Islamic University of Technology (IUT)
Department of Computer Science and Engineering (CSE)

Bangla Text Summarization using Deep Learning

Authors
Ahmed Sadman Muhib, 160041025
Shakleen Ishfar, 160041029
AKM Nahid Hasan, 160041043

Supervisor
Dr. Abu Raihan Mostofa Kamal
Professor, Department of CSE

A thesis submitted to the Department of CSE
in partial fulfillment of the requirements for the degree of B.Sc.
Engineering in CSE
Academic Year: 2019-2020
March, 2021

DECLARATION OF AUTHORSHIP

This is to certify that the work presented in this thesis is the outcome of the analysis and experiments carried out by under the supervision of Dr. Abu Raihan Mostofa Kamal, Professor of the Department of Computer Science and Engineering (CSE), Islamic University of Technology (IUT), Dhaka, Bangladesh. It is also declared that neither of this thesis nor any part of this thesis has been submitted anywhere else for any degree or diploma. Information derived from the published and unpublished work of others has been acknowledged in the text and a list of references is given.

Authors:

Ahmed Sadman

Ahmed Sadman Muhib
Student ID - 160041025

Shakleen Ishfar

Shakleen Ishfar
Student ID - 160041029

Nahid Hasan

AKM Nahid Hasan
Student ID - 160041043

Approved By:

Approved By:

Dept. of Computer Science and Engineering (CSE)
DHAKA
IUT

Supervisor

[illegible]

Supervisor:

[illegible] 22.3.24

Dr. Abu Raihan Mostofa Kamal, PhD
Professor
Department of Computer Science and Engineering (CSE)
Islamic University of Technology (IUT), OIC

Acknowledgement

We would like to express our grateful appreciation for **Dr. Abu Raihan Mostafa Kamal**, Department of Computer Science & Engineering, Islamic University of Technology for being our adviser and mentor. His motivation, suggestions and insights for this research have been invaluable. Without his support and proper guidance this research would never have been possible. His valuable opinion, time and input provided throughout the thesis work, from first phase of thesis topics introduction, subject selection, proposing algorithm, modification till the project implementation and finalization which helped us to do our thesis work in proper way. We are really grateful to him.

ABSTRACT

In this thesis, we present our work regarding text summarization. Text summarization is the technique for generating concise and precise summaries of voluminous texts while focusing on the sections that convey useful information without losing the overall meaning. In this age of information, there are vast quantities of textual data available. Example sources include online documents, articles, news, and user reviews of various products and services. We can present the underlying information present in these texts concisely through summaries. However, generating summaries for such a large source of text documents by hand is troublesome. We can utilize neural machine summarization systems to generate summaries automatically. These systems leverage the power of deep learning models. Recently, with the invention of Transformer architecture, modern summarization systems have achieved revolutionary performance gains. Efficient transformer-based summarization systems exist for English and other popular languages, but not Bangla. In this research, we present an efficient transformer-based text summarization system for the Bangla language. We use subword encoding to eliminate the problem of rare and unknown words. We have created a large dataset, consisting of 600 thousand news articles, to train our model. We trained a 6 million parameter model that is capable of producing accurate summaries. We evaluated out summaries by observing it's generative performance.

Contents

List of Figures

List of Tables

Listings

List of Algorithms

1 Introduction

1.1 Text Summarization

Text summarization is the technique for generating concise and precise summaries of voluminous texts while focusing on the sections that convey useful information without losing the overall meaning. In this age of information, there are vast quantities of textual data available. Examples include online documents, articles, news, and reviews. We can present the underlying information present in these large pieces of texts concisely through summaries.

1.1.1 Neural Machine Summarization

Automatic text summarization aims to transform lengthy documents into shortened versions, which could be difficult and costly to undertake if done manually. We can use machine learning models to automate this task. This process can be decomposed into two parts:

1. **Natural Language Understanding (NLU)**: This part of the task is about understanding the input document. From this understanding, the model will identify key-points for later use.
2. **Natural Language Generation (NLG)**: This part deals with generating the actual summary. The model will generate the summaries using the key points from the NLU part.

1.1.2 Text Summarization Approaches

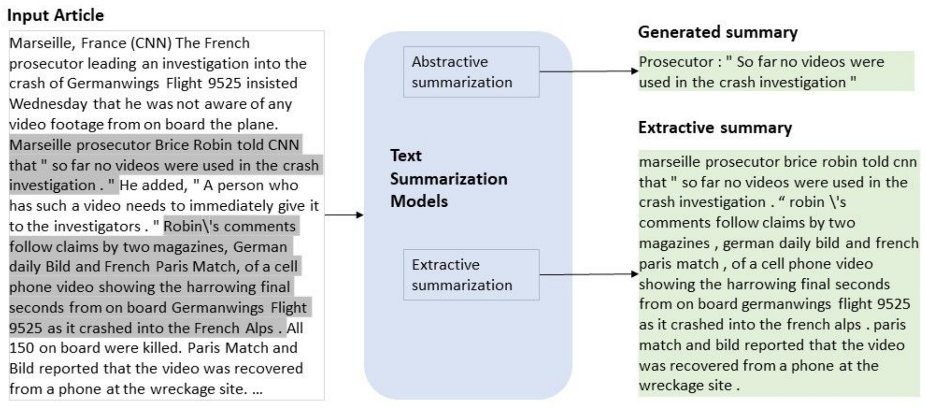
The two dominant summarization approaches are as follows:

1. **Extractive approach**: We first pull a subset of words representing the key-points. Then we combine these words to make a summary.
2. **Abstractive approach**: In this approach, we try to understand the input document and then generate the summary from scratch based on that information.

We can see both approaches illustrated in figure 1.

1.1.3 Necessity of Automated Summarization

We currently enjoy quick access to enormous amounts of digital information. Most of the data circulating in the digital space are non-structured textual data. However, most of this information may not convey the intended meaning. For example, imagine that you are looking for specific information from an online news article. To get the information you want, you may have to dig through the entire news content. You might waste lots of time weeding out the unnecessary data before getting the information you want. Therefore, using automatic text summarizers, capable of extracting useful information, is becoming vital. Implementing summarization can enhance the readability of documents, reduce the time spent searching for information, and allow for more information to be fitted in a particular area.



Marseille, France (CNN) The French prosecutor leading an investigation into the crash of Germanwings Flight 9525 insisted Wednesday that he was not aware of any video footage from on board the plane. Marseille prosecutor Brice Robin told CNN that " so far no videos were used in the crash investigation . " He added, " A person who has such a video needs to immediately give it to the investigators . " Robin\'s comments follow claims by two magazines, German daily Bild and French Paris Match, of a cell phone video showing the harrowing final seconds from on board Germanwings Flight 9525 as it crashed into the French Alps . All 150 on board were killed. Paris Match and Bild reported that the video was recovered from a phone at the wreckage site. …

Prosecutor : " So far no videos were used in the crash investigation "

marseille prosecutor brice robin told cnn that " so far no videos were used in the crash investigation . " robin \'s comments follow claims by two magazines , german daily bild and french paris match , of a cell phone video showing the harrowing final seconds from on board germanwings flight 9525 as it crashed into the french alps . paris match and bild reported that the video was recovered from a phone at the wreckage site .

Figure 1: Automatic text summarization approaches

1.2 Problem Statement

The main goal of this thesis research is to employ modern deep learning techniques to perform automated text summarization. In short, we wish to design an automated summarization system that will take voluminous text as input and produce a summary.

1.2.1 Problem Fomulation

Text summarization is a popular natural language processing task. The deep learning models used to perform this task are typically composed of two modules which are as follows:

1. **Encoder module**: This module is responsible for reading, understanding, and retaining context information from the input text. It carries out the *natural language understanding* portion of text summarization task.

2. **Decoder module**: The responsibility of this module is to generate the summary sequence, which is the *natural language generation* portion of summarization task.

These models are called *sequence-to-sequence* models, or *seq2seq* in short. Just as the name implies, these models can take variable length input and produce variable length output sequences.

1.2.2 Research Objectives

The objectives of our research are

1. Use deep learning to develop an abstractive text summarization system for the Bangla language.

2. Generate summaries that are grammatically accurate and factually corrent.

3. Explore algorithms to handle rare and unknown words while keeping a small vocabulary size.

4. Develop a system which is computationally feasible to build.

1.2.3 Contributions

This thesis provides several insights regarding Bangla text summarization. We highlight our main contributions here:

1. **Utilize transformer architecture**: After studying many related works in this field, we have found none are using Transformers. Most of the existing papers propose systems that leverage the Recurrent Neural Network architectures. However, transformers are currently at the center of modern NLP research. In this research, we explore how we can use transformers for Bangla text summarization.

