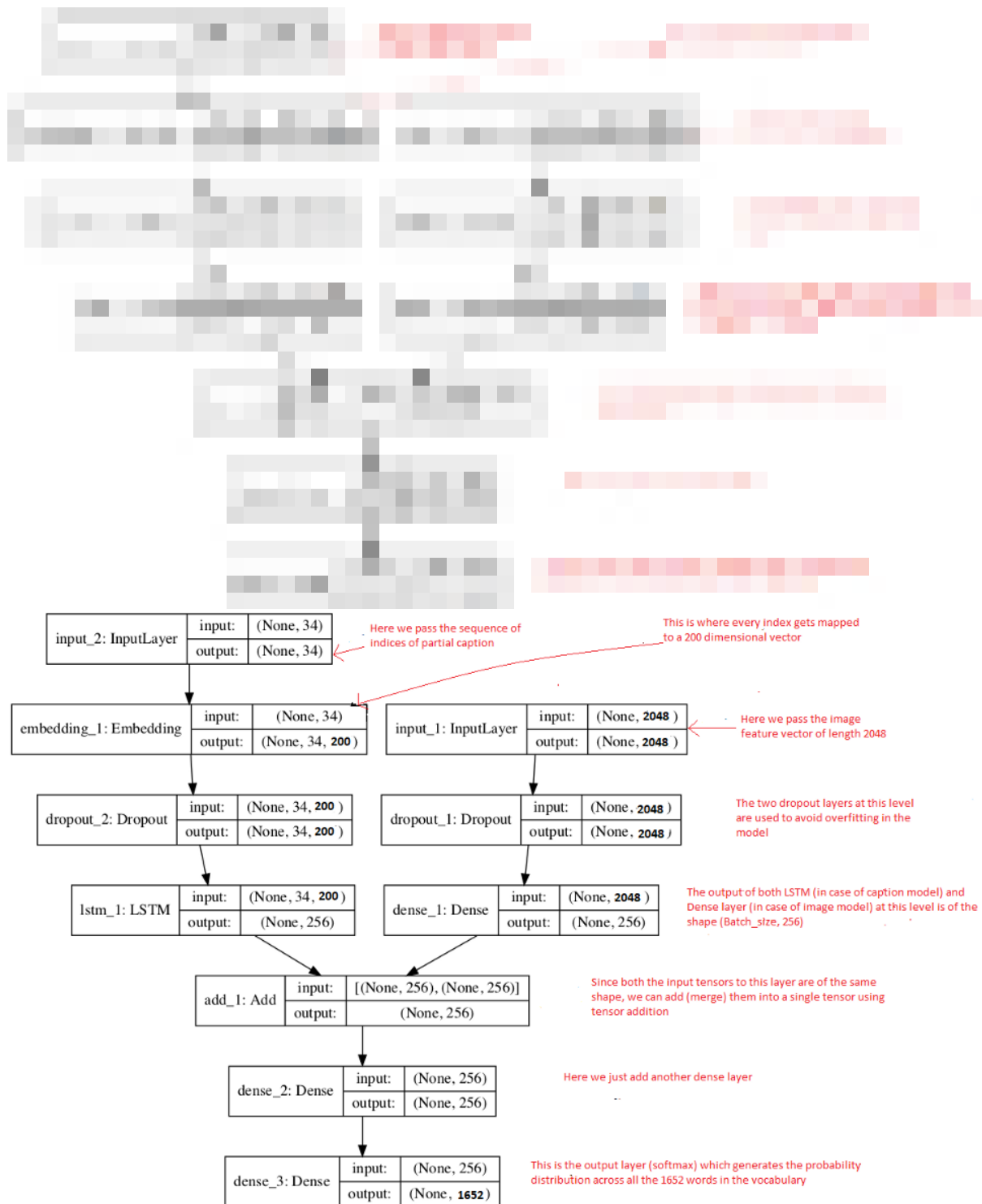


```
model.summary()
```

Layer (type)	Output Shape	Param #	Connected to
input_4 (InputLayer)	(None, 34)	0	
input_3 (InputLayer)	(None, 2048)	0	
embedding_2 (Embedding)	(None, 34, 200)	330400	input_4[0][0]
dropout_3 (Dropout)	(None, 2048)	0	input_3[0][0]
dropout_4 (Dropout)	(None, 34, 200)	0	embedding_2[0][0]
dense_2 (Dense)	(None, 256)	524544	dropout_3[0][0]
lstm_2 (LSTM)	(None, 256)	467968	dropout_4[0][0]
add_2 (Add)	(None, 256)	0	dense_2[0][0] lstm_2[0][0]
dense_3 (Dense)	(None, 256)	65792	add_2[0][0]
dense_4 (Dense)	(None, 1652)	424564	dense_3[0][0]
Total params: 1,813,268			
Trainable params: 1,813,268			
Non-trainable params: 0			

Summary of the parameters in the model

The below plot helps to visualize the structure of the network and better understand the two streams of input:



Flowchart of the architecture

The text in red on the right side are the comments provided for you to map your understanding of the data preparation to model architecture.

