IMAGE CAPTIONING USING DEEP LEARNING

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABBREVIATIONS

NLP	Natural Language Processing
BLEU	Bilingual Evaluation Understudy
ROUGE	Recall-Oriented Understudy for Gisting Evaluation
CIDEr	Consensus-based Image Description Evaluation
LCS	Longest Common Subsequence
TF-IDF	Term Frequency Inverse Document Frequency
METEOR	Metric for Evaluation of Translation with Explicit ORdering

CHAPTER 1

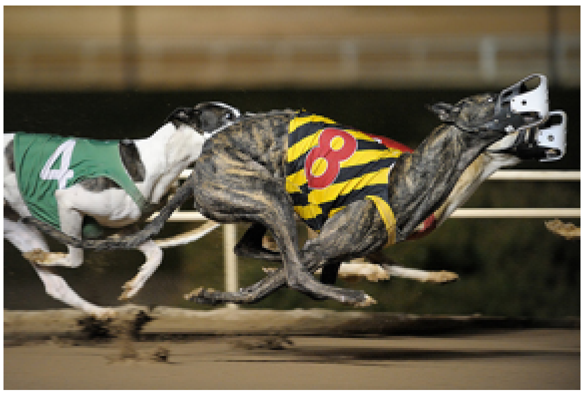
INTRODUCTION

1.1 Problem Definition

Human being can vividly describe the environment they are present in. Given a certain visual context, human beings can easily describe the various components, the focus of the image and also the actions that may be happening in the scene. Although for a machine or computer to do the same is an arduous challenge. The task of making a computer imitate the cognitive abilities of human being with respect to interpreting visual context is termed as image captioning.

The intricacy of the problem lies in its multi-label nature. Image captioning brings together the computer vision and natural language processing research communities. Individually both fields have made progress in recent times. With regard to computer vision, object detection, object segmentation have shown great improvement in terms of accuracy. Whereas in the field of NLP, text generation, text summarization seem to be making great head way. However, the combination of the two field is a fairly new and exciting domain. Figure 1.1 shows examples of image captioning. The image is given to the system as input and a description is produced as output.

Broadly speaking image captioning can be divided into two sub tasks, namely image understanding and linguistic generation or processing. It requires the understanding of context, which may comprise of an unordered list of identified features, actions and objects in the scene and to further create syntactically correct description for the same. The description should be crisp, concise, and syntactically accurate. It should also be able to capture a human kindred intuition of the image. Computer Vision deals with visual comprehension and the natural language processing helps with the generation of the needful context.


(a) Two dogs are running down track.

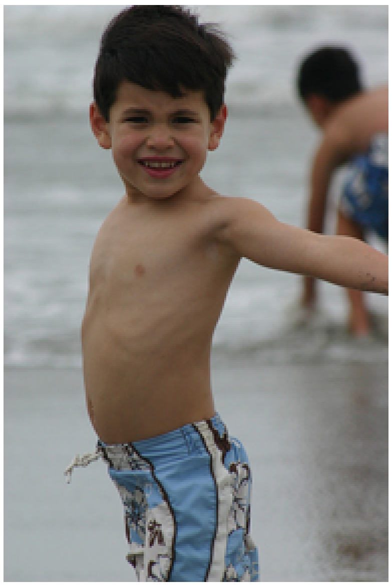

(b) Young boy in swimming trunks is playing in the water.

Figure 1.1: Image Captioning Examples

1.2 Objectives

In this project, we have defined a set of goals we aim to accomplish by the time of its completion.

- To make a moderately robust and performing system with limited resources in terms of memory and RAM, and processing power.
- To understand the factors affecting the training in terms of time and memory as well as the performance.
- To compare the different architectures mentioned in literature, and support findings with metric values.
- To understand the obstacles in the process, and the ways they are dealt with, when having adequate resources.

1.3 Scope and Application

The task of image captioning can be developed into tangible systems that can assist human beings in many a mundane tasks. A robust captioning system can be used for assisting the visually impaired. Over the past ten years the amount of data has grown exponentially and the amount of visual context on the internet is abundant. It is no longer feasible to screen or regulate the data manually. Hence such a captioning system can also be purposed into a visual search engine, a filter to screen discreet information and also a auto tagging system for social media platforms.

1.4 Report Structure

In chapter 2, we present a survey of the various methods used to solve the image captioning problem, including traditional methods. Chapter 3 will discuss the overall framework design involved in building an image captioning system. The major components, including the technical aspects, of the encoder and decoder structures as well as text preprocessing will be studied. Chapter 4 will discuss the neural network structure used to build the model, and a detailed look at the possible architectures will be taken. Chapter 5 discusses the various experiments and modifications we performed to enhance our model performance. Chapter 6 will demonstrate the actual code and logic used. Chapter 7 will discuss the available datasets and performance metrics employed in the evaluation of our model. Furthermore, we discuss the quantitative performance of various architectures that we experimented with. Finally, chapter 8 and 9 conclude the report and discuss the future research areas respectively.

CHAPTER 2

LITERATURE REVIEW

Image Captioning research has existed since the early twenty first century. The earlier methods use probabilistic learning models, and thus were limited in their capacity to compute such non linear functionality with flexibility. With the popularity of neural networks and deep learning, the problem gained back traction. Several new datasets prompted the use of many architectures and techniques that are much more successful than earlier methods, though still not perfect. In this chapter, we discuss these methodologies.

2.1 Template Based

One of the earlier papers that used template-based approach for image captioning was done by Yang et al. [1]. They used a quadruple template which consisted of noun, verb, scene (where the image occurs for example, lake, beach, playground etc.) and prepositions. The image dataset used was the UIUC Pascal Sentence dataset and support vector machines (SVM) were used for the detection of image components. In the language end, the English Gigaword corpus was used to provide semantic grounding, the Gigaword corpus is a bank of large generic English words. The model used Hidden Markov Chain model for the final generation of the sentences.

Kulkarnietal. [2] uses a similar concept but uses Conditional Random Fields (CRF) to match the words to the visual context. It is a fairly graphical approach where the nodes represent object attributes, objects and the spatial relationship between them. The CRF inference is used for shortlisting attributes or words that describe the given image and the templates are used to place them in a semantically coherent manner. The paper by Ushiku et al. [3] proposed a novel phrase learning technique they coined as Common Subspace for Model and Similarity (CoSMoS). Their model was able to make the descriptions more concise and short by using the same subspace to map image features and phrase features.

Although the template based approach is able to generate syntactically correct sentences, it is a very constraint method. This often forces such models to have a narrow intuition of the visual context. They seem less natural and lack creativity and complexity.

2.2 Retrieval Based

Farhadi et al. [4] uses a scoring procedure for finding the similarity between an image and a sentence. The approach they proposes depends on a common meaning phase. Each image in their dataset is represented or annotated in form of a triplet <object, action, scene>. The pool of sentences are then scored according to the greatest similarity or hits to these triplets and the sentence with highest score is assigned as caption for the given image.

Another approach was proposed by Gupta et al. [5] in which they use a training data of loosely labelled images,to find a similarity measure with the test image. Three of the best scored matches are used to generate a sentence. The approach is to extract phrases using the Stanford CoreNLP toolkit from the dataset of human annotated images and then given a new image, use the extracted phrases of the images whose similarity scores are in the top three with correspondence with the test image.

A method proposed by Hadosh et al. [6] entails a dictionary of image and language templates to fetch and generate captions for without forming any clutters. They use a set of eight thousand images that are each given or paired with five captions. They use Kernal Cannonical Correlation Analysis for finding the sentence image pair which shows maximum correlation. The paper not only proposes image captioning, but also proposes a K- nearest neighbour approach for similar image retrieval.

Retrieval based generates syntactically accurate descriptions but it is heavily dependent on the predefined dataset. It does not create new combinations and does not interpret minute changes in the scene. Moreover, many of the approaches work on image similarity,if the measure of similarity is inaccurate then the final generated outcome also skews away from the actual visual context.

2.3 Neural Network Based

2.3.1 Encoder-Decoder

The neural network learning based methods came to prominence since the 2015 MSCOCO Captioning Task. The first architectures were inspired from machine translation sequence-to-sequence architectures. In these, one recurrent unit was used to encode the language that was being translated from, and another to decode the intermediate representation into the language the text is translated to. Thus, they were named as encoder-decoder models. Similarly, in image captioning, encoder-decoder architectures have an encoding unit which is a CNN, and a decoding unit which is a recurrent neural network. It can be thought of as translating from an image to a language. Vinyal et al, Donahue et al and Wu et al have used this method [6,7]. Usually, the last fully connected layer of pre-trained CNN is used as the feature set representing the image. LSTMs are used as the recurrent unit. The image features are fed into the initial or every time-step of the LSTM.

2.3.2 Multimodal Systems

These are an enhancement of the encoder-decoder structures. Due to the different forms of data being used (image, text), these models are referred to as multimodal.[13,15] In particular, both types of data are parallelly processed and mapped to a common subspace. These are further used as the features for predicting the next word.

2.3.3 Attention Based Architectures

In another type of architecture, there is an effective concept used in machine translation called Attention Mechanism which has shown to give very good performance in Image

Captioning as well. Attention mechanism is usually a multilinear perceptron unit that calculates a “context“ vector at each time-step.[17] Context vectors help decide which of the encoder timesteps should be focused on when calculating the next word prediction. For images, it is referred to as visual attention. It is very useful when trying to analyse the performance of the model when it produces faulty captions. [12,13] have used a visual sentinel in addition to attention to determine when to apply attention. [10,16,18] have used attention in both encoding i.e. visual attention, and decoding i.e. language model.

2.3.4 Other optimization techniques

The architectures described until now used the full image to produce the captions. Some other papers used region-based captioning, in which an R-CNN identifies the different semantically important objects and the regional annotations are combined to form the final sentence.[7,8] use attribute detectors to find high end, commonly occurring words at each time step. In addition, beam search is used to rank the sentences generated and to generate the most probabilistic one.

2.3.5 Other Learning Methods

Until now, we have discussed only supervised learning method in which the network learns from minimizing a loss function calculated using given ground truth captions. Another method used is self-critical reinforcement learning networks which use two networks, with one being the baseline. The other network learns from rewards and punishments based on baseline comparison. Generative Adversarial Networks(GANs) have also been used for image captioning, with better variability in the generated captions but lesser performance.

CHAPTER 3

PROPOSED METHODOLOGY

The overall system design is given in Figure 3.1 The final systems, with all components in place, is given an image as input and generates a string caption as output.

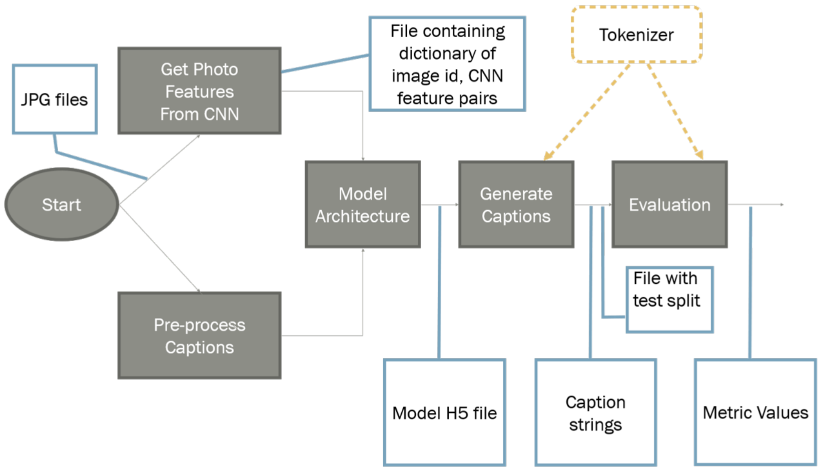


Figure 3.1: System Architecture

In this chapter, we will discuss the major components i.e. encoder model, captions processor, the language model, and the generator module which combine the 3.

3.1 Encoder Model

The encoder is a pretrained VGG16 CNN. By pretrained, we mean that we use weights which have been already determined by training on a huge dataset, specifically ImageNet. VGG16 came as first runner-up in the 2014 ILSVRC. The structure of VGG has been given in Figure 3.2.

The CNN was trained on single-class classification task on 1000 classes. Our problem, is essentially a multi-class multi-label classification as we wish to detect as many objects in an image as possible, belonging to possibly several unique classes. For this, we have fine-tuned our architecture, which will be discussed in the next chapter. Regardless, we only need to be concerned with a good representation of the image, and

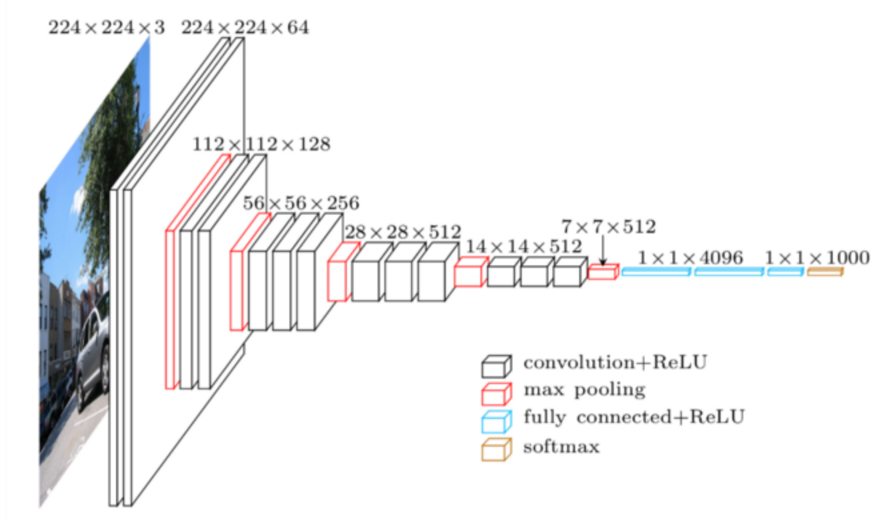


Figure 3.2: VGG16 Architecture

not the actual detection. We use the 4096 sized fully connected layer, as a feature set, as it represents high-level features of the image. In some implementations, even the convolution features have been used, but we omit those, for the sake of memory and time complexity involved in the training.

