University of Nottingham
UK | CHINA | MALAYSIA

# Efficient Summarization of Research Papers Using Fine-Tuned Small Context Language Models

Submitted September 2024, in partial fulfillment of
the conditions for the award of the degree **MSc Computer Science (AI).**

**Shruti Sundaram**
**20543520**

**Supervised by Dr. Warren Jackson**

School of Computer Science
University of Nottingham

I hereby declare that this dissertation is all my own work

Signature: Shruti Sundaram

Date: 06 / 09 / 2024

# Abstract

Summarizing long documents, especially research papers, is challenging due to their complexity and length. Traditional large language models (LLMs) are designed for short text and struggle with the extensive context needed for academic content. New models like Longformer handle larger contexts but are costly, hard to fine-tune, and resource-intensive.

This paper introduces a cost-effective two-step approach for summarizing long documents, focusing on aiding researchers in literature reviews. The first step involves supervised training of the BART-large-CNN model to enhance its ability to generate high-quality summaries of scientific papers. The second step uses an innovative adaptive recursive summarization strategy, allowing the model to manage lengthy documents despite token limits by employing techniques like dynamic recursion, memory management, and strategic chunking. This method efficiently summarizes extensive texts while maintaining coherence, reducing computational costs, and minimizing environmental impact.

To further extend the practical utility of this research, an application has been developed that allows users to upload research paper PDFs, which the model then processes to generate concise, coherent summaries, streamlining the literature review process by making it easier to extract key insights from vast amounts of academic content.

Empirical results show that the adaptive recursion strategy preserves core content and outperforms the LED Longformer model in quality, offering a scalable, sustainable solution for researchers.

## Acknowledgements

I would like to express my sincere gratitude to my dissertation supervisor, Dr Warren Jackson, for their invaluable guidance and support throughout this research. Their expertise and insightful feedback were crucial to the completion of this work. I am also deeply grateful to my family for their unwavering support and patience. Their encouragement has been a steady source of motivation. Lastly, I would like to thank my friends for their understanding and support during this journey. Their companionship and encouragement have been invaluable.

# Contents

# List of Tables

# List of Figures

# Chapter 1

# Introduction

The process of conducting a literature review can be both time-consuming and exhausting, as researchers and practitioners often need to sift through numerous lengthy academic papers to extract essential information quickly. Given the proliferation of textual data across various domains, the ability to efficiently summarize these long documents has become increasingly crucial. Traditionally, large language models (LLMs) with long context windows have been employed to perform this task. However, these models come with significant computational requirements, making them resource-intensive both during their pre-training and fine-tuning phases. The pre-training of these models demands extensive computational power and energy over prolonged periods, while fine-tuning often exceeds 100 hours, leading to high costs and a substantial environmental footprint. Such demands pose notable challenges in terms of efficiency and sustainability, especially for researchers seeking to streamline the literature review process.

This paper addresses the need for efficient summarization by exploring the use of small-context LLMs for summarizing lengthy texts. Unlike large-context models, small-context models with reduced context windows can lower computational load, decrease energy consumption, and reduce fine-tuning times. However, small-context LLMs often truncate long documents to fit within their limited input token length, risking the loss of important content. To address this, a novel summarization strategy using an adaptive recursive function with memory updates and dynamic text chunking is introduced.

## 1.1 Motivation

My interest in this research is driven by the challenge of balancing cost-effectiveness with performance in document summarization while reducing computational and energy demands. This is particularly important for researchers and practitioners who need to quickly grasp the essence of lengthy academic papers and automate literature reviews—a task that is often time-consuming and exhaustive. Traditional long-context large language models (LLMs) are powerful for summarizing lengthy texts in a single pass, but they come with significant computational and financial costs due to their intensive pre-training requirements.

In contrast, small-context LLMs offer a more cost-effective alternative, especially in research settings where reducing costs and maintaining summary quality are crucial. Fine-tuning these models is less resource-intensive than training large models from scratch. Techniques such as recursive functions or chunking allow small-context LLMs to handle long documents without truncation by processing smaller segments. Although recursive processing may require multiple passes, the overall training and fine-tuning costs can be lower, making these models more accessible to researchers with limited resources.

However, there are trade-offs. Long-context LLMs, designed to handle extensive documents in one pass, typically require more resources but can directly process long texts, potentially reducing inference costs and improving summary coherence. Small-context models, with effective recursive strategies, might offer a balanced solution, maintaining high-quality summaries at a lower cost.

This research aims to explore alternative methods that leverage smaller context windows without sacrificing the quality of generated summaries, ultimately aiding researchers and practitioners in efficiently understanding complex academic texts. Given my strong interest in AI advancements and environmental sustainability, I am concerned with the high costs and inefficiencies associated with long-context LLMs. The prospect of adapting small-context LLMs to handle long documents is particularly appealing, as they offer a promising approach to reducing energy consumption and computational costs. This aligns with my commitment to promoting sustainable technology and making summarization tools more accessible to a broader audience.

## 1.2 Aims and Objectives

This paper aims to test the hypothesis that a fine-tuned LLM with a small context window can be more effective than long-context LLMs for summarizing research papers in order to perform literature reviews. The LLM model will be trained to generate high-quality summaries that align with the structure and style of research papers, ensuring that the output is not only concise but also academically rigorous. To address the token size limitations inherent in small-context models, this study will employ adaptive recursion as the primary strategy, combined with dynamic chunking. By demonstrating that these methods enable small-context LLMs to effectively manage and summarize long research papers without loss of information, the goal is to provide a more resource-efficient, environmentally friendly, and cost-effective alternative to traditional long-context models.

The primary objective of this research is to evaluate whether small-context LLM models, fine-tuned using supervised learning, when combined with advanced summarization strategies can process long research papers without truncation or information loss, and match the performance of their long-context counterparts. This research is expected to contribute to the field by:

- Performing Transfer Learning to finetune an LLM model to a specific use-case and optimizing hyperparameters with early stopping and validation metrics.
- Demonstrating the feasibility of small-context LLM models for long-document summarization through innovative strategies.
- Providing insights into more cost-efficient and environmentally friendly summarization methods.
- Offering practical recommendations for implementing small-context models in real-world applications.

## 1.3 Description of the work

The project will involve manipulating LLM models with small context windows to perform long document summarization through the use of appropriate recursive functions and chunking strategies. Specifically, this study aims to:

**Fine-tune a Small-Context LLM**

Fine-tune a large language model (LLM) with a small context window on a research paper dataset. This step will involve preparing the dataset, training the model to adapt it to the specific language and structure of research papers through supervised transfer learning, and evaluating its initial summarization performance on shorter texts.

**Develop an Adaptive Recursive Summarization Function**

Design and implement a adaptive recursive function to handle long documents. This function will process long research papers by breaking them into manageable chunks, recursively summarizing each chunk and performing memory updates through concatenation of intermediate summaries, and integrating these summaries into a cohesive final summary.

**Evaluate and Compare Performance**

Perform testing on a designated test dataset to compare the performance of the fine-tuned small-context LLM against a long-context model, such as a Longformer model. This comparison will focus on the ROUGE metric. The aim is to assess whether the small-context model, when enhanced with adaptive recursion and dynamic chunking, meets or exceeds the performance of the long-context model.

# Chapter 2

# Background and Related Work

Artificial Intelligence (AI) models are typically divided into two categories: predictive and generative. Predictive models are designed to make specific decisions by classifying labeled data. On the other hand, models that have the ability to create data, such as text, images, or other forms of data, are known as Generative AI. Figure 2.1 shows the relationship between the different categories of AI [1].

