# AI IMAGE CAPTION BOT

**A Thesis Submitted**
**In Partial Fulfilment of the Requirements**
**For the Degree of**

## BACHELOR OF TECHNOLOGY

**In**
**Computer Science & Engineering**
**By**


**ANUP KUMAR**
**(17EUCCS304)**


**Under the Supervision of**
**DR. NIRMALA SHARMA**
**(Assistant Professor, Rajasthan Technical University)**


**To the**

**Faculty of Computer Science & Engineering**
**University Department, Rajasthan Technical University**
**(Rajasthan Technical University, Rawatbhata Road, Kota – 324010)**

**JUNE, 2021**

# CERTIFICATE

Certified that **Anup Kumar( 17EUCCS304)** has carried out their search work presented in this thesis entitled **"AI Image Caption Bot"** for the award of **Bachelor of Technology** from University department, Rajasthan Technical University, Kota, under my supervision. The thesis embodies results of original work, and studies are carried out by the student herself and the contents of the thesis do not form the basis for the award of any other degree to the candidate or to anybody else from this or any other University/Institution.

Signature

DR NIRMALA SHARMA

Assistant Professor (Computer Science & Engineering Department)

Rajasthan Technical University, Kota

Date: 05-07-2021

# ABSTRACT

Image captioning means automatically generating a caption for an image. As a recently emerged research area, it is attracting more and more attention. To achieve the goal of image captioning, semantic information of images needs to be captured and expressed in natural languages.

In Artificial Intelligence (AI), the contents of an image are generated automatically which involves computer vision and NLP (Natural Language Processing). The neural model which is regenerative, is created. It depends on computer vision and machine translation. This model is used to generate a natural sentence which eventually describes the image. This model consists of Convolutional Neural Network (CNN) as well as Recurrent Neural Network (RNN). The CNN is used for feature extraction from image and RNN is used for sentence generation. The model is trained in such a way that if input image is given to model it generates captions which nearly describes the image.

We have implemented a neural model which is consists of one encoder and one decoder. The encoder adopts ResNet50 based on the convolutional neural network, which creates an extensive representation of the given image by embedding it into a fixed length vector. The decoder is designed with LSTM, a recurrent neural network, to selectively focus the attention over certain parts of an image to predict the next sentence. And this model is trained and tested on Flickr8k datasets. Results of the implemented model are evaluated using a cosine similarity mechanism between the generated captions and the actual ground truth captions.

# ACKNOWLEDGEMENTS

I am extremely thankful to my project supervisor, Dr Nirmala Sharma (Assistant Professor, Department of Computer Science & Engineering, Rajasthan Technical University, Kota), for your patience, guidance, and support. I have benefited greatly from your wealth of knowledge and meticulous editing. I am extremely grateful that you took me on as a student and continued to have faith in me over the years.

I am also grateful to Prof. Vikas Tripathi (Assistant Professor, Department of Computer Science & Engineering, Rajasthan Technical University, Kota), for his valuable suggestions, ever encouraging and motivating guidance. He has provided me with valuable insights during the course of this project which have helped me to overcome several obstacles during the project.

Finally, last but not the least, I must express my very profound gratitude to my parents for providing me with unfailing support and continuous encouragement throughout my years of study and through the process of researching and writing this project. This accomplishment would not have been possible without them.

Thank you.

# CONTENTS


Page No.

**Title**

**Certificate** ii

**Abstract** iii

**Acknowledgements** iv

**List of Tables** ix

**List of Figures** x-xii

**CHAPTER 1: INTRODUCTION 1-4**

1.1. MOTIVATION 2

1.2. GENERAL 2

1.3. RESEARCH OBJECTIVES 4

**CHAPTER 2: LITERATURE REVIEW 5-33**

2.1. CNN (CONVOLUTIONAL NEURAL NETWORK) 5

2.1.1 What is CNN? 5

2.1.2. Why should we use CNN? 5

2.1.3 Image Representation 6

2.1.4 Edge Detection 7

2.1.5 Stride and Padding 8

2.1.6 Layers in CNN 11

2.1.6.1 Input Layer 12

2.1.6.2 Convo Layer 12

2.1.6.3. Pooling Layer 13

2.1.6.4. Fully Connected Layer (FC) 14

2.1.6.5. Softmax / Logistic Layer 14

2.1.6.6. Output Layer 14

2.1.7. Keras Implementation 14

2.2 RECURRENT NEURAL NETWORK (RNN) 15

2.2.1 Recurrent Neural Networks 17

2.2.2 Parameter Sharing 18

2.2.3 Deep RNNs 19

2.2.4 Bidirectional RNNs 20

2.2.5 Recursive Neural Networks 21

2.3 LSTMs 23

2.3.1 The Problem, Short-term Memory 24

2.3.2 LSTM’s and GRU’s as a solution 24

2.3.3 Intuition 25

2.3.4 Review of Recurrent Neural Networks 26

2.3.4.1 Tanh activation 28

2.3.5 LSTM 29

2.3.5.1 Core Concept 30

2.3.5.2 Sigmoid 30

2.3.5.3 Forget gate 30

2.3.5.4 Input Gate 31

2.3.5.5 Cell State 32

2.3.5.6 Output Gate 33

**CHAPTER 3: METHODOLOGY 34-37**

3.1 MODEL OVERVIEW 34

3.2 DATASET 35

3.3 .IMAGE FEATURES DETECTION 36

3.4. TEXT GENERATION USING LSTM 36

**CHAPTER 4: IMPLEMENTATION 38-68**

4.1 TYPES OF ARCHITECTURES 38

4.2 GOOGLE'S ARCHITECTURE 43

4.3 MICROSOFT'S ARCHITECTURE 44

4.4 A PRACTICAL IMPLEMENTATION ON FLICKR 8K DATASET 45

4.5 MY MODEL IMPLEMENTATION 47

4.5.1. Data Collection 47

4.5.2. Data Cleaning 48

4.5.3 Loading the training set 49

4.5.4. Data Preprocessing — Images 49

4.5.5. Data Preprocessing — Captions 51

4.5.6. Data Preparation using Generator Function 52

4.5.7. Word Embeddings 59

4.5.8. Model Architecture 60

4.5.9. Inference 63

**CHAPTER 5: TESTING AND RESULTS 68-79**

5.1 TESTING APPROACH 68

5.2 ACCURACY MEASUREMENT 70

5.3 METHOD FOR ACCURACY MEASUREMENT 70

5.4 SOME RESULTS GENERATED BY MODEL 71

5.5 SOME MORE RESULTS 76

**CHAPTER 6: CONCLUSION AND FUTURE SCOPE 80**

6.1 CONCLUSION 80

6.2 FUTURE SCOPE 80

**REFERENCES 81**

# LIST OF TABLES


| Table No. | Table Name | Page No. |
|---|---|---|
| 4.1 | Data points corresponding to one image and its caption | 54 |
| 4.2 | Data Matrix for both the images and captions | 55 |
| 4.3 | Data matrix after replacing the words by their indices | 56 |
| 4.4 | Appending zeros to each sequence to make them all of same length 34 | 57 |

# LIST OF FIGURES


| Figure no. | Figure Caption | Page No. |
|---|---|---|
| 1.1 | Output of a given input image | 3 |
| 1.2 | caption generation | 3 |
| 1.3 | Image Captioning Model based on Neural Networks | 4 |
| 2.1 | Down sampling | 6 |
| 2.2 | RGB representation of an image | 7 |
| 2.3 | Convolution operation | 8 |
| 2.4 | Convolution with Stride 1 | 9 |
| 2.5 | Stride 1 with Padding 1 | 10 |
| 2.6 | After applying padding | 11 |
| 2.7 | Different layers of CNN | 12 |
| 2.8 | Convolutional Neural Network | 13 |
| 2.9 | Model visualization | 15 |
| 2.10 | A vanilla network representation, with an input of size 3 and one hidden layer and one output layer of size 1 | 16 |
| 2.11 | Can I simply not call a vanilla network repeatedly for a ‘series’ input? | 16 |
| 2.12 | A Recurrent Neural Network, with a hidden state that is meant to carry pertinent information from one input item in the series to others | 17 |
| 2.13 | Parameter sharing helps get rid of size limitations | 18 |
| 2.14 | We can increase depth in three possible places in a typical RNN | 20 |
| 2.15 | Bidirectional RNNs | 21 |
| 2.16 | Recursive Neural Net | 22 |

| | | |
|---|---|---|
| 2.17 | Encoder Decoder or Sequence to Sequence RNNs | 23 |
| 2.18 | Gradient Update Rule | 24 |
| 2.19 | LSTM and GRU | 25 |
| 2.20 | Processing sequence one by one | 26 |
| 2.21 | Passing hidden state to next time step | 27 |
| 2.22 | RNN Cell | 27 |
| 2.23 | Tanh squishes values to be between -1 and 1 | 28 |
| 2.24 | vector transformations without tanh | 28 |
| 2.25 | vector transformations with tanh | 29 |
| 2.26 | LSTM Cell and It’s Operations | 29 |
| 2.27 | Sigmoid squishes values to be between 0 and 1 | 30 |
| 2.28 | Forget gate operations | 31 |
| 2.29 | Input gate operations | 32 |
| 2.30 | Calculating cell state | 32 |
| 2.31 | output gate operations | 33 |
| 3.1 | An overview of the image captioning model | 34 |
| 3.2 | Sample image and corresponding captions from the Flickr8k dataset | 35 |
| 3.3 | VGG16 architecture | 36 |
| 3.4 | Four interacting layers in a LSTM layer | 37 |
| 4.1 | types of architecture | 38 |
| 4.2 | Injecting methodology | 39 |
| 4.3 | Overview of our approach | 41 |
| 4.4 | image sentence correlation | 41 |
| 4.5 | image sentence correlation | 42 |
| 4.6 | Working of the image captioning model | 43 |
| 4.7 | Illustration of our image caption pipeline | 45 |

| | | |
|---|---|---|
| 4.8 | Model Architecture | 46 |
| 4.9 | Feature Vector Extraction (Feature Engineering) from InceptionV3 | 50 |
| 4.10 | Train image's captions | 53 |
| 4.11 | High level architecture | 60 |
| 4.12 | Summary of the parameters in the model | 61 |
| 4.13 | Flowchart of the architecture | 62 |
| 4.14 | Test image | 64 |
| 5.1 | Distribution of dataset for Testing Purpose | 69 |
| 5.2 | Testing process Overview | 70 |
| 5.3 | After performing several tweaks and optimisations we were able to get an acceptable accuracy of 67.5 % on 2000 test datasets | 71 |
| 5.4 | Some Results generated by Model(output 1-10) | 75 |
| 5.5 | some more results(output1-5) | 79 |

# CHAPTER 1

# INTRODUCTION

Image Captioning is the process of generating a textual description for given images. It has been a very important and fundamental task in the Deep Learning domain. Image captioning has a huge amount of application. NVIDIA is using image captioning technologies to create an application to help people who have low or no eyesight.

Several approaches have been made to solve the task. One of the most notable work has been put forward by Andrej Karpathy, Director of AI, Tesla in his Ph.D. at Stanford. In this article, we will be talking about the most used and well-known approaches proposed as a solution to this problem. We will also be looking at a python demo example on the Flickr Dataset in Python.

## 1.1 Motivation

Generating captions for images is a vital task relevant to the area of both Computer Vision and Natural Language Processing. Mimicking the human ability of providing descriptions for images by a machine is itself a remarkable step along the line of Artificial Intelligence.

Human mind is the most complex and difficult part to understand and it contains different neurons yet computer neurons are specific as it contains some particular datasets. Neural systems are one strategy which can be utilized for picture acknowledgment and descriptor. It is an essential test for machine learning calculations, as it adds up to emulating the amazing human capacity to pack tremendous measures of remarkable visual data into graphic.

One primary motivation of computational visual recognition models is to emulate remarkable human capability to comprehend visual scenes and extract detailed information from them with astonishing accuracy [1]. Many sophisticated models have been developed to extract visual information from images based on visual categorization of objects in the images [2]. The visual recognition procedures thus pursued in most cases are demanding both in terms of computational complexity and obtaining desired accuracy. One popular endeavour of the visual recognition modelling is to extract concise natural language description of an image, i.e., to generate a single sentence representing an image most faithfully by reducing the complexities

The main challenge of this task is to capture how objects relate to each other in the image and to express them in a natural language. We As a team have worked and made use of neural networks for generating captions by taking image as input and predicting captions in next lexical unit as an output sequence.

## 1.2 General

The people communicate through language, whether written or spoken. They often use this language to describe the visual world around them. Images, signs are another way of communication and understanding for the physically challenged people. The generation of descriptions from the image automatically in proper sentences is a very difficult and challenging task [1], but it can help and have a great impact on visually impaired people for better understanding of the description of images on the web.

A good description of an image is often said for ’Visualizing a picture in the mind’. The creation of an image in mind can play a significant role in sentence generation. Also, human can describe the image after having a quick glance at it. The progress in achieving complex goals of human recognition will be done after studying existing natural image descriptions.

This task of automatically generating captions and describing the image is significantly harder than image classification and object recognition. The description of an image must involve not only the objects in the image, but also relation between the objects with their attributes and activities shown in images [20]. Most of the work done in visual recognition previously has concentrated to label images with already fixed classes or categories leading

to the large progress in this field. Eventually, vocabularies of visual concepts which are closed, makes a suitable and simple model for assumption.

Goal: To generate a caption for a given image.

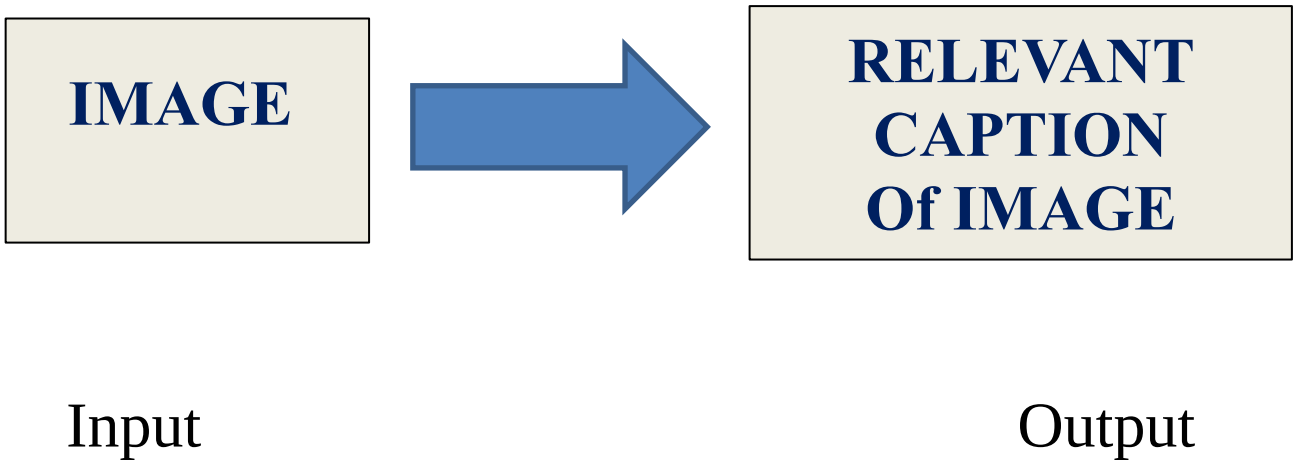



**Fig.1.1:** Output of a given input image

**EXAMPLE:**

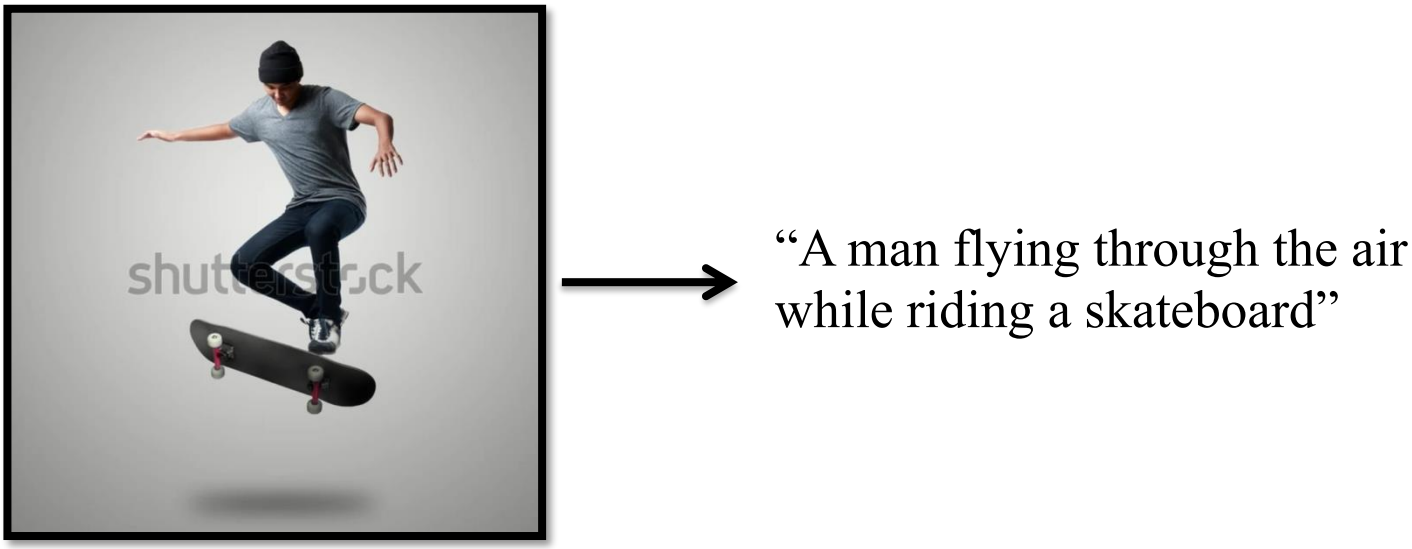



**Fig.1.2:** caption generation

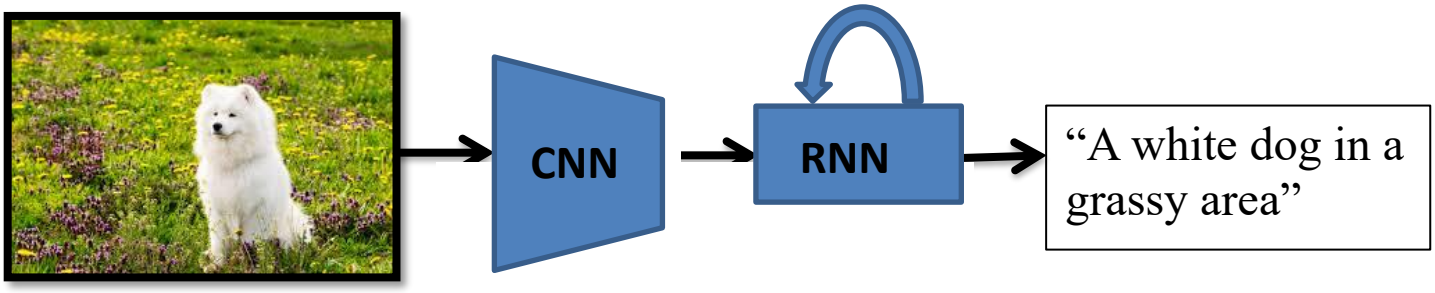



**Figure 1.3:** Image Captioning Model based on Neural Networks

## 1.3 Research Objectives

The main objective of this research is given below:

i. Developed a model for benchmark dataset: Flickr8K.
Change layer size, learning rate, optimization for achieve better accuracy.

ii. To prepare a dataset that consists of images in English geographical context and annotates them in English language.

iii. To train our model to generate caption from dataset.

# CHAPTER 2

# LITERATURE REVIEW

**Deep learning** is a machine learning technique that teaches computers to do what comes naturally to humans: learn by example. Deep learning is a key technology behind driverless cars, enabling them to recognize a stop sign, or to distinguish a pedestrian from a lamppost. It is the key to voice control in consumer devices like phones, tablets, TVs, and hands-free speakers. Deep learning is getting lots of attention lately and for good reason. It's achieving results that were not possible before. In deep learning, a computer model learns to perform classification tasks directly from images, text, or sound. Deep learning models can achieve state-of-the art accuracy, sometimes exceeding human-level performance. Models are trained by using a large set of labelled data and neural network architectures that contain many layers.

## 2.1. CNN (Convolutional neural network)

### 2.1.1 What is CNN?

Computer vision is evolving rapidly day-by-day. It's one of the reason is deep learning. When we talk about computer vision, a term convolutional neural network (abbreviated as CNN) comes in our mind because CNN is heavily used here. Examples of CNN in computer vision are face recognition, image classification etc. It is similar to the basic neural network. CNN also have learnable parameter like neural network i.e., weights, biases etc.

### 2.1.2. Why should we use CNN?

Problem with Feedforward Neural Network:

Suppose we are working with MNIST dataset, we know each image in MNIST is 28 x 28 x 1(black & white image contains only 1 channel). Total numbers of neurons in input layer will 28 x 28 = 784, this can be manageable. What if the size of image is 1000 x 1000 which means you need $10^6$ neurons in input layer. This seems a huge number of neurons are required for operation. It is computationally ineffective right. So here comes Convolutional Neural Network or CNN. In simple word what CNN does is, it extract the feature of image and convert it into lower dimension without losing its characteristics. In the following example you can see that initial the size of the image is 224 x 224 x 3. If you proceed without convolution then we need 224 x 224 x 3 = 100, 352 numbers of neurons in input layer but

after applying convolution you input tensor dimension is reduced to 1 x 1 x 1000. It means we only need 1000 neurons in first layer of feedforward neural network.

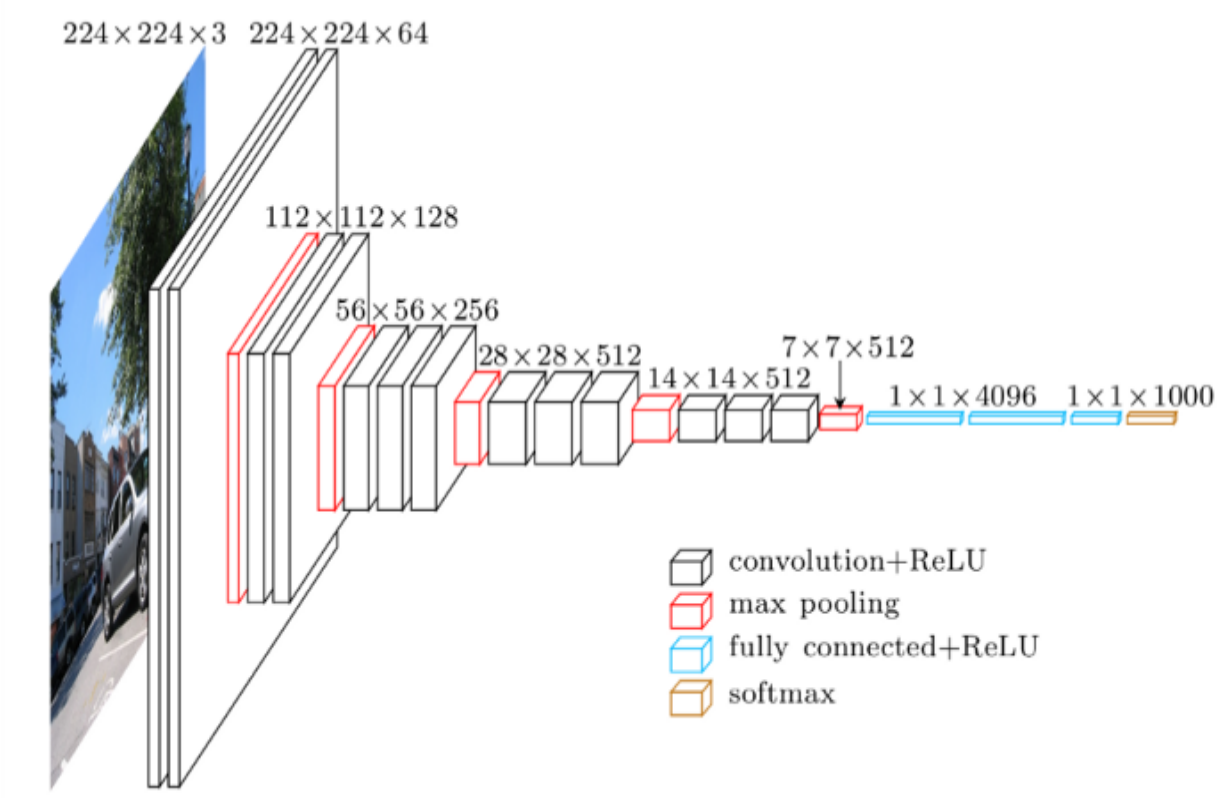



**Fig.2.1:** Down sampling

There are few definitions you should know before understanding CNN

### 2.1.3 Image Representation

Thinking about images, it's easy to understand that it has a height and width, so it would make sense to represent the information contained in it with a two dimensional structure (a matrix) until you remember that images have colours, and to add information about the colours, we need another dimension, and that is when Tensors become particularly helpful.