We experimented with VGG16 and InceptionV3 convolution neural nets, using pre-trained ImageNet weights. These weights have been set by running the CNN architecture through the ‘Image Net Large Scale Visual Recognition Challenge.‘ The challenge consists of 1000 categories that include common objects that are seen in a high frequency in our daily lives.

The VGG16 architecture is made up of 16 convolution layers, 5 max pool layers and 3 fully connected layers. For our problem we do not require the last fully connected layer since that was designed for the classification task on the ISVRC dataset, with a 1000 categories. Hence we pop the last layer and use the second last fully connected layer which gives us a vector of size 4096. This is the required image feature map.

The InceptionV3 is 42 layers deep and uses factorized 7×7 convolutions. It also used batch norm instead of the auxiliary filters as the auxiliary filters were not contributing much. Much like in VGG16 the last fully connected layer was designed for the challenge. For our experiments we used the second last layer that gives us a vector of 2048.

The InceptionV3 model worked better on the ISVRC challenge as compared to the

VGG16 but we did not find a significant difference in the performance on the Flickr8k dataset for the task of image captioning. This can be because of the variation of the images in the dataset. While the ISVRC dataset has a significant high level features, the Flickr8k has discernable low level features. The InceptionV3 model also shows a decrease in performance while fine tuning. This can be because of the batch normalization problem. For these reasons we conducted most of our model architectures by using the VGG16 architecture, and further performed fine tuning in the same.

3.1.1 Module Working

In our module, we preprocess all the image to extract features before the training and store them in a python dictionary data structure. This is done to make the training process more optimized, since we can directly take the features from a file and save time. For this, all images should have the correct properties as required by the VGG16 structure. They have to be of size 224 X 224, and have scaled values, so as to pass through the convolution layers in a predictable manner. We get the list of all image identifiers as part of the dataset, using which we extract the images from the file names, convert them into an array and pass them through the encoder module. We create a dictionary of identifier to feature and save it for use in training later. The file is saved using a library called pickle, which makes it very convenient to store python data structures. Note that, when generating, we will need the actual model, and not the saved file.

3.2 Caption Preprocessor

This is a step to process the provided ground truth captions. The data set comes with a file containing image identifiers and caption pairs. Each identifier is repeated 5 times for the 5 reference captions per image. This module extracts a dictionary of identifier paired with a list of all associated captions.

Furthermore, some basic language processing operations are applied. The texts are all converted to lowercase. This is due to the fact that language generation is concerned more with semantic and structural relation among words and less with superficial grammatical rules such as capitalization. In addition, the capitalized words would result in a vocabulary with multiple occurrences of the same words, increasing it two-fold.

Another step is to remove all punctuation. The sentences are generally expected to be less complex while capturing enough information, thus ideally not requiring semantically complex, punctuation nuances. In addition, regardless of presence of punctuation, the semantic effect can be captured in language structure as well. In addition, like with the case of capital lettered words, punctuation increases the vocabulary size and renders the training inefficient. Sentence beginnings endings are marked by a special token and we add these to the descriptions.

While numbers are important for scene understanding, numbers in digit form also complicate the vocabulary and the consequent learning process.So we also remove these and only keep numbers in word form. In any circumstance, an object present in big numbers can be referred as a group and our model is successful in doing so.

Finally, this module is also used to determine the vocabulary size. The size of the vocabulary has shown to have an effect on training. By reducing vocabulary size, the performance generally increases. This could be due to less variance and will be discussed in later chapters. Finally, we save the processed captions as a dictionary, which is used in all the following modules.

3.2.1 Tokenizer

Words, for a language model, are usually represented as one-hot encodings. In our model, we have to feed the generated sentences as a sequence of words. When dealing with sequences, it is convenient to represent words as indexed integers. For this purpose, we use a tokenizer object in python for generation and training. These objects contain useful information about a text. These include an index of the vocabulary, the documents each word appears in and other important information. We need the index to convert the output vectors into words and then to convert a string to a sequence of integers. We build a tokenizer object on the description dictionary, and save it for use during training.

3.3 Decoder

3.3.1 Long short Term Memory

The LSTM are a form of recurrent neural nets that form a gated system to tackle the shortcomings of the vanilla neural nets, namely the vanishing gradient problem and the exploding gradient problem. LSTM‘s are used to learn and generate long term dependencies or sequential data. The system works on the concept of transitional probabilities where each new state depends on the set of previously learned states. The error is back-propagated through the length of the sequence that has been set to learn.

The repeating unit of an LSTM is not the same as that of a simple or Vanilla RNN. As opposed to a single neural net unit in the simple RNN, the LSTM have four neural units used to enhance the learning of long term dependencies. To understand the structure and the shortcoming of the simple RNN the problem of exploding gradients and vanishing gradient is elucidated in section 1.1.1, the structure and the nuances of the LSTM architecture is underscored in the section 1.1.2.

The LSTM work well with data with long term dependences. Data such as music and language work well with its architecture. The English language is structured in a manner that each word generated depends on a length of words that precedes it. Hence generating language is a task that the LSTM are seldom used for.

3.3.2 Exploding and Vanishing Gradients

The problem of exploding and vanishing gradients appear as an effect of back propagation through a lengthy sequence. Since the value of the derivative decreases through every time step propagated backwards the value of the gradients become negligent and hence tend to move towards zero or vanish.

Conversely with a succinctly high learning rate the gradients can overshoot and continue to rise through the backward propagation. This is known as the exploding gradient problem and it gives erroneous results.

Both exploding and vanishing gradients are observed in the Vanilla RNN as it simply propagates the weights of the hidden layer into the hidden layer of the next time step. The LSTM net however uses four neural networks that act as gates to provide a context that can prevent the problems discussed above.

3.3.3 Structure of LSTM

The LSTM has four nets, the first net decides the information to be discarded as we propagate forward. To accomplish this a sigmoid activation is used. The sigmoid returns a value between 0 and 1 this is added to the context from the previous state.

The second step the LSTM performs is decide the context of the next step. To do so a sigmoid and tanh function is used, the sigmoid layer selects information to be updated and the tanh helps extrapolate candidate values that can be propagated forward. The values from both these nets are multiplied and the output is added to the context.

The final step is to generate the output of the present state. The output is generated by running the formed context through a tanh function and multiplying this with the output of the sigmoid layer.

The final context is the summation of the first three nets, while the last net helps create the hidden state to be sent forward.

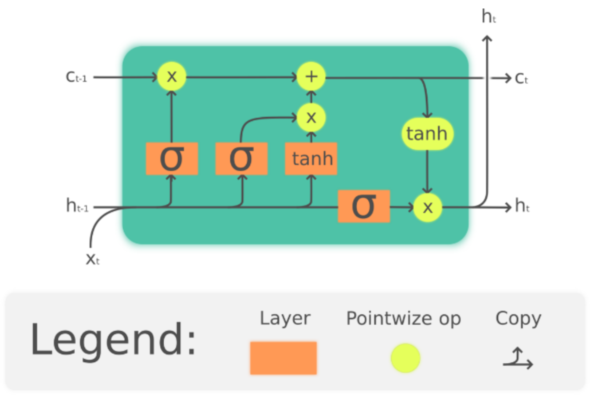


Figure 3.3: LSTM Cell

3.4 Generator

After we have determined the two main components: the encoder model and the language model, the generator module acts as the system in action. In a real world system, it is this module that will run to provide real time image captioning. It requires three components to independently work: a feature extractor model, a tokenizer object representing vocabulary, and a language model. All we have to do is plug these in and the system is good to go.

In the generator module, we can have a before unseen image. The encoder model then extracts the features of the image, after some basic preprocessing by the module. The photo feature is then passed to the language module. The language module takes in photo feature and the tokenizer object and produces the string description. The “start“ token is first fed as a sequence to initiate description. The sequences are of maximum length encountered by the model while training. The words available are first filled after which the sequence is zero-padded till the maximum length. The output is one-hot vector of the predicted word. The index with the highest probability is selected. The word at the specified index is extracted from the tokenizer object, and is consequently appended to the sequence, starting the next cycle of the generation. The process continues until the “end“ token or the maximum word limit is reached. Thus, the process is halted and the system outputs the string generated.

CHAPTER 4

FULL MODEL ARCHITECTURE

In this chapter, we define the actual layers to be used and various other components. In practicality, the architecture takes 2 inputs. An image feature set, and the sequence formed till now. Each caption starts with a “start“ token, which initiates generation of the rest of sentence, until the “end“ token or maximum caption length is reached.

For e.g., if an image has a ground truth caption “a dog is playing with ball“, then the input-output pairs to process the caption in Table 4.1

Table 4.1: Data Format

Input		Output
Photo Feature	Input Sequence	
P	"start"	"a"
P	"start a"	"dog"
P	"start a dog"	"is"
P	"start a dog is"	"playing"
P	"start a dog is playing"	"with"
P	"start a dog is playing with"	"ball"
P	"start a dog is playing with ball"	"end"

4.1 Types of Encoder-Decoder Architectures

4.1.1 Inject Architecture

Figure 4.1 shows the basic concept of inject architecture. In these, the LSTM receives both the image and sequence information while training. Intuitively, this means that the words are predicted using the visual cues from the image. At the same time, the image information is not necessary to learn the syntactic and structural rules of the language. Thus, it can add an overhead for an action that may not have a positive effect on the model.

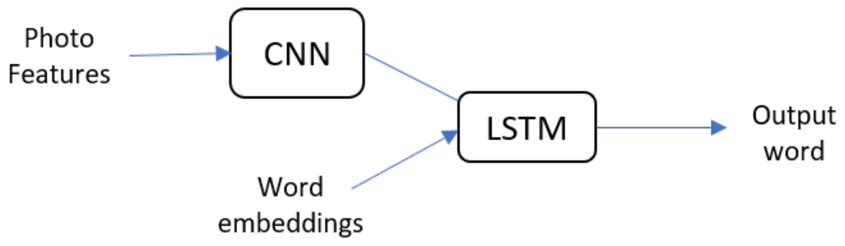


Figure 4.1: Inject Architecture

4.1.2 Merge Architecture

Figure 4.2 shows the merge architecture. In this type of architecture, the LSTM layer only processes the sequential information provided by the caption. Intuitively, it misses some important image information, while predicting the next word. At the same time, it is less complex, as there are less information to learn from. We experiment with both the above architectures to find out which factor has a more prominent effect.

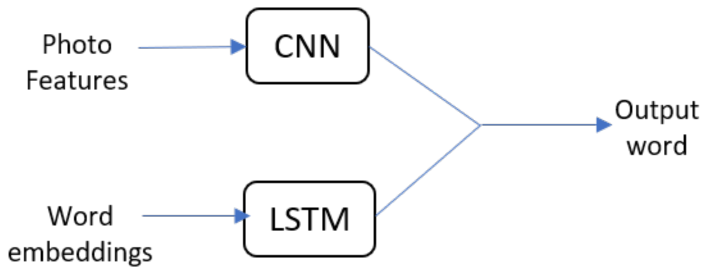


Figure 4.2: Merge Architecture

4.2 Other Components

While the two major components of the architecture are the CNN and LSTM layers, as they allow the network to learn such a complex problem, there are still some additional components, that enable the model to maintain the characteristics of a good machine learning system, such as regularization, and generalization.In this section, we will take a look at some of the other layers that we include in the model, for the reasons of making a more robust system.

4.2.1 Embedding Layer

Word Embeddings have been used in a lot of language related tasks, and is known to make good improvement. Embeddings were invented to have better word representations. Traditionally, words were represented using 1-in-k or one-hot-vector representations. In these, the word was determined by putting 1 in a specified index of a vocabulary sized vector. This had a two major disadvantages. Firstly, the sequences took a lot of space. In many NLP tasks, it is not uncommon to have vocabulary of millions of words. In these cases, the computational complexity can become unnecessarily large, as most of the vectors are empty. Secondly, the vector space defined by these representations do not have any semantic significance. For example. In these representations, the words “apple“ and “orange“ are as closely related as the pair of words “guitar“ and “sea“, i.e. to the model training on these sequences, both those words are separate, unrelated classes of objects. We, on the other hand know, that most of the sentences containing the word apple, will be valid sentences even if apple is replaced by orange. For example, “I like to eat apples“ and “I like to eat oranges“ are valid in the sense both are fruits. Thus one-hot encodings do not catch this hierarchy properly. Word embeddings were designed to do exactly that.

Word embeddings are a vector representation of words. We can think of them as scoring a word on a number of different properties such as colour, gender, living/non living etc. This system automatically makes vector of similar concepts similar as well. In this system, apple and orange will have similar values for properties such as gender (both being genderless), living (both are inanimate objects) and maybe the same score for a “food“ category. The vector space of such an embedding system would have clusters of objects that are similar. For example, all animal words will be closer to each other, and separate from human profession words. In addition, it becomes possible to have vector operations on these words that make semantic sense as well. For example, the vector operation “king“ - “boy“ + “girl“, would yield the representation that is closest to the word “queen“.