One of the most prominent examples of Generative AI is the Large Language Model (LLM) which is trained on vast amounts of text data to generate human-like language. They are based on Transformers, a type of neural network architecture invented by Google. What made the Transformer architecture powerful was its ability to scale effectively, allowing to train these models on massive text datasets. LLMs can understand context,

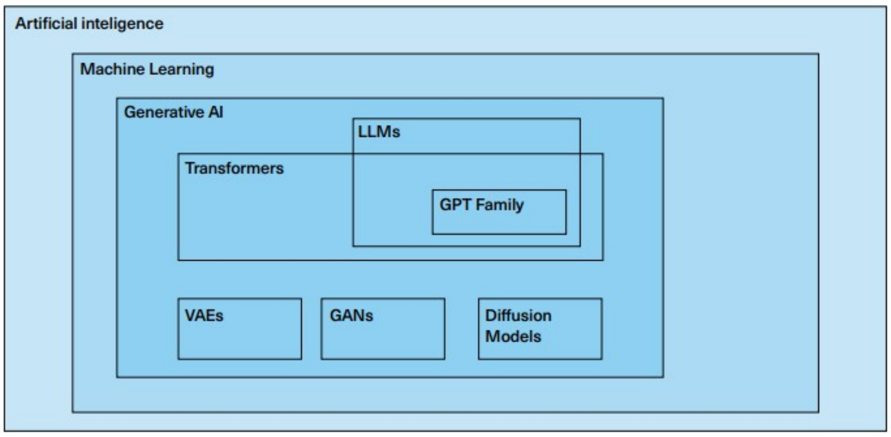


Figure 2.1: Relation between LLMs, GenAI architectures, and broader fields of ML and AI (Source: The Alan Turing Institute, 2024 [1])

generate coherent responses, and be used for a variety of tasks such as chat, summarizing, translation etc.

While LLMs are a type of Artificial Neural Network (ANN) in a broad sense, their specific design and capabilities are rooted in the Transformer framework. The following sections will discuss the evolution of modern LLM models.

## 2.1 Evolution of Large Language Models

### 2.1.1 Early Chatbots and Rule-Based Language Models

In 1966, ELIZA - widely regarded as the first chatbot - was developed by Joseph Weizenbaum [42] at MIT. Unlike modern conversational agents, such as ChatGPT, ELIZA did not understand the context of conversations. Instead, it employed fixed response templates and fallback replies to maintain the flow of dialogue, creating the illusion of conversation by rephrasing users' statements into questions. It's responses were generated through rule-based mechanisms, relying on handcrafted rules and grammatical structures for text analysis and generation. However, this meant that ELIZA could easily fall into repetitive loops or give non-sequitur responses if the input didn't match any recognized patterns.

### 2.1.2 Statistics-based Language Models

Moving into the 1980s and 1990s, researchers continued to develop new approaches to natural language processing (NLP), including statistical-based models. During this decade, NLP increasingly relied on probabilistic models to analyze language. One of the pioneering studies was conducted by Karen Spärck Jones [34], who introduced the concept of Inverse Document Frequency (IDF), a key component in modern search algorithms. IDF is a statistical measure used in information retrieval and text mining to evaluate the importance of a word within a collection of documents, often referred to as a corpus.

**Term Frequency (TF)** measures the frequency with which a word appears in a specific document. Words that appear frequently are often considered important within that document.

**Document Frequency (DF)** counts how many documents in the corpus contain the word. Common words that appear in many documents are less distinctive.

**Inverse Document Frequency (IDF)** is calculated as the logarithm of the total number of documents divided by the number of documents containing the word. This measure helps to weigh down the common words and highlight the more distinctive ones.

$$IDF(t) = log(N/DF(t))$$

N is the total number of documents in the corpus.

DF(t) is the number of documents that contain the term t

IDF is typically combined with Term Frequency (TF) to form the **TF-IDF score**, which is widely used in search engines and text analysis to rank the relevance of documents to a search query. Higher TF-IDF scores indicate that the term is more relevant to a particular document.
For example, common words like "the" or "and" would have a high document frequency, resulting in a low IDF score. A rare or specific term, like "machine learning," would appear in fewer documents, giving it a higher IDF score.

### N-grams

IDF laid the foundations for later models to incorporate probabilistic techniques into language processing. N-grams are a statistical method for predicting the next item in a sequence, commonly used in language models to calculate the probability of a word based on the preceding words. N-gram models predicted the probability of a word given its previous n-1 words. Although they improved language understanding to some extent, they lacked the ability to capture long-range dependencies.

#### Hidden Markov Models

In the late 1980s, natural language processing (NLP) began integrating machine learning approaches, utilizing algorithms to learn from extensive language data. A notable early application was the use of Hidden Markov Models (HMMs) for speech recognition. HMMs model sequences of words through hidden states connected by probabilistic transitions, with each state emitting words according to specific probabilities.

The paper ”A tutorial on hidden Markov models and selected applications in speech recognition” (Rabiner et al. [31]) introduces HMMs as statistical tools for modeling sequences with hidden states inferred from observable data. It covers key components like hidden states, transition and emission probabilities, and the initial-state distribution. The paper also presents the Forward algorithm for sequence likelihood estimation, the Viterbi algorithm for decoding the most probable sequence of hidden states, and the Baum-Welch algorithm for training HMMs by optimizing model parameters using observed data.

### 2.1.3 Neural Language Models

In the late 1990s, the resurgence of neural networks led to the development of early neural language models that incorporated architectures such as Recurrent Neural Networks (RNNs), Convolutional Neural Networks (CNNs), and Gated Recurrent Units (GRUs). These models marked a major departure from conventional statistical methods in language processing.

#### Recurrent Neural Networks (RNNs):

Introduced in 1990 in the paper ”Finding Structure in Time” (Elman et al.[13]), RNNs were designed to handle sequential data by maintaining a hidden state that could capture information from previous time steps. Unlike traditional feedforward neural networks, where the flow of information was in one direction, RNNs could remember previous inputs in their internal state or memory - thanks to their feedback loop - making them more suitable for natural language processing.

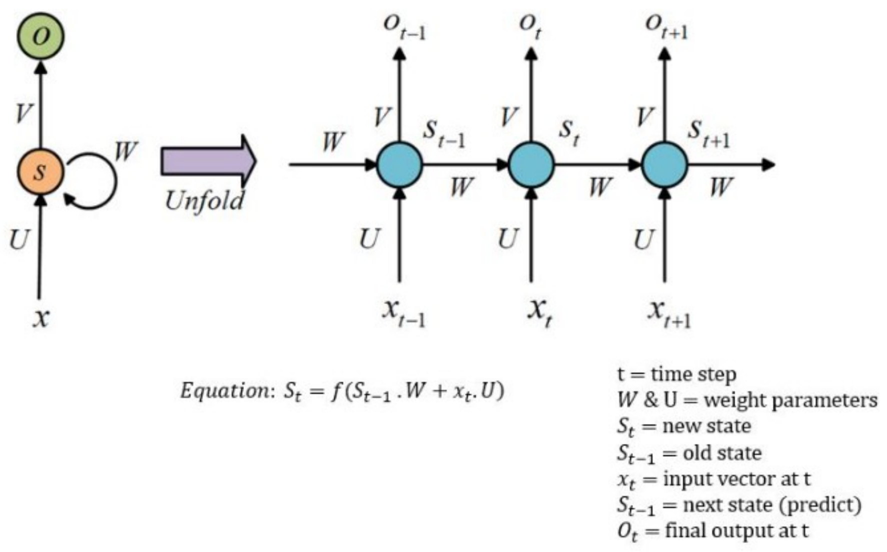


Figure 2.2: RNN architecture (Source: Zhu, 2019 [45])

As shown in Figure 2.2, RNNs have a state $S_t$ that is updated at each time step as a sequence is processed. A recurrence relation is applied at every time step to process a sequence, allowing the network to maintain and use context from previous steps to inform the processing of future steps. This enables RNNs to effectively handle sequences of varying lengths and capture temporal dependencies within the data.
However, RNNs struggled with long-range dependencies due to the vanishing gradient problem, where gradients during training would become exceedingly small, making it difficult to learn long-term dependencies.