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](#).

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:

```
model.layers[2].set_weights([embedding_matrix])  
model.layers[2].trainable = False
```

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 backpropagation.

Finally we compile the model using the adam optimizer

```
model.compile(loss='categorical_crossentropy', optimizer='adam')
```

Finally the weights of the model will be updated through backpropagation 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 with the initial learning rate of 0.001 and 3 pictures per batch (batch size). However after 20 epochs, the learning rate was reduced to 0.0001 and the model was trained on 6 pictures per batch.

This generally makes sense because during the later stages of training, since the model is moving towards convergence, we must lower the learning rate so that we take smaller steps towards the minima. Also increasing the batch size over time helps your gradient updates to be more powerful.

Time Taken: I used the GPU+ Gradient Notebook on www.paperspace.com and hence it took me approximately an hour to train the model. However if you train it on a PC without GPU, it could take anywhere from 8 to 16 hours depending on the configuration of your system.

13. Inference

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.

Recall that in the example where we saw how to prepare the data, we used only first two images and their captions. Now let's use the third image and try to understand how we would like the caption to be generated.

The third image vector and caption were as follows:



Test image

Caption -> the black cat is walking on grass

Also the vocabulary in the example was:

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

We will generate the caption iteratively, one word at a time as follows:

Iteration 1:

Input: Image vector + “startseq” (as partial caption)

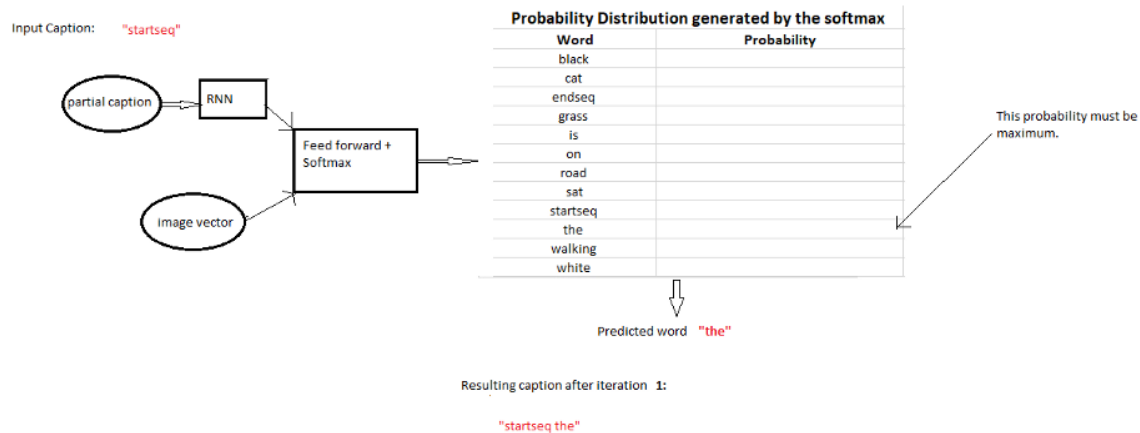
Expected Output word: “the”

(You should now understand the importance of the token ‘startseq’ which is used as the initial partial caption for any image during inference).

But wait, the model generates a 12-long vector(in the sample example while 1652-long vector in the original example) which is a probability distribution across all the words in the vocabulary. For this reason we **greedily** select the word with the maximum probability, given the feature vector and partial caption.

If the model is trained well, we must expect the probability for the word “the” to be maximum:





Iteration 1

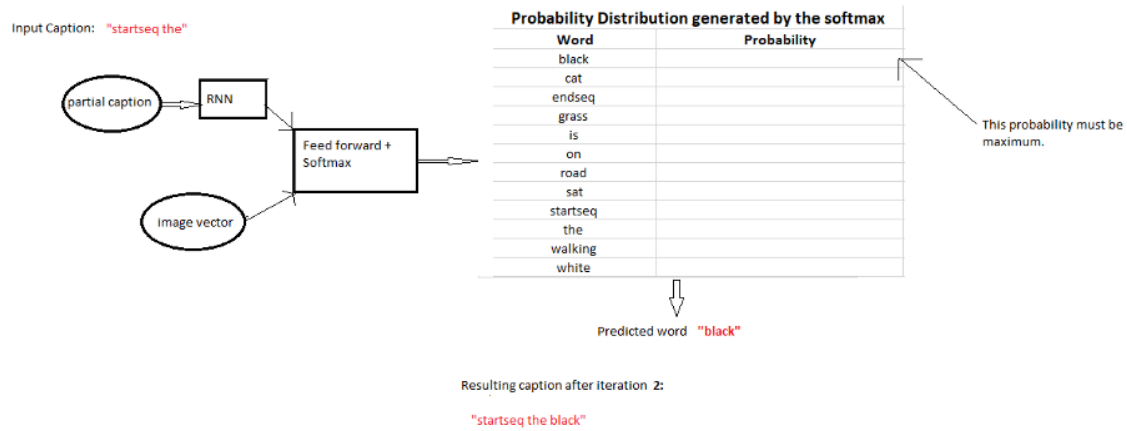
This is called as **Maximum Likelihood Estimation (MLE)** i.e. we select that word which is most likely according to the model for the given input. And sometimes this method is also called as **Greedy Search**, as we greedily select the word with maximum probability.