There are two ways to implement the concept of word embeddings. The first is to use pre-trained word embeddings available on the internet. These have been trained on corpus of millions of words or more, and can capture a wide range of concepts. They have an advantage in the case that you do not have a very large vocabulary and want a decent system. The word embeddings can recognize unencountered words and concepts and get a word that is closest to it, thus resulting in a robust and generalized system. Yet, they have known not to have significant performance boost in language generation tasks. In such cases, Keras provides Embedding layers, that can be trained on your specific task and are compatible to your specific network.

Since, our problem is about generation of natural captions, we have decided to go for the second option. The embedding layer is used to find the parameter “Embedding Matrix“ which learns the embeddings for every word in your vocabulary. It allows us to have an intermediate word representation, that makes the sequence learning a lot more efficient and better performing as compared to one-hot vector representation. It also reduces space.

4.2.2 Dropout

Dropout is the regularization technique used in Neural Networks. Regularization was introduced to counter the problem of overfitting in machine learning problems. Overfitting means that the model achieves very good training set performance, but poor test set scores. This means that your model fails to generalizes when encountering newer instances of a problem. In linear regression, a regularization term was introduced in loss functions, that mathematically reduced the effect of the higher degree terms that could create an overfitting polynomial curve.

Dropout layer acts as a regularization mechanism in neural networks, by randomly switching of a fraction of units of a particular layer in each forward propagation and weight update cycle. As a result, those units do not affect the output and weights for that particular cycle. While defining the dropout layer, we can choose the fraction of units we wish to ignore. For our problem, we have used a 0.5 value. We have placed 2 Dropout Layers: one after processing the image features, and one after the embedding Layers. These are the layers closely related to the inputs and thus are good candidates for regularization. It is important to note, that overfitting is very easy in our particular problem, which can be seen by the fact that not even state-of-the-art models have yet achieved a good overall test set performance.

4.2.3 Activation and Loss Functions

Activation functions in a neural network are a mechanism to provide non linearity and the ability to compute complex functions to the network. In this manner, they are the life of the concept. There are a variety of activation functions such as sigmoid, tanh and others. For the intermediate layers, we have used Rectified Linear Unit as activation. It has the advantage over tanh and Sigmoid of avoiding the vanishing gradient problem. Vanishing gradient problem causes problem in learning a function by having too small a gradient for having significant weight updates. Relu ensures non zero gradient and thus solves the vanishing gradient problem.

For our output layer, we have used Softmax activation. Softmax activation outputs a set of numbers that sum to 1, essentially giving a probability distribution. This is helpful when learning a classification problem with multiple classes, as it calculates the probability of each. Since our problem involves predicting the next word from a number of possible words (vocabulary), it is essentially a multi-class classification problem as well.

We have used a categorical crossentropy loss function for our problem. Categorical crossentropy compares the predicted class and the expected class, and sums it all over the training examples. Thus, this loss function modifies weights in a manner so that the correct class is selected. At each step, we predict one word and add it to the existing sequence. Thus, the number of classes is vocabulary size.

CHAPTER 5

EXPERIMENTS AND PARAMETERS

5.1 Finetuning

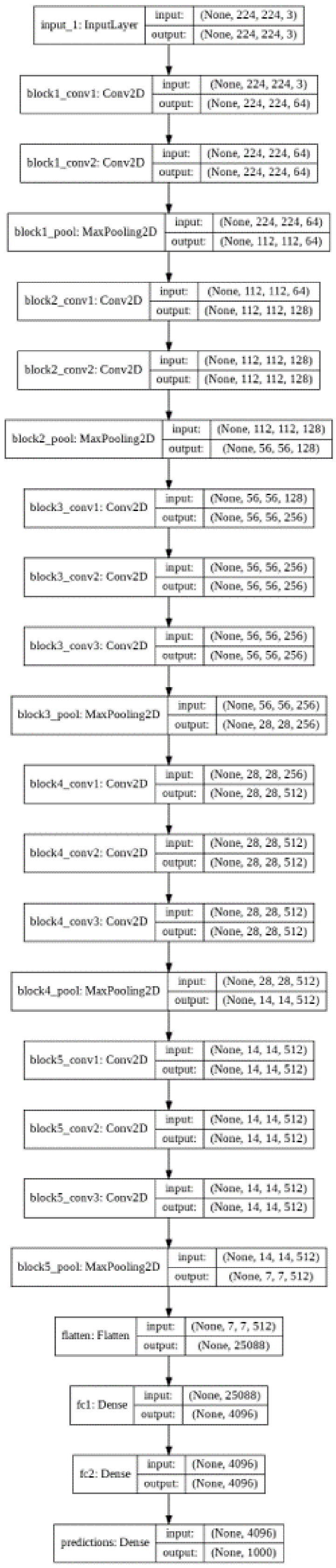
Fine tuning is the method in which we use pre-defined architectures and train it on our own dataset. Usually the last layer of the CNN is removed and the layer predicting our personal classes is inserted. Using a lower learning rate can provide a significant boost in performance. This is because the nets have already been initialized and a lower learning rate will move the net to convergence faster and with a better efficiency.

Another method of fine tuning includes freezing of the initial layers, this helps because most of the initial layers of the CNN account for a higher feature map of the visual context, and hence they are not as significant as the lower ones which account for finer details. We experimented and fine-tuned the VGG16 model.

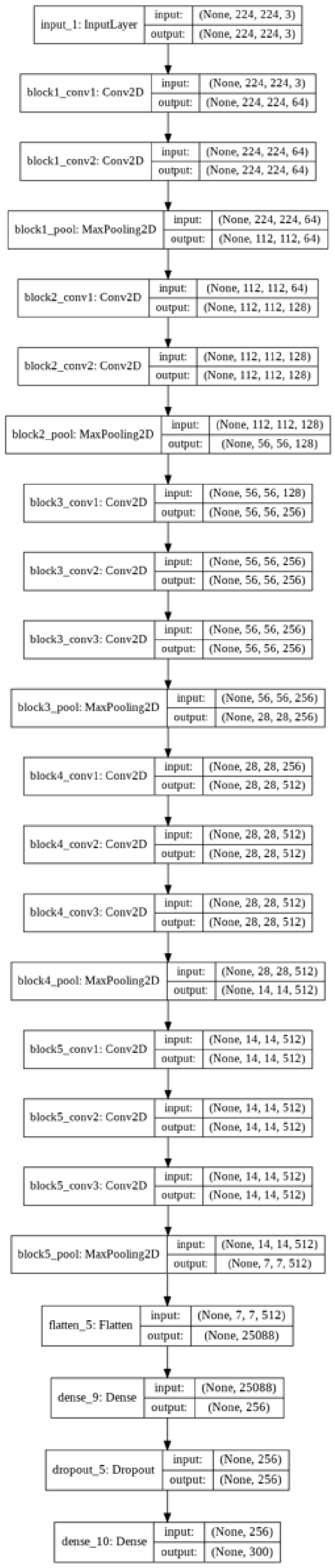
Fine-tuning of pre-trained CNN models is done to get better classification results for a particular dataset. The architecture of the pre-trained CNN is modified according to the number of classes of the new dataset. The new and unknown weights cannot be initialized randomly, thus we initialize weights by training of the network with original layers frozen.

After initialization, some of the layers are unfrozen and then training is done normally. As the CNNs are trained for classiïňĄcation, we labelled the images of the Flickr8k dataset ourselves. From the captions, the nouns were extracted and manually reïňĄned.

The 300 most common nouns were taken. For each image, target labels were generated by checking occurrence of each noun in any of the 5 captions. In addition, the architecture of the CNN was modiïňĄed to perform multi-label classiïňĄcation instead of single class classiïňĄcation. The feature space was kept at original size. Fine-tuning performed multi-label classiïňĄcation decently on the training set.


input_1: InputLayer | input: (None, 224, 224, 3) | output: (None, 224, 224, 3)
block1_conv1: Conv2D | input: (None, 224, 224, 3) | output: (None, 224, 224, 64)
block1_conv2: Conv2D | input: (None, 224, 224, 64) | output: (None, 224, 224, 64)
block1_pool: MaxPooling2D | input: (None, 224, 224, 64) | output: (None, 112, 112, 64)
block2_conv1: Conv2D | input: (None, 112, 112, 64) | output: (None, 112, 112, 128)
block2_conv2: Conv2D | input: (None, 112, 112, 128) | output: (None, 112, 112, 128)
block2_pool: MaxPooling2D | input: (None, 112, 112, 128) | output: (None, 56, 56, 128)
block3_conv1: Conv2D | input: (None, 56, 56, 128) | output: (None, 56, 56, 256)
block3_conv2: Conv2D | input: (None, 56, 56, 256) | output: (None, 56, 56, 256)
block3_conv3: Conv2D | input: (None, 56, 56, 256) | output: (None, 56, 56, 256)
block3_pool: MaxPooling2D | input: (None, 56, 56, 256) | output: (None, 28, 28, 256)
block4_conv1: Conv2D | input: (None, 28, 28, 256) | output: (None, 28, 28, 512)
block4_conv2: Conv2D | input: (None, 28, 28, 512) | output: (None, 28, 28, 512)
block4_conv3: Conv2D | input: (None, 28, 28, 512) | output: (None, 28, 28, 512)
block4_pool: MaxPooling2D | input: (None, 28, 28, 512) | output: (None, 14, 14, 512)
block5_conv1: Conv2D | input: (None, 14, 14, 512) | output: (None, 14, 14, 512)
block5_conv2: Conv2D | input: (None, 14, 14, 512) | output: (None, 14, 14, 512)
block5_conv3: Conv2D | input: (None, 14, 14, 512) | output: (None, 14, 14, 512)
block5_pool: MaxPooling2D | input: (None, 14, 14, 512) | output: (None, 7, 7, 512)
flatten: Flatten | input: (None, 7, 7, 512) | output: (None, 25088)
fc1: Dense | input: (None, 25088) | output: (None, 4096)
fc2: Dense | input: (None, 4096) | output: (None, 4096)
predictions: Dense | input: (None, 4096) | output: (None, 1000)

(a) Original VGG16 Architecture


input_1: InputLayer | input: (None, 224, 224, 3) | output: (None, 224, 224, 3)
block1_conv1: Conv2D | input: (None, 224, 224, 3) | output: (None, 224, 224, 64)
block1_conv2: Conv2D | input: (None, 224, 224, 64) | output: (None, 224, 224, 64)
block1_pool: MaxPooling2D | input: (None, 224, 224, 64) | output: (None, 112, 112, 64)
block2_conv1: Conv2D | input: (None, 112, 112, 64) | output: (None, 112, 112, 128)
block2_conv2: Conv2D | input: (None, 112, 112, 128) | output: (None, 112, 112, 128)
block2_pool: MaxPooling2D | input: (None, 112, 112, 128) | output: (None, 56, 56, 128)
block3_conv1: Conv2D | input: (None, 56, 56, 128) | output: (None, 56, 56, 256)
block3_conv2: Conv2D | input: (None, 56, 56, 256) | output: (None, 56, 56, 256)
block3_conv3: Conv2D | input: (None, 56, 56, 256) | output: (None, 56, 56, 256)
block3_pool: MaxPooling2D | input: (None, 56, 56, 256) | output: (None, 28, 28, 256)
block4_conv1: Conv2D | input: (None, 28, 28, 256) | output: (None, 28, 28, 512)
block4_conv2: Conv2D | input: (None, 28, 28, 512) | output: (None, 28, 28, 512)
block4_conv3: Conv2D | input: (None, 28, 28, 512) | output: (None, 28, 28, 512)
block4_pool: MaxPooling2D | input: (None, 28, 28, 512) | output: (None, 14, 14, 512)
block5_conv1: Conv2D | input: (None, 14, 14, 512) | output: (None, 14, 14, 512)
block5_conv2: Conv2D | input: (None, 14, 14, 512) | output: (None, 14, 14, 512)
block5_conv3: Conv2D | input: (None, 14, 14, 512) | output: (None, 14, 14, 512)
block5_pool: MaxPooling2D | input: (None, 14, 14, 512) | output: (None, 7, 7, 512)
flatten_5: Flatten | input: (None, 7, 7, 512) | output: (None, 25088)
dense_9: Dense | input: (None, 25088) | output: (None, 256)
dropout_5: Dropout | input: (None, 256) | output: (None, 256)
dense_10: Dense | input: (None, 256) | output: (None, 300)

(b) Modified VGG16 Architecture

Figure 5.1: Finetuning VGG16

with 99.65 of the labels of each image on average being correctly identiïňĄed. Fine tuning helped boast performance and stands as our best model. The fine tuning experiment was able to better the performance of the model by predicting visual context which were at par and concerned with the vocabulary of our dataset. We further used the fine training on Inception V3 model but due to the batch normalization problem with regard to this architecture, the accuracy of the tuning drops. Although most pre-trained model support transfer learning, fine tuning the net to your dataset makes the system more robust and efficient. Despite the enormous size of the Image net, many of the images in the dataset for captioning fail to fall in the 1000 classes, hence it makes sense to create a personal classification task for our own data.

5.2 Reduced Vocabulary

Another experiment that has proven to improve results is the reduction of vocabulary. After the preliminary task of cleaning the captions which included removing punctuations, converting to lower case, adding <start> and <end> tags etc. We went ahead and removed words that do not occur frequently in our dataset. This proved to improve performance as well as decrease training time. The reason to reduce vocabulary was because infrequent words won‘t occur enough times in the dataset to be learned well enough by the model. Removing these words further did not affect the semantic relationships in the captions of our dataset and the generated captions continued to be coherent and comprehensive.

Reducing vocabulary also decreases training time because it reduces the size of the embedding. Since we go through an ample amount of data word by word through the LSTM for the task of image captioning, a succinct embedding layer immensely helps the training time.