2. **Handle rare and unknown words**: From our background studies, we found that most related works either use word or character level tokenization. We propose the use of subword tokenization to overcome the obstacles faced by these methods.

3. **Accumulate large text corpus**: Training deep learning models requires large amounts of data. For text summarization, our input data and output are respectively long and short sequences of text. We have opted to use news article and their corresponding titles as our dataset. We created a large dataset consisting of 600 thousand examples by scraping online news articles.

4. **Utilize the power of beam search**: Our deep learning model outputs word probabilities at each time step. A sentence can have multiple time steps. Thus, a summary can be one of the many possible sentences in our search space. To efficiently search this huge space we employ the use of beam search decoding. We discuss the effectiveness of beam search and compare it to the widely used greedy search approach.

2 Literature Review

2.1 Bengali abstractive text summarization using sequence to sequence RNNs

The authors proposed a summarization system that utilizes a bi-directional LSTM sequence-to-sequence model. To better handle context, they implemented the attention mechanism in their encoder and decoder modules. Their training text corpus is composed of online Bangla news articles and social media posts from Facebook. Text preprocessing, word embedding, unknown words are some of the challenges faced in this paper. [1]

2.1.1 Core Components

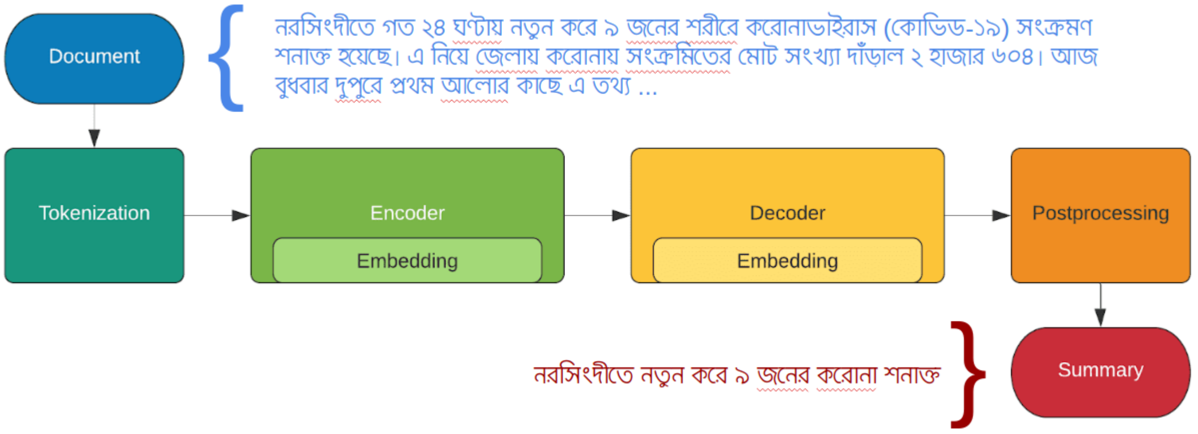
The pipeline introduced in this paper can be generalized into a system composed of 5 modules. The modules are as follows:

1. Tokenization module

2. Embedding module

3. Encoder module
4. Decoder module
5. Postprocessing module

This system is illustrated in figure 2.


নরসিংদীতে গত ২৪ ঘণ্টায় নতুন করে ৯ জনের শরীরে করোনাভাইরাস (কোভিড-১৯) সংক্রমণ শনাক্ত হয়েছে। এ নিয়ে জেলায় করোনায় সংক্রমিতের মোট সংখ্যা দাঁড়াল ২ হাজার ৬০৪। আজ বুধবার দুপুরে প্রথম আলোর কাছে এ তথ্য ...

নরসিংদীতে নতুন করে ৯ জনের করোনা শনাক্ত

Figure 2: Text Summarization Pipeline

2.2 Tokenization

According to *AnalyticsVidya*, *"Tokenization is a way of separating a piece of text into smaller units called tokens."* Tokens are the fundamental building blocks of natural language. They can be either a character, a sequence of characters or subwords, or entire words. The most common way of forming tokens is using space and punctuations as the delimiter.

2.2.1 Word level tokenization

This is the most commonly used among the three. In Bangla, we have many variations of a single word to denote different meanings. Word level tokenization will consider each of these as separate words, and they will be stored separately. As a result, our tokenizer will have a humongous vocabulary, which will be memory intensive. If we use a modest vocabulary size, we will run into the problem of unknown words. These are words that aren't recognized by the tokenizer as they are out-of-vocabulary words. The tokenizer replaces these words with <UNK> tokens. <UNK> tokens act like blanks. They negatively impact our deep learning model by making it harder for the model to understand the text.

2.2.2 Character level tokenization

This appraoch solves the problem of unknown words. The tokenizer breaks all the words in the text down to their constituent characters. In this case, unique characters make up the vocabulary, making it compact. However, the tokenized sequence becomes disastrously lengthy. Deep learning models lose their ability to retain context with increasing lengths. For text summarization, our input text will be long text sequences. Character level encoding will make these even longer.

2.2.3 Subword level tokenization

This is currently the preferred tokenization approach. We know that words are composed of meaningful subwords. Word prefix, base, and suffix parts are all subwords. For example, geology, geological, geography, and geologist have the same 'geo' subword base. We can alter its meaning of the base word by appending different suffix subwords. Subword encoding allows us to use a compact vocabulary, as it consists of unique subwords, without having to deal with the unknown words issue. The tradeoff is the generation of longer tokenized sequences.

2.2.4 Neural Machine Translation of Rare Words with Subword Units

We train language models on fixed-length vocabulary, which suffers from out-of-vocabulary words. Language processing is an attempt to solve this problem. It is the responsibility of the tokenizer to resolve this issue. Modern tokenizers use word segmentation to overcome this obstacle. Byte-Pair Encoding is an unsupervised subword segmentation algorithm. The intuition is that various word classes are translatable and understandable via smaller units than words. [2] The general procedure for generating subwords is detailed in algorithm 1.

Algorithm 1: Byte Pair Encoding

```
1 Start with an enormous text corpus.
2 Identify all unique symbols or characters present in the text corpus.
3 Perform symbol filtering to exclude unnecessary symbols.
4 Break each word into constituent symbols.
5 while required vocabulary size not reached do
6     Iteratively merge frequently occurring symbol pairs to be one symbol.
7 end
```

2.3 Embedding

According to Wikipedia, *"A word embedding is a learned representation for text where words having the same meaning have a similar representation."*

2.3.1 Distributional Hypothesis

A text corpus is not merely a bag of words evident in its distributional nature. Distribution takes many forms.

- **Dispersion**: Words in a corpus tend to concentrate in particular contexts and be more diffused in others.
- **Collocation**: They tend to co-occur with other specific words.
- **Colligation**: Words frequently appear in specific grammatical contexts.

For the above reasons, corpus linguistics is mainly a distributional science. *Distributional Hypothesis*, an obvious inspiration for corpus linguistics, states that lexemes with a similar distribution have similar meanings.

2.3.2 Word Vectors

Word embeddings are the computational implementation of the *Distributional Hypothesis* theory. A vector of numbers represents each word in the vocabulary. Homogenous words will appear close in the vector space. Cosine similarity is a popular measure of similarity used in word embeddings.

Figure 3 displays a handful of words in a three dimensional vector space. Let us assume each of the three axes to align with the meaning of sky, engine, and wings. We can see that the words helicoptor, drone, and rocket appear together in the vector space. All three words have the notion of sky and engine. The same can be said for the words goose, eagle, and bee. They are related to the notion of sky and wings. It is noticable that all six words are related to the notion of sky and appear towards that axis. But the words in green don't carry the notion of engine and appear on its negative axis. Similarly, the blue words don't carry the notion of wings and appear on its negative side.

2.4 Attention Is All You Need

Before 2017, the dominant sequence-to-sequence models used complex recurrent and convolutional neural networks. The best of these models connected the encoder and decoder modules using an attention mechanism. Vaswani proposed a simple architecture, dubbed as Transformer, that used only attention mechanism and removed the recurrent neural network layers. They published their findings in the paper, appropriately named, *"Attention is all you need"*.[3] All subsequent breakthroughs in modern NLP, including *BERT*[4], *RoBERTa*[5], *AlBERT*[6], builds on top of this transformer architecture.