Images are encoded into colour channels, the image data is represented into each colour intensity in a colour channel at a given point, the most common one being RGB, which

means Red, Blue and Green. The information contained into an image is the intensity of each channel colour into the width and height of the image, just like this-


**Fig.2.2:** RGB representation of an image

So the intensity of the red channel at each point with width and height can be represented into a matrix, the same goes for the blue and green channels, so we end up having three matrices, and when these are combined they form a tensor.

### 2.1.4 Edge Detection

Every image has vertical and horizontal edges which actually combining to form an image. Convolution operation is used with some filters for detecting edges. Suppose you have grey scale image with dimension 6 x 6 and filter of dimension 3 x 3(say). When 6 x 6 grey scale image convolve with 3 x 3 filter, we get 4 x 4 image. First of all 3 x 3 filter matrix get multiplied with first 3 x 3 size of our grey scale image, then we shift one column right up to end , after that we shift one row and so on.

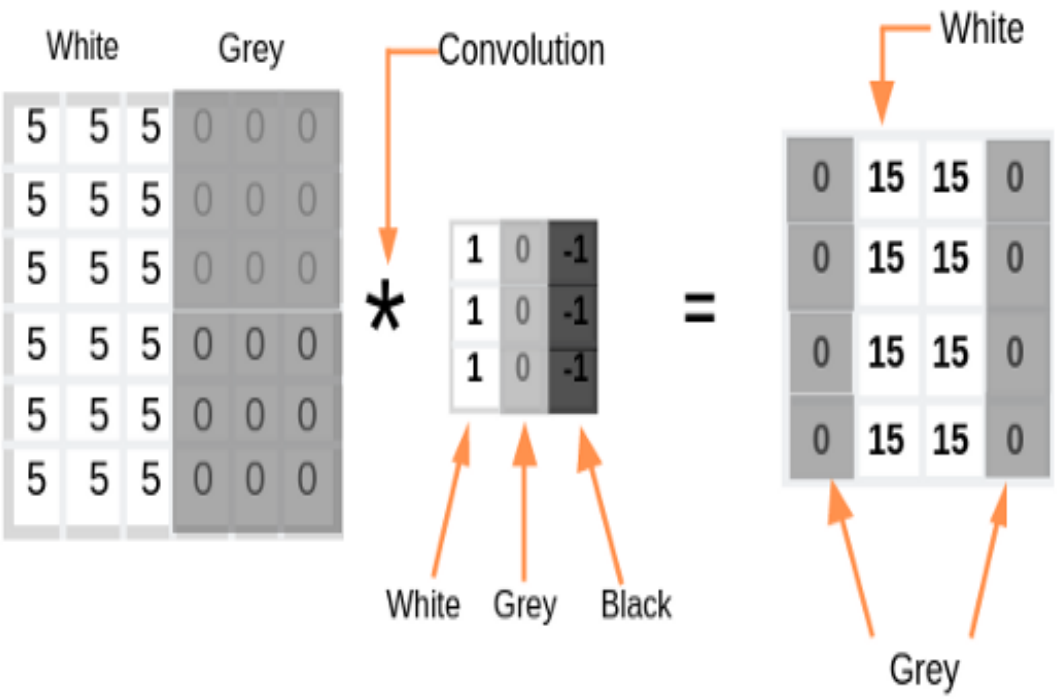



**Fig.2.3:** Convolution operation

The convolution operation can be visualized in the following way. Here our image dimension is 4 x 4 and filter is 3 x 3, hence we are getting output after convolution is 2 x 2.

If we have N x N image size and F x F filter size then after convolution result will be:

(N x N) * (F x F) = (N-F+1) x (N-F+1) (Apply this for above case)

### 2.1.5 Stride and Padding

Stride denotes how many steps we are moving in each steps in convolution. By default it is one.

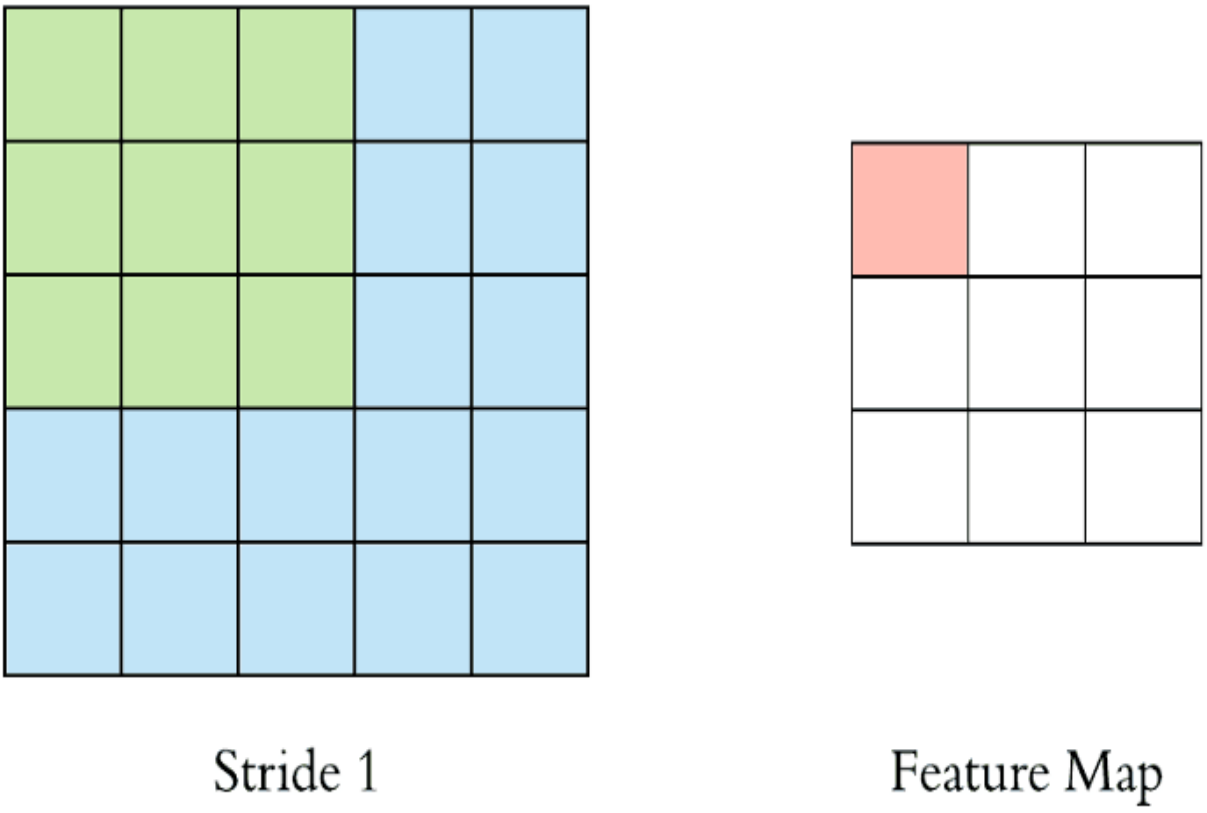
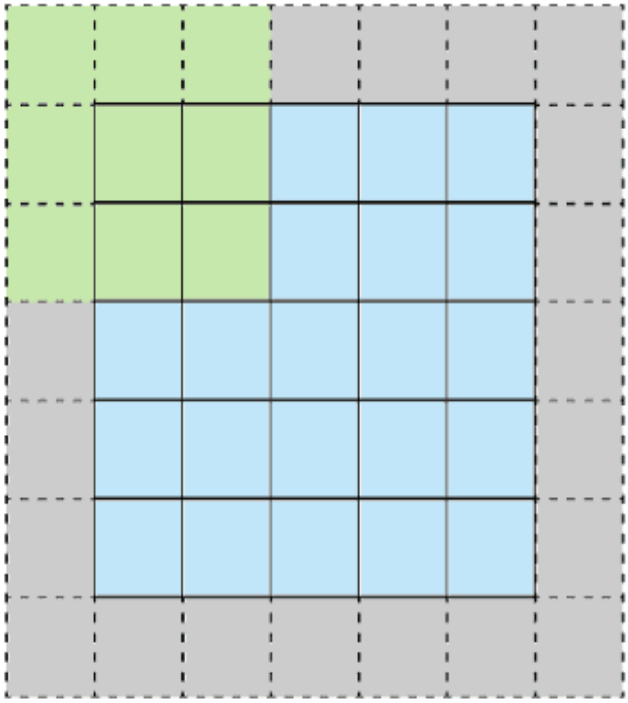
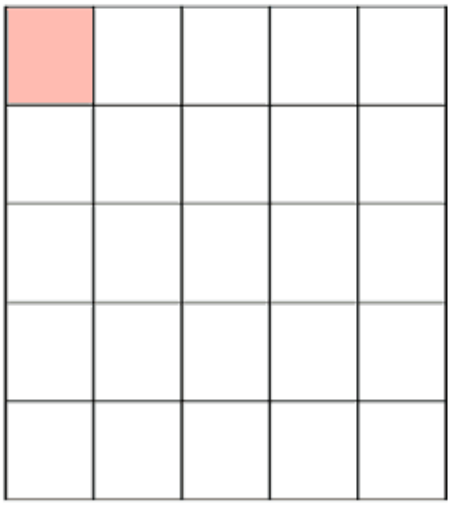



**Fig.2.4:** Convolution with Stride 1

We can observe that the size of output is smaller than input. To maintain the dimension of output as in input, we use padding. Padding is a process of adding zeros to the input matrix symmetrically. In the following example, the extra grey blocks denote the padding. It is used to make the dimension of output same as input.



Stride 1 with Padding



Feature Map

**Fig.2.5:** Stride 1 with Padding 1

Let say ‘p’ is the padding.

Initially (without padding) :

$$(N \times N) * (F \times F) = (N-F+1) \times (N-F+1) \quad \text{---(1)}$$

After applying padding:

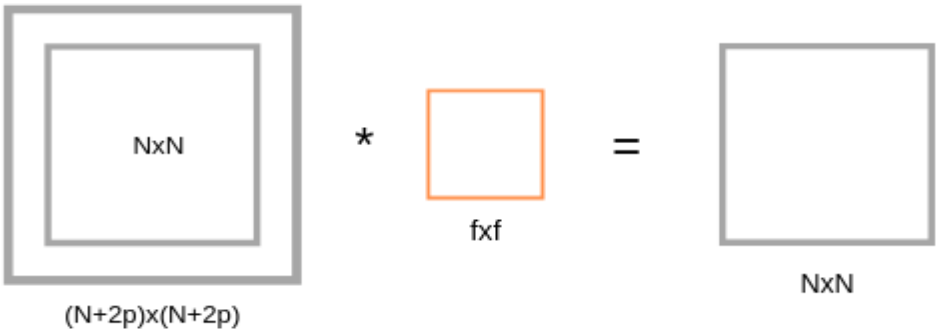



**Fig.2.6:** After applying padding

If we apply filter F x F in (N+2p) x (N+2p) input matrix with padding, then we will get output matrix dimension (N+2p-F+1) x (N+2p-F+1). As we know that after applying padding we will get the same dimension as original input dimension (N x N).

Hence we have:

$$(N+2p-F+1) \times (N+2p-F+1) \text{ equivalent to } N \times N$$

$$N+2p-F+1 = N \quad \text{---(2)}$$

$$p = (F-1)/2 \quad \text{---(3)}$$

The equation (3) clearly shows that Padding depends on the dimension of filter.

### 2.1.6 Layers in CNN

There are five different layers in CNN:

- Input layer
- Convo layer (Convo + ReLU)
- Pooling layer
- Fully connected(FC) layer
- Softmax/logistic layer
- Output layer

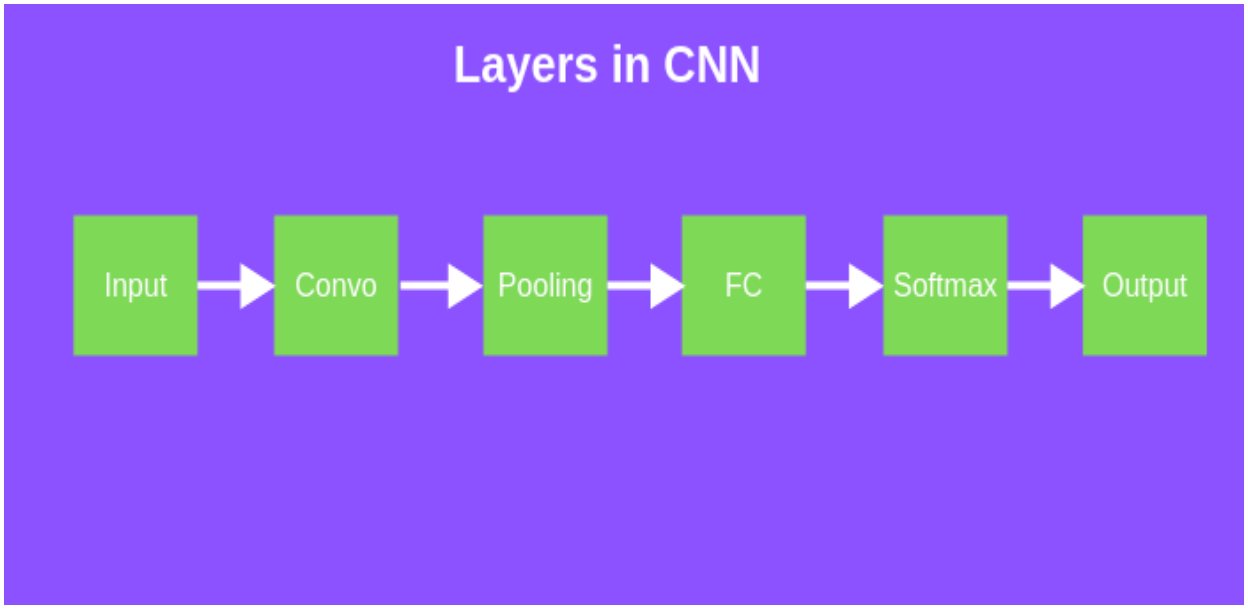



**Fig.2.7:** Different layers of CNN

### 2.1.6.1 Input Layer

Input layer in CNN should contain image data. Image data is represented by three dimensional matrix as we saw earlier. You need to reshape it into a single column. Suppose you have image of dimension 28 x 28 =784, you need to convert it into 784 x 1 before feeding into input. If you have "m" training examples then dimension of input will be (784, m).

### 2.1.6.2 Convo Layer

Convo layer is sometimes called feature extractor layer because features of the image are get extracted within this layer. First of all, a part of image is connected to Convo layer to perform convolution operation as we saw earlier and calculating the dot product between receptive fields (it is a local region of the input image that has the same size as that of filter) and the filter. Result of the operation is single integer of the output volume.

Then we slide the filter over the next receptive field of the same input image by a Stride and do the same operation again. We will repeat the same process again and again until we go through the whole image. The output will be the input for the next layer.

Convo layer also contains ReLU activation to make all negative value to zero.

### 2.1.6.3. Pooling Layer

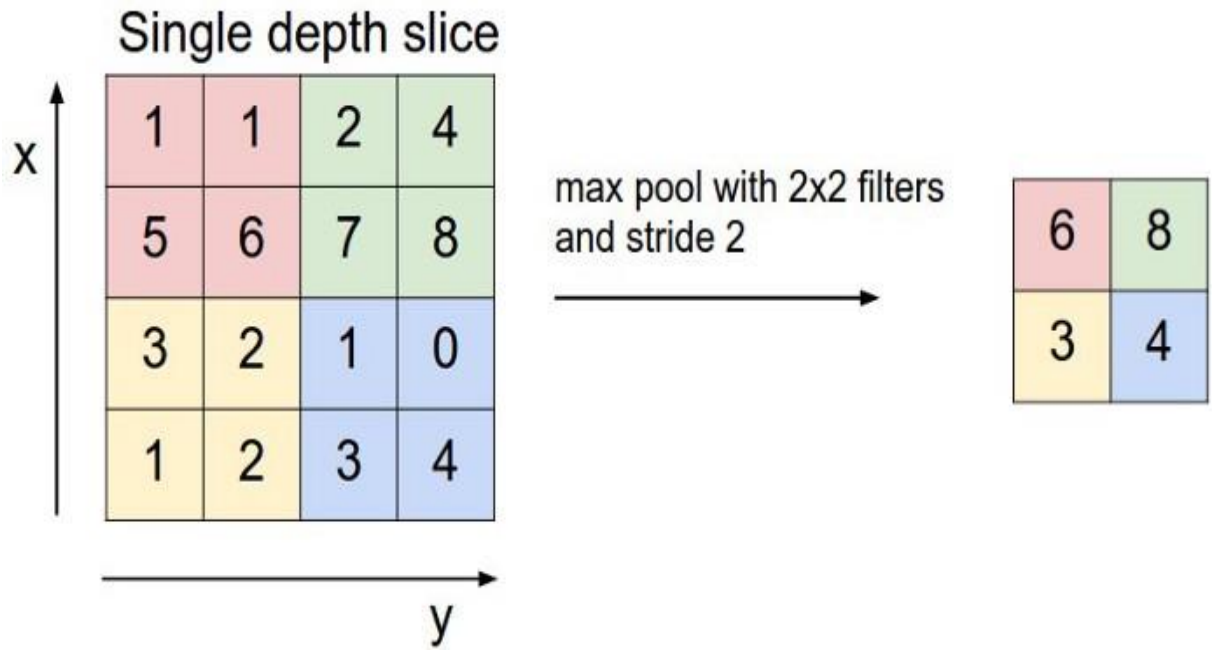



**Fig.2.8**: Convolutional Neural Network

Pooling layer is used to reduce the spatial volume of input image after convolution. It is used between two convolution layers. If we apply FC after Convo layer without applying pooling or max pooling, then it will be computationally expensive and we don't want it. So, the max pooling is only way to reduce the spatial volume of input image. In the above example, we have applied max pooling in single depth slice with Stride of 2. You can observe the 4 x 4 dimension input is reduced to 2 x 2 dimensions.

There is no parameter in pooling layer but it has two hyper parameters — Filter (F) and Stride(S).

In general, if we have input dimension W1 x H1 x D1, then

$$W2 = (W1-F)/S+1$$

$$H2 = (H1-F)/S+1$$

$$D2 = D1$$

Where W2, H2 and D2 are the width, height and depth of output.

#### 2.1.6.4. Fully Connected Layer (FC)

Fully connected layer involves weights, biases, and neurons. It connects neurons in one layer to neurons in another layer. It is used to classify images between different categories by training.

#### 2.1.6.5. Softmax / Logistic Layer

Softmax or Logistic layer is the last layer of CNN. It resides at the end of FC layer. Logistic is used for binary classification and softmax is for multi-classification.

#### 2.1.6.6. Output Layer

Output layer contains the label which is in the form of one-hot encoded.

Now you have a good understanding of CNN. Let's implement a CNN in Keras.

### 2.1.7. Keras Implementation

We will use CIFAR-10 dataset to build a CNN image classifier. CIFAR-10 dataset has 10 different labels

- Airplane
- Automobile
- Bird
- Cat
- Deer
- Dog
- Frog
- Horse
- Ship
- Truck

It has 50,000 training data and 10,000 testing image data. Image size in CIFAR-10 is 32 x 32 x 3. It comes with Keras library.

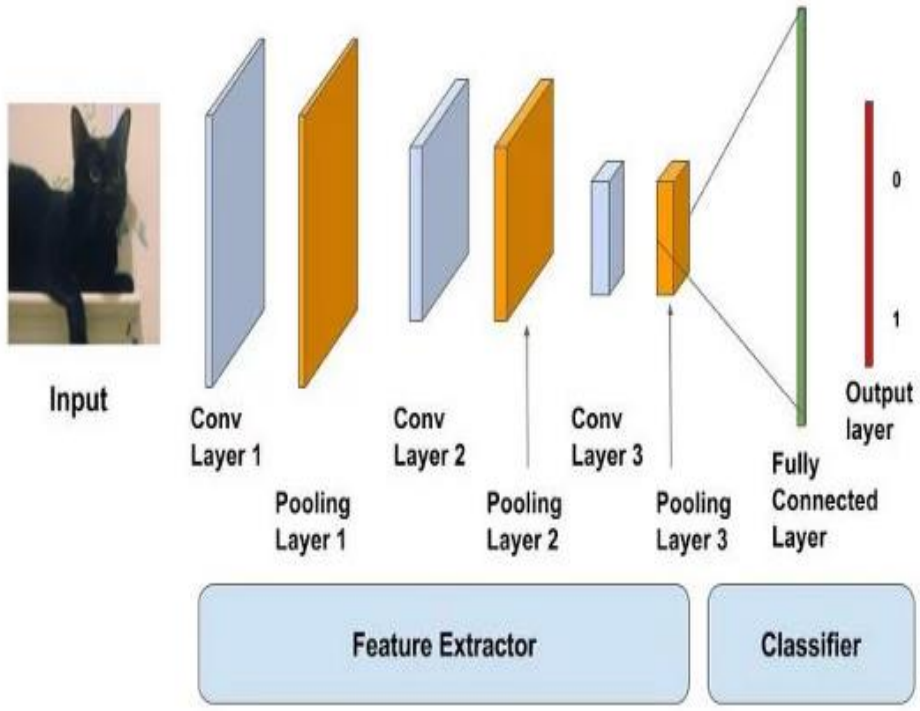



**Fig.2.9**: Model visualization

If you are using google collaborator, then make sure you are using GPU. To check whether your GPU is on or not.

## 2.2 RECURRENT NEURAL NETWORK (RNN)

Recurrent Neural Networks (RNNs) add an interesting twist to basic neural networks. A vanilla neural network takes in a fixed size vector as input which limits its usage in situations that involve a 'series' type input with no predetermined size.

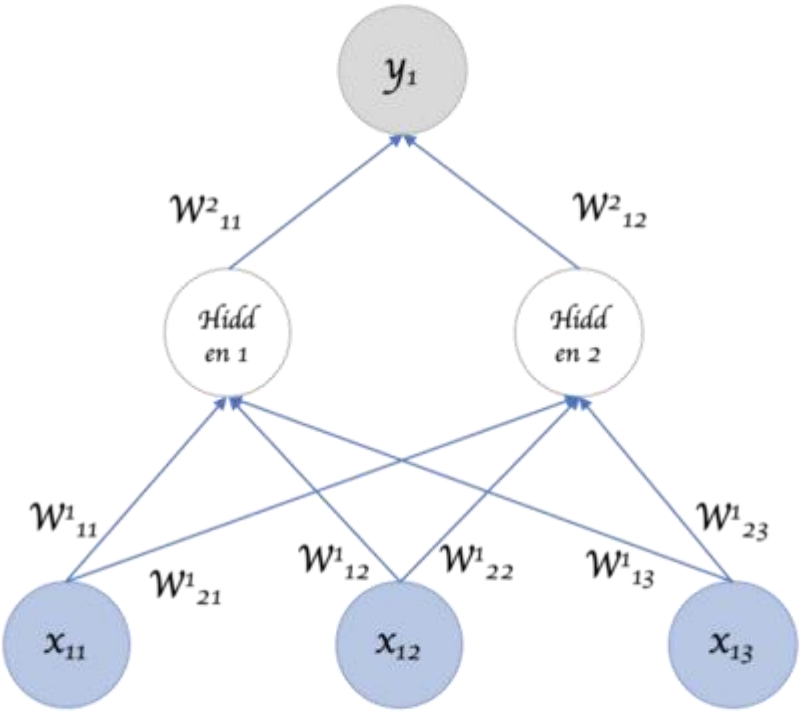



**Fig.2.10**: A vanilla network representation, with an input of size 3 and one hidden layer and one output layer of size 1.