Although it is a good idea to reduce vocabulary, the improvement shown by the model by the application of such a method is not as significant as the results shown through fine-tuning. However the increment in performance cannot be ignored and proves to be a good augment to our final model.

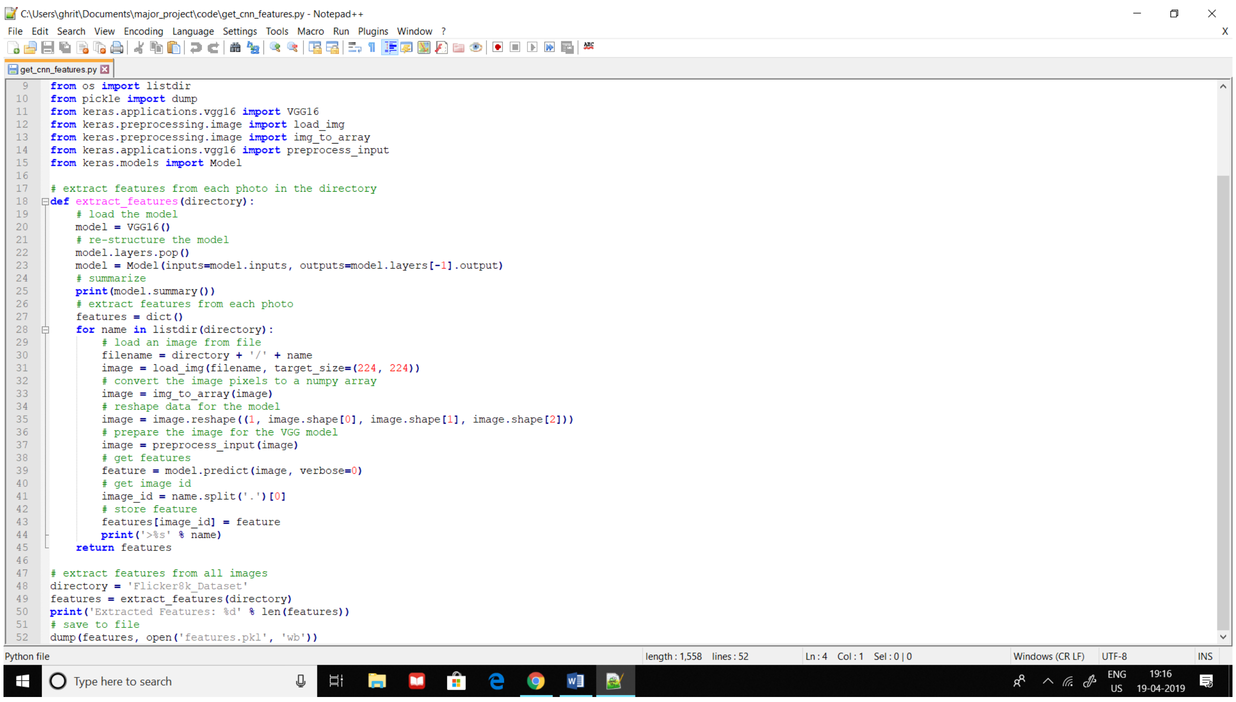
Further the reduction of vocabulary helps with drawing a distinction between the various architectures that we have tried. It was observed that the reduction of vocabulary helped more in the case of the merge architectures than with respect to inject architecture.

CHAPTER 6

CODE

6.1 Encoder

We ran two variations for acquiring our image feature vector. In both instances we used the VGG16 model. We first trained our images using pre trained weights. Since the last layer is the classification layer and this does not concern the task of image captioning, we popped the last layer and worked with the fully connected layer that returns a vector of 4096.

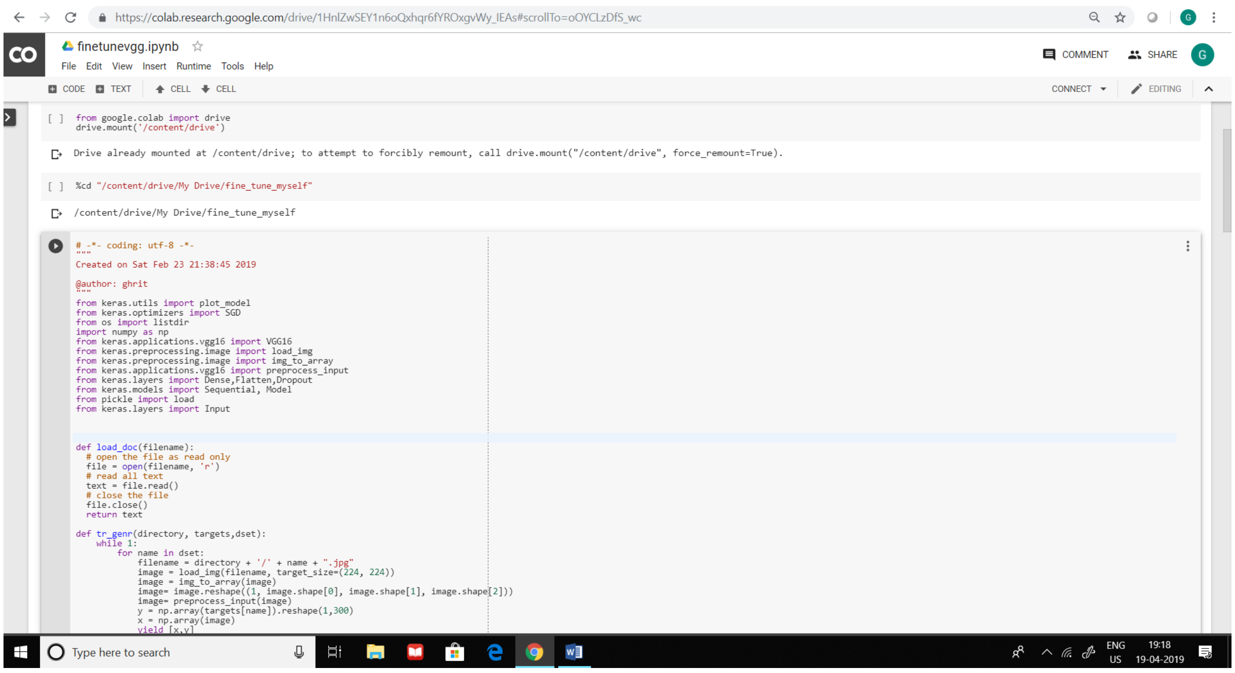


```python
from os import listdir
from pickle import dump
from keras.applications.vgg16 import VGG16
from keras.preprocessing.image import load_img
from keras.preprocessing.image import img_to_array
from keras.applications.vgg16 import preprocess_input
from keras.models import Model

# extract features from each photo in the directory
def extract_features(directory):
    # load the model
    model = VGG16()
    # re-structure the model
    model.layers.pop()
    model = Model(inputs=model.inputs, outputs=model.layers[-1].output)
    # summarize
    print(model.summary())
    # extract features from each photo
    features = dict()
    for name in listdir(directory):
        # load an image from file
        filename = directory + '/' + name
        image = load_img(filename, target_size=(224, 224))
        # convert the image pixels to a numpy array
        image = img_to_array(image)
        # reshape data for the model
        image = image.reshape((1, image.shape[0], image.shape[1], image.shape[2]))
        # prepare the image for the VGG model
        image = preprocess_input(image)
        # get features
        feature = model.predict(image, verbose=0)
        # get image id
        image_id = name.split('.')[0]
        # store feature
        features[image_id] = feature
        print('>%s' % name)
    return features

# extract features from all images
directory = 'Flicker8k_Dataset'
features = extract_features(directory)
print('Extracted Features: %d' % len(features))
# save to file
dump(features, open('features.pkl', 'wb'))
```

C:\Users\ghrit\Documents\major_project\code\get_cnn_features.py - Notepad++
File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window ?
get_cnn_features.py
Python file
length : 1,558 lines : 52
Ln : 4 Col : 1 Sel : 0 | 0
Windows (CR LF) UTF-8 INS
Type here to search
ENG US 19:16 19-04-2019

Figure 6.1: Feature Extraction

The second variation is the fine tuning of VGG net, instead of using the pre trained weights, we acquired weights by creating our own classification layer and training it on our personal dataset.


https://colab.research.google.com/drive/1HnlZwSEY1n6oQxhqr6fYROxgvWy_IEAs#scrollTo=oOYCLzDfS_wc
finetunevgg.ipynb
File Edit View Insert Runtime Tools Help
CODE TEXT CELL CELL
COMMENT SHARE
CONNECT EDITING
Type here to search
ENG US 19:18 19-04-2019

```
from google.colab import drive
drive.mount('/content/drive')
```

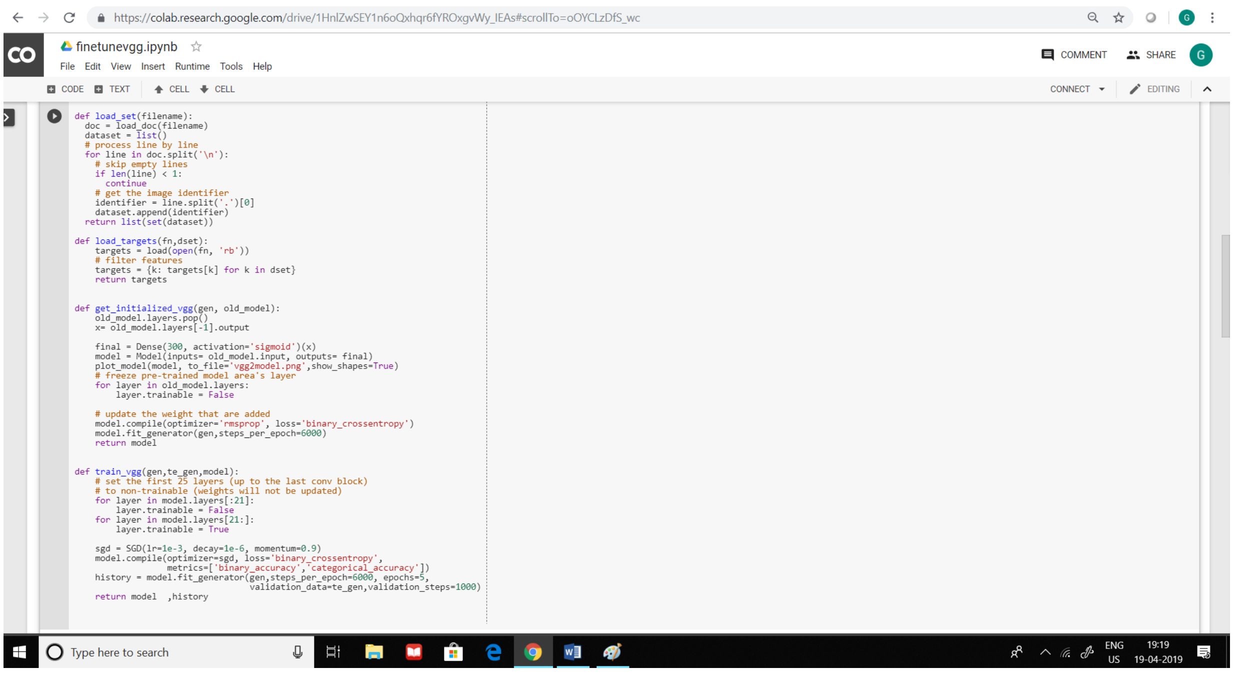
```
Drive already mounted at /content/drive; to attempt to forcibly remount, call drive.mount("/content/drive", force_remount=True).
```

```
%cd "/content/drive/My Drive/fine_tune_myself"
```

```
/content/drive/My Drive/fine_tune_myself
```

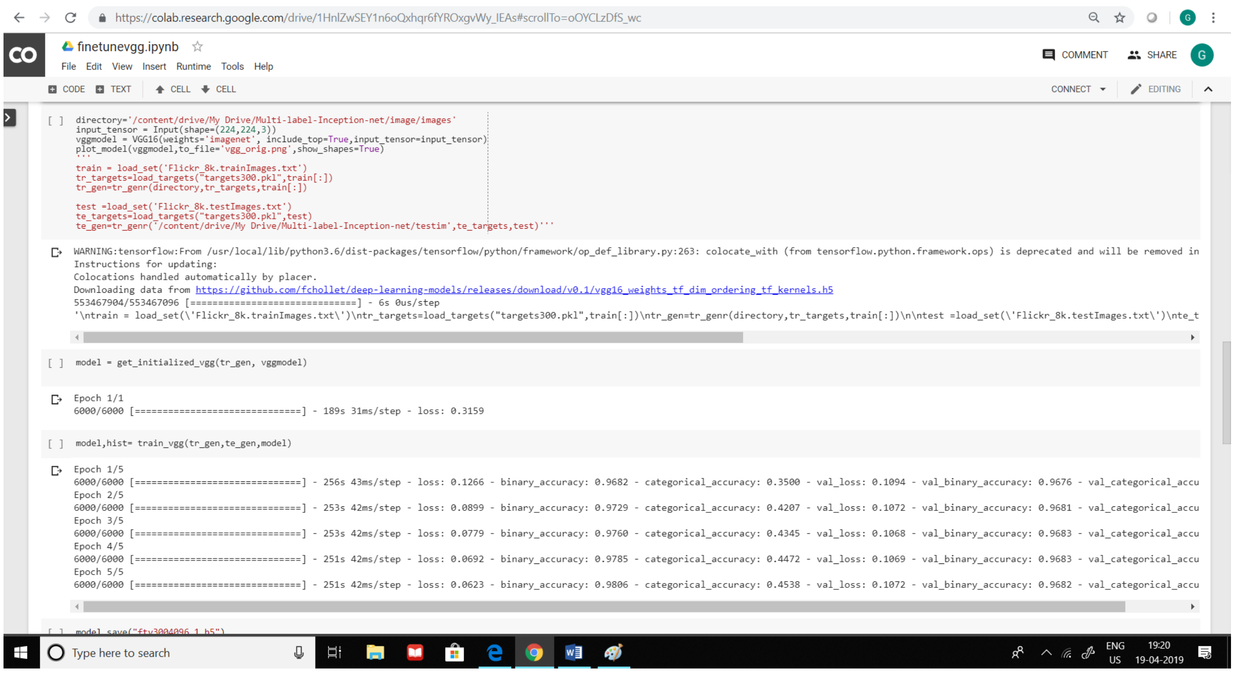
```
# -*- coding: utf-8 -*-
"""
Created on Sat Feb 23 21:38:45 2019

@author: ghrit
"""
from keras.utils import plot_model
from keras.optimizers import SGD
from os import listdir
import numpy as np
from keras.applications.vgg16 import VGG16
from keras.preprocessing.image import load_img
from keras.preprocessing.image import img_to_array
from keras.applications.vgg16 import preprocess_input
from keras.layers import Dense,Flatten,Dropout
from keras.models import Sequential, Model
from pickle import load
from keras.layers import Input


def load_doc(filename):
  # open the file as read only
  file = open(filename, 'r')
  # read all text
  text = file.read()
  # close the file
  file.close()
  return text

def tr_genr(directory, targets,dset):
    while 1:
        for name in dset:
            filename = directory + '/' + name + ".jpg"
            image = load_img(filename, target_size=(224, 224))
            image = img_to_array(image)
            image= image.reshape((1, image.shape[0], image.shape[1], image.shape[2]))
            image= preprocess_input(image)
            y = np.array(targets[name]).reshape(1,300)
            x = np.array(image)
            yield [x,y]
```