**Long Short-Term Memory (LSTM) Networks:**

Introduced in the 1997 paper ”Long Short-Term Memory” (Hochreiter et al.[17]), Long Short-Term Memory (LSTM) networks improve on traditional RNNs by using a memory cell and three gates—input, forget, and output—to manage the flow of information across time steps. The memory cell retains important information over long sequences, while the input gate updates the cell state with new data, the forget gate discards unnecessary information, and the output gate determines the final output and hidden state. This architecture, as shown in Figure 2.3, allows LSTMs to effectively capture long-term dependencies and mitigate vanishing gradients, although they are computationally more complex.

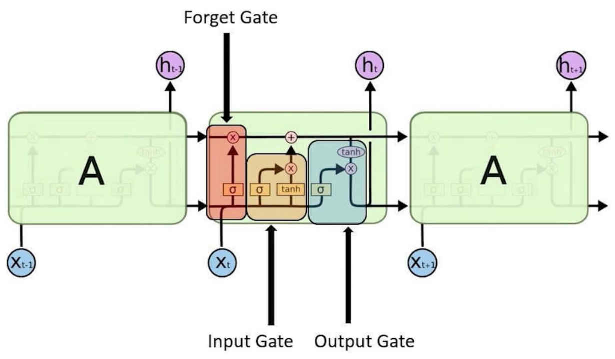


Figure 2.3: LSTM Architecture (Source: Benabderrahmane, 2018 [6])

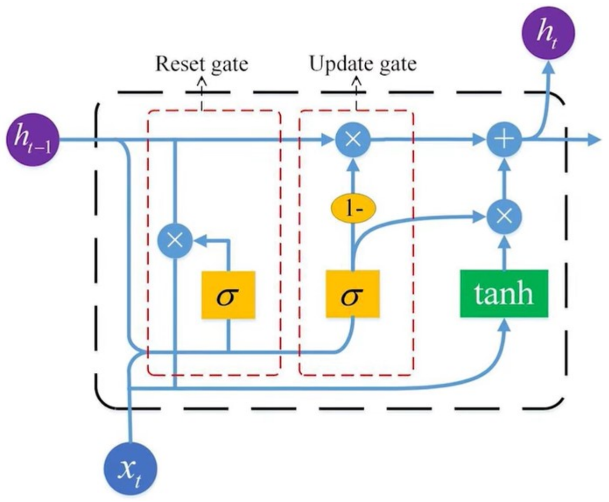


Figure 2.4: Gated Recurrent Unit (GRU) architecture (Source: Chen at al. 2021 [9])

### Gated Recurrent Units (GRU)

In 2014, Gated Recurrent Units (GRUs) were introduced as an alternative to Long Short-Term Memory (LSTM) networks, aimed at addressing similar issues with a more streamlined and simplified architecture (Cho et al.[10]). Like LSTMs, GRUs were designed to mitigate the problem of vanishing gradients, thereby enhancing the retention of long-term dependencies within sequences. GRUs achieve this by employing a reduced gating mechanism, utilizing only two gates: the Update Gate and the Reset Gate. Figure 2.4 shows the architecture of a GRU. The Update Gate determines the extent to which the previous information is retained versus the new information that is integrated, while the Reset Gate governs the extent to which the previous information is discarded. This simplification in the gating structure contributes to a greater computational efficiency in GRUs compared to LSTMs (Nosouhian et al.[27]).

### 2.1.4 Word Embeddings

Around 2013, word embeddings were introduced in the paper "Efficient estimation of word representations in vector space" (Thomas Mikolov, [24]) revolutionized the field of natural language processing (NLP) by providing a more effective way to represent words in computational models. Traditional methods, such as one-hot encoding, represented words as sparse vectors with high dimensionality, where each word was encoded as a unique vector with a single non-zero entry. However, this approach lacked the ability to capture semantic relationships between words, which led to limitations in many NLP tasks.

Word embeddings, such as Word2Vec, GloVe, and FastText, addressed these limitations by representing words as dense vectors in a continuous vector space. These vectors are typically of much lower dimensionality than one-hot encoded vectors, and importantly, they capture semantic similarities between words based on their context in large text corpora.

### 2.1.5 Seq2Seq Models

The paper "Sequence to Sequence Learning with Neural Networks" (Sutskever et al.[36]) introduces an innovative architecture for handling sequence-to-sequence tasks using Long Short-Term Memory (LSTM) networks. The proposed model consists of two key components: an encoder and a decoder, both implemented with LSTM networks. The encoder processes the input sequence and maps it to a fixed-dimensional vector representation, which captures the entire sequence's information. This vector is then fed into the decoder, which generates the output sequence one step at a time. To improve the model's ability to learn dependencies, especially in long sentences, the authors reversed the order of words in the input sequence. This architectural design (shown in Figure 2.5), particularly the use of LSTMs in both encoding and decoding phases, enables the model to handle input and output sequences of varying lengths, making it highly effective for tasks like machine translation. The methodology proved successful, achieving strong results in translation tasks, and demonstrated the power of deep learning for complex sequence

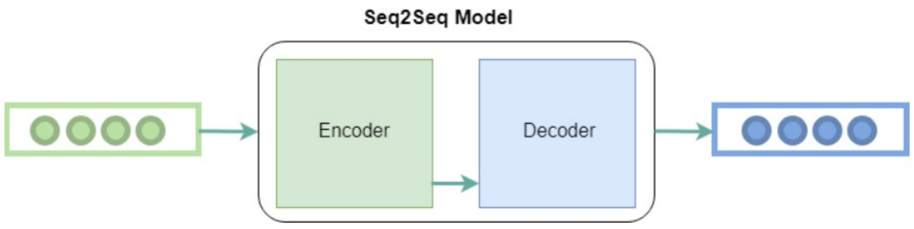


Figure 2.5: Encoder and Decoder Stack in seq2seq model (Source: He et al. 2022 [16])

prediction problems.

## 2.1.6 Attention Mechanism

A major limitation of the basic Seq2Seq model is that it relies on compressing all the input sequence information into one fixed-length vector, which can be challenging for long sequences or when the input has varying importance at different points.
The Attention mechanism was introduced later to address this limitation by allowing the decoder to access different parts of the input sequence directly, rather than relying solely on the fixed-length vector. Instead of generating the output sequence based only on a single vector, the Attention mechanism creates a weighted sum of all the encoder's hidden states. This allows the model to "attend" to different parts of the input sequence at each step of the decoding process, dynamically focusing on relevant parts of the input as it generates each word in the output sequence (Niu et al.[26]). This enables the model to better handle the relationship between different parts of the input and output sequences.

