Pattern Recognition (EC-416)

Innovative Project Report

Submitted to: - Dr. Rajiv Kapoor

Submitted by: -

Name: - Aditya Agrawal

Roll No: - DTU/2K17/EC/008

Table of Contents

Image Caption Generator using ResNet50 model and Neural Networks.

Abstract

Image caption generator is a task that involves computer vision and natural language processing concepts to recognize the context of an image and describe them in a natural language like English. Generating a caption for a given image is a challenging problem in the deep learning domain.

In this project, different techniques of computer vision and NLP are used to recognize the context of an image and describe them in a natural language like English. The dataset used for training is the Flickr8k dataset and encoding of the images for training purposes is done using the pre-trained model called ResNet50 model. The working model of the image caption generator is built by using CNN (Convolutional Neural Networks) and LSTM (Long short term memory) units. The code is tested on various images and their captions are generated. The project is run on Kaggle GPU to have faster model training. The parameter setting and results are presented in this report.

Introduction

With the rapid development of digitalization, there are a huge number of images, accompanied with a lot of related texts. Automatic image captioning has recently attracted much research interest. The objective of automatic image captioning is to generate properly formed English sentences to describe the content of an image automatically, which is of great impact in various domains such as virtual assistants, image indexing, recommendation in editing applications, and the help of the disabled. Although it is an easy task for a human to describe an image, it becomes exceedingly difficult for a machine to perform such a task. Image captioning does not only need to detect the objects contained in an image but also capture how these objects related to each other and their attributes as well as the activities involved in. Moreover, the semantic knowledge should be expressed in a natural language, which requires a language model to be developed based on the visual understanding.

Much research effort has been devoted to automatic image captioning, and it can be categorized into template-based image captioning, retrieval-based image captioning, and novel image caption generation.

1. Template-based image captioning first detects the objects/attributes/actions and then fills the blanks slots in a fixed template.
2. Retrieval-based approaches first find the visually similar images with their captions from the training dataset, and then the image caption is selected from similar images with captions. These methods can generate syntactically correct captions but are unable to generate image-specific and semantically correct captions.
3. The novel image caption generation approaches are to analyse the visual content of the image and then to generate image captions from the visual content using a language model.

Compared to the first two categories, novel caption generation can generate new captions for a given image that are semantically more accurate than previous approaches. Most of the works in this category rely on machine learning and deep learning, which is also the approach adopted in this project. One common framework used in this category is the encoder-decoder framework for image captioning. This framework was first introduced to describe a multimodal log-bilinear model for image captioning with a fixed context window by Kiros et al. [2]. Recent research works have used the deep convolutional neural network (CNN) as the encoder and the deep recurrent neural network (RNN) as the decoder, which is proven to be promising [1, 3, 4]. However, it remains challenging to identify the proper CNN and RNN models for the image captioning.

In this project, an automatic image caption generator is built using ResNet50 (a convolutional neural network) and LSTM (long short-term memory). The generator consists of an encoder and a decoder. ResNet50 is adopted as the encoder to create an extensive representation of an input image by embedding it into a vector. Meanwhile, the LSTM is utilized as the decoder which selectively focuses the attention over a certain part of an image to predict the next sentences. The structure of the model is empirically and experimentally determined and the model hyperparameters are fine-tuned. Experimental evaluation of the generator indicates that the model is effective to generate proper captions for the images.

The rest of the paper is organized as follows: Theory of the concepts used is shown. Then the methodology of the creation of the project is shown with code included. Next, the results of the model on test images are shown. Finally, conclusion and references are given.

Theory

Convolutional Neural networks

Convolutional Neural networks are specialized deep neural networks which can process the data that has input shape like a 2D matrix. Images are easily represented as a 2D matrix and CNN is very useful in working with images. CNN is basically used for image classifications and identifying if an image is a bird, a plane or Superman, etc. It scans images from left to right and top to bottom to pull out important features from the image and combines the feature to classify images. It can handle the images that have been translated, rotated, scaled and changes in perspective.

Convolutional neural networks are composed of multiple layers of artificial neurons. Artificial neurons, a rough imitation of their biological counterparts, are mathematical functions that calculate the weighted sum of multiple inputs and outputs an activation value.

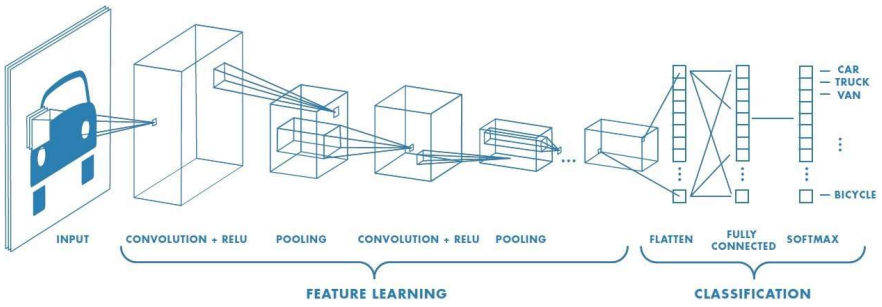


Fig. 1. Example of a Convolutional Neural Network

The behaviour of each neuron is defined by its weights. When fed with the pixel values, the artificial neurons of a CNN pick out various visual features.

When you input an image into a ConvNet, each of its layers generates several activation maps. Activation maps highlight the relevant features of the image. Each of the neurons takes a patch of pixels as input, multiplies their colour values by its weights, sums them up, and runs them through the activation function.

The first (or bottom) layer of the CNN usually detects basic features such as horizontal, vertical, and diagonal edges. The output of the first layer is fed as input of the next layer, which extracts more complex features, such as corners and combinations of edges. As you move deeper into the convolutional neural network, the layers start detecting higher-level features such as objects, faces, and more.

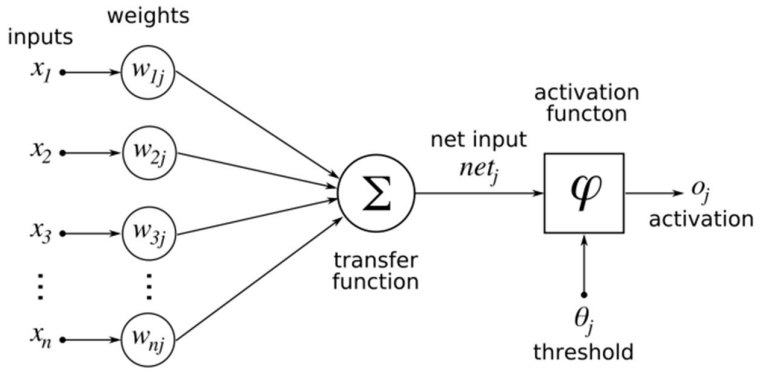


Fig. 2. The structure of an artificial neuron, the basic component of artificial neural networks

The operation of multiplying pixel values by weights and summing them is called "convolution" (hence the name convolutional neural network). A CNN is usually composed of several convolution layers, but it also contains other components. The final layer of a CNN is a classification layer, which takes the output of the final convolution layer as input (remember, the higher convolution layers detect complex objects).

Based on the activation map of the final convolution layer, the classification layer outputs a set of confidence scores (values between 0 and 1) that specify how likely the image is to belong to a "class." For instance, if you have a ConvNet that detects cats, dogs, and horses, the output of the final layer is the possibility that the input image contains any of those animals.