(a)


https://colab.research.google.com/drive/1HnlZwSEY1n6oQxhqr6fYROxgvWy_IEAs#scrollTo=oOYCLzDfS_wc
finetunevgg.ipynb
File Edit View Insert Runtime Tools Help
CODE TEXT CELL CELL
COMMENT SHARE
CONNECT EDITING
Type here to search
ENG US 19:19 19-04-2019

```
def load_set(filename):
  doc = load_doc(filename)
  dataset = list()
  # process line by line
  for line in doc.split('\n'):
    # skip empty lines
    if len(line) < 1:
      continue
    # get the image identifier
    identifier = line.split('.')[0]
    dataset.append(identifier)
  return list(set(dataset))

def load_targets(fn,dset):
    targets = load(open(fn, 'rb'))
    # filter features
    targets = {k: targets[k] for k in dset}
    return targets


def get_initialized_vgg(gen, old_model):
    old_model.layers.pop()
    x= old_model.layers[-1].output

    final = Dense(300, activation='sigmoid')(x)
    model = Model(inputs= old_model.input, outputs= final)
    plot_model(model, to_file='vgg2model.png',show_shapes=True)
    # freeze pre-trained model area's layer
    for layer in old_model.layers:
        layer.trainable = False

    # update the weight that are added
    model.compile(optimizer='rmsprop', loss='binary_crossentropy')
    model.fit_generator(gen,steps_per_epoch=6000)
    return model


def train_vgg(gen,te_gen,model):
    # set the first 25 layers (up to the last conv block)
    # to non-trainable (weights will not be updated)
    for layer in model.layers[:21]:
        layer.trainable = False
    for layer in model.layers[21:]:
        layer.trainable = True

    sgd = SGD(lr=1e-3, decay=1e-6, momentum=0.9)
    model.compile(optimizer=sgd, loss='binary_crossentropy',
                  metrics=['binary_accuracy','categorical_accuracy'])
    history = model.fit_generator(gen,steps_per_epoch=6000, epochs=5,
                                  validation_data=te_gen,validation_steps=1000)
    return model  ,history
```

(b)



```
[ ] directory='/content/drive/My Drive/Multi-label-Inception-net/image/images'
    input_tensor = Input(shape=(224,224,3))
    vggmodel = VGG16(weights='imagenet', include_top=True,input_tensor=input_tensor)
    plot_model(vggmodel,to_file='vgg_orig.png',show_shapes=True)
    '''
    train = load_set('Flickr_8k.trainImages.txt')
    tr_targets=load_targets("targets300.pkl",train[:])
    tr_gen=tr_genr(directory,tr_targets,train[:])

    test =load_set('Flickr_8k.testImages.txt')
    te_targets=load_targets("targets300.pkl",test)
    te_gen=tr_genr('/content/drive/My Drive/Multi-label-Inception-net/testim',te_targets,test)'''
```

```
WARNING:tensorflow:From /usr/local/lib/python3.6/dist-packages/tensorflow/python/framework/op_def_library.py:263: colocate_with (from tensorflow.python.framework.ops) is deprecated and will be removed in
Instructions for updating:
Colocations handled automatically by placer.
Downloading data from https://github.com/fchollet/deep-learning-models/releases/download/v0.1/vgg16_weights_tf_dim_ordering_tf_kernels.h5
553467904/553467096 [==============================] - 6s 0us/step
'\ntrain = load_set(\'Flickr_8k.trainImages.txt\')\ntr_targets=load_targets("targets300.pkl",train[:])\ntr_gen=tr_genr(directory,tr_targets,train[:])\n\ntest =load_set(\'Flickr_8k.testImages.txt\')\nte_t
```

```
[ ] model = get_initialized_vgg(tr_gen, vggmodel)
```

```
Epoch 1/1
6000/6000 [==============================] - 189s 31ms/step - loss: 0.3159
```

```
[ ] model,hist= train_vgg(tr_gen,te_gen,model)
```

```
Epoch 1/5
6000/6000 [==============================] - 256s 43ms/step - loss: 0.1266 - binary_accuracy: 0.9682 - categorical_accuracy: 0.3500 - val_loss: 0.1094 - val_binary_accuracy: 0.9676 - val_categorical_accu
Epoch 2/5
6000/6000 [==============================] - 253s 42ms/step - loss: 0.0899 - binary_accuracy: 0.9729 - categorical_accuracy: 0.4207 - val_loss: 0.1072 - val_binary_accuracy: 0.9681 - val_categorical_accu
Epoch 3/5
6000/6000 [==============================] - 253s 42ms/step - loss: 0.0779 - binary_accuracy: 0.9760 - categorical_accuracy: 0.4345 - val_loss: 0.1068 - val_binary_accuracy: 0.9683 - val_categorical_accu
Epoch 4/5
6000/6000 [==============================] - 251s 42ms/step - loss: 0.0692 - binary_accuracy: 0.9785 - categorical_accuracy: 0.4472 - val_loss: 0.1069 - val_binary_accuracy: 0.9683 - val_categorical_accu
Epoch 5/5
6000/6000 [==============================] - 251s 42ms/step - loss: 0.0623 - binary_accuracy: 0.9806 - categorical_accuracy: 0.4538 - val_loss: 0.1072 - val_binary_accuracy: 0.9682 - val_categorical_accu
```

```
[ ] model.save("ftv3004096_1.h5")
```

(c)

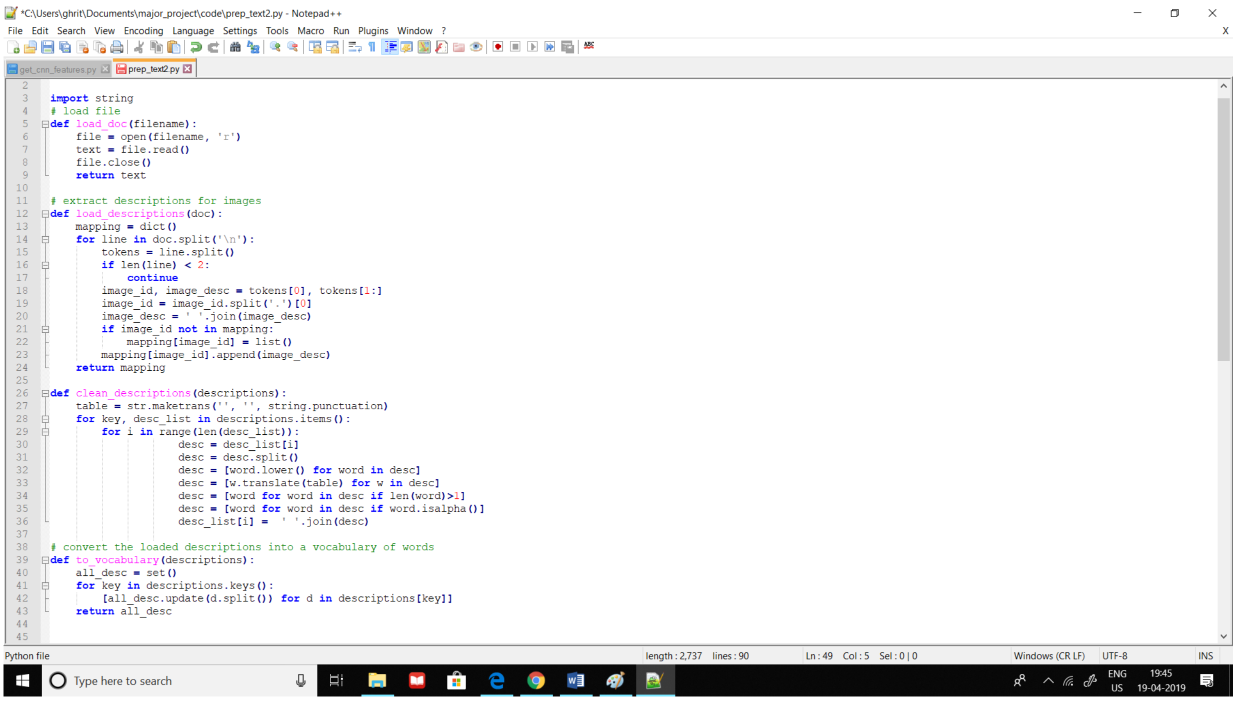
Figure 6.1: Finetuning

6.2 Pre-Processing Captions

Each caption was stripped off of punctuations such as commas and full stops, words of length less than 3 were also removed since they do not perturb the captionâĂŹs meaning and they remain comprehensible. Every word was converted to lower case to make the captions consistent, and a <start> and <end> tag was added to each caption to mark the beginning and termination of training. Furthermore we removed words which were rarely occurring in our dataset. Removing infrequently occurring words proved to boast performance, this can be because rare words do not occur enough times in our datasets to be properly learned by our model.

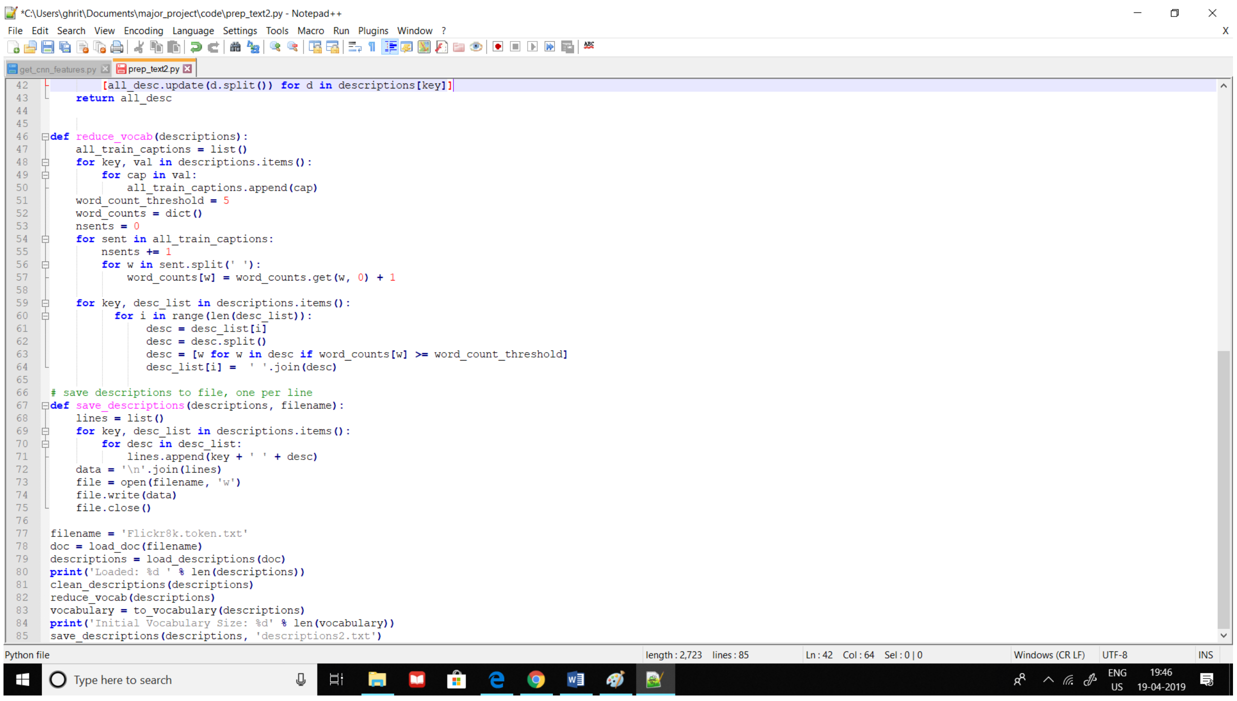
6.3 Sequencing of Data

The data needs to be sequenced to be trained word by word through our LSTM. Each image has five captions. The maximum length of the caption is 34 without the reduction of vocabulary and 33 with the reduction of vocabulary. We pad the captions that do not reach the length of the longest caption. Each word is converted to a one hot vector and



```
import string
# load file
def load_doc(filename):
    file = open(filename, 'r')
    text = file.read()
    file.close()
    return text

# extract descriptions for images
def load_descriptions(doc):
    mapping = dict()
    for line in doc.split('\n'):
        tokens = line.split()
        if len(line) < 2:
            continue
        image_id, image_desc = tokens[0], tokens[1:]
        image_id = image_id.split('.')[0]
        image_desc = ' '.join(image_desc)
        if image_id not in mapping:
            mapping[image_id] = list()
        mapping[image_id].append(image_desc)
    return mapping

def clean_descriptions(descriptions):
    table = str.maketrans('', '', string.punctuation)
    for key, desc_list in descriptions.items():
        for i in range(len(desc_list)):
                desc = desc_list[i]
                desc = desc.split()
                desc = [word.lower() for word in desc]
                desc = [w.translate(table) for w in desc]
                desc = [word for word in desc if len(word)>1]
                desc = [word for word in desc if word.isalpha()]
                desc_list[i] =  ' '.join(desc)

# convert the loaded descriptions into a vocabulary of words
def to_vocabulary(descriptions):
    all_desc = set()
    for key in descriptions.keys():
        [all_desc.update(d.split()) for d in descriptions[key]]
    return all_desc
```

