GAUTAM BUDDHA UNIVERSITY

|  |  |
| --- | --- |
| **automatic image captioning**  Using convolutional neural networks, recurrent neural networks and transfer learning. | MAJOR PROJECT DONE UNDER SUPERVISON OF  **MS. nidhi gulati**  Submitted by :  Rohan Choudhary (16/IEC/034)  Shivam Tiwari (16/IEC/040)  Tushar Jain (16/IEC/052)  Shubham Singh (16/IEC/044) |

DEPARTMENT OF INFORMATION AND COMMUNICATION TECHNOLOGY

Yamuna Expressway, Greater Noida, Gautam Budh Nagar,   
Uttar Pradesh (India) - 201308



**CERTIFICATE**

This is to certify that the project report entitled “**Automatic Image Captioning”**, is submitted by **Rohan Choudhary (16/IEC/034), Tushar Jain (16/IEC/052), Shivam Tiwari (16/IEC/040) and Shubham Singh (16/IEC/044)** for the purpose of major project to the department of information and communication technology of Gautam Buddha University. This record is a bona fide work carried out by them under my supervision. The results embodied in this report have not been submitted to any other university or institute for the award of any degree or other major projects.

**Ms. Nidhi Gulati**

*Project Guide*

*School of information & communication technology*

**DECLARATION**

We hereby declare that the work presented in this project titled **“Automatic Image Captioning”** submitted towards the submission of major project in eight semester of Integrated B.tech and M.tech programme (Electronics and communication engineering with major in Artificial Intelligence and Robotics) at the department of Information and communication technology at Gautam Buddha University, Greater Noida is our authentic work and had not been submitted to any university or institute for any award.

*Place: Greater Noida*

*Date: 10/05/2020*

**Rohan Choudhary**

*(16/IEC/034)*

**Tushar Jain**

*(16/IEC/052)*

**Shivam Tiwari**

*(16/IEC/040)*

**Shubham Singh**

*(16/IEC/044)*

**ACKNOWLEDGEMENT**

We placed on record and warmly acknowledge the continuous encouragement, invaluable supervision of **Ms Nidhi Gulati**, Department of Information and communication technology, Gautam Buddha University, Greater Noida in bringing this report to a successful completion.

We are grateful to **Dr. Sandeep Sharma**, HOD of Information and communication technology, Gautam Buddha University, Greater Noida, for permitting us to make use of the facilities available in the department to carry out the project successfully. Last but not the least, we express our sincere thanks to all our friends who have patiently extended all sorts of help for accomplishing this undertaking.

Finally, we extend our gratitude to one and all who are directly or indirectly involved in the successful completion of the project work.

**Table of Contents**

[Abstract 5](#_Toc40847108)

[Introduction 6](#_Toc40847109)

[1 Neural Network 7](#_Toc40847110)

[1.1 Perceptron 7](#_Toc40847111)

[1.2 Multi-Layer Neural Network 7](#_Toc40847112)

[1.2.1 Fully connected Layer 8](#_Toc40847113)

[1.2.2 Number of Parameters 9](#_Toc40847114)

[1.3 Deep Learning 10](#_Toc40847115)

[1.4 Convolutional Neural Networks 10](#_Toc40847116)

[1.4.1 Receptive Field 11](#_Toc40847117)

[1.4.2 Convolution in CNNs 11](#_Toc40847118)

[1.4.3 Pooling Layer 12](#_Toc40847119)

[1.5 Recurrent Neural Networks 12](#_Toc40847120)

[1.5.1 Long Short-Term Memory 13](#_Toc40847121)

[2 Transfer Learning 14](#_Toc40847122)

[2.1 Convolutional Neural Networks 14](#_Toc40847123)

[3 Motivation 15](#_Toc40847124)

[4 Data 15](#_Toc40847125)

[4.1 Understanding the data 15](#_Toc40847126)

[5 Project structure and workflow 16](#_Toc40847127)

[5.1 Data cleaning 16](#_Toc40847128)

[5.2 Loading the training set 16](#_Toc40847129)

[5.3 Data pre-processing (images) 17](#_Toc40847130)

[5.4 Data pre-processing (captions) 17](#_Toc40847131)

[5.5 Data preparation 18](#_Toc40847132)

[5.6 Model architecture 21](#_Toc40847133)

[5.7 Hyper parameters during training 22](#_Toc40847134)

[6 Source code 23](#_Toc40847135)

[7 Predicting captions 34](#_Toc40847136)

[8 Results 39](#_Toc40847137)

[Conclusion and future work 41](#_Toc40847138)

[References 42](#_Toc40847139)

# Abstract

Automatic image captioning refers to the problem of constructing natural language description of an image. This is an important problem with practical significance that involves two major artificial intelligence domains - computer vision and natural language processing. In this project, we used multi-task learning to solve the automatic image captioning problem.

This project aims at generating captions for images using neural language models. There has been a substantial increase in number of proposed models for image captioning task since neural language models and convolutional neural networks (CNN) became popular. Our project has its base on one of such works, which uses a variant of recurrent neural network coupled with a CNN. We intend to enhance this model by making subtle changes to the architecture and using transfer learning for the image processing part instead of creating features from the image on our own, which may lead to better semantic and syntactical captions.

Keywords: image captioning, dense captioning, convolutional neural networks, long short-term memory, transfer learning.

# Introduction

For most computer vision researchers the classification task has always been dominant in the field. Either it was a scene understanding in the pioneer 1960s or a traffic sign detection in the modern days, the task has been rooted in the soil of computer vision. It is not surprising that one of the most significant competition in the field comprises the image classification task among others.

The ImageNet Large Scale Visual Recognition Challenge (ILSVRC) awards annually the algorithm which is most successful at predicting the class of an image in its five estimates (known as top-5 error). For the record, the lowest top-5 classification error reached 28.2% at the ILSVRC2010 and 25.8% a year later, respectively.

Nonetheless, an unexpected breakthrough came in the year 2012 when A. Krizhevsky presented decades old algorithms enhanced by novel training techniques achieving so-far-not-seen results. In particular, the top-5 classification error was pushed to 16.4%. At the latest contest in 2015, the lowest top-5 error was brought to 3.5%, drawing on the work of Krizhevsky. After this success, neural networks has revolutionised the field and brought in new challenges that had not been merely considerable before. One of those newly feasible techniques – image captioning – is discussed in this major project. In fact, as an arising discipline with promising potential, image captioning still is an active area of research nowadays, striving to answer unsolved questions. Consecutively, since the field has not been entirely established yet, one must rely mainly on recently published papers and on-line lectures only.

Considering recent work, we aim to define image captioning as a task in which an algorithm describes a particular image with a statement. However, it is expected that the statement is meaningful, self-contained and grammatically and semantically correct. In other words, the caption shall describe the image concretely, shall not require or rely on additional information and, last but not least, be consisted of a grammatically correct sentence that semantically corresponds to the image.

# 1 Neural Network

In order to tackle the image captioning task, recent work shows it is in one’s interest to utilize neural networks. This frequently used term dates back to 1950s when notions such as the Perceptron Learning Algorithm were introduced. Modern neural networks draw on notions discovered in the era of a Perceptron. In this section, we first define a neuron as a fundamental part of modern neural networks. Then we elaborate on Convolutional Networks and Recurrent Networks.

## 1.1 Perceptron

For the purposes of this work, a perceptron is defined generally as it became a fundamental part of modern neural networks and the notation is utilized further on. Thus, a perceptron is compounded of one neuron. The neuron’s output, known as the activation a, is mapped by as follows:

Where is a feature is vector, and are weights and is a non-linear function. In case of the Perceptron, stands for

(2)

In other words, a perceptron is a non-linear function separating data into two classes each associated with either 1 or 0. A perceptron is parameterized by weights w and b. By setting proper weights, one effects the output and the perceptron’s behaviour for a given feature vector. Therefore, such weights trimming is essential, yet non-trivial task. In order to find the weights, a learning algorithm was introduced, named the Perceptron Learning Algorithm. This algorithm has a limiting property such that successful learning is achieved if and only if the data are linearly separable which is a major drawback pointed out in by Minsky and Papert in 1969. For example, there is no vector w and bias b that would make a perceptron mimic the XOR function.

## 1.2 Multi-Layer Neural Network

Taking the perceptron as inspiration, the XOR problem can be overcome by aligning neurons into layers and interconnecting those layers. This function is called a Feed forward Neural Network, an Artificial Neural Network or simply a Neural Network.

In a neural network, each layer comprises neurons processing inputs coming from the previous layer and producing activations used later in the following layer. For the sake of simplicity, it is now assumed that the number of neurons N is same for all layers. However, this varies very often, for example, usually the output layer consists of fewer neurons corresponding to the nature of the problem being solved. To conclude, the activations of the k-th layer are each a function of the activations of the previous layer, noted as :

.

Where is a function defining the properties of the i-th neuron in the layer k, often having the form similar to Eq. (1). However, there are exceptions that proved to be essential to modern neural networks designs.

In practise, to lower the complexity of a network, in the given layer k all the functions are always of the same form, only distinct in weights. Therefore, it is convenient to use vector notation. This is done by simplifying Eq. (3) into the following form:

As an example, we show a neural network with one hidden layer. The network takes in a vector that is propagated forward into the hidden layer. The processes in the hidden layer are noted here as . Further, the activations of the hidden layer are again propagated, analogically, into the second layer whose activations are the output of the network. The network’s design is shown in Fig. 1. Formally, the network is fed with a feature vector producing a vector :

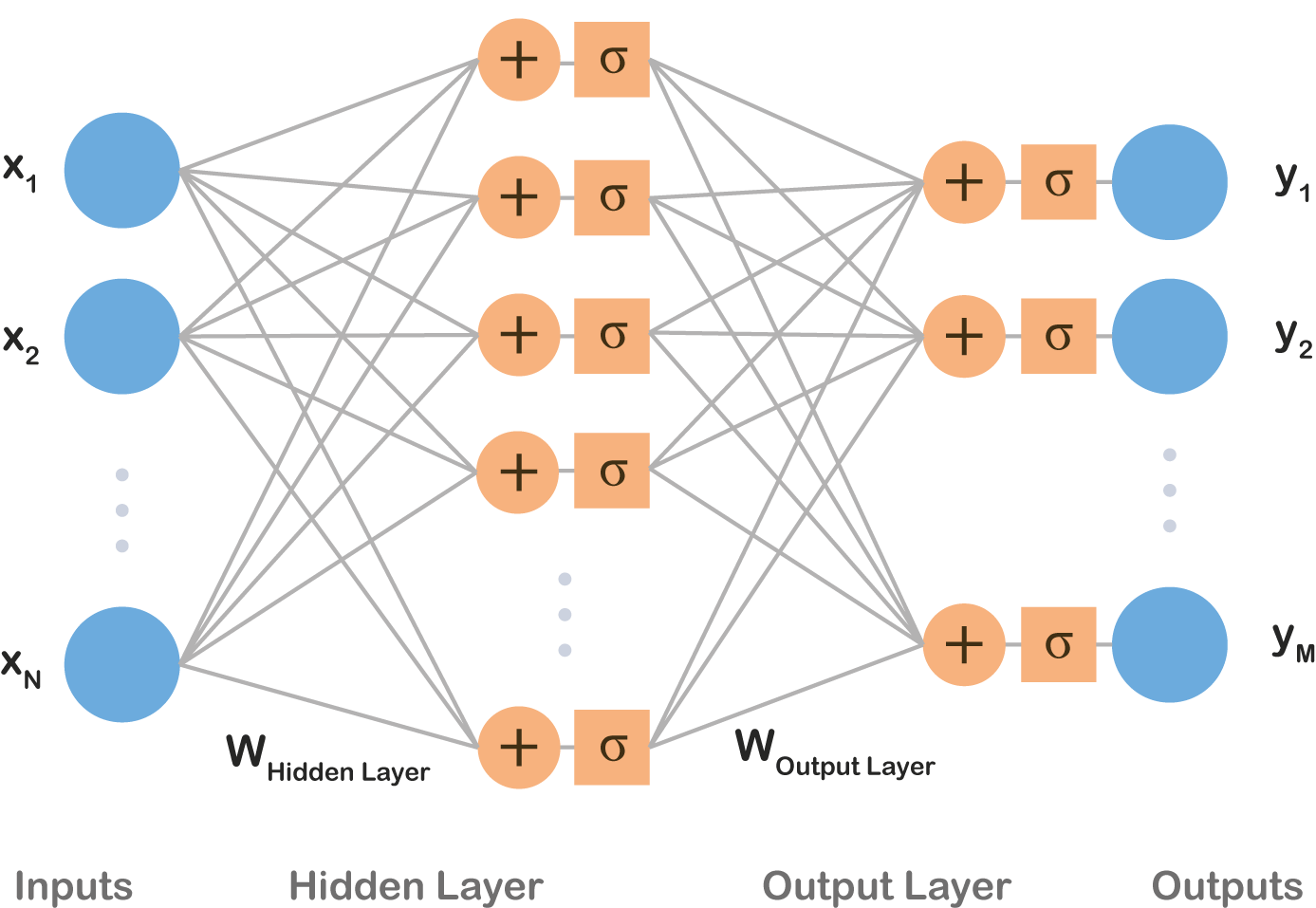
where is called the hidden layer and is called the output layer. Note that the number of neurons in the hidden layer H is a hyper-parameter.

Although the structure of the network in the example has been defined, there are still other hyper-parameters to be determined. For example, the form of the layer mappings and needs to be specified. A layer with the simplest form of its mapping is called a Fully Connected Layer and is discussed in the following subsection.

### **1.2.1 Fully connected Layer**

In the most basic neural network - a feedforward neural network comprising fullyconnected layers only – each neuron processes activations of all neurons in the previous layer and is activated using defined in Eq. (1). Thus, based on vector notation in Eq. (4), the activations of the k-th fully-connected layer are defined as:

where is a weights matrix with weights vectors aligned in rows, is a vector of biases and is a non-linear function. Note that, since in practise the elements are identical single-variable functions, we refer to them as simply



**Figure 1** : Inputs x1, . . . , xN are processed by a hidden layer and, consecutively, by an output layer producing outputs y1, . . . , yM . Biases b were omitted for the sake of simplicity.

In contrast to a perceptron, *σ*(·) is generally required to be differentiable due to the nature of learning algorithms used in the field. For example, *σ*(·) used to be set to a sigmoid curve as similar to the perceptron’s activation fuction shown in Eq. [(2).](#_bookmark6) Most commonly, tanh() or the logistic function (Eq. [(7))](#_bookmark14) were used :

Nevertheless, when used in deep learning sigmoids suffer from problems such as vanishing or exploding gradients, therefore those were replaced with a Rectified Linear Unit (ReLU):

(8)

In modern networks, it is recommended to use ReLUs as they proved to provide better results and are thus the most common activation function used nowadays.

Drawing on the example presented above, we now assume that both layers are fully connected, meaning that layer mappings have a form of Eq. [(6).](#_bookmark12) Then Eq. [(5)](#_bookmark10) can be rewritten as follows:

(9)

### **1.2.2 Number of Parameters**

Let us now assume , the activations of the hidden layer and . Then we can calculate the number of parameters as . Considering a small gray-scale image, 28 × 28, of a hand-written number taken from the MNIST dataset, that is classified as 0-9 digit, *N* = 784 and *M* = 10. Then the number of parameters, needed to be learned, is 795 × *H* + 784 where *H*, the number of hidden layers, is a hyper-parameter. For a hidden layer having the same width as the input vector, i.e. *H* = 28 in this example, the number of parameters reaches 23*,* 044.

Truly, this is a large number for such a shallow network suggesting that fully con- nected layers extensively increase the number of parameters.

## 1.3 Deep Learning

In spite of former beliefs, it was found that it is more efficient to insert several hidden layers one by one and propagate information sequentially creating a deep structure, instead of utilizing a shallow network given in the example. This concept is called deep learning and, surprisingly, has its roots already in the pioneer 1960s as it was assumed that an intelligent algorithm solving complex problems shall work with hierarchy of concepts that was rather deep. This is why we get the name deep learning.

The notion was later found in the idea of modern neural networks which, as stated above, consist of numerous nested layers each extracting more abstract and complex features as information is propagated forward the network. Therefore, the modern neural networks and techniques used for learning them are usually nowadays referred to as deep learning.

Deep neural networks were introduced already in 1998 and the optimization algorithm (back-propagation) was known by then and used frequently[.](#_bookmark71) Yet, the deep nets were found too complex to be trained. In their books, Ian Goodfellow list those reason that enabled the boom of neural networks in 2012: firstly, more data were available as well, therefore, the deep nets have started to outperform other models. Secondly, deeper models require decent architectures both in software and hardware and those had become available. Then on, promising results enabled advent of neural networks, especially in their deep form. Models such as convolutional neural networks or recurrent nets are considered state-of-the-art building blocks nowadays. Their detailed design is discussed in the following subsections.