RNNs are designed to take a series of input with no predetermined limit on size. One could ask what's the big deal, I can call a regular NN repeatedly too?

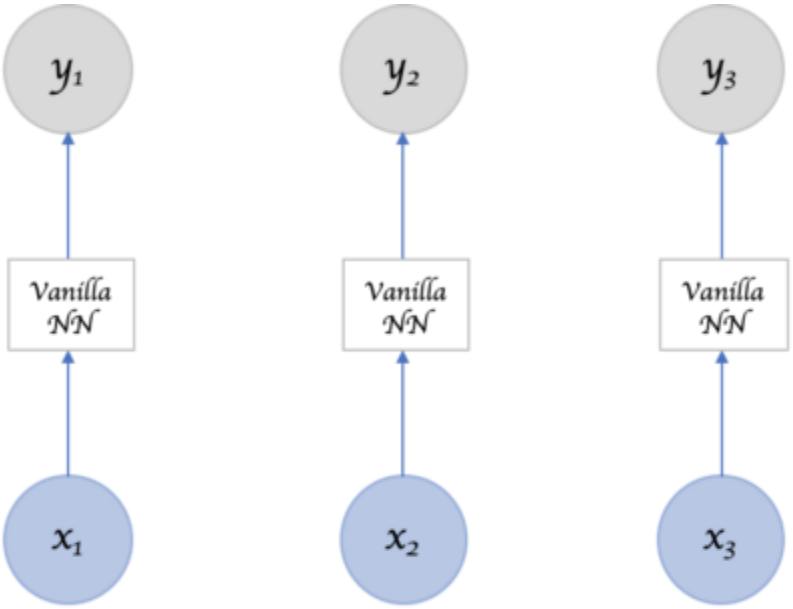


**Fig.2.11:** Can I simply not call a vanilla network repeatedly for a 'series' input?

Sure can, but the 'series' part of the input means something. A single input item from the series is related to others and likely has an influence on its neighbours. Otherwise it's just "many" inputs, not a "series" input (duh!).

So we need something that captures this relationship across inputs meaningfully.

### 2.2.1 Recurrent Neural Networks

Recurrent Neural Network remembers the past and its decisions are influenced by what it has learnt from the past. Note: Basic feed forward networks "remember" things too, but they remember things they learnt during training. For example, an image classifier learns what a "1" looks like during training and then uses that knowledge to classify things in production.

While RNNs learn similarly while training, in addition, they remember things learnt from prior input(s) while generating output(s). It's part of the network. RNNs can take one or more input vectors and produce one or more output vectors and the output(s) are influenced not just by weights applied on inputs like a regular NN, but also by a "hidden" state vector representing the context based on prior input(s)/output(s). So, the same input could produce a different output depending on previous inputs in the series.

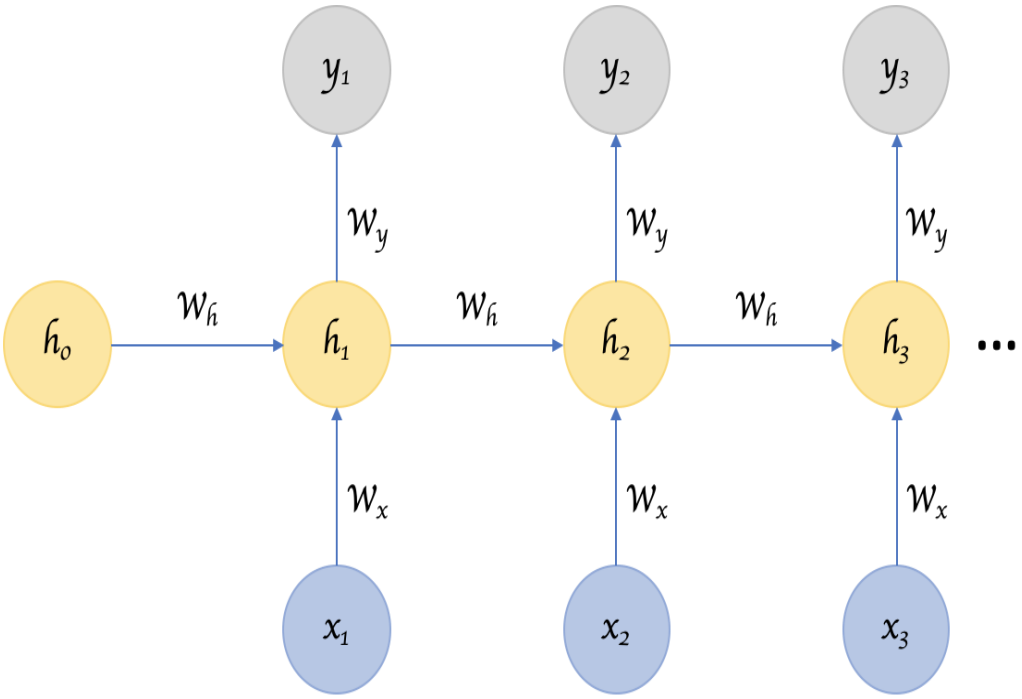


**Fig.2.12:** A Recurrent Neural Network, with a hidden state that is meant to carry pertinent information from one input item in the series to others.

In summary, in a vanilla neural network, a fixed size input vector is transformed into a fixed size output vector. Such a network becomes "recurrent" when you repeatedly apply the transformations to a series of given input and produces a series of output vectors. There is no pre-set limitation to the size of the vector. And, in addition to generating the output which is a function of the input and hidden state, we update the hidden sate itself based on the input and use it in processing the next input.

### 2.2.2 Parameter Sharing

You might have noticed another key difference between Figure 1 and Figure 3. In the earlier, multiple different weights are applied to the different parts of an input item generating a hidden layer neuron, which in turn are transformed using further weights to produce an output. There seems to be a lot of weights in play here. Whereas in Figure 3, we seem to be applying the same weights over and over again to different items in the input series.

I am sure you are quick to point out that we are kind of comparing apples and oranges here. The first figure deals with "a" single input whereas the second figure represents multiple inputs from a series. But nevertheless, intuitively speaking, as the number of inputs increase, shouldn't the number of weights in play increase as well? Are we losing some versatility and depth in Figure 3?

Perhaps we are. We are sharing parameters across inputs in Figure 3. If we don't share parameters across inputs, then it becomes like a vanilla neural network where each input node requires weights of their own. This introduces the constraint that the length of the input has to be fixed and that makes it impossible to leverage a series type input where the lengths differ and is not always known.

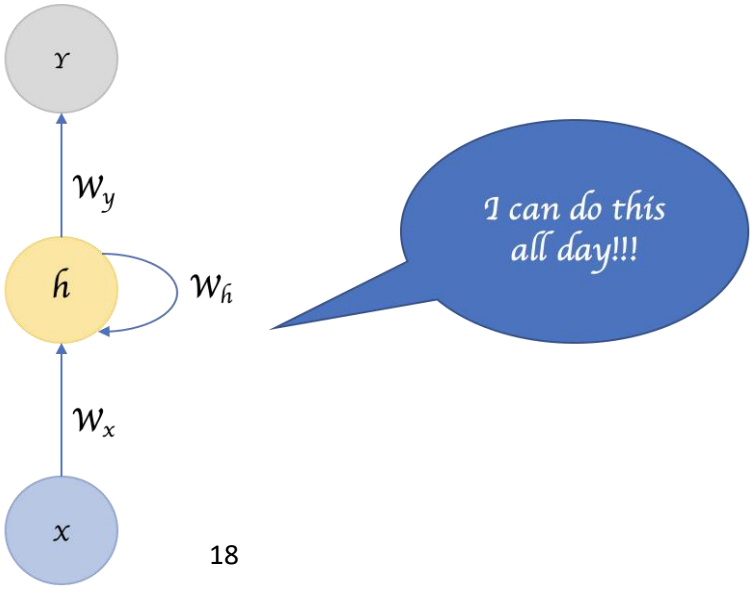


**Fig.2.13**: Parameter sharing helps get rid of size limitations

But what we seemingly lose in value here, we gain back by introducing the "hidden state" that links one input to the next. The hidden state captures the relationship that neighbours might have with each other in a serial input and it keeps changing in every step, and thus effectively every input undergoes a different transition!

Image classifying CNNs have become so successful because the 2D convolutions are an effective form of parameter sharing where each convolutional filter basically extracts the presence or absence of a feature in an image which is a function of not just one pixel but also of its surrounding neighbour pixels.

In other words, the success of CNNs and RNNs can be attributed to the concept of "parameter sharing" which is fundamentally an effective way of leveraging the relationship between one input item and its surrounding neighbours in a more intrinsic fashion compared to a vanilla neural network.


### 2.2.3 Deep RNNs

While it's good that the introduction of hidden state enabled us to effectively identify the relationship between the inputs, is there a way we can make a RNN "deep" and gain the multi-level abstractions and representations we gain through "depth" in a typical neural network?

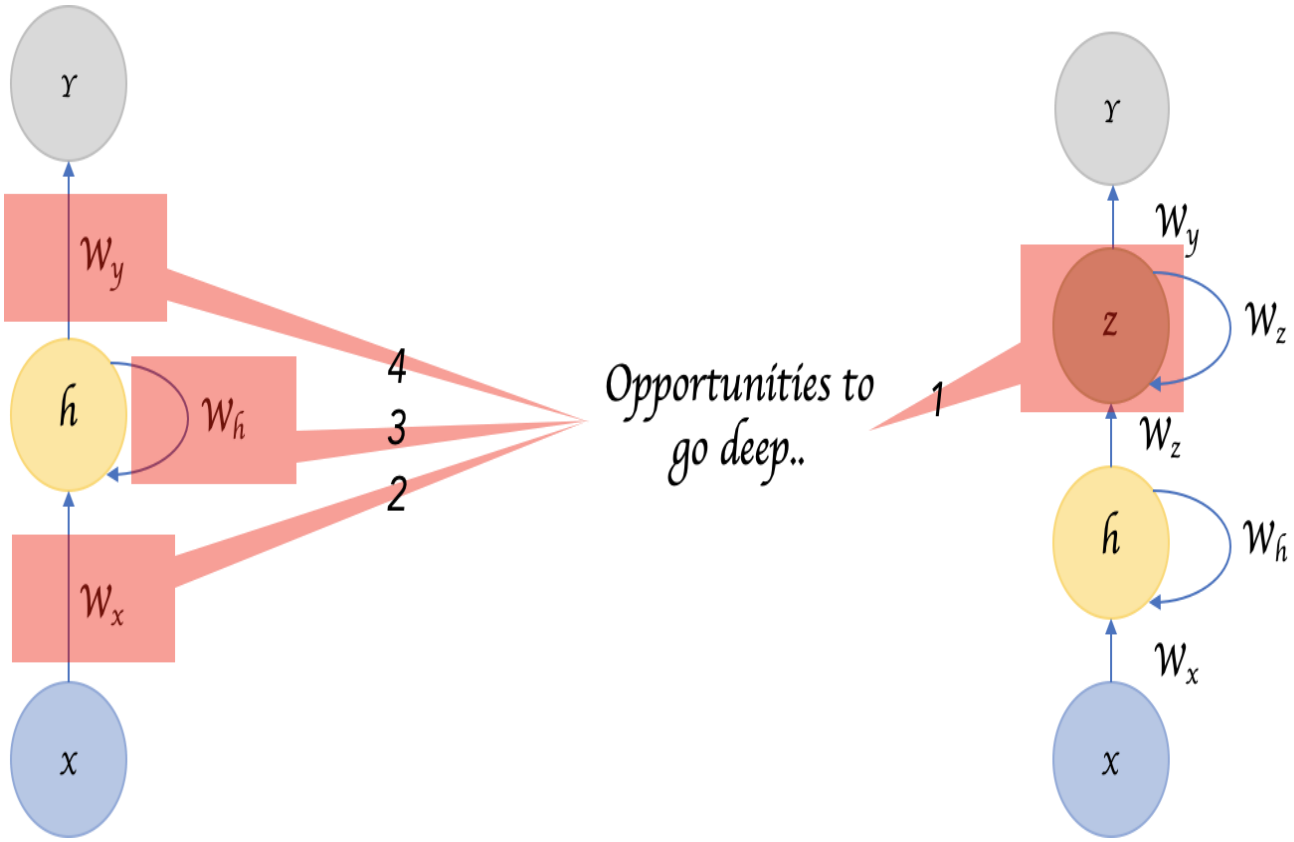



**Fig.2.14:** We can increase depth in three possible places in a typical RNN

Here are four possible ways to add depth:

(1) Perhaps the most obvious of all, is to add hidden states, one on top of another, feeding the output of one to the next.

(2) We can also add additional nonlinear hidden layers between input to hidden state

(3) We can increase depth in the hidden to hidden transition

(4) We can increase depth in the hidden to output transition. This paper by Pascanu et al., explores this in detail and in general established that deep RNNs perform better than shallow RNNs.

### 2.2.4 Bidirectional RNNs

Sometimes it's not just about learning from the past to predict the future, but we also need to look into the future to fix the past. In speech recognition and handwriting recognition tasks,

where there could be considerable ambiguity given just one part of the input, we often need to know what's coming next to better understand the context and detect the present.

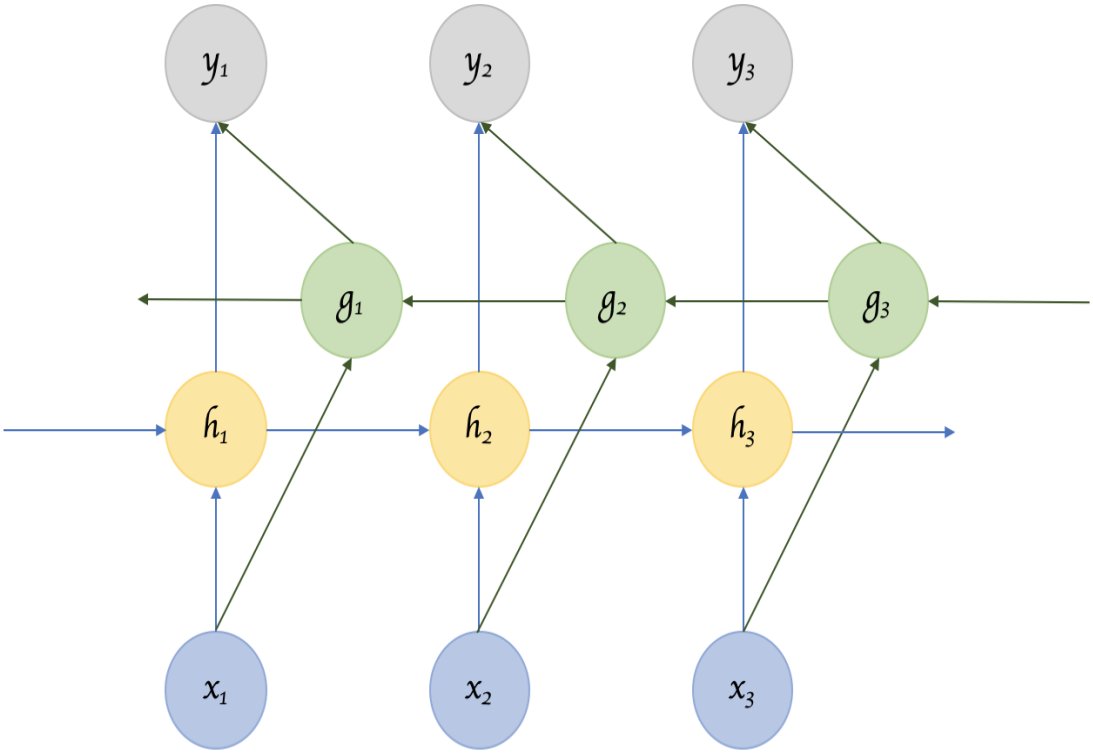



**Fig.2.15**: Bidirectional RNNs

This does introduce the obvious challenge of how much into the future we need to look into, because if we have to wait to see all inputs then the entire operation will become costly. And in cases like speech recognition, waiting till an entire sentence is spoken might make for a less compelling use case. Whereas for NLP tasks, where the inputs tend to be available, we can likely consider entire sentences all at once. Also, depending on the application, if the sensitivity to immediate and closer neighbours is higher than inputs that come further away, a variant that looks only into a limited future/past can be modelled.

### 2.2.5 Recursive Neural Networks

A recurrent neural network parses the inputs in a sequential fashion. A recursive neural network is similar to the extent that the transitions are repeatedly applied to inputs, but not necessarily in a sequential fashion. Recursive Neural Networks are a more general form of Recurrent Neural Networks. It can operate on any hierarchical tree structure. Parsing through input nodes, combining child nodes into parent nodes and combining them with other child/parent nodes to create a tree like structure. Recurrent Neural Networks do the same, but

the structure there is strictly linear. i.e. weights are applied on the first input node, then the second, third and so on.

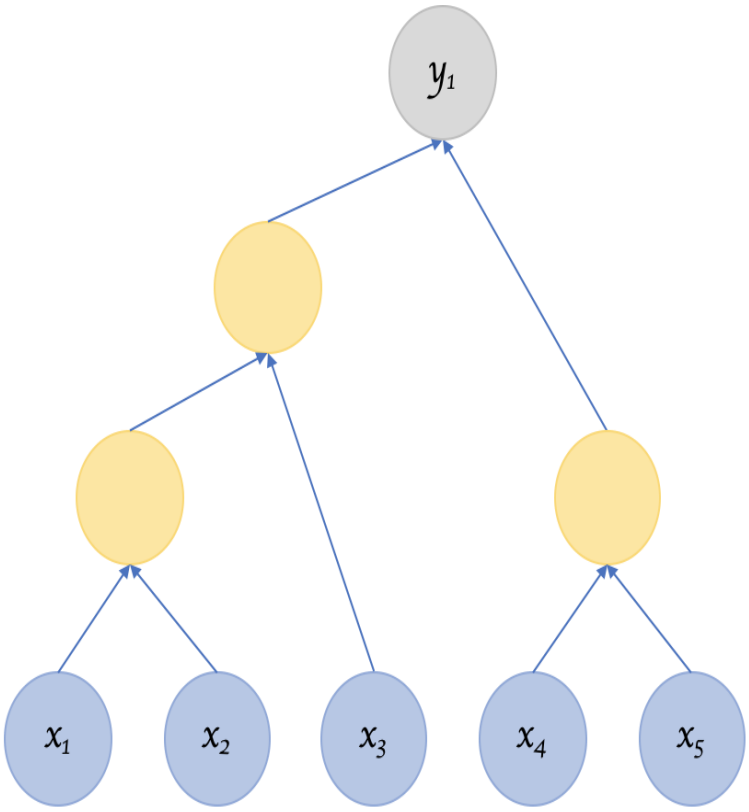



**Fig.2.16**: Recursive Neural Net

But this raises questions pertaining to the structure. How do we decide that? If the structure is fixed like in Recurrent Neural Networks then the process of training, backprop etc. makes sense in that they are similar to a regular neural network. But if the structure isn't fixed, is that learnt as well? This paper and this paper by Socher et al., explores some of the ways to parse and define the structure, but given the complexity involved, both computationally and even more importantly, in getting the requisite training data, recursive neural networks seem to be lagging in popularity to their recurrent cousin.

**Encoder Decoder Sequence to Sequence RNNs:**

Encoder Decoder or Sequence to Sequence RNNs are used a lot in translation services. The basic idea is that there are two RNNs, one an encoder that keeps updating its hidden state and produces a final single "Context" output. This is then fed to the decoder, which translates this context to a sequence of outputs. Another key difference in this arrangement is that the length of the input sequence and the length of the output sequence need not necessarily be the same.

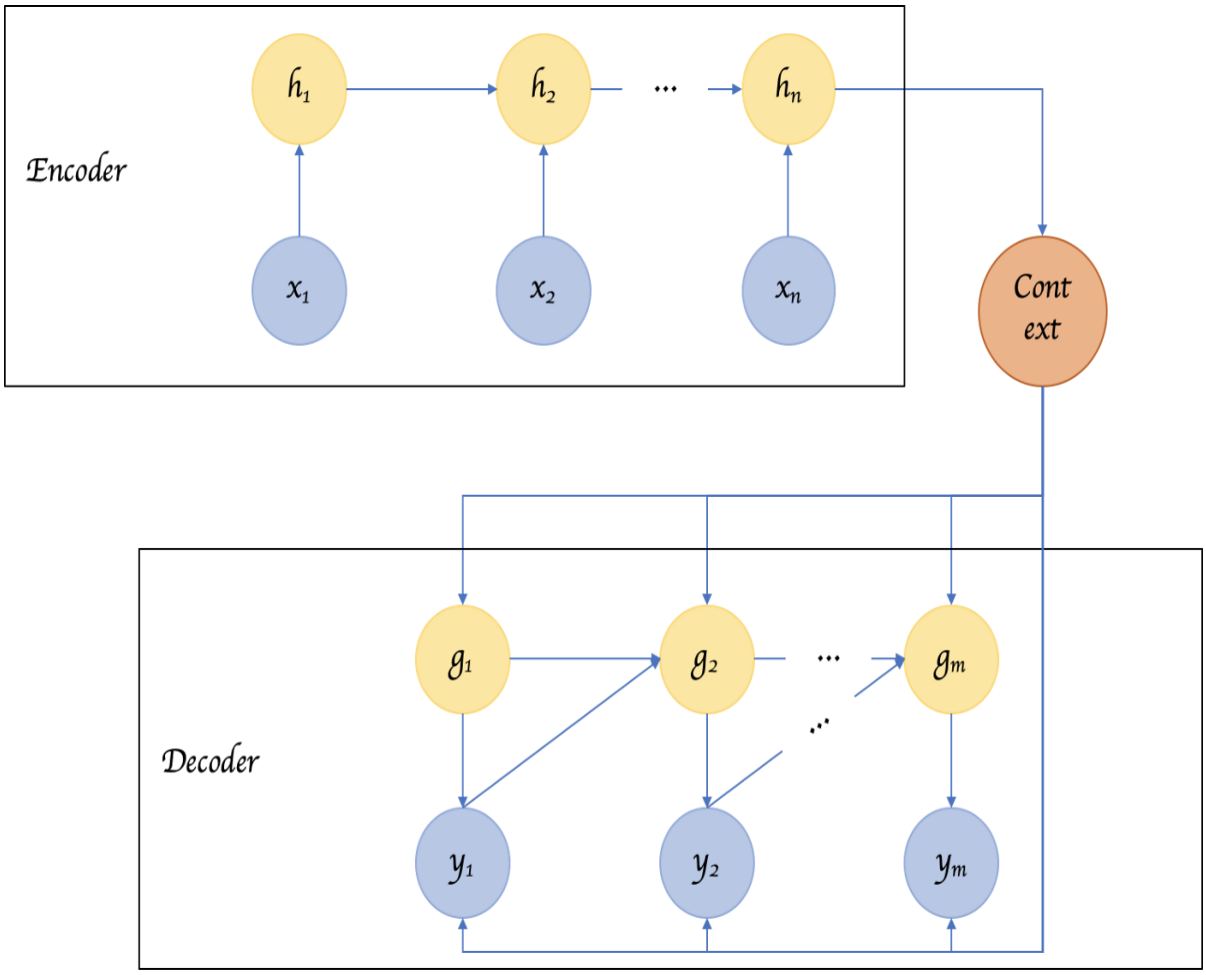



**Fig.2.17**: Encoder Decoder or Sequence to Sequence RNNs

### 2.3 LSTMs

We cannot close any post that tries to look at what RNNs and related architectures are without mentioning LSTMs. This is not a different variant of RNN architecture, but rather it introduces changes to how we compute outputs and hidden state using the inputs.