2.4.1 Architecture Overview

The Transformer model is a sequence-to-sequence model composed of an encoder and a decoder module. Figure 4 highlights the main parts of the transformer architecture.

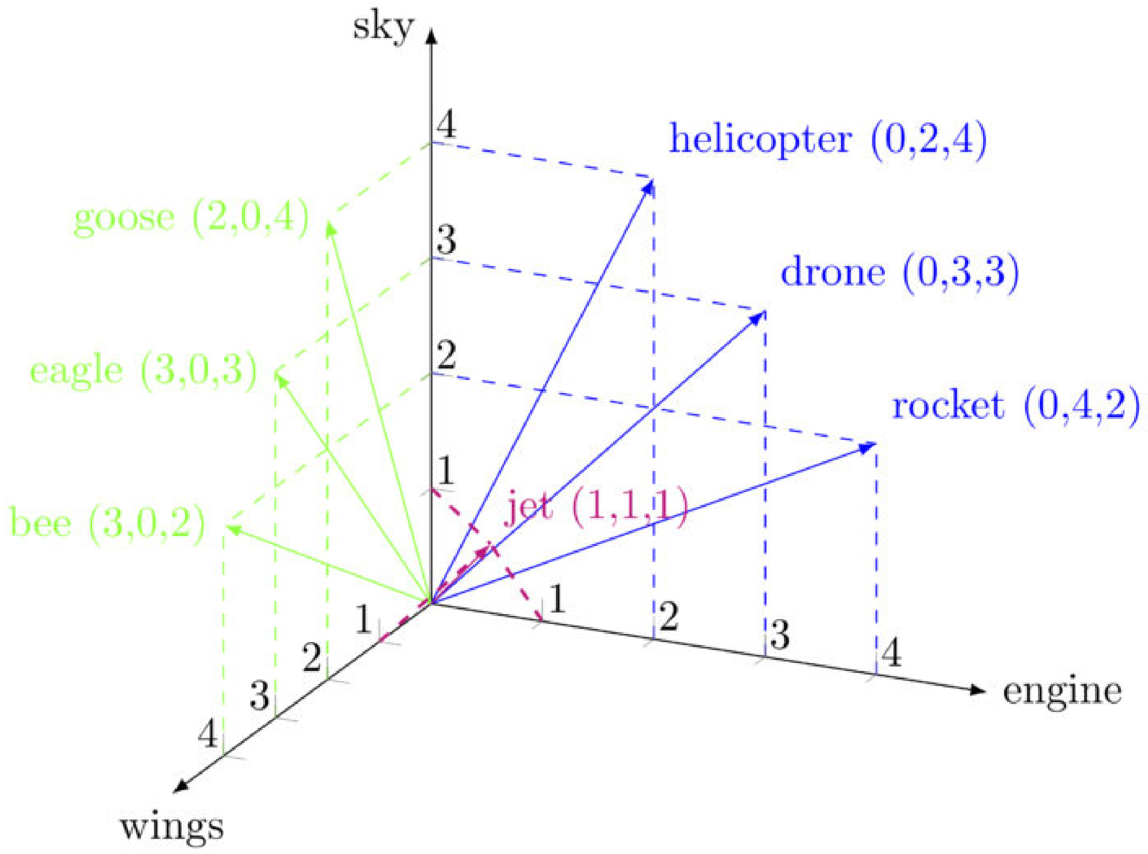


Figure 3: A simple three dimensional vector space

- **Positional encoding**: Unlike Recurrent Neural Networks, this model does not operate sequentially. As a result, we lose the position of tokens respective to each other. To overcome this the authors added an extra feature to each embedding. This feature is positional encoding information. We can learn this feature or generate it using a mathematical function. The authors chose the latter option. They generated positional information using equation 1 for even and 2 for odd positions.

$$\mathtt{PE}_{(pos,2i)} = sin(\frac{pos}{10000^{\frac{2i}{d_{\mathtt{model}}}}}) \tag{1}$$

$$\mathtt{PE}_{(pos,2i+1)} = cos(\frac{pos}{10000^{\frac{2i}{d_{\mathtt{model}}}}}) \tag{2}$$

- **Encoder**: The encoder module is comprised of 2 types of layers, which are as follows:

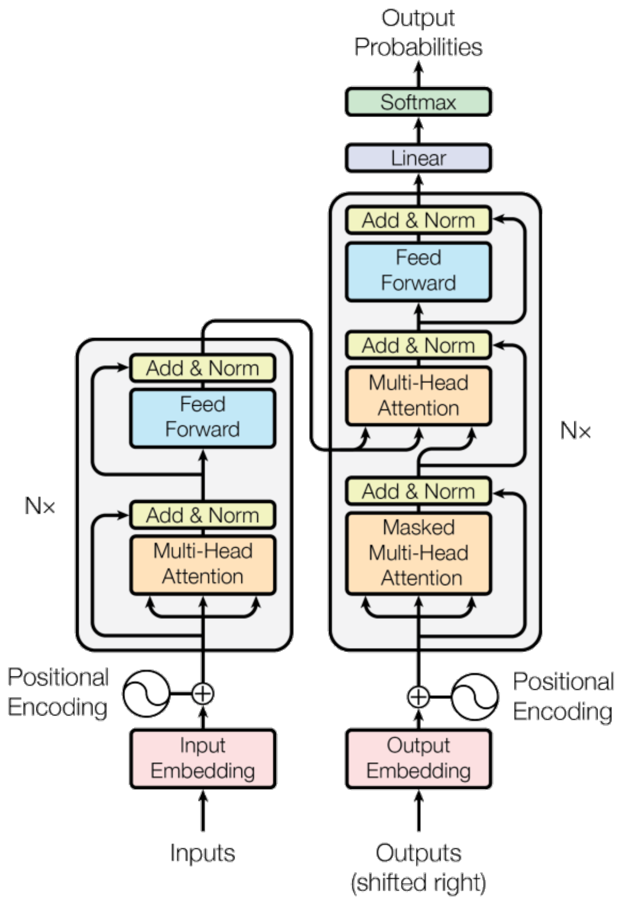


Figure 4: Transformer Architecture

1. Multihead Attention Layer
2. Fully connected feed forward layer

The authors implement residual connections within the layers to tackle the effects of vanishing gradient. Moreover, to prevent overfitting the architecture leverages drop out regularization technique.[7] The main purpose of the encoder module remains natural language understanding, just like the traditional RNN based seq2seq models.

- **Decoder**: The decoder module is comprised of three types of layers as mentioned below:

  1. Multihead Masked Attention Layer: Attends over input sentence to generate output word.
  2. Multihead Attention Layer: Attends over the output sequence generated thus far.
  3. Fully connected feed forward layer

Similar to the encoder, residual connections and drop outs are used in this module as well.

- **Stacking modules**: The "**Nx**" term present next to the encoder and decoder signifies the presence of multiple exact instances. In the original paper, the authors used 6 layers stacked on top of one another.

2.4.2 Attention

In simple terms, attention can be broadly interpreted as a vector of importance weights. We use this vector to find correlation with other elements. Using this correlation we try to approximate the next word in a generated sequence. Consider the sentence, *"May is walking by the bank of the river"*. If we are asked *"Where is May"*, we will look at the sentence and focus or attend to the words *"bank"* and *"river"*. This is an example of attention. To generate our response we attend over the input sentence. The authors use *self attention* or more specifically *dot product attention* mechanism. The mechanism is shown in figure 5. Multihead attention performs multiple instances of this computation in parallel.