## 1.4 Convolutional Neural Networks

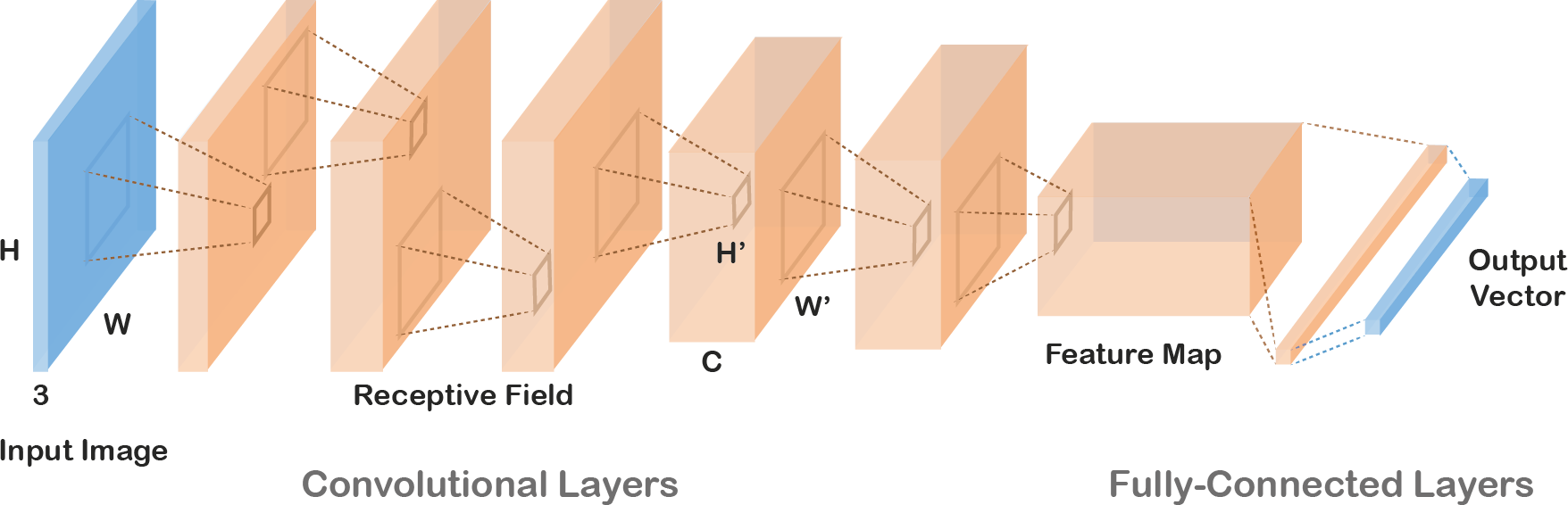
In image analysis, many of recent advances in deep learning are built on the work of LeCun who introduced a Convolutional Neural Network (CNN) which had a large impact on the field. A CNN is a type of a neural network that is designed to process an image and represent it with a vector code. The architecture of CNNa draws on fully-connected neural networks. Similarly, a convolutional neural network is a compounded structure of several layers processing signals and propagating them forward.

However, in contrast to a vector activation in a fully-connected layer, activations in CNNs have a shape of three-dimensional tensors. Commonly, this output tensor is called a feature map. For instance, an input image of shape is transformed by the first convolutional layer into a feature map of shape , where *C* is the number of features. In other words, a convolutional layer transforms a volume into a volume.

A typical CNN consists of several convolutional layers and, at the top, fully connected layers that flatten convolutional volumes into a vector output. In the field’s terminology, this vector code of an image is often called fc7 features as it used to be extracted from the seventh fully connected layer of AlexNet. Even though AlexNet has already been outperformed by many and the state-of-the-art designs are different from AlexNet, the term maintained its popularity. Additionally, depending on a problem the network is supposed to solve, an additional layer, such as soft-max, can be added on top of fc7 features. A common design of a CNN is depicted in Fig. 2

### **1.4.1 Receptive Field**

As mentioned above, a convolutional layer takes a tensor on input and produces a tensor, too. Note that these tensors have two spatial dimensions *W* and *H*, and one feature dimension *C* as they copy the form images are stored in. The context conveyed by the spatial dimensions is utilized in the CNN design which takes into account correlations in small areas of the input tensor called receptive fields. Concretely, in contrast to a neuron in a fully connected layer that processes all activations of the previous layer, a neuron in a convolutional layer ”sees” only activations in its receptive field. Instead of transforming layer’s activations it restraints to a specific small rectangular shaped subset of the activations. When mentioning a receptive field, it is often expected only spatial dimensions of the input volume are referred to, i.e. a receptive field defines an area in the *W* × *H* grid. The shape of the receptive field is a hyper-parameter and varies across the models.



**Figure 2 :** A convolutional neural network takes an image on input (in blue) and transforms it into a vector code (in blue). Convolutional Neural Networks are characteristic for processing volumes. An output of each layer is illustrated as an orange volume. Each neuron process only activations in the previous layer that belong to its receptive field. The same set of weights is used for neurons across the whole grid. On top of convolutional layers, fully-connected layers are commonly connected.

### **1.4.2 Convolution in CNNs**

A neuron’s receptive field is processed similarly to fully connected layer neurons. The values below the receptive field along the input tensor’s full depth are transformed by a non-linear function, typically ReLU (Eq. [(8)).](#_bookmark15)

However, in contrast to fully connected layer neurons, the same set of weights (referred to as a kernel) is used for all receptive fields in the input volume resulting into a transformation that has a form of convolution across the input. A kernel is convolved across *W* and *H* spatial dimensions. Then, a different kernel is again convolved across the input volume producing another 2D tensor. Aligning up the output tensors into a volume assembles the layer’s output feature map.

This is an important property of convolutional neural networks because each kernel detects a specific feature in the input. For example, in the first layer, the first kernel would detect presence of horizontal lines in the receptive fields, the second kernel would look for vertical lines, and similarly further on. In fact, learning such types of detectors in the bottom layers is typical for CNNs.

The design of CNNs has an immensely practical implication – since a kernel is convolved across the input utilizing the same set of weights and it covers only the receptive field, the number of parameters is significantly reduced. Therefore, convolutional layers are less costly in terms of memory usage and the training time is shorter.

### **1.4.3 Pooling Layer**

Convolutional layers are designed in such a way the spatial dimensions are preserved and the depth is increased along the network flow. However, it is practical to reduce spatial dimensions, especially in higher layers. Dimensions reduction can be obtained by using stride when convolving, leading to dilution of receptive fields overlap. Nevertheless, a more straightforward technique was developed called a pooling layer. An input is partitioned into non-overlapping rectangles and the layer simply outputs a grid of maximum values of each rectangle. In practice, pooling layers are inserted often in between convolutional layers to reduce dimensionality.

## Recurrent Neural Networks

Convolutional and fully connected layers are designed to process input in one time step without temporal context. Nonetheless, some tasks require concerning sequences where data are temporally interdependent. For that, a Recurrent Neural Network (RNN) – an extension of fully connected layers – has been introduced. RNNs are neural networks concerning information from previous time steps.

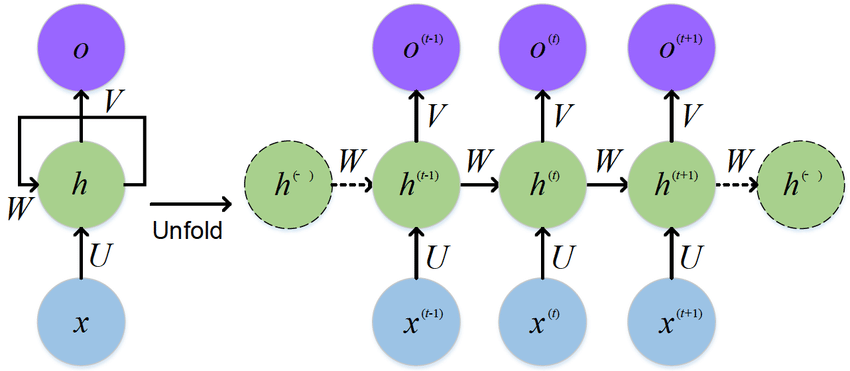
RNNs are used in a variety of tasks: transforming a static input into a sequence (e.g. image captioning); processing sequences into a static output (e.g. video labelling); or transforming sequences into sequences (e.g. automatic translation).

A simple recurrent network is typically designed by taking the layer’s output from the previous step and concatenating it with the current step input:

(10)

The function *f* is a standard fully-connected layer that processes both inputs indistinctly as one vector. Due to its simplicity, this approach is rather not sufficient and does not yield promising results. Thus, in past years, a great number of meaningful designs have been tested. The notion was advanced and designs have become more complex. For example, an inner state vector was introduced to convey information between times steps:

(11)

The most popular architecture nowadays is a Long Short-Term Memory (LSTM) – a rather complex design, yet outperforming others.

**Figure 3** : The standard RNN and unfolded RNN

### **1.5.1 Long Short-Term Memory**

A standard LSTM layer is given as follows:

(12)

(13)

(14)

(15)

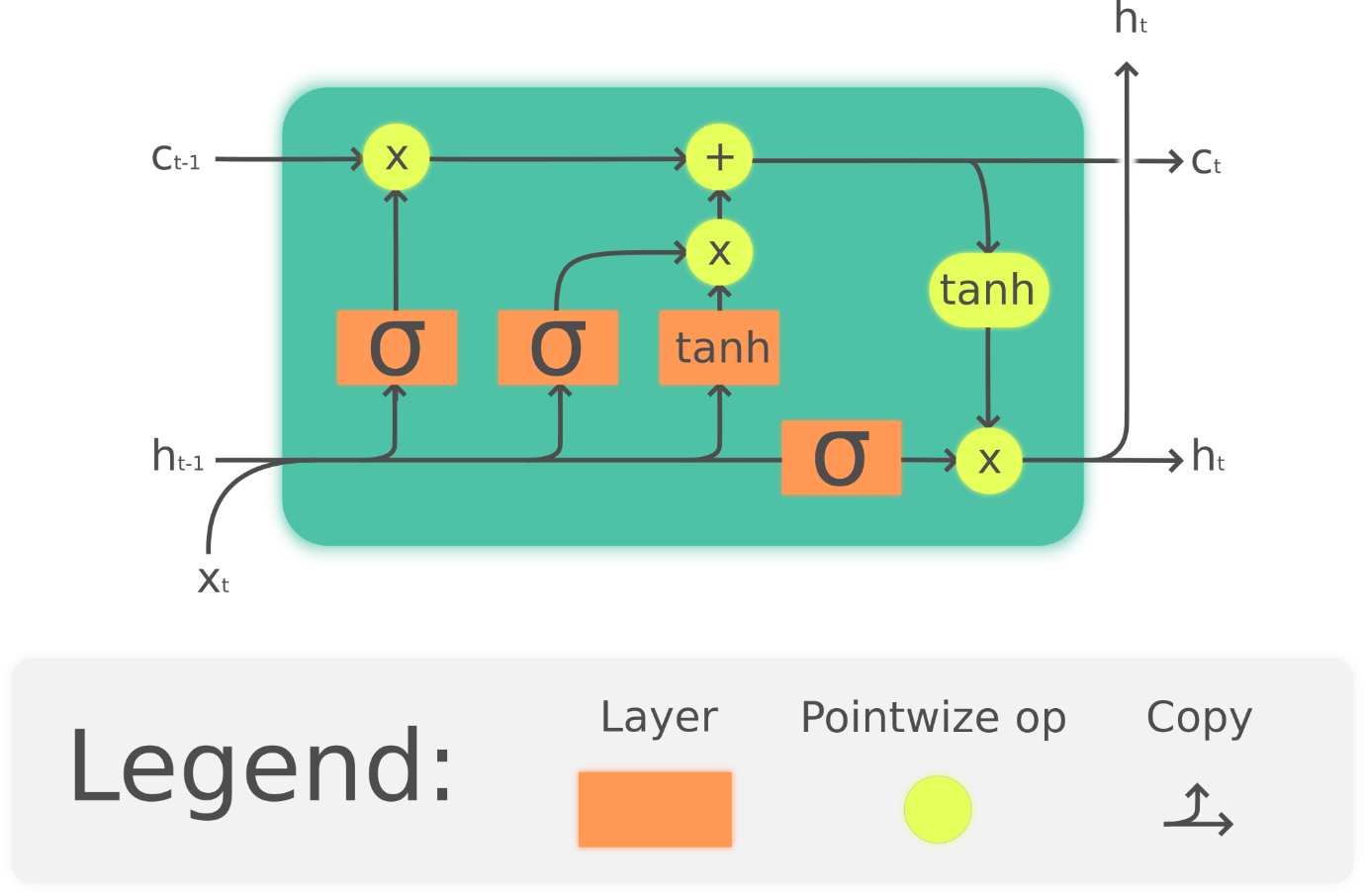
(16)

where is an input vector and is an output vector. All matrices **W** and **U** and biases *b* are weights that together, with which is a logistic function Eq. [(7),](#_bookmark14) represent a standard neural network layer.

Thus, the forget gate vector , the input gate vector and output gate vector are outputs of three distinct one-layer neural nets each having its output between −1 and 1. is a cell state vector that, as a hidden output, is propagated to the next time step. is an output of the LSTM cell. Ⓢ stands for element-wise multiplication, and are usually set to tanh.

Note that is a combination of the previous time step, element-wisely adjusted by the forget gate, and the output of a neural network, gated similarly by the input gate .

The output of LSTM is a function of the cell state vector, first squashed between 0 and 1, and then adjusted by the output gate .

Connected to a network, LSTM consists typically of one layer only. LSTMs are known to preserve long-term dependencies, as shown for example by Karpathy.

**Figure 4** : The Long Short-Term Memory (LSTM) cell can process data sequentially and keep its hidden state through time.

# 2 Transfer Learning

Transfer learning is a popular method in computer vision because it allows us to **build accurate models in a timesaving way** (Rawat & Wang 2017). With transfer learning, instead of starting the learning process from scratch, we start from patterns that have been learned when solving a different problem. This way we leverage previous learnings and avoid starting from scratch.

In computer vision, transfer learning is usually expressed through the use of **pre-trained models**. A pre-trained model is a model that was trained on a large benchmark dataset to solve a problem similar to the one that we want to solve. Accordingly, due to the computational cost of training such models, it is common practice to import and use models from published literature (e.g. [VGG](https://arxiv.org/pdf/1409.1556.pdf), [Inception](https://arxiv.org/pdf/1512.00567.pdf), [MobileNet](https://arxiv.org/pdf/1704.04861.pdf" \t "_blank)). We have used InceptionV3 for this project.

## 2.1 Convolutional Neural Networks

Several pre-trained models used in transfer learning are based on large convolutional neural networks (CNN) (Voulodimos et al. 2018). In general, CNN was shown to excel in a wide range of computer vision tasks (Bengio 2009). Its high performance and its easiness in training are two of the main factors driving the popularity of CNN over the last years. A typical CNN has two parts:

1. Convolutional base, which is composed by a stack of convolutional and pooling layers. The main goal of the convolutional base is to generate features from the image. For an intuitive explanation of convolutional and pooling layers, please refer to Chollet (2017).
2. Classifier, which is usually composed by fully connected layers. The main goal of the classifier is to classify the image based on the detected features. A fully connected layer is a layer whose neurons have full connections to all activation in the previous layer.

Figure 5 shows the architecture of a model based on CNN. Note that this is a simplified version, which fits the purposes of this text. In fact, the architecture of this type of model is more complex than what we suggest here. One important aspect of these deep learning models is that they can automatically learn **hierarchical feature representations**. This means that features computed by the first layer are general and can be reused in different problem domains, while features computed by the last layer are specific and depend on the chosen dataset and task. According to Yosinski et al. (2014), ‘if first-layer features are general and last-layer features are specific, then there must be a transition from general to specific somewhere in the network’. As a result, the convolutional base of our CNN — especially its lower layers (those who are closer to the inputs) — refer to general features, whereas the classifier part, and some of the higher layers of the convolutional base, refer to specialised features.

Figure 5 : Architecture of model based on CNN

# 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.

1. 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.
2. 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
3. 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.
4. 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 Data

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 major project, we have used the Flickr 8k dataset which isprovided 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, so for the sake of computational capacity we chose this dataset.

This dataset contains 8000 images each with 5 captions. These images are bifurcated as follows:

1. Training set - 6000 images
2. Dev set - 1000 images
3. Test Set - 6000 images

## 4.1 Understanding the data

We have two folders namely, Flicker8k\_Dataset and Flickr8k\_text. The first one contains all the eight thousand images whereas the other one contains text files related to those images. The file, Flickr8k.token.txt contains image names with five captions corresponding to each image.