Key Components of the Attention Mechanism include:

**Encoder** Like in a basic Seq2Seq model, the encoder processes the input sequence and generates a sequence of hidden states $h_1$, $h_2$,...,$h_T$ where $T$ is the length of the input sequence. Each hidden state $h_i$ represents the encoder's understanding of the input at time step $i$.

**Attention Layer** The attention mechanism introduces a context vector that changes at each time step of the decoding process. For each time step $t$ in the output sequence, the decoder computes a score for each hidden state of the encoder. These scores are typically calculated using a similarity function, such as:

- Dot Product: score($h_i$, $s_t$) = $h_i^T s_t$
- Additive (Bahdanau): score($h_i$, $s_t$) = $v_\alpha^T$ tanh( $W_\alpha[S_t;h_i]$)
- Scaled Dot Product (used in Transformers): score($h_i$, $s_t$) = $h_i^T$ $s_t$ / sqrt(d)

### 2.1.7 Transformers

The shift from recurrent neural networks (RNNs) to transformers represents a major advancement in handling sequential data. While RNNs, including LSTMs and GRUs, addressed temporal dependencies, they struggled with long sequences and inefficiency. Sequence-to-Sequence (Seq2Seq) models improved this with attention mechanisms but still relied on RNNs. Transformers, introduced by Vaswani et al. [39], replaced RNNs with self-attention mechanisms, enabling parallel processing of sequences, which improved efficiency, scalability, and the modeling of complex dependencies. This innovation laid the groundwork for large language models (LLMs) and accelerated natural language processing progress.

Figure 2.6 shows the overall architecture of a Transformer model. Several types of attention mechanisms are used in the Transformer model, such as:

**Scaled Dot-Product Attention:** Calculates token relevance by computing scaled dot products of queries and keys, followed by softmax to determine attention weights.

**Self-Attention:** Allows tokens to attend to all others, capturing dependencies regardless of distance, enhancing context understanding.

**Multi-Head Attention:** Computes attention with multiple heads to capture different relationship aspects between words.

**Encoder-Decoder Attention:** Aligns input and output sequences by focusing on different encoded input parts during output generation.

**Positional Encoding:** Adds order information to embeddings, helping the model understand token positions.

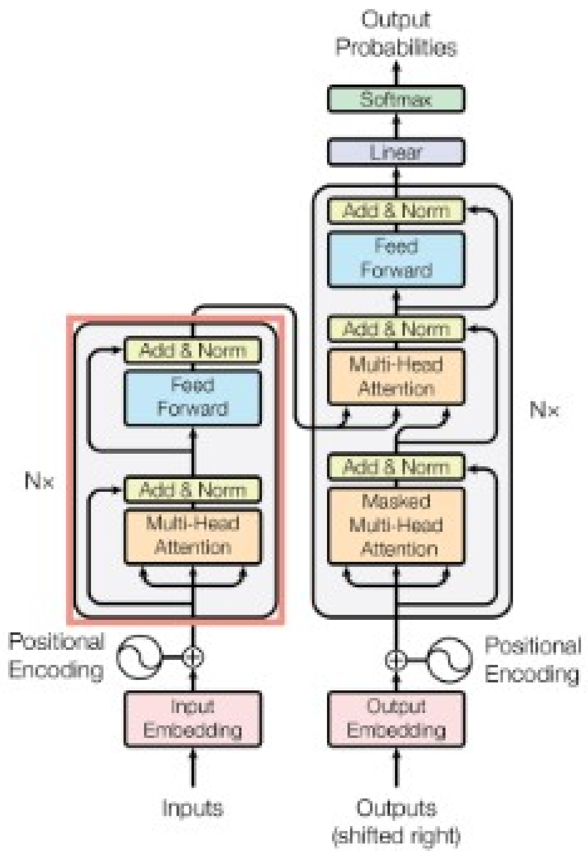


Figure 2.6: Transformer architecture (Source: Vaswani, 2017 [39])

**Feedforward Neural Network:** Processes data after attention, applied identically to each position.

**Layer Normalization and Residual Connections:** Stabilize deep model training by connecting sub-layers with residual connections and layer normalization.

### 2.1.8 Pre-trained large language models

The advent of large language models that utilize the Transformer architecture, introduced in 2017, marked a significant advancement in natural language processing. The Transformer's self-attention mechanism and parallel processing capabilities paved the way for highly effective models trained on vast amounts of text data.

Pre-trained large language models (LLMs) are deep learning models that have been trained on vast amounts of text data in an unsupervised manner. These models, such as GPT (Generative Pre-trained Transformer), BERT (Bidirectional Encoder Representations from Transformers), and others, have learned to understand and generate human language by predicting the next word in a sentence or by understanding context from surrounding words. This training allows them to acquire a broad understanding of language, grammar, and even some world knowledge.

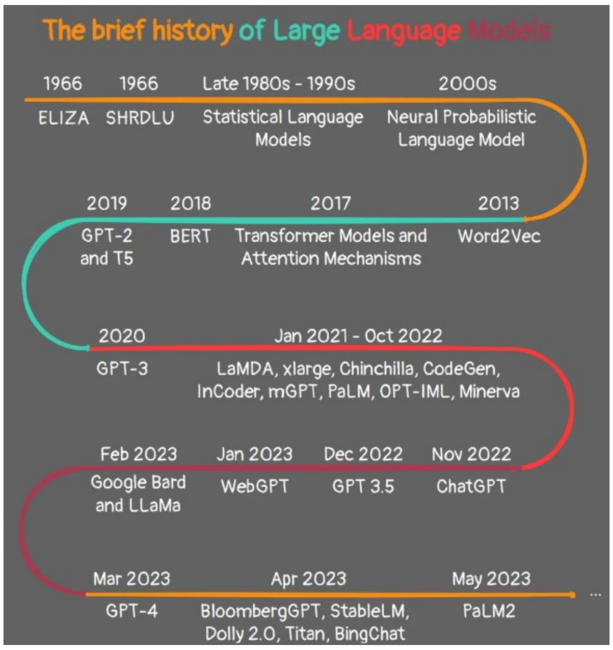

The brief history of Large Language Models
1966
ELIZA
1966
SHRDLU
Late 1980s - 1990s
Statistical Language Models
2000s
Neural Probabilistic Language Model
2013
Word2Vec
2017
Transformer Models and Attention Mechanisms
2018
BERT
2019
GPT-2 and T5
2020
GPT-3
Jan 2021 - Oct 2022
LaMDA, xlarge, Chinchilla, CodeGen, InCoder, mGPT, PaLM, OPT-IML, Minerva
Nov 2022
ChatGPT
Dec 2022
GPT 3.5
Jan 2023
WebGPT
Feb 2023
Google Bard and LLaMa
Mar 2023
GPT-4
Apr 2023
BloombergGPT, StableLM, Dolly 2.0, Titan, BingChat
May 2023
PaLM2
...

Figure 2.7: Infographics of the evolution of language models over time (Source: Medium [21])

During the pre-training phase, LLMs are exposed to a large corpus of text, often comprising billions of words from diverse sources such as books, articles, websites, and social media. The model learns to predict words or phrases in context, effectively capturing a wide range of linguistic patterns, word associations, and contextual relationships. This process equips the model with a general understanding of language, but the knowledge is not specifically tailored to any one application.

In 2018, Google introduced BERT (Bidirectional Encoder Representations from Transformers), which improved NLP tasks by capturing context in both directions within a sentence. OpenAI followed in 2019 with GPT-2, a large-scale unsupervised model known for its impressive language generation capabilities. In November 2022, OpenAI released ChatGPT, based on the GPT-3 architecture, which further advanced human-like text generation and complex query handling, drawing significant attention in NLP and generative AI. Since then, models such as PaLM, T5, LaMDA, DALL-E, and LLaMa have continued to push the boundaries of language model capabilities.