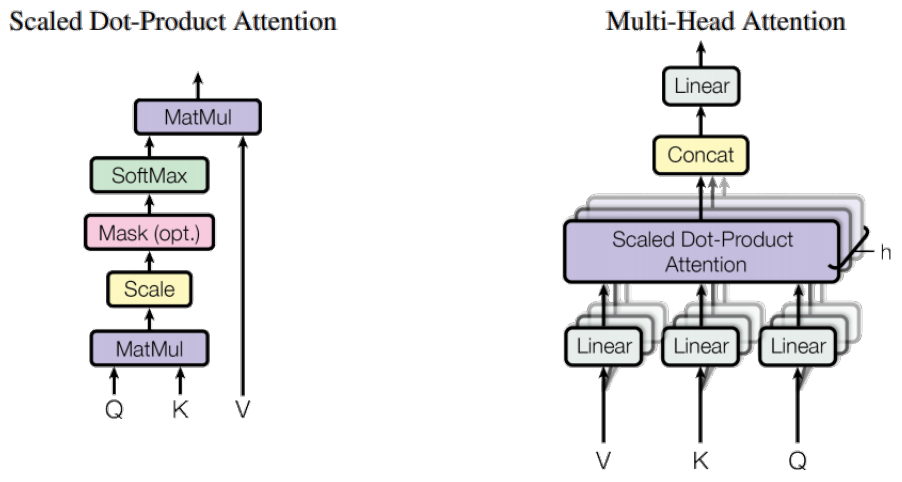


Figure 5: Self attention and Multihead attention

2.5 Search Strategies for Neural Language Generation

Language models output probability distribution of words that are to appear in a specific time stamp of the generated sequence. The generated sequence can have multiple such timestamps. There are 3 main strategies that we can adopt when generating the sequence.

2.5.1 Greedy search

Pick the word with the highest probability to be outputted in each timestep. This is the simplest and most efficient decoding approach. However, once an improper word is generated at a timestep there is no way to undo it. For example, let us imagine we have the label sequence *"I have rice"*. At timestep 1, our model generates *"I"* from the *"<start>"*. After that it generates *"am"* from *"<start> I"*. This is the wrong generated text, but there is no way to undo this generation. Next, it generates *"rice"* from *"<start> I am"* and ends up with the sequence *"I am rice"*. This example illustrates the problems of greedy decoding.

2.5.2 Exhaustive search

In this case, we construct all posible sequences and then output the best sequence. Needless to say, this search pattern would give us the optimal solution but is infeasible to implement and deploy.

2.5.3 Beam search

This search strategy combines the best features of the above two approaches. In this decoding technique, we maintain k of the best sequences seen so far. Each of these k sequences are fed into the model to get more generated sequences in the next timestamp. Among those timestamp the top k sequences are kept for the next timestamp. For example, we might get *"I am rice"* and *"I have rice"* as 2 sequences. The former sequence would be discarded and the later would be kept. At the end of the sequence generation process the best sequence among the k sequences is returned.

2.6 Evaluation

Text summarization is a Natural Language Processing task. To evaluate how well out system is performing, we need evaluation metrics. There are two standard procedures to follow when evaluating summaries. These are as follows:

1. **Automatic Evaluation**: The standard for evaluating summaries automatically is *ROUGE* or *Recall-Oriented Understudy for Gisting Evaluation.*

2. **Human Evaluation**: The gold standard for evaluating summaries is through humans.

2.6.1 ROUGE Metrics

ROUGE is a set of metrics for evaluating automatic summarization systems proposed by Lin et. al. [8] There are many varients of ROUGE, but the 3 most commons ones used are ROUGE-1, ROUGE-2, ROUGE-L.

- **ROUGE-1**: ROUGE-1 measures the number of unigram overlaps between the system generated summary and reference summary.

- **ROUGE-2**: ROUGE-2 measures the number of bigrams overlaps.

- **ROUGE-L**: ROUGE-L measure the longest common subsequence length between the system generated and reference summary.

Each of these metrics return three values which are as follows:

1. **Recall**: The ratio between the overlap count and reference summary length. The higher the recall value the more amount of information from the reference summary is present in the computer generated summary.
2. **Precision**: The ratio between the overlap count and system generated summary length. Precision measures conciseness of the generated summary.
3. **F1**: This is a unified value derived from both recall and precision. It measures how concisely the system summary captured elements from the reference summary. The formular for calculating the F1 score is given in 3

$$\texttt{f1} = \frac{2 * \texttt{precision} * \texttt{recall}}{\texttt{precision} + \texttt{recall}} \tag{3}$$

2.6.2 Human Evaluation

The gold standard of NLP evaluation is human evaluation. In summarization, the system's performance is measure by getting human evaluators to assign scores to the generated summaries. There is no standard procedure that is followed everywhere. However, the evaluation system follows a set of guidelines. These are as follows:

1. The evaluators mustn't know which is the computer generated summary and which is the reference summary as it might introduce bias.
2. Evaluation should be taken from a large group of people from different backgrounds.
3. Evaluators should have minimum literacy competency to perform the evaluation.

3 Proposed Methodology

To perform Bangla text summarization, we have the following proposals

1. **Accumulate large text corpus**: Training deep learning models require volumnous amounts of data. Our Transformer network will easily overfit small training datasets. As a result, the generalization performance of the model will be quite horrible. To train a capable model, we need to acquire large amounts of data for summarization. We propose text mining news articles from online Bangla news portals as a way of collecting this data. We build an efficient text mining tool using Python to perform this task. We discuss the data accumulation procedure in subsection 4.1

2. **Utilize Transformer Archtecture**: Transformer is at the forefront of modern NLP research. We propose the use of this revolutionary architecture to perform text summarization instead of the dated RNN based Sequence to sequence networks. The details of our deep learning system is discussed in subsection 4.3

3. **Encorporate Subword tokenization**: Word tokenization is plagued with many issues including large vocabulary size and unknown words. We propose the use of subword tokenization to keep a small vocabulary and to handle the issue of unknown words. To perform our tokenization we use Byte Pair encoding algorithm and use the word piece encoding approach.

4 Experimentation

4.1 Dataset Accumulation

Deep learning models are data-hungry. Vast amounts of data are required to train these models. The transformer architecture is no exception.

4.1.1 Limitations of Existing Datasets

After searching the internet, we found no single large dataset that fulfills our purpose. Most datasets for Bangla text summarization are relatively small, containing at best 100 thousand examples. Powerful deep learning models easily overfit small datasets. As a result, they perform poorly on unseen data. Only by training these models on large amounts of data can we uncover their true potential. To satisfy our data requirements, we started compiling a large dataset. We chose online Bangla news articles as a source of data.

4.1.2 Text Mining Procedure

To gather this massive amount of news data, we built several scrappers that could fetch and format data from different Bangla news portals. We used the following tools:

- **Python**: General-purpose programming language
- **Selenium**: A library for scraping dynamic websites
- **Requests**: A library to handle HTTP requests and responses

We mainly employed the following approaches to developing our scraper:

1. **Initial approach - Scraping text after loading website**: We used *Python* and *Selenium* to build a traditional scraper. It went through the website, parsed the HTML data, and saved it in a CSV format. The system was slow as it had to load and process each page. It could scrap around 800 articles an hour. To gather enormous amounts of data within a brief timeframe, we required a faster solution.

2. **Improved approach - Reverse engineering news API**: Instead of parsing the HTML, we reverse-engineered the news portal's API endpoints and sent direct requests to those APIs using the *Requests* library. In this way, we could get raw JSON data, for which parsing was very fast. Using this new system, we could scrape *500 articles every second* or *30,000 per hour.* We used this improved version for further data collection.

4.1.3 Dataset Structure

Our text summarization model requires the following sequences of texts for training:

1. **Input sequence**: A long piece of text. We decided to use news articles for this purpose.
2. **Output sequence**: A short piece of text. We decided to use the title of news articles as our reference summary.

We list some of the subtle yet significant details of our data set here:

- **Source**: We chose the *"Prothom Alo"* news website as our data source. This reputable news publisher maintains high editorial quality. Thus, we can be sure that the mined text data will be of high quality.
- **Article Time Distribution**: We scraped news articles from 2010 to 2020. Figure 6(a) shows number of articles belonging a particular year.
- **Dataset Size**: In total, we have accumulated above 600 thousand examples. Listing 1 shows exact details of the dataset.
- **Example Tuple**: Alongside the article content and title, we also saved the article tags for further data processing needs.
- **Article Variation**: The news articles span almost all popular categories, including but not limited to politics, sports, fashion, technology, education. A histogram of the top 10 article tags are shown in figure 6(b).
- **Text Length**: On average the articles had a title and content spanning respectively 31 and 650 characters. The detailed quantile information is depicted in the two boxplots in figure 6 (c) and (d)
- **Storing Format**: The dataset is a collection of CSV files where each file has a name of the format year_month.csv. We chose this naming convention to group articles by their publication month and year.