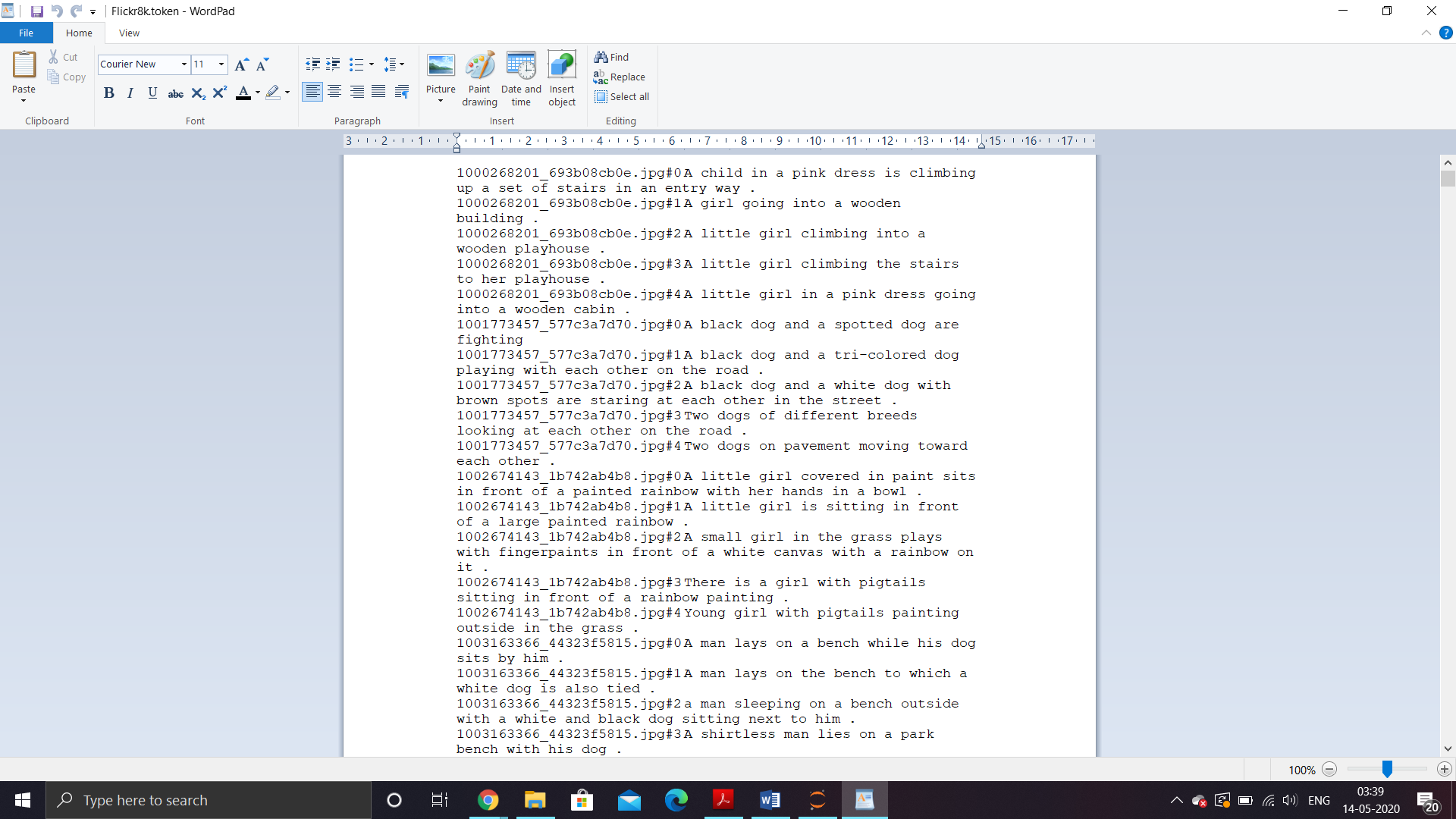


Figure 6 : Flickr8k.token.txt sample

# 5 Project structure and workflow

In this section we will discuss the pre-processing done to the data and the model infrastructure. We begin by creating a dictionary named, ‘descriptions’ which contains the name of the image (without the .jpeg extension) as keys and a list of the five captions for the corresponding images as values. Following is an example of how the dictionary would look like for an image:

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 .']

## 5.1 Data cleaning

We 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.). We Create a vocabulary of all the unique words present across all the 8000\*5 (i.e. 40000) image captions (**corpus**) in the data set. We end up with 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. After this, we have 1651 unique words in our vocabulary. We will append 0’s (zero padding explained later) and thus total words = 1651+1 = **1652**(one index for the 0).

## 5.2 Loading the training set

We load all the names of the 6000 images that belong to the training set via Flickr\_8k.trainImages.txt, into a list named “train”, followed by loading the description of these images from ‘descriptions.txt’ to a python dictionary ‘train\_descriptions’. However, when we load them, we add two tokens in every caption as follows:

1. **‘startseq’**: This is a start sequence token which will be added at the start of every caption.‘
2. **‘endseq**’: This is an end sequence token which will be added at the end of every caption.

## 5.3 Data pre-processing (images)

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.**

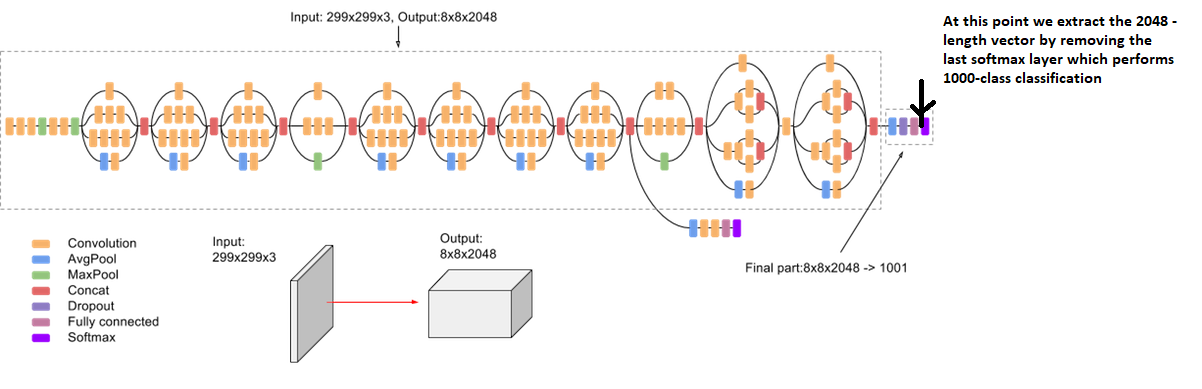
Hence, we just remove the last softmax layer from the model and extract a 2048 length vector (**bottleneck features**) for every image as shown in figure 7.

Figure 7 : Feature Vector Extraction (Feature Engineering) from InceptionV3

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**”. We are saving all the features as a pickle file so that we don’t need to extract features for our whole dataset again and again.

## 5.4 Data pre-processing (captions)

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. 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’

Also we need to find out the maximum length of a caption, after which we’ll get 34 as the max length.

## 5.5 Data preparation

This is probably the most important portion of this project as here we will discuss how to prepare our data which would be convenient to be given as input to the deep learning model. We will try to understand our process through the following example. Consider two images with their corresponding captions:



Caption 1: The black cat sat on grass

Caption 2: The white cat is walking on road

Let’s consider for now that we are going to train our model on these two images.

Firstly, we will convert both the images to their respective 2048 length feature vector. Let Image 1 and Image 2 be those feature vectors for the first two images respectively.

Secondly, we will build a vocabulary for these two images by adding the two tokens “startseq” and “endseq” in both of them:

Caption\_1: “startseq the black cat sat on grass endseq”

Caption\_2: “startseq the white cat is walking on road endseq”

Vocabulary: {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 we’ll try to frame it as a supervised learning problem where we have a set of data points , where is the feature vector of data point ‘’ and is the corresponding target variable. Consider 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…..

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.

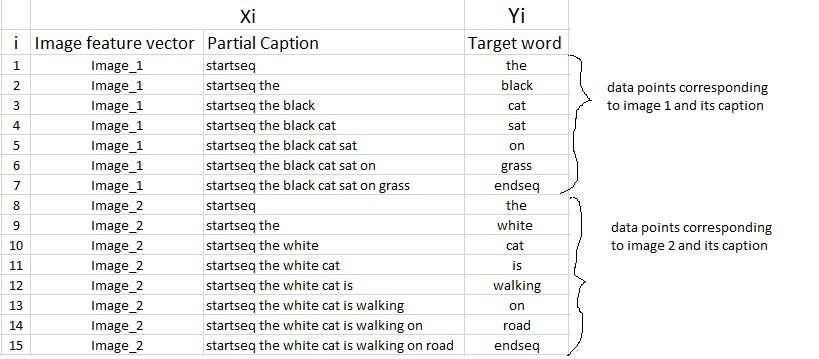
Similarly if we consider both the images and their captions, our data matrix will then look as follows:

Table 1: 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. 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 would be doing **batch processing,** 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. Earlier we calculated the length of the largest caption, so we will append those many number of zeros which will lead to every sequence having a length of 34.

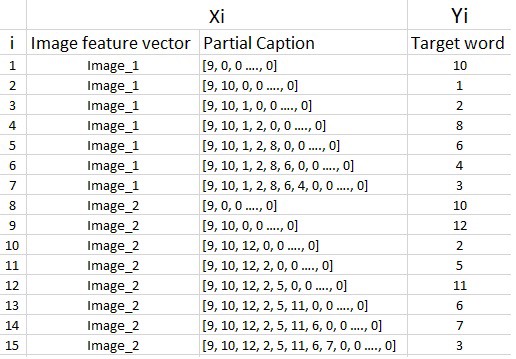


Table 2: Using indices and appending zeros to each sequence to make them all of same length 34

In the above example, we have only considered 2 images and captions which 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.

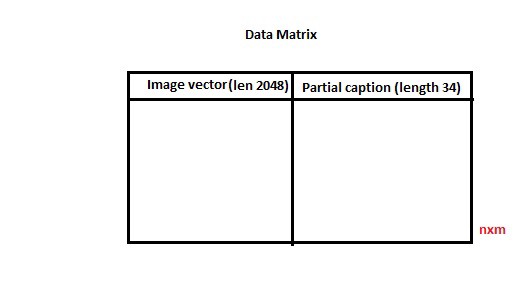


Table 3: Data matrix

Size of the data matrix =, where n is the number of data points (assumed as 210000) and m is the length of each data point.

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

The value of x here won’t be 34 because every word (or index) will be mapped (embedded) to higher dimensional space through one of the word embedding techniques. We will map each word/index 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

Hence,

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. We are going to use stochastic gradient descent (SGD**)**, where 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.

## 5.6 Model architecture

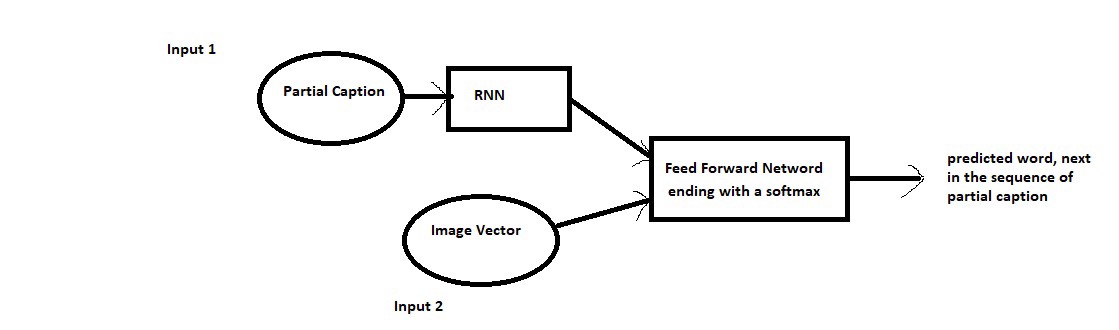
Since the input consists of two parts, an image vector and a partial caption, we cannot use the Sequential API provided by the Keras library. For this reason, we use the Functional API which allows us to create Merge Models. First, let’s look at the brief architecture which contains the high level sub-modules:

Figure 8: High level architecture

The model summary after creating the model in python is as follows (find the source code for reference) :

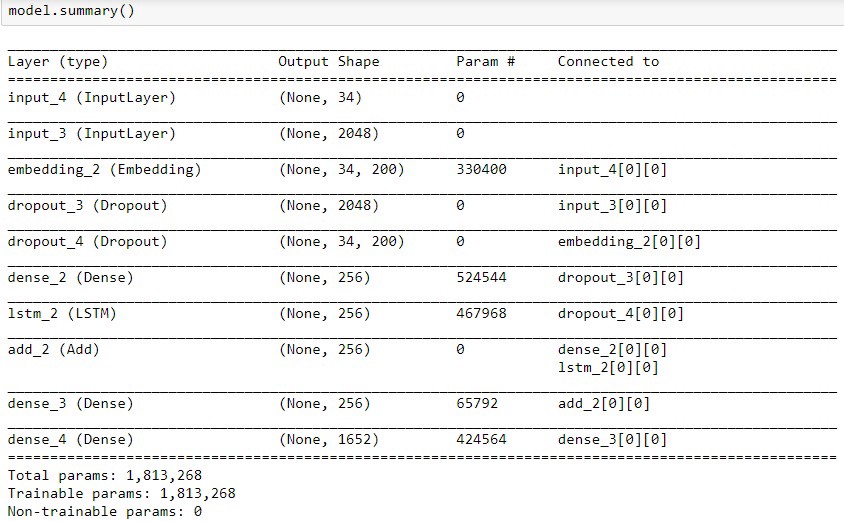


Figure 9: Summary of the parameters in the model

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

We had created an embedding matrix from a pre-trained Glove model which we need to include in the model before starting the training. Since we are using a pre-trained embedding layer, we need to **freeze**it before training the model, so that it does not get updated during the back propagation. 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

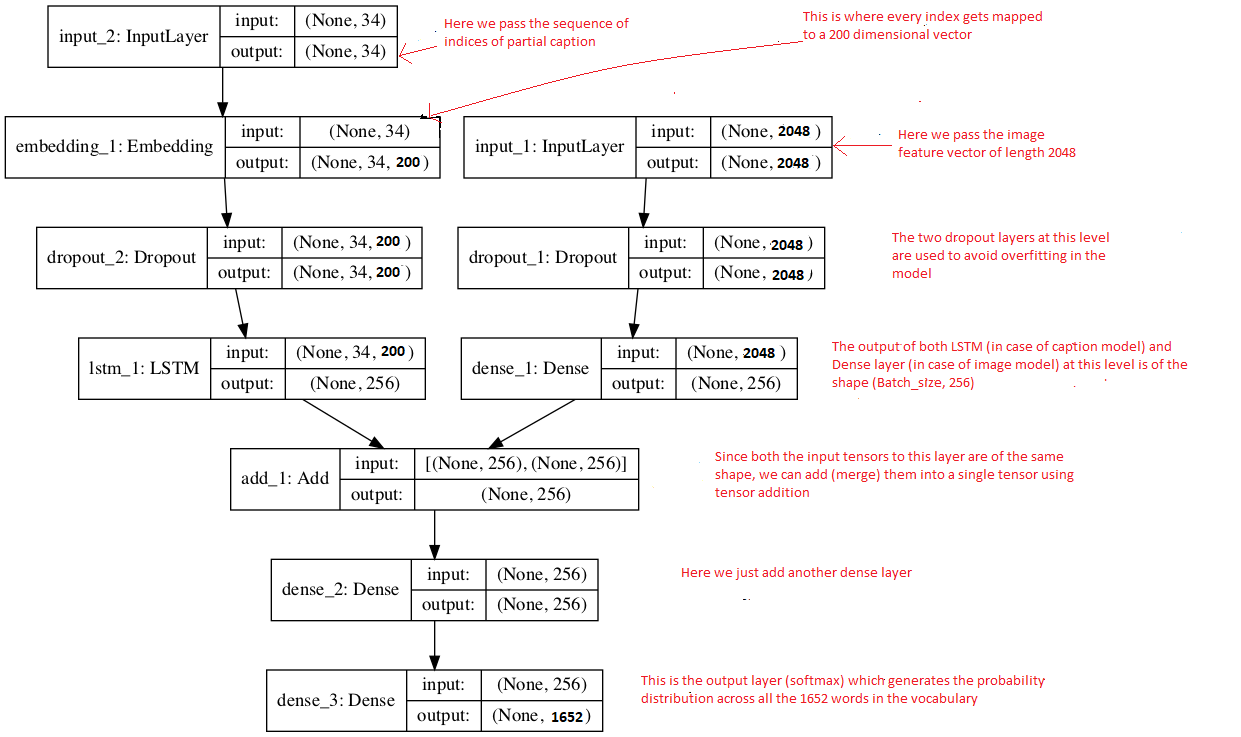


Figure 10: Flowchart of the architecture

## 5.7 Hyper parameters during training

Used Anaconda navigator to create a new environment for using GPU and the code was written and implemented in Jupyter notebook using python.