This post provides an excellent introduction to LSTMS. In a vanilla RNN, the input and the hidden state are simply passed through a single tanh layer. LSTM (Long Short Term Memory) networks improve on this simple transformation and introduce additional gates and a cell state, such that it fundamentally addresses the problem of keeping or resetting context, across sentences and regardless of the distance between such context resets. There are variants of LSTMs including GRUs that utilize the gates in different manners to address the problem of long term dependencies.

### 2.3.1 The Problem, Short-term Memory

Recurrent Neural Networks suffer from short-term memory. If a sequence is long enough, they'll have a hard time carrying information from earlier time steps to later ones. So if you are trying to process a paragraph of text to do predictions, RNN's may leave out important information from the beginning.

During back propagation, recurrent neural networks suffer from the vanishing gradient problem. Gradients are values used to update neural networks weights. The vanishing gradient problem is when the gradient shrinks as it back propagates through time. If a gradient value becomes extremely small, it doesn't contribute too much learning.


**Fig.2.18:** Gradient Update Rule

So in recurrent neural networks, layers that get a small gradient update stops learning. Those are usually the earlier layers. So because these layers don't learn, RNN's can forget what it seen in longer sequences, thus having a short-term memory. If you want to know more about the mechanics of recurrent neural networks in general, you can read my previous post here.

### 2.3.2 LSTM's and GRU's as a solution

LSTM's and GRU's were created as the solution to short-term memory. They have internal mechanisms called gates that can regulate the flow of information.

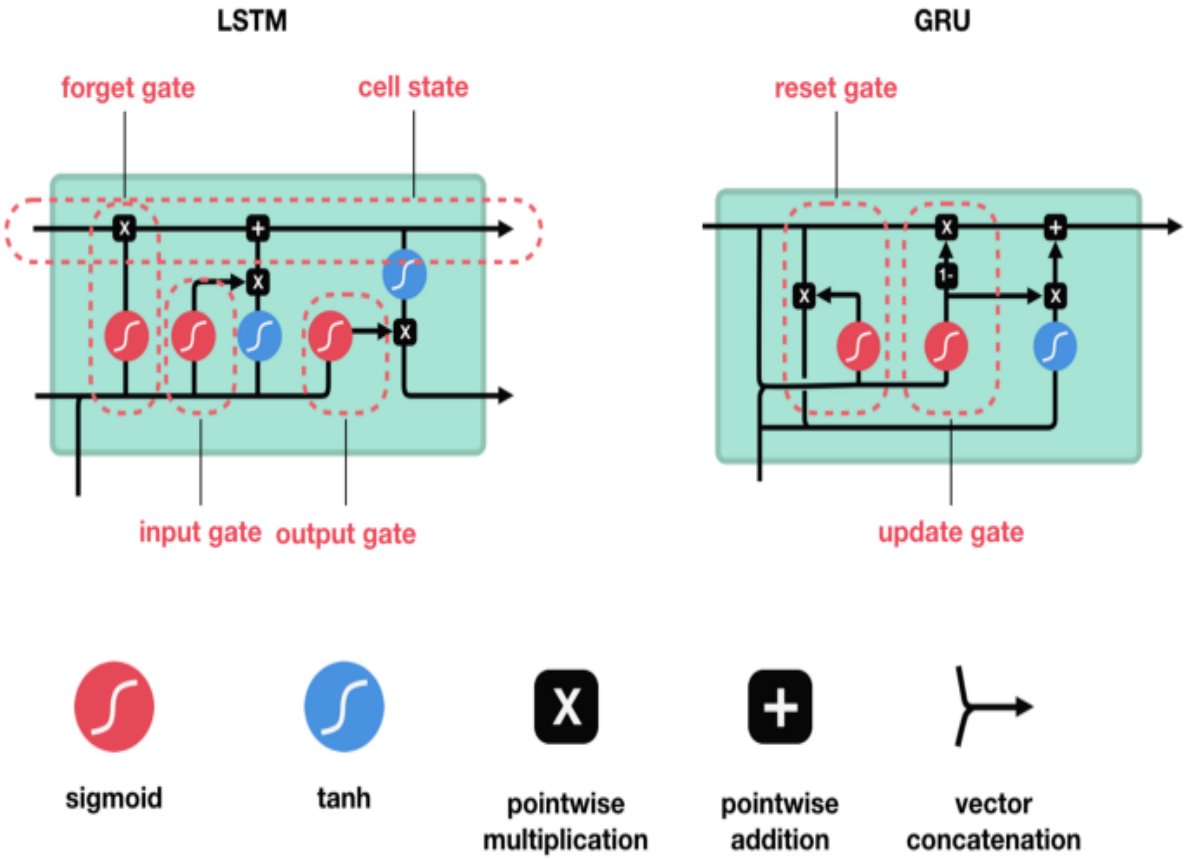



**Fig.2.19:** LSTM and GRU

These gates can learn which data in a sequence is important to keep or throw away. By doing that, it can pass relevant information down the long chain of sequences to make predictions. Almost all state of the art results based on recurrent neural networks are achieved with these two networks. LSTM's and GRU's can be found in speech recognition, speech synthesis, and text generation. You can even use them to generate captions for videos.

Ok, so by the end of this post you should have a solid understanding of why LSTM's and GRU's are good at processing long sequences. I am going to approach this with intuitive explanations and illustrations and avoid as much math as possible.

### 2.3.3 Intuition

Ok, Let's start with a thought experiment. Let's say you're looking at reviews online to determine if you want to buy Life cereal (don't ask me why). You'll first read the review then determine if someone thought it was good or if it was bad.

When you read the review, your brain subconsciously only remembers important keywords. You pick up words like "amazing" and "perfectly balanced breakfast". You don't care much

for words like “this”, “gave“, “all”, “should”, etc. If a friend asks you the next day what the review said, you probably wouldn’t remember it word for word. You might remember the main points though like “will definitely be buying again”. If you’re a lot like me, the other words will fade away from memory.

And that is essentially what an LSTM or GRU does. It can learn to keep only relevant information to make predictions, and forget non relevant data. In this case, the words you remembered made you judge that it was good.

### 2.3.4 Review of Recurrent Neural Networks

To understand how LSTM’s or GRU’s achieves this, let’s review the recurrent neural network. An RNN works like this; First words get transformed into machine-readable vectors. Then the RNN processes the sequence of vectors one by one.

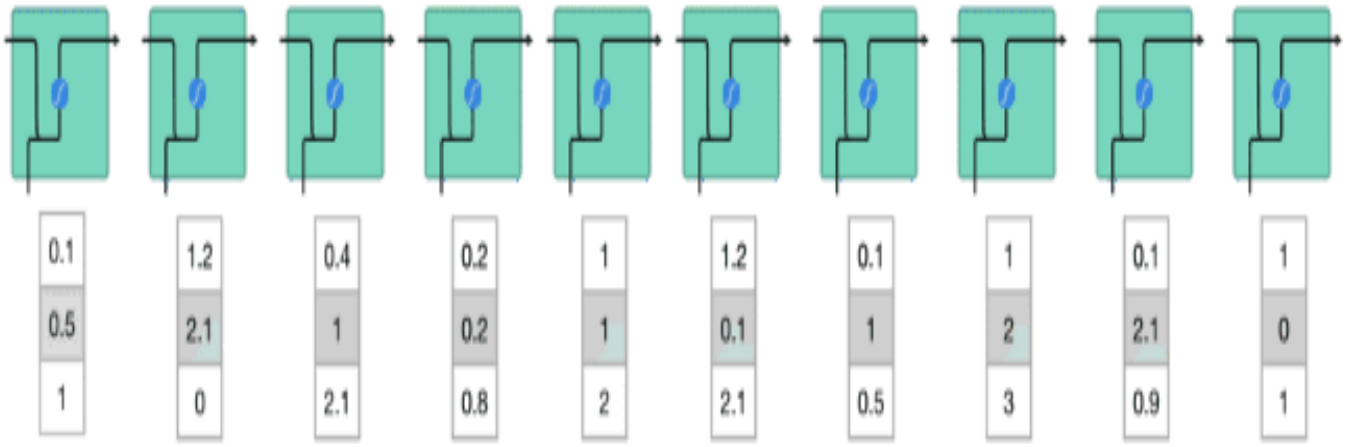



**Fig.2.20:** Processing sequence one by one

While processing, it passes the previous hidden state to the next step of the sequence. The hidden state acts as the neural networks memory. It holds information on previous data the network has seen before.

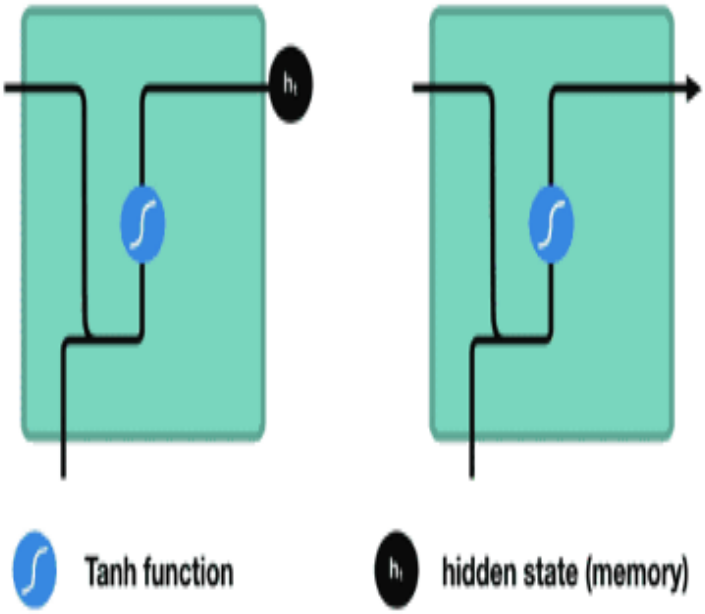



**Fig.2.21:** Passing hidden state to next time step

Let's look at a cell of the RNN to see how you would calculate the hidden state. First, the input and previous hidden state are combined to form a vector. That vector now has information on the current input and previous inputs. The vector goes through the tanh activation, and the output is the new hidden state, or the memory of the network.

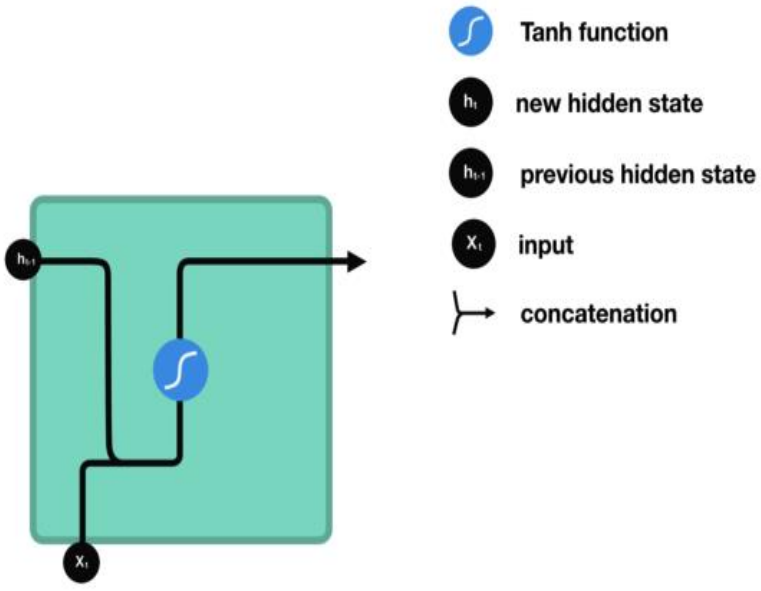



**Fig.2.22:** RNN Cell

### 2.3.4.1 Tanh activation

The tanh activation is used to help regulate the values flowing through the network. The tanh function squishes values to always be between -1 and 1.

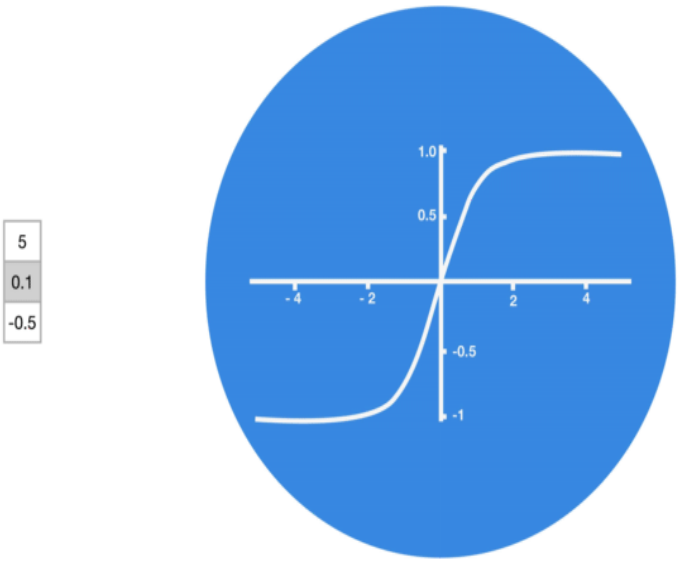



**Fig.2.23:** Tanh squishes values to be between -1 and 1

When vectors are flowing through a neural network, it undergoes many transformations due to various math operations. So imagine a value that continues to be multiplied by let's say ***3***. You can see how some values can explode and become astronomical, causing other values to seem insignificant.

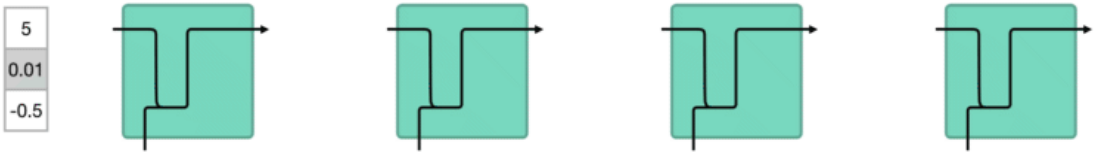



**Fig.2.24:** vector transformations without tanh

A tanh function ensures that the values stay between -1 and 1, thus regulating the output of the neural network. You can see how the same values from above remain between the boundaries allowed by the tanh function.

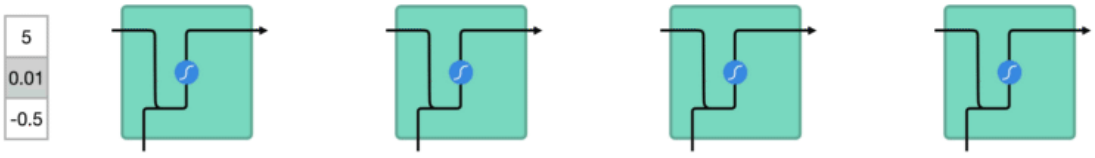



**Fig.2.25:** vector transformations with tanh

So that's an RNN. It has very few operations internally but works pretty well given the right circumstances (like short sequences). RNN's uses a lot less computational resources than its evolved variants, LSTM's and GRU's.

### 2.3.5 LSTM

An LSTM has a similar control flow as a recurrent neural network. It processes data passing on information as it propagates forward. The differences are the operations within the LSTM's cells.

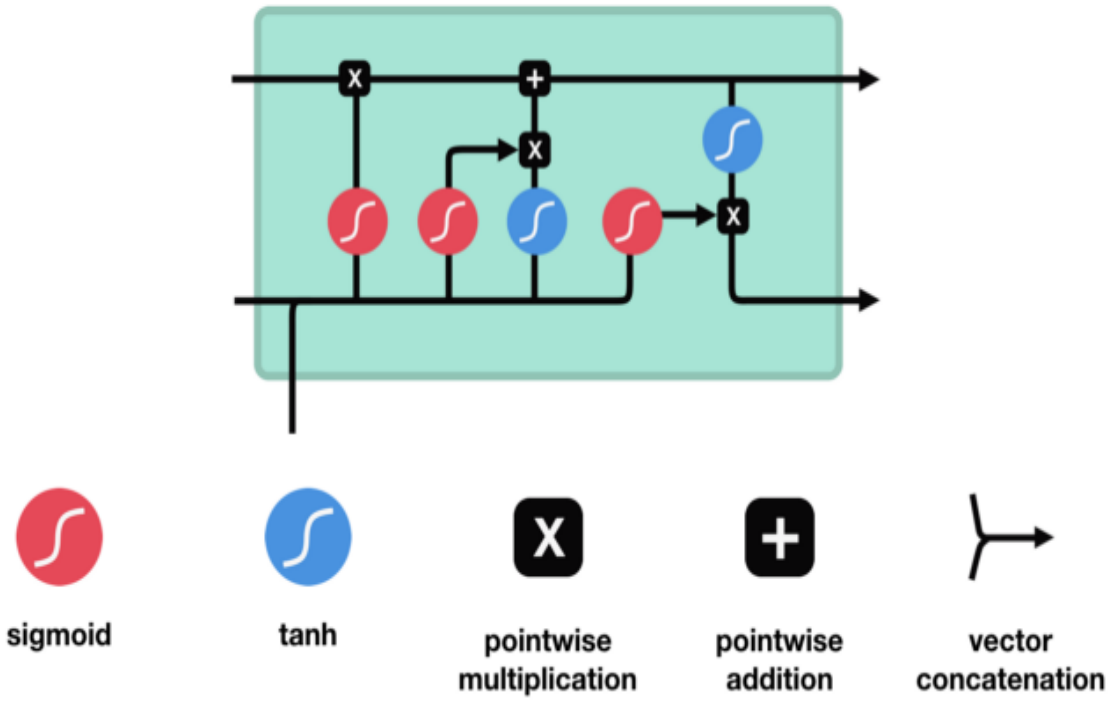



**Fig.2.26:** LSTM Cell and its Operations

These operations are used to allow the LSTM to keep or forget information. Now looking at these operations can get a little overwhelming so we'll go over this step by step.

### 2.3.5.1 Core Concept

The core concept of LSTM's are the cell state, and its various gates. The cell state act as a transport highway that transfers relative information all the way down the sequence chain. You can think of it as the "memory" of the network. The cell state, in theory, can carry relevant information throughout the processing of the sequence. So even information from the earlier time steps can make its way to later time steps, reducing the effects of short-term memory. As the cell state goes on its journey, information gets added or removed to the cell state via gates. The gates are different neural networks that decide which information is allowed on the cell state. The gates can learn what information is relevant to keep or forget during training.

### 2.3.5.2 Sigmoid

Gates contains sigmoid activations. A sigmoid activation is similar to the tanh activation. Instead of squishing values between -1 and 1, it squishes values between 0 and 1. That is helpful to update or forget data because any number getting multiplied by 0 is 0, causing values to disappears or be "forgotten." Any number multiplied by 1 is the same value therefore that value stay's the same or is "kept." The network can learn which data is not important therefore can be forgotten or which data is important to keep.

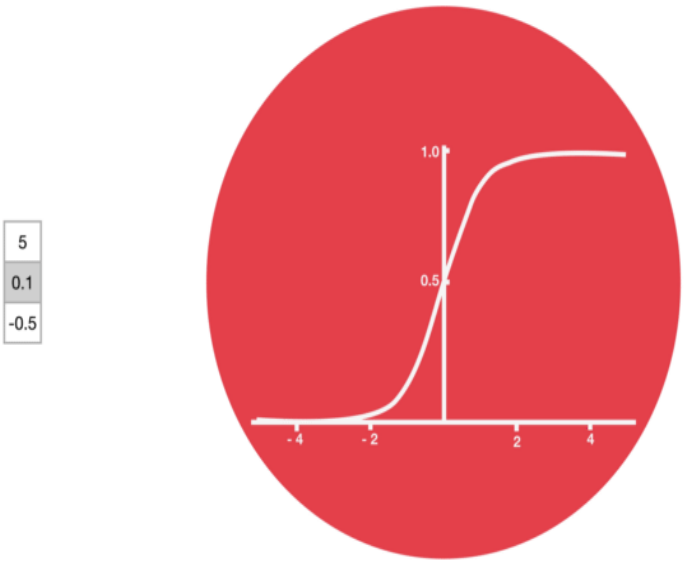



**Fig.2.27:** Sigmoid squishes values to be between 0 and 1

Let's dig a little deeper into what the various gates are doing, shall we? So we have three different gates that regulate information flow in an LSTM cell. A forget gate, input gate, and output gate.

### 2.3.5.3 Forget gate

First, we have the forget gate. This gate decides what information should be thrown away or kept. Information from the previous hidden state and information from the current input is

passed through the sigmoid function. Values come out between 0 and 1. The closer to 0 means to forget and the closer to 1 means to keep.

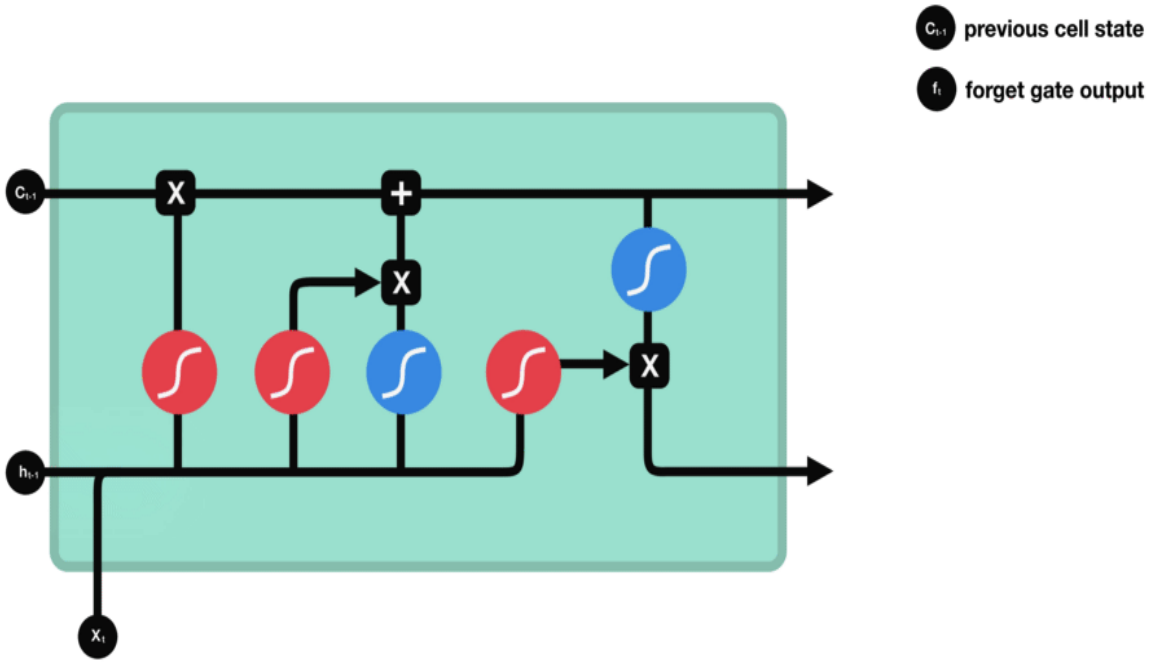



**Fig.2.28:** Forget gate operations

### 2.3.5.4 Input Gate

To update the cell state, we have the input gate. First, we pass the previous hidden state and current input into a sigmoid function. That decides which values will be updated by transforming the values to be between 0 and 1. 0 means not important and 1 means important. You also pass the hidden state and current input into the tanh function to squish values between -1 and 1 to help regulate the network. Then you multiply the tanh output with the sigmoid output. The sigmoid output will decide which information is important to keep from the tanh output.