Figure 2.7 shows visual representation of the progression of language models throughout history [21].

## 2.2 Summarization Overview

Text Summarization with Large Language Models (LLMs) involves using advanced NLP techniques to condense long documents into concise summaries while preserving key information and meaning. LLMs, such as GPT-4, are particularly effective for this task due to their ability to understand and generate human-like text based on large-scale training data.
LLMs use attention mechanisms and context understanding to process and summarize texts. Using their vast training data, they can grasp the context, identify crucial information, and generate summaries that reflect the essence of the original content.

### 2.2.1 Types of Summarization

- **Extractive Summarization:** This approach selects and extracts key sentences or phrases directly from the source text to create a summary. LLMs can be fine-tuned to identify and extract the most relevant parts of the text.

- **Abstractive Summarization:** This method involves generating new sentences that convey the main ideas of the text, often rephrasing or rewording the content. LLMs excel at this by generating coherent, human-like summaries that may include paraphrasing and new phrasing.

Figure 2.8 is an illustration of the sub-fields within text summarization.

LLMs are typically pre-trained on diverse datasets to understand general language patterns. For specific summarization tasks, they can be fine-tuned on summarization datasets, learning to generate more accurate and contextually appropriate summaries.

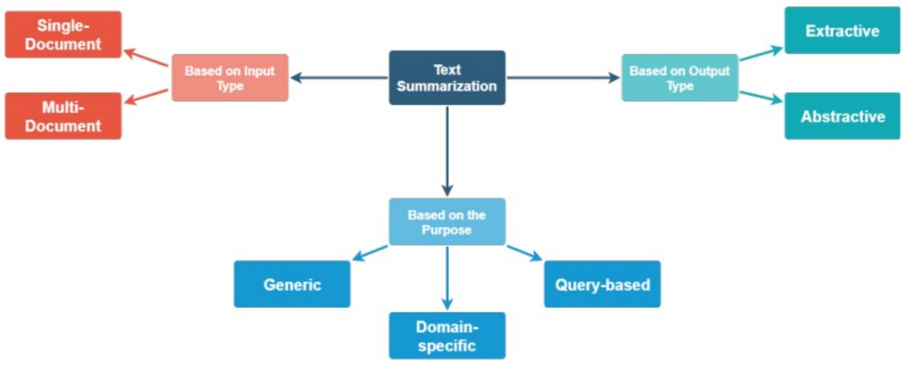


Figure 2.8: Overview of text summarization (Source: Gaskell, 2020 [14])

Applications include summarizing news articles, generating executive summaries for reports, creating concise content for social media, and providing quick overviews of lengthy documents.

## 2.3 Popular LLMs for Text Summarization

### 2.3.1 BERT (Bidirectional Encoder Representations from Transformers)

BERT, based on the Transformer architecture, uses only the encoder portion, consisting of layers with multi-head self-attention and feed-forward networks. Unlike unidirectional models like GPT, BERT is bidirectional, capturing context from both the left and right of a word, leading to a deeper understanding of language. It processes sentence pairs simultaneously, using token, segment, and position embeddings to maintain context and word order.

Pre-trained with Masked Language Modeling (MLM) and Next Sentence Prediction (NSP), BERT is fine-tuned for specific NLP tasks like text classification and question answering. It comes in two main variants: BERT-Base with 12 layers and 110 million parameters, and BERT-Large with 24 layers and 340 million parameters. BERT's bidirectional approach and effective pre-training have made it foundational in modern NLP tasks, including search engines, chatbots, and summarization.

### 2.3.2 GPT-2 (Generative Pre-trained Transformer 2)

The GPT (Generative Pre-trained Transformer) model, developed by OpenAI in the paper "Language Models are Unsupervised Multitask Learners" [32], is based on the Transformer architecture, introduced by Vaswani et al. in the paper "Attention is All You Need" (2017) [39].

The GPT is a stack of transformer decoders. It is pre-trained on the language modeling objective of predicting the next word in a given sentence. It processes text in a unidirectional manner and generates text by predicting each word based on the words that precede it. This sequential approach allows the model to build up context as it generates each word, making it particularly effective at producing coherent and contextually relevant text.

The self-attention mechanism in GPT allows the model to weigh the importance of different words in the input sequence relative to each other. This mechanism enables GPT to focus on the most relevant parts of the input when generating the next word, which is crucial for maintaining coherence in tasks like summarization.

Since Transformers do not inherently understand the order of tokens, GPT includes positional encoding to incorporate the sequential nature of language. It employs masked attention in the decoder layers to prevent tokens from attending to future positions, ensuring that predictions are based only on previously seen tokens, which is crucial for autoregressive text generation.

Figure 2.9 illustrates that tokens in GPT are predicted auto-regressively, allowing the model to be used for generation; however, since words can only condition on leftward context, GPT cannot learn bidirectional interactions.

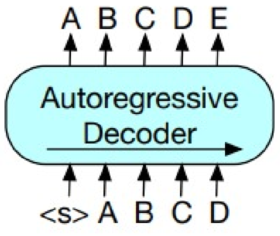


Figure 2.9: GPT architecture (Source: Lewis et al. 2019 [18])

### 2.3.3 T5 (Text-To-Text Transfer Transformer)

T5 (Text-To-Text Transfer Transformer), introduced by Google in the paper ”Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer” (2020) [33], is a versatile language model designed to handle a wide range of natural language processing (NLP) tasks using a unified text-to-text framework.

T5 is based on the Transformer architecture, specifically incorporating both the encoder and decoder components. The encoder processes the input text, and the decoder generates the output text. This structure is similar to that of sequence-to-sequence models traditionally used in machine translation but is adapted for a wider range of tasks.

To guide the model in performing different tasks, T5 uses task-specific prefixes. For example, to perform summarization, the input might be prefixed with ”summarize:”. This prefix informs the model of the task at hand, allowing it to generate the appropriate type of output.

### 2.3.4 BART (Bidirectional and Auto-Regressive Transformers)

BART is a sequence-to-sequence (seq2seq) model developed by researchers at Facebook AI and introduced in the paper titled ”BART: Denoising Sequence-to-Sequence Pre-training for Natural Language Generation, Translation, and Comprehension” by Mike Lewis et al. (2019) [18]. BART combines the strengths of both bidirectional and autoregressive Transformers, making it highly effective for various natural language processing (NLP) tasks, especially text summarization.

BART combines the best of BERT’s bidirectional encoding with GPT’s autoregressive decoding. It is trained as a denoising autoencoder, learning to reconstruct original text from corrupted versions using various noise functions. This design enables BART to effectively capture context and generate coherent sequences, making it highly suitable for tasks like text summarization.

Figure 2.10 demonstrates that in BART, the encoder’s input does not need to align with the decoder’s outputs, enabling flexible, noisy transformations. In this example, a document is corrupted by replacing text spans with mask symbols. The corrupted

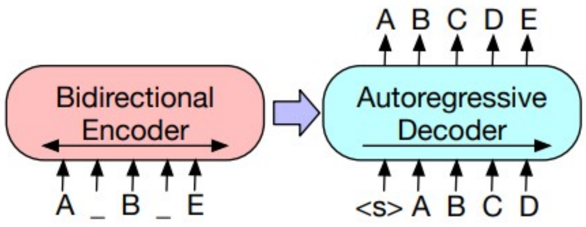


Figure 2.10: BART architecture (Source: Lewis et al. 2019 [18])

document (left) is encoded using a bidirectional model, and the likelihood of the original document (right) is then computed using an autoregressive decoder.