The model was trained for 20 epochs with the initial learning rate of 0.001 and 3 pictures per batch (batch size).

# 6 Source code

1. # In[1]:
2. **import** numpy as np
3. **from** numpy **import** array
4. **import** pandas as pd
5. **import** matplotlib.pyplot as plt
6. get\_ipython().run\_line\_magic('matplotlib', 'inline')
7. **import** string
8. **import** os
9. **from** PIL **import** Image
10. **import** glob
11. **from** pickle **import** dump, load
12. **from** time **import** time
13. **from** keras.preprocessing **import** sequence
14. **from** keras.models **import** Sequential
15. **from** keras.layers **import** LSTM, Embedding, TimeDistributed, Dense, RepeatVector, Activation, Flatten, Reshape, concatenate, Dropout, BatchNormalization
16. **from** keras.optimizers **import** Adam, RMSprop
17. **from** keras.layers.wrappers **import** Bidirectional
18. **from** keras.layers.merge **import** add
19. **from** keras.applications.inception\_v3 **import** InceptionV3
20. **from** keras.preprocessing **import** image
21. **from** keras.models **import** Model
22. **from** keras **import** Input, layers
23. **from** keras **import** optimizers
24. **from** keras.applications.inception\_v3 **import** preprocess\_input
25. **from** keras.preprocessing.text **import** Tokenizer
26. **from** keras.preprocessing.sequence **import** pad\_sequences
27. **from** keras.utils **import** to\_categorical

30. # In[2]:

33. # load doc into memory
34. **def** load\_doc(filename):
35. # open the file as read only
36. file = open(filename, 'r')
37. # read all text
38. text = file.read()
39. # close the file
40. file.close()
41. **return** text
43. filename = 'Flickr8k\_text/Flickr8k.token.txt'
44. # load descriptions
45. doc = load\_doc(filename)
46. **print**(doc[:300])

49. # In[3]:

52. **def** load\_descriptions(doc):
53. mapping = dict()
54. # process lines
55. **for** line **in** doc.split('\n'):
56. # split line by white space
57. tokens = line.split()
58. **if** len(line) < 2:
59. **continue**
60. # take the first token as the image id, the rest as the description
61. image\_id, image\_desc = tokens[0], tokens[1:]
62. # extract filename from image id
63. image\_id = image\_id.split('.')[0]
64. # convert description tokens back to string
65. image\_desc = ' '.join(image\_desc)
66. # create the list if needed
67. **if** image\_id **not** **in** mapping:
68. mapping[image\_id] = list()
69. # store description
70. mapping[image\_id].append(image\_desc)
71. **return** mapping
73. # parse descriptions
74. descriptions = load\_descriptions(doc)
75. **print**('Loaded: %d ' % len(descriptions))

78. # In[4]:

81. list(descriptions.keys())[:5]

84. # In[5]:

87. descriptions['1000268201\_693b08cb0e']

90. # In[6]:

93. **def** clean\_descriptions(descriptions):
94. # prepare translation table for removing punctuation
95. table = str.maketrans('', '', string.punctuation)
96. **for** key, desc\_list **in** descriptions.items():
97. **for** i **in** range(len(desc\_list)):
98. desc = desc\_list[i]
99. # tokenize
100. desc = desc.split()
101. # convert to lower case
102. desc = [word.lower() **for** word **in** desc]
103. # remove punctuation from each token
104. desc = [w.translate(table) **for** w **in** desc]
105. # remove hanging 's' and 'a'
106. desc = [word **for** word **in** desc **if** len(word)>1]
107. # remove tokens with numbers in them
108. desc = [word **for** word **in** desc **if** word.isalpha()]
109. # store as string
110. desc\_list[i] =  ' '.join(desc)
112. # clean descriptions
113. clean\_descriptions(descriptions)

116. # In[7]:

119. descriptions['1000268201\_693b08cb0e']

122. # In[8]:

125. # convert the loaded descriptions into a vocabulary of words
126. **def** to\_vocabulary(descriptions):
127. # build a list of all description strings
128. all\_desc = set()
129. **for** key **in** descriptions.keys():
130. [all\_desc.update(d.split()) **for** d **in** descriptions[key]]
131. **return** all\_desc
133. # summarize vocabulary
134. vocabulary = to\_vocabulary(descriptions)
135. **print**('Original Vocabulary Size: %d' % len(vocabulary))

138. # In[9]:

141. # save descriptions to file, one per line
142. **def** save\_descriptions(descriptions, filename):
143. lines = list()
144. **for** key, desc\_list **in** descriptions.items():
145. **for** desc **in** desc\_list:
146. lines.append(key + ' ' + desc)
147. data = '\n'.join(lines)
148. file = open(filename, 'w')
149. file.write(data)
150. file.close()
152. save\_descriptions(descriptions, 'descriptions.txt')

155. # In[10]:

158. # load a pre-defined list of photo identifiers
159. **def** load\_set(filename):
160. doc = load\_doc(filename)
161. dataset = list()
162. # process line by line
163. **for** line **in** doc.split('\n'):
164. # skip empty lines
165. **if** len(line) < 1:
166. **continue**
167. # get the image identifier
168. identifier = line.split('.')[0]
169. dataset.append(identifier)
170. **return** set(dataset)
172. # load training dataset (6K)
173. filename = 'Flickr8k\_text/Flickr\_8k.trainImages.txt'
174. train = load\_set(filename)
175. **print**('Dataset: %d' % len(train))

178. # In[11]:

181. # Below path contains all the images
182. images = 'Flicker8k\_Dataset/'
183. # Create a list of all image names in the directory
184. img = glob.glob(images + '\*.jpg')

187. # In[12]:

190. # Below file conatains the names of images to be used in train data
191. train\_images\_file = 'Flickr8k\_text/Flickr\_8k.trainImages.txt'
192. # Read the train image names in a set
193. train\_images = set(open(train\_images\_file, 'r').read().strip().split('\n'))
195. # Create a list of all the training images with their full path names
196. train\_img = []
198. **for** i **in** img: # img is list of full path names of all images
199. **if** i[len(images):] **in** train\_images: # Check if the image belongs to training set
200. train\_img.append(i) # Add it to the list of train images

203. # In[13]:

206. # Below file conatains the names of images to be used in test data
207. test\_images\_file = 'Flickr8k\_text/Flickr\_8k.testImages.txt'
208. # Read the validation image names in a set# Read the test image names in a set
209. test\_images = set(open(test\_images\_file, 'r').read().strip().split('\n'))
211. # Create a list of all the test images with their full path names
212. test\_img = []
214. **for** i **in** img: # img is list of full path names of all images
215. **if** i[len(images):] **in** test\_images: # Check if the image belongs to test set
216. test\_img.append(i) # Add it to the list of test images

219. # In[14]:

222. # load clean descriptions into memory
223. **def** load\_clean\_descriptions(filename, dataset):
224. # load document
225. doc = load\_doc(filename)
226. descriptions = dict()
227. **for** line **in** doc.split('\n'):
228. # split line by white space
229. tokens = line.split()
230. # split id from description
231. image\_id, image\_desc = tokens[0], tokens[1:]
232. # skip images not in the set
233. **if** image\_id **in** dataset:
234. # create list
235. **if** image\_id **not** **in** descriptions:
236. descriptions[image\_id] = list()
237. # wrap description in tokens
238. desc = 'startseq ' + ' '.join(image\_desc) + ' endseq'
239. # store
240. descriptions[image\_id].append(desc)
241. **return** descriptions
243. # descriptions
244. train\_descriptions = load\_clean\_descriptions('descriptions.txt', train)
245. **print**('Descriptions: train=%d' % len(train\_descriptions))

248. # In[15]:

251. **def** preprocess(image\_path):
252. # Convert all the images to size 299x299 as expected by the inception v3 model
253. img = image.load\_img(image\_path, target\_size=(299, 299))
254. # Convert PIL image to numpy array of 3-dimensions
255. x = image.img\_to\_array(img)
256. # Add one more dimension
257. x = np.expand\_dims(x, axis=0)
258. # preprocess the images using preprocess\_input() from inception module
259. x = preprocess\_input(x)
260. **return** x

263. # In[16]:

266. # Load the inception v3 model
267. model = InceptionV3(weights='imagenet')

270. # In[17]:

273. # Create a new model, by removing the last layer (output layer) from the inception v3
274. model\_new = Model(model.input, model.layers[-2].output)

277. # In[20]:

280. # Function to encode a given image into a vector of size (2048, )
281. **def** encode(image):
282. image = preprocess(image) # preprocess the image
283. fea\_vec = model\_new.predict(image) # Get the encoding vector for the image
284. fea\_vec = np.reshape(fea\_vec, fea\_vec.shape[1]) # reshape from (1, 2048) to (2048, )
285. **return** fea\_vec
287. # In[ ]:

290. # Call the funtion to encode all the train images
291. # This will take a while on CPU - Execute this only once
292. start = time()
293. encoding\_train = {}
294. **for** img **in** train\_img:
295. encoding\_train[img[len(images):]] = encode(img)
296. **print**("Time taken in seconds =", time()-start)

299. # In[ ]:

302. # Save the bottleneck train features to disk
303. **import** pickle
304. with open("encoded\_train\_images.pkl", "wb") as encoded\_pickle:
305. pickle.dump(encoding\_train, encoded\_pickle)

308. # In[ ]:

311. # Call the funtion to encode all the test images - Execute this only once
312. start = time()
313. encoding\_test = {}
314. **for** img **in** test\_img:
315. encoding\_test[img[len(images):]] = encode(img)
316. **print**("Time taken in seconds =", time()-start)

319. # In[ ]:

322. # Save the bottleneck test features to disk
323. with open("encoded\_test\_images.pkl", "wb") as encoded\_pickle:
324. pickle.dump(encoding\_test, encoded\_pickle)

327. # In[ ]:

330. train\_features = load(open("encoded\_train\_images.pkl", "rb"))
331. **print**('Photos: train=%d' % len(train\_features))

334. # In[ ]:

337. # Create a list of all the training captions
338. all\_train\_captions = []
339. **for** key, val **in** train\_descriptions.items():
340. **for** cap **in** val:
341. all\_train\_captions.append(cap)
342. len(all\_train\_captions)

345. # In[ ]:

348. # Consider only words which occur at least 10 times in the corpus
349. word\_count\_threshold = 10
350. word\_counts = {}
351. nsents = 0
352. **for** sent **in** all\_train\_captions:
353. nsents += 1
354. **for** w **in** sent.split(' '):
355. word\_counts[w] = word\_counts.get(w, 0) + 1
357. vocab = [w **for** w **in** word\_counts **if** word\_counts[w] >= word\_count\_threshold]
358. **print**('preprocessed words %d -> %d' % (len(word\_counts), len(vocab)))

361. # In[ ]:

364. ixtoword = {}
365. wordtoix = {}
367. ix = 1
368. **for** w **in** vocab:
369. wordtoix[w] = ix
370. ixtoword[ix] = w
371. ix += 1

374. # In[ ]:

377. vocab\_size = len(ixtoword) + 1 # one for appended 0's
378. vocab\_size

381. # In[ ]:

384. # convert a dictionary of clean descriptions to a list of descriptions
385. **def** to\_lines(descriptions):
386. all\_desc = list()
387. **for** key **in** descriptions.keys():
388. [all\_desc.append(d) **for** d **in** descriptions[key]]
389. **return** all\_desc
391. # calculate the length of the description with the most words
392. **def** max\_length(descriptions):
393. lines = to\_lines(descriptions)
394. **return** max(len(d.split()) **for** d **in** lines)
396. # determine the maximum sequence length
397. max\_length = max\_length(train\_descriptions)
398. **print**('Description Length: %d' % max\_length)

401. # In[ ]:

404. # data generator, intended to be used in a call to model.fit\_generator()
405. **def** data\_generator(descriptions, photos, wordtoix, max\_length, num\_photos\_per\_batch):
406. X1, X2, y = list(), list(), list()
407. n=0
408. # loop for ever over images
409. **while** 1:
410. **for** key, desc\_list **in** descriptions.items():
411. n+=1
412. # retrieve the photo feature
413. photo = photos[key+'.jpg']
414. **for** desc **in** desc\_list:
415. # encode the sequence
416. seq = [wordtoix[word] **for** word **in** desc.split(' ') **if** word **in** wordtoix]
417. # split one sequence into multiple X, y pairs
418. **for** i **in** range(1, len(seq)):
419. # split into input and output pair
420. in\_seq, out\_seq = seq[:i], seq[i]
421. # pad input sequence
422. in\_seq = pad\_sequences([in\_seq], maxlen=max\_length)[0]
423. # encode output sequence
424. out\_seq = to\_categorical([out\_seq], num\_classes=vocab\_size)[0]
425. # store
426. X1.append(photo)
427. X2.append(in\_seq)
428. y.append(out\_seq)
429. # yield the batch data
430. **if** n==num\_photos\_per\_batch:
431. **yield** [[array(X1), array(X2)], array(y)]
432. X1, X2, y = list(), list(), list()
433. n=0

436. # In[ ]:

439. # Load Glove vectors
440. glove\_dir = 'glove'
441. embeddings\_index = {} # empty dictionary
442. f = open(os.path.join(glove\_dir, 'glove.6B.200d.txt'), encoding="utf-8")
444. **for** line **in** f:
445. values = line.split()
446. word = values[0]
447. coefs = np.asarray(values[1:], dtype='float32')
448. embeddings\_index[word] = coefs
449. f.close()
450. **print**('Found %s word vectors.' % len(embeddings\_index))

453. # In[ ]:

456. embedding\_dim = 200
458. # Get 200-dim dense vector for each of the 10000 words in out vocabulary
459. embedding\_matrix = np.zeros((vocab\_size, embedding\_dim))
461. **for** word, i **in** wordtoix.items():
462. #if i < max\_words:
463. embedding\_vector = embeddings\_index.get(word)
464. **if** embedding\_vector **is** **not** None:
465. # Words not found in the embedding index will be all zeros
466. embedding\_matrix[i] = embedding\_vector

469. # In[ ]:

472. embedding\_matrix.shape

475. # ## Creating the model
477. # In[143]:

480. inputs1 = Input(shape=(2048,))
481. fe1 = Dropout(0.5)(inputs1)
482. fe2 = Dense(256, activation='relu')(fe1)
483. inputs2 = Input(shape=(max\_length,))
484. se1 = Embedding(vocab\_size, embedding\_dim, mask\_zero=True)(inputs2)
485. se2 = Dropout(0.5)(se1)
486. se3 = LSTM(256)(se2)
487. decoder1 = add([fe2, se3])
488. decoder2 = Dense(256, activation='relu')(decoder1)
489. outputs = Dense(vocab\_size, activation='softmax')(decoder2)
490. model = Model(inputs=[inputs1, inputs2], outputs=outputs)

493. # In[115]:

496. model.summary()

499. # In[112]:

502. model.layers[2]

505. # In[38]:

508. model.layers[2].set\_weights([embedding\_matrix])
509. model.layers[2].trainable = False
510. model.compile(loss='categorical\_crossentropy', optimizer='adam')
512. # In[40]:

515. epochs = 10
516. number\_pics\_per\_bath = 3
517. steps = len(train\_descriptions)//number\_pics\_per\_bath

520. # In[ ]:

523. **for** i **in** range(epochs):
524. generator = data\_generator(train\_descriptions, train\_features, wordtoix, max\_lengt h, number\_pics\_per\_bath)
525. model.fit\_generator(generator, epochs=1, steps\_per\_epoch=steps, verbose=1)
526. model.save('model\_weights/model\_' + str(i) + '.h5')

529. # In[ ]:

532. **for** i **in** range(epochs):
533. generator = data\_generator(train\_descriptions, train\_features, wordtoix, max\_length, number\_pics\_per\_bath)
534. model.fit\_generator(generator, epochs=1, steps\_per\_epoch=steps, verbose=1)
535. model.save('model\_weights/model\_' + str(i) + '.h5')

538. # ## Trying on test set :
540. # In[145]:

543. model.load\_weights('model\_weights/model\_9.h5')

546. # In[146]:

549. images = 'Flicker8k\_Dataset/'

552. # In[147]:

555. with open("encoded\_test\_images.pkl", "rb") as encoded\_pickle:
556. encoding\_test = load(encoded\_pickle)