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 603167 entries, 0 to 4220
Data columns (total 2 columns):
 #   Column   Non-Null Count   Dtype
---  ------   --------------   -----
```

```
 0   title     602597 non-null   object
 1   content   603121 non-null   object
dtypes: object(2)
memory usage: 13.8+ MB
```

Listing 1: Raw Dataset Details

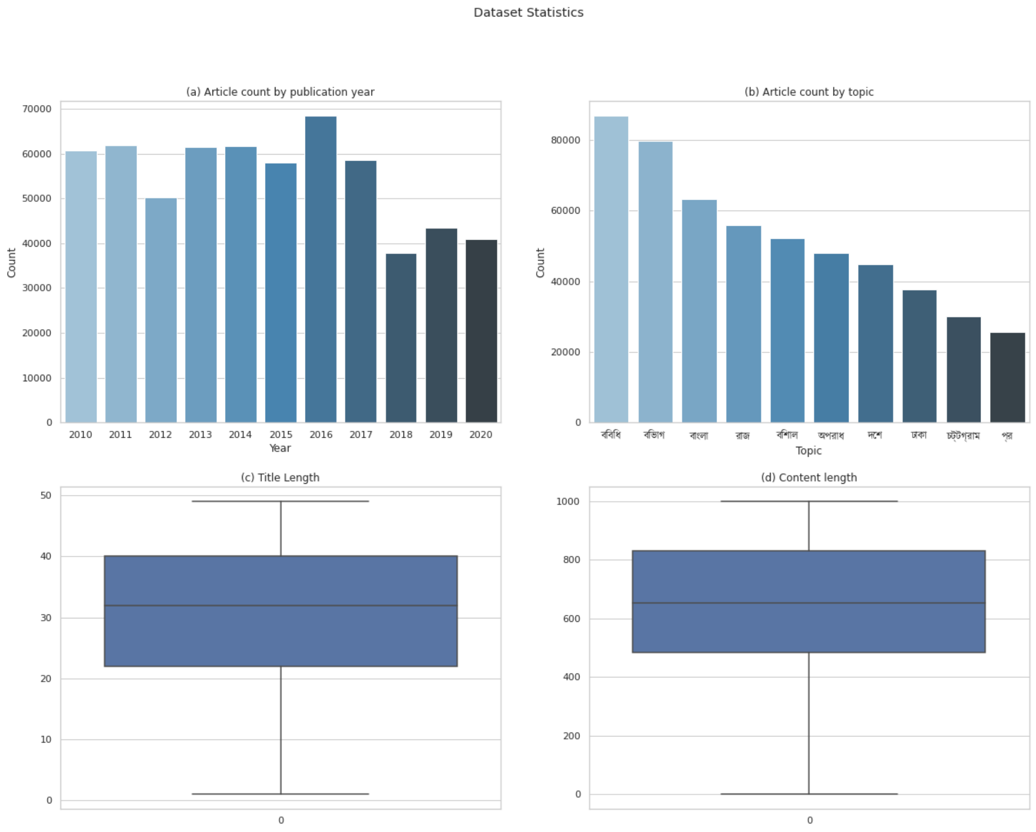


Figure 6: Dataset Statistics

4.2 Data Processing Pipeline

Before we can feed out accumulated data into our model, we first need to process the data and make it usable. To perform this task, we created a data processing pipeline. The steps of the pipeline is described in algorithm 2.

Algorithm 2: Data Processing Pipeline

```
1 for each csv file do
2     Read content from file.
3     Perform data cleaning operations.
4     Perform subword tokenization.
5     Pad and truncate token sequences.
6     Split dataset into train and test parts.
7     Generate TFRecords from processed examples.
8 end
9 Optimize fetching and retrieval operations.
```

4.2.1 Data Cleaning

Data cleaning is the first step of our processing pipeline. We perform the activities mentioned below to clean our dataset:

1. **Removing instances with NaN attributes**: Our scrapper sometimes failed to scrap either the news content or the title or both. As a result, some examples in the dataset were unusable for training our model. Thus, we removed examples where both the article content and title weren't present, which gave us the dataset shown in Listing 2.

2. **Removing string artifacts**: Sometimes we got unnecessary details with our desired article contents and titles. These details included names and email addresses of the journalists who wrote the article, links, ad links, image links, and much more. These artifacts would be detrimental to our model training process. Hence, we removed these artifacts from our dataset.

3. **Replacing whitespaces**: From observing the various instances, we noticed that the strings have unnecessary white spaces. These white spaces came in the form of multiple whitespaces, newlines, tabs. We replaced each of these with a single whitespace.

4. **Removing non-Bangla text**: We found some string instances to have characters from other languages, most frequently English. We deleted these characters from the strings.

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 602551 entries, 0 to 4220
Data columns (total 2 columns):
 #   Column   Non-Null Count   Dtype
---  ------   --------------   -----
 0   title    602551 non-null  object
 1   content  602551 non-null  object
dtypes: object(2)
memory usage: 13.8+ MB
```

Listing 2: NaN removed dataset info

4.2.2 Subword Tokenization

The next step in our data processing pipeline is the tokenization process. As discussed in the literature review section, we chose subword level tokenization. To perform this, we selected the Byte Pair Encoding algorithm. We opted to use a pre-trained tokenizer created by Benjamin Heinzerling and Michael Strube in 2018.[9] Pre-trained tokenizers for over 275 different languages, including Bangla, are available. We chose a tokenizer with a vocabulary size of 10,000 and an embedding size of 100. We refrained from using large vocabulary sized and embedding sized tokenizers due to computational requirement.

There is one problem with this tokenizer. It doesn't recognize digits. All digits in Bangla are replaced by the symbol '0'. As a result, we lose any relevant numbers in the input document during this process, which is one of the substantial drawbacks of our text summarization system. We theorize that the usage of word piece tokenizer will solve this problem.

The actual process of tokenization is quite simple. We used the open-sourced *"BPEmb"* package to perform the encoding process. Both our input and output texts are encoded. However, there is one subtlety. We add two tokens to our output text after the encoding process, one at the beginning and the end. These two tokens signify the beginning and end of the sequence. We initially feed our deep learning model with the start token, which signals to the model that it should start the sequence generation process. Once the model has generated the end token, we stop the process and output the generated sequence as the summary. We talk more about this process in the deep learning model subsection.

4.2.3 Padding and Truncating

In this step, we make sure all input sequences have a uniform length. We perform the following operations for this purpose:

1. **Truncating operation**: We truncate sequences to have a maximum of MAX_LEN tokens. Excess tokens are discarded and not fed to the model as input.

2. **Padding operation**: We pad with 0s when input sequences have less than MAX_LEN tokens.

We considered the following constraints when choosing the optimal value for MAX_LEN:

1. ***MAX_ LEN* should be an exponent of 2**: Digital electronics used in computers have two states: on and off. So memory is a collection of elements that can either be on or off. Hence, in computers, everything naturally is organized in the powers of two. When inputs have this length, the storage, processing, and retrieval operations become efficient and fast.

2. **Truncating and padding should be the bare minimum**: Truncating discards parts of the input sequence. If MAX_LEN is too small, we risk losing a large portion of our input. As a result, our deep learning model might fail to capture the underlying meaning. Similarly, if we pad too many zeros to a short input article, our model might under-perform. To choose the appropriate value for MAX_LEN, we looked at the title and content length quantiles. The quantile information is displayed in Figure 7 and in table 1. The optimal number is the one that is closest to the 75% quantile value.

Upon considering these conditions, we chose MAX_LEN to be 512 for our input content and 16 for our output summary. We perform the actual truncating and padding operation using TensorFlow's Sequence API. We truncated tokens from and padded zeros to the back of our sequences.