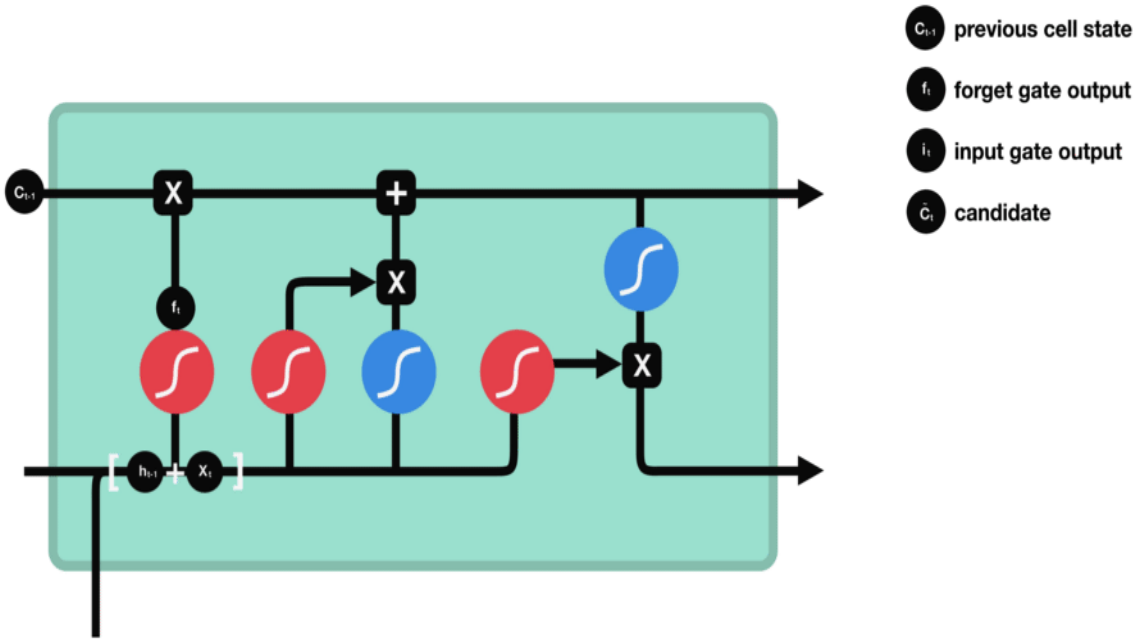



**Fig.2.29:** Input gate operations

### 2.3.5.5 Cell State

Now we should have enough information to calculate the cell state. First, the cell state gets point wise multiplied by the forget vector. This has a possibility of dropping values in the cell state if it gets multiplied by values near 0. Then we take the output from the input gate and do a point wise addition which updates the cell state to new values that the neural network finds relevant. That gives us our new cell state.

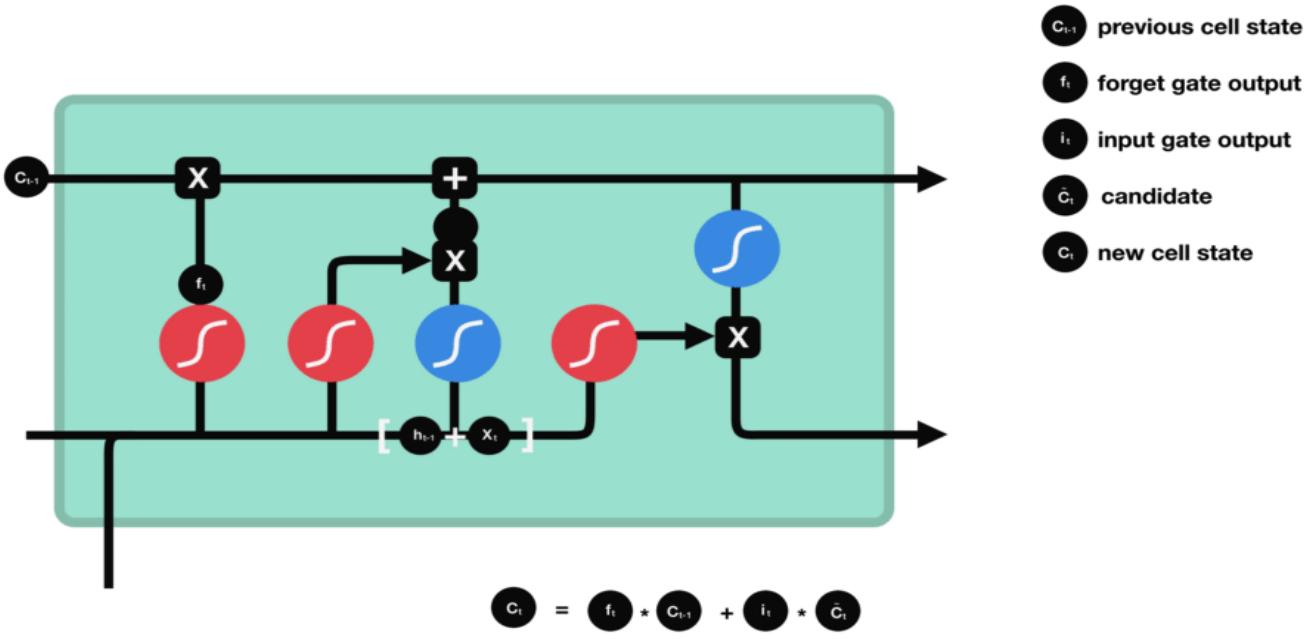



**Fig.2.30:** Calculating cell state

### 2.3.5.6 Output Gate

Last we have the output gate. The output gate decides what the next hidden state should be. Remember that the hidden state contains information on previous inputs. The hidden state is also used for predictions. First, we pass the previous hidden state and the current input into a sigmoid function. Then we pass the newly modified cell state to the tanh function. We multiply the tanh output with the sigmoid output to decide what information the hidden state should carry. The output is the hidden state. The new cell state and the new hidden is then carried over to the next time step.

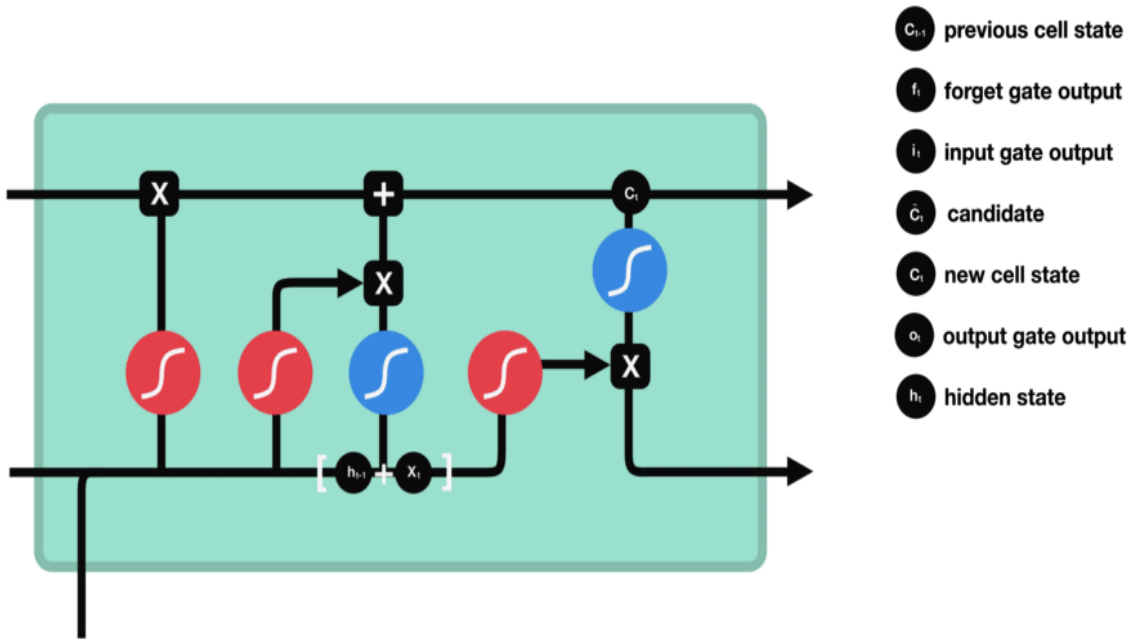



**Fig.2.31:** output gate operations

To review, the Forget gate decides what is relevant to keep from prior steps. The input gate decides what information is relevant to add from the current step. The output gate determines what the next hidden state should be.

That's it! The control flow of an LSTM network are a few tensor operations and a for loop. You can use the hidden states for predictions. Combining all those mechanisms, an LSTM can choose which information is relevant to remember or forget during sequence processing.

# CHAPTER 3

# METHODOLOGY

## 3.1 Model Overview

The model proposed takes an image I as input and is trained to maximize the probability of p(S|I) [1] where S is the sequence of words generated from the model and each word St Is generated from a dictionary built from the training dataset. The input image is fed into a deep vision Convolutional Neural Network (CNN) which helps in detecting the objects present in the image. The image encodings are passed on to the Language Generating Recurrent Neural Network (RNN) which helps in generating a meaningful sentence for the image as shown in the fig. 3.1. An analogy to the model can be given with a language translation RNN model where we try to maximize the p (T|S) where T is the translation to the sentence S. However, in our model the encoder RNN which helps in transforming an input sentence to a fixed length vector is replaced by a CNN encoder. Recent research has shown that the CNN can easily transform an input image to a vector.

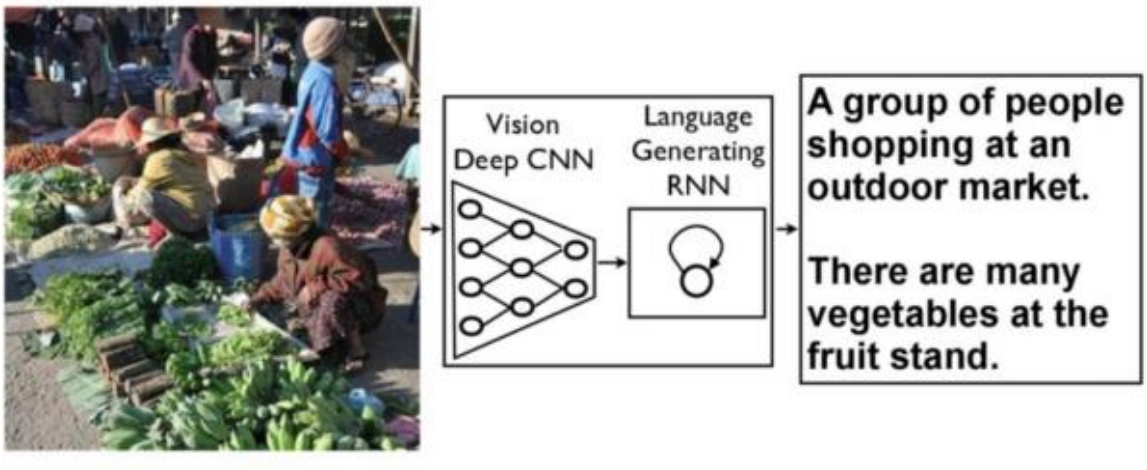



**Fig.3.1:** An overview of the image captioning model

For the task of image classification, we use a pretrained model VGG16. The details of the models are discussed in the following section. A Long Short-Term Memory (LSTM) network follows the pretrained VGG16 [2]. The LSTM network is used for language generation. LSTM differs from traditional Neural Networks as a current token is dependent on the previous tokens for a sentence to be meaningful and LSTM networks take this factor into

account. In the following sections we discuss the components of the model i.e. the CNN encoder and the Language generating RNN in details.

## 3.2 Dataset

For the task of image captioning we use Flickr8k dataset. The dataset contains 8000 images with 5 captions per image. The dataset by default is split into image and text folders. Each image has a unique id and the caption for each of these images is stored corresponding to the respective id. The dataset contains 6000 training images, 1000 development images and 1000 test images. A sample from the data is given in fig. 3.2.

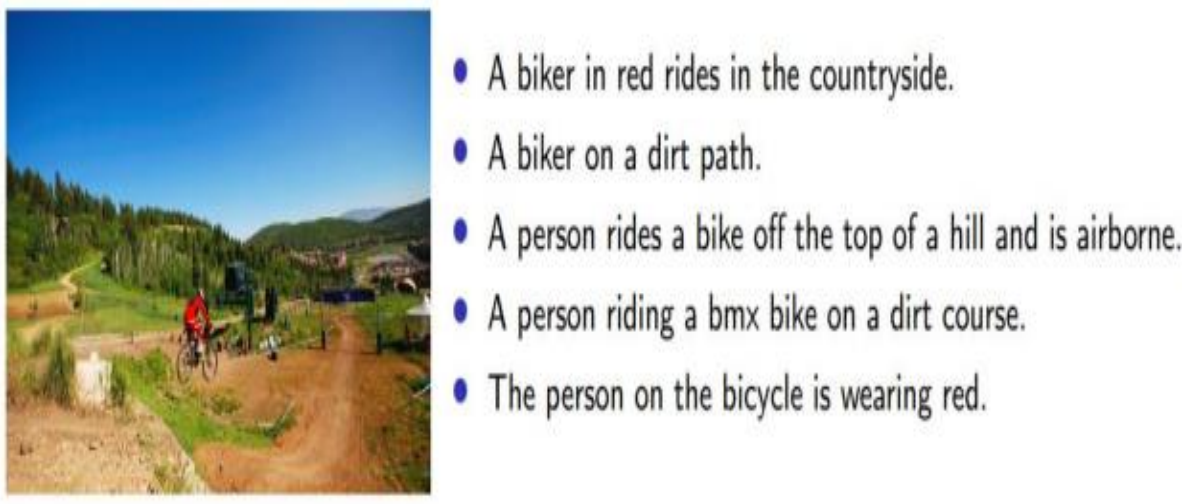



**Fig.3.2:** Sample image and corresponding captions from the Flickr8k dataset

Other datasets like Flickr30k and MSCOCO for image captioning exists but both these datasets have more than 30,000 images thus processing them becomes computationally very expensive. Captions generated using these datasets may prove to be better than the ones generated after training on Flickr8k because the dictionary of words used by RNN decoder would be larger in case of Flickr30k and MSCOCO.

## 3.3 .Image Features Detection:

For image Detection, we are using a pre-trained model which is VGG16. VGG16 is already installed in the Keras library.VGG 16 was proposed by Karen Simonyan and Andrew Zisserman of the Visual Geometry Group Lab of Oxford University in 2014 in the paper VERY DEEP CONVOLUTIONAL NETWORKS FOR LARGE-SCALE IMAGE RECOGNITION.

Here is the model representation in 3-D and in 2-D:

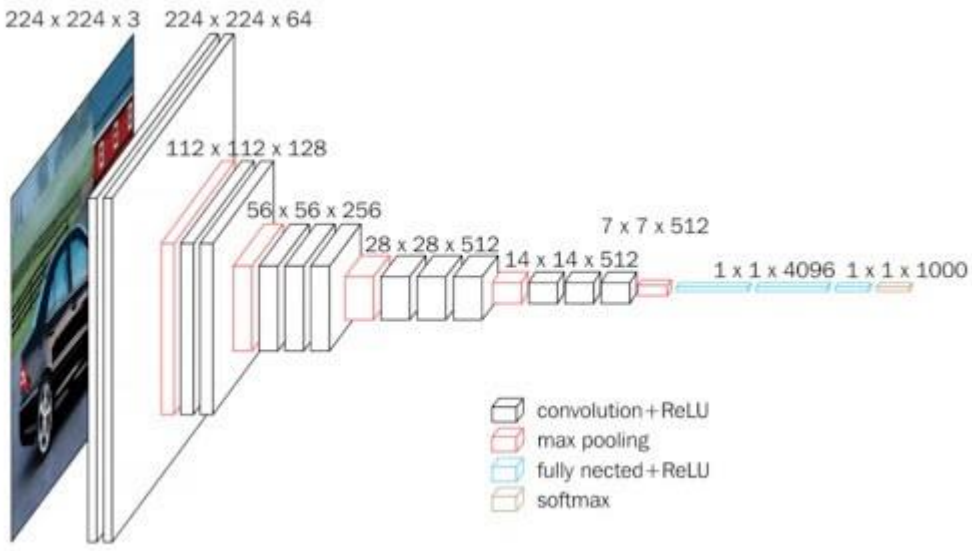
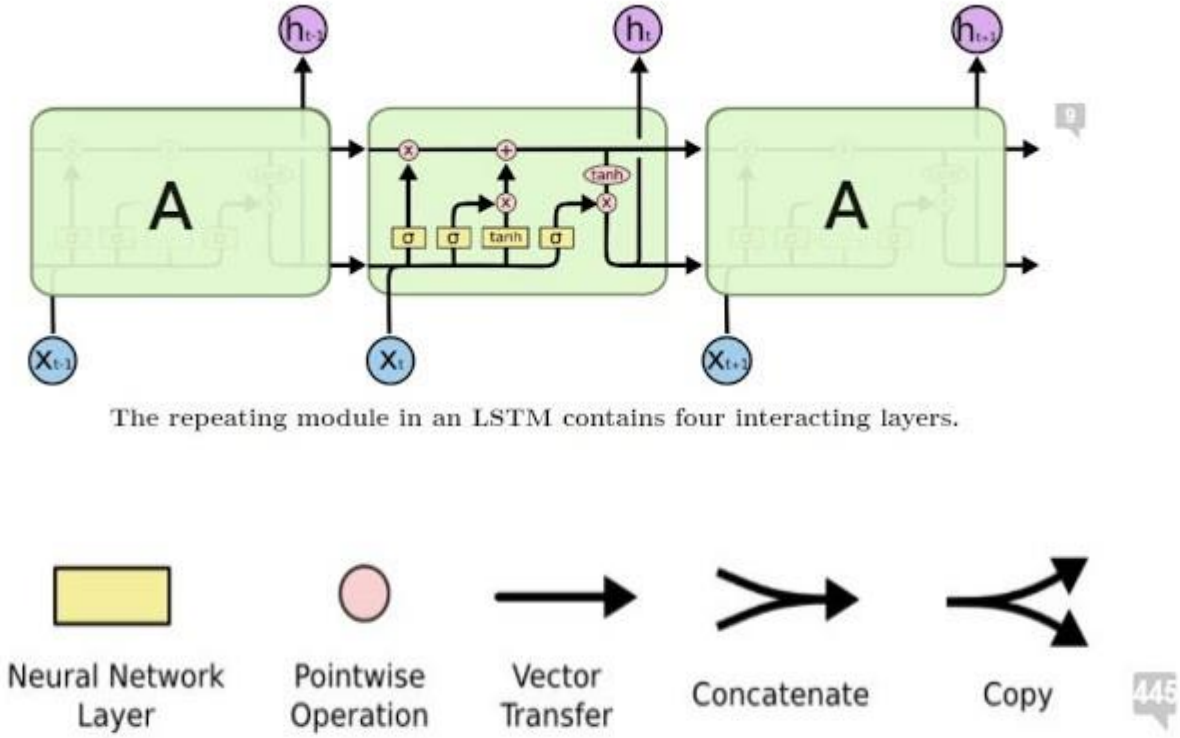



**Fig.3.3:** VGG16 architecture

## 3.4. Text Generation using LSTM

Long Short Term Memory networks — usually just called "LSTMs" — are a special kind of RNN, capable of learning long-term dependencies. They were introduced by Hochreiter &

Schmidhuber (1997) and were refined and popularized by many people in the following work. They work tremendously well on a large variety of problems and are now widely used.




The repeating module in an LSTM contains four interacting layers.

**Fig.3.4:** Four interacting layers in a LSTM layer

# CHAPTER 4

# IMPLEMENTATION

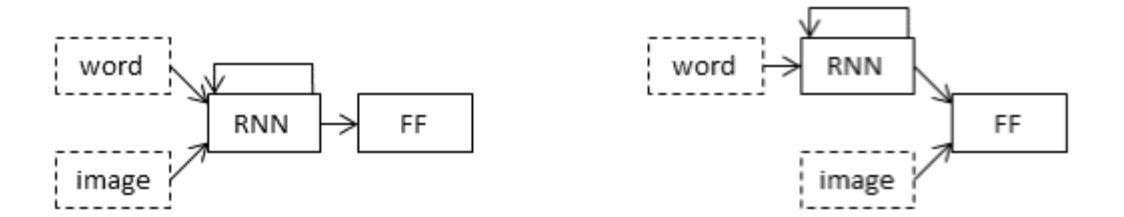
We have seen that we need to create a multimodal neural network that uses feature vectors obtained using both RNN and CNN, so consequently, we will have two inputs. One is the image we need to describe, a feed to the CNN, and the second is the words in the text sequence produced till now as a sequence as the input to the RNN.

We are dealing with two types of information, a language one and another image one. So, the question arises how or in what order should we introduce the pieces of information into our model? Elaborately speaking**,** we need a language RNN model as we want to generate a word sequence, so, when should we introduce the image data vectors in the language model. A paper by Marc Tanti and Albert Gatt**,** Institute of Linguistics and Language Technology, University of Malta covered a comparison study, of all the approaches.

Let's look into the approaches-

## 4.1 Types of Architectures

There are two basic types of architectures:




(a) Conditioning by injecting the image means injecting the image into the same RNN that processes the words.

(b) Conditioning by merging the image means merging the image with the output of the RNN after processing the words.

**Fig.4.1:** types of architecture

The first architecture is called **Injecting Architecture** and the second one is called **Merging architecture**. FF shows feed-forward networks.

In the Injecting Architecture, the image data is introduced along with the language data, and the image and language data mixture is represented together. The RNN trains on the mixture.

So, at every step of training, the RNN uses the mixture of both pieces of information to predict the next word, and consequently, the RNN finetunes image information as well during training.

In the Merging Architecture, the image data is not introduced in the RNN network. So, the image and the language information are encoded separately and introduced together in a feed-forward network, creating multimodal layer architecture.

Overall in general, there are 4 approaches to this problem as described by the paper. The injecting methodology has three variants.

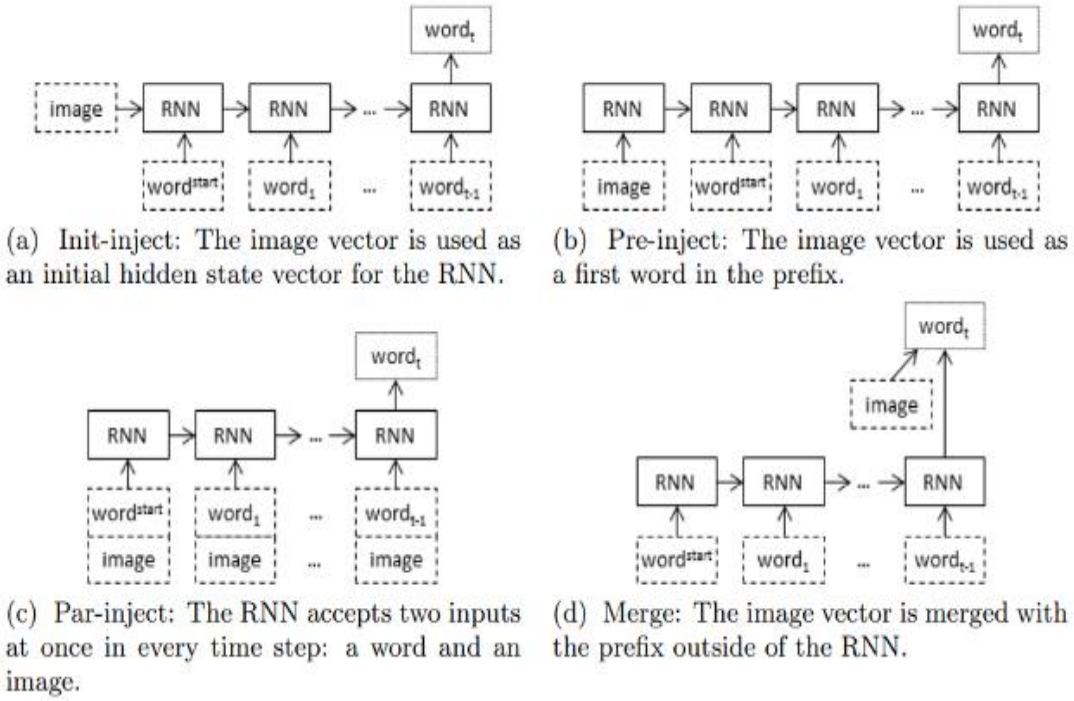



**Fig.**4.2: Injecting methodology

**Init-Inject:** Normally in the case of an RNN we use an initial state vector which is set to a zero vector of the given dimension. In the case of Init-inject, we obtain the Image feature vector using the CNN network, as the same size of the RNN hidden state vector and pass the Image vector as the initial state of the RNN.