Iteration 2:

Input: Image vector + "startseq the"

Expected Output word: "black"





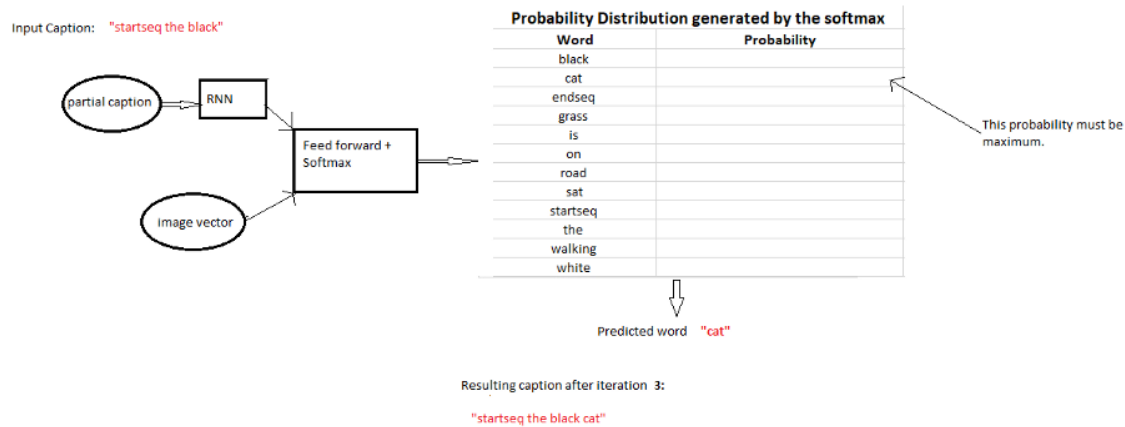
Iteration 2

Iteration 3:

Input: Image vector + "startseq the black"

Expected Output word: "cat"





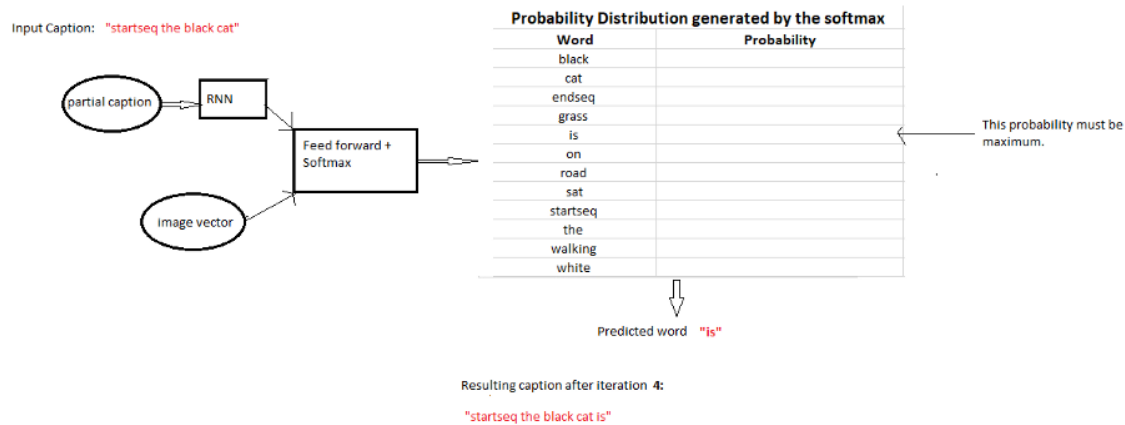
Iteration 3

Iteration 4:

Input: Image vector + "startseq the black cat"

Expected Output word: "is"