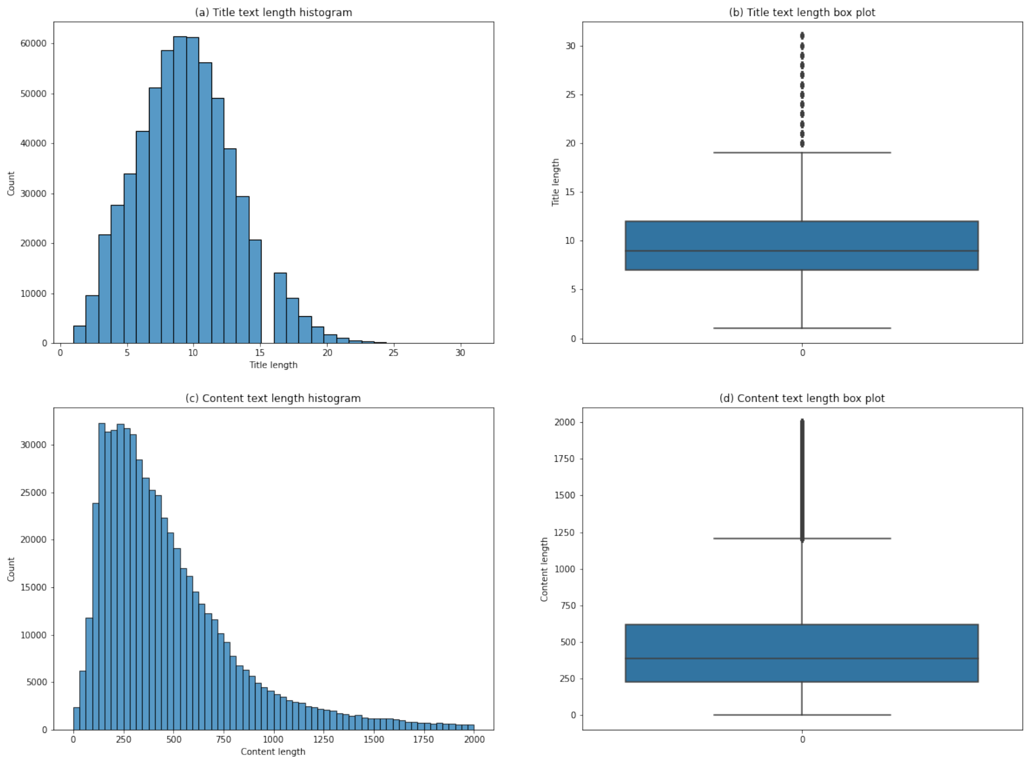


Figure 7: Text Length Statistics

4.2.4 Train and Test Splitting

We need to prepare the dataset for training our deep learning model. Towards this end, we need to create two subsets of our dataset, which are

- **Training set**: We will train our deep learning model on this subset. It will contain the majority of the data.
- **Testing set**: We will use this subset to determine the performance of our model on unseen data.

Quantile	Content Length	Title Length
min	1	3
25%	169.0	7.0
mean	369.0	8.7
75%	464.0	11.0
max	14739.0	651.0

Table 1: Tokenized article content and title length quantile

We perform the splitting operation on each CSV file. As mentioned before, we stored articles published in the same month and year in the same file. Performing splitting at the file level ensure that the training and testing set will have the same ratio of files published at different times. We keep *90%* of the records for training the deep learning model and the rest for testing. We can see in listing 3 that the training and testing set consists of 520 and 60 thousand examples respectively.

```
...
Processing: dataset/csvs/2020_jun.csv       Train: 4534     Test: 504
Processing: dataset/csvs/2020_mar.csv       Train: 2356     Test: 262
Processing: dataset/csvs/2020_may.csv       Train: 3967     Test: 441
Processing: dataset/csvs/2020_nov.csv       Train: 1086     Test: 121
Processing: dataset/csvs/2020_oct.csv       Train: 3861     Test: 430
Processing: dataset/csvs/2020_sep.csv       Train: 3754     Test: 418
Training set size: 523135
Testing set size: 58191
FINISHED in 917.0775
```

Listing 3: Train test splitting log

4.2.5 Generating TFRecords

To train our deep learning model, we can read records from the CSV files, split them, and then perform the processing steps. Then we can directly feed them into the model. But this is an inefficient process because we would have to process the data repeatedly before feeding it into the model. Making an already lengthy procedure much longer. The standard norm is to convert the dataset into *TFRecords* when using the *TensorFlow* framework. The *TFRecord* format is a simple format for storing a sequence of binary records for efficient serialization of structured data. We perform this operation only once. We can directly feed the generated *TFRecords* into the model without needing to perform any further processing. As a result, we don't have repeatedly waste precious time in processing the data.

4.2.6 Minimizing IO Bottleneck

Finally, we need to minimize the IO bottleneck during training time, allowing overall training efficiency to be as high as possible. TensorFlow's *Data API* provides the following functionalities to make this possible:

1. The **cache** transformation can store a dataset, either in memory or on local storage. It will save some operations (like file opening and data reading) from being executed during each epoch. Transformations, like the file opening and data reading, are performed only during the first epoch. The next ones will reuse the data cached by the cache transformation. The data execution time plot in figure 8 depicts this process.

2. **Prefetching** overlaps the preprocessing and model execution of a training step. While the model in training step s, the input pipeline is reading the data for step s+1. Doing so reduces the step time to the maximum (as opposed to the sum) of the training and the time it takes to extract the data. The data execution time plot for this process is shown inn figure 8.

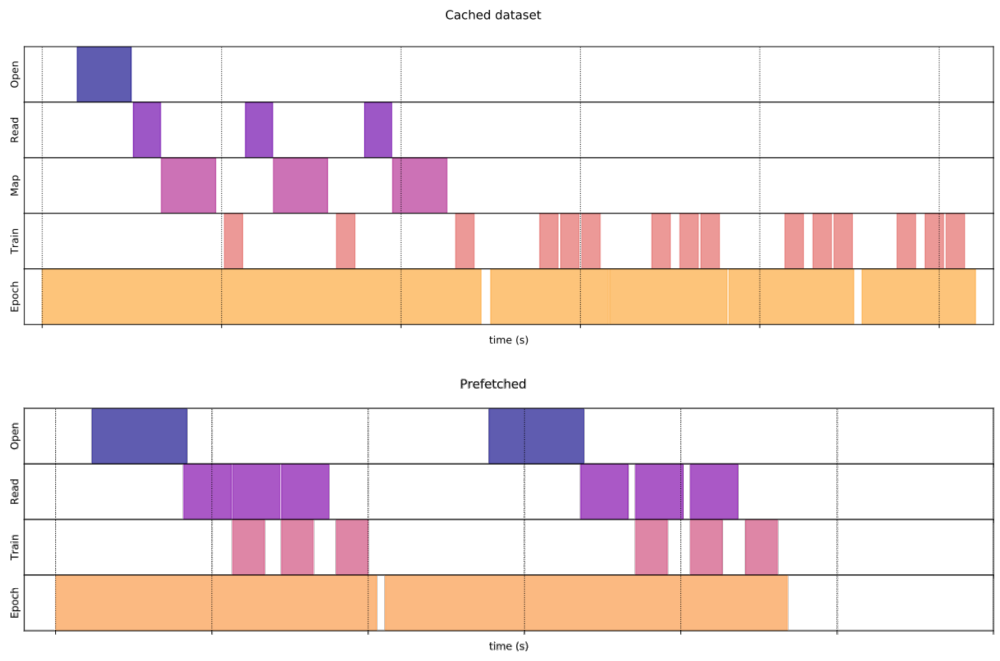


Figure 8: Data execution time plot for cache and prefetch operation

4.3 Model Training

4.3.1 Model Architecture

We're utilizing the Transformer architecture as our deep learning model. We applied the vanilla variant of this architecture with a few minor alterations. Transformers, like most advanced deep learning models, are computationally demanding. They require powerful hardware to run. Unfortunately, we didn't have access to this level of computing resources. So bearing the constraints in mind, we chose to use a smaller version of the vanilla architecture. Table 2 details the parameter values used for our model.

Hyperparameter Name	**Code Repository Name**	**Value**
Number of layers	NUM_LAYERS	4
Embedding Size	D_Model	128
Fully Connected Layer Nodes	DFF	2048
Encoder sequence size	TEXT_LENGTH	512
Decoder sequence size	SUMMARY_LENGTH	16

Table 2: Selecter parameter values for small Transformer model

4.3.2 Loss Function

The transformer model outputs an array of probabilities at each time step t. We chose *Sparse Categorical Cross-Entropy* as our loss function to measure the deviation at each time step. We calculate the loss using equation 4

$$J_{\texttt{cross-entropy}}(w) = -\frac{1}{N}\sum_{i=1}^{N}[y_i log(\hat{y_i}) + (1-y_i) log(1-\hat{y_i})] \tag{4}$$

But there will many such time steps in the summary generation process. We consider the mean of cross-entropy losses over all the time steps as the final loss. We calculate the mean using equation 5

$$J_{mean}(w) = \frac{1}{T}\sum_{t=1}^{T} J_{\texttt{t-th cross-entropy}}(W) \tag{5}$$

Here,

1. w refers to the parameters of the model
2. y_i is the true label
3. $\hat{y_i}$ is the predicted label
4. N is the number of instances
5. T is the total number of time steps/tokens in the generated text

4.3.3 Performance Metrics

We use **Sparse Categorial Accuracy** to measure the model's performance during the training phase. This metric emphasizes the correctness of the predictions at each time step. The model achieves higher accuracy by generating the right tokens at the right time step.

