

St. Xavier's College(Autonomous), Kolkata

30, Mother Teresa Sarani, Kolkata 700016

Department of Computer Science

www.sxccal.edu | csc@sxccal.edu | 2255 1271



Mimic Human-Level Understanding of Image

By

Aniket Singh - 519, Riddhick Dalal - 504, Rhitaza Jana- 554

Under the guidance Of

Dr. Sonali Sen

*(Assoc. Professor, Postgraduate & Research Department of Computer
Science)*

St. Xavier's College(Autonomous), Kolkata

April 2024

Submitted to the Department of Computer Science in partial fulfillment of the
requirements for the degree of M.Sc. Computer Science

CERTIFICATE OF AUTHENTICATED WORK

This is to certify that the project report entitled, ‘**Mimic Human Level Understanding of Image**’ submitted to the Postgraduate and Research Department of Computer Science, St. Xavier’s College (Autonomous), Kolkata, in partial fulfillment of the requirement for the award of the degree of Master of Science (M.Sc.) is an original work carried out by:

Name	Roll No.	Registration No.
Aniket Singh	2-72-22-0519	012-1111-1259-19
Riddhick Dalal	2-72-22-0504	A03-1142-0045-19
Rhitaza Jana	2-72-22-0554	A01-2112-0067-22

under my guidance. The matter embodied in this project is authentic and is genuine work done by the student and has not been submitted whether to this college or to any other institute for the fulfillment of the requirement of any course of study.

.....

Signature of the Supervisor

.....

Name of Supervisor

Date:.....

.....

Signature of the Head of Dept.

Date:

Name and Signature of the Project Team Members:

	Name	Signature
1.
2.
3.

ABSTRACT

Image Descriptions are fundamental ways of conveying the representational content of images. An image description is the amalgamation of language and visual fields, allowing for an easier understanding of visual material. It encourages inclusive communication and accessibility and improves the user experience in general. Recent studies have made it possible to use Image Processing and Natural language Processing to generate an image description automatically. Nevertheless, these models have a significant challenge that arises when describing an image using a single caption, potentially losing many crucial details. In this paper, we propose methodologies that overcome this challenge by creating a coherent image description that mimics the human-level understanding of the image.

In our method, we employ Graph-based Convolutional Neural Networks to identify the semantic

relations and hierarchical LSTM - BiLSTM models with Attention mechanisms to generate a coherent description of the image. Pre-trained models like VGG19 and Faster R-CNN are used to extract features from images.

Keywords : *CNN, GCN, LSTM, BiLSTM, Attention Mechanism, Faster R-CNN, VGG19*

ACKNOWLEDGEMENT

We would like to express our deep and sincere gratitude to our project guide Dr. Sonali Sen, for her invaluable guidance throughout the project. Working and studying under her guidance was a great privilege and honor. The completion of this project motivated us to have a deeper understanding of the subject. In addition, we thank our classmates and faculty for their vital suggestions which helped us bring numerous improvements to the project. We express our heartfelt gratitude to all the people supporting us throughout the project either directly or indirectly which immensely contributed to the evolution of the ideas involved in the project.

TABLE OF CONTENTS

Chapter	Content	Page No.
Chapter 1 : Introduction	1.1. Background	viii-ix
	1.2. Objectives	ix
	1.3. Purpose, Scope, Applicability	x
	1.4. Achievements	xi
Chapter 2 : Survey Of Technologies	2.1. Convolutional Neural Networks	xiii-xiv
	2.2. LSTM	xiv-xvi
	2.3. Bi-directional LSTM	xvi-xvii
	2.4. VGG16	xvii-xviii
	2.5. GCN	xviii-xix
	2.6. Attention Mechanism	xx
	2.7. Faster R-CNN	xx-xxi
Chapter 3: Requirements and Analysis	3.1. Problem Definition	xxiii
	3.2. Software Requirements	xxiii-xxiv
Chapter 4: System Design	4.1. Overall Workflow	2
	4.2. Data Collection	2
	4.3. Data Preprocessing and Cleaning	3
	4.4. Object Relationship Graph Creation	4-5

	4.5. Model Implementation	6-12
Chapter 5: Implementation and Testing	5.1. Bi-directional LSTM with Attention (Benchmark Model)	14-15
	5.2. GCN with Dual Attention Model	16-34
Chapter 6: Results and Discussion	6.1. Output	36-39
	6.2. Working of Multi Headed Attention	40-42
Chapter 7: Conclusion		44
References		xxv-xxvi

CHAPTER 1: INTRODUCTION

1.1. BACKGROUND

Images serve as a powerful medium of communication, encapsulating a wealth of intricate and nuanced information. Understanding and articulating these images in a way that mirrors human cognition is a significant challenge, and it is this challenge that our project seeks to address.

Human cognition involves recognizing various elements within an image, such as objects and scenes, and understanding their contextual relationships, emotions, and abstract concepts. This level of visual comprehension allows humans to describe images in a logical and detailed manner. Despite considerable progress in computer vision systems, replicating this remarkable human ability remains a formidable benchmark.

1.1.1. Related Works

The authors Reshmi Sasibhooshan, Suresh Kumaraswamy, and Santhoshkumar Sasidharan [1] present a Wavelet transform-based Convolutional Neural Network (WCNN) using two-level discrete wavelet decomposition as an encoder-decoder framework. This approach generates visual feature maps from images that highlight spatial, spectral, and semantic details.

The GLA model [2] addresses two major issues in image description: object missing, which occurs when essential objects are omitted when generating the picture description, and misprediction, which occurs when an object is erroneously classified.

The Recurrent Fusion Network (RFNet) proposed by Wenhao Jiang et. al. [3] employs multiple Convolutional Neural Networks (CNNs) as encoders, followed by a recurrent fusion technique, to improve image understanding beyond single-view models like ResNet or Inception-X.

The authors [4] provide a new design within the context of an attention-based encoder-decoder system. They propose a Graph Convolutional Networks Plus Long Short Term Memory (GCN-LSTM) architecture for picture encoding that incorporates both semantic and spatial object relationships.

Kun Fu et al. [5] propose an image captioning system that exploits the parallel structures between images and sentences. In addition, their technique provides scene-specific contexts, which capture higher-level semantic information inherent in an image.

Huo, L [6] provides a novel two-phase system for producing image captions. The first phase is a hybrid deep learning phase in which a novel factored three-way interaction machine is presented to hierarchically learn the relational properties of human-object pairs. The second phase is the production of visual descriptions

Xiao et al. [7] propose a Dense Semantic Embedding Network (DSEN) for image captioning, which densely embeds attributes with multi-modal information at each word generation step. To enhance attribute discrimination, a Threshold ReLU (TReLU) is introduced in addition to a bidirectional LSTM structure for capturing both previous and future contexts.

Krause, Johnson, Krishna, and Li Fei-Fei [8] introduce a hierarchical approach for generating descriptive image paragraphs. Their model crafts detailed, coherent narratives by decomposing images and paragraphs and employing a hierarchical recurrent neural network.

Al-Malla et al. [9] present an attention-based, Encoder-Decoder deep architecture that makes use of convolutional features extracted from a CNN model pre-trained on ImageNet (Xception), together with object features extracted from the YOLOv4 model.

Johnson et al. [10] introduce dense captioning, a task combining image localization and description. Their Fully Convolutional Localization Network (FCLN) architecture processes images efficiently without external proposals, achieving improvements in speed and accuracy over existing methods on the Visual Genome dataset. This approach enhances the understanding and description of images, advancing applications in computer vision.

1.2. OBJECTIVES

The objective of this project is to develop a machine learning-based system capable of generating descriptive captions for a given image. Leveraging state-of-the-art deep learning techniques such as convolutional neural networks (CNNs) and recurrent neural networks (RNNs), the system will analyze the visual content of images and generate natural language descriptions that accurately depict the scene.

1.3. PURPOSE, SCOPE, APPLICABILITY

1.3.1. Purpose

The purpose of this project is to develop an advanced machine-learning system capable of generating descriptive captions for images, thereby facilitating enhanced understanding and accessibility of visual content through natural language processing. By harnessing the power of state-of-the-art deep learning techniques, the project aims to bridge the semantic gap between visual information and textual descriptions, empowering machines to effectively communicate visual content in a human-like manner.

1.3.2. Scope

The scope of this project entails developing a machine learning-based system for generating descriptive captions for images, encompassing CNN-LSTM-based model development, data collection and preprocessing, algorithm implementation, comparative studies with existing technologies, and performance optimization. The focus will be on designing a deep learning model integrating computer vision and natural language processing techniques to accurately analyze images and generate coherent textual descriptions. The system will be evaluated using both automated metrics and human evaluation studies, and a user-friendly interface will be developed to facilitate interaction. Performance optimization techniques will be employed to ensure efficient and scalable deployment. The project will exclude generating descriptions for non-static multimedia content and will consider constraints such as computational resources and time limitations while aiming to deliver a functional and efficient image description generation system.

1.3.3. Applicability

The successful completion of this project will have significant implications across various domains, including assistive technologies for visually impaired individuals, content generation for multimedia applications, and enhanced accessibility in human-computer interaction. By enabling machines to accurately describe visual content, this project aims to bridge the gap between the visual and textual modalities, unlocking new opportunities for AI-driven applications and services.

1.4. ACHIEVEMENTS

The models are capable of producing accurate image descriptions. Especially if the objects present in the image are clearly distinguishable and can be classified correctly by the pre-trained networks, then the output descriptions are quite close to the actual descriptions. Even with the unknown images, the devised algorithms work very well. In general, we have achieved an average BLEU 1 score of 0.54 which is considered a good measure for this type of Natural Language Generation.

CHAPTER 2: SURVEY OF TECHNOLOGIES

2.1. CONVOLUTIONAL NEURAL NETWORK

Convolutional Neural Networks are one of the most used machine learning approaches in tasks like image recognition, classification, and other fields of computer vision. CNNs are based on the idea of replicating how the visual cortex functions in the brain.

CNNs are composed of layers, each layer processes and extracts information from the input data. There are three primary layers present in a CNN, these are - a) Convolution Layers, b) Pooling Layers, and c) Fully Connected Layers.

2.1.1. Convolution Layers

These layers use a collection of learnable filters, or kernels, to execute convolution operations to the input data. These filters extract different features, including edges, textures, or patterns, by slicing over the input data and applying element-wise multiplications and summations. Each convolutional layer often uses several filters, enabling the network to simultaneously learn various properties at various spatial locations.

2.1.2. Pooling Layers

Pooling layers take the outputs of the Convolution Layers and downsample the feature maps. This reduces the dimensionality while preserving the important features for further processing. The pooling layer makes the learned features more robust to small variations and also reduces computational complexity. Two main pooling operations are max pooling and average pooling, where max pooling retains the maximum value in a certain region, and average pooling retains the average value in a given region.

2.1.3. Fully Connected Layers

These are the last layers of the CNN, they take the high-level features coming from the previous layers and map them to the output classes. Each neuron in these layers is connected with all the neurons of the preceding layer. These are

also called the Dense Layers. Thus the network learns about the complex patterns and relationships about the data.