Long short term memory

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

This is what an LSTM cell looks like –

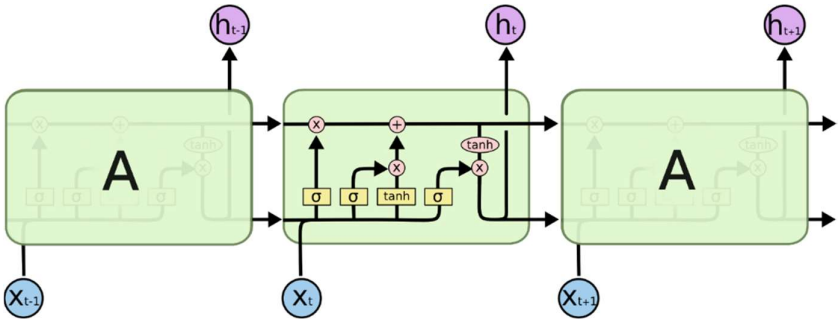


Fig. 3. The repeating module in an LSTM contains four interacting layers

All recurrent neural networks have the form of a chain of repeating modules of neural network. In standard RNNs, this repeating module will have a very simple structure, such as a single tanh layer. LSTMs also have this chain like structure, but the repeating module has a different structure. Instead of having a single neural network layer, there are four, interacting in a very special way.

The key to LSTMs is the cell state, the horizontal line running through the top of the diagram. The cell state is kind of like a conveyor belt. It runs straight down the entire chain, with only some minor linear interactions. It is very easy for information to just flow along it unchanged.

ResNet 50 Model

ResNet, short for Residual Networks is a classic neural network used as a backbone for many computer vision tasks. This model was the winner of ImageNet challenge in 2015. The fundamental breakthrough with ResNet was it allowed us to train extremely deep neural networks with 150+layers successfully. Prior to ResNet training very deep neural networks was difficult due to the problem of vanishing gradients.

AlexNet, the winner of ImageNet 2012 and the model that apparently kick started the focus on deep learning had only 8 convolutional layers, the VGG network had 19 and Inception or GoogleNet had 22 layers and ResNet 152 had 152 layers. ResNet-50 is a smaller version of ResNet 152 and frequently used as a starting point for transfer learning.

The main benefit of a very deep network is that it can represent very complex functions. It can also learn features at many different levels of abstraction, from edges (at the lower layers) to very complex features (at the deeper layers). However, using a deeper network doesn't always help. Deep networks are hard to train because of the notorious vanishing gradient problem — as the gradient is back-propagated to earlier layers, repeated multiplication may make the gradient extremely small. As a result, as the network goes deeper, its performance gets saturated or even starts degrading rapidly. This problem is solved using ResNet.

Skip Connection — The Strength of ResNet

In ResNet, a "shortcut" or a "skip connection" allows the gradient to be directly backpropagated to earlier layers:

Fig. 4. Skip Connection Image from DeepLearning.AI

The image on the left shows the "main path" through the network. The image on the right adds a shortcut to the main path. By stacking these ResNet blocks on top of each other, you can form a very deep network. Having ResNet blocks with the shortcut also makes it very easy for one of the blocks to learn an identity function. This means that you can stack on additional ResNet blocks with little risk of harming training set performance.

Two reasons why Skip connections work here:

- They mitigate the problem of vanishing gradient by allowing this alternate shortcut path for gradient to flow through.
- They allow the model to learn an identity function which ensures that the higher layer will perform at least as good as the lower layer, and not worse.

In fact, since ResNet skip connections are used in a lot more model architectures like the Fully Convolutional Network (FCN) and U-Net. They are used to flow information from earlier layers in the model to later layers. In these architectures they are used to pass information from the down-sampling layers to the up-sampling layers.

Architecture

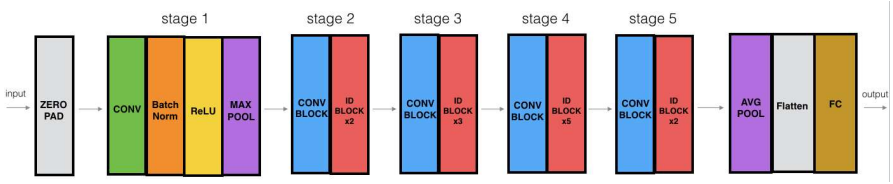


Fig. 5. ResNet-50 Model

The ResNet-50 model consists of 5 stages each with a convolution and Identity block. Each convolution block has 3 convolution layers, and each identity block also has 3 convolution layers. The ResNet-50 has over 23 million trainable parameters.

The details of this ResNet-50 model are:

- Zero-padding pads the input with a pad of (3,3)
- Stage 1:
  - The 2D Convolution has 64 filters of shape (7,7) and uses a stride of (2,2). Its name is "conv1".
  - BatchNorm is applied to the channels axis of the input.
  - MaxPooling uses a (3,3) window and a (2,2) stride.
- Stage 2:
  - The convolutional block uses three set of filters of size [64,64,256], "f" is 3, "s" is 1 and the block is "a".
  - The 2 identity blocks use three set of filters of size [64,64,256], "f" is 3 and the blocks are "b" and "c".
- Stage 3:
  - The convolutional block uses three set of filters of size [128,128,512], "f" is 3, "s" is 2 and the block is "a".
  - The 3 identity blocks use three set of filters of size [128,128,512], "f" is 3 and the blocks are "b", "c" and "d".
- Stage 4:
  - The convolutional block uses three set of filters of size [256, 256, 1024], "f" is 3, "s" is 2 and the block is "a".
  - The 5 identity blocks use three set of filters of size [256, 256, 1024], "f" is 3 and the blocks are "b", "c", "d", "e" and "f".
- Stage 5:
  - The convolutional block uses three set of filters of size [512, 512, 2048], "f" is 3, "s" is 2 and the block is "a".
  - The 2 identity blocks use three set of filters of size [512, 512, 2048], "f" is 3 and the blocks are "b" and "c".
- The 2D Average Pooling uses a window of shape (2,2) and its name is "avg_pool".
- The flatten doesn't have any hyperparameters or name.
- The Fully Connected (Dense) layer reduces its input to the number of classes using a softmax activation. Its name should be 'fc' + str(classes).

For more regular use it is faster to use the pretrained ResNet-50 in Keras. Keras has many of these backbone models with their ImageNet weights available in its library.

Conclusion

ResNet is a powerful backbone model that is used very frequently in many computer vision tasks. ResNet uses skip connection to add the output from an earlier layer to a later layer. This helps it mitigate the vanishing gradient problem.

Methodology

1. Dataset Overview

For this project, the Flickr_8K dataset is used. In this dataset data is properly labelled and for each image 5 captions are given

After extracting zip files from the flickr8k dataset these are the following folders (Size: 1GB)

- Images: Contains a total of 8092 images in JPEG format with different shapes and sizes.
- Flickr_TextData: Contains text files describing train_set, test_set. Flickr8k.token.txt contains 5 captions for each image i.e., total 40460 captions.

The Flickr8k.token.txt is the main file of our dataset that contains image name and their respective captions separated by newline("\n").

2. Importing Libraries

The following libraries are used in this project.

```
import numpy as np
import os
import matplotlib.pyplot as plt
import pickle
import pandas as pd
import random
from IPython.display import Image, display


from keras.preprocessing import image, sequence
from keras.applications.resnet50 import ResNet50
from keras.optimizers import Adam
from keras.layers import Dense, Flatten, Input
from keras.layers import Convolution2D, Dropout, LSTM, TimeDistributed, Embedding, Bidirectional, Activation, RepeatVector,Conca
tenate
from keras.models import Sequential,Model
from keras.utils import np_utils, plot_model
```

Fig. 6.