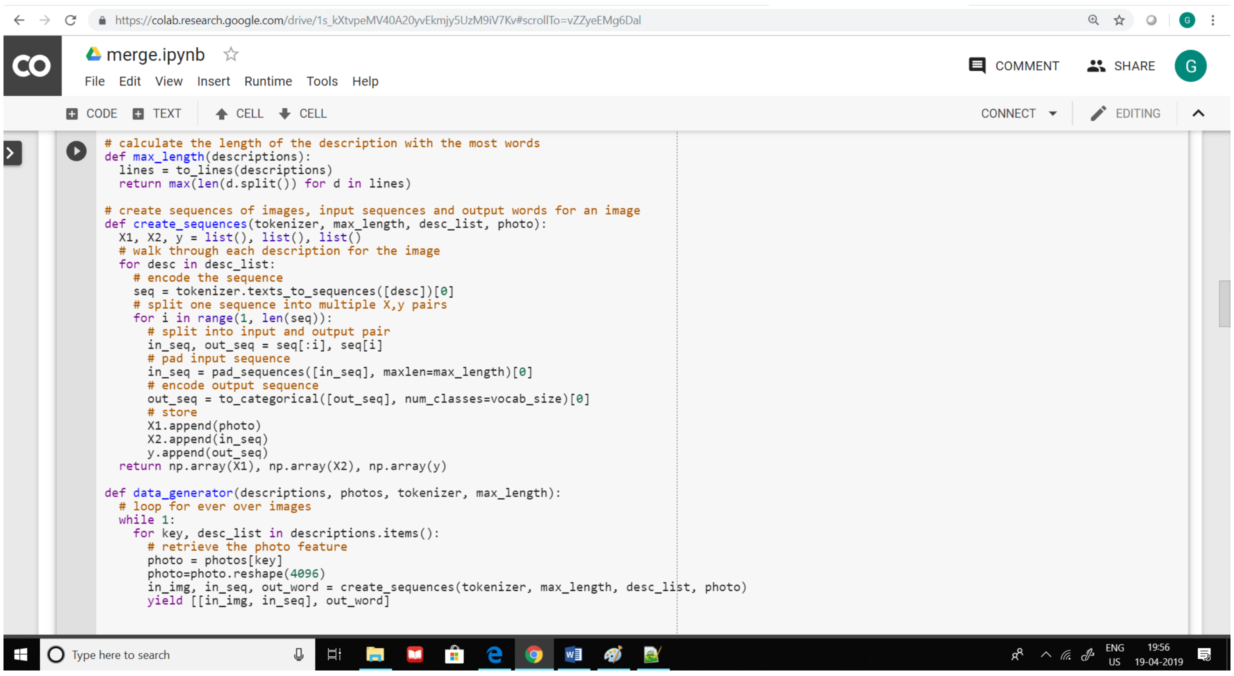
(d)



```
        [all_desc.update(d.split()) for d in descriptions[key]]
    return all_desc


def reduce_vocab(descriptions):
    all_train_captions = list()
    for key, val in descriptions.items():
        for cap in val:
            all_train_captions.append(cap)
    word_count_threshold = 5
    word_counts = dict()
    nsents = 0
    for sent in all_train_captions:
        nsents += 1
        for w in sent.split(' '):
            word_counts[w] = word_counts.get(w, 0) + 1

    for key, desc_list in descriptions.items():
          for i in range(len(desc_list)):
              desc = desc_list[i]
              desc = desc.split()
              desc = [w for w in desc if word_counts[w] >= word_count_threshold]
              desc_list[i] =  ' '.join(desc)

# save descriptions to file, one per line
def save_descriptions(descriptions, filename):
    lines = list()
    for key, desc_list in descriptions.items():
        for desc in desc_list:
            lines.append(key + ' ' + desc)
    data = '\n'.join(lines)
    file = open(filename, 'w')
    file.write(data)
    file.close()

filename = 'Flickr8k.token.txt'
doc = load_doc(filename)
descriptions = load_descriptions(doc)
print('Loaded: %d ' % len(descriptions))
clean_descriptions(descriptions)
reduce_vocab(descriptions)
vocabulary = to_vocabulary(descriptions)
print('Initial Vocabulary Size: %d' % len(vocabulary))
save_descriptions(descriptions, 'descriptions2.txt')
```

(e)

Figure 6.2: Text Preprocessing



```python
# 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)

# create sequences of images, input sequences and output words for an image
def create_sequences(tokenizer, max_length, desc_list, photo):
  X1, X2, y = list(), list(), list()
  # walk through each description for the image
  for desc in desc_list:
    # encode the sequence
    seq = tokenizer.texts_to_sequences([desc])[0]
    # split one sequence into multiple X,y pairs
    for i in range(1, len(seq)):
      # split into input and output pair
      in_seq, out_seq = seq[:i], seq[i]
      # pad input sequence
      in_seq = pad_sequences([in_seq], maxlen=max_length)[0]
      # encode output sequence
      out_seq = to_categorical([out_seq], num_classes=vocab_size)[0]
      # store
      X1.append(photo)
      X2.append(in_seq)
      y.append(out_seq)
  return np.array(X1), np.array(X2), np.array(y)

def data_generator(descriptions, photos, tokenizer, max_length):
  # loop for ever over images
  while 1:
    for key, desc_list in descriptions.items():
      # retrieve the photo feature
      photo = photos[key]
      photo=photo.reshape(4096)
      in_img, in_seq, out_word = create_sequences(tokenizer, max_length, desc_list, photo)
      yield [[in_img, in_seq], out_word]
```

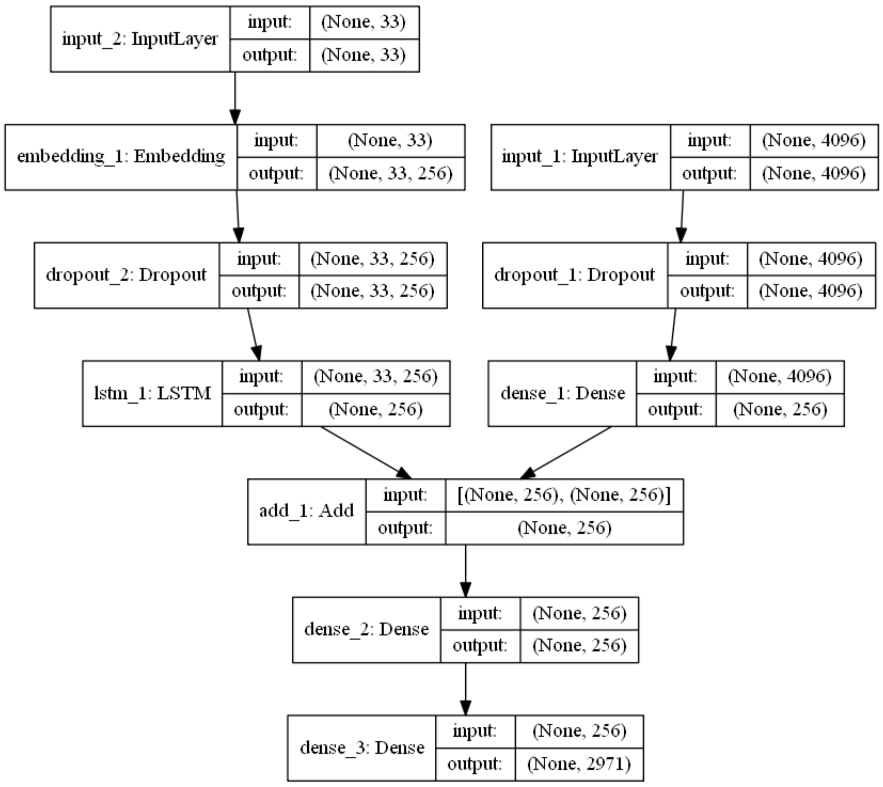
Figure 6.3: Sequence Generation

fed word by word. The photo features are also added for every time step of the LSTM. The sequencing is made easier using the tokenizer which streamlines are data.

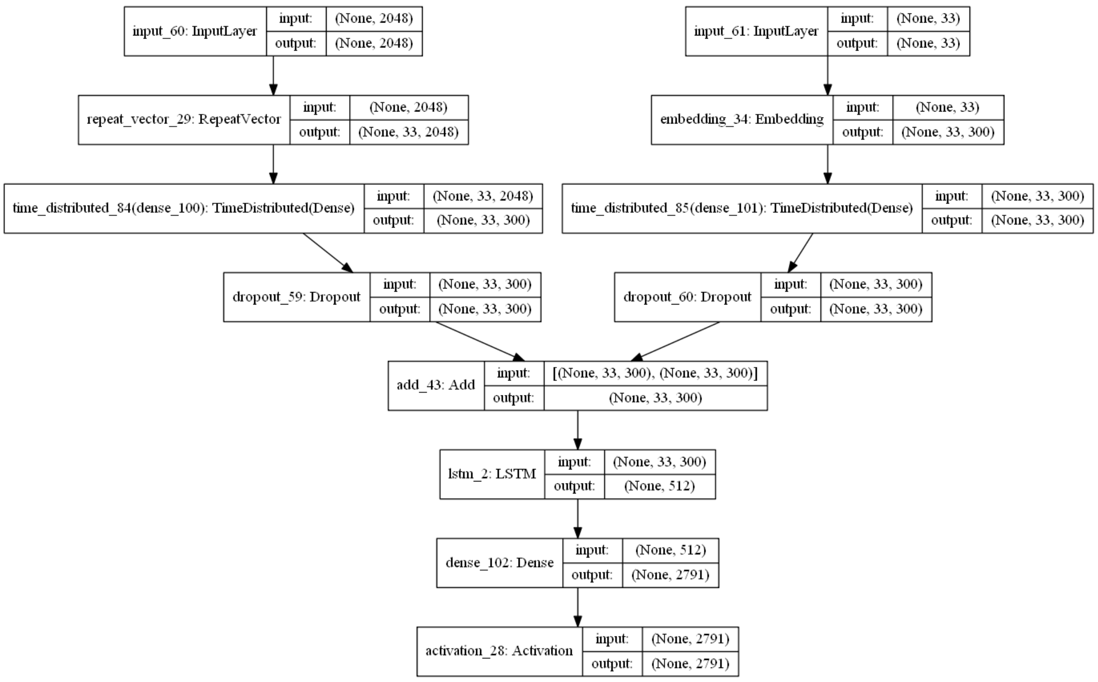
6.4 Model Architecture

Many architectures were experimented with, which includes the feeding both image and language features through the LSTM, this is known as an inject architecture. Another architecture that fared better than the inject model is the merge model, in which the language features are fed separately into the LSTM, the final prediction is made by a dense layer in merging both the image features and the output sequence from the LSTM.

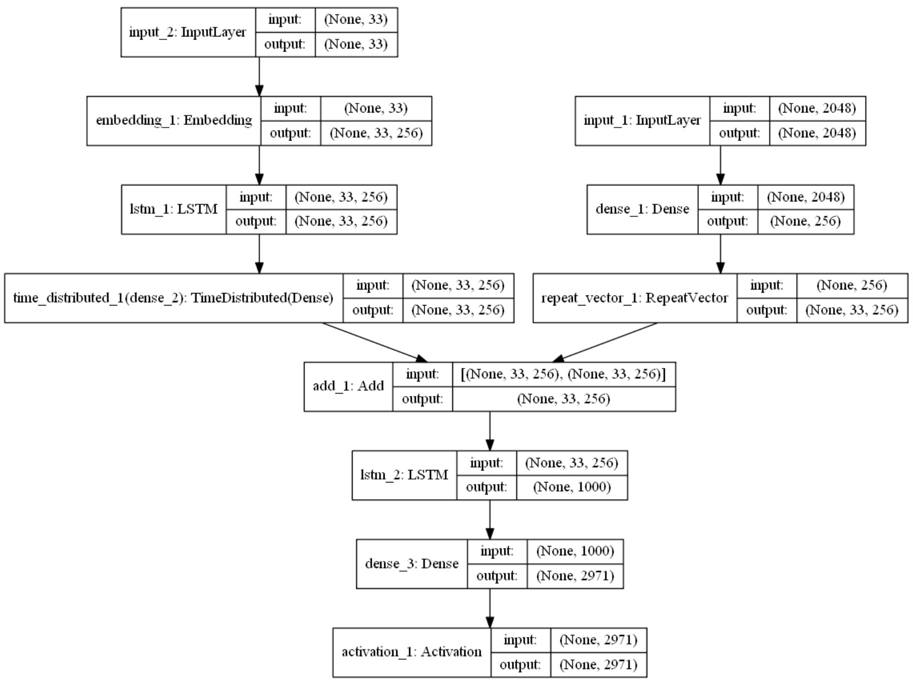
Figure 6.3 shows the plotted images obtained using a keras library showing the various architectures we defined for our model. These only represent the basic skeleton we have used. For each type, we have changed hyperparameters, vocabulary sizes, layer orders etc., which we will omit here. Note the placement of LSTM layers in each. Composite architecture is a mix of both merge and inject types, and was trained but was over-performed by merge.