2.1.4. Training Of CNNs

CNN uses Supervised Machine Learning algorithms. CNNs are fed with labeled input-output pairs and the network parameters (weights and biases) are adjusted iteratively using backpropagation. CNNs mainly use optimizing algorithms like stochastic gradient descent (SGD) or its variants to minimize the loss functions.

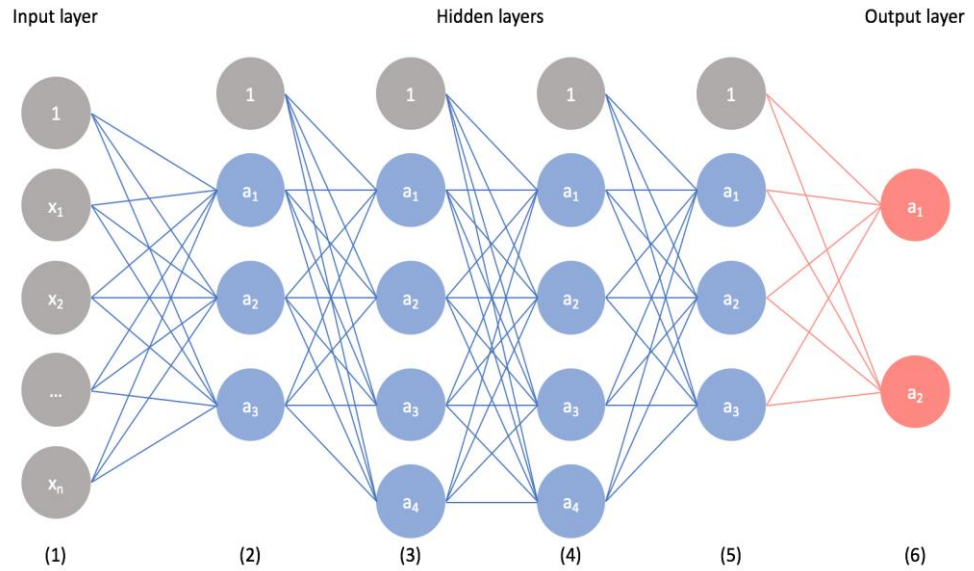


Figure 1: Convolutional Neural Network

2.2. LONG SHORT-TERM MEMORY NETWORKS

Long Short-Term Memory (LSTM)[11] is a type of recurrent neural network (RNN) that can handle long time-series data. It is an advancement over the Recurrent Neural Networks to overcome vanishing gradient problems and capture long-range dependencies in sequential data. LSTM has become an extremely useful architecture of neural networks in the areas of Natural Language Processing, Speech Recognition, Time Series Prediction, and more.

The LSTM architecture mainly consists of Memory Cells and Gating Mechanisms which control the flow of information through the network.

2.2.1. Memory Cell

Memory Cells are the core component of the LSTM. Memory cells store information from previous time steps and use it to influence the output of the cell at the current time step. The output of each LSTM cell is passed to the next cell in the network, allowing the LSTM to process and analyze sequential data over multiple time steps.

2.2.2. Forget Gate

At every time step, the LSTM network decides what information to retain or discard from the time step before using the forget gate. The forget vector takes the input from the hidden state of the previous time step and the input from the current time step and outputs a forgetting vector that tells the LSTM what information to keep in the memory cell.

2.2.3. Input Gate

The input gate controls the memory cell update. It determines the amount of new information to be added to the memory cell at the time step. Just like the forget gate, the input gate takes the previous hidden state of the memory cell, takes the current input, and creates an input gate vector.

2.2.4. Cell State Update

This step includes overhauling the memory cell state utilizing the forget gate, input gate, and an intermediate update candidate. The forget gate decides what to expel from the cell state, the input gate decides what to include in the cell state, and the update candidate represents the new candidate values to add to the cell state.

2.2.5. Output Gate

The output gate controls the data stream from the memory cell to the current hidden state/output. It chooses which parts of the memory cell ought to be exposed to the rest of the network at the current time step.

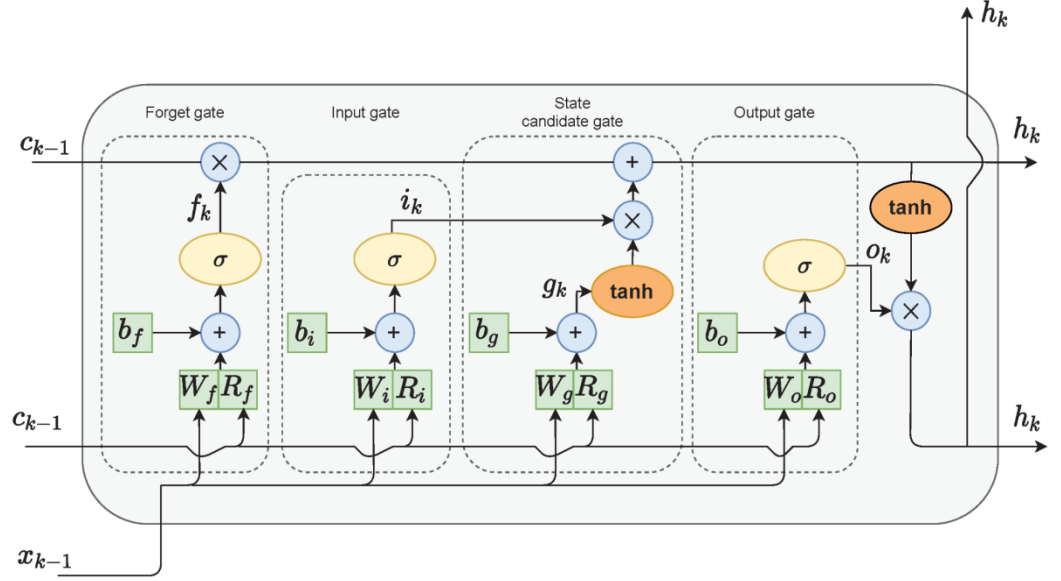


Figure 2: Structure of LSTM Network

2.3. BI-DIRECTIONAL LSTM

A Bidirectional LSTM (Long Short-Term Memory) is a modified version of the regular LSTM, which is widely utilized in tasks involving natural language processing (NLP) and sequential data.

In a typical LSTM model, data moves linearly within the network, causing it to be influenced by the sequence of the input. However, in numerous circumstances, data from previous and upcoming time points can be advantageous for creating precise forecasts. This is when bidirectional LSTMs are utilized.

A bidirectional LSTM consists of two LSTM layers, where one processes the input sequence forward and the other processes it backward. Normally, the results from the two LSTM layers are joined together or merged in some manner before moving on to the next layers or being utilized for forecasting.

The forward LSTM layer analyzes the input sequence chronologically and gathers data from previous time steps, whereas the backward LSTM layer

examines the sequence in reverse order, gathering information from future time steps. By merging data from two sources, the model can enhance its understanding of the context at each time point and improve its predictive abilities.

Bidirectional LSTMs are especially beneficial in tasks that require considering information from both previous and upcoming time points, like sequence labeling (for example, named entity recognition, part-of-speech tagging), sentiment analysis, machine translation, and speech recognition.

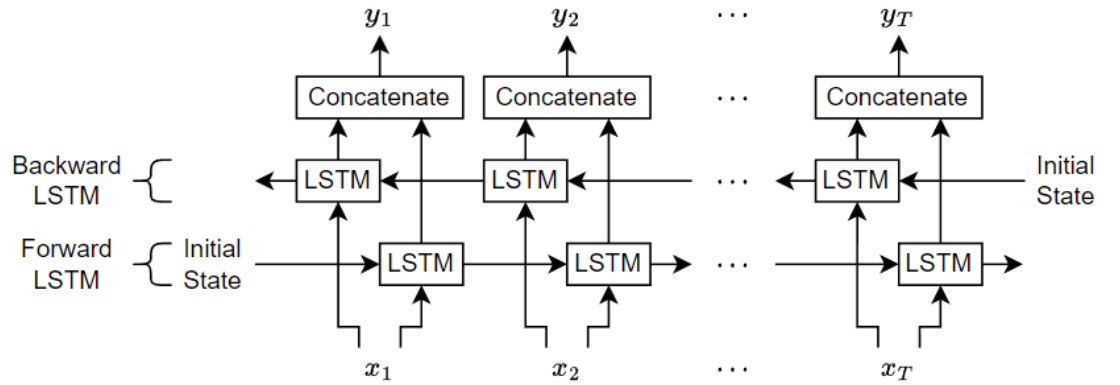


Figure 3: Bi-directional LSTM

2.4. VGG16

VGG16 [13] is a 16-layer deep neural network. VGG16 is therefore a relatively large network, with a total of 138 million parameters, which is enormous even by today's standards. However, the simplicity of the VGGNet16 architecture is its main attraction. The VGGNet architecture contains the most important features of convolutional neural networks. A VGG network consists of small convolution filters. VGG16 has three fully connected layers and 13 convolutional layers.

VGGNet receives input images of size 224x224. In the ImageNet competition, modelers kept the size of the input images constant by cutting a 224x224 section from the center of each image. The VGG convolutional filter uses the smallest possible 3x3 receptive field. VGG also uses a 1x1 convolutional filter as the input linear transformation. ReLU is a linear function that provides consistent output for positive input and zero output for negative input. VGG has a convolution stride set to 1 pixel to maintain spatial resolution after convolution. The stride value reflects the number of pixels the filter "moves"

to cover the entire image space. All hidden layers in the VGG network use ReLU instead of local response normalization. It increases training time and memory consumption with little improvement in overall accuracy. The pooling layer follows several convolutional layers. This helps reduce the dimensionality and number of feature map parameters generated at each convolution step. Pooling is very important considering that the number of available filters in the final layer increases rapidly from 64 to 128 to 256 and finally to 512. VGGNet contains three fully connected layers. The first two levels have 4096 channels each, and the third level has 1000 channels, one for each class.

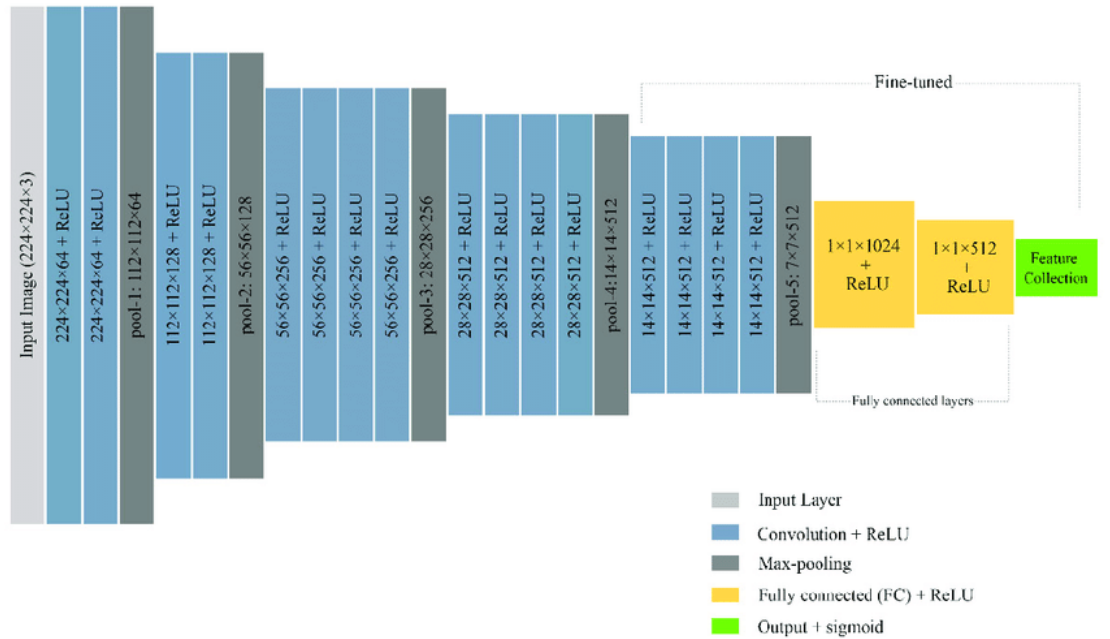


Figure 4: Architecture of VGG model

2.5. GRAPH CONVOLUTION NETWORK

Graph Convolutional Networks [12] are special modifications over traditional Convolutional Networks, specifically tailored for analysis of graph-based data. GCNs are useful in extracting and exploiting relational information inherent in graph structures. This specialized capability makes GCNs suitable for tasks like node classification, link prediction, and graph classification within diverse domains.