**Pre-Inject:** In the case of the Pre-inject architecture; we pass the image vector as the 1st word. For RNN, at each step, we need to send the words in encoded vector versions at each time step as a sequence. So, for the first time step, we send the image vector.

**Par-Inject:** In this case, at every step, we merge the word vector and the image vector into a similar-sized embedding space and pass it to be trained by the RNNs.

In all these cases, the image feature vectors are generated by architectures like Inception Networks and complex Resnet structures. One thing common in all the Inject architectures is, the hidden state of the RNN is affected by the image vector. This is the point of difference between the inject and merge architectures.

According to tested works, pre-inject works better than par-inject. Several studies have also proven that merging architectures works better than injecting architectures for some cases. Most studies use Bidirectional RNNs and LSTMs for better results. Pre-trained embeddings like Glove and word2vec are used in the studies.

Now, having seen the possible kinds of architectures, let's look at some most famous architectures proposed and how they work.

## Andrej Karpathy's Proposed Architecture

First, let's talk about the architecture introduced by **Andrej Karpathy** in his Ph.D. I think it is literally the most talked-about approach. I personally like his approach because it is so similar to the way we as humans will describe a scene.

Karpathy's work mostly contributed to two areas:

**Agenda 1**

Suppose, we as humans are describing the scene, given above (Fig 1). So, we first try to recognize the objects in the image, like the ball, the dog, and the towel. Then we create the description, which obviously has the words representing those objects. Karpathy focussed on the same logic. He pointed out that every continuous segment of words in the description corresponds to a particular position in the image, but obviously to the machine the positions are unknown. So, his work used these mappings to create a description generation model. Now, if we think of it, we as humans also create chunks of words seeing some important parts of the image having objects and then move to form the sentence. The approach given by the paper is similar.

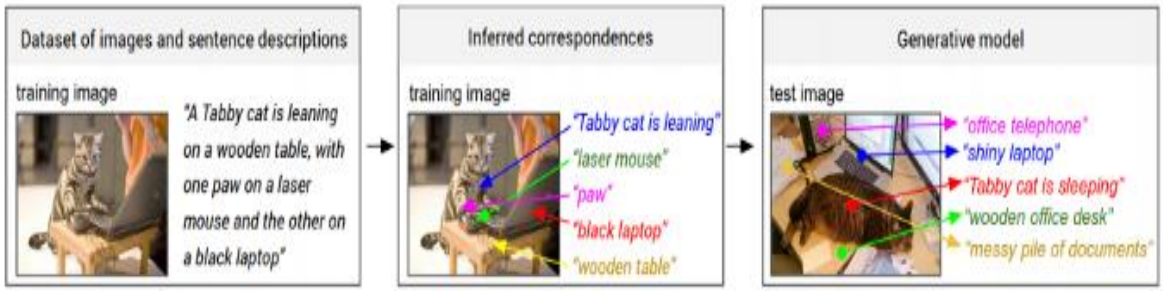



Figure 2. Overview of our approach. A dataset of images and their sentence descriptions is the input to our model (left). Our model first infers the correspondences (middle, Section 3.1) and then learns to generate novel descriptions (right, Section 3.2).

**Fig.4.3: Overview of our approach**

The model proposed created a multimodal embedding space using the two modalities to find the alignments between the segments of the sentences and their corresponding represented areas in the image. RCNN or Regional Convolutional Neural Networks were used in the model to detect the object region and a CNN trained on the ImageNet dataset having 200 classes was used to recognize the objects. For language modelling, The paper suggests the use of BRNN or Bidirectional Neural Networks as it best represents the inter-modal relationships among the n-grams of the sentences. A 300d word2vec embedding is used for obtaining the word vectors.

The paper established an image-sentence score to obtain a correlation between the image and the sentence. It is a function of the individual word and image region score. If the latter is high, then the words are said to have high support for the image.

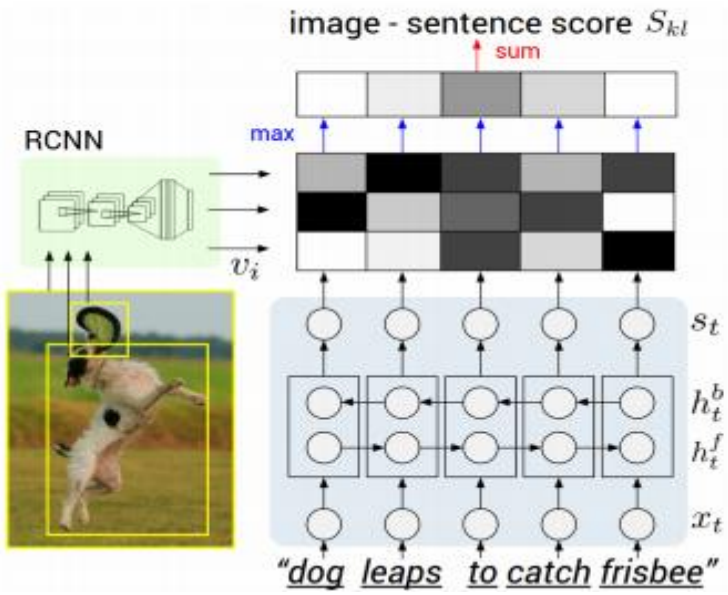



**Fig.4.4:** Image sentence correlation

The above diagram expresses the approach. The RCNN basically creates a bounding box, so if we regard it as the i-th region of the image, its confidence is matched with every t-th word in the description. So, for every region and word pair, the dot product $V_T$'s between the i-th region and t-th word is a measure of similarity.

$$S_{kl} = \sum_{t \in g_l} max_{i \in g_k} v_i^T s_t.$$

The formula defines the total sentence score.

**Agenda 2:**

The second part deals with the development of a Multimodal Recurrent Neural Network for generating captions.

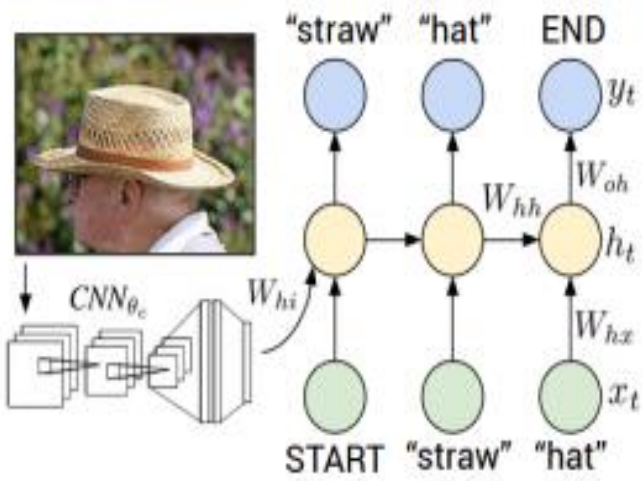



**Fig.4.5:** image sentence correlation

The above image shows the proposed model. The model takes in the image pixels I and sequence of input word vectors (x1, x2….., xn) and calculates the sequence of hidden states (h1,h2……., hn) to give the sequence of outputs (y1,y2,….., yn). The image feature vectors are passed only once as the initial hidden state. So, the next hidden state is calculated from the image vector I, the previously hidden state ht-1, and the current input $x_t$.

The current output $y_t$ is produced by using a softmax layer on the given hidden state activation functions.

$$b_v = W_{hi}[CNN_{\theta_c}(I)]$$
$$h_t = f(W_{hx}x_t + W_{hh}h_{t-1} + b_h + \mathbb{1}(t=1) \odot b_v)$$
$$y_t = softmax(W_{oh}h_t + b_o).$$

The above set of equations govern the model.

The above discussed were the contributions of Andrej Karpathy's paper. I will be providing the references for all the material used in this post below.

Next, let's talk about two of the other most commonly used architectures.

## 4.2 Google's Architecture

The architecture was proposed in a paper titled "**Show and Tell: A Neural Image Caption Generator**" by Google in 2k15.

The architecture by Google uses LSTMs instead of plain RNN architecture.

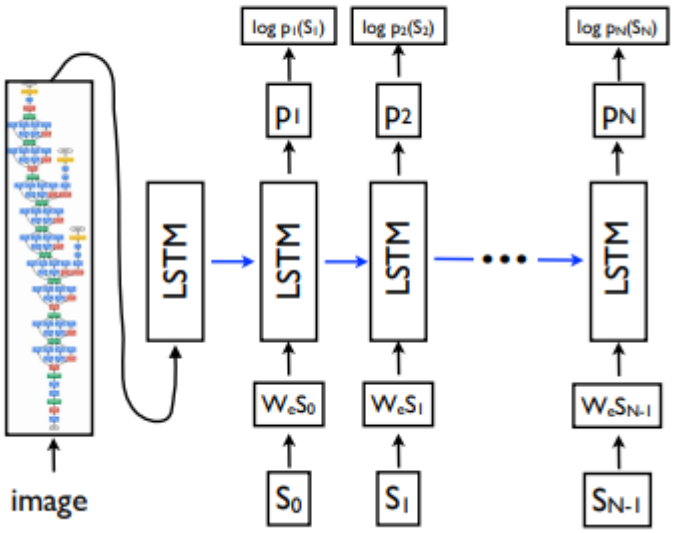



**Fig.4.6 :** Working of the image captioning model

The above image shows the architecture. The other parts of the functioning are similar to the functions of the model introduced by Karpathy. The image feature vector 'I' is inserted into the LSTM sequence at t=-1, only once. After that from t=0, the sequence of word vectors are

input. The final output word of each step is the word with maximum probability obtained from the activation function of the hidden state at that particular step. The governing equations are given below.

$$
\begin{aligned}
x_{-1} &= \mathrm{CNN}(I) \\
x_t &= W_e S_t, \quad t \in \{0 \ldots N-1\} \\
p_{t+1} &= \mathrm{LSTM}(x_t), \quad t \in \{0 \ldots N-1\}
\end{aligned}
$$

For obtaining the best prediction for the captions, the Architecture uses a **Beam Search** method. The method considers k best sentences as candidates at each time step t. At every time step t+1, k most probable words are taken and with k sentences and k words, $k^2$ probable sentences arise, from them again k best probable sentences are chosen for the candidates of time step t+1.

## 4.3 Microsoft's Architecture

Microsoft proposed its architecture in the paper titled "**Rich Image Captioning in the Wild**". The architecture was taken one step further from the two architectures discussed above. The architecture is inspired by machine translation's encoder-decoder framework architecture. The paper stated as most of the image captioning architecture provides generic content without identifying entities like famous landmarks, celebrities, etc., their model will take into consideration all these factors separately.

Microsoft architecture uses deep Resnet implementation in the convolutional networks, and detect if the image is of some famous celebrity or landmark. If identified, that serves as an input feature to the model along with the image vector itself.

The main component of the model includes

1. A deep Resnet based model for image feature extraction
2. A language model for caption candidate generation and ranking
3. An entity recognition for landmark and celebrities
4. A classifier to estimate the confidence score.

The last point is another modification by Microsoft. Often during captioning, the image becomes too hard for generating a caption. In such situations, the confidence score mechanism helps to determine how much can we depend on the generated captions.

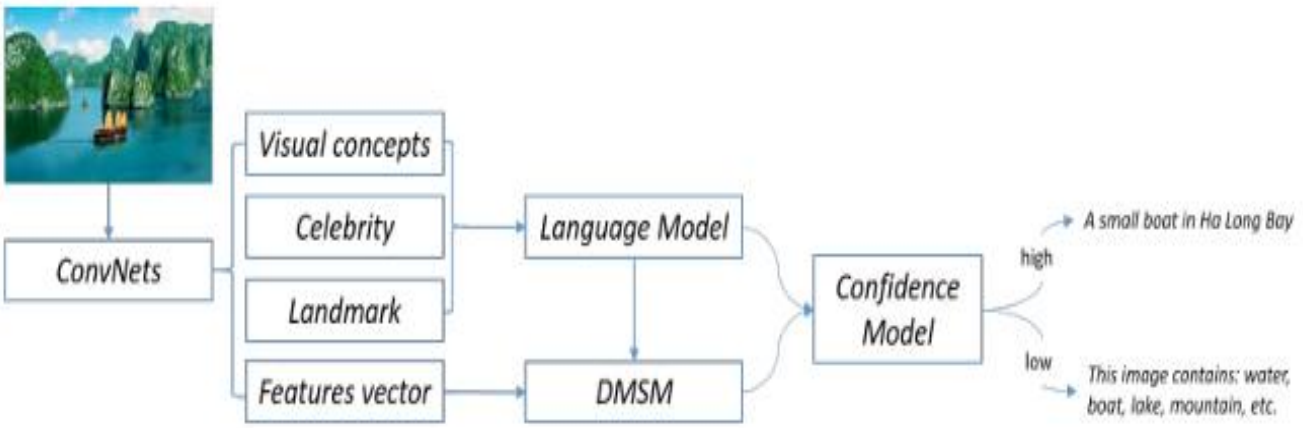



**Fig.4.7:** Illustration of our image caption pipeline

The above image gives the visualization of Microsoft's pipeline.

## 4.4 A Practical Implementation on Flickr 8K dataset

The Flickr 8k dataset has 8091 images and for each image, there are 5 descriptions. The dataset can be found at the University of Illinois site.

I have used merging architecture and I have created my own convolutional network though normally the image feature extractions are done using pre-trained CNN architectures using transfer learning.

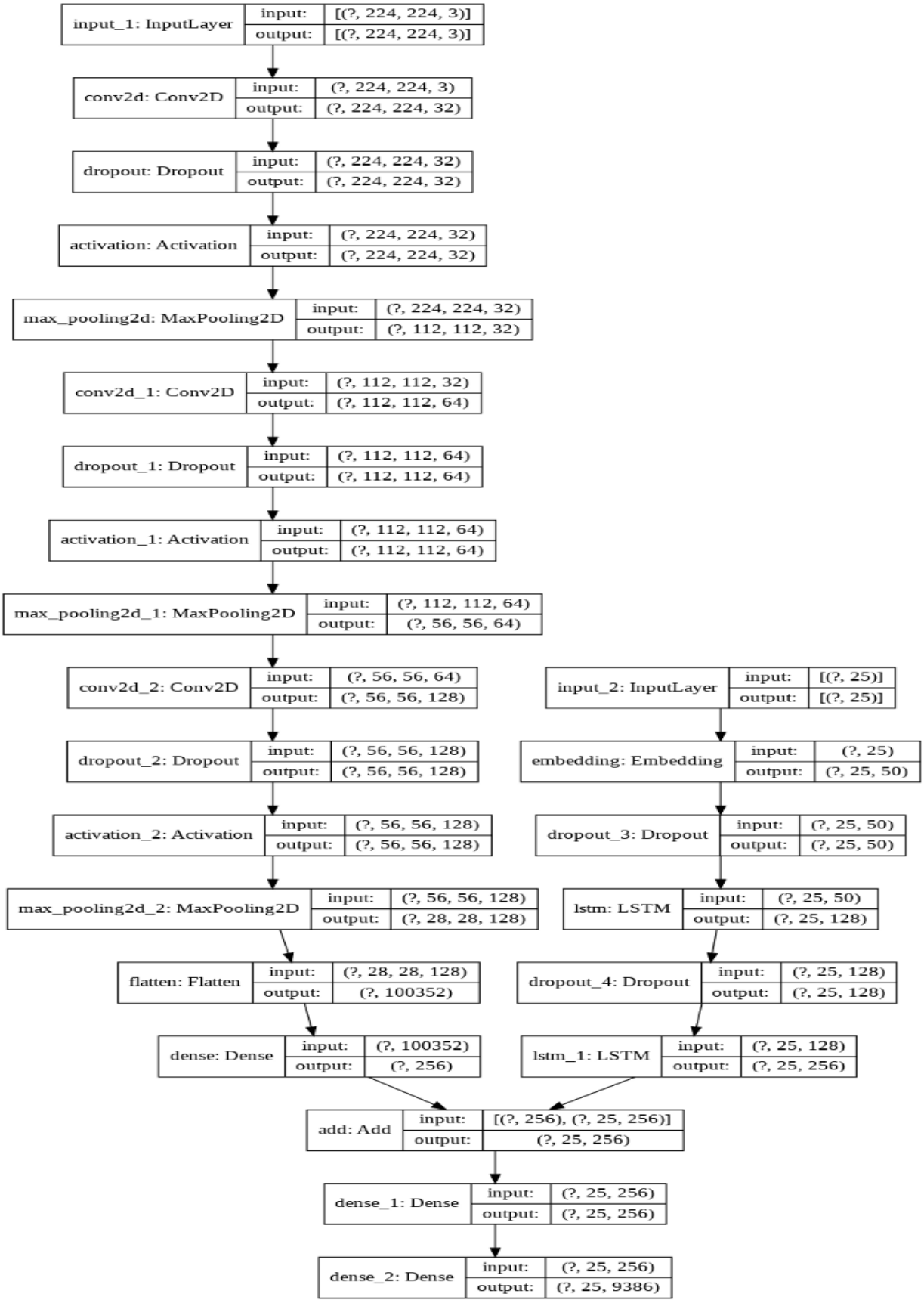



**Fig.4.8:** Model Architecture

The above is the architecture I used. As you can see I have not trained the RNN on the image features, so it is a merging architecture.

I have created my own tokenizer mapping by the name “word_map.json” for the RNN training and used Glove-50d for the embedding layer.

## 4.5 MY MODEL IMPLEMENTATION

### 4.5.1. Data Collection

There are many open source datasets available for this problem, like Flickr 8k (containing8k images), Flickr 30k (containing 30k images), MS COCO (containing 180k images), etc.

But for the purpose of this case study, I have used the Flickr 8k dataset which you can download 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

**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≤i≤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 .']

### 4.5.2. 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"e*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 = **1652** (one index for the 0).

### 4.5.3 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.

### 4.5.4. 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 49oftMax layer from the model and extract a 2048 length vector (**bottleneck features**) for every image as follows:

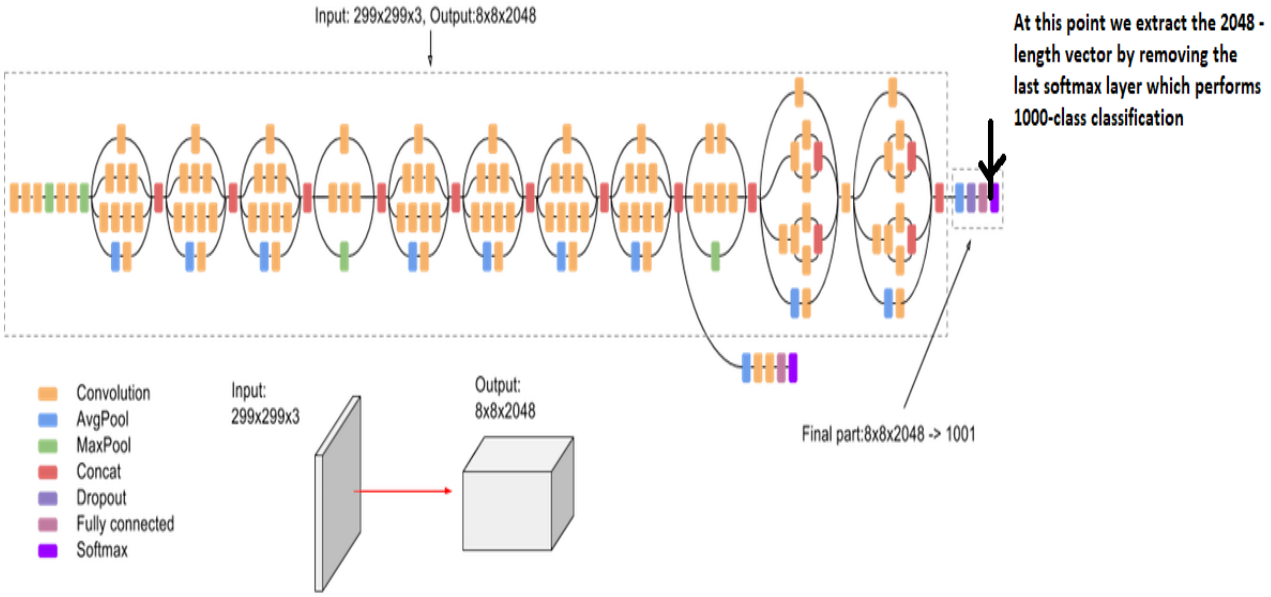



**Fig.4.9:** Feature Vector Extraction (Feature Engineering) from InceptionV3

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**".

### 4.5.5. 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.

### 4.5.6. 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

**Fig.4.10:** Train image's captions

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 = {Xi, Yi}, where Xi is the feature vector of data point 'i' and Yi is the corresponding target variable.

Let's take the first image vector Image_1 and its corresponding caption "startseq the black cat sat on grass endseq". Recall that, Image vector is the input and the caption is what we need to predict. But the way we predict the caption is as follows:
For the first time, we provide the image vector and the first word as input and try to predict the second word, i.e.:
Input = Image_1 + 'startseq'; Output = 'the'
Then we provide image vector and the first two words as input and try to predict the third word, i.e.:
Input = Image_1 + 'startseq the'; Output = 'cat'
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 |
| | | | |

**Table 4.1:** 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 | |

**Table 4.2:** 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:

| | Xi | | Yi |
|---|---|---|---|
| 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 |

**Table 4.3**: 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 0's** (zero padding) at the end of each sequence. But **how many** zeros should we append in each sequence?

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

The data matrix will then look as follows:

| | 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 |

**Table 4.4:** 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
Size of the data matrix = n*m
Where n-> number of data points (assumed as 210000)
And m-> length of each data point
Clearly m= Length of image vector(2048) + Length of partial caption(x).
m = 2048 + x

But what is the value of x?
Well you might think it is 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.