(a) Merge



(b) Inject

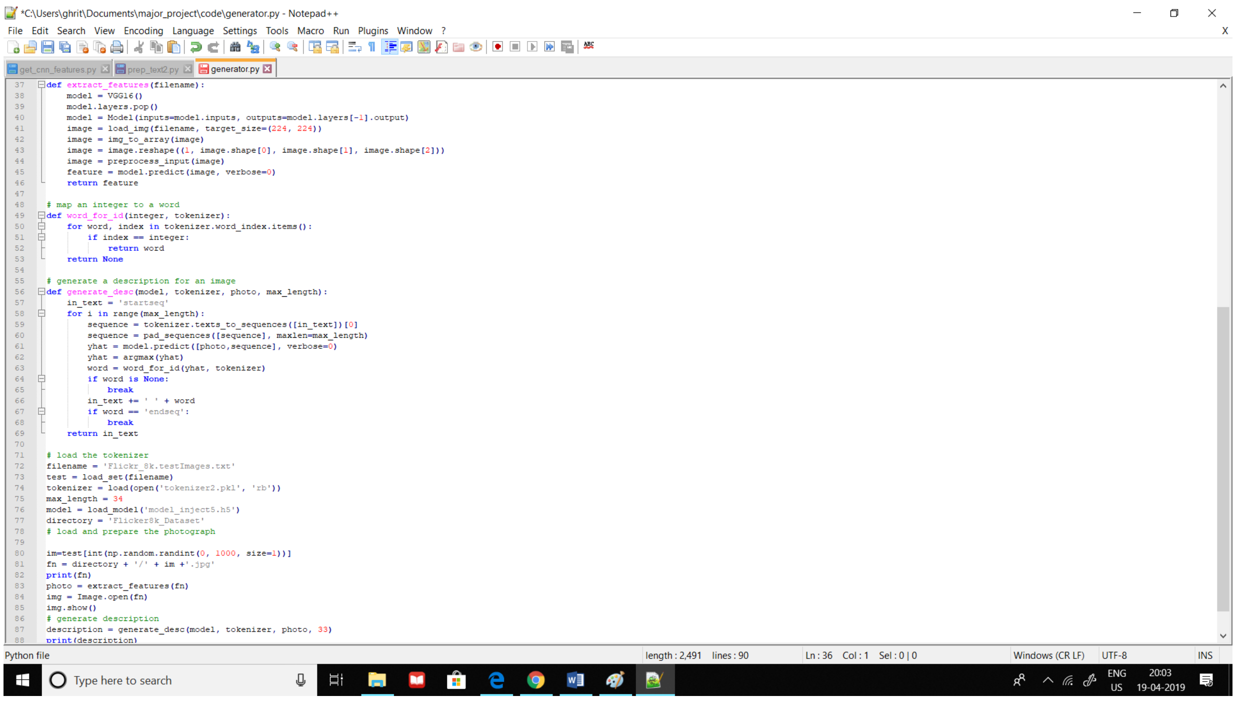


(c)

Figure 6.3: Plots of model architectures

6.5 Generation

The generation is done word by word. The test image is fed with the <start> tag and the generation continues until we encounter the <end> tag. The last layer of our model uses the Softmax activation, this returns a vector of predictions for the next word in our sequence. We use an argmax function which returns the word at the index of the most likely or the highest value in our output vector. This word is sequenced with our previous predictions and fed again into our model to make the next word predication.



```
def extract_features(filename):
    model = VGG16()
    model.layers.pop()
    model = Model(inputs=model.inputs, outputs=model.layers[-1].output)
    image = load_img(filename, target_size=(224, 224))
    image = img_to_array(image)
    image = image.reshape((1, image.shape[0], image.shape[1], image.shape[2]))
    image = preprocess_input(image)
    feature = model.predict(image, verbose=0)
    return feature

# map an integer to a word
def word_for_id(integer, tokenizer):
    for word, index in tokenizer.word_index.items():
        if index == integer:
            return word
    return None

# generate a description for an image
def generate_desc(model, tokenizer, photo, max_length):
    in_text = 'startseq'
    for i in range(max_length):
        sequence = tokenizer.texts_to_sequences([in_text])[0]
        sequence = pad_sequences([sequence], maxlen=max_length)
        yhat = model.predict([photo,sequence], verbose=0)
        yhat = argmax(yhat)
        word = word_for_id(yhat, tokenizer)
        if word is None:
            break
        in_text += ' ' + word
        if word == 'endseq':
            break
    return in_text

# load the tokenizer
filename = 'Flickr_8k.testImages.txt'
test = load_set(filename)
tokenizer = load(open('tokenizer2.pkl', 'rb'))
max_length = 34
model = load_model('model_inject5.h5')
directory = 'Flicker8k_Dataset'
# load and prepare the photograph

im=test[int(np.random.randint(0, 1000, size=1))]
fn = directory + '/' + im +'.jpg'
print(fn)
photo = extract_features(fn)
img = Image.open(fn)
img.show()
# generate description
description = generate_desc(model, tokenizer, photo, 33)
print(description)
```

*C:\Users\ghrit\Documents\major_project\code\generator.py - Notepad++
File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window ?
get_cnn_features.py
prep_text2.py
generator.py
Python file
length : 2,491 lines : 90
Ln : 36 Col : 1 Sel : 0 | 0
Windows (CR LF) UTF-8 INS
Type here to search
ENG US 20:03 19-04-2019

Figure 6.4: Caption Generator

CHAPTER 7

RESULTS AND ANALYSIS

We test our model on language metrics as the output is text. Evaluation is done by comparing the test set groundtruth captions with the respective predicted captions. Note that there are 5 captions for a single image. Some set of scores are calculated for the text, which are corpus level. Thus they give over all summary of performance for language performance. It is important to note that that doesn't necessarily tell us how many images have been properly captioned. We describe our dataset and metrics in the next sections.

7.1 Benchmark Datasets

Since this is a deep learning problem, we will be relying on reasonably sized datasets, which have properly formatted inputs and outputs. Table 7.1 shows the dataset summary. For this project to be feasible with our limited resources, we have used Flickr8k dataset, which has 1 GB size. It contains 8000 images with 6000 training set images and 1000 images each for test and validation sets. Each image contains 5 captions per image, for a total of 30,000 captions. Flickr30k also exists, which contains 30,000 images, which is more feasible than COCO, yet still difficult to handle with limited resources.

Table 7.1: Dataset Summary

Dataset	Split	Reference Captions per Image
Flickr8K	7K : 1K	5
Flickr30K	28K : 2K	5
MSCOCO	165,482 :81,208 : 81,434	5, 40

A newer dataset was developed when MSCOCO image captioning challenge was started which released the biggest such dataset available for many computer vision tasks like classification, detection, etc. This dataset contains 91 common object categories with 82 of them having more than 5,000 labeled instances. In total the dataset has

2,500,000 labeled instances in 328,000 images. This dataset is 12 GBs in size and is not feasible for experimentation with current resources.

The latest dataset is Conceptual Captions by google AI, a new dataset consisting of 3.3 million image/caption pairs that are created by automatically extracting and filtering image caption annotations from billions of web pages. Conceptual Captions represents an order of magnitude increase of captioned images over the human-curated MS-COCO dataset. As measured by human raters, it has an accuracy of 90 per cent. Furthermore, it represents a wider variety of image-caption styles than previous datasets, allowing for better training of image captioning models. To track progress on image captioning, they also announced the Conceptual Captions Challenge for the machine learning community to train and evaluate their own image captioning models on the Conceptual Captions test bed.

7.2 Performance Metrics

The literature on Image Captioning involves the usage of several metrics including BLEU, ROUGE, METEOR and CIDEr. For our problem, we have made use of the MSCOCO Evaluation Engine, that provides open source code to get the value of the following metrics.

7.2.1 BLEU

Bilingual Evaluation Understudy (BLEU) is a metric initially designed for machine translation problem, but can be used for other language generation tasks. The basic methdology involves counting the number of n-grams occuring in both reference and output sentences. n-gram refers to ‘n‘ contiguous words and it can be 1 (unigram), 2 (bigram), 3 or 4. In general, scores reduce from 1 to 4. Here, word order is irrelevant. It is referred to as “modified n-gram precision“ in the paper, and it can go up to 1.0. It is also worthwhile to note, that to not get an unusually high precision value, each reference gram instance is only considered once.

This value is calculated for each reference-candidate sentence pair occurring in a sentence as shown in equation A.1, and summed up over the whole sentence and subsequently all sentences, as depicted by equation A.2. There is also a penalty for shorter sentences, as they tend to automatically get a higher score do to a mathematical bias. Geometric mean is taken over all sentences to get the corpus level score. This metric has several advantages such as its good agreement with human evaluation, simplicity, and language independence.

7.2.2 ROUGE

Recall-Oriented Understudy for Gisting Evaluation (ROUGE) is a set of scores made for text summarization models. In our system, we use the ROUGE-L metric. It is based on Longest Common Subsequence (LCS). LCS is the longest common contiguous set of words that occur in reference and candidate sentences, i.e. order is relevant. It calculates the F-score (Equation A.10) which requires both precision (Equation A.3) and recall (Equation A.4) measures.

7.2.3 METEOR

Metric for Evaluation of Translation with Explicit ORdering (METEOR) is another Machine Translation metric, that aims to measure the alignment between reference and candidate sentence. This is done by minimizing common chunks between the two. The harmonic mean of precision and recall of n-grams is taken.

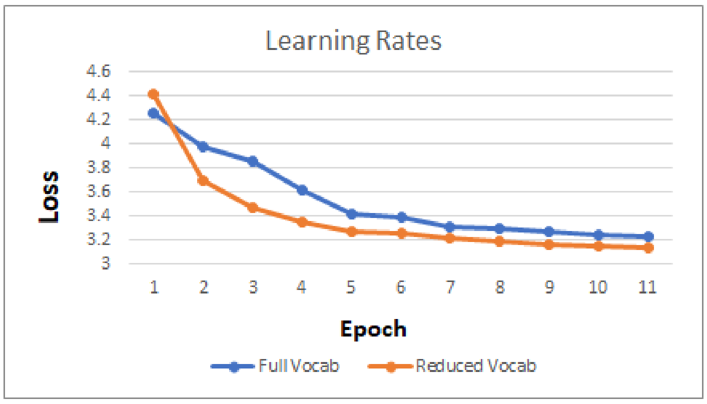
7.2.4 CIDEr

Consensus-based Image Description Evaluation (CIDEr) is a metric which was created for captioning evaluation, and measures a quantity called consensus. Mathematically, the Term Frequency Inverse Document Frequency (TF-IDF) is calculated for each n-gram. while Term Frequency for each n-gram scores it based on frequency of co-occurence in reference and candidate sentences. The Inverse Document Frequence measures the rarity of an n-gram and penalizes it based on overall frequency in corpus.

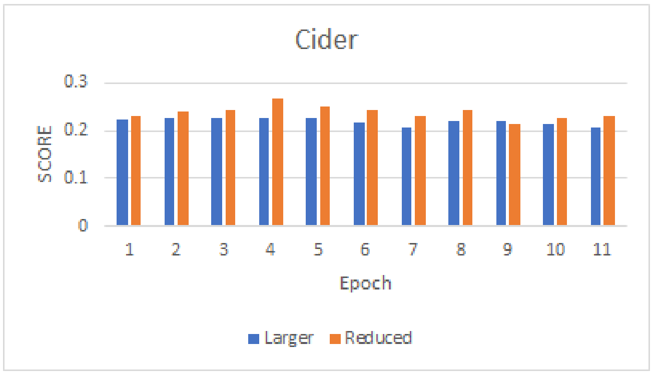
7.3 Results

Our various results have been compiled and tabulated below. the effect of reduction of vocabulary and fine tuning have been underscored and juxtaposed with other models.

7.3.1 Full v/s Reduced vocabulary



(a) Learning Curve



(b) CIDEr

Figure 7.1: Comparison of vocabulary size

The reduced vocabulary helps decrease training time and also shows better results on the error metric as shown above. The reduction by removing infrequent words from the vocabulary. Words that rarely occur in vocabulary are often not learned well by the system and hence result in errors.

7.3.2 Model Comparison

Table 7.2 and 7.3 show the performance for various inject and merge architectures.