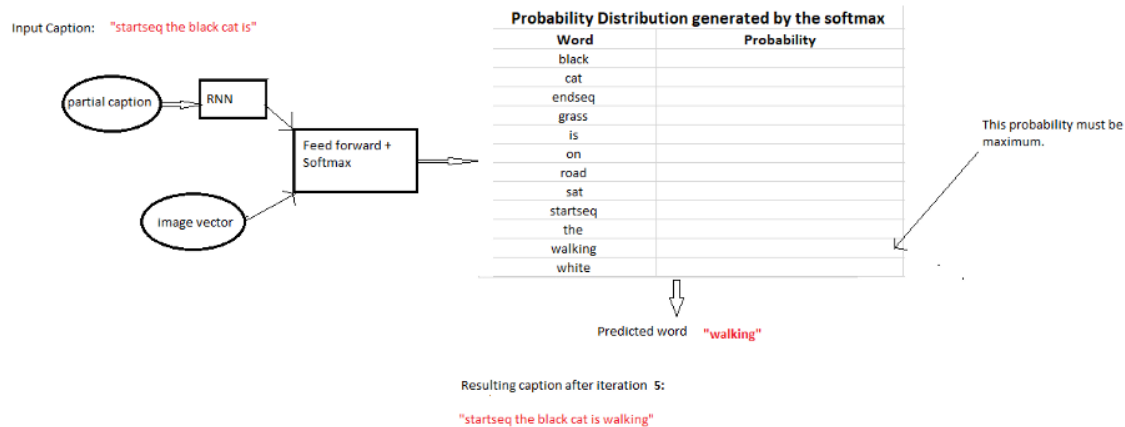
Iteration 4

Iteration 5:

Input: Image vector + "startseq the black cat is"

Expected Output word: "walking"





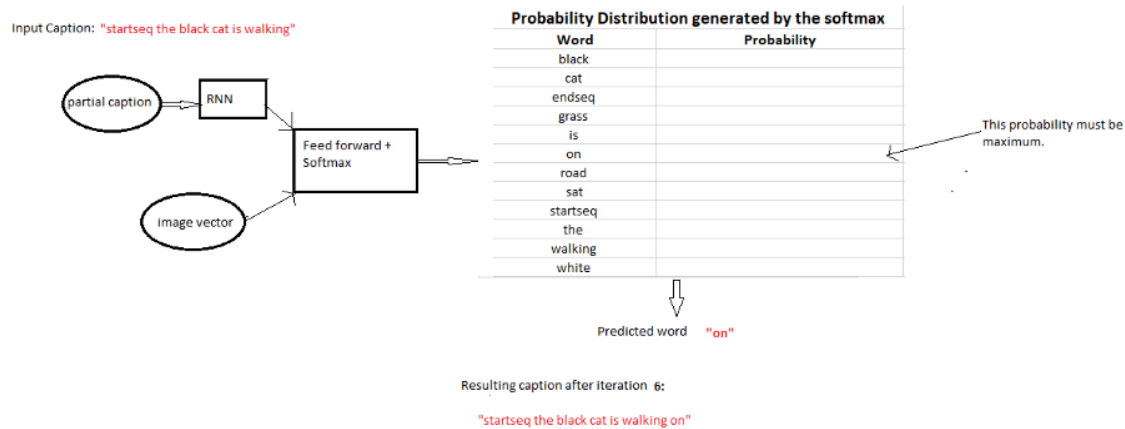
Iteration 5

Iteration 6:

Input: Image vector + "startseq the black cat is walking"

Expected Output word: "on"





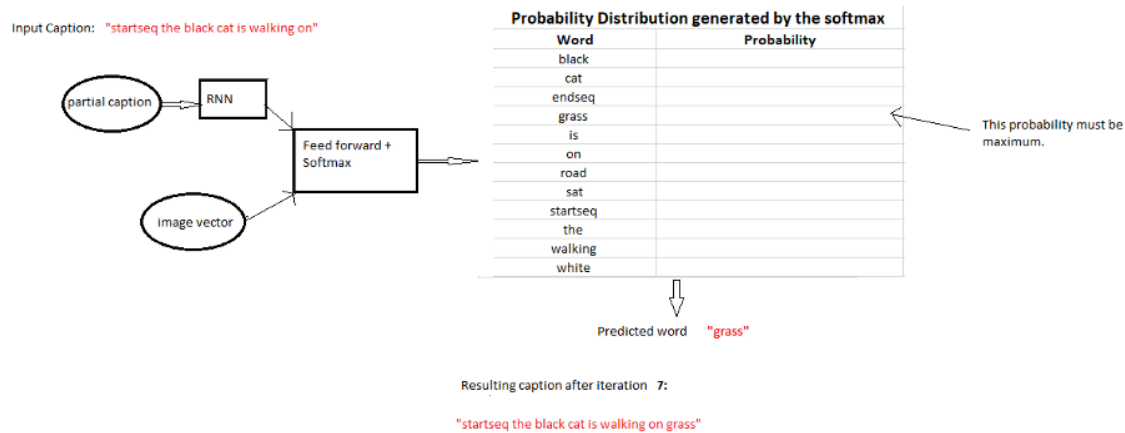
Iteration 6

Iteration 7:

Input: Image vector + "startseq the black cat is walking on"

Expected Output word: "grass"