## 4.5.7. 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 out 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

### 4.5.8. 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:

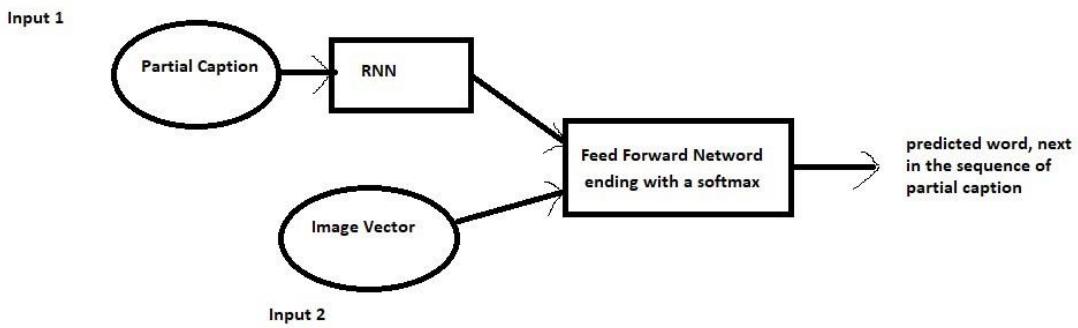



**Fig.4.11:** High level architecture

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 #   
=================================================================
input_4 (InputLayer)         (None, 34)                0         
_________________________________________________________________
input_3 (InputLayer)         (None, 2048)              0         
_________________________________________________________________
embedding_2 (Embedding)      (None, 34, 200)           330400    
_________________________________________________________________
dropout_3 (Dropout)          (None, 2048)              0         
_________________________________________________________________
dropout_4 (Dropout)          (None, 34, 200)           0         
_________________________________________________________________
dense_2 (Dense)              (None, 256)               524544    
_________________________________________________________________
lstm_2 (LSTM)                (None, 256)               467968    
_________________________________________________________________
add_2 (Add)                  (None, 256)               0         
_________________________________________________________________
dense_3 (Dense)              (None, 256)               65792     
_________________________________________________________________
dense_4 (Dense)              (None, 1652)              424564    
=================================================================
Total params: 1,813,268
Trainable params: 1,813,268
Non-trainable params: 0
```

**Fig.4.12:** 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:

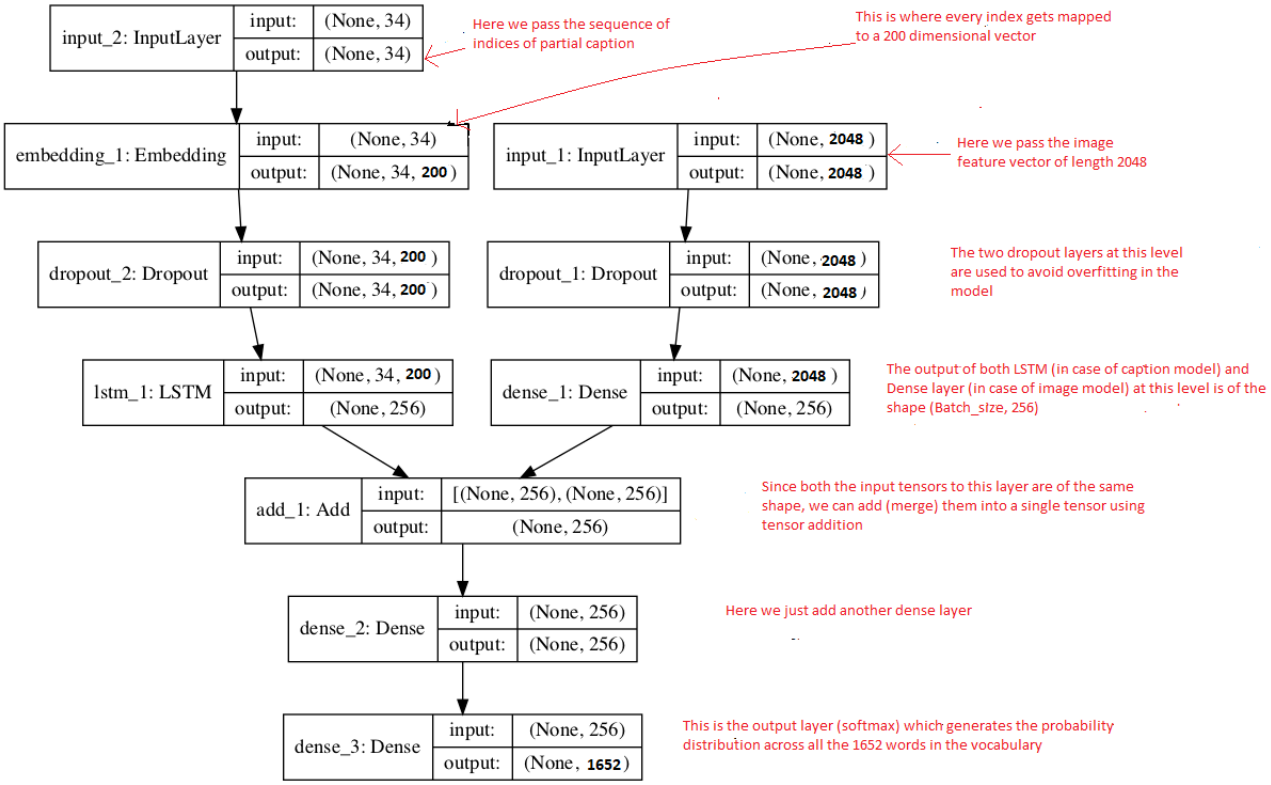



**Fig.4.13:** Flowchart of the architecture

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.

### 4.5.9. 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:


**Fig.4.14:** Test image

Caption -> the black cat is walking on grass
Also the vocabulary in the example was:

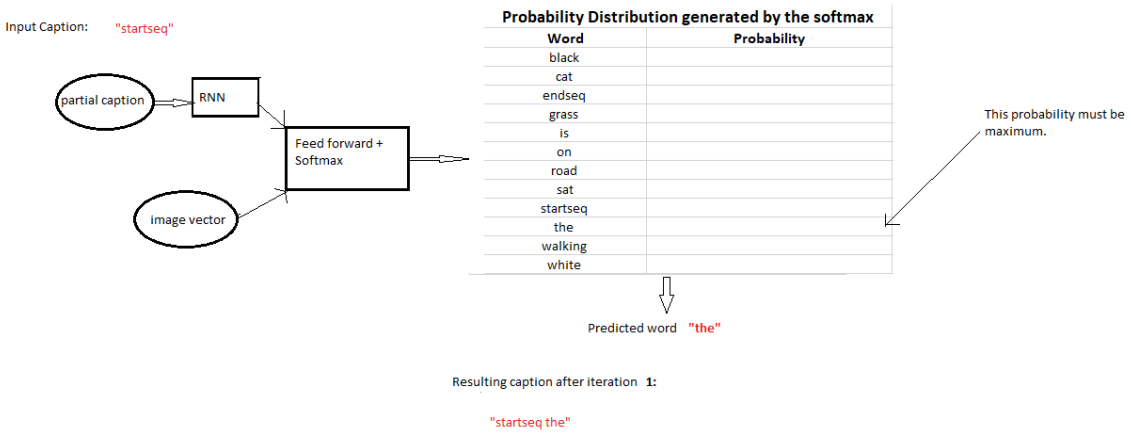
vocab = {black, cat, endseq, grass, is, on, road, sat, startseq, the, walking, white}
We will generate the caption iteratively, one word at a time as follows:

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:



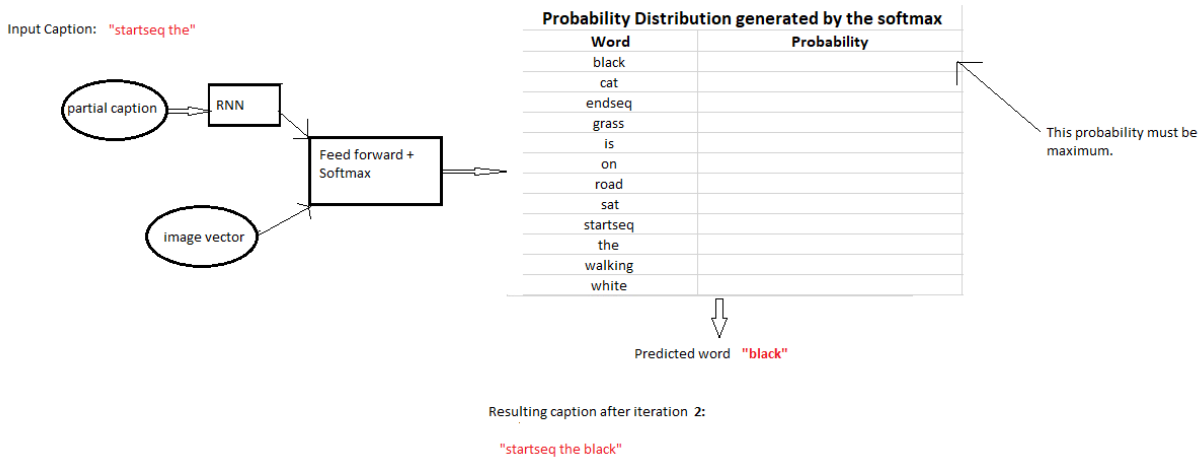

Iteration 1

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”

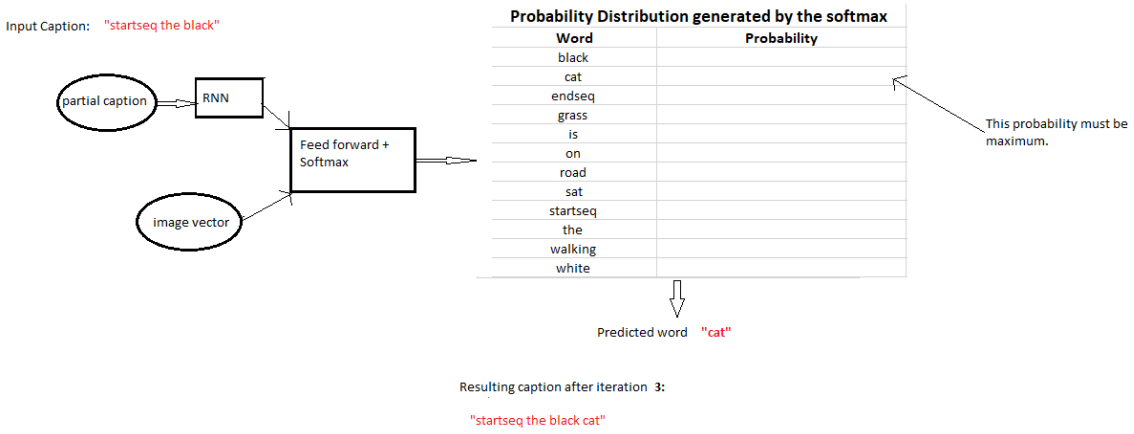



Iteration 2

Iteration 3:

Input: Image vector + “startseq the black”

Expected Output word: “cat”

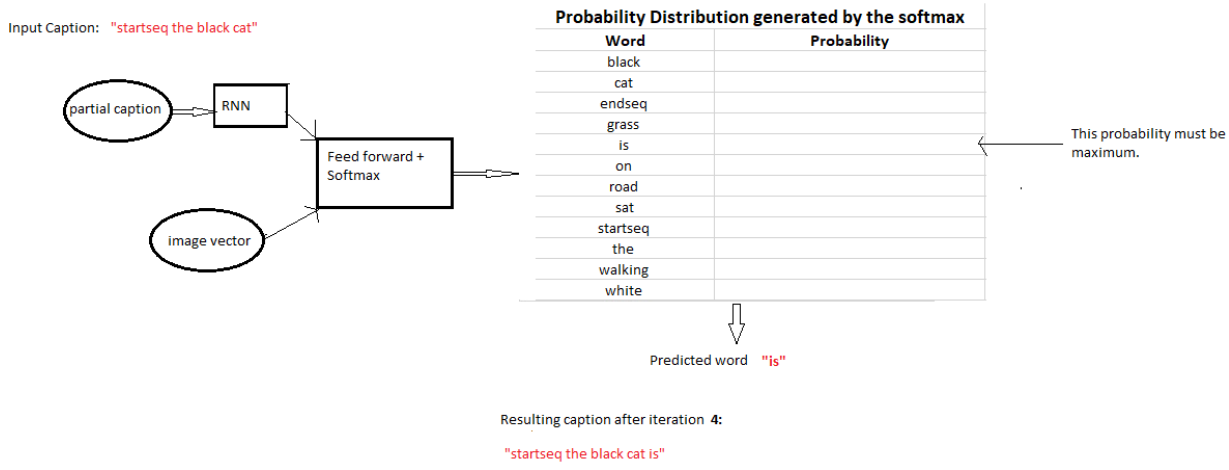



Iteration 3

Iteration 4:

Input: Image vector + “startseq the black cat”

Expected Output word: “is”

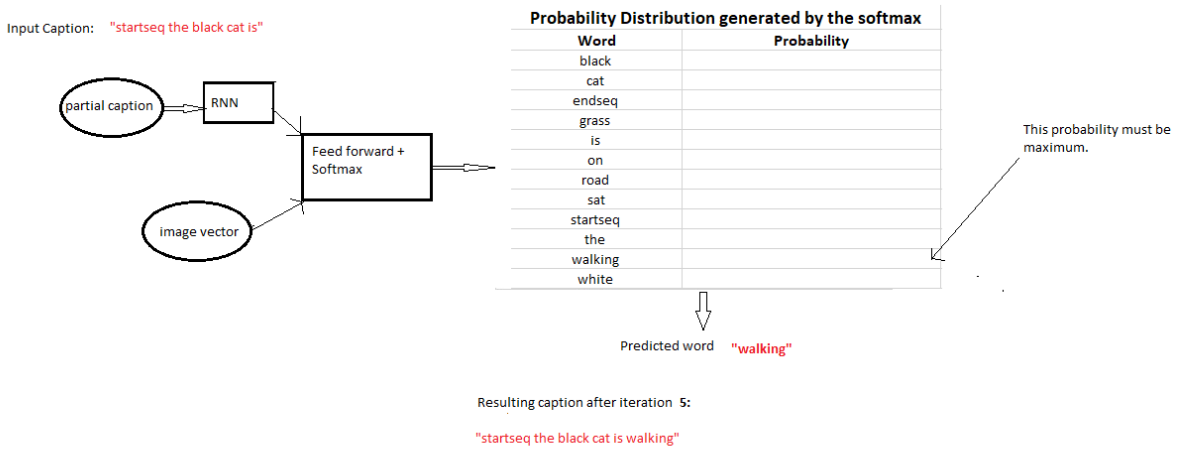



Iteration 4

**Iteration 5:**

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

Expected Output word: “walking”

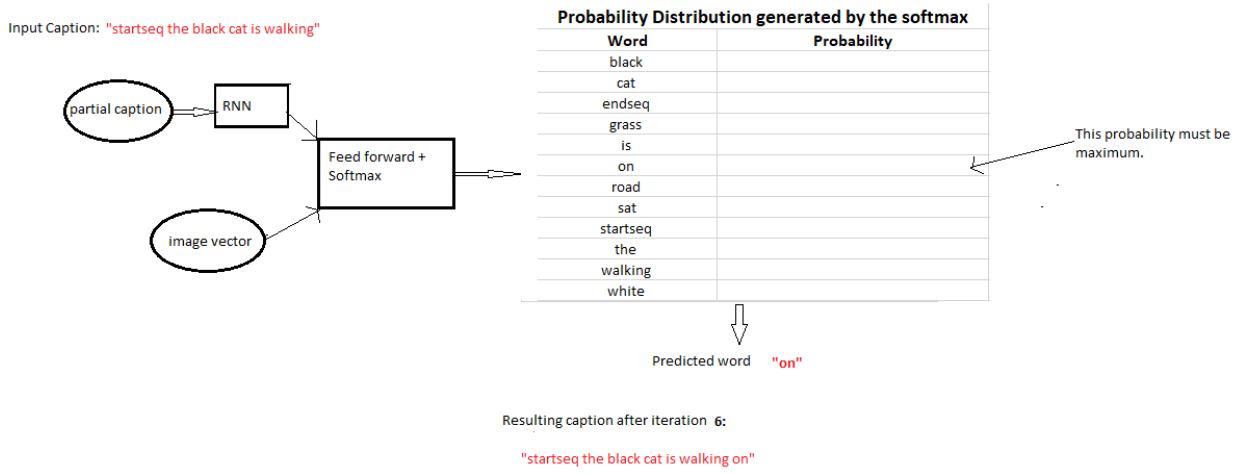



Iteration 5

**Iteration 6:**

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

Expected Output word: “on”

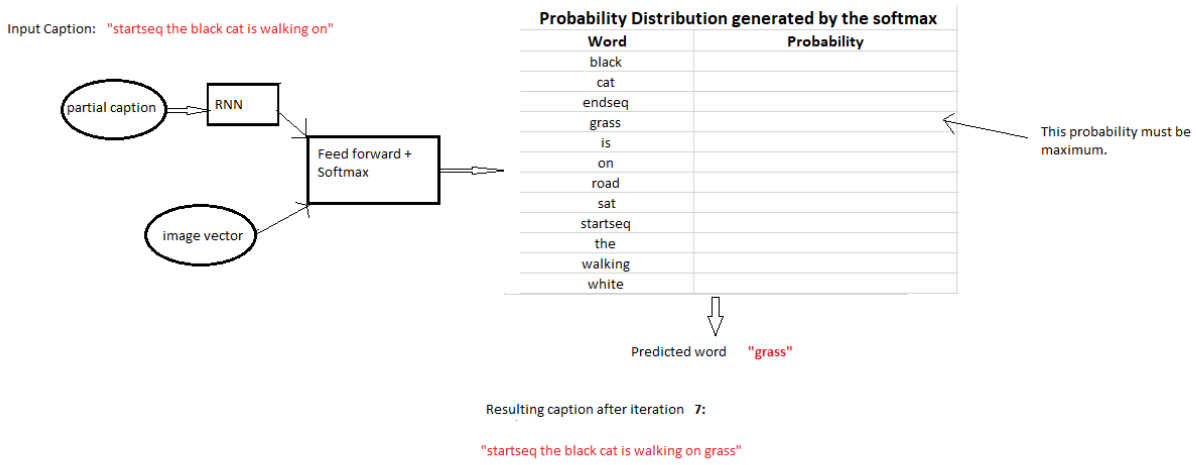



Iteration 6

**Iteration 7:**

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

Expected Output word: “grass”

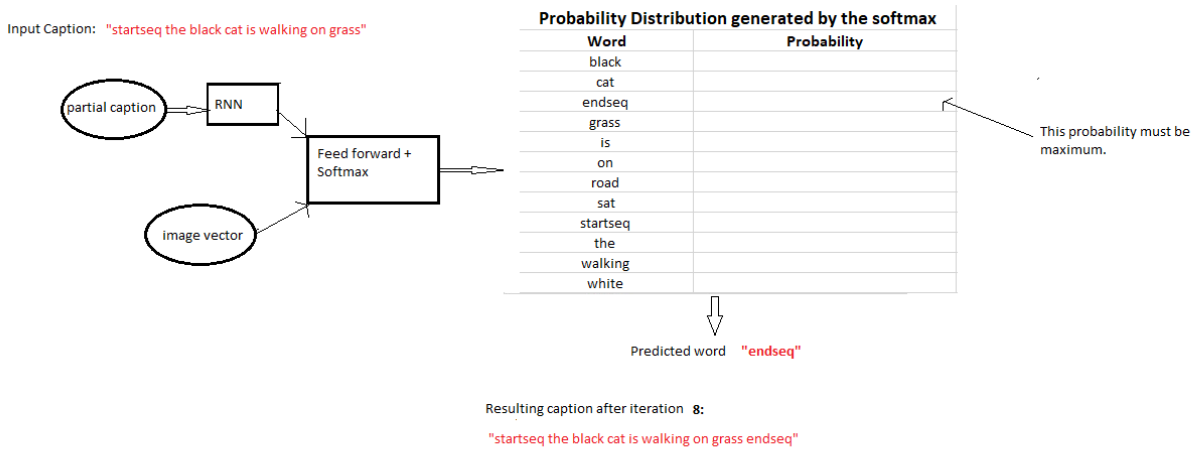



Iteration 7

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.

# CHAPTER 5

# TESTING AND RESULTS

## 5.1 Testing Approach

Our dataset FLICR8K contains 8000 datasets . we have divided the datasets into two parts of 6000 and 2000 . 6000 images is used for Training and the remaining 2000 datasets are used for testing and validation purpose.

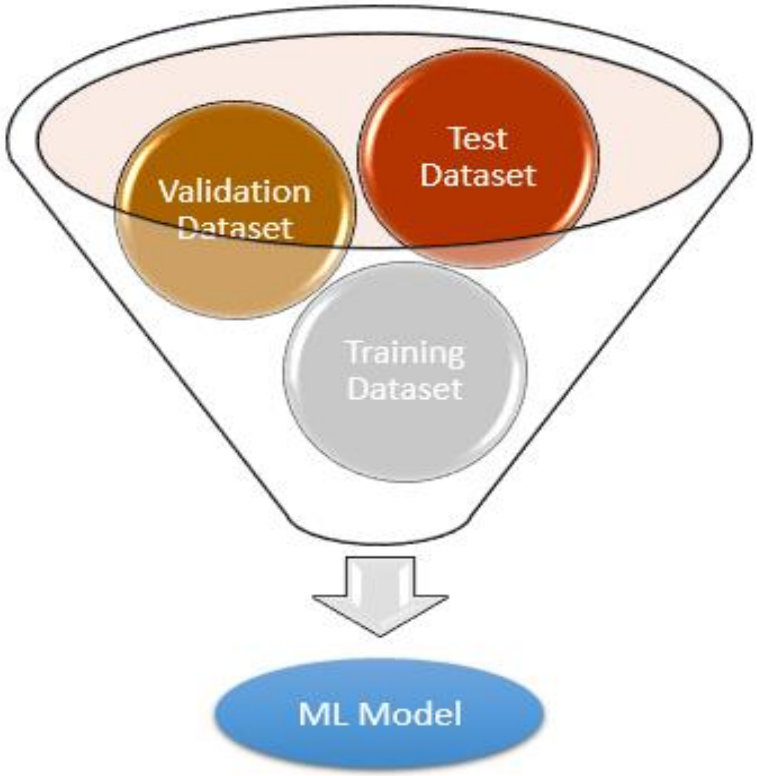



**Fig.5.1:** Destribution of dataset for Testing Purpose.

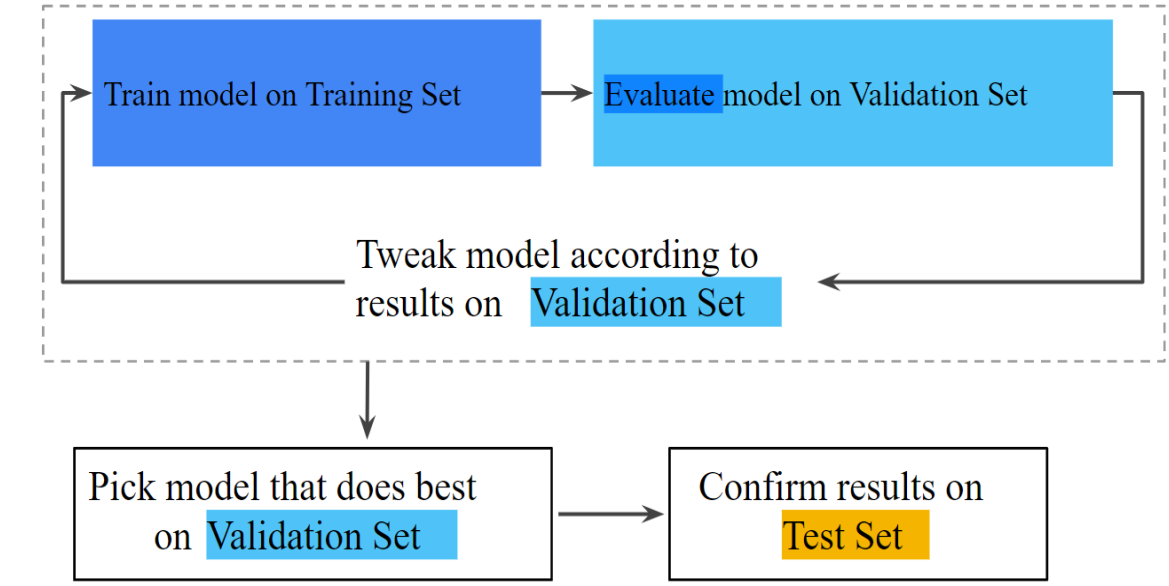



**Fig.5.2:** Testing process Overview

### 5.2 Accuracy Measurement

Accuracy Measurement is important to analyse how our model performs. In our project we

There is an important need to understand the fact that the captions generated can be subjective to everyone . Our Model can predict captions which are different from the test and training data and it still can be valid.

For true accuracy measurement we need to analyse the similarity of the captions generated and the captions present in our datasets.

### 5.3 Method For Accuracy Measurement

As we need to observe the similarity of the captions generated to the captions present in the dataset we have used **COSINE SIMILARITY** for accuracy calculation.

Cosine similarity is a measure of similarity between two non-zero vectors of an inner product space that measures the cosine of the angle between them.

**Similarity** = (A.B) / (||A||.||B||) where A and B are vectors.

We have 5 captions per image , Therefore for testing accuracy thus We have taken Maximum Cosine Similarity from all 5 captions