3. Loading Descriptions

The text files containing captions of the images are read and stored as strings in the programs. The strings are then split into lists of strings, separated by "\n". The Flickr8k.token.txt file contains names of images and captions separated by a "#" and index number of captions. This file is converted into a dictionary (named as token) with image names as keys and the captions as values. We have 40460 captions total in "token" dictionary.

```
In [4]: # Load data
images_dir = os.listdir("../input/adityaflickr8k/Flickr8K/Flickr_Data/Flickr_Data/Images")

images_path = '../input/adityaflickr8k/Flickr8K/Flickr_Data/Flickr_Data/Images/'
captions_path = '../input/adityaflickr8k/Flickr8K/Flickr_Data/Flickr_Data/Flickr_TextData/Flickr8k.token.txt'
train_path = '../input/adityaflickr8k/Flickr8K/Flickr_Data/Flickr_Data/Flickr_TextData/Flickr_8k.trainImages.txt'
val_path = '../input/adityaflickr8k/Flickr8K/Flickr_Data/Flickr_Data/Flickr_TextData/Flickr_8k.devImages.txt'
test_path = '../input/adityaflickr8k/Flickr8K/Flickr_Data/Flickr_Data/Flickr_TextData/Flickr_8k.testImages.txt'

captions = open(captions_path, 'r').read().split("\n")
x_train = open(train_path, 'r').read().split("\n")
x_val = open(val_path, 'r').read().split("\n")
x_test = open(test_path, 'r').read().split("\n")

In [5]: # Loading captions as values and images as key in dictionary
tokens = {}

for ix in range(len(captions)-1):
    temp = captions[ix].split("#")
    if temp[0] in tokens:
        tokens[temp[0]].append(temp[1][2:])
    else:
        tokens[temp[0]] = [temp[1][2:]]
```

Fig. 7.

4. Creating Train, Test and Validation Dataset

First, train, test and validation dataset files are created with header as 'image_id' and 'captions'. They are named 'flickr_8k_train_dataset.txt','flickr_8k_test_dataset.txt' &'flickr_8k_val_dataset.txt' respectively. These files are then populated with image ids and captions for each of these images. The files are tokenized by adding <start> at the starting of each sentence and <end> at the end of each sentence.

```
In [10]: # Creating train, test and validation dataset files with header as 'image_id' and 'captions'
train_dataset = open('./flickr_8k_train_dataset.txt','wb')
train_dataset.write(b"image_id\tcaptions\n")

val_dataset = open('./flickr_8k_val_dataset.txt','wb')
val_dataset.write(b"image_id\tcaptions\n")

test_dataset = open('./flickr_8k_test_dataset.txt','wb')
test_dataset.write(b"image_id\tcaptions\n")

Out[10]: 18

In [11]: # Populating the above created files for train, test and validation dataset with image ids and captions for each of these images
for img in x_train:
    if img == '':
        continue
    for capt in tokens[img]:
        caption = "<start> " + capt + " <end>"
        train_dataset.write((img + "\t" + caption + "\n").encode())
        train_dataset.flush()
train_dataset.close()

In [12]: for img in x_test:
    if img == '':
        continue
    for capt in tokens[img]:
        caption = "<start> " + capt + " <end>"
        test_dataset.write((img + "\t" + caption + "\n").encode())
        test_dataset.flush()
test_dataset.close()

In [13]: for img in x_val:
    if img == '':
        continue
    for capt in tokens[img]:
        caption = "<start> " + capt + " <end>"
        val_dataset.write((img + "\t" + caption + "\n").encode())
        val_dataset.flush()
val_dataset.close()
```

Fig. 8.

5. Image feature Extraction

The images need to be converted into an encoding so that the machine can understand the patterns in it. For this task, we use a pre-trained model called ResNet50 that has been already trained on large datasets and extract the features from these models and use them for our work.

```
In [14]: ResNet50_Model_1 = ResNet50(include_top=False, weights='imagenet', input_shape=(224, 224,3), pooling='avg')

ResNet50_Model_1.summary()

Downloading data from https://storage.googleapis.com/tensorflow/keras-applications/resnet/resnet50_weights_tf_dim_ordering_tf_k
ernels_notop.h5
94773248/94765736 [==============================] - 1s 0us/step
Model: "resnet50"
__________________________________________________________________________________________________
Layer (type)                    Output Shape         Param #     Connected to
==================================================================================================
input_1 (InputLayer)            [(None, 224, 224, 3) 0
__________________________________________________________________________________________________
conv1_pad (ZeroPadding2D)       (None, 230, 230, 3)  0           input_1[0][0]
__________________________________________________________________________________________________
conv1_conv (Conv2D)             (None, 112, 112, 64) 9472        conv1_pad[0][0]
__________________________________________________________________________________________________
conv1_bn (BatchNormalization)   (None, 112, 112, 64) 256         conv1_conv[0][0]
__________________________________________________________________________________________________
conv1_relu (Activation)         (None, 112, 112, 64) 0           conv1_bn[0][0]
__________________________________________________________________________________________________
pool1_pad (ZeroPadding2D)       (None, 114, 114, 64) 0           conv1_relu[0][0]
__________________________________________________________________________________________________
pool1_pool (MaxPooling2D)       (None, 56, 56, 64)   0           pool1_pad[0][0]
__________________________________________________________________________________________________
conv2_block1_1_conv (Conv2D)    (None, 56, 56, 64)   4160        pool1_pool[0][0]
```

Fig. 9.

We omit the last layer (which is the softmax layer) because we only need to extract the features, not to classify the images. Besides that, we also do transfer learning by using the pretrained parameters (weights) from ImageNet. ImageNet has been trained over 14 million images and has been grouped into specific categories and sub-categories. Global average pooling will be applied to the output of the last convolutional block, and thus the output of the model will be a 2D tensor. After executing the code, the output would be a 1x2048 vector for each image, containing the features from the images.

The features for all training data images are extracted using ResNet 50 and we have mapped each image name with their respective feature array. We dump the features dictionary into a "train_encoded_images.p" pickle file.

```
In [16]: # Helper function to process images
def preprocessing(img_path):
    im = image.load_img(img_path, target_size=(224,224,3))
    im = image.img_to_array(im)
    im = np.expand_dims(im, axis=0)
    return im

In [17]: start_time_1 = time.time()

In [18]: img_train_data = {}
ctr=0
for ix in x_train:
    if ix == "":
        continue
    if ctr >= 3000:
        break
    ctr+=1
    if ctr%1000==0:
        print(ctr)
    path = images_path + ix
    img = preprocessing(path)
    pred = ResNet50_Model_1.predict(img).reshape(2048)
    img_train_data[ix] = pred
    print(ctr,end="\r")

In [21]: # opening train_encoded_images.p file and dumping it's content
with open("./train_encoded_images.p", "wb") as pickle_f:
    pickle.dump(img_train_data, pickle_f)
```

Fig. 10.

6. Reading Train Dataset and Extracting Unique Words

Images and their corresponding captions stored in "flickr_8k_train_dataset.txt" are loaded into a data frame.

```
In [22]: # Loading image and its corresponding caption into a dataframe and then storing values from dataframe into 'ds'
pd_dataset = pd.read_csv("./flickr_8k_train_dataset.txt", delimiter='\t')
ds = pd_dataset.values
print("Shape of Train Dataframe: -",ds.shape)

Shape of Train Dataframe: - (30000, 2)
```

Fig. 11.