GCNs extend the paradigm of convolution to irregular graph topologies. GCNs generalize the principles of convolutional networks to accommodate the inherent irregularity and connectivity nuances intrinsic to graph structures.

The graph convolutional layers propagate the node features across the entire graph. This process includes aggregation of information from the neighboring nodes and performing transformation operations to refine the node representations in the network. Through the integration of multiple graph convolution layers, the network iteratively distills the abstract representations of the input graph.

Due to the versatile nature of graphs, GCNs can be used in domains like social network analysis, bioinformatics, recommendation systems, drug discovery, etc. Though GCNs are useful in many fields, challenges including scalability of large graphs, and generalization to unseen graph structures still exist.

For our project, we will be using GCN to represent the various parts of an image and the relationship between them, where the nodes of the graph represent a smaller region from the image, and the edge between two nodes defines the relationship between those regions.

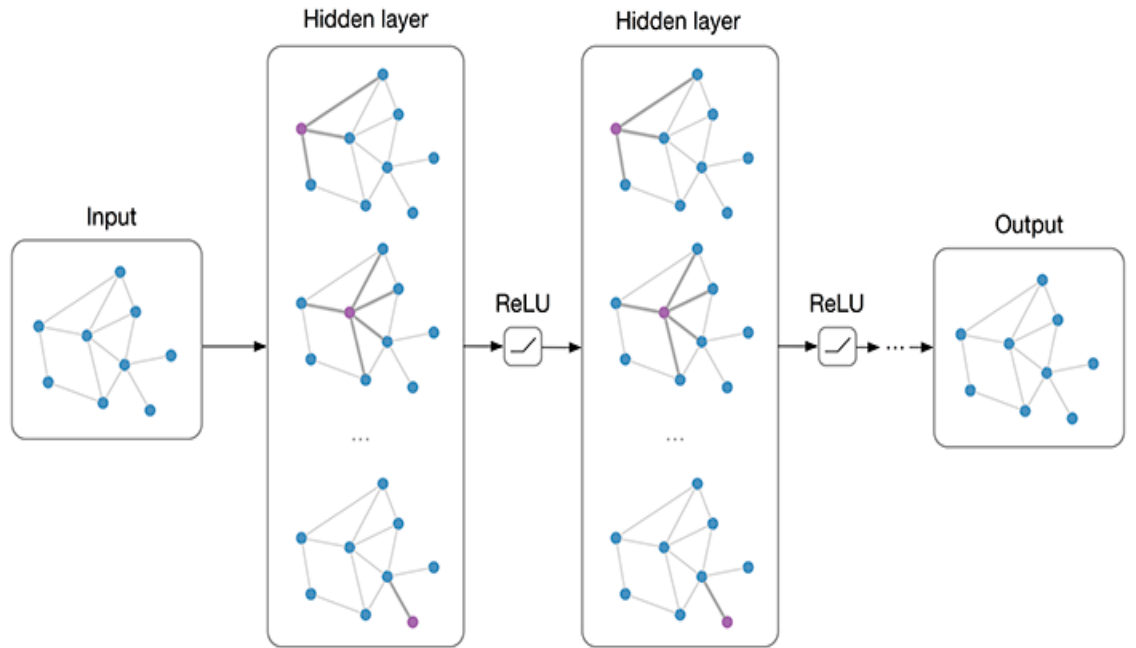


Figure 5: General Structure of a GCN

2.6. ATTENTION MECHANISM

Attention mechanisms [14] are a great innovation within the realm of neural network architectures. This provides neural networks the ability to dynamically focus on specific parts of input data. The neural networks can selectively emphasize or de-emphasize different elements based on the relevance of the task. This ability helps in processing sequential data like natural sentences or time series data.

The core idea of attention mechanisms is to make the neural networks capable of denoting the importance or relevance of different input parts based on the attention weights. These weights are computed dynamically based on the input and current state of the network.

The common variance of attention mechanism includes the self-attention mechanism, which is prevalent in NLP. In self-attention, the network compares each element of the input sequence with every other element, and based on that it computes the attention weights. Thus it captures the dependencies and relation between different parts of the sequence. The widespread adoption of attention mechanisms across diverse domains indicates their versatility and effectiveness in enhancing the capabilities of neural networks.

2.7. FASTER R-CNN

The Faster R-CNN (Region-based Convolutional Network) [15] is a specialized model for object detection and computer vision. It uses a combination of convolutional neural networks and region proposal mechanisms. This model has two key components- a. Region Proposal Network, b. Region Based CNN.

The RPN provides an efficient and accurate mechanism for generating region proposals within an image. This is done using a sliding window over the feature map extracted from the input image. For each sliding window position, the RPN simultaneously predicts bounding box proposals. These predictions are refined through a non-maximum suppression (NMS) algorithm to yield a compact set of high-quality region proposals.

The RCNN module is a series of convolutional layers followed by region-specific pooling and fully connected layers. These layers are used to extract features from the proposed regions and predict the class labels. It also refines the bounding box coordinates for each detected object.

Faster R-CNN offers a combination of speed, accuracy, and efficiency for extracting region-based features from the image.

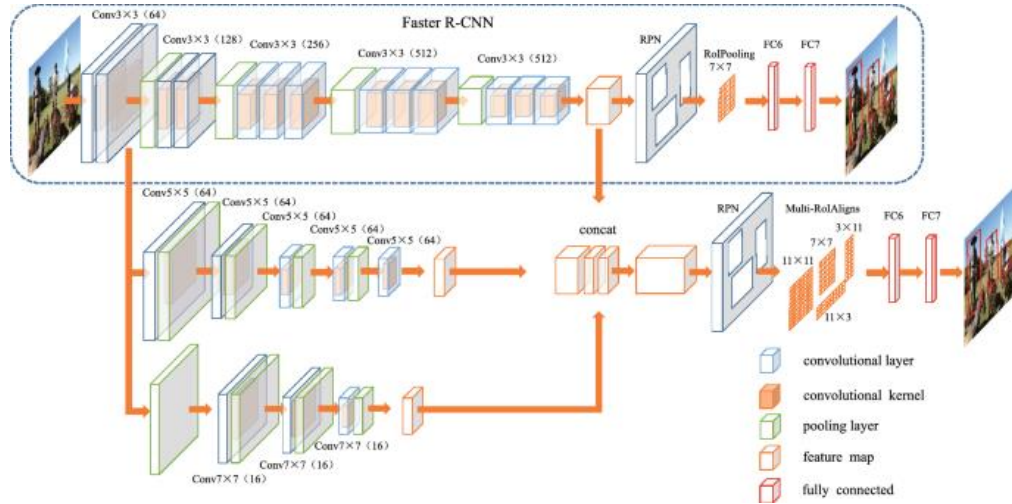


Figure 6: Architecture of Faster R-CNN

CHAPTER 3: REQUIREMENT AND ANALYSIS

3.1. PROBLEM DEFINITION

The process of generating a description from a given image requires identifying the different objects present in the image, learning the different features of the image, and forming a description that correctly correlates with the input image. This entire process can be divided into two main subproblems - 1) Feature extraction from image, and 2) Generate description from the extracted features.

3.1.1. Feature Extraction From Image

Features of an image consist of both semantic and spatial information of the image. The different objects and unique items present in the image and their interrelations needed to be defined correctly to generate accurate description of an image. The different features extracted from an image are combined and encoded for further processing.

3.1.2. Generate Description From Features

Once we successfully retrieve all the information from the given image we use this information to predict the description that accurately represents the image. Feature vectors are passed through an LSTM or BiLSTM to predict and generate readable descriptions [16]. . The LSTM models are provided with attention mechanisms to improve the meaning of generated outputs.

3.2. SOFTWARE REQUIREMENTS

The packages and libraries required to implement this project are listed below-

1. Python 3 (Version 3.8 and above)
2. Tensorflow 2.0 and above
3. Pytorch 2.2
4. TorchVision
5. DGL 2.0

6. Spacy
7. Pandas
8. Matplotlib
9. Fasterrcnn_resnet_50
10. VGG19
11. Kears_self_attention
12. Keras Bidirectional

CHAPTER 4: SYSTEM DESIGN

4.1. OVERALL WORKFLOW

The entire project can be divided into the following substeps:-

- 1) Data Collection
- 2) Data Preprocessing and Cleaning
- 3) Feature Extraction from the Image
- 4) Benchmark model: CNN-Bi-LSTM Attention model
- 5) Model Implementation: GCN with Dual Attention-based Decoder model

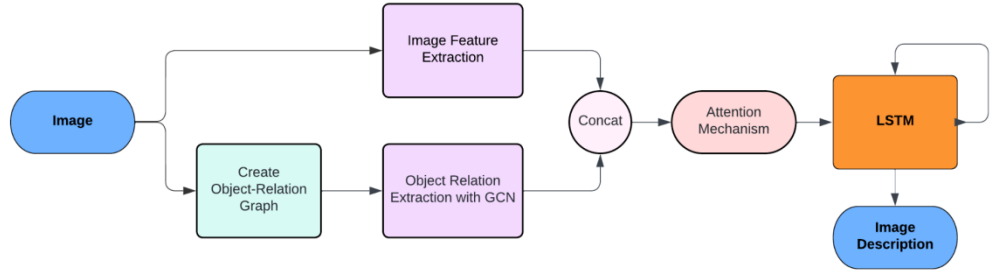


Figure 7: Diagram of the GCN with Dual Attention-based Decoder model

4.2. DATA COLLECTION

According to our addressed problem, the dataset that has been taken under consideration after specific research is the Flickr8k dataset [18]. It is a widely used benchmarked dataset in the field of Computer Vision. It contains a diverse range of images covering various scenes, objects, and activities captured in the real world. Each image in the dataset is also provided with multiple human-generated textual descriptions. The dataset contains 8000 images and 5 textual descriptions per image. We specifically chose this dataset because it will be easier to replace this dataset with larger Flickr30k or MSCOCO datasets for future enhancement of the project.