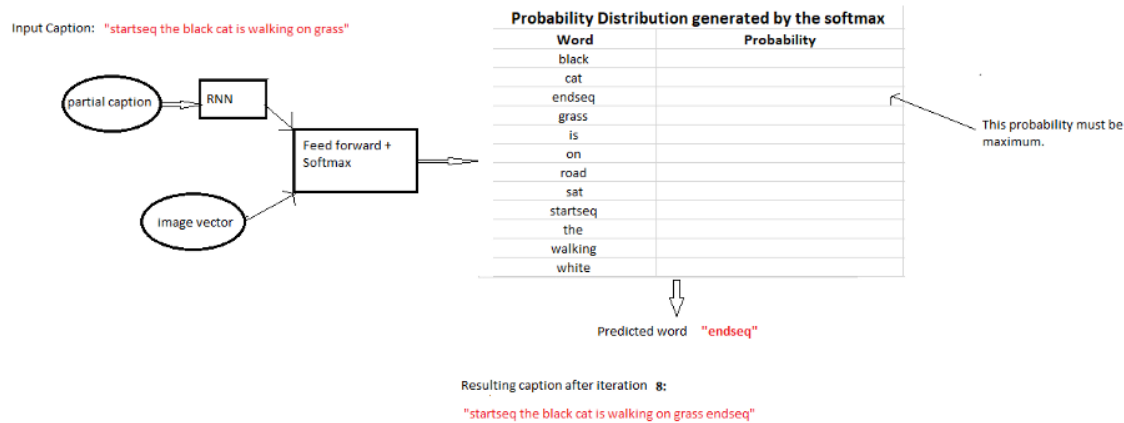
Iteration 7

Iteration 8:

Input: Image vector + "startseq the black cat is walking on grass"

Expected Output word: "endseq"





Iteration 8

This is where we **stop** the iterations.

So we stop when either of the below two conditions is met:

- We encounter an '**endseq**' token which means the model thinks that this is the end of the caption. (You should now understand the importance of the 'endseq' token)
- We reach a maximum **threshold** of the number of words generated by the model.

If any of the above conditions is met, we break the loop and report the generated caption as the output of the model for the given image. The code for inference is as follows:

Inference with greedy search

14. Evaluation

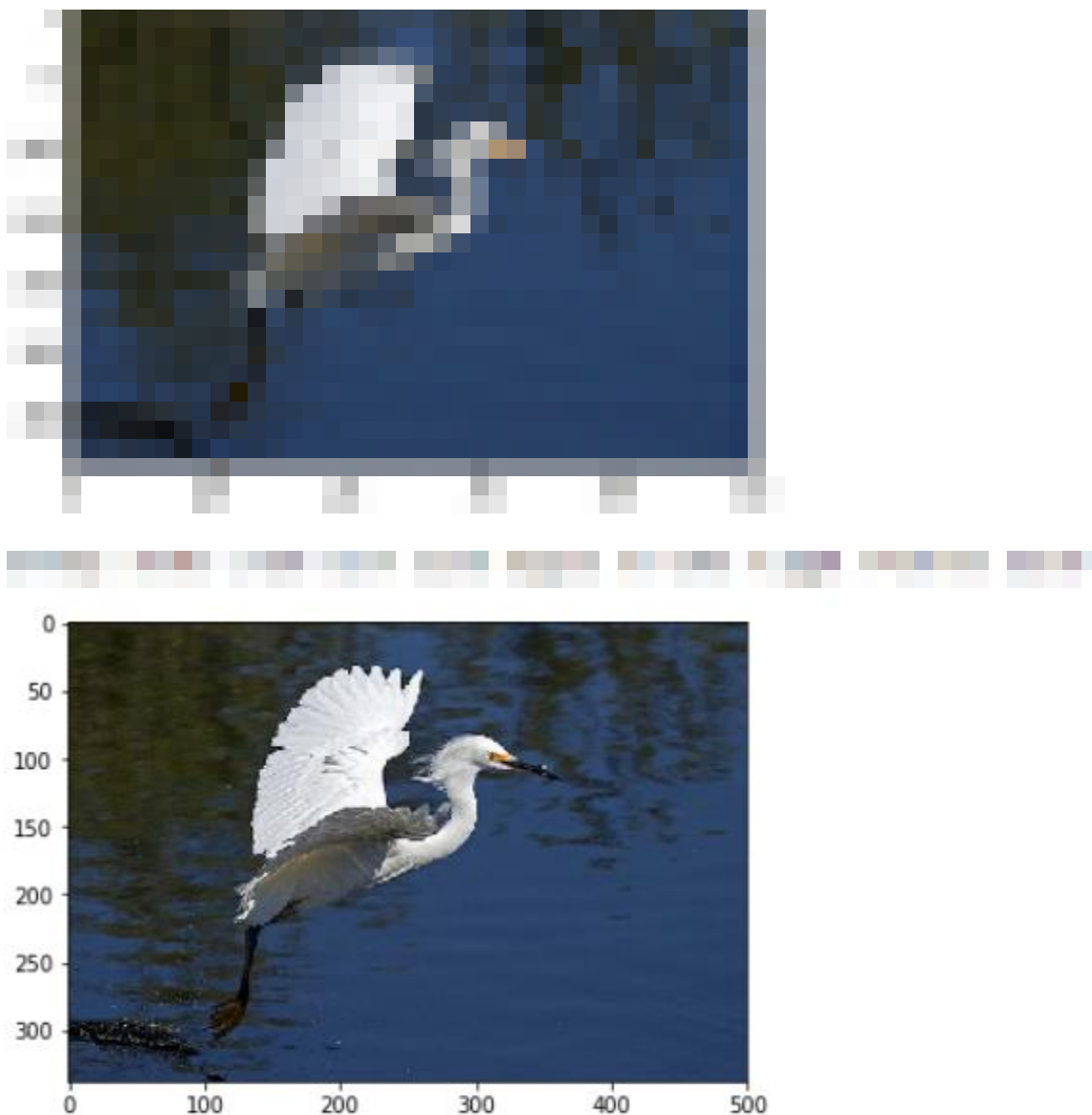
To understand how good the model is, let's try to generate captions on images from the test dataset (i.e. the images which the model did not see during the training).