559. # In[148]:

562. **def** greedySearch(photo):
563. in\_text = 'startseq'
564. **for** i **in** range(max\_length):
565. sequence = [wordtoix[w] **for** w **in** in\_text.split() **if** w **in** wordtoix]
566. sequence = pad\_sequences([sequence], maxlen=max\_length)
567. yhat = model.predict([photo,sequence], verbose=0)
568. yhat = np.argmax(yhat)
569. word = ixtoword[yhat]
570. in\_text += ' ' + word
571. **if** word == 'endseq':
572. **break**
573. final = in\_text.split()
574. final = final[1:-1]
575. final = ' '.join(final)
576. **return** final

579. # In[149]:

582. z=100
583. **while** z<=150:
584. pic = list(encoding\_test.keys())[z]
585. image = encoding\_test[pic].reshape((1,2048))
586. x=plt.imread(images+pic)
587. plt.imshow(x)
588. plt.show()
589. **print**("Greedy:",greedySearch(image))
590. z=z+1

593. # ## Trying on a custom Image :
595. # In[46]:

598. **from** keras.preprocessing **import** image
599. **from** PIL **import** Image
600. **from** keras.models **import** load\_model
601. **from** keras.applications.inception\_v3 **import** InceptionV3
602. **from** keras.models **import** Model
603. **import** numpy as np
604. **from** keras.applications.inception\_v3 **import** preprocess\_input
605. # Below path contains all the images
606. #images = 'Flicker8k\_Dataset/'
607. # Create a list of all image names in the directory
608. #img = glob.glob(images + '\*.jpg')
609. **def** preprocess(image\_path):
610. # Convert all the images to size 299x299 as expected by the inception v3 model
611. img = image.load\_img(image\_path, target\_size=(299, 299))
612. # Convert PIL image to numpy array of 3-dimensions
613. x = image.img\_to\_array(img)
614. # Add one more dimension
615. x = np.expand\_dims(x, axis=0)
616. # preprocess the images using preprocess\_input() from inception module
617. x = preprocess\_input(x)
618. **return** x
619. incep = InceptionV3(weights='imagenet')
620. incep\_new = Model(incep.input, incep.layers[-2].output)
621. **def** encode(image):
622. image = preprocess(image) # preprocess the image
623. fea\_vec = incep\_new.predict(image) # Get the encoding vector for the image
624. fea\_vec = np.reshape(fea\_vec, fea\_vec.shape[1]) # reshape from (1, 2048) to (2048, )
625. **return** fea\_vec

628. # In[47]:

631. a='images/club.jpg'
632. b=encode(a)

635. # In[48]:

638. model=load\_model('model\_weights/model\_9.h5')
639. **def** greedySearch(photo):
640. in\_text = 'startseq'
641. **for** i **in** range(max\_length):
642. sequence = [wordtoix[w] **for** w **in** in\_text.split() **if** w **in** wordtoix]
643. sequence = pad\_sequences([sequence], maxlen=max\_length)
644. yhat = model.predict([photo,sequence], verbose=0)
645. yhat = np.argmax(yhat)
646. word = ixtoword[yhat]
647. in\_text += ' ' + word
648. **if** word == 'endseq':
649. **break**
650. final = in\_text.split()
651. final = final[1:-1]
652. final = ' '.join(final)
653. **return** final

656. # In[49]:

659. **import** matplotlib.pyplot as plt
660. get\_ipython().run\_line\_magic('matplotlib', 'inline')
661. pic = b
662. image = pic.reshape((1,2048))
663. x=plt.imread(a)
664. plt.imshow(x)
665. plt.show()
666. **print**("Greedy:",greedySearch(image))

*NOTE: Since the code is converted from an ‘.ipynb’ file to a ‘.py’ file, there can be several indentation errors and several other errors as well.*

# 7 Predicting captions

In this section we will understand how we test (infer) our model by passing in new images, i.e. how can we generate a caption for a new test image. Earlier we used two examples to understand data preparation, now let’s use a third image and try to understand how we would like the caption to be generated. The third image vector and caption are as follows:



Figure 11: Test image

Caption: the black cat is walking on grass

The vocabulary in the example was:

Vocabulary = {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”

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 greedilyselect the word with the maximum probability, given the feature vector and partial caption.

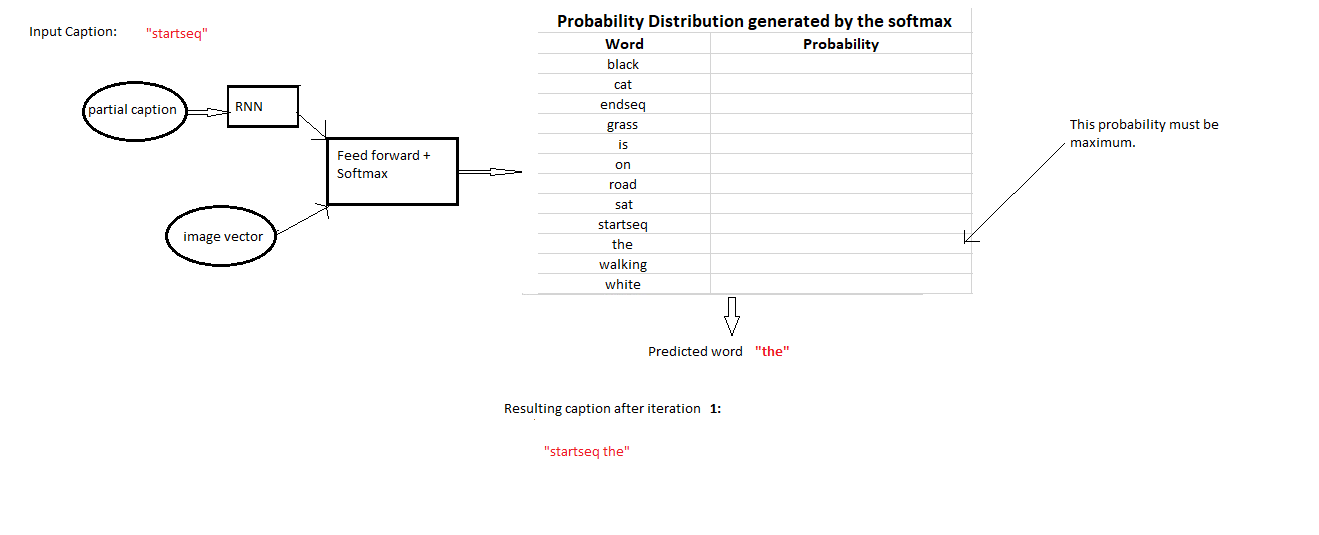
If the model is trained well, we must expect the probability for the word “the” to be maximum:

Figure 12: 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”

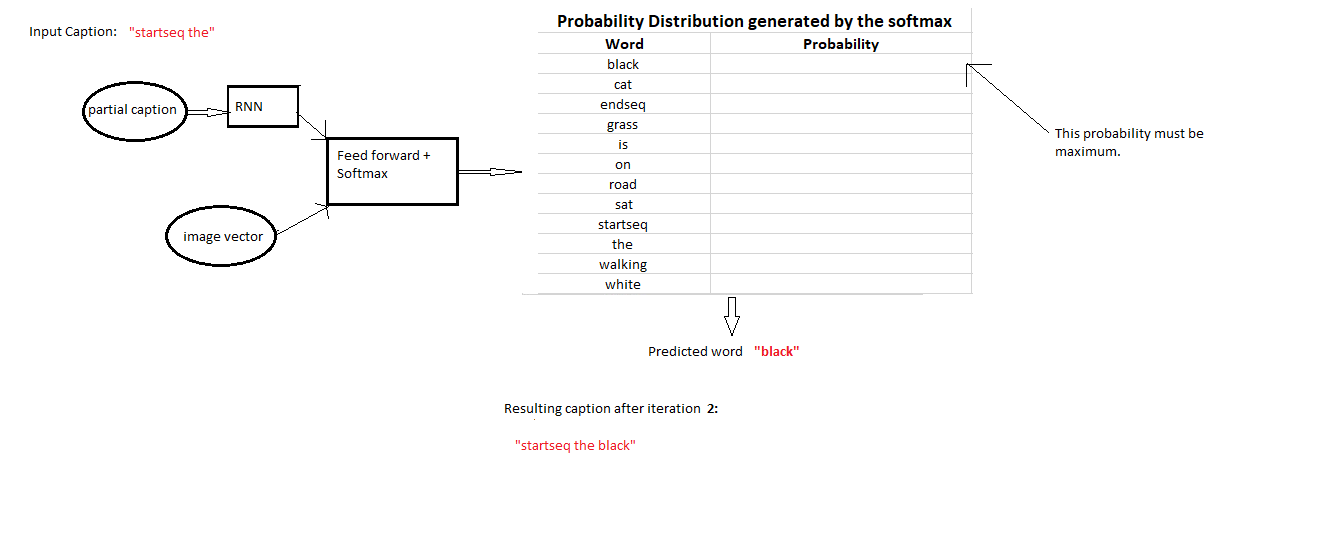
**Iteration 3:**

Figure 13: Iteration 2

Input: Image vector + “startseq the black”

Expected Output word: “cat”

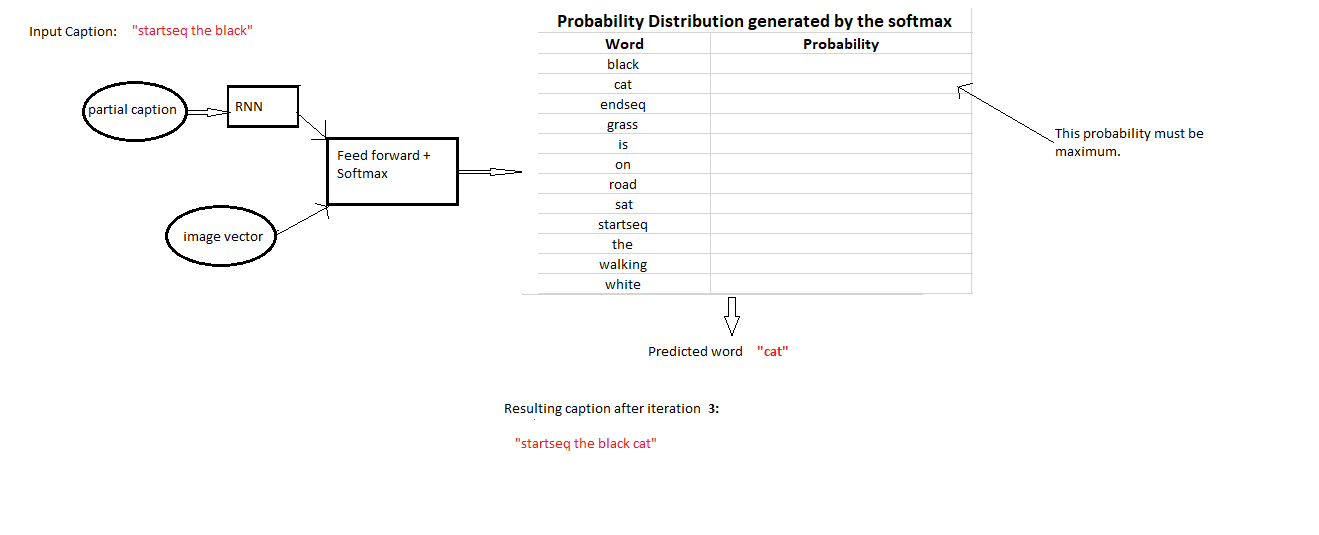


Figure 14: Iteration 3

**Iteration 4:**

Input: Image vector + “startseq the black cat”

Expected Output word: “is”

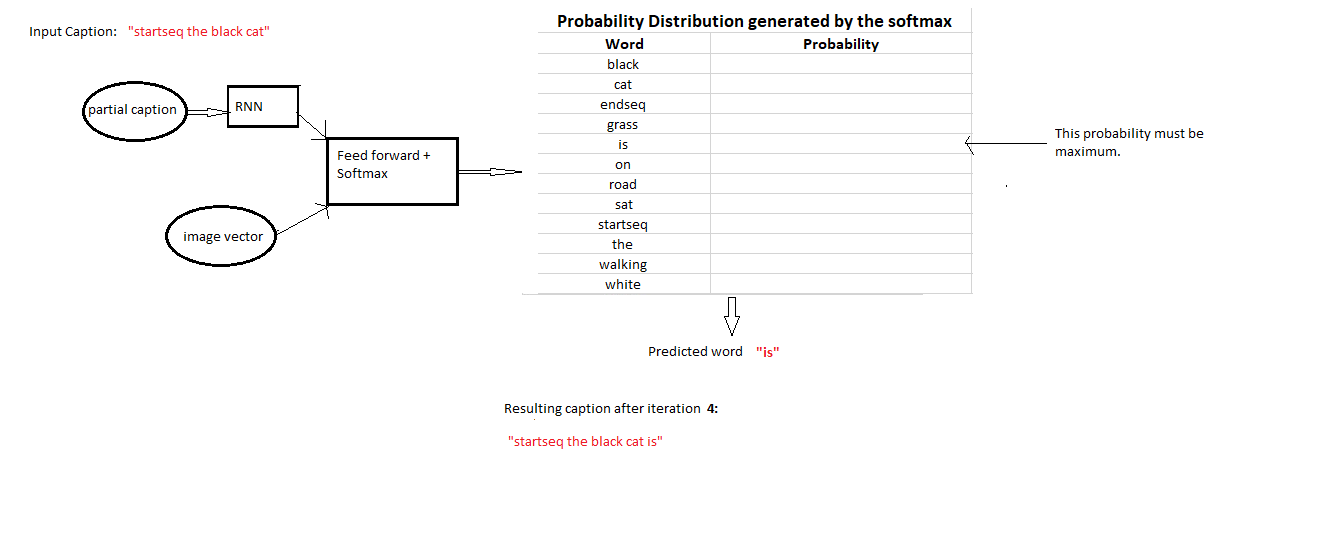


Figure 15: Iteration 4

**Iteration 5:**

Input: Image vector + “startseq the black cat is”

Expected Output word: “walking”

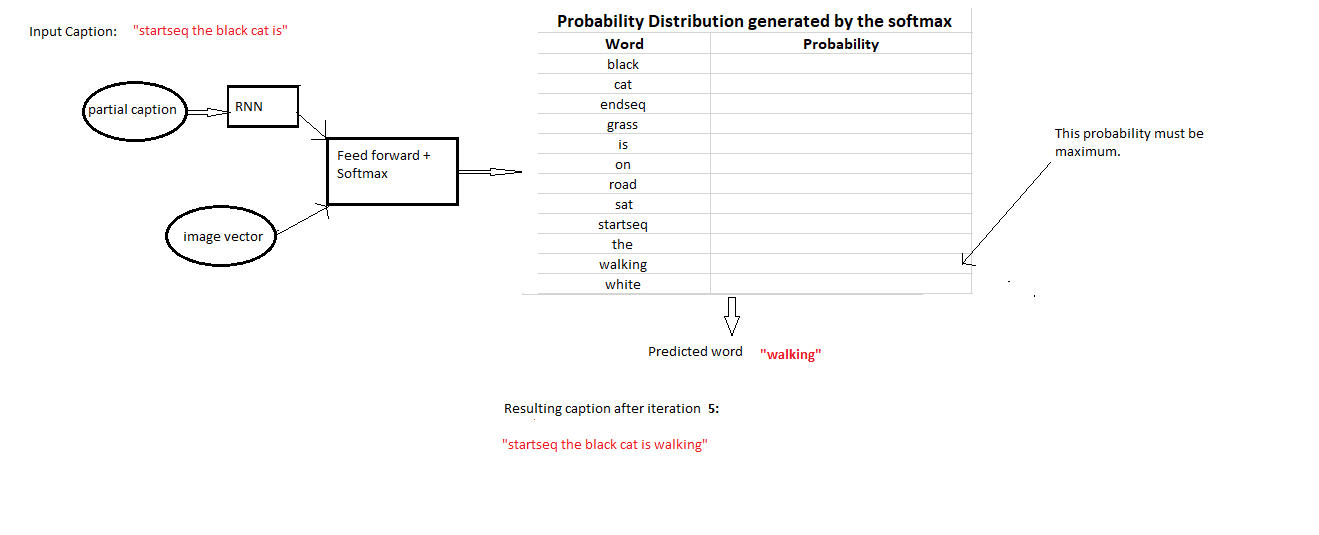


Figure 16: Iteration 5

**Iteration 6:**

Input: Image vector + “startseq the black cat is walking”

Expected Output word: “on”