4.3. DATA PREPROCESSING AND CLEANING

4.3.1. Preprocessing of Textual Data

The Flickr8k dataset comes with 5 textual descriptions for each image. We use two types of preprocessing over the textual descriptions -

- a. For our traditional Bi-LSTM-based model we first convert the descriptions into lowercase then append <startseq> at the beginning of each description and <endseq> at the end of each description. These define the start and end of a description. Then we tokenize the entire dataset to generate a vocabulary, this vocabulary will be used to train the model in a later stage.
- b. The caption text undergoes several preprocessing steps before it is used in the dataset [17]. We use a library function spaCy to tokenize the text into words (or tokens). This method also converts the text into lowercase and splits it into individual tokens.

Then, a vocabulary is built from a list of captions, where words that meet a specified frequency threshold are added to the vocabulary with unique integer indices, creating a mapping between words to indices and vice versa. This allows text to be numericalized by converting words into their corresponding indices based on the vocabulary, with special handling for out-of-vocabulary words (mapped to an "unknown" token). The preprocessed captions are further structured by adding "start of sequence" and "end of sequence" tokens to indicate the beginning and end of each caption. In the context of the dataset, when retrieving an item, the text is converted to its numerical representation and returned as a PyTorch tensor.

4.3.2. Preprocessing of Image Data :

We are using VGG19 for feature extraction from the images. Both of these networks are pre-trained networks, thus the input dimensions of these networks are already predefined. We have to resize our images into a shape of 224x224 for them to be accepted by these networks.

4.4. OBJECT-RELATIONSHIP GRAPH CREATION

We are using the VGG19 pre-trained network to extract features from the image. For our project we are modifying the VGG19 by removing its last two layers, thus we can have access to the raw features coming from the flattened layer of the network.

Once we extract features of the entire image, we use the Fast R-CNN to detect objects present in the image. It creates a list of objects for each image, using this list we create the initial graph for every image. We again use the VGG19 to extract features for all the detected regions we get as outputs from the Fast R-CNN. The nodes of the graph define the extracted object features and the edge defines the relationship between two such nodes. The decision of two nodes having any dependencies is made using the ROI method. If the regions of two nodes intersect each other over a certain threshold value we conclude that there is a relationship between those two nodes.

Algorithm 1 Object-Relationship Graph Creation

Require: An image `image`, pre-trained object detection and feature extraction models, Intersection over Union (IoU) threshold `iou_threshold`

Ensure: A DGL graph `G` representing detected objects and their relationships

```
1: Initialization:
2: Set up object detection model object_detection_model with Faster R-CNN
   (fasterrcnn_resnet50_fpn) in evaluation mode
3: Set up feature extraction model feature_extractor with VGG19 (layers
   0-35), freezing its parameters
4: Initialize an empty DGL graph G
5: Object Detection:
6: Run object_detection_model on image to obtain detections, including
   bounding boxes (boxes)
7: if no objects are detected then
8:   Extract features from the entire image using feature_extractor
9:   Add a node to G with these features
10:  Add a self-loop edge to this node
11:  return G
12: end if
13: Node Creation:
14: for each bounding box box in boxes do
15:   Extract coordinates x1, y1, x2, y2 from box
16:   if x1 >= x2 or y1 >= y2 then
17:     Skip this bounding box (invalid)
18:   end if
19:   Crop image to the region defined by box
20:   if object region has zero size then
21:     Skip this box
22:   end if
23:   Resize object region to (224, 224)
24:   Extract features from the resized region using feature_extractor
25:   Add a node to G with these features
26: end for
27: Edge Creation:
28: Let num_objects represent the number of nodes in G
29: for each pair of nodes (i, j) where i does not equal j do
30:   Calculate IoU between their bounding boxes
31:   if IoU exceeds iou_threshold then
32:     Add an edge from i to j
33:   end if
34: end for
35: return G
```

4.5. MODEL IMPLEMENTATION

4.5.1. Bi-LSTM Based Benchmark Model

In this model, we used the traditional approach to generate descriptions from images. We used the features extracted from the VGG19 and the tokenized descriptions as the input of the Bidirectional Network. We attached a self-attention layer in the network to improve contextual meaning detection. It has three Dense layers, each with 256 units, and a single Bidirectional layer with 512 units of LSTM.

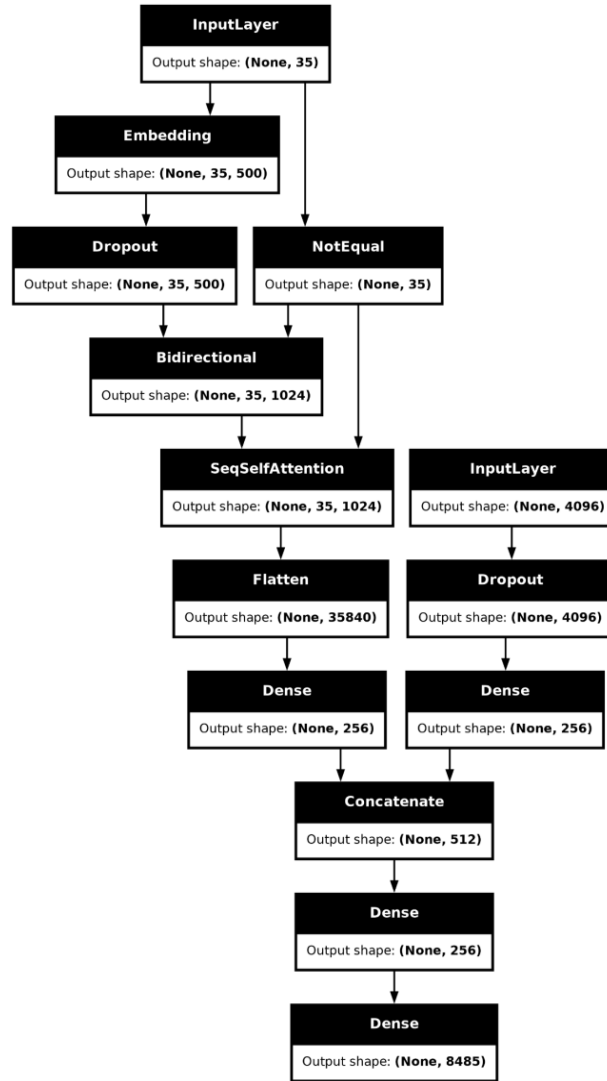


Figure 8: Architecture of the Bi-LSTM with an attention mechanism

4.5.2. GCN and Dual Attention Based Model

Image captioning requires understanding both the content and the context of an image, along with a mechanism to generate a coherent sequence of words. We combine the following components into our implementation:

1. **Graph Convolutional Networks (GCNs):** Used to extract relational information from graph structures.
2. **Multi-Head Attention Mechanisms:** Employed for self-attention and cross-attention to enhance sequence generation and improve contextual understanding.
3. **Long Short-Term Memory (LSTM) Networks:** Used for sequence modeling and caption generation.

Graph Convolutional Network (GCN)

The GCN component is used to extract additional relational features from a given graph structure. The GCN contains two Graph Convolutional layers, followed by a linear layer for edge feature computation. Given a graph G with node features H , the GCN layers perform convolution operations as follows:

$$H_1 = ReLU(G_1(H)),$$

$$H_2 = ReLU(G_2(H_1)),$$

where G_1 and G_2 represent the first and second convolution layers, respectively. The edge features are computed by concatenating the source and destination node features for all edges:

$$E = concat(H_2[src], H_2[dst]),$$

where src and dst refer to source and destination nodes. The edge features are processed through a linear layer to compute edge values, representing the importance of each edge. These edge features are then combined with the encoded CNN features to enhance relational information

Algorithm 1 GCN with Edge-Based Readout

1: **Input:** Graph $G = (V, E)$ with node features h_i , for all $i \in V$, and number of input features in_feats .

2: **Output:** Mean edge feature μ_{edges} computed from transformed edge features.

3: **Step 1: Graph Convolution Layer 1**

4: Apply the first graph convolutional layer:

$$h_1 = ReLU(GraphConv(G, h, in_feats, in_feats))$$

5: Set $h = h_1$.

6: **Step 2: Graph Convolution Layer 2**

7: Apply the second graph convolutional layer:

$$h_2 = ReLU(GraphConv(G, h_1, in_feats, in_feats))$$

8: Set $h = h_2$.

9: **Step 3: Edge Feature Computation**

10: For each edge $(u, v) \in E$, concatenate the features of source node u and destination node v :

$$edge_feats = concat([h_2[u], h_2[v]], dim = 1)$$

11: Apply a linear transformation to compute edge values:

$$edge_values = Linear(edge_feats)$$

12: Store these edge values in G :

$$G.edata['edge_values'] = edge_values$$

13: **Step 4: Edge Readout**

14: Compute the mean of all edge values:

$$m_{edges} = mean(G.edata['edge_values'])$$

Return m_{edges} as the final output.

Multi-Head Self-Attention

The Multi-Head Self-Attention module applies attention mechanisms to extract contextual information from the embedding sequences. The module uses multiple attention heads, with each head responsible for focusing on different parts of the sequence. Given an input sequence X with dimensions (B, S, D) , where S represents the sequence length, the self-attention process involves the following steps

1. **Linear Transformation:** The input is projected into a higher-

$$P = \text{Linear}(X),$$

dimensional space:

resulting in a tensor P with dimensions (B, S, AD), where AD = 512 represents the total attention dimension.

2. **Reshape and Permutation:** The projected tensor is reshaped and

$$P' = P.\text{view}(B, N, S, H),$$

permuted to align with the attention heads:

where N = 8 represents the number of attention heads, and H = AD/N = 64.

3. **Scaled Dot-Product Attention:** The attention energy is calculated using the dot product between reshaped inputs and their transpose, then

$$E = \text{softmax}\left(\frac{P \cdot P^T}{\sqrt{H}}\right),$$

normalized with softmax

resulting in an attention tensor E with dimensions (B, N, S, S).

4. **Context Calculation:** The attention tensor is used to compute context by applying it to the reshaped inputs:

$$C = \text{matmul}(E, P'),$$

yielding context with dimensions (B, S, AD).

5. **Output Generation:** The final output is obtained by linear

$$O = \text{Linear}(C),$$

transformation and dropout:

where O has dimensions (B, S, D).

Multi-Head Cross-Attention

The Cross-Attention module computes attention between the encoder's output and the decoder's hidden state. Given the encoder's output EE and the decoder's hidden state DD , the cross-attention process involves the following steps:

1. **Linear Transformation:** Encoder output and decoder hidden state are projected into a common space:

$$E' = Linear(E),$$

$$D' = Linear(D),$$

resulting in E' with dimensions (B, L, AD) , and D' with dimensions $(B, 1, H)$

2. **Energy Calculation:** The attention energy is calculated by the dot product between the encoder and decoder projections, then normalized with softmax:

$$EC = matmul(E', D'^T),$$

$$A = softmax(\frac{EC}{\sqrt{H}}),$$

resulting in A with dimensions (B, L) .

3. **Context Calculation:** The context is calculated by applying the attention weights to the encoder projections:

$$C = matmul(A^T, E'),$$

yielding context with dimensions (B, AD) .

4. **Output Generation:** The context is transformed back to the original decoder dimension:

$$O = Linear(C),$$

resulting in the final output of the cross-attention module.