## 2.4 Challenges in Summarization with standard LLMs

### Context Window

Many LLMs have a fixed context window or token limit, which can impact their ability to summarize long documents effectively. BART, designed specifically for tasks like summarization, has a typical input token limit of 1024 tokens. GPT-2 has a context window of 1024 tokens, which is shorter than GPT-3's 2048 tokens but similar to models like BART. In comparison, BERTSUM (which adapts BERT for summarization) (Lewis et al.[18]) and T5 share a similar token limit of 512 tokens (Raffel et al.[33]). The limited token size can lead to potentially missing crucial information in longer documents.

Techniques like chunking the document into smaller sections and summarizing each separately can help. However, this may require additional steps to ensure coherence between chunks.

### Accuracy and Relevance

Ensuring that the summary accurately reflects the key points of the original text without introducing errors or omissions is a challenge. LLMs can sometimes generate plausible but incorrect or irrelevant information.

#### Bias and Information Loss

Summarization models can inadvertently introduce biases or omit critical information, leading to summaries that do not fully represent the source material.

#### Coherence and Consistency

Maintaining coherence and consistency in generated summaries can be challenging, especially when summarizing complex or lengthy documents.

## 2.5 Specialized Models for Long Document Summarization

### 2.5.1 Longformer

Longformer, introduced in the paper ”Longformer: The Long-Document Transformer” by Beltagy et al. [5], is designed to address the limitations of traditional Transformer models when dealing with long documents. Regular Transformers use a self-attention mechanism where every token in the input text attends to every other token, resulting in computational costs that grow quadratically with the sequence length. This makes them impractical for processing long texts. Longformer extends the Transformer model's ability to handle long sequences by introducing a more efficient attention mechanism that scales linearly with sequence length.

Longformer introduces four types of self-attention, as shown in diagram 2.11, to efficiently handle long documents. It uses a sliding window attention mechanism, where each token attends to a local context, reducing computational complexity compared to standard Transformers. To maintain global context, Longformer adds global attention, allowing key tokens to attend to all others in the sequence.

It also incorporates dilated sliding windows to capture longer-range dependencies without increasing window size. Longformer retains the core Transformer architecture but supports context windows up to 4096 tokens, significantly more than models like BERT and

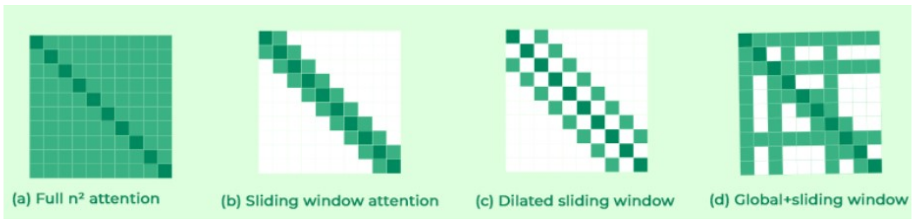

(a) Full n² attention
(b) Sliding window attention
(c) Dilated sliding window
(d) Global+sliding window

Figure 2.11: Longformer self-attention mechanism (Source: Medium [22])

GPT-2.

### 2.5.2 BigBird

Big Bird, introduced in the paper "Big Bird: Transformers for Longer Sequences" by Irfan Abid et al. (2020) [43], extends the context size capability of Transformers significantly, allowing it to handle sequences with up to 8192 tokens in its standard configuration. Traditional Transformers face computational challenges due to the quadratic complexity of the self-attention mechanism, where every token attends to every other token in the sequence. Big Bird overcomes this by introducing a sparse attention mechanism that reduces this complexity, making it feasible to manage much longer sequences.

Big Bird's sparse attention mechanism is a combination of three attention patterns:

- **Windowed (or Sliding) Attention:** Each token attends to a fixed-size window of nearby tokens, capturing local context and ensuring that adjacent information is processed efficiently.

- **Global Attention:** Certain important tokens, such as special tokens or key phrases, can attend to all other tokens in the sequence. This global attention allows the model to maintain a coherent understanding of the entire sequence.

- **Random Attention:** Each token also attends to a random subset of tokens throughout the sequence, which helps capture global dependencies and ensures that even distant parts of the sequence are connected, albeit indirectly.

Figure 2.12 illustrates the Big Bird Attention Mechanism.

By combining these attention types, Big Bird approximates full attention with much

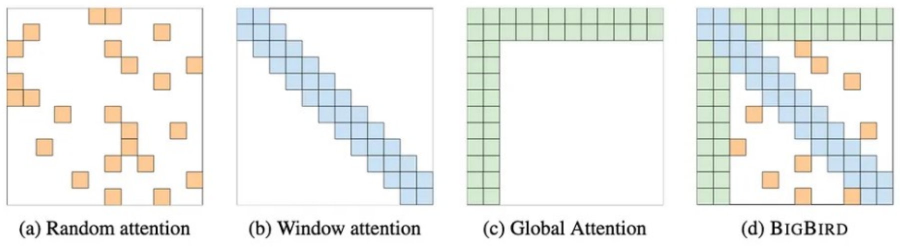


Figure 2.12: Big Bird Attention Mechanism (Source: Medium [7])

lower computational costs, reducing the complexity from quadratic to linear (or sublinear in some cases) with respect to sequence length. This approach allows Big Bird to manage context sizes up to 8192 tokens efficiently, making it particularly well-suited for tasks involving long documents or sequences.

### 2.5.3 LED (Longformer Encoder-Decoder)

The LED (Longformer Encoder-Decoder) model, introduced by Beltagy et al. [5], is designed to handle long documents by extending the Longformer architecture to a sequence-to-sequence (encoder-decoder) framework. While the Longformer is optimized for processing long sequences in tasks like document classification and question answering, LED builds upon this by enabling the generation of new sequences, making it particularly suited for tasks such as document summarization and translation.

LED retains the Longformer's sparse attention mechanism, which combines local (sliding window) attention and global attention to efficiently process long sequences. However, LED extends this sparse attention to both the encoder and the decoder, whereas in Longformer, sparse attention is used solely in the encoder. In the encoder, the sparse attention focuses on understanding the input sequence, while in the decoder, it manages both the generation of the output sequence and the cross-attention with the encoded input. This dual application of sparse attention allows LED to handle sequence generation tasks with long documents effectively.

By leveraging sparse attention in both the encoder and the decoder, LED can handle sequences of up to 16,384 tokens or more, depending on the implementation, making it suitable for long-document tasks.

## 2.6 Challenges with Fine-tuning Specialized Models

### 2.6.1 Training Time

Longformer and Big Bird are designed to handle long sequences with their sparse attention mechanisms, but fine-tuning these models still requires significant time. Training involves adjusting millions to billions of parameters, which can take hours to days depending on the size of the dataset and the computational power available.

LED (Longformer Encoder-Decoder), which builds on Longformer, also demands extensive training time due to its encoder-decoder architecture. This complexity increases the training duration compared to simpler models.

### 2.6.2 Computational Power

Specialized models like Longformer, Big Bird, and LED require powerful hardware, including high-performance GPUs or TPUs, to handle their large-scale computations. The need for large batch sizes, extended sequence lengths, and frequent model updates results in substantial computational demands.

For instance, training such models often involves distributed computing across multiple GPUs or TPUs, which necessitates sophisticated infrastructure and substantial memory resources.