Greedy: motorcyclist is riding an orange motorcycle

Output — 1

Note: We must appreciate how the model is able to identify the colors precisely.



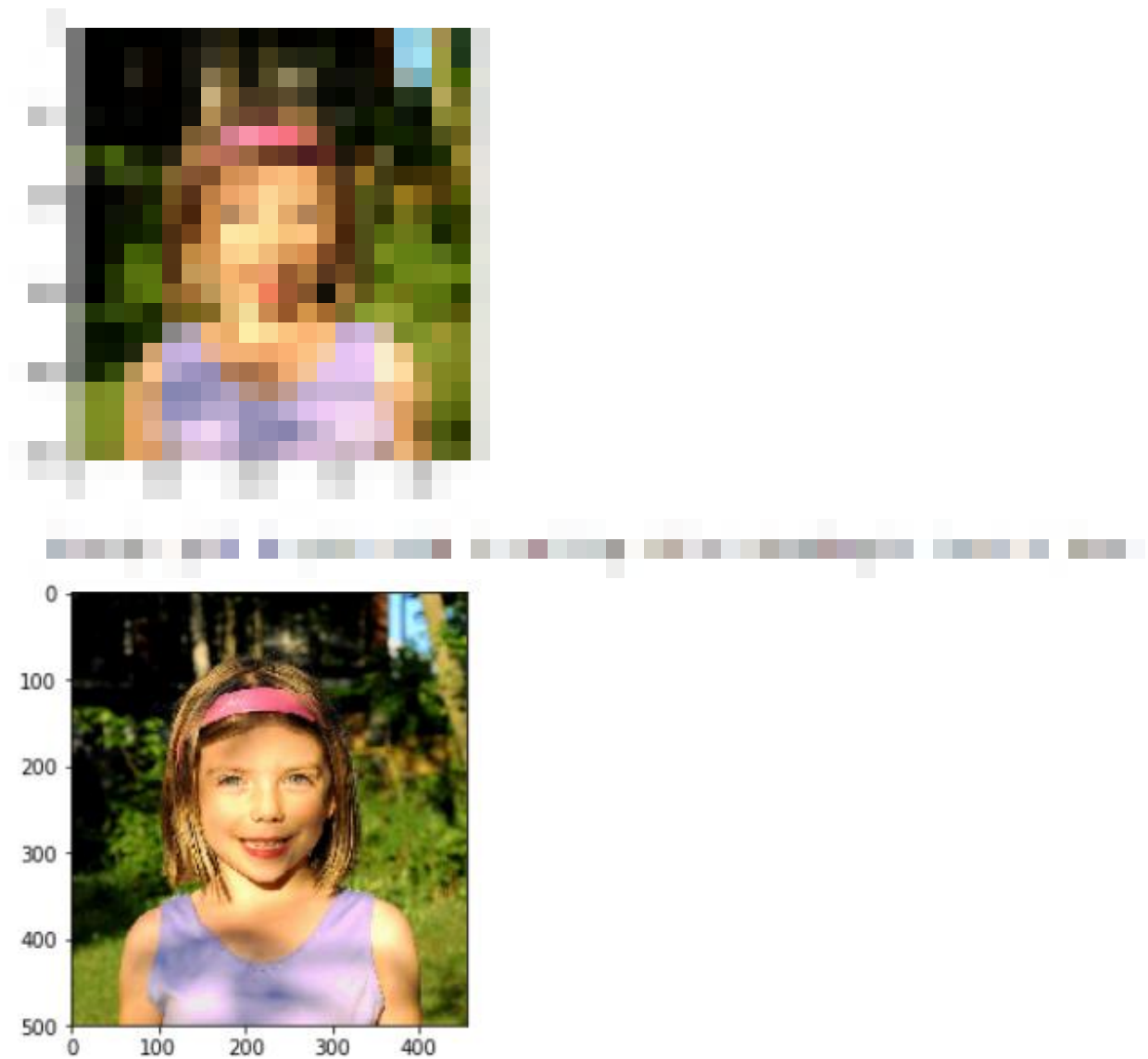
Greedy: white crane with black begins to take flight from the water