4.3.4 Optimizer

We used Adam's optimizer with custom learning rate scheduling.[10] We perform the scheduling mechanism using the equation mentioned in equation 6. Figure 9 displays the learning rate changes with step number increasing.

$$\texttt{learning\_rate} = d_{model}^{-0.5} * min(\texttt{step\_num}^{-0.5}, \texttt{step\_num} * \texttt{warmup\_steps}^{-1.5}) \tag{6}$$

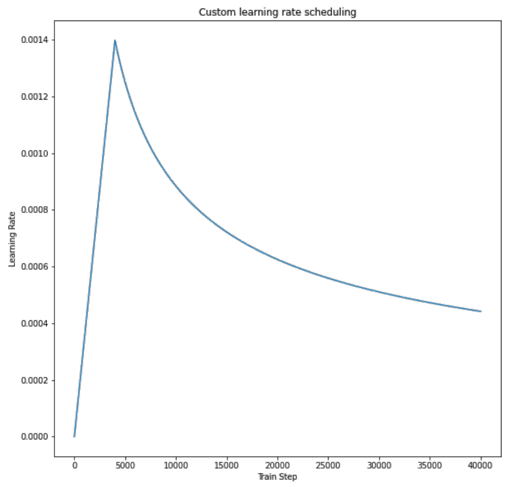


Figure 9: Custom learning rate scheduling

Adaptive Moment Estimation or *Adam*, is an optimization algorithm that uses exponentially weighted averages to perform gradient updates. It combines the best of both momentum and RMSprop and generally works well for a wide variety of deep learning tasks. Adam updates the gradient values by using two moments called the first and second moment. The general update procedure of adam is shown in algorithm listing 3 and the selected hyperparameter values are shown in table 3

Hyperparameters	Learning Rate (α)	β_1	β_2	ϵ
Values	Custom Rate	0.9	0.99	$1e^{-8}$

Table 3: Selected hyperparameter values for Adam optmizer

Algorithm 3: Adam Optimizer

```
   // Initialize
1  $v_{dw} = v_{db} = 0$;
2  $s_{dw} = s_{db} = 0$;
3  for $t \leftarrow 0$ to $T$ do
       // Mementum like update
4      $v_{dw} = \beta_1 * v_{dw} + (1 - \beta_1) * \mathtt{dw}$, $v_{db} = \beta_1 * v_{db} + (1 - \beta_1) * \mathtt{db}$;
       // RMSProp like update
5      $s_{dw} = \beta_2 * s_{dw} + (1 - \beta_2) * \mathtt{dw}^2$, $s_{db} = \beta_2 * s_{db} + (1 - \beta_2) * \mathtt{db}^2$;
       // Bias correction
6      $v_{dw}^{\mathtt{corrected}} = v_{dw}/(1 - \beta_1^t)$, $v_{db}^{\mathtt{corrected}} = v_{db}/(1 - \beta_1^t)$;
7      $s_{dw}^{\mathtt{corrected}} = s_{dw}/(1 - \beta_2^t)$, $s_{db}^{\mathtt{corrected}} = s_{db}/(1 - \beta_2^t)$;
       // Parameter update rule
8      $W = W - \alpha * v_{dw}^{\mathtt{corrected}}/(\sqrt{s_{dw}^{\mathtt{corrected}}} + \epsilon)$, $b = b - \alpha * v_{db}^{\mathtt{corrected}}/(\sqrt{s_{db}^{\mathtt{corrected}}} + \epsilon)$;
9  end
```

4.3.5 Checkpoint Management

Training deep learning models requires enormous amounts of time. Complications can arise out of the blue and halt the training process. Systems can crash due to unforeseeable reasons. These will be quite devastating for us if we need to start the training from scratch every time. To avoid such a catastrophe, we chose to use the checkpoint saving mechanism available in the TensorFlow framework. We perform the saving process at the end of each epoch using a custom callback.

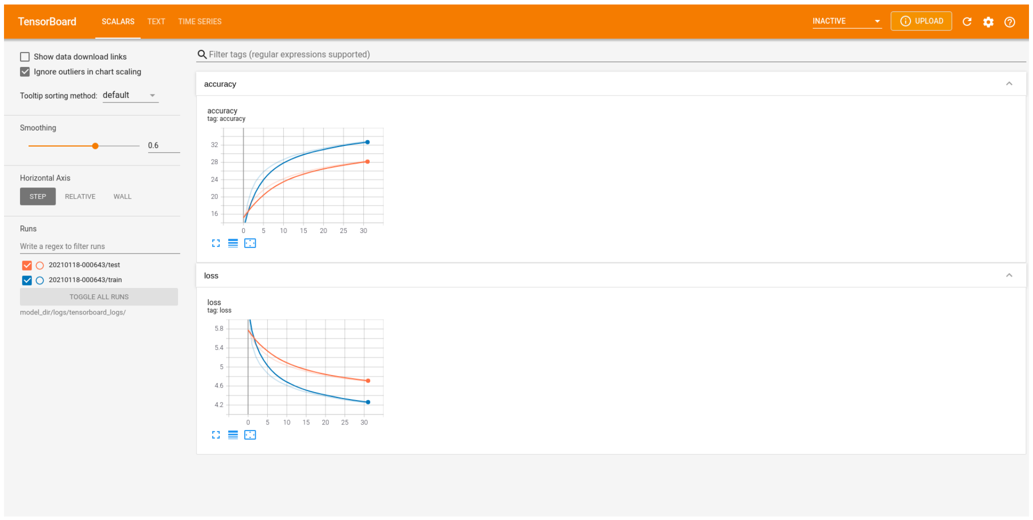
4.3.6 Training Process Monitoring

Monitoring loss and evaluation metrics are critical when training neural networks. By observing them, we can identify bugs in our code and anomalies in the training process. TensorBoard provides the visualization and tooling needed for machine learning experimentation. We used it to visualize the loss and metrics during the training phase. A snapshot of tensorboard is shown in figure 10.

4.3.7 Preventing Overfitting

Modern deep learning architectures excel at recognizing patterns in data. If we train them long enough, they'll start to uncover patterns from the noise. We say that the model has overfitted the dataset, and thus it won't generalize well to unseen data. An overfitted model will have significantly higher training accuracy compared to its test accuracy. We avoid this issue by stopping encorporating the following strategies:

1. **Early Stopping**: A problem with training neural networks is deciding the appropriate number of epochs to train our model. We risk overfitting the training dataset if it is too large. Conversely, it



☐ Show data download links
☑ Ignore outliers in chart scaling
Tooltip sorting method: default
Smoothing: 0.6
Horizontal Axis: [✓ STEP] RELATIVE WALL
Runs
☑ 20210118-000643/test
☑ 20210118-000643/train
model_dir/logs/tensorboard_logs/

Figure 10: A snapshot of tensorboard

may underfit if not long enough. Early stopping is a method that allows us to specify an arbitrarily large number of training epochs and stop training once the model performance stops improving on a hold-out validation dataset. TensorFlow provides a callback class to perform this task.[11]

2. **Dropout**: Dropout is a regularization method that approximates training multiple neural networks with varying architectures in parallel. During training, some number of layer outputs are randomly ignored or dropped out. By dropping a unit out, we mean temporarily removing it from the network, along with all its incoming and outgoing connections. Dropout has the effect of making the training process noisy, forcing nodes within a layer to take on responsibility for the inputs. It simulates a sparse activation from a given layer, which encourages the network to learn a sparse representation as a side-effect. The result is a substantial reduction in overfitting.[7]

5 Result Analysis

5.1 Loss and Accuracy

We trained our transformer model for 60 epochs. The loss and accuracy metrics are shown in figure 11.