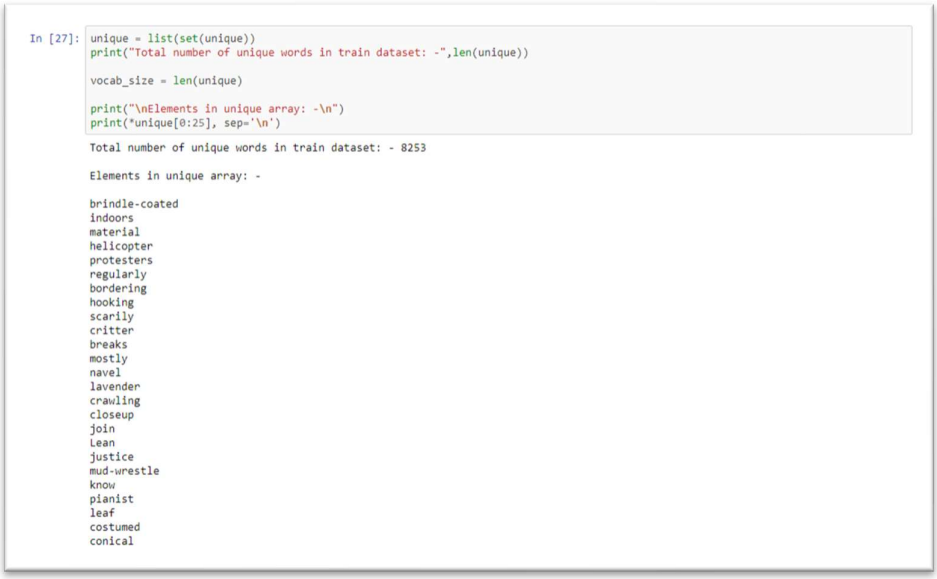
All captions in this data frame are stored into a list. Each of these captions are then split into words and stored in a separate list.

```
In [24]: # Storing all the captions from ds into a list
         sentences = []
         for ix in range(ds.shape[0]):
             sentences.append(ds[ix,1])
         print ("Number of captions/sentences in train dataset: -",len(sentences))

         print("\nElements in setences array: -\n")
         print(*sentences[0:5], sep='\n')

         Number of captions/sentences in train dataset: - 30000

         Elements in setences array: -

         <start> A black dog is running after a white dog in the snow . <end>
         <start> Black dog chasing brown dog through snow <end>
         <start> Two dogs chase each other across the snowy ground . <end>
         <start> Two dogs play together in the snow . <end>
         <start> Two dogs running through a low lying body of water . <end>

In [25]: # Splitting each captions stored in 'sentences' and storing them in 'words' as list of list
         words = [i.split() for i in sentences]

         print("Elements in words array: -\n")
         print(*words[0:5], sep='\n')

         Elements in words array: -

         ['<start>', 'A', 'black', 'dog', 'is', 'running', 'after', 'a', 'white', 'dog', 'in', 'the', 'snow', '.', '<end>']
         ['<start>', 'Black', 'dog', 'chasing', 'brown', 'dog', 'through', 'snow', '<end>']
         ['<start>', 'Two', 'dogs', 'chase', 'each', 'other', 'across', 'the', 'snowy', 'ground', '.', '<end>']
         ['<start>', 'Two', 'dogs', 'play', 'together', 'in', 'the', 'snow', '.', '<end>']
         ['<start>', 'Two', 'dogs', 'running', 'through', 'a', 'low', 'lying', 'body', 'of', 'water', '.', '<end>']
```

Fig. 12.

Then, a list of unique words is created using the extend() & set() function on the list of words previously created.

```
In [26]: # Creating a list of all unique words
         unique = []
         for i in words:
             unique.extend(i)
```



```
In [27]: unique = list(set(unique))
         print("Total number of unique words in train dataset: -",len(unique))

         vocab_size = len(unique)

         print("\nElements in unique array: -\n")
         print(*unique[0:25], sep='\n')

         Total number of unique words in train dataset: - 8253

         Elements in unique array: -

         brindle-coated
         indoors
         material
         helicopter
         protesters
         regularly
         bordering
         hooking
         scarily
         critter
         breaks
         mostly
         navel
         lavender
         crawling
         closeup
         join
         Lean
         justice
         mud-wrestle
         know
         pianist
         leaf
         costumed
         conical
```

Fig. 13.

Two dictionaries are then created. Each unique word is assigned an index value. “word_2_indices” has words as keys and indices as values, whereas “indices_2_word” has indices as keys and words as values. A new word “UNK”, denoting an unknown word, is added in both the dictionaries at index 0.

```
In [28]: #Vectorization
         word_2_indices = {val:index for index, val in enumerate(unique)}
         indices_2_word = {index:val for index, val in enumerate(unique)}

In [29]: print("Elements in word_2_indices dictionary: -\n")
         print(dict(list(word_2_indices.items())[0: 5]))

         print("\nElements in indices_2_word dictionary: -\n")
         print(dict(list(indices_2_word.items())[0: 5]))

         Elements in word_2_indices dictionary: -

         {'brindle-coated': 0, 'indoors': 1, 'material': 2, 'helicopter': 3, 'protesters': 4}

         Elements in indices_2_word dictionary: -

         {0: 'brindle-coated', 1: 'indoors', 2: 'material', 3: 'helicopter', 4: 'protesters'}

In [30]: #UNK - Unknown Word

         word_2_indices['UNK'] = 0
         word_2_indices['raining'] = 8253

         indices_2_word[0] = 'UNK'
```

Fig. 14.

The length of the longest caption is extracted, and it is found to be 40. This value is stored in variable "max_len".

7. Creating Padded Sequences & Subsequent Words

The data is converted into sequential form. We create two arrays called "padded_sequences" and "subsequent_words". The captions can have different length, even for the same image. To handle this, the sliced caption has to be padded with "UNK", to the length of the longest caption in the train set. The padded sequences are demonstrated in the images below.

```
In [34]: padded_sequences, subsequent_words = [], []

         for ix in range(ds.shape[0]):
             partial_seqs = []
             next_words = []
             text = ds[ix,1].split()
             text = [word_2_indices[i] for i in text]
             for i in range(1, len(text)):
                 partial_seqs.append(text[:i])
                 next_words.append(text[i])
             padded_partial_seqs = sequence.pad_sequences(partial_seqs, max_len, padding='post')

             next_words_1hot = np.zeros([len(next_words), vocab_size], dtype=np.bool)

             #Vectorization
             for i,next_word in enumerate(next_words):
                 next_words_1hot[i, next_word] = 1

             padded_sequences.append(padded_partial_seqs)
             subsequent_words.append(next_words_1hot)


         padded_sequences = np.asarray(padded_sequences,dtype=object)
         subsequent_words = np.asarray(subsequent_words,dtype=object)

         print("Shape of padded_sequences array",padded_sequences.shape)
         print("Shape of subsequent_words array",subsequent_words.shape)

         Shape of padded_sequences array (30000,)
         Shape of subsequent_words array (30000,)
```

Fig. 15.

```
In [35]: idx= random.randrange(0, len(padded_sequences))
         print("Example of a element in padded_sequences: -\n")
         print(padded_sequences[idx])

Example of a element in padded_sequences: -

[[4548    0    0    0    0    0    0    0    0    0    0    0    0    0
     0    0    0    0    0    0    0    0    0    0    0    0    0    0
     0    0    0    0    0    0    0    0    0    0    0    0]
 [4548 6221    0    0    0    0    0    0    0    0    0    0    0    0
     0    0    0    0    0    0    0    0    0    0    0    0    0    0
     0    0    0    0    0    0    0    0    0    0    0    0]
 [4548 6221 4309    0    0    0    0    0    0    0    0    0    0    0
     0    0    0    0    0    0    0    0    0    0    0    0    0    0
     0    0    0    0    0    0    0    0    0    0    0    0]
 [4548 6221 4309 4960    0    0    0    0    0    0    0    0    0    0
     0    0    0    0    0    0    0    0    0    0    0    0    0    0
     0    0    0    0    0    0    0    0    0    0    0    0]
 [4548 6221 4309 4960 7944    0    0    0    0    0    0    0    0    0
     0    0    0    0    0    0    0    0    0    0    0    0    0    0
     0    0    0    0    0    0    0    0    0    0    0    0]
 [4548 6221 4309 4960 7944 2048    0    0    0    0    0    0    0    0
     0    0    0    0    0    0    0    0    0    0    0    0    0    0
     0    0    0    0    0    0    0    0    0    0    0    0]
 [4548 6221 4309 4960 7944 2048 1407    0    0    0    0    0    0    0
     0    0    0    0    0    0    0    0    0    0    0    0    0    0
     0    0    0    0    0    0    0    0    0    0    0    0]
 [4548 6221 4309 4960 7944 2048 1407 2086    0    0    0    0    0    0
     0    0    0    0    0    0    0    0    0    0    0    0    0    0
     0    0    0    0    0    0    0    0    0    0    0    0]
 [4548 6221 4309 4960 7944 2048 1407 2086 3678    0    0    0    0    0
     0    0    0    0    0    0    0    0    0    0    0    0    0    0
     0    0    0    0    0    0    0    0    0    0    0    0]]
```