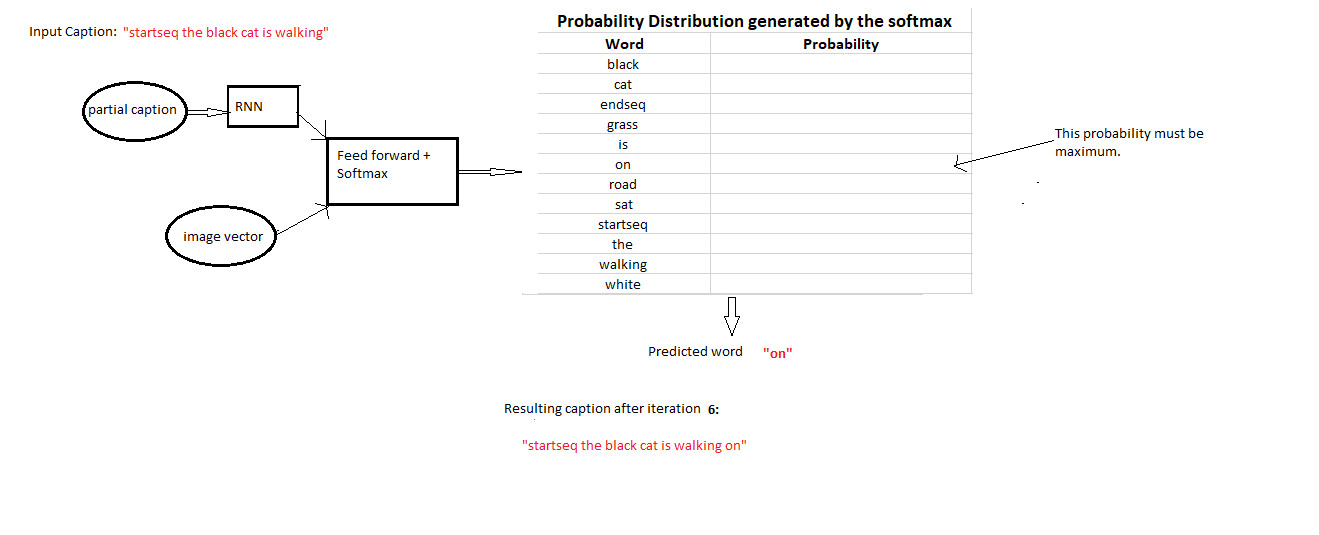


Figure 17: Iteration 6

**Iteration 7:**

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

Expected Output word: “grass”

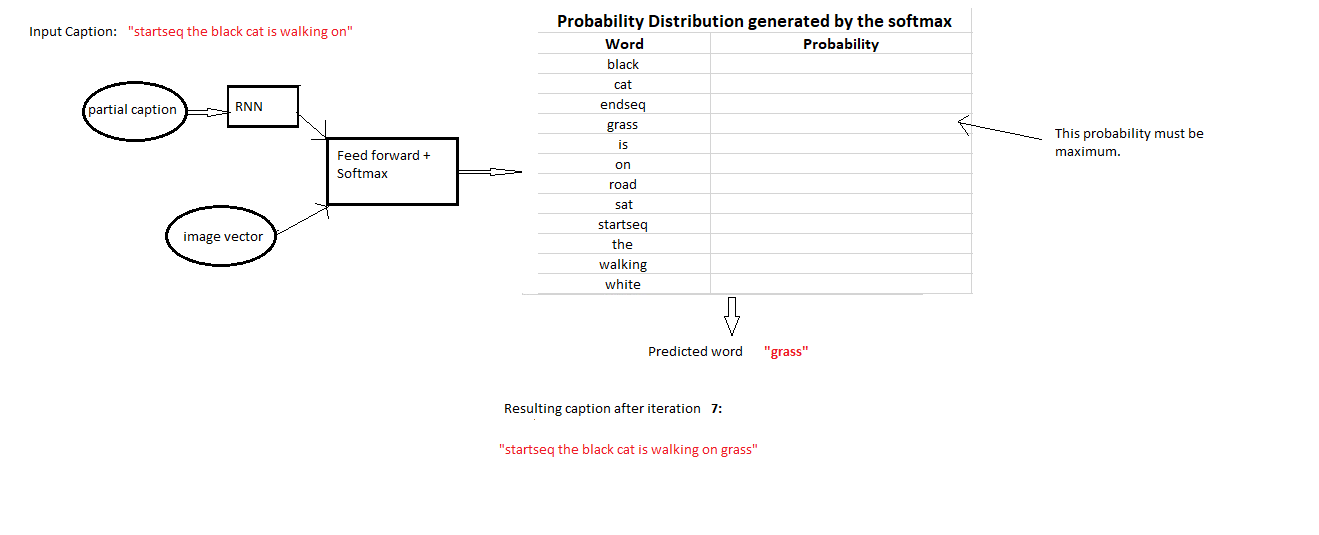


Figure 18: Iteration 7

**Iteration 8:**

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

Expected Output word: “endseq”

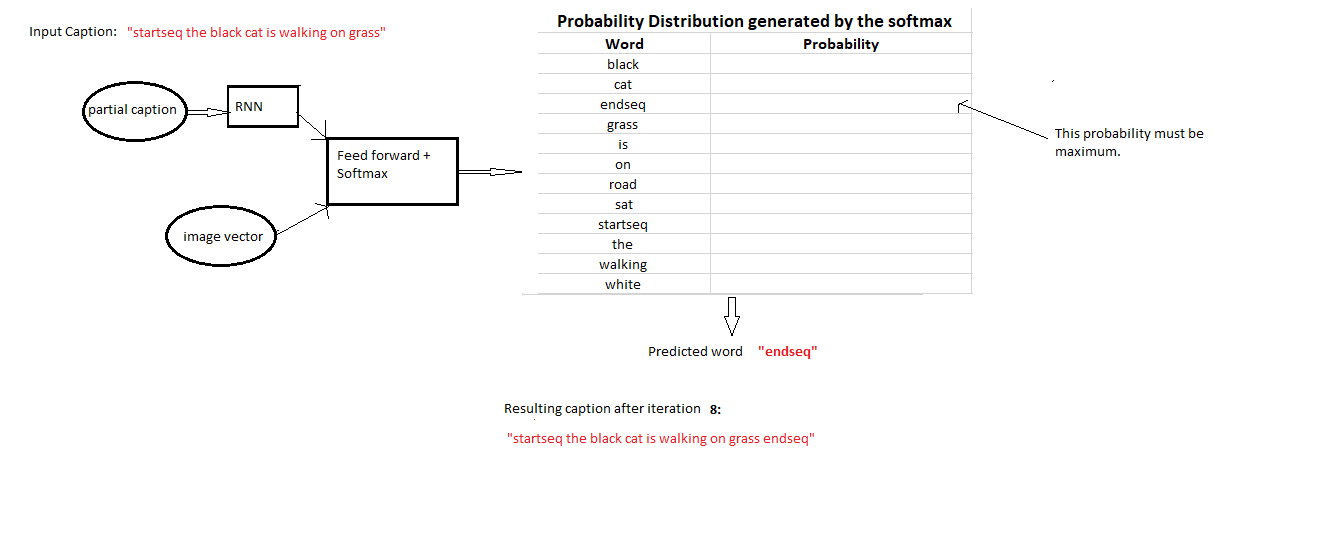


Figure 19: Iteration 8

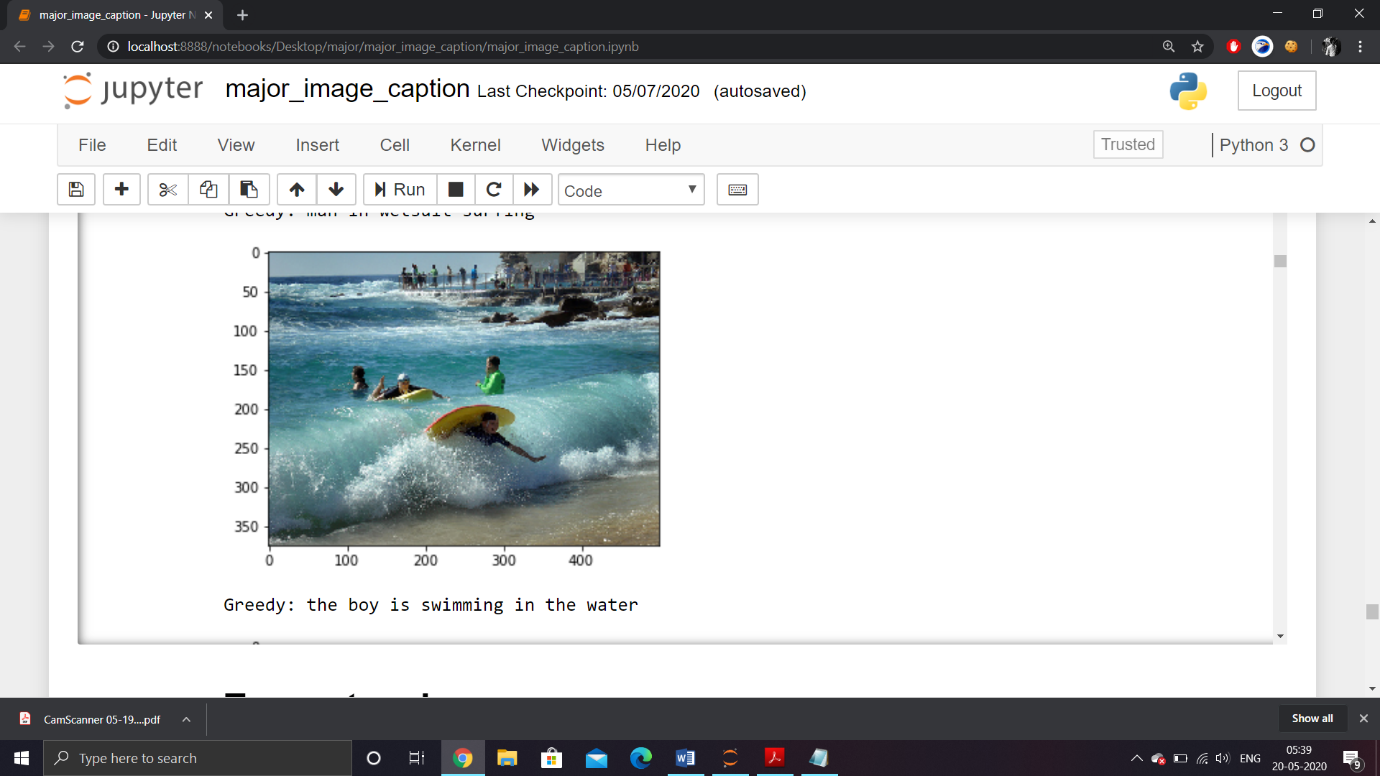
This is where we stopthe iterations. So we stop when either of the below two conditions is met:

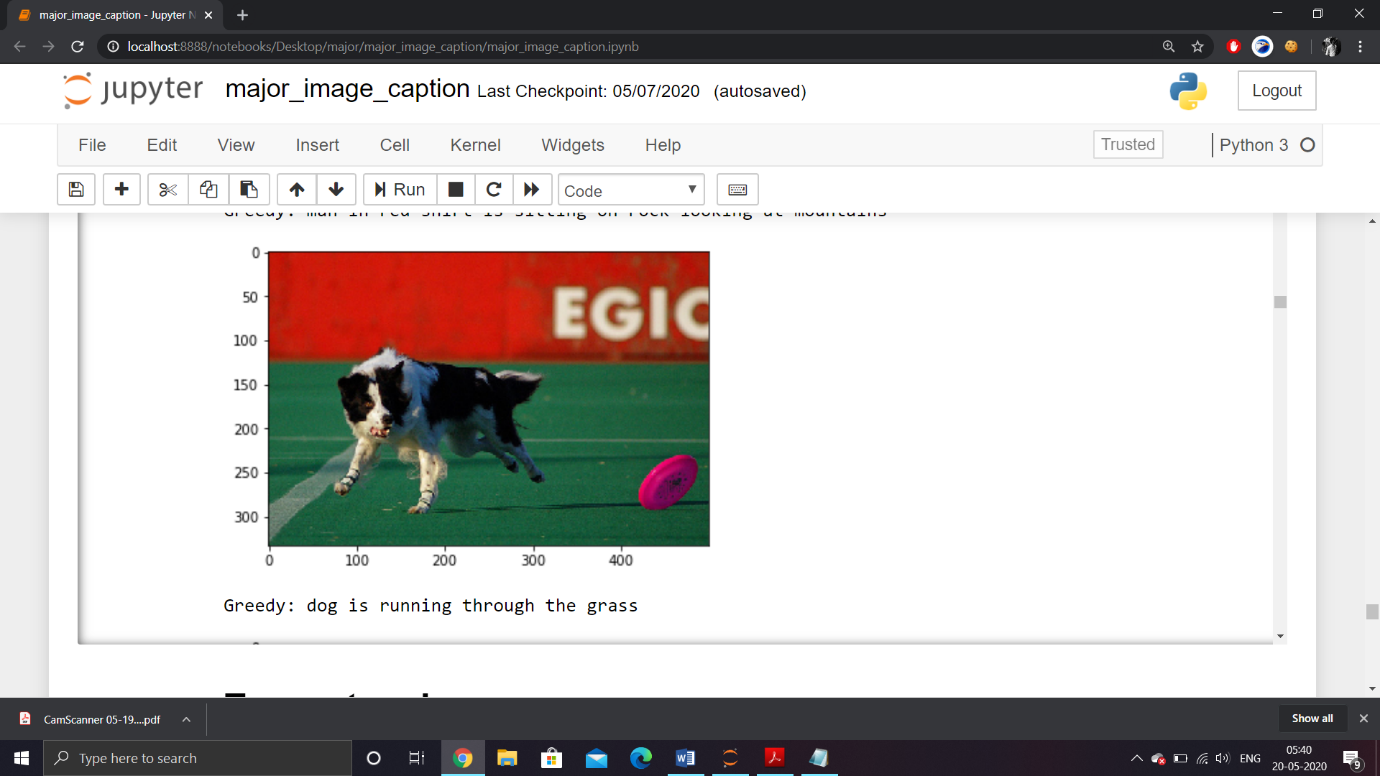
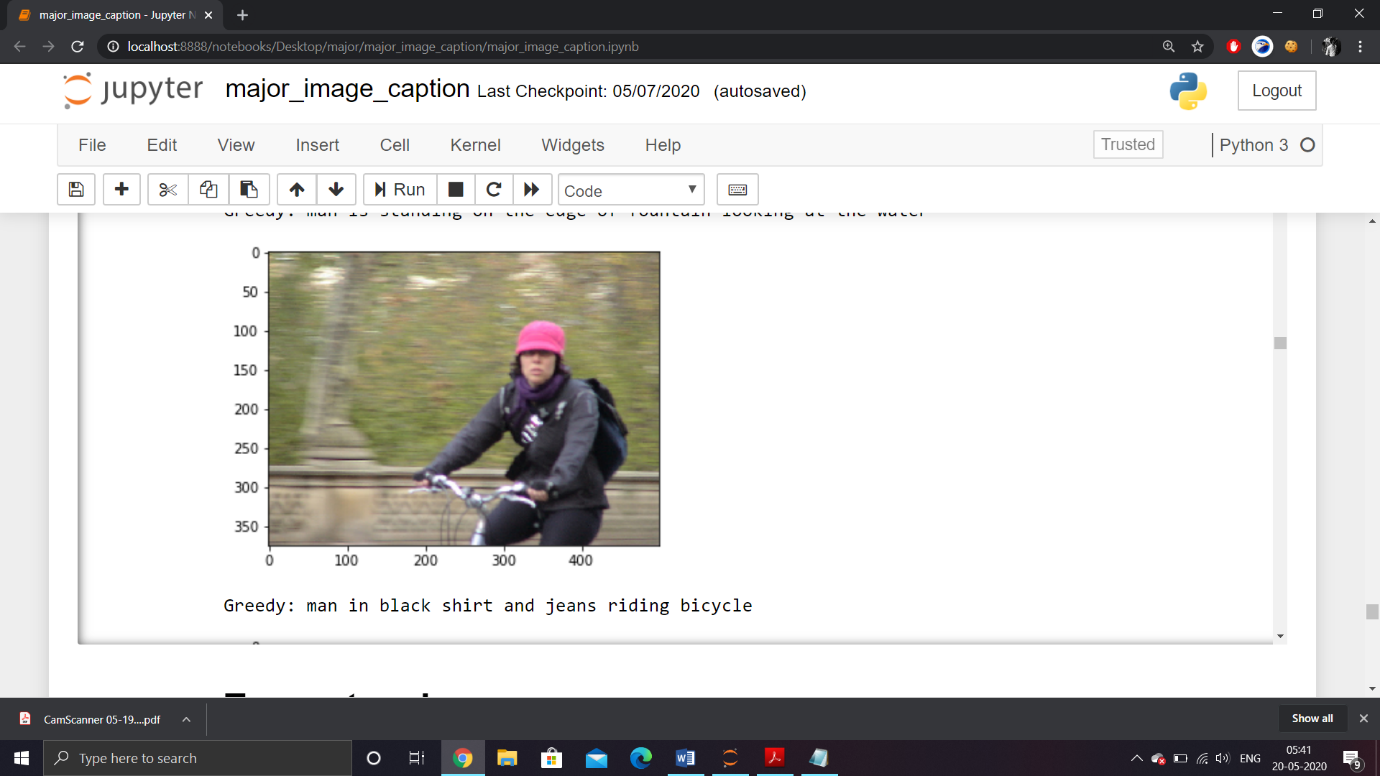
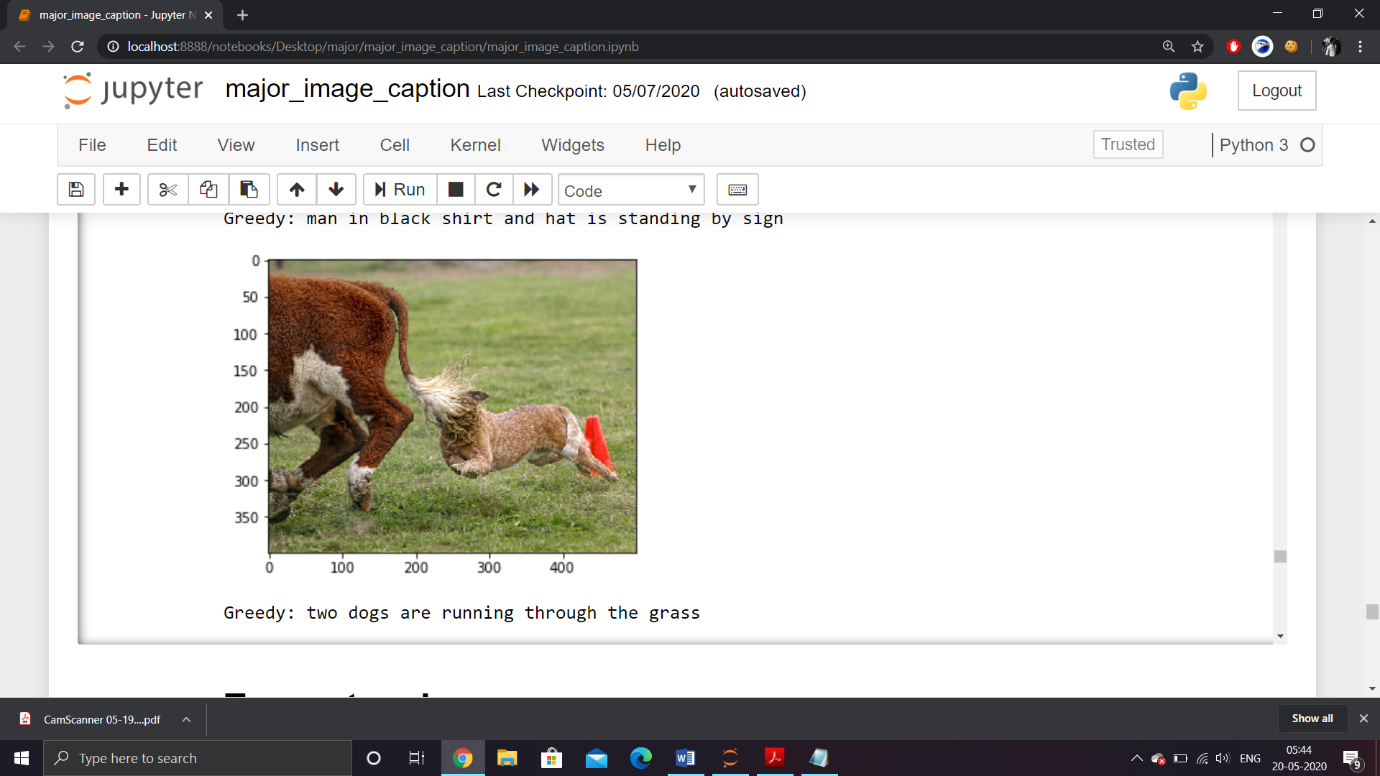
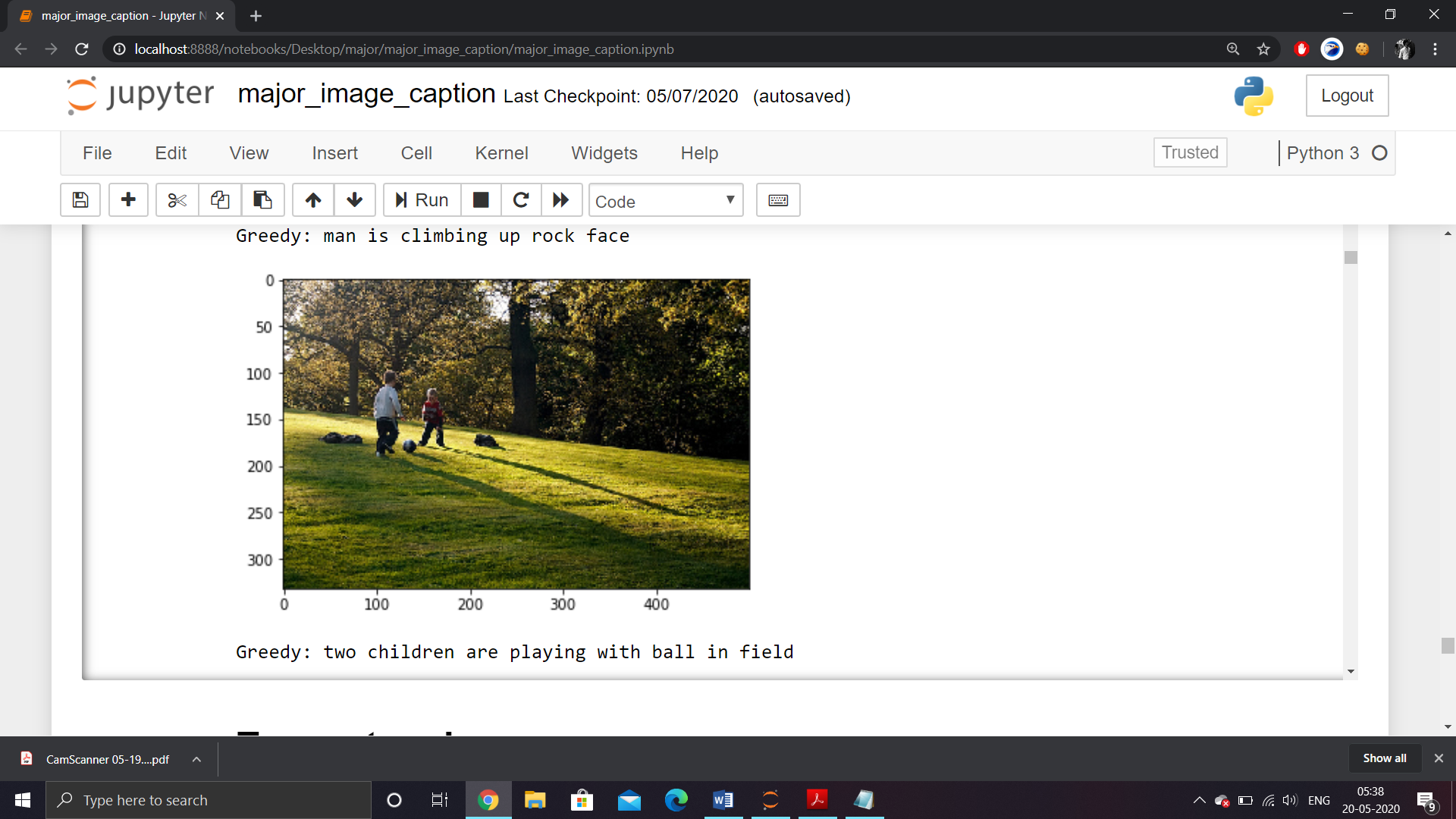
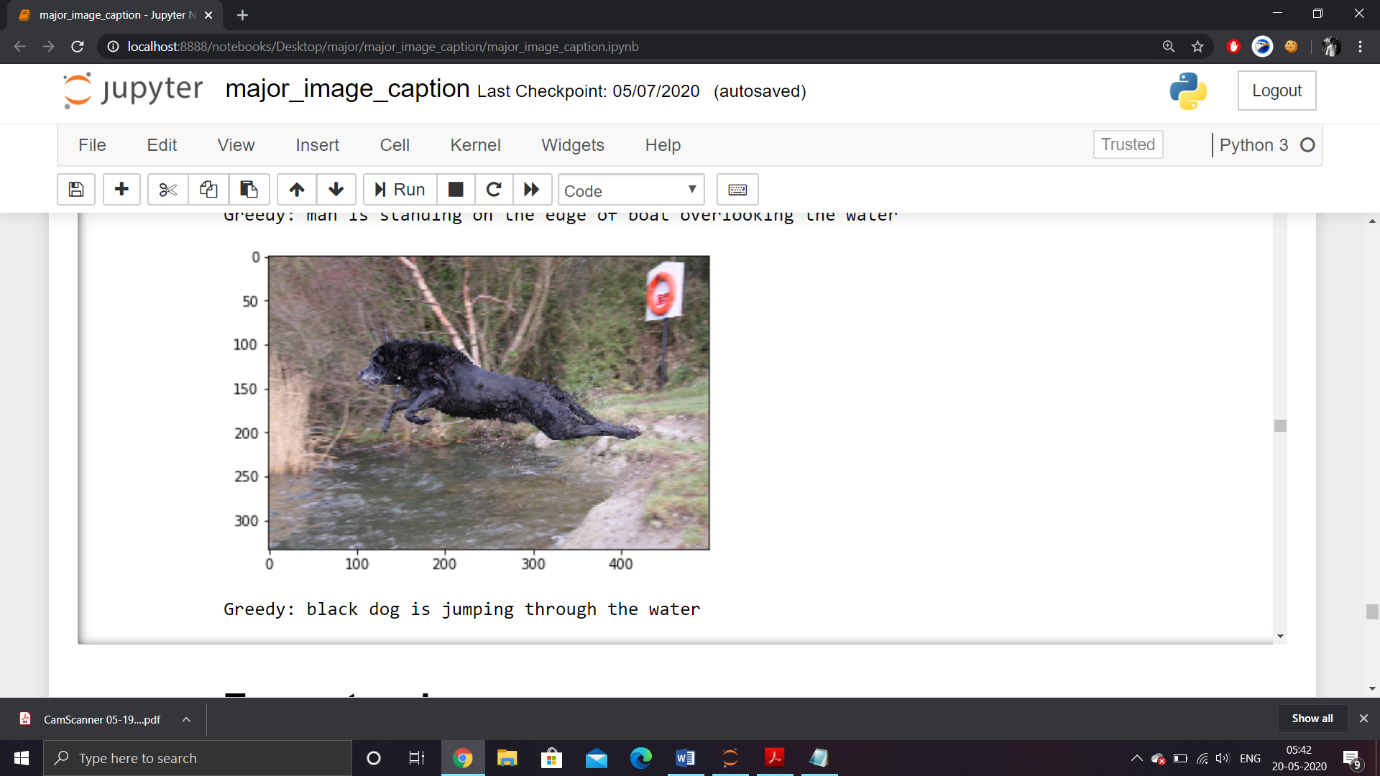
1. We encounter an ‘endseq’ token which means the model thinks that this is the end of the caption.
2. We reach a maximum thresholdof the number of words generated by the model.

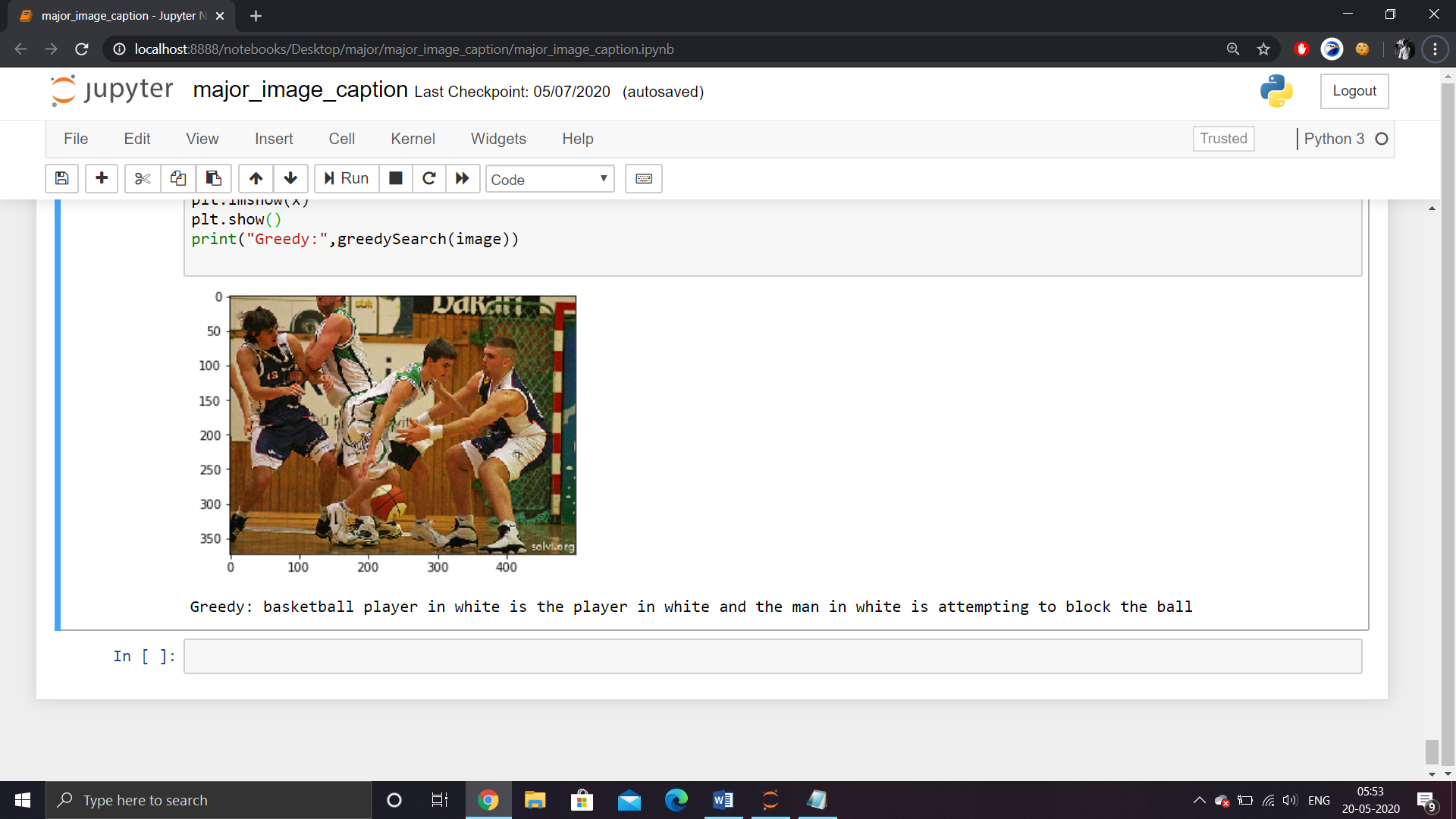
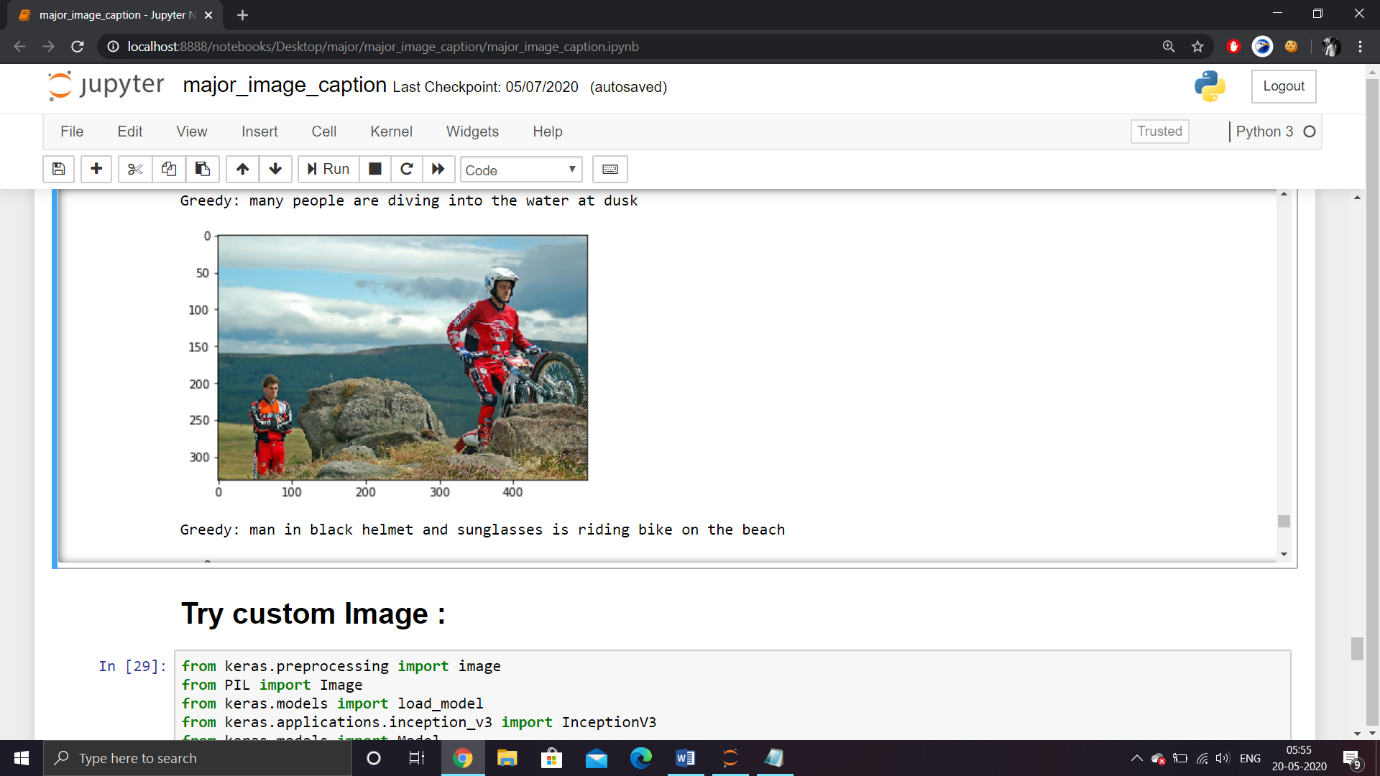
If any of the above conditions is met, we break the loop and report the generated caption as the output of the model for the given image.