### 2.6.3 Environmental Impact

The computational resources required for training and fine-tuning these models lead to high energy consumption. Large-scale training tasks can consume significant amounts of electricity, contributing to the carbon footprint associated with machine learning research.

Research has shown that training large models can be energy-intensive (E Strubell et al., 2020 [35]). For example, training state-of-the-art models can lead to emissions equivalent to several metric tons of CO2, depending on the efficiency of the data center and energy sources used.

### 2.6.4 Implementation Complexity

Fine-tuning specialized models often requires deep expertise in machine learning and natural language processing. Researchers and practitioners must navigate complex model configurations, adjust hyperparameters, and troubleshoot issues related to long-sequence processing.

Specialized models like Longformer and Big Bird have unique architectural adjustments (e.g., sparse attention), which adds complexity to their implementation and fine-tuning compared to standard Transformer models.

## 2.7 Evaluation Metrics

### 2.7.1 Automated Metrics

#### ROUGE (Recall-Oriented Understudy for Gisting Evaluation)

- **ROUGE-N:** Measures n-gram overlap between the generated summary and reference summaries. For instance, ROUGE-1 assesses unigram overlap, while ROUGE-2 measures bigram overlap (Chin-Yew Lin, 2004 [19]).
- **ROUGE-L:** Evaluates the longest common subsequence between the generated and reference summaries, capturing the fluency and coherence of the summaries (CY Lin et al., 2003 [20]).
- **ROUGE-W:** A variant that incorporates weighted LCS, focusing on the weights of the longest common subsequences to emphasize key information (Mastropaolo et al. [23]).

#### BLEU (Bilingual Evaluation Understudy)

Primarily used in machine translation, BLEU measures the precision of n-grams in the generated text relative to reference texts. While less common for summarization, it can still provide insights into the overlap of generated and reference phrases (Papineni et al.

2002 [28]).

**METEOR (Metric for Evaluation of Translation with Explicit ORdering)**

METEOR evaluates precision and recall of n-grams and incorporates stemming, synonymy, and paraphrasing to provide a more nuanced assessment compared to BLEU (Banerjee et al., 2005 [3]).

**BERTScore**

Utilizes BERT embeddings to evaluate the similarity between generated and reference summaries. It computes similarity scores based on contextual word embeddings, offering a more semantic evaluation than traditional n-gram metrics (Zhang, Tianyi et al., 2019 [44]).

### 2.7.2 Human Judgment Metrics

**Fluency**

Assess the readability and grammatical correctness of the generated summary. Human evaluators check if the text is smooth and free of errors.

**Relevance**

Evaluate whether the summary accurately reflects the main points and important information from the source document. Assessors determine whether the summary includes relevant content and excludes irrelevant details.

**Coherence**

Measure the logical flow and overall structure of the summary. Human judges assess whether the summary maintains a clear and logical progression of ideas.

#### Coverage

Examine whether the summary includes all the key points and important information from the original text. This metric ensures that critical content is not omitted.

## 2.8 Summarization Models for Research paper

Several researchers have specifically focused on fine-tuning LLM models for research paper summarization, with the arXiv dataset being a popular choice for such tasks.
The most widely used model used among all research papers was the LED (Longformer Encoder-Decoder) model, fine-tuned on the summarization datasets.
The token limit for LED models - allowing them to process entire documents or long sections of text in a single pass - is 16,384 tokens. This is approximately 50 pages (double-spaced) or 25 pages (single-spaced).

### 2.8.1 Use of LED Model

The article "Fine-tuning a Longformer Model for Summarization of Machine Learning Articles" by Dimitrii Bakhitov [12], details a study focused on enhancing the summarization of specific domain long documents, particularly machine learning articles, by fine-tuning an LED-base model on a focused subset of the arXiv scientific dataset. The fine-tuning process was computationally intensive, taking over 150 hours on an Nvidia RTX 3070 GPU - this despite limiting the max input token size to 7,168 tokens. To decrease the computational power, the 'LED-base' model was used over the 'LED-large' model, yet the fine-tuning process took no less than 100 hours.

In the paper "On the Summarization and Evaluation of Long Documents" by Alexander Gaskell [14], an LED-large model was fine-tuned on the arXiv and PubMed datasets. The max input token size was limited to 4096 tokens, and the model was trained for 2 epochs on the PubMed dataset and 1 epoch on the arXiv dataset. This was done using a 12 Gb Nvidia GeForce GTX TITAN X GPU (provided by Imperial College London Department of Computing), and these runs took 45 and 40 hours respectively.

### 2.8.2 Challenges with LED model

One of the main challenges in long-input summarization with Transformer-based models is a quadratic scaling of memory consumption as the input length doubles (Beltagy et al. [5]). As an example, let us consider a Transformer-based summarization model with 12 encoder and 12 decoder layers, which were initially pre-trained on short input sequences of 512 tokens and then fine-tuned for specific tasks that involved longer input sequences of 16384 tokens, with an output length of 512 tokens in both cases.

Contrary to the usual practice where fine-tuning is less resource-intensive than pre-training, in this case, fine-tuning is more resource-intensive and slower. This is primarily due to the significant increase in input sequence length during finetuning, which results in a quadratic scaling of memory consumption for self-attention operations in the encoder. Specifically, the encoder's self-attention consumes 1024 times more memory during fine-tuning than pretraining due to the 32-fold increase in input length.

Even if an efficient Transformer variant is used, such as Longformer (Beltagy et al. [5]), to mitigate memory and computation issues, both the encoder's self-attention and the decoder's cross-attention operations still consume 32 times more memory compared to the pretraining stage. Additionally, operations such as the Feed-Forward Network (FFN) that scale linearly with input length also significantly increase the computational requirements during training and inference (Phang et al. [29]).

## 2.9 Use of Small-Context LLMs for summarization

The paper titled "A Divide-and-Conquer Approach to the Summarization of Long Documents" by Alexios Gidiotis and Grigorios Tsoumakas [15] presents a novel method for summarizing long documents by breaking them into smaller, more manageable parts. This approach, termed DANCER (Divide-ANd-ConquER), is designed to reduce the computational complexity and improve the quality of summaries by leveraging the structure of the documents. The DANCER approach divides the document into sections based on a heuristic keyword matching technique. Specifically, the code uses a set of predefined keywords to classify sections of a document into types such as introduction, methods, results, and conclusion. The headers of each section are checked against these keywords, and if a match is found, the section is classified accordingly. Sections that do not match any predefined keywords are ignored. This method relies on a hard-coded logic for keyword matching to determine section types and does not use machine learning for this part of the process.

In the Divide Phase, the long document is segmented into sections based on its discourse structure, such as introduction and methods. Each section's summary is then aligned using sentence-level ROUGE similarity scores, with sentences matched to the most relevant section based on ROUGE-L precision scores. These aligned pairs are subsequently used to create source-target examples for training, where each document section serves as the input and its corresponding summary portion as the target output. In the Conquer Phase, each section of the document is independently summarized by the trained model during inference, and these partial summaries are then concatenated to produce the final summary of the entire document. Figure 2.13 gives the visual representation of the DANCER (Divide-and-Conquer) approach.