Fig. 16.

```
In [36]: for ix in range(len(padded_sequences[idx])):
             for iy in range(max_len):
                 print(indices_2_word[padded_sequences[idx][ix][iy]], end =" ")
             print("\n")

         print(len(padded_sequences[idx]))

<start> UNK UNK UNK UNK UNK UNK UNK UNK UNK UNK UNK UNK UNK UNK UNK UNK UNK UNK UNK UNK UNK UNK UNK UNK UNK UNK UNK UNK UNK UNK
UNK UNK UNK UNK UNK UNK UNK UNK UNK

<start> A UNK UNK UNK UNK UNK UNK UNK UNK UNK UNK UNK UNK UNK UNK UNK UNK UNK UNK UNK UNK UNK UNK UNK UNK UNK UNK UNK UNK UNK U
NK UNK UNK UNK UNK UNK UNK UNK UNK

<start> A man UNK UNK UNK UNK UNK UNK UNK UNK UNK UNK UNK UNK UNK UNK UNK UNK UNK UNK UNK UNK UNK UNK UNK UNK UNK UNK UNK UNK U
NK UNK UNK UNK UNK UNK UNK UNK UNK

<start> A man waterskies UNK UNK UNK UNK UNK UNK UNK UNK UNK UNK UNK UNK UNK UNK UNK UNK UNK UNK UNK UNK UNK UNK UNK UNK UNK UN
K UNK UNK UNK UNK UNK UNK UNK UNK UNK

<start> A man waterskies with UNK UNK UNK UNK UNK UNK UNK UNK UNK UNK UNK UNK UNK UNK UNK UNK UNK UNK UNK UNK UNK UNK UNK UNK U
NK UNK UNK UNK UNK UNK UNK UNK UNK UNK UNK

<start> A man waterskies with a UNK UNK UNK UNK UNK UNK UNK UNK UNK UNK UNK UNK UNK UNK UNK UNK UNK UNK UNK UNK UNK UNK UNK UNK
UNK UNK UNK UNK UNK UNK UNK UNK UNK UNK

<start> A man waterskies with a life UNK UNK UNK UNK UNK UNK UNK UNK UNK UNK UNK UNK UNK UNK UNK UNK UNK UNK UNK UNK UNK UNK UN
K UNK UNK UNK UNK UNK UNK UNK UNK UNK UNK

<start> A man waterskies with a life jacket UNK UNK UNK UNK UNK UNK UNK UNK UNK UNK UNK UNK UNK UNK UNK UNK UNK UNK UNK UNK UNK
UNK UNK UNK UNK UNK UNK UNK UNK UNK UNK

<start> A man waterskies with a life jacket . UNK UNK UNK UNK UNK UNK UNK UNK UNK UNK UNK UNK UNK UNK UNK UNK UNK UNK UNK UNK U
NK UNK UNK UNK UNK UNK UNK UNK UNK UNK UNK

9
```

Fig. 17.

8. Creating Arrays of Captions, Next Words, Images & Image Names

First two numpy arrays of captions and next words are created and stored as “captions.npy” & “next_words.npy”, respectively.

```
In [39]: captions = np.zeros([0, max_len])
         next_words = np.zeros([0, vocab_size])

         for ix in range(num_of_images):
             captions = np.concatenate([captions, padded_sequences[ix]])
             next_words = np.concatenate([next_words, subsequent_words[ix]])
             print(ix,end="\r")

         np.save("./captions.npy", captions)
         np.save("./next_words.npy", next_words)

         print("Shape of captions array: -",captions.shape)
         print("Shape of next_words array: -",next_words.shape)

         Shape of captions array: - (25493, 40)
         Shape of next_words array: - (25493, 8254)
```

Fig. 18.

Next, using encoded images in “train_encoded_images.p” pickle file, two numpy arrays of images and image names are created and stored as “images.npy” & “image_names.npy”, respectively.

```
In [41]: with open('./train_encoded_images.p', 'rb') as f:
             encoded_images = pickle.load(f, encoding="bytes")

In [42]: print("No of encoded_images: -",len(encoded_images))

         No of encoded_images: - 3000

In [43]: imgs = []

         for ix in range(ds.shape[0]):
             if ds[ix, 0] in encoded_images.keys():
                 imgs.append(list(encoded_images[ds[ix, 0]]))

         imgs = np.asarray(imgs)
         print("Shape of imgs array: -",imgs.shape)

         Shape of imgs array: - (15000, 2048)
```

```
In [44]: images = []

         for ix in range(num_of_images):
             for iy in range(padded_sequences[ix].shape[0]):
                 images.append(imgs[ix])

         images = np.asarray(images)

         np.save("./images.npy", images)

         print("Shape of images array: -",images.shape)

         Shape of images array: - (25493, 2048)

In [45]: image_names = []

         for ix in range(num_of_images):
             for iy in range(padded_sequences[ix].shape[0]):
                 image_names.append(ds[ix, 0])

         image_names = np.asarray(image_names)

         np.save("./image_names.npy", image_names)

         print("Shape of image_names array: -",image_names.shape)

         Shape of image_names array: - (25493,)
```

Fig. 19.

9. Model Architecture

First, we create an image model for extracting the feature vector from the image. This has a Dense layer and RepeatVector later and uses Rectified Linear Unit activation.

```
In [47]: embedding_size = 128
         max_len = 40

In [48]: image_model = Sequential()
         image_model.add(Dense(embedding_size, input_shape=(2048,), activation='relu'))
         image_model.add(RepeatVector(max_len))

         image_model.summary()

         Model: "sequential"
         _________________________________________________________________
         Layer (type)                 Output Shape              Param #
         =================================================================
         dense (Dense)                (None, 128)               262272
         _________________________________________________________________
         repeat_vector (RepeatVector) (None, 40, 128)           0
         =================================================================
         Total params: 262,272
         Trainable params: 262,272
         Non-trainable params: 0
```

Fig. 20.