```
[ ]:
accuracy_list = list()
vt_idx = .32
# Pick Some Random Images and See Results
plt.style.use("seaborn")
for i in range(100):
    idx = np.random.randint(0,1000)
    all_img_names = list(encoding_test.keys())
    img_name = all_img_names[idx]
    photo_2048 = encoding_test[img_name].reshape((1,2048))

    i = plt.imread("../input/flickr8k/Images/"+img_name+".jpg")

    caption = predict_caption(photo_2048, img_name)
```

+ Code + Markdown

```
[ ]:
model_accuracy = (sum(accuracy_list)/len(accuracy_list))
```

```
[4]:
print(model_accuracy)
```

```
67.56851201
```

Console

**Fig.5.3:** After performing several tweaks and optimisations we were able to get an acceptable accuracy of 67.5 % on 2000 test datasets .

### 5.4 Some Results generated by Model

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**

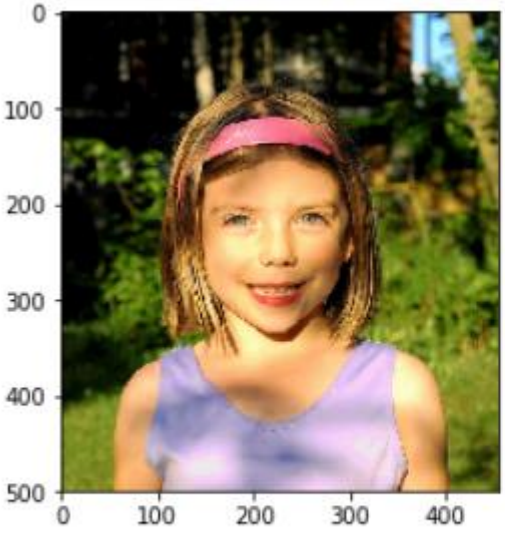
Note: We must appreciate how the model is able to identify the colors precisely.


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

**Output — 2**

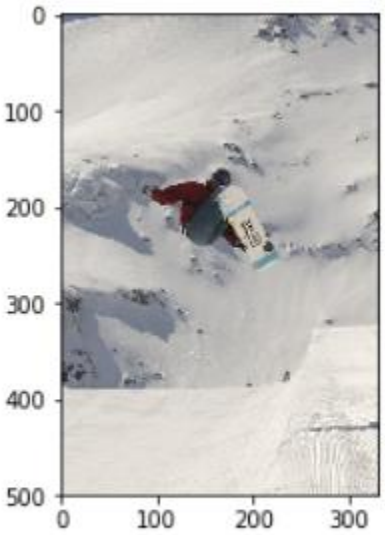

```
15 Greedy: race car spins down the road as spectators watch
```

**Output — 3**




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

**Output — 4**




```
Greedy: man in red jacket snowboarding
```

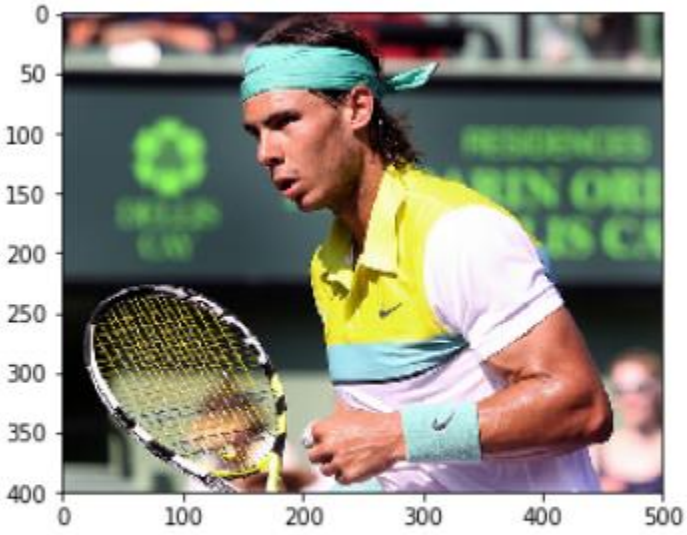
**Output — 5**

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 .
```

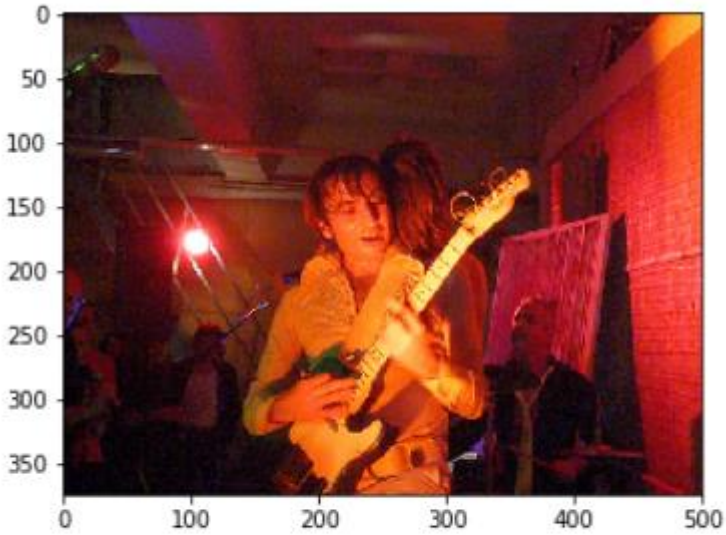
**Output — 7**

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 — 8**

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**

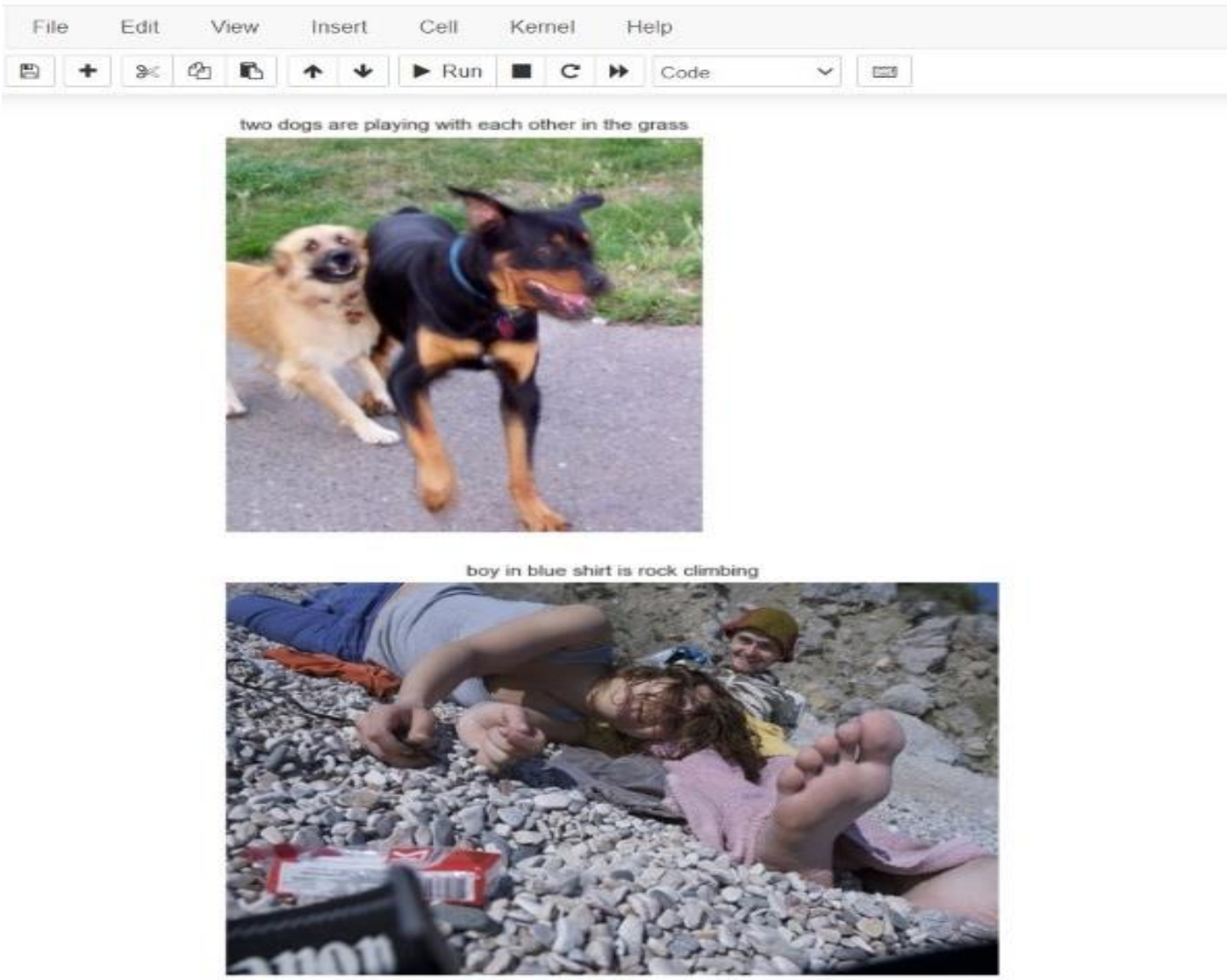
**Fig.5.4:** Some Results generated by Model (output 1-10)

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

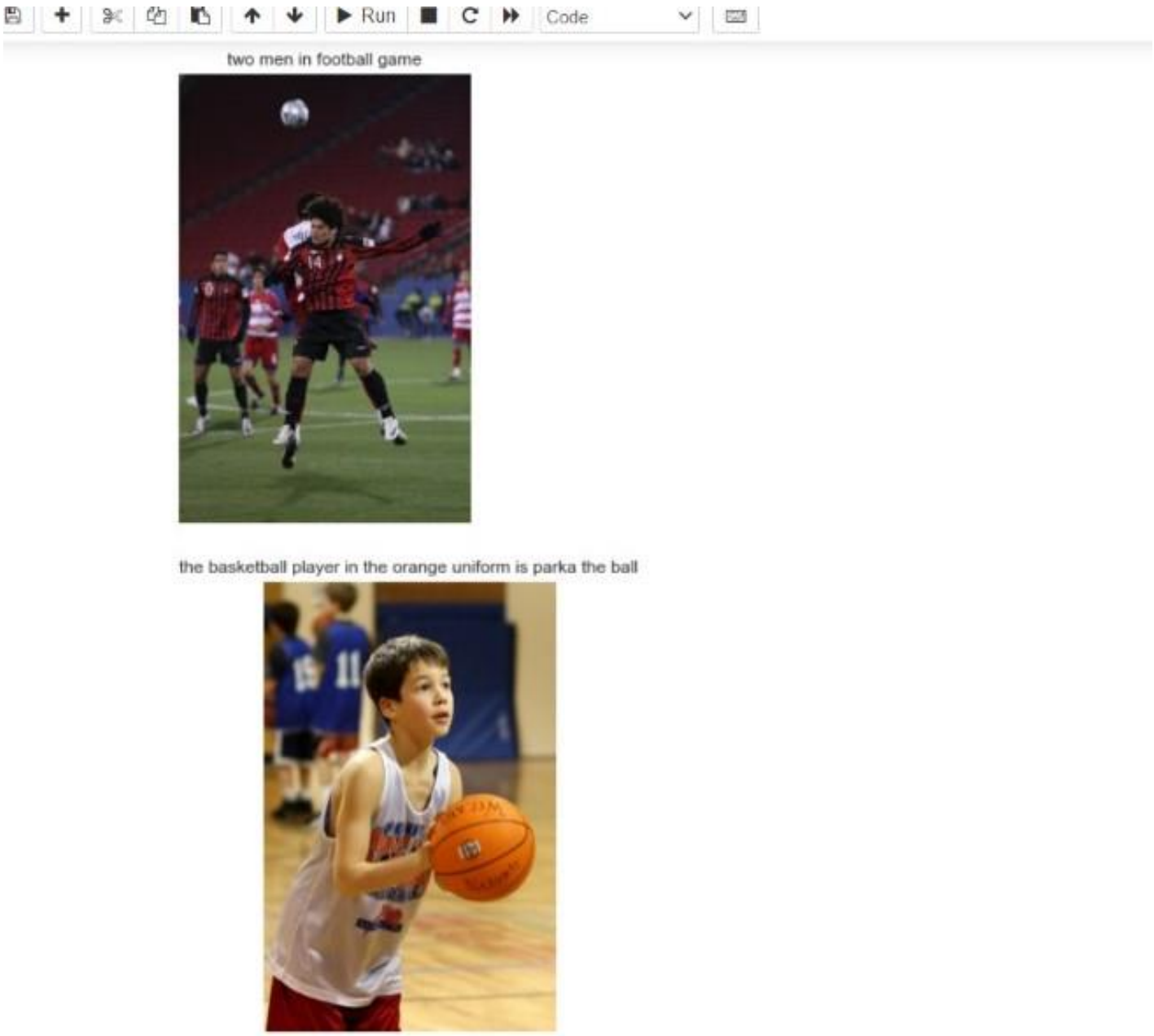
**Important Point:**

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.

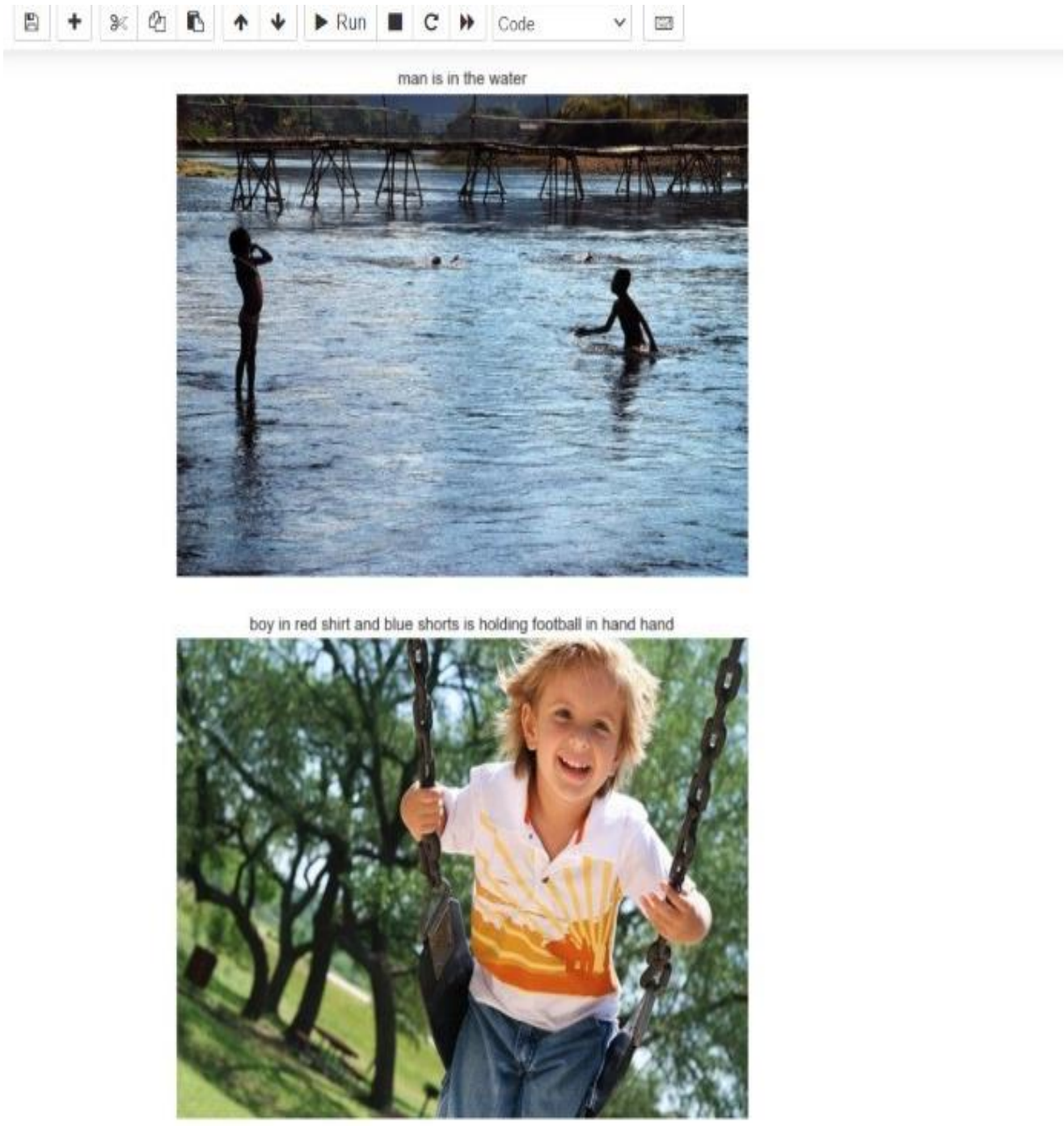
## 5.5 Some More Results


File
Edit
View
Insert
Cell
Kernel
Help
Run
Code
two dogs are playing with each other in the grass
boy in blue shirt is rock climbing

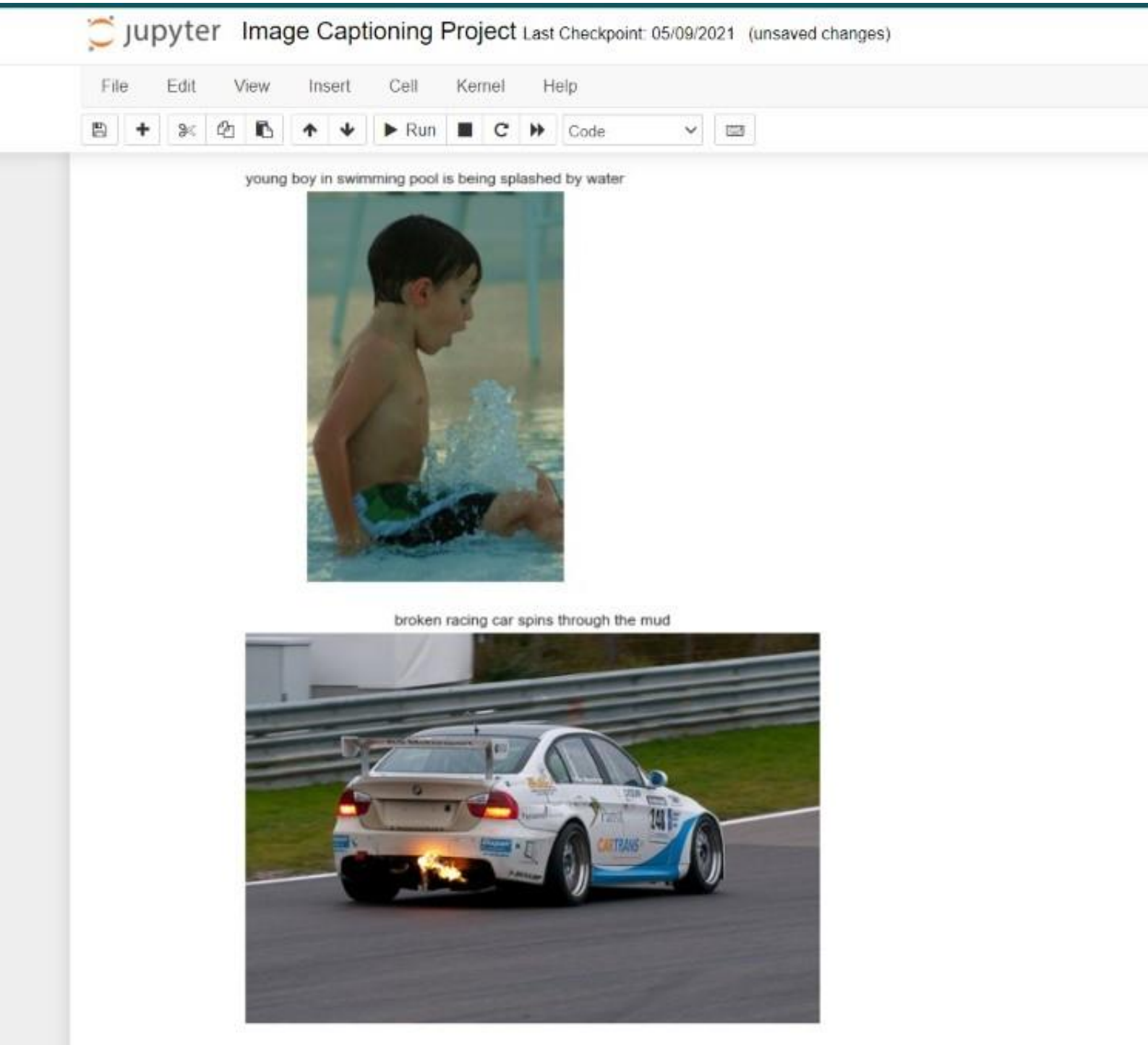
**Output 1**


Run
Code
two men in football game
the basketball player in the orange uniform is parka the ball

**Output 2**


Run
Code
man is in the water
boy in red shirt and blue shorts is holding football in hand hand

**Output 3**




**Output 4**

**Fig.5.5 :** some more results (output 1-5)

# CHAPTER 6

# CONCLUSION AND FUTURE SCOPE

## 6.1 CONCLUSION

In this paper, we have presented a model, which is a neural network that can automatically view an image and generate appropriate captions in natural language like English. The model is trained to produce the sentence or description from given image.

This model is based on ResNet50 and LSTM for automatic image captioning. It is designed with one encoder-decoder architecture. We adopted ResNet50, a convolutional neural network, as the encoder to encode an image into a compact representation as the graphical features. After that, a language model LSTM was selected as the decoder to generate the description sentence. The experimental evaluations indicate that the proposed model is able to generate good captions for images automatically.

After so much of experiments, it is conclusive that use of larger datasets increases performance of the model. The larger dataset will increase accuracy as well as reduce losses.

## 6.2 FUTURE SCOPE

**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 famous applications of Deep Learning.

**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.

# REFERENCES


1. https://towardsdatascience.com/a-guide-to-image-captioning-e9fd5517f350
2. https://www.hindawi.com/journals/wcmc/2020/8909458/
3. https://www.kaggle.com/adityajn105/flickr8k/activity
4. https://medium.com/@raman.shinde15/image-captioning-with-flickr8k-dataset-bleu-4bcba0b52926
5. https://cs.stanford.edu/people/karpathy/cvpr2015.pdf
6. https://arxiv.org/abs/1411.4555https://machinelearningmastery.com/develop-a-deep-learning-caption-generation-model-in-python/
7. https://arxiv.org/abs/1708.02043
8. https://www.youtube.com/watch?v=yk6XDFm3J2c
9. http://www.wildml.com/2015/10/recurrent-neural-network-tutorial-part-4-implementing-a-grulstm-rnn-with-python-and-theano/
10. http://colah.github.io/posts/2015-08-Understanding-LSTMs/
11. A. Karpathy, A. Joulin, and L. Fei-Fei., "Deep fragment embeddings for bidirectional image sentence mapping", arXiv preprint arXiv:1406.5679, 2014.
12. Kuznetsova, Polina, Ordonez, Vicente, Berg, Tamara L, and Choi, Yejin., "Treetalk: Composition and compression of trees for image descriptions", TACL, 2(10):351â˘A ¸S362, 2014.
13. X. Chen and C. L. Zitnick., "Learning a recurrent visual representation for image caption generation", CoRR, abs/1411.5654, 2014.
14. Hochreiter, Sepp and Schmidhuber, and J Ì´Lurgen, "Long Short-Term Memory, In Neural Comput.", 1997.
15. M. Hodosh, P. Young, and J. Hockenmaier., "Framing image description as a ranking task: data, models and evaluation metrics", Journal of Artificial Intelligence Research, 2013.
16. P. Young, A. Lai, M. Hodosh, and J. Hockenmaier., "From image descriptions to visual denotations: New similarity metrics for semantic inference over event descriptions", TACL, 2014.
17. T.-Y. Lin, M. Maire, S. Belongie, J. Hays, P. Perona, D. Ramanan, P. DollÂt'ar, and C. L. Zitnick., "Microsoft coco: Common objects in context", arXiv preprint arXiv:1405.0312, 2014.
18. Tao Xu, Pengchuan Zhang, Qiuyuan Huang, Han Zhang, Zhe Gan, Xiaolei Huang, Xiaodong He, "AttnGAN: Fine-Grained Text to Image Generation with Attentional Generative Adversarial Networks", arXiv:1711.10485v1 [cs.CV], 2017.
19. Shikhar Sharma, Dendi Suhubdy, Vincent Michalski, Samira Ebrahimi Kahou, Yoshua Bengio, "ChatPainter: Improving Text to Image Generation using Dialogue", arXiv:1802.08216v1 [cs.CV], 2018.
20. Richard Socher, Andrej Karpathy, Quoc V. Le*, Christopher D. Manning, Andrew Y. Ng, " Grounded Compositional Semantics for Finding and Describing Images with Sentences", TACL, 2014.