Output — 2



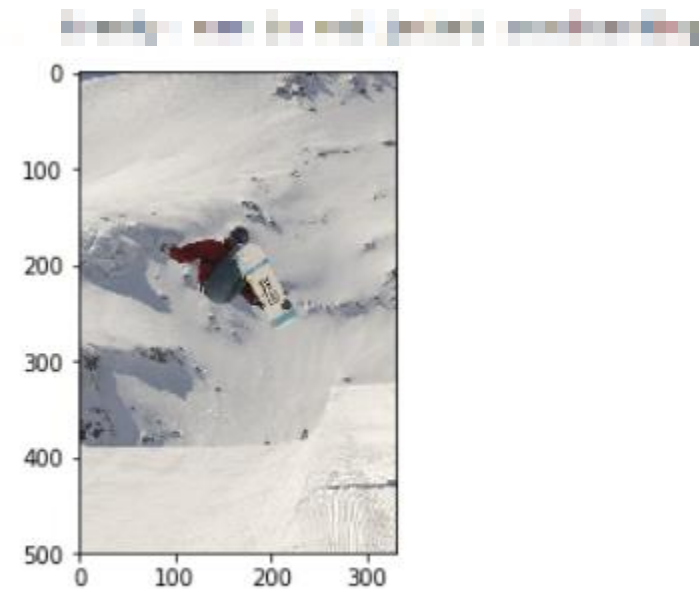
15 Greedy: race car spins down the road as spectators watch

Output — 3



Greedy: girl in pink shirt is smiling whilst standing in front of tree

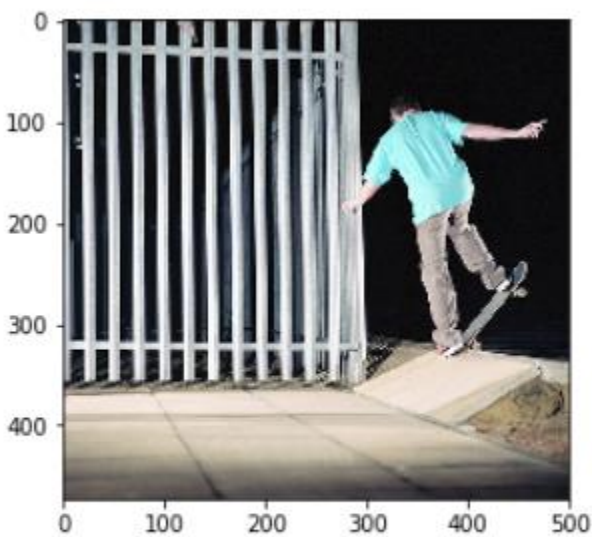
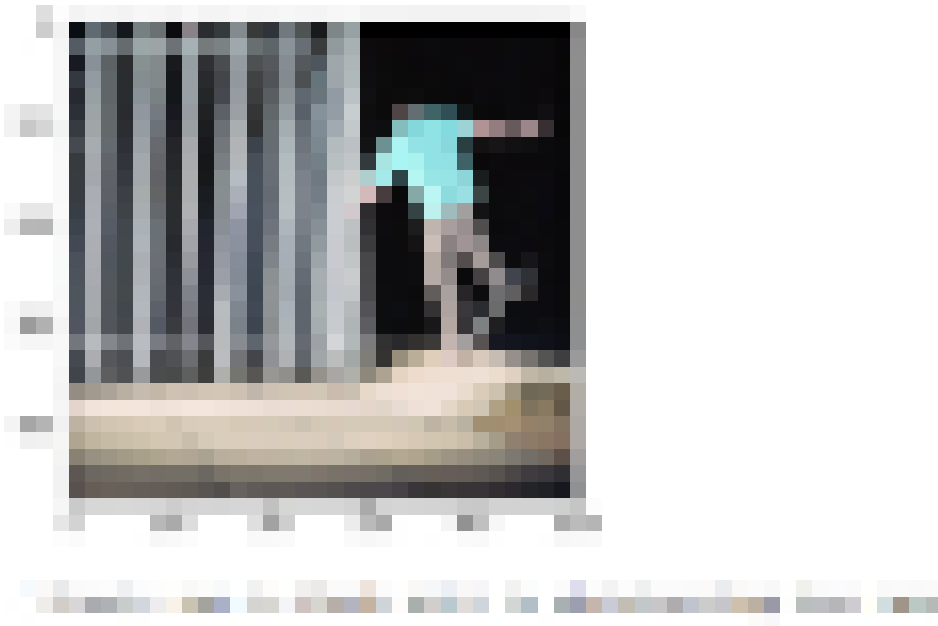
Output — 4