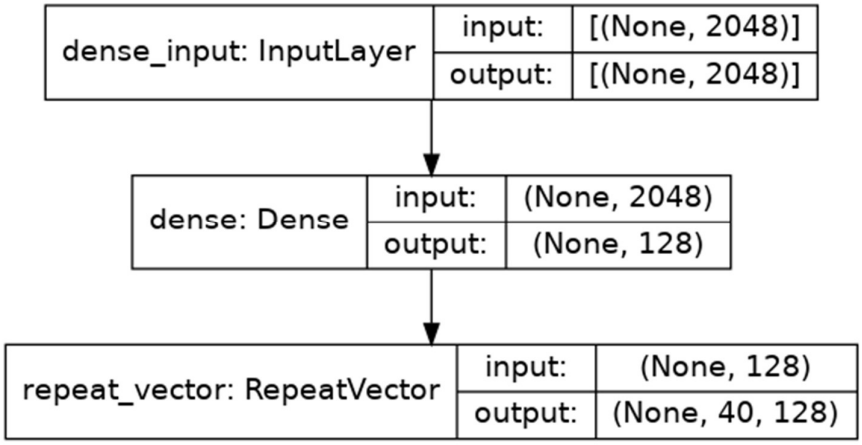


Fig. 21. Structure of Image_Model

Next, we create a language model for processing the sequence from the text. It has a Embedding layer, LSTM layer and TimeDistributed Layer.

```
In [50]: language_model = Sequential()
         language_model.add(Embedding(input_dim=vocab_size, output_dim=embedding_size, input_length=max_len))
         language_model.add(LSTM(256, return_sequences=True))
         language_model.add(TimeDistributed(Dense(embedding_size)))

         language_model.summary()

         Model: "sequential_1"
         _________________________________________________________________
         Layer (type)                 Output Shape              Param #
         =================================================================
         embedding (Embedding)        (None, 40, 128)           1056512
         _________________________________________________________________
         lstm (LSTM)                  (None, 40, 256)           394240
         _________________________________________________________________
         time_distributed (TimeDistri (None, 40, 128)           32896
         =================================================================
         Total params: 1,483,648
         Trainable params: 1,483,648
         Non-trainable params: 0
         _________________________________________________________________
```

Fig. 22.

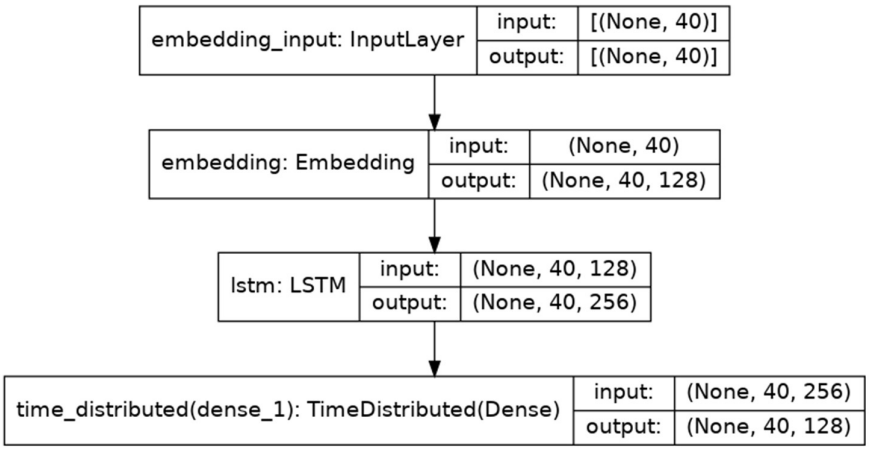


Fig. 23. Structure of language_model

Finally, we create a Image caption Generator Model that decode the outputs by concatenating the above two layers. The two concatenated layers are followed by 2 LSTM layers, 1 Dense layer and in the end, a softmax activation layer. The image model and language model layers take the input. The Caption generator model uses Categorical Cross entropy loss and RMSprop optimizer.

```
In [52]: conca = Concatenate()([image_model.output, language_model.output])
         x = LSTM(128, return_sequences=True)(conca)
         x = LSTM(512, return_sequences=False)(x)
         x = Dense(vocab_size)(x)
         out = Activation('softmax')(x)
         Image_caption_model = Model(inputs=[image_model.input, language_model.input], outputs = out)

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

         Image_caption_model.summary()
```

```
Model: "model"
__________________________________________________________________________________________________
Layer (type)                    Output Shape         Param #     Connected to
==================================================================================================
embedding_input (InputLayer)    [(None, 40)]         0
__________________________________________________________________________________________________
dense_input (InputLayer)        [(None, 2048)]       0
__________________________________________________________________________________________________
embedding (Embedding)           (None, 40, 128)      1056512     embedding_input[0][0]
__________________________________________________________________________________________________
dense (Dense)                   (None, 128)          262272      dense_input[0][0]
__________________________________________________________________________________________________
lstm (LSTM)                     (None, 40, 256)      394240      embedding[0][0]
__________________________________________________________________________________________________
repeat_vector (RepeatVector)    (None, 40, 128)      0           dense[0][0]
__________________________________________________________________________________________________
time_distributed (TimeDistribut (None, 40, 128)      32896       lstm[0][0]
__________________________________________________________________________________________________
concatenate (Concatenate)       (None, 40, 256)      0           repeat_vector[0][0]
                                                                 time_distributed[0][0]
__________________________________________________________________________________________________
lstm_1 (LSTM)                   (None, 40, 128)      197120      concatenate[0][0]
__________________________________________________________________________________________________
lstm_2 (LSTM)                   (None, 512)          1312768     lstm_1[0][0]
__________________________________________________________________________________________________
dense_2 (Dense)                 (None, 8254)         4234302     lstm_2[0][0]
__________________________________________________________________________________________________
activation (Activation)         (None, 8254)         0           dense_2[0][0]
==================================================================================================
Total params: 7,490,110
Trainable params: 7,490,110
Non-trainable params: 0
__________________________________________________________________________________________________
```

Fig. 24.