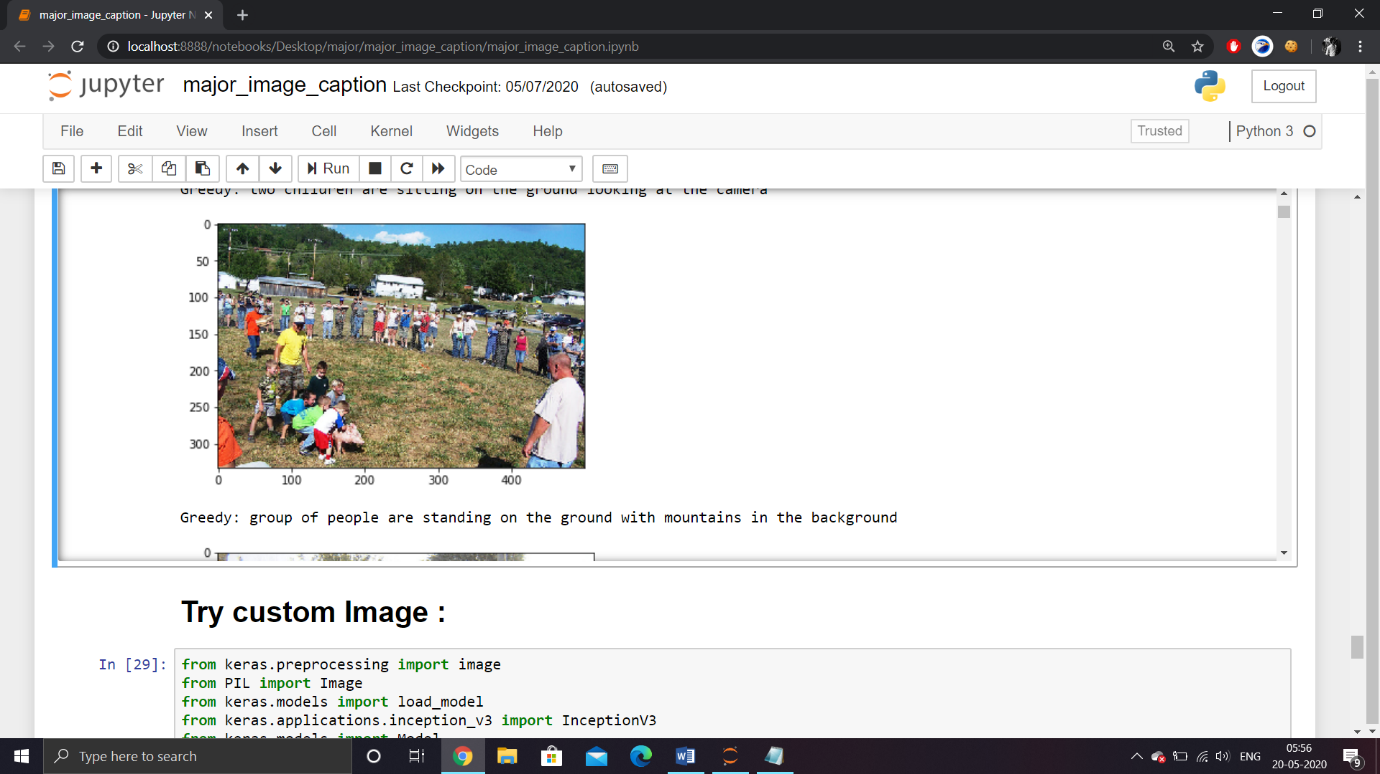
# 8 Results

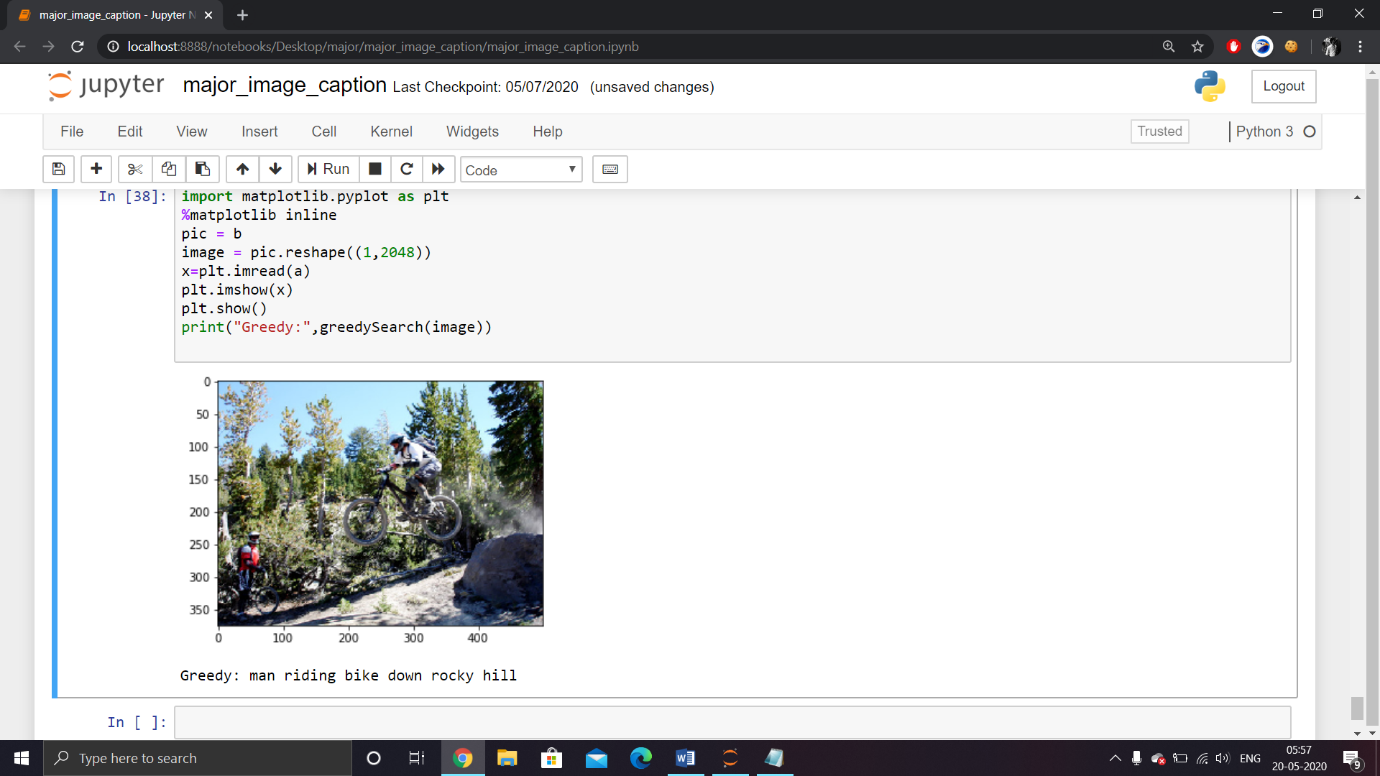
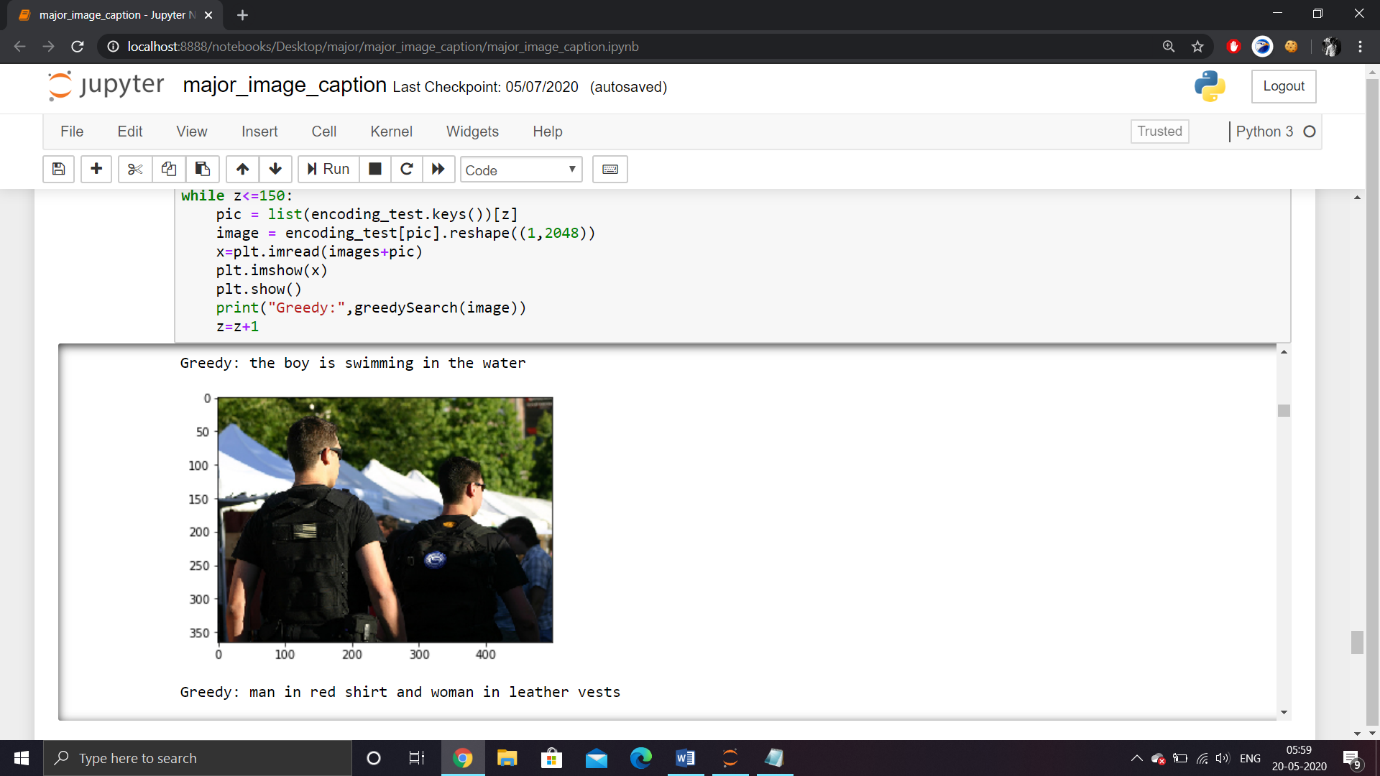
To understand how good the model is, we will generate captions on images from the test dataset and also few completely new images. Following are a few captions our model generated for the test images, note how the model is correctly identifying the colours and sometimes predicting totally wrong captions:











# Conclusion and future work

After examining the results we can say that without any rigorous hyper-parameter tuning our model does a decent job in generating captions for images. However all the captions generated were not always grammatically or semantically correct.

**Also, w**e 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.

This is a basic functioning image caption generating model and can be improved in many ways, some of which are:

* 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.

# References

[1] F. Rosenblatt, “The perceptron: A probabilistic model for information storage and organization in the brain,” Psychological Review, pp. 65–386, 1958

[2] M. Minsky and S. Papert, “Perceptrons : an introduction to computational geometry,” 1969.

[3] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, “Parallel distributed processing: Explorations in the microstructure of cognition, vol. 1,” ch. Learning Internal Representations by Error Propagation, pp. 318–362, Cambridge, MA, USA: MIT Press, 1986.

[4] S. Hochreiter and J. Schmidhuber, “Long short-term memory,” Neural Comput., vol. 9, pp. 1735–1780, Nov. 1997.

[5] Y. LeCun, L. Bottou, G. Orr, and K. Muller, “Efficient backprop,” in Neural Networks: Tricks of the trade (G. Orr and M. K., eds.), Springer, 1998.

[6] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, “Gradient-based learning applied to document recognition,” in Proceedings of the IEEE, pp. 2278–2324, 1998.

[7] X. Glorot, A. Bordes, and Y. Bengio, “Deep sparse rectifier neural networks,” in Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics (AISTATS-11) (G. J. Gordon and D. B. Dunson, eds.), vol. 15, pp. 315–323, Journal of Machine Learning Research - Workshop and Conference Proceedings, 2011.

[8] A. Krizhevsky, I. Sutskever, and G. E. Hinton, “Imagenet classification with deep convolutional neural networks.,” in NIPS (P. L. Bartlett, F. C. N. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, eds.), pp. 1106–1114, 2012.

[9] K. Simonyan and A. Zisserman, “Very deep convolutional networks for large-scale image recognition,” CoRR, vol. abs/1409.1556, 2014.

[10] R. Jozefowicz, W. Zaremba, and I. Sutskever, “An empirical exploration of recurrent network architectures,” Journal of Machine Learning Research, 2015.

[11] K. Greff, R. K. Srivastava, J. Koutn´ık, B. R. Steunebrink, and J. Schmidhuber, “LSTM: A search space odyssey,” CoRR, vol. abs/1503.04069, 2015.

[12] Anadi Chaman and K.V.Sameer Raja Proposal “Image Captioning using Visual Attention”, October 4, 2015

[13] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei, “ImageNet Large Scale Visual Recognition Challenge,” International Journal of Computer Vision (IJCV), vol. 115, no. 3, pp. 211–252, 2015.

[14] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, “Rethinking the inception architecture for computer vision,” CoRR, vol. abs/1512.00567, 2015.

[15] Oriol Vinyals, Alexander Toshev, Samy Bengio, Dumitru Erhan, “Show and Tell: A Neural Image Caption Generator” arXiv:1411.4555v2 [cs.CV] 20 Apr 2015

[16] A. Karpathy, J. Johnson, and F. Li, “Visualizing and understanding recurrent networks,” CoRR, vol. abs/1506.02078, 2015.

[17] I. Goodfellow, Y. Bengio, and A. Courville, “Deep learning.” Book in preparation for MIT Press, 2016.

[18] C. Szegedy, S. Ioffe, and V. Vanhoucke, “Inception-v4, inception-resnet and the impact of residual connections on learning,” CoRR, vol. abs/1602.07261, 2016.

[19] Marc Tanti Albert Gatt, Kenneth P. Camilleri, “What is the Role of Recurrent Neural Networks (RNNs) in an Image Caption Generator?”, 25 Aug 2017

[20] Marc Tanti Albert Gatt, Kenneth P. Camilleri, “Where to put the Image in an Image Caption Generator”, 12-Mar-2018

[21] Andrej Karpathy Li Fei-Fei, “Deep Visual-Semantic Alignments for Generating Image Descriptions”, Department of Computer Science, Stanford University