Decoder

The Decoder generates text-based captions using Long Short-Term Memory (LSTM) cells. Given an input sequence of embeddings and hidden states, the LSTM cell computes the following operations:

1. **Input Concatenation:** The LSTM input is generated by concatenating the embeddings and context from the cross-attention module:

$$I = \text{concat}(E', O),$$

2. **LSTM Cell Computation:** The LSTM cell processes the input to compute hidden state and cell state updates:

$$(h, c) = \text{LSTM}(I, (h_0, c_0)),$$

where h_0 and c_0 represent the initial hidden and cell states, respectively. The LSTM's hidden state is then passed through a linear layer to generate vocabulary logits, which are subsequently used to compute word probabilities through softmax.

The Complete Algorithm for the description generation process:

Initialization: Initialize the `EncoderCNN`, `GCH` (Graph Convolutional Network), and Decoder.

Set necessary hyperparameters (e.g., embedding size, vocabulary size, attention dimensions, etc.).

Input : A batch of images `IMAGE`, corresponding captions `CAPTION`, and graph representations `GRAPH`.

Output : `preds`, `alphas_cross`, and `alphas_self` as the model's output.

1 **Feature Extraction:**

- Pass `IMAGE` through `EncoderCNN` to extract visual features `encoded_input`.
- Pass `GRAPH` through `GCH` to extract graph-based features `edge_features`.

Combined Feature Processing:

- Concatenate `encoded_input` with `edge_features` along the feature dimension to get `encoder_out`.

Decoder Initialization:

- Initialize the hidden state `h` and cell state `c` of the LSTM using the mean of `encoder_out`.

Caption Generation:

- Embed the captions using an embedding layer `embedding`.
- Initialize a tensor `preds` to store predicted word probabilities.
- Initialize tensors `alphas_cross` and `alphas_self` to store attention scores.
- for each `s` in the sequence do
 1. Apply multi-head self-attention on the embeds of captions to get `self_context` and `self_attention`.
 2. Apply multi-head cross-attention on `encoder_out` with the current hidden state `h` to get `cross_context` and `cross_attention`.
 3. Concatenate `cross_context` with `self_context` to form the input for the LSTM.
 4. Update the LSTM cell state and hidden state.
 5. Pass the LSTM hidden state through a linear layer to get predicted word probabilities `preds[:, s]`.
 6. Store the attention scores `alphas_cross[:, s]` and `alphas_self[:, s]`.

Output:

- Return `preds`, `alphas_cross`, and `alphas_self` as the output from the model.

Caption Generation for Inference:

- Initialize the first word as (`<SOS>`).
 - For caption generation during inference, initiate a loop until `max_len` or a stopping token (`<EOS>`) is reached:
 1. Embed the current word and apply self-attention.
 2. Apply cross-attention with `encoder_out`.
 3. Predict the next word using the LSTM and a linear layer.
 4. Append the predicted word to the captions.
 5. If the predicted word is `<EOS>`, break the loop.
 - Return the generated caption and attention scores.
-

CHAPTER 5: IMPLEMENTATION AND TESTING

5.1. BiLSTM BASED TRADITIONAL MODEL (BENCHMARK MODEL)

5.1.1. Model Creation :

```
input_image = Input(shape=(4096, ),
name='Image_Feature_input')
fe1 = Dropout(0.5,
name='Dropout_image')(input_image)
fe2 = Dense(256, activation='relu',
name='Activation_Encoder')(fe1)
input_text = Input(shape=(max_length, ),
name='Text_input')
se1 = Embedding(vocab_size, 500, mask_zero=True,
name='Text_Feature')(input_text)
se2 = Dropout(0.5, name='Dropout_text')(se1)
se3 = Bidirectional(LSTM(512, name='Bidirectional-
LSTM', return_sequences=True))(se2)
se4 =
SeqSelfAttention(attention_activation='sigmoid',
name='Self-Attention')(se3)
se5 = Flatten()(se4)
se6 = Dense(256, activation='relu')(se5)
decoder1 = Concatenate(name='Concatenate')([fe2,
se6])
decoder2 = Dense(256, activation='relu',
name='Activation_Decoder')(decoder1)
output = Dense(vocab_size,
activation='softmax', name='Output')(decoder2)
model = Model(inputs=[input_image, input_text],
outputs=output)
model.compile(loss='categorical_crossentropy',
optimizer='adam')
```

5.1.2. Calculate BLEU Score :

```
actual, predicted = list(), list()

for key in tqdm(test):
    # get actual caption
    captions = mapping[key]
    # predict the caption for image
    y_pred = predict_description(model,
features[key], tokenizer, max_length)
    # split into words
    actual_captions = [caption.split() for caption
in captions]
    y_pred = y_pred.split()
    # append to the list
    actual.append(actual_captions)
    predicted.append(y_pred)

# calculate BLEU score
print("BLEU-1: %f" % corpus_bleu(actual, predicted,
weights=(1.0, 0, 0, 0)))
print("BLEU-2: %f" % corpus_bleu(actual, predicted,
weights=(0.5, 0.5, 0, 0)))
```

We have obtained the highest BLEU 1 score 0.551901 and BLEU 2 score 0.328970 in this approach after 25 epochs of training using the Generator.

5.2. GCN with Dual Attention Model

5.2.1. Vocabulary Creation :

```
#using spacy for the better text tokenization
spacy_eng = spacy.load("en_core_web_sm")
class Vocabulary:
    def __init__(self, freq_threshold):

        self.itos =
{0:"<PAD>", 1:"<SOS>", 2:"<EOS>", 3:"<UNK>"}

        self.stoi = {v:k for k,v in
self.itos.items()}

        self.freq_threshold = freq_threshold

    def __len__(self): return len(self.itos)

    @staticmethod
    def tokenize(text):
        return [token.text.lower() for token in
spacy_eng.tokenizer(text)]

    def build_vocab(self, sentence_list):
        frequencies = Counter()
        idx = 4

        for sentence in sentence_list:
            for word in self.tokenize(sentence):
                frequencies[word] += 1

                #add the word to the vocab if it
reaches minimum frequency threshold
                if frequencies[word] ==
self.freq_threshold:
```



```

        self.stoi[word] = idx
        self.itos[idx] = word
        idx += 1

    def numericalize(self, text):
        """ For each word in the text corresponding
        index token for that word form the vocab built as
        list """
        tokenized_text = self.tokenize(text)
        return [ self.stoi[token] if token in
self.stoi else self.stoi["<UNK>"] for token in
tokenized_text ]

```

5.2.2. Image Object-Relationship Graph:

```

class ObjectDetectorAndFeatureExtractor(nn.Module):
    def __init__(self, pretrained=True,
iou_threshold=0.4):
        super(ObjectDetectorAndFeatureExtractor,
self).__init__()

        # Object detection model (Faster R-CNN)
        self.object_detection_model =
fasterrcnn_resnet50_fpn(pretrained=pretrained)
        self.object_detection_model.eval()

        # Feature extraction model (VGG19)
        vgg = models.vgg19(pretrained=pretrained)
        for param in vgg.parameters():
            param.requires_grad = False
        self.feature_extractor = vgg.features[:36]
        self.feature_extractor =
nn.Sequential(*self.feature_extractor)
        self.iou_threshold = iou_threshold

    def forward(self, image):

```

```

        #graph_list = []
        image = image.to(device)
        #for image in images:
        # Perform object detection
        with torch.no_grad():
            detections =
self.object_detection_model([image])

        g = dgl.DGLGraph()
        g = g.to(device)

        if len(detections[0]['boxes']) == 0:
            # If no objects detected, use whole
image features
            image_feature =
self.feature_extractor(image.unsqueeze(0))
            image_feature =
image_feature.view(image_feature.size(0), 49, -1)  #
(batch_size, 49, 2048)
            g.add_nodes(1, {'features':
image_feature})
            g.add_edges(0, 0)  # self-loop edge
            #graph_list.append(g)
            return g

        # Add nodes for each detected object and
assign features
        for box in detections[0]['boxes']:
            x1, y1, x2, y2 = map(int, box.tolist())
            if x1 >= x2 or y1 >= y2:
                # If Invalid bounding box, skip it
                continue

            # Extract object region
            object_region = image[:, y1:y2, x1:x2]
            if object_minimum frequency == 0 or
object_region.size(2) == 0:
                # If Zero-size crop, skip it
                continue

```

```

        # Resize to a common size
        object_region = f.resize(object_region,
(224, 224))

        # Extract features using VGG19
        features =
self.feature_extractor(object_region.unsqueeze(0))
        features =
features.view(features.size(0), 49, -1) #
(batch_size, 49, 2048)
        g.add_nodes(1, {'features': features})

num_objects = g.number_of_nodes()

# Add edges based on IoU
for i in range(num_objects):
    for j in range(num_objects):
        if i != j:
            bbox_i =
detections[0]['boxes'][i]
            bbox_j =
detections[0]['boxes'][j]
            iou = self.calculate_iou(bbox_i,
bbox_j)

            if iou > self.iou_threshold:
                g.add_edges(i, j)

    return g

def calculate_iou(self, bbox1, bbox2):
    xmin = max(bbox1[0], bbox2[0])
    ymin = max(bbox1[1], bbox2[1])
    xmax = min(bbox1[2], bbox2[2])
    ymax = min(bbox1[3], bbox2[3])

    intersection_area = max(0, xmax - xmin) *
max(0, ymax - ymin)

```

```

        area1 = (bbox1[2] - bbox1[0]) * (bbox1[3] -
bbox1[1])
        area2 = (bbox2[2] - bbox2[0]) * (bbox2[3] -
bbox2[1])
        union_area = area1 + area2 -
intersection_area

        if union_area == 0:
            return 0

        iou = intersection_area / union_area
        return iou
class EncoderCNN(nn.Module):
    def __init__(self, pretrained=True):
        super(EncoderCNN, self).__init__()

        # Load pre-trained VGG19 for feature
extraction
        vgg = models.vgg19(pretrained=pretrained)
        for param in vgg.parameters():
            param.requires_grad_(False)
        self.vgg_features = vgg.features[:36]
        self.vgg_features =
nn.Sequential(*self.vgg_features)

    def forward(self, images):
        features = self.vgg_features(images) #
(batch_size, 512, 14, 14)

        # Reshape the feature map
        features = features.view(features.size(0), -
1, 49) # (batch_size, 2048, 49)
        features = features.permute(0, 2, 1) #
(batch_size, 49, 2048)

        return features

```

5.2.3. GCN Model:

```
class GCN(nn.Module):
    def __init__(self, in_feats):
        super(GCN, self).__init__()

        # GCN layers
        self.conv1 = GraphConv(in_feats,
                                in_feats, allow_zero_in_degree=True)
        self.conv2 = GraphConv(hidden_feats,
                                in_feats, allow_zero_in_degree=True)

        # linear layer for edge computation
        self.edge_linear = nn.Linear(in_feats * 2,
                                       in_feats)

    def forward(self, g):

        h = g.ndata['features']
        # Perform GCN
        h = self.conv1(g, h)
        h = torch.relu(h)
        h = self.conv2(g, h)
        h = torch.relu(h)
        # Concatenate src and dest node features for
        all edges
        src_feats = h[g.edges()[0]]
        dst_feats = h[g.edges()[1]]

        edge_feats = torch.cat([src_feats,
                                dst_feats], dim=2)
        edge_values =
self.edge_linear(edge_feats).squeeze(dim=1)

        g.edata['edge_values'] = edge_values
```

```

        mean_edge_feat = dgl.readout_edges(g,
'edge_values', op='mean')

    return mean_edge_feat