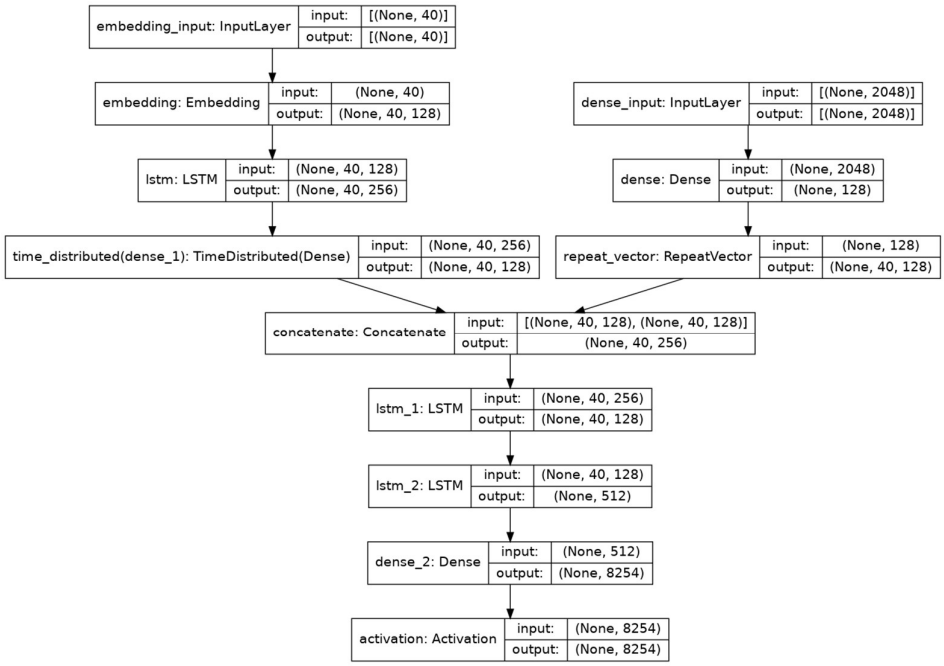


Fig. 25. Structure of Image_caption_model

10. Feature Learning in Model

The Image_Caption_model created above is now trained on the images, captions and next_words data with batch size=512 and no of epochs=200. The weights are then saved as "model_weights.h5". With Kaggle GPU, the model training time is 14 minutes.

```
In [54]: start_time_3 = time.time()

In [55]: hist = Image_caption_model.fit([images, captions], next_words, batch_size=512, epochs=200)

Epoch 1/200
50/50 [==============================] - 9s 82ms/step - loss: 6.2475 - accuracy: 0.0663
Epoch 2/200
50/50 [==============================] - 4s 83ms/step - loss: 5.2820 - accuracy: 0.0897
Epoch 3/200
50/50 [==============================] - 4s 82ms/step - loss: 5.0451 - accuracy: 0.1146
Epoch 4/200
50/50 [==============================] - 4s 82ms/step - loss: 5.1925 - accuracy: 0.0746
Epoch 5/200
50/50 [==============================] - 4s 82ms/step - loss: 5.0645 - accuracy: 0.0909
Epoch 6/200
50/50 [==============================] - 4s 84ms/step - loss: 4.8936 - accuracy: 0.1163
Epoch 7/200
50/50 [==============================] - 4s 87ms/step - loss: 4.8128 - accuracy: 0.1186
Epoch 8/200
50/50 [==============================] - 4s 82ms/step - loss: 4.8592 - accuracy: 0.1068
Epoch 9/200
50/50 [==============================] - 4s 82ms/step - loss: 4.8734 - accuracy: 0.0961
Epoch 10/200
```

```
Epoch 190/200
50/50 [==============================] - 4s 83ms/step - loss: 0.2484 - accuracy: 0.9059
Epoch 191/200
50/50 [==============================] - 4s 81ms/step - loss: 0.2447 - accuracy: 0.9055
Epoch 192/200
50/50 [==============================] - 4s 83ms/step - loss: 0.2494 - accuracy: 0.9068
Epoch 193/200
50/50 [==============================] - 4s 88ms/step - loss: 0.2529 - accuracy: 0.9044
Epoch 194/200
50/50 [==============================] - 4s 85ms/step - loss: 0.2399 - accuracy: 0.9114
Epoch 195/200
50/50 [==============================] - 4s 82ms/step - loss: 0.2453 - accuracy: 0.9076
Epoch 196/200
50/50 [==============================] - 4s 81ms/step - loss: 0.2372 - accuracy: 0.9093
Epoch 197/200
50/50 [==============================] - 4s 81ms/step - loss: 0.2487 - accuracy: 0.9048
Epoch 198/200
50/50 [==============================] - 4s 83ms/step - loss: 0.2813 - accuracy: 0.9019
Epoch 199/200
50/50 [==============================] - 4s 81ms/step - loss: 0.2440 - accuracy: 0.9061
Epoch 200/200
50/50 [==============================] - 4s 81ms/step - loss: 0.2471 - accuracy: 0.9028
```

```
In [56]: print("Time taken to fit Image_caption_model = %s minutes" % ((time.time() - start_time_3)/60))
         Time taken to fit Image_caption_model = 14.085781478881836 minutes

In [57]: Image_caption_model.save_weights("./model_weights.h5")
```

Fig. 26.

Results

Accuracy and Loss during training

The accuracy and loss of the model is plotted for each epoch of training and shown in Fig. 27 & Fig. 28. The accuracy increases and loss decreases till 180 epochs and then starts oscillating.

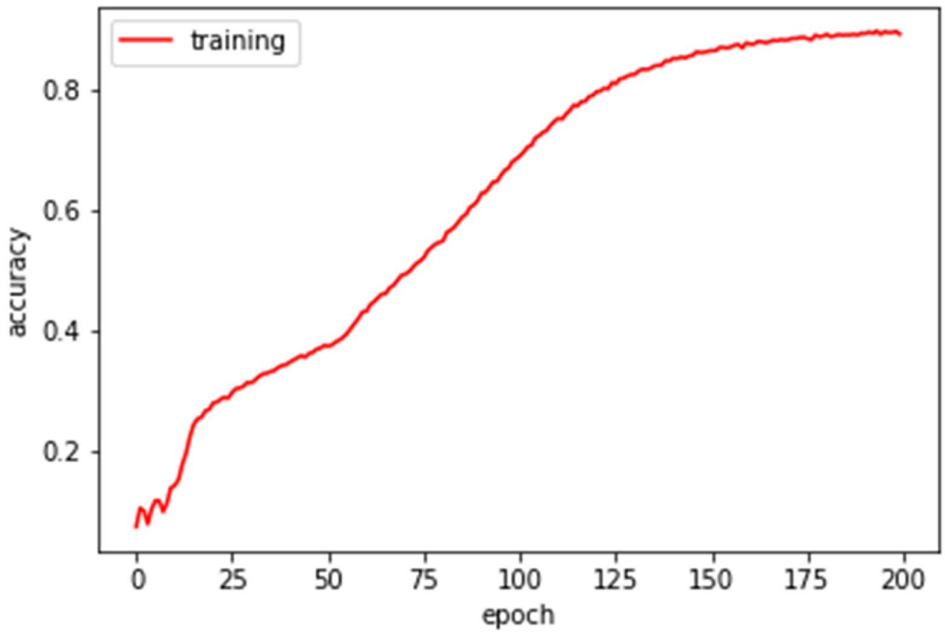


Fig. 27. Plot of accuracy vs epoch for Image Caption Generator Model.

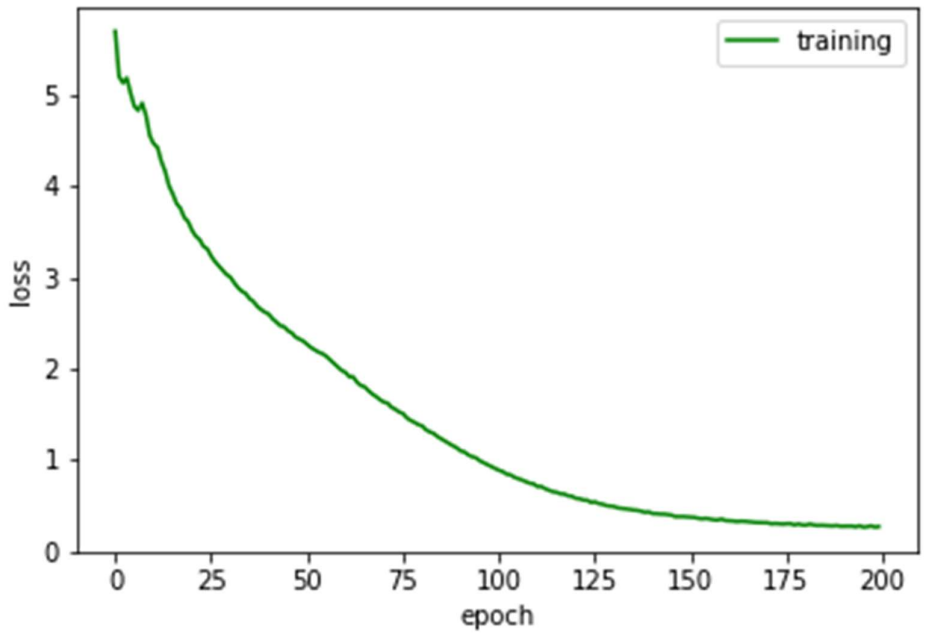


Fig. 28. Plot of loss vs epoch for Image Caption Generator Model.

Testing the Image Caption Generator model

Once the model training is complete, various test images are loaded and are encoded using ResNet50 model. The encoded images are used in the trained Image Caption Generator to predict captions for each test image. The generated caption for each test image is shown in Fig. 29. The generated captions match the image quite well, but the captions are not 100% accurate.