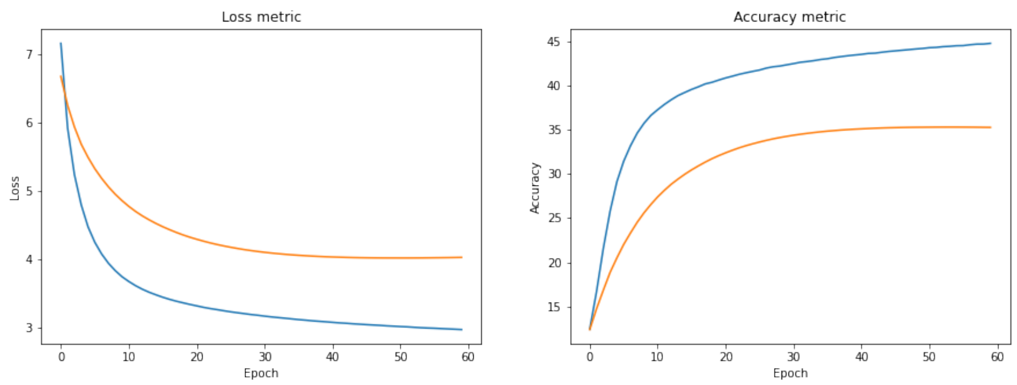


Figure 11: Loss and accuracy graph for summarizer model

From metrics figure we can see that our peak accuracy was around 46%. This might seem low, but there are several limitations of the automatic evaluation metric we used. First of all, even if a sentence is written differently but conveys the same meaning, it will be assigned a high loss and low accuracy.

5.2 Summary Generation

To really understand how well our model performs, we need to analyze the summary generation performance. To perform this we fed the model a constant article text at the end of each article and asked the model to generate a summary. Five of the most important results are shown in Figure 12.

Epoch 01: নিহত নিহত নিহত নিহত নিহত নিহত নিহত নিহত
Epoch 10: সড়ক দুর্ঘটনায় নিহত
Epoch 25: ট্রাকের ধাক্কায় বাঘ নিহত
Epoch 50: বাঘায় ট্রাকের ধাক্কায় সাইকেল নিহত
Epoch 60: যশোরে ট্রাকের ধাক্কায় সাইকেল আরোহী নিহত

Figure 12: Generated summaries

We can see that the model is indeed learning as the epochs go up. In the first epoch the model only generated the same word repeatedly. By epoch 10, it learnt that the input article was conveying information about a road accident. At the 25th epoch, the model learnt the cause of the accident. At epoch 50 it learnt location information and tried to fit that into the summary. Finally, by epoch 60, the model learnt where the accident happended to whom.

5.3 Challenges and Future Work

5.3.1 Complex model

The biggest room for improvement can be attained by using a larger and more complex version of transformer. We used the small varient of transformer to perform summarization. The two main reasons our small transformer model under performed are as follows:

1. **Large vocabulary**: Compared to the complexity of our model, our vocabulary size was much larger. As a result, our model couldn't effectively capture the complexity of the large set of words from the text corpus. Current state of the art models like ProphetNet [12] and Pegasus [13] are huge models with hundreds of millions of parameters. Although these models have state of the art performance, they require immense computing power to train and deploy.

2. **Long input sequence**: Typically, large varients of transformers are needed to effectively work with long input sequences. For example, BERT-large, a 768 million parameter model, is used to capture context from a sequence containing 512 tokens. We tried to perform this task with a much smaller model and failed as a result. Recent papers like Big Bird [14] and SMITH [15] try to tackle the problem of long input sequences.

5.3.2 Shorter Input Length

Simple models perform admirably on short input sequences. However, our dataset mostly contains long news content texts. Most of the news articles have a length near or exceeding 512 after tokenization. This means that we are computationally limited by the amount of context we can capture and retain. Hence, a better approach would be to target short news articles to train our simple model. This way its within the reach of the model to effectively capture and retain contextual information from the input.

5.3.3 Human Supervision

Even though our dataset is large, it isn't the best in terms of quality. We tried to clean our dataset using automated methods. However, the quality of the writing hasn't been judged by human experts. This is an important step. We need human supervision to create a quality dataset. The better the quality of our dataset the better our model will perform at its task.

5.3.4 Fine Tuning

Fine tuning approaching have become very successful in recent times. It is possible to get checkpoints of large models that were trained on a huge text corpus. These models can be fine tuned to our text summarization task. This will result in much greater performance than training from scratch because

1. These models were trained on billions of examples by a large organization like Google, Microsoft, etc.
2. The models capture contextual representation extremely well because they were trained on large datasets. As a result, they outperform models after being fine tuned on an end task.

5.3.5 Dataset Debiasing

The biggest and hardest problem to solve is dataset bias. Our dataset is heavily biased. Let us first describe bias in terms of news articles. Most of the news in the year 2020 is about the covid-19 pandemic. As a result, the model related almost everything to covid-19 and tries to generate a summary mentioning this particular topic. To train an unbiased model, the dataset needs to avoid containing such skewness. This is hard to avoid and organizations like Google and Microsoft tackle the same problem when training their models. One of the ways to minimize this issue is by hiring human supervisors to monitor the distribution of articles going into the dataset.

5.3.6 Better Tokenization Approach

As mentioned earlier, byte pair encoding has some major flaws. One of them is replacing digits with the token 0. This means we lose numbers from our input data. This is catatrophic because we can't associate quantities into our model. We can use other tokenization schemes, like wordpiece tokenizer, to overcome this drawback.

6 Conclusion

In this thesis research, we tried creating a text summarization system by utilizing the power of a transformer model. We successfully trained a simple and small varient of the transformer model to perform text summarization. Although, it was held back by its simplicity and computational constraints, our model performed well in capturing context and generating grammatically accurate summaries. The performance is evident from the sentences it generated. The performance of our summarizer can be improved by training a larger model. In the future, we'd like to continue this research endevour by utilizing pretrained models and fine tuning them to perform summarization. We also hope to manually clean the our accumulated dataset by reviewing the articles and accessing their quality. This will make the task of detecting patterns easier for our small model, which will further uplift its performance.

References

[1] Md Talukder, Sheikh Abujar, Abu Mohammad Masum, Fahad Faisal, and Syed Hossain. Bengali abstractive text summarization using sequence to sequence rnns. 07 2019.

[2] Rico Sennrich, Barry Haddow, and Alexandra Birch. Neural machine translation of rare words with subword units, 2016.

[3] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need, 2017.

[4] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding, 2019.

[5] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. Roberta: A robustly optimized bert pretraining approach, 2019.

[6] Zhenzhong Lan, Mingda Chen, Sebastian Goodman, Kevin Gimpel, Piyush Sharma, and Radu Soricut. Albert: A lite bert for self-supervised learning of language representations, 2020.

[7] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15(56):1929–1958, 2014.

[8] Chin-Yew Lin. ROUGE: A package for automatic evaluation of summaries. In *Text Summarization Branches Out*, pages 74–81, Barcelona, Spain, July 2004. Association for Computational Linguistics.

[9] Benjamin Heinzerling and Michael Strube. BPEmb: Tokenization-free Pre-trained Subword Embeddings in 275 Languages. In Nicoletta Calzolari (Conference chair), Khalid Choukri, Christopher Cieri, Thierry Declerck, Sara Goggi, Koiti Hasida, Hitoshi Isahara, Bente Maegaard, Joseph Mariani, Hélène Mazo, Asuncion Moreno, Jan Odijk, Stelios Piperidis, and Takenobu Tokunaga, editors, *Proceedings of the Eleventh International Conference on Language Resources and Evaluation (LREC 2018)*, Miyazaki, Japan, May 7-12, 2018 2018. European Language Resources Association (ELRA).

[10] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2017.

[11] Lutz Prechelt. Early stopping - but when? 03 2000.

[12] Weizhen Qi, Yu Yan, Yeyun Gong, Dayiheng Liu, Nan Duan, Jiusheng Chen, Ruofei Zhang, and Ming Zhou. Prophetnet: Predicting future n-gram for sequence-to-sequence pre-training, 2020.

[13] Jingqing Zhang, Yao Zhao, Mohammad Saleh, and Peter J. Liu. Pegasus: Pre-training with extracted gap-sentences for abstractive summarization, 2020.

[14] Manzil Zaheer, Guru Guruganesh, Avinava Dubey, Joshua Ainslie, Chris Alberti, Santiago Ontanon, Philip Pham, Anirudh Ravula, Qifan Wang, Li Yang, and Amr Ahmed. Big bird: Transformers for longer sequences, 2021.

[15] Liu Yang, Mingyang Zhang, Cheng Li, Michael Bendersky, and Marc Najork. Beyond 512 tokens: Siamese multi-depth transformer-based hierarchical encoder for long-form document matching, 2020.