Table 7.2: Inject Results

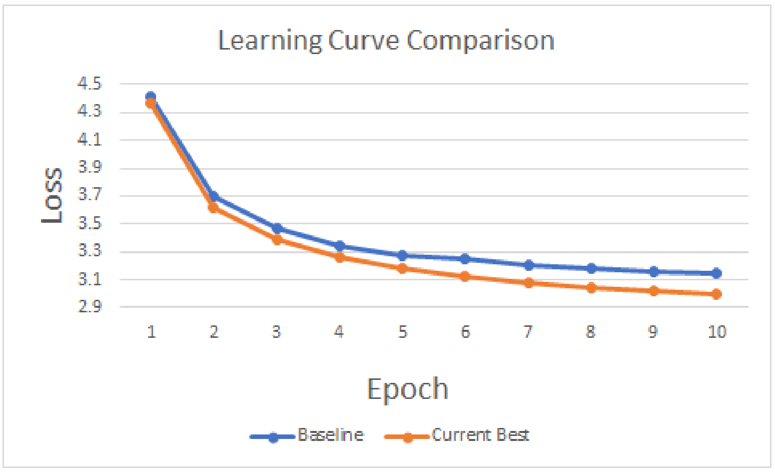
Score	Inject, Full	Inject,Reduced	Deeper Inject	After Fine -Tuning
B1	0.5725	0.582	0.566	0.559
B2	0.273	0.285	0.291	0.319
B3	0.189	0.203	0.180	0.228
B4	0.0.85	0.095	0.843	0.113
METEOR	0.212	0.215	0.202	0.220
ROUGE	0.442	0.450	0.448	0.454
CIDEr	0.232	0.252	0.208	0.263

Table 7.3: Merge Results

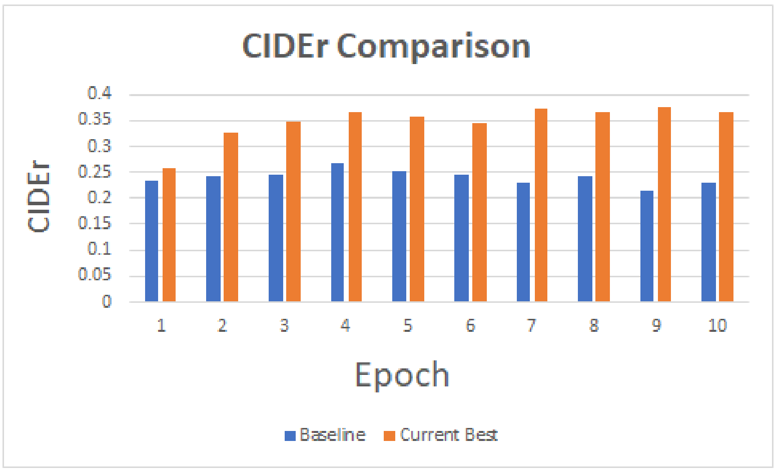
Score	Merge,Full	Merge, Reduced	Deeper Merge	After Fine -Tuning
B1	0.575	0.585	0.569	0.602
B2	0.316	0.313	0.309	0.365
B3	0.212	0.219	0.217	0.265
B4	0.098	0.105	0.101	0.139
METEOR	0.212	0.219	0.220	0.230
ROUGE	0.455	0.459	0.461	0.491
CIDEr	0.229	0.267	0.278	0.377

The two main models that we dealt with were the inject and merge models. Both models work with different arrangement of data processing. Through our various experiments we were able to observe that the merge architecture performs better than the inject architecture. This can be because the language model is handled separately into the LSTM and the final model is made by a dense layer of neurons that make the predictions. The fine tuning of the model gave the best results with respect to the our metrics.

The loss comparison between our baseline model and the fine tuning model are shown below. Our final fine tuning model converges faster and shows a significant improvement in all the error metric.



(a) Learning Curve



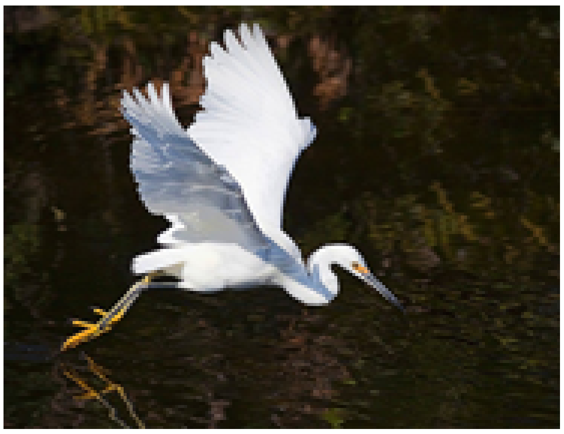
(b) CIDEr

Figure 7.2: Baseline versus Final model Comparison

7.4 Generated Examples

The following captions were generated by our captioning system. The subsequent figures show the where the system does well and also draws inference on the shortcomings of our system.

The model is able to recognize a group of people consistently, as well as actions like jumping, playing and running and even flying (figure 7.3(a)), which may not always be obvious. The final model was able to capture more information than previous models, which would predict a sentence like "group of people are sitting on the street" for image (figure 7.3(b)). This is not a wrong description, but misses major aspects of the image, which a human would immediately notice and mention. Thus, the final model shows visible improvement on previous models.


(a) **Baseline**: The water is in the water. **Best**: White bird is flying over the water.

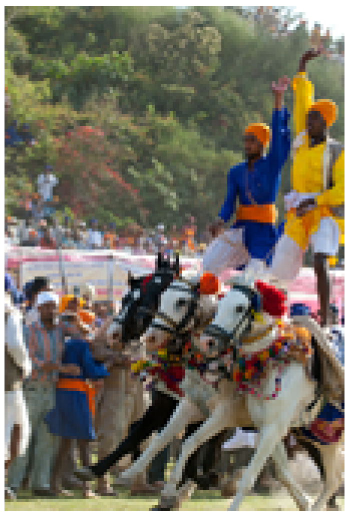

(b) **Baseline**: Group of people are sitting on the street. **Best**: Group of people are riding horses on the road

Figure 7.3: Output Comparison

The shortcomings was in the issue of overfitting. The word "man" is often seen to be followed by the phrase "in red shirt" for many images, without it being so in the image. This overfitting can be attributed to language model. Also, due to dog being the most common noun and images of dogs being very common in the dataset, lots of dog images get a similar description even though they are very different. This can be attributed to the size of the dataset, which can be overcome by training on a bigger dataset such as MSCOCO. Secondly, generalization is noticeably difficult for this model. In pictures that are relatively more complex, and full of visual information, the descriptions are not correct, like in image. This is firstly due to the size of the training dataset, and also due to the fine-tuning performed on it.

CHAPTER 8

CONCLUSION

We have implemented a deep neural network based multimodal architecture for image captioning problem. We have experimented with a number of variations of the multimodal architecture to determine the final system. Many other sophisticated architectures have been proposed for the same, and have achieved great results, such as attention mechanisms and self-critical systems. However, there is still a lack of a system that can give good performance for all images, as well as a metric to quantify in a single value, how many images give a satisfactory description. These can be areas of further research, as they can help direct the iterative process of building a good deep learning model and make it more efficient. Image captioning is still largely an open problem, as well as a very interesting one, that can lead us to sophisticated AI systems that have the capability to process the world similar to us.

CHAPTER 9

FUTURE ENHANCEMENT

Through our experiments we were able to concur that the merge models serves as a better encoder-decoder architecture as compared to the inject architecture. We further learned that fine tuning of pre-trained models and training them on the captioning dataset provides a significant boost in performance. There has been a lot of research on the topic of image captioning and methods such as attention, and self-critical nets. We would like to further augment our present model with such mechanisms to further improve on our results. The use of attention models seem to have shown significant results for the task of image captioning and we will try adding attention to our architecture. Image Captioning being a very resource intensive and time complex task, we would like to be able to work with more robust hardware or online deep learning platforms. There continues to be an interest shown in this topic and new more abundant data sets are being put out to perfect the system. We also hope to work with such elaborate datasets. Furthermore, the captioning task remains to be very general and we would like to look into ways to build a system that can be used in specific environments or situations such as assistance for the visually impaired.

APPENDIX A

METRIC EQUATIONS

A.1 BLEU

$$CP_n(C,S) = \frac{\Sigma_i \Sigma_k min(h_k(c_i), maxh_k(s_{ij}))}{\Sigma_i \Sigma_k h_k(c_i)} \tag{A.1}$$

$$BLEU_N(C,S) = b(C,S) \exp \sum_{n=1}^{N} w_n logCP_n(C,S) \tag{A.2}$$

A.2 ROUGE

$$P_l = \max_j \frac{l(c_i, s_{ij})}{\mid c_i \mid} \tag{A.3}$$

$$R_l = \max_j \frac{l(c_i, s_{ij})}{\mid s_{ij} \mid} \tag{A.4}$$

$$ROUGE_L(c_i, S_i) = \frac{(1+\beta^2)R_l P_l}{R_l + \beta^2 P_l} \tag{A.5}$$

A.3 METEOR

$$P_{en} = \gamma \left(\frac{ch}{m}\right)^{\theta} \tag{A.6}$$

$$F_{mean} = \frac{P_m R_m}{\alpha P_m + (1-\alpha) R_m} \tag{A.7}$$

$$METEOR = (1 - P_{en}) F_{mean} \tag{A.8}$$

A.4 CIDEr

$$g_k(s_{ij}) = \frac{h_k(s_{ij})}{\Sigma_{w_l \epsilon \Omega} h_l(s_{ij})} log\left(\frac{\mid I \mid}{\Sigma_{I_p \epsilon I} min(1, \Sigma_q h_k(s_{pq}))}\right) \tag{A.9}$$

$$CIDEr_n(c_i, S_i) = \frac{1}{m} \sum_j \frac{g^n(c_i).g^n(s_{ij})}{\mid\mid g^n(c_i) \mid\mid \mid\mid g^n(s_{ij}) \mid\mid} \tag{A.10}$$

REFERENCES

1. Sharma, Piyush, et al. "Conceptual captions: A cleaned, hypernymed, image alt-text dataset for automatic image captioning." Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers). Vol. 1. 2018.

2. Y. Yang, C. L. Teo, H. Daume, Y. Aloimono, Corpus-guided sentence generation of natural images, in: Proceedingsof the Conference on Empirical Methods in Natural Language Processing, 2011

3. G. Kulkarni, V. Premraj, V. Ordonez, S. Dhar, S. Li, Y.Choi, A. C. Berg, T. L. Berg, Babytalk: Understanding and generating simple image descriptions, IEEE Transactions on Pattern Analysis and Machine Intelligence 35 (2013)

4. A. Farhadi, M. Hejrati, M. A. Sadeghi, P. Young, C.Rashtchian, J. Hockenmaier, D. Forsyth, Every picture tells a story: Generating sentences from images.

5. A. Gupta, Y. Verma, C. V. Jawahar., Choosing linguistics over vision to describe images.

6. O. Vinyals, A. Toshev, S. Bengio, D. Erhan, Show and tell: A neural image caption generator, in: IEEE Conference on Computer Vision and Pattern Recognition, 2015.

7. J. Donahue, L. Hendricks, S. Guadarrama, M. Rohrbach, S. Venugopalan, Long-term recurrent convolutional networks for visual recognition and description.

8. Q. Wu, C. Shen, L. Liu, A. Dick, A. van den Hengel, What value do explicit high level concepts have in vision to language problems?

9. A. Karpathy, F. Li, Deep visual-semantic alignments for generating image descriptions, in: IEEE Conference on Computer Vision and Pattern Recognition, 2015.

10. J. Mao, W. Xu, Y. Yang, J. Wang, A. L. Yuille, Explain images with multimodal recurrent neural networks.

11. Ting Yao, Yingwei Pan, Yehao Li, Zhaofan Qiu, Tao Mei, Boosting Image Captioning with Attributes.

12. K. Xu, J. Ba, R. Kiros, K. Cho, A. Courville, R. Salakhutdinov, R. Zemel, Y. Bengio, Show, attend and tell: Neural image caption generation with visual attention.

13. Q. You, H. Jin, Z. Wang, C. Fang, J. Luo, Image captioning with semantic attention, in: IEEE Conference on Computer Vision and Pattern Recognition, 2016.

14. J. Lu, C. Xiong, D. Parikh and R. Socher, "Knowing When to Look: Adaptive Attention via a Visual Sentinel for Image Captioning," 2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), Honolulu, HI, 2017.

15. Anderson, Peter He, Xiaodong Buehler, Chris Teney, Damien Johnson, Mark Gould, Stephen Zhang, Lei. Bottom-Up and Top-Down Attention for Image Captioning and VQA. (2017).

16. S. J. Rennie, E. Marcheret, Y. Mroueh, J. Ross and V. Goel, "Self-Critical Sequence Training for Image Captioning," 2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR).

17. Bo Dai, Dahua Lin, Raquel Urtasun, and Sanja Fidler. 2017. Towards Diverse and Natural Image Descriptions via a Conditional GAN. In Proceedings of the IEEE conference on computer vision and pattern recognition (CVPR).

18. Rakshith Shetty, Marcus Rohrbach, Lisa Anne Hendricks, Mario Fritz, and Bernt Schiele. 2017. Speaking the Same Language: Matching Machine to Human Captions by Adversarial Training. In IEEE International Conference on Computer Vision (ICCV).

19. Lin, Tsung-Yi et al. âĂIJMicrosoft COCO: Common Objects in Context.âĂİ ECCV (2014).

20. Papineni, Kishore, et al. "BLEU: a method for automatic evaluation of machine translation." Proceedings of the 40th annual meeting on association for computational linguistics. Association for Computational Linguistics, 2002.

21. M. Denkowski and A. Lavie, âĂIJMeteor universal: Language speciïňĄc translation evaluation for any target language,âĂİ in EACL Workshop on Statistical Machine Translation, 2014.

22. C.-Y. Lin, âĂIJRouge: A package for automatic evaluation of summaries,âĂİ in ACL Workshop, 2004.

23. R. Vedantam, C. L. Zitnick, and D. Parikh, âĂIJCider: Consensus-based image description evaluation,âĂİ arXiv preprint arXiv:1411.5726, 2014.

24. Simonyan, Karen, and Andrew Zisserman. "Very deep convolutional networks for large-scale image recognition." arXiv preprint arXiv:1409.1556 (2014).

25. Szegedy, Christian, et al. "Rethinking the inception architecture for computer vision." Proceedings of the IEEE conference on computer vision and pattern recognition. 2016.