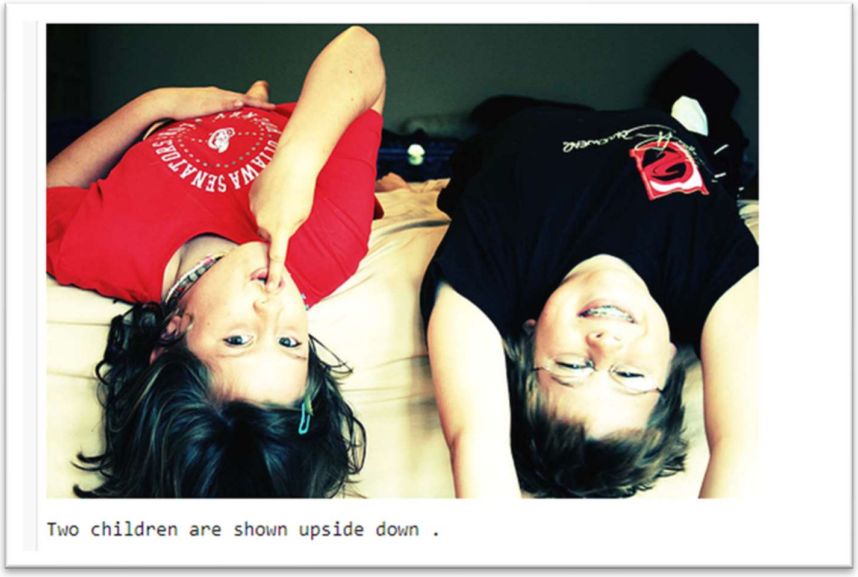


Fig. 29(a).

Fig. 29(b).

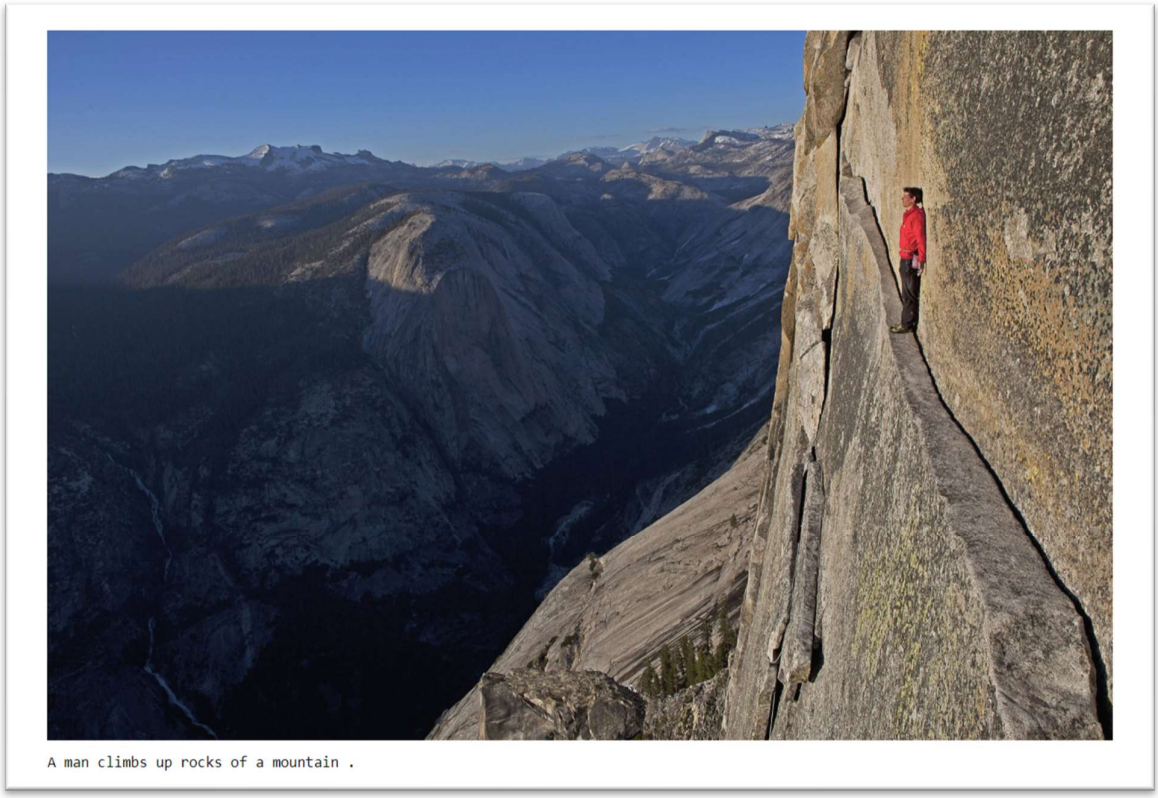


Fig. 29(c).

Fig. 29(d).

Fig. 29(e).

Fig. 29(f).

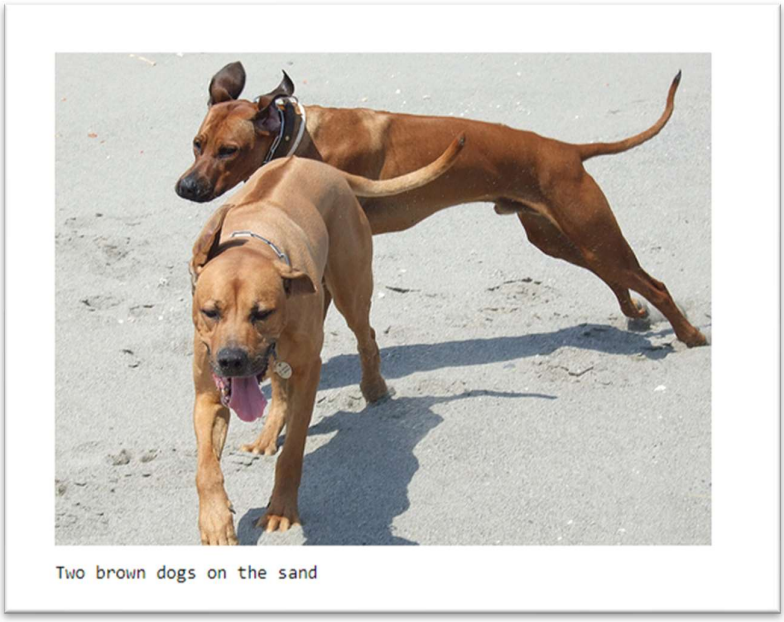


Fig. 29(g).

Fig. 29(h).

Fig. 29. Results of Image Caption Generator on various test images.

Conclusion & Future Work

Automatically image captioning is far from mature and there are a lot of ongoing research projects aiming for more accurate image feature extraction and semantically better sentence generation. We used a smaller dataset (Flickr8k) due to limited computational power. There can be potential improvements if given more time. First of all, we directly used pre-trained ResNet50 network as part of our pipeline, so the network does not adapt to this specific training dataset. Thus, by experimenting with different CNN pre-trained networks and enabling fine-tuning, we expect to achieve a slightly higher accuracy. Another potential improvement is by training on a combination of Flickr8k, Flickr30k, and MSCOCO. In general, the more diverse training dataset the network has seen, the more accurate the output will be. Other possible modifications include: -

- Doing more hyper parameter tuning (learning rate, batch size, number of layers, number of units, dropout rate, batch normalization etc.).
- Using BLEU score to evaluate and measure the performance of the model.

References

1. O. Vinyals, A. Toshev, S. Bengio, and D. Erhan, “Show and tell: a neural image caption generator,” in Proceedings of the IEEE conference on computer vision and pattern recognition, pp. 3156–3164, Boston, MA, USA, 2015.
2. R. Kiros, R. Salakhutdinov, and R. Zemel, “Multimodal neural language models,” in Proceedings of the 31st international conference on machine learning (ICML-14), pp. 595–603, Beijing, China, 2014.
3. Karpathy and L. Fei-Fei, Deep visual-semantic alignments for generating image descriptions, Stanford University, 2017.
4. K. Xu, J. Ba, R. Kiros et al., “Show, attend and tell: neural image caption generation with visual attention,” in International conference on machine learning, pp. 2048–2057, Lile, France, 2015.
5. https://www.deeplearning.ai/
6. https://towardsdatascience.com/understanding-and-coding-a-resnet-in-keras-446d7ff84d33
7. https://keras.io/api/applications/resnet/#resnet50-function
8. https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53
9. https://colah.github.io/posts/2015-08-Understanding-LSTMs/