```

5.2.4. Multi Head Self Attention Module :

```

class MultiHeadSelfAttention(nn.Module):
    def __init__(self, input_dim, num_heads,
attention_dim):
        super(MultiHeadSelfAttention,
self).__init__()
        self.num_heads = num_heads
        self.attention_dim = attention_dim
        self.head_dim = attention_dim // num_heads

        self.linear = nn.Linear(input_dim,
attention_dim)
        self.output_linear =
nn.Linear(attention_dim, input_dim)

        # Initialize weights
        self._init_weights()

    def _init_weights(self):
        # Xavier initialization for the linear
layers
        init.xavier_uniform_(self.linear.weight)
init.xavier_uniform_(self.output_linear.weight)

    def forward(self, inputs):
        batch_size, seq_length, input_dim =
inputs.size()

```

```

        projected_inputs = self.linear(inputs) #
        (batch_size, seq_length, attention_dim)
        reshaped_inputs =
projected_inputs.view(batch_size, seq_length,
self.num_heads, self.head_dim) \

.permute(0, 2, 1, 3) # (batch_size, num_heads,
seq_length, head_dim)

        # Calculate attention energy
        transposed_inputs =
reshaped_inputs.transpose(-2, -1) # (batch_size,
num_heads, head_dim, seq_length)
        energy = torch.matmul(reshaped_inputs,
transposed_inputs) # (batch_size, num_heads,
seq_length, seq_length)

        # Calculate attention weights
        attention = F.softmax(energy /
torch.sqrt(torch.tensor(self.head_dim,
dtype=inputs.dtype)), dim=-1) # (batch_size,
num_heads, seq_length, seq_length)

        # Apply attention to get the context
        context = torch.matmul(attention,
reshaped_inputs) # (batch_size, num_heads,
seq_length, head_dim)

        # Rearrange context and prepare output
        context = context.transpose(1,
2).contiguous().view(batch_size, seq_length,
self.attention_dim) # (batch_size, seq_length,
attention_dim)
        output = self.output_linear(context)
        attention = attention.permute(0, 2, 1, 3) #
(batch_size, seq_length, num_heads , seq_length)

        return output, attention

```

5.2.5. Multi Head Cross Attention Module :

```
class MultiHeadCrossAttention(nn.Module):
    def __init__(self, encoder_dim, decoder_dim,
num_heads, attention_dim):
        super(MultiHeadCrossAttention,
self).__init__()
        self.num_heads = num_heads
        self.attention_dim = attention_dim
        self.head_dim = attention_dim // num_heads

        self.encoder_linear = nn.Linear(encoder_dim,
attention_dim)
        self.decoder_linear = nn.Linear(decoder_dim,
self.head_dim)
        self.output_linear =
nn.Linear(attention_dim, decoder_dim)

        self._init_weights()

    def _init_weights(self):
        # Initialize weights with Xavier

init.xavier_uniform_(self.encoder_linear.weight)

init.xavier_uniform_(self.decoder_linear.weight)

init.xavier_uniform_(self.output_linear.weight)

    def forward(self, encoder_out, decoder_hidden):
        batch_size, num_features, _ =
encoder_out.size()
```



```

        encoder_proj =
self.encoder_linear(encoder_out)  # (batch_size,
num_features, attention_dim)

        decoder_proj =
self.decoder_linear(decoder_hidden)  # (batch_size,
head_dim)

        # Prepare the reshaped encoder projection
        encoder_proj = encoder_proj.view(batch_size,
self.num_heads, num_features, self.head_dim)  #
(batch_size, num_heads, num_features, head_dim)

        # Reshape decoder projection for cross-
attention
        decoder_proj =
decoder_proj.unsqueeze(1).unsqueeze(1)  #
(batch_size, 1, 1, head_dim)

        # Compute energy for cross-attention
        energy = torch.matmul(encoder_proj,
decoder_proj.transpose(-2, -1))  # (batch_size,
num_heads, num_features, 1)

        # Apply scaled dot-product softmax for
attention scores
        attention = F.softmax(energy /
torch.sqrt(torch.tensor(self.head_dim,
dtype=encoder_out.dtype)), dim=-2)  # (batch_size,
num_heads, num_features, 1)

        # Get context by applying attention to
encoder projections

```

```

        context = torch.matmul(attention.transpose(-
2, -1), encoder_proj).squeeze(-2)  # (batch_size,
num_heads, head_dim)

        # Merge multi-head context into one
attention_dim

        context = context.view(batch_size,
self.head_dim * self.num_heads)  # (batch_size,
attention_dim)

        # Final output with a linear layer
output = self.output_linear(context)
return output, attention.squeeze(-1)

```

5.2.6. LSTM Decoder Module :

```

class Decoder(nn.Module):
    def __init__(self, embed_size, vocab_size,
attention_dim, gcn_encode_dim, encoder_dim,
decoder_dim, num_heads, drop_prob=0.2):
        super().__init__()
        # model parameters
        self.vocab_size = vocab_size
        self.attention_dim = attention_dim
        self.decoder_dim = decoder_dim
        self.num_heads = num_heads

        self.edge_feature = GCN(gcn_encode_dim)
        self.embedding = nn.Embedding(vocab_size,
embed_size)

        # Multi-head cross attention

```

```

        self.attention =
MultiHeadCrossAttention(encoder_dim, decoder_dim,
num_heads, attention_dim)

        # Multi-head self-attention
        self.attention_self =
MultiHeadSelfAttention(embed_size, num_heads,
attention_dim)

        self.init_h = nn.Linear(encoder_dim,
decoder_dim)

        self.init_c = nn.Linear(encoder_dim,
decoder_dim)

        self.lstm_cell = nn.LSTMCell(embed_size +
decoder_dim, decoder_dim, bias=True)

        self.encoded = nn.Dropout(drop_prob)

        self.fcn = nn.Linear(decoder_dim,
vocab_size)

        self.drop = nn.Dropout(drop_prob)

    def forward(self, encoded_input, captions,
graphs):
        embeds = self.embedding(captions)

        seq_length = len(captions[0])-1 # Exclude
the last one

        batch_size = captions.size(0)

        edge_features = self.edge_feature(graphs)
        encoded_input = torch.cat((edge_features,
features), dim =2)

        encoder_out = self.encoded(encoder_out)

        h, c = self.init_hidden_state(encoder_out)

        preds = torch.zeros(batch_size, seq_length,
self.vocab_size).to(encoder_out.device)

```

```

        alphas = torch.zeros(batch_size, seq_length,
self.num_heads,
encoder_out.size(1)).to(encoder_out.device)
        alphas_self = torch.zeros(batch_size,
seq_length+1, self.num_heads,
(seq_length+1)).to(encoder_out.device)
        for s in range(seq_length):
            # Multi-head self-attention over
captions
            embeds_attended_self, alpha_self =
self.attention_self(embeds)

            # Multi-head cross-attention with
encoder output
            output, alpha =
self.attention(encoder_out, h)

            lstm_input =
torch.cat((embeds_attended_self[:, s], output),
dim=1)
            h, c = self.lstm_cell(lstm_input, (h,
c))
            output = self.fcn(self.drop(h))

            preds[:, s] = output
            alphas[:, s] = alpha
            alphas_self[:,s] = alpha_self[:, -1, :, :]
        return preds, alphas, alphas_self
    def generate_caption(self, encoder_out,
max_len=25, vocab=None):
        batch_size = encoder_out.size(0)
        h, c = self.init_hidden_state(encoder_out)

```

```

        alphas = []
        alphas_self = []
        captions = []
        word =
torch.tensor(vocab.stoi['<SOS>']).view(1, -
1).to(encoder_out.device)
        embeds = self.embedding(word)
        for i in range(max_len):
            embeds_attended_self, alpha_self =
self.attention_self(embeds)

        alphas_self.append(alpha_self.cpu().detach().numpy()
)

            output, alpha =
self.attention(encoder_out, h)

        alphas.append(alpha.cpu().detach().numpy())
            lstm_input =
torch.cat((embeds_attended_self[:, 0], output),
dim=1)

            h, c = self.lstm_cell(lstm_input, (h,
c))

            output = self.fcn(self.drop(h))
            output = output.view(batch_size, -1)
            predicted_word_idx =
output.argmax(dim=1)

        captions.append(predicted_word_idx.item())
            if vocab.itos[predicted_word_idx.item()]
== "<EOS>":
                break

```

```

        embeds =
self.embedding(predicted_word_idx.unsqueeze(0))
        return [vocab.itos[idx] for idx in
captions], alphas, alphas_self
    def init_hidden_state(self, encoder_out):
        mean_encoder_out = encoder_out.mean(dim=1)
        h = self.init_h(mean_encoder_out)
        c = self.init_c(mean_encoder_out)
        return h, c

```

5.2.7. Encoder Decoder Module :

```

class EncoderDecoder(nn.Module):
    def __init__(self, embed_size, vocab_size,
attention_dim, gcn_encode_dim, encoder_dim, decoder_dim
, drop_prob=0.3):
        super().__init__()
        self.encoder = EncoderCNN()
        self.obj_feature =
ObjectDetectorAndFeatureExtractor()
        #self.edge_feature = GCN(2048, 512, 2048)
        self.decoder = Decoder(
            embed_size=embed_size,
            vocab_size = len(train_dataset.vocab),
            attention_dim=attention_dim,
            gcn_encode_dim = gcn_encode_dim,
            encoder_dim=encoder_dim,
            decoder_dim=decoder_dim,
            num_heads = 2
        )

    def forward(self, images, captions, graphs):
        features = self.encoder(images)

        outputs = self.decoder(features, captions,
graphs)

```

```

        #print(graphs[0])
        return outputs

#init model
learning_rate = 0.001
model = EncoderDecoder(
    embed_size=200,
    vocab_size = len(train_dataset.vocab),
    attention_dim=256,
    Gcn_encode_dim = 2048,
    encoder_dim=4096,
    decoder_dim=256
).to(device)

print(model)
total_params = sum(p.numel() for p in
model.parameters())
print("Total parameters:", total_params)
for name, param in model.named_parameters():
    print(name, param.shape)

criterion =
nn.CrossEntropyLoss(ignore_index=train_dataset.vocab
.stoi["<PAD>"])
optimizer = optim.Adam(model.parameters(),
lr=learning_rate)

```

5.2.6. Model Training :

```

num_epochs = 10
print_every = 100

for epoch in range(1,num_epochs+1):
    for idx, (image, captions, graph) in
enumerate(iter(data_loader)):