Greedy: man in red jacket snowboarding

Output — 5

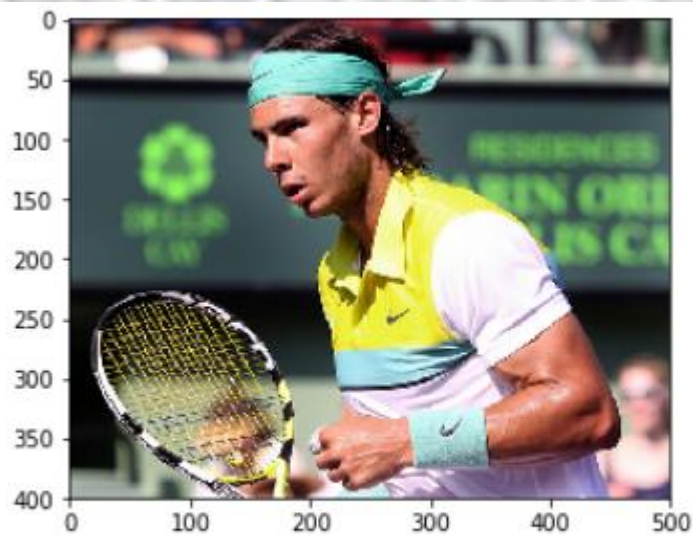
Of course, I would be fooling you if I only showed you the appropriate captions. No model in the world is ever perfect and this model also makes mistakes. Let's look at some examples where the captions are not very relevant and sometimes even irrelevant.



Greedy: man in black shirt is skateboarding down ramp

Output — 6

Probably the color of the shirt got mixed with the color in the background



Greedy: a woman in a tennis racket on the court .

Output — 7

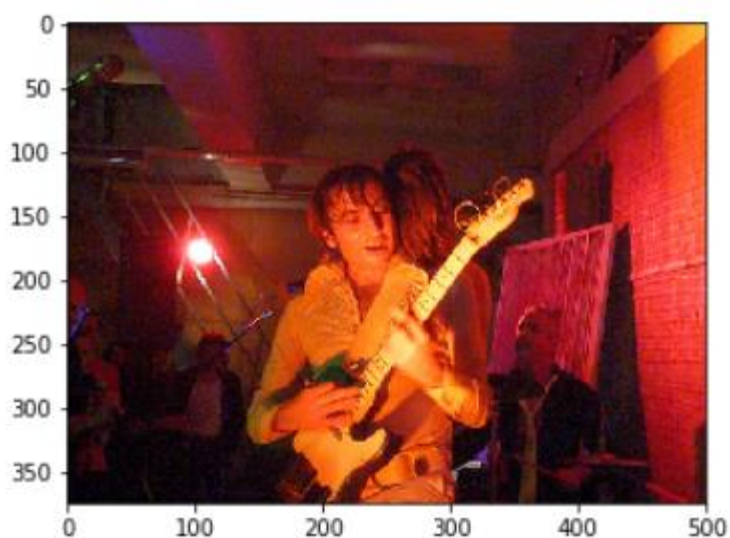
Why does the model classify the famous Rafael Nadal as a woman :-)? Probably because of the long hair.



Greedy: a boy is walking on the beach with the ocean .

Output — 7

The model gets the grammar incorrect this time



Greedy: a man in a guitar for a .

Output — 9

Clearly, the model tried its best to understand the scenario but still the caption is not a good one.



Greedy: a little boy is on the water on the floor with a over the floor

Output — 10

Again one more example where the model fails and the caption is irrelevant.

So all in all, I must say that my naive first-cut model, without any rigorous hyper-parameter tuning does a decent job in generating captions for images.

Important Point:

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.

15. Conclusion and Future work

Thanks a lot if you have reached here. This is my first attempt in blogging so I expect the readers to be a bit generous and ignore the minor mistakes I might have made.

Please refer my GitHub link [here](#) to access the full code written in Jupyter Notebook.

Note that due to the stochastic nature of the models, the captions generated by you (if you try to replicate the code) may not be exactly similar to those generated in my case.

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 **overfitting**.
- 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 :-)

16. References

1. <https://cs.stanford.edu/people/karpathy/cvpr2015.pdf>
2. <https://arxiv.org/abs/1411.4555>
3. <https://arxiv.org/abs/1703.09137>
4. <https://arxiv.org/abs/1708.02043>
5. <https://machinelearningmastery.com/develop-a-deep-learning-caption-generation-model-in-python/>
6. <https://www.youtube.com/watch?v=yk6XDFm3J2c>
7. <https://www.appliedaicourse.com/>

PS: Feel free to provide comments/criticisms if you think they can improve this blog, I will definitely try to make the required changes.