```

```

        image,captions, graph=
image.to(device),captions.to(device),
graph.to(device)

        # Zero the gradients.
optimizer.zero_grad()

        # Feed forward
outputs,attentions_cross, attentions_self =
model(image, captions, graph)

        # Calculate the batch loss.
targets = captions[:,1:]
loss = criterion(outputs.view(-1,
vocab_size), targets.reshape(-1))

        # Backward pass.
loss.backward()

        # Update the parameters in the optimizer.
optimizer.step()

        if (idx+1)%print_every == 0:
            print("Epoch: {} loss:
{:0.5f}".format(epoch,loss.item()))

            #generate the caption
model.eval()
            with torch.no_grad():
                dataiter = iter(test_data_loader)
                img,_,graph = next(dataiter)
                features =
model.encoder(img[0:1].to(device))
                edge_features =
model.decoder.edge_feature(graph.to(device))
                encoded_input =
torch.cat((edge_features, features), dim =2)

```



```

        caps, alphas, alphas_self =
model.decoder.generate_caption(encoded_input, vocab=train_dataset.vocab)
        caption = ' '.join(caps)
        show_image(img[0], title=caption)

```

5.2.7. Description Generation :

```

#generate caption
def get_caps_from(features_tensors, graph):
    #generate the caption
    model.eval()
    with torch.no_grad():
        features =
model.encoder(features_tensors.to(device))
        edge_features =
model.decoder.edge_feature(graph.to(device))
        encoded_input = torch.cat((edge_features,
features), dim =2)

        caps, alphas, alphas_self =
model.decoder.generate_caption(encoded_input, vocab=train_dataset.vocab)
        caption = ' '.join(caps)

show_image(features_tensors[0], title=caption)

    return caps, alphas, alphas_self

```

5.2.8. Testing the Model :

```

dataiter = iter(test_data_loader)

```

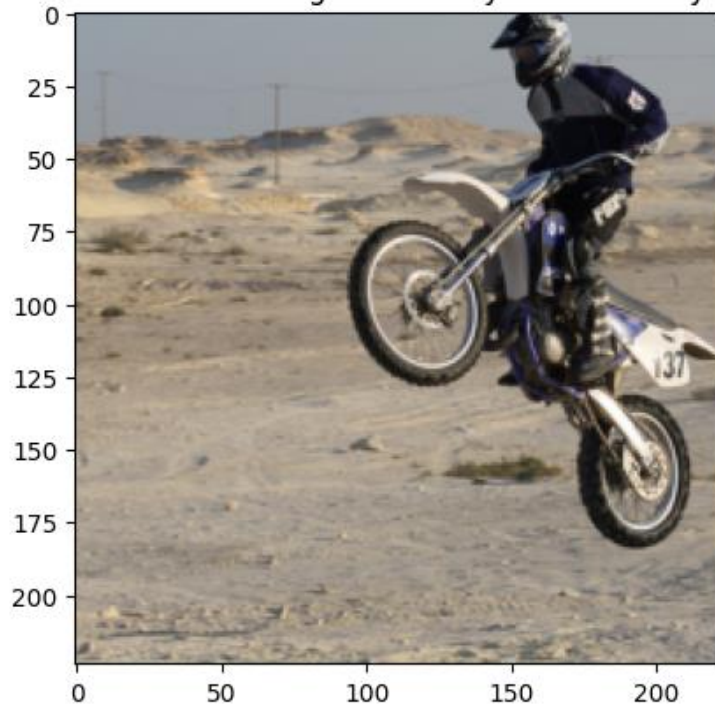
```

images,_,graph = next(dataiter)

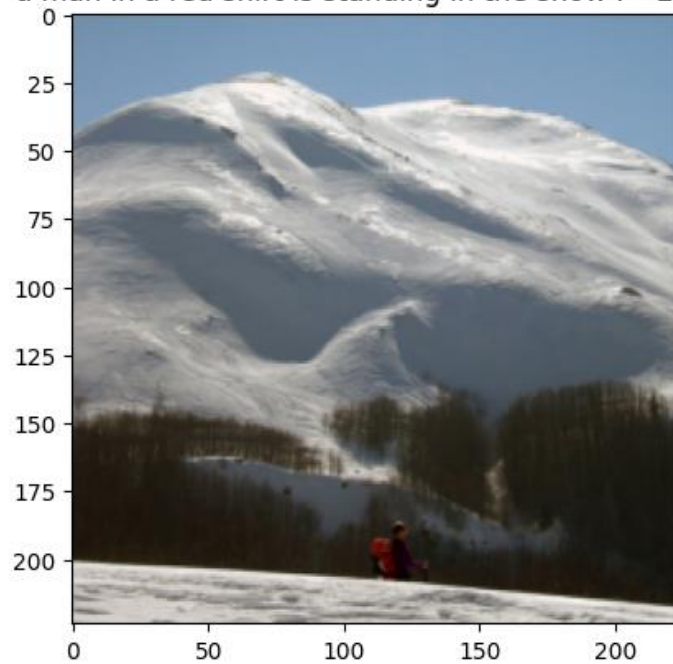
img = images[0].detach().clone()
img1 = images[0].detach().clone()
caps,alphas,alphas_self =
get_caps_from(img.unsqueeze(0),graph)

```

a man in a red shirt is riding a red and yellow motorcycle . <EOS>






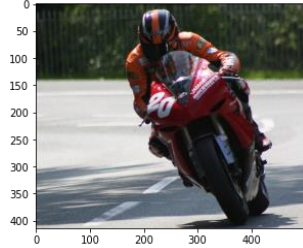

a man in a red shirt is standing in the snow . <EOS>

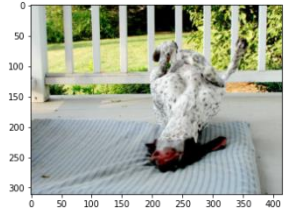





CHAPTER 6: RESULTS AND DISCUSSION

6.1. OUTPUT

Image	Original Descriptions	BiLSTM Output	GCN Output
<p>2654514044_a70a6e2c21.jpg</p> 	<ul style="list-style-type: none"> • brown dog running • brown dog running over grass • brown dog with its front paws off the ground on grassy surface near red and purple flowers • dog runs across grassy lawn near some flowers • yellow dog is playing in grassy area near flowers 	<p>brown dog running through green grass</p>	<p>A brown dog is running through a grassy field</p>
<p>466956209_2ffcea3941.jpg</p> 	<ul style="list-style-type: none"> • distant person is climbing up very sheer mountain • panoramic distance view of rock climber going up steep mountain wall • person rock climbing • there is person scaling very steep rock wall • this person is climbing the side of mountain 	<p>man is climbing down mountain path</p>	<p>A man in a red shirt is climbing a rock wall</p>

<p>1554713437_61b64527dd.jpg</p> 	<ul style="list-style-type: none"> ● big hound dog walking on log in the woods ● brown dog walks on top of felled tree trunk ● grey dog walks on top of fallen tree in the woods ● skinny brown dog walks across large piece of fallen tree ● the skinny brown dog is walking on fallen tree 	<p>skinny brown dog walks on top of felled tree</p>	<p>A brown dog walks through a forest</p>
<p>166321294_4a5e68535f.jpg</p> 	<ul style="list-style-type: none"> ● man is riding on red motorcycle ● motorcycle driver dressed in orange gear swerves to the right ● motorcyclist on red speed bike leans into sharp turn ● motorcyclist crouches low as he rounds turn ● this person is on red motorcycle 	<p>motorcycle rider is wearing black helmet and matching protective gear</p>	<p>A man in a red shirt is riding a red and white motorcycle</p>
<p>216172386_9ac5356dae.jpg</p> 	<ul style="list-style-type: none"> ● bike sits atop rise with mountains in the background ● man wearing red uniform and helmet stands on his motorbike ● motocross bike is being ridden over rocks ● motocross biker about to descend ● the motorcyclist has reached the summit 	<p>man wearing black outfit is riding bike through the sand</p>	<p>A man on a dirt bike is riding a hill</p>

<p>128912885_8350d277a4.jpg</p> 	<ul style="list-style-type: none"> • dog lays on mattress on the porch • dog rolls on mattress placed on porch and scratches his back • spotted dog rolling over on pad placed on porch • white dog rolling on its back on porch • the spotted dog rolls on its back 	<p>the dog rolls over over the red brick windows</p>	<p>A white dog is jumping over a red and white dog</p>
<p>112178718_87270d9b4d.jpg</p> 	<ul style="list-style-type: none"> • guy stands in the sand with snowboard behind him • man holds surfboard on the beach • man holds his snowboard in the sand • man with his surfboard stands in the sand • man is standing on white sand and holding snowboard 	<p>young boy wearing yellow shirt is jumping in the air on his skateboard</p>	<p>A man in a black shirt and white shorts is jumping in a sand dune</p>

<p>454686980_7517fe0c2e.jpg</p> 	<ul style="list-style-type: none"> ● girl who looks upset with her arms crossed ● an indian woman in black shirt stands with her arms crossed looking at two other people talking to each other ● person in red and black shirt has their arms crossed and looks at the camera ● teenage girl is standing with her arms crossed in busy street ● woman in dark shirt standing alone in front of yellow car 	<p>woman in black shirt and hat is walking past ice cream</p>	<p>A woman in a blue shirt and blue jeans is standing on a sidewalk with a woman in a blue shirt</p>
<p>561417861_8e25d0c0e8.jpg</p> 	<ul style="list-style-type: none"> ● city street has three people walking in different directions ● man stands by building while others are nearby ● men walk down street with skyscrapers in the background ● the buildings are brown and the sky is blue ● three men on city street with the shot taken up and out toward the buildings 	<p>man in black hat is standing in front of white building</p>	<p>Three man are standing outside a building</p>

6.2. WORKING OF MULTI HEADED ATTENTION

The multihead attention mechanism runs through the image several times in parallel. In this project there are two attention heads for each image. This allows for attending to parts of the sequence differently. Once the output from each attention head is obtained, they are concatenated and linearly transformed into expected dimensions.

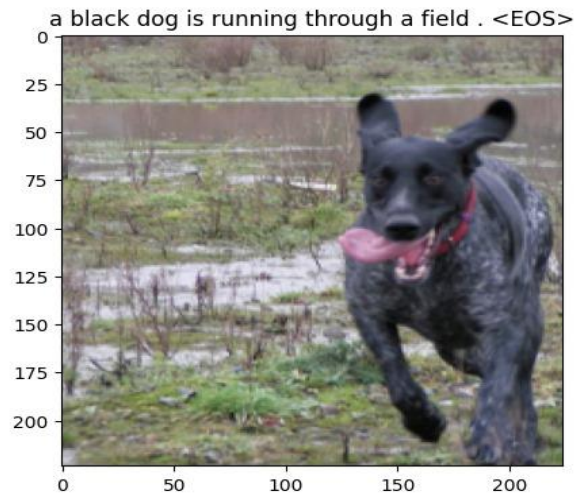


Figure 10(a)

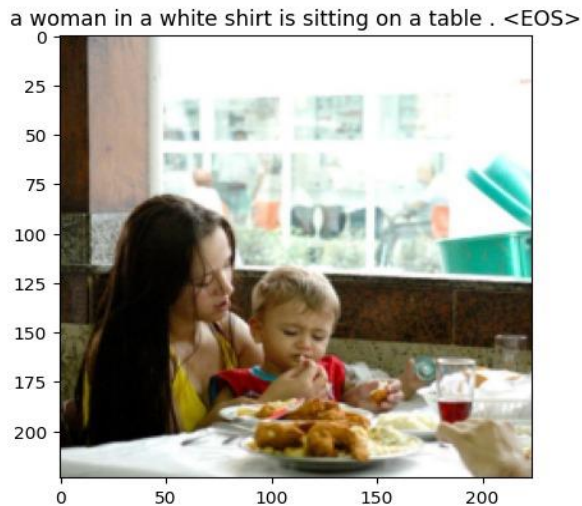


Figure 10(b)

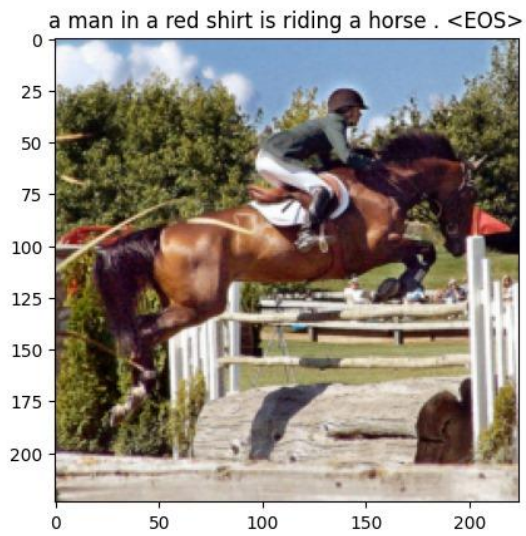


Figure 10(c)

Figure 10 : Few Outputs of the GCN Model

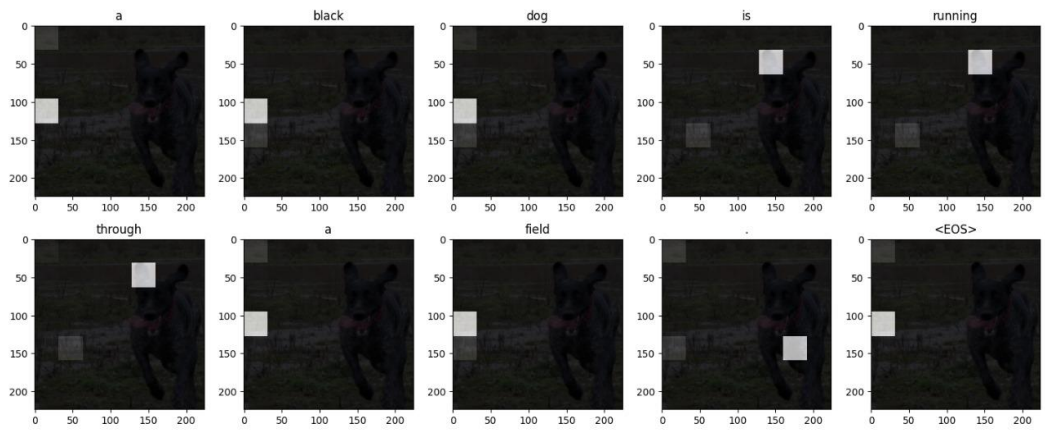


Figure 11(a)

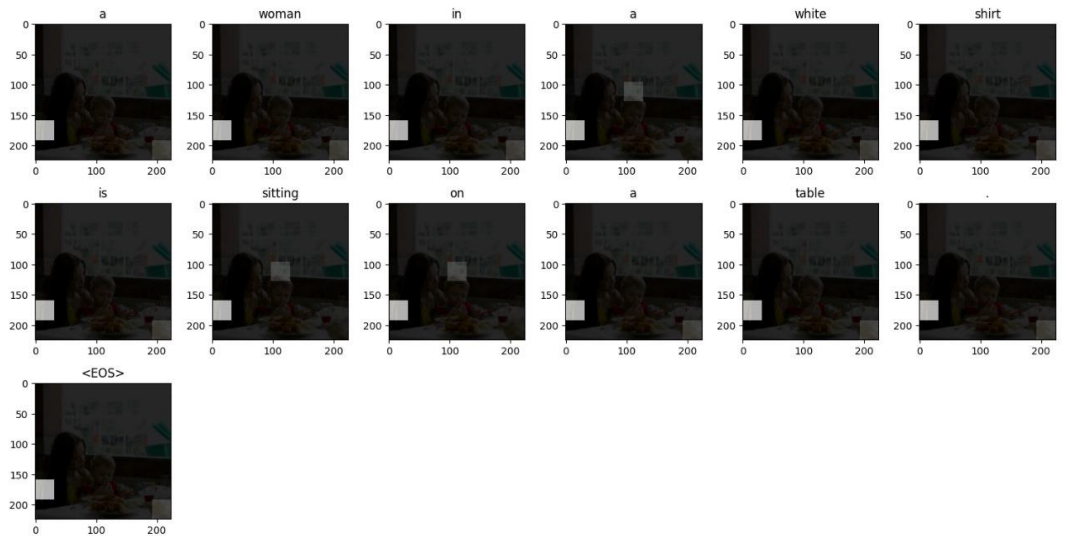


Figure 11(b)

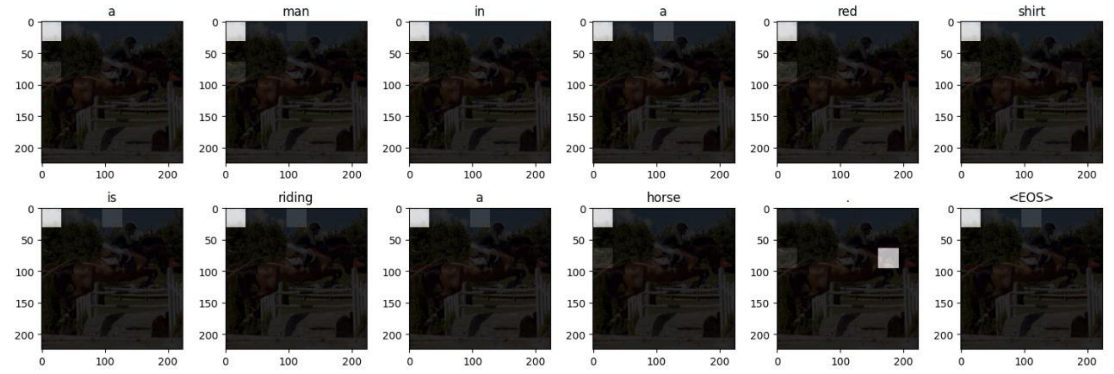


Figure 11(c)

Figure 11 : Working of two attention heads for the images in Figure 10 respectively.

There are two squares present in each of the images, those squares represent the attention heads. The attention heads run in parallel and based on their combined output we can predict the words to describe the image in a given instant.

CHAPTER 7: CONCLUSION

In conclusion, our project has demonstrated the utility and effectiveness of GCN combined with attention mechanisms to generate descriptions for images. By leveraging the relationships of different objects in image, the GCN enables us to capture complex contextual dependencies effectively. And the attention mechanisms allow our model to generate human-like descriptions.

7.1. LIMITATIONS OF THE PROJECT

The model is trained with only 7000 image data from the ‘flicker8k’ dataset, the descriptions present in the dataset are very short, this makes the model unable to generate longer contextual descriptions. Also we are using Faster R-CNN to detect objects present in the given image, this leads to the problem when there are objects present in the image which are not identified by the Faster R-CNN model. Due to the lack of training data the model performs poorly while describing the color of a certain object.

7.2. FUTURE SCOPE

In future, the model has to be trained with larger datasets like MSCOCO or flicker30k for better prediction on unknown images. For better results, a custom description data should be used, which will describe the images in more detail. And lastly, instead of using the pre-trained Faster R-CNN directly, we can custom-train the Faster R-CNN with more data to identify more objects in the image.

References

1. Sasibhooshan, R., Kumaraswamy, S. & Sasidharan, S. Image caption generation using Visual Attention Prediction and Contextual Spatial Relation Extraction. J Big Data 10, 18 (2023). <https://doi.org/10.1186/s40537-023-00693-9>
2. L. Li, S. Tang, Y. Zhang, L. Deng, and Q. Tian, "GLA: Global–Local Attention for Image Description," in IEEE Transactions on Multimedia, vol. 20, no. 3, pp. 726-737, March 2018, doi: 10.1109/TMM.2017.2751140.
3. Jiang, W., Ma, L., Jiang, YG., Liu, W., Zhang, T. (2018). Recurrent Fusion Network for Image Captioning. In: Ferrari, V., Hebert, M., Sminchisescu, C., Weiss, Y. (eds) Computer Vision – ECCV 2018. ECCV 2018. Lecture Notes in Computer Science(), vol 11206. Springer, Cham. https://doi.org/10.1007/978-3-030-01216-8_31
4. Yao, Ting, et al. "Exploring visual relationship for image captioning," Proceedings of the European Conference on Computer Vision (ECCV). 2018.
5. Fu, Kun, et al. "Aligning where to see and what to tell: Image captioning with region-based attention and scene-specific contexts," IEEE Transactions on Pattern Analysis and Machine Intelligence 39.12 (2016): 2321-2334.
6. L. Huo, L. Bai and S. -M. Zhou, "Automatically Generating Natural Language Descriptions of Images by a Deep Hierarchical Framework," in IEEE Transactions on Cybernetics, vol. 52, no. 8, pp. 7441-7452, Aug. 2022, doi: 10.1109/TCYB.2020.3041595.
7. Xia, X., Wang, L., Ding, K., Xiang, S., & Pan, C., 2019, Dense semantic embedding network for image captioning, <http://www.elsevier.com/locate/patcog>
8. Krause, J., Johnson, J., Krishna, R., & Fei-Fei, L. A Hierarchical Approach for Generating Descriptive Image Paragraphs. At the Conference on Computer Vision and Pattern Recognition(CVPR), 2017
9. Al-Malla, M. A., Jafar, A., & Ghneim, N. Image captioning model using attention and object features to mimic human image understanding, Journal of Big Data (2022) <https://doi.org/10.1186/s40537-022-00571-w>

10. J. Johnson, A. Karpathy, and L. Fei-Fei. DenseCap: Fully Convolutional Localization Networks for Dense Captioning. In CVPR, 2016.
11. Sepp Hochreiter, Jürgen Schmidhuber; Long Short-Term Memory. *Neural Comput* 1997; 9 (8): 1735–1780. doi:
<https://doi.org/10.1162/neco.1997.9.8.1735>
12. Kipf, T.N. and Welling, M., 2016. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907*.
13. Simonyan, K. and Zisserman, A., 2014. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*.
14. Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A.N., Kaiser, Ł. and Polosukhin, I., 2017. Attention is all you need. *Advances in neural information processing systems*, 30.
15. Ren, S., He, K., Girshick, R. and Sun, J., 2015. Faster r-cnn: Towards real-time object detection with region proposal networks. *Advances in neural information processing systems*, 28.
16. Vinyals, O., Toshev, A., Bengio, S. and Erhan, D., 2015. Show and tell: A neural image caption generator. In *Proceedings of the IEEE conference on computer vision and pattern recognition* (pp. 3156-3164).
17. <https://www.kaggle.com/code/mdteach/torch-data-loader-flicker-8k>
(29/04/24)
18. **Dataset link** - <https://paperswithcode.com/dataset/flickr-8k>