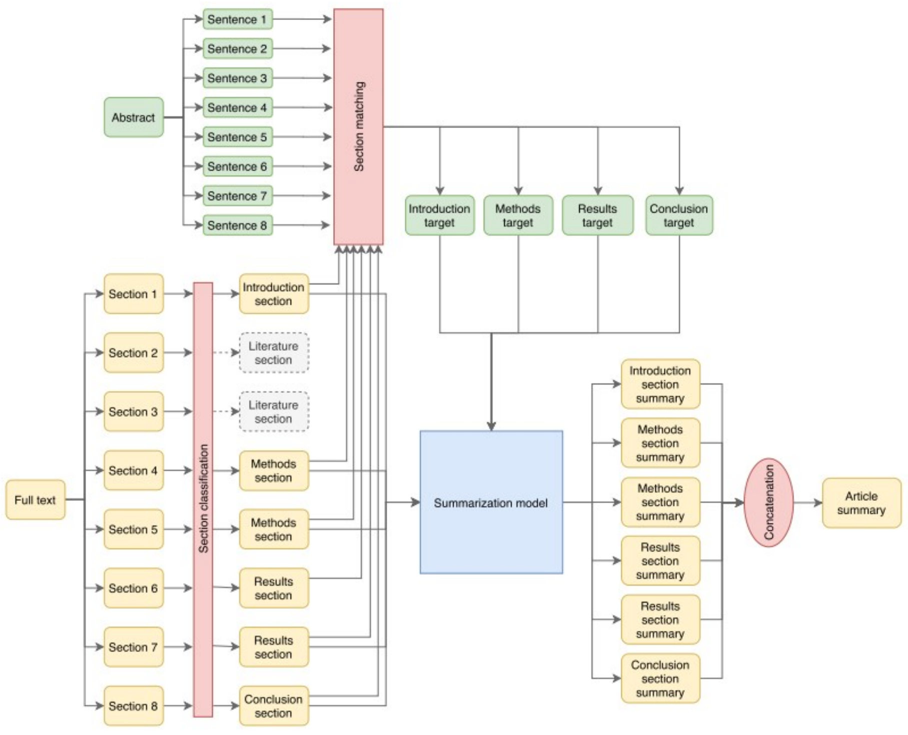


Figure 2.13: DANCER Approach (Source: Gidiotis et al. [15])

The DANCER PEGASUS model has proven effective, matching the performance of state-of-the-art models like BigBird-PEGASUS on arXiv and PubMed datasets, showcasing its competitiveness even with simpler architectures and making it a valuable option for scenarios with limited computational resources. However, its reliance on hard-coded keywords for section classification can be a drawback, as it may struggle with documents that have unconventional or non-standard section headers, indicating a need for further development to address these limitations more robustly.

The paper titled "Recursively Summarizing Enables Long-Term Dialogue Memory in Large Language Models" by Wang et al. [41] explores a novel method to enhance the long-term memory capabilities of Large Language Models (LLMs) like GPT-3.5 and Llama2 in the context of extended dialogues. The research focuses on addressing the common issue in which LLMs struggle to maintain consistency and recall past information accurately during long conversations.

The method proposed uses a recursive summarization technique to manage memory within LLMs. This approach prompts the language model to create summaries of smaller dialogue contexts first. These summaries serve as memory units that capture essential information from the initial part of the conversation. As the conversation progresses, the model is prompted to update the memory. This is achieved by combining the previously generated summaries (memory) with new contextual dialogue. This recursive updating process continues throughout the conversation, allowing the model to maintain a coherent memory of the dialogue context. The updated memory is then used by the LLM to generate responses that are consistent with the historical context of the conversation. By using the most recent memory, the model aims to produce responses that are relevant and contextually appropriate.

Figure 2.14 gives an overview of their method. At each step in the dialogue, "$u_T$" and "$r_T$" represent the user's and system's spoken contributions, respectively. When a session concludes, the memory from the previous session is updated to the current one by incorporating the complete context of that session.

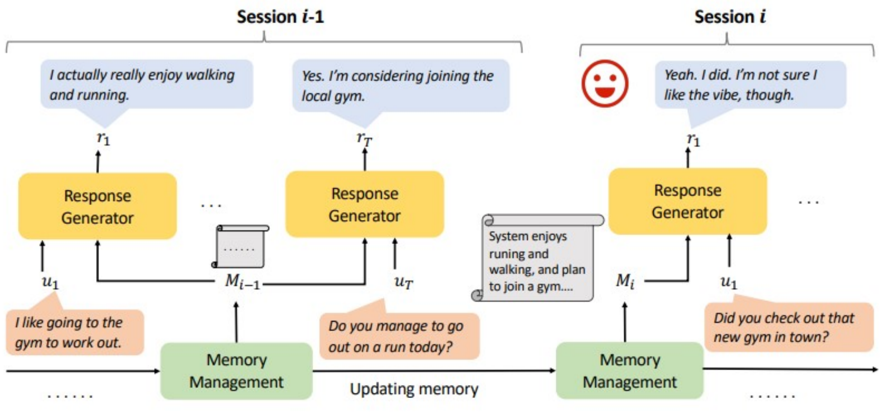

Session *i*-1
Session *i*
I actually really enjoy walking and running.
Yes. I'm considering joining the local gym.
Yeah. I did. I'm not sure I like the vibe, though.
$r_1$
$r_T$
$r_1$
Response Generator
Response Generator
Response Generator
System enjoys runing and walking, and plan to join a gym....
$u_1$
$M_{i-1}$
$u_T$
$M_i$
$u_1$
I like going to the gym to work out.
Do you manage to go out on a run today?
Did you check out that new gym in town?
Memory Management
Memory Management
Updating memory

Figure 2.14: Recursive Summarization for long-term dialogue memory (Source: Wang et al. [41])

The recursive summarization method is particularly useful in scenarios where the conversation spans multiple sessions or when the context extends beyond the model's input length capacity. The study mainly uses GPT-3.5-turbo and Llama2-7b as the base LLMs. These models are tested with and without the recursive summarization mechanism to evaluate its effectiveness. The experiments demonstrate that the recursive summarization technique significantly improves the consistency, coherence, and fluency of the generated responses, especially in longer dialogues. Although the method improves dialogue memory, it still suffers from issues such as hallucinations, where the model might generate inaccurate or fabricated details.

### 2.9.1 Comparison of strategies

The **Divide-and-Conquer** approach offers several advantages: it is scalable, making it suitable for very long texts that exceed the model's context window; it is efficient, as each chunk is processed independently, which can speed up the summarization process; and it is modular, allowing for parallel processing with multiple instances of the model working simultaneously. However, this method also has disadvantages, such as potential cohesion issues where the final summary may lack coherence or miss connections between sections due to the independent summarization of chunks. Additionally, there is a risk of information loss, as dividing the text can result in the omission of important context, particularly if the text does not have clear, natural segments. The DANCER method [15] specifically utilizes the structure of the document (e.g. sections) to divide the text. Chunks formed based on document sections may not always align with the token size limits of a model, potentially leading to truncation and loss of information. Furthermore, the reliance on pattern matching to identify sections is inherently unreliable, as section names can vary significantly across different documents. This method proves less effective when applied to small-context large language models (LLMs). In contrast, a recursive approach offers a more efficient solution by better fitting content within the model's context window.

The **Recursion approach** has several advantages, including better detail preservation, as key points are refined across iterations; enhanced coherence, since the entire text is reconsidered in each round; and the ability to produce multi-level summaries that gradually abstract the content. However, the Recursion approach also has drawbacks: it is computationally intensive as compared to the Divide-and-Conquer approach, requiring a new inference for each round, which can be slower and more resource-demanding. There is also a risk of over-summarization, where repeated iterations may oversimplify or omit crucial details if not carefully managed. Additionally, the recursive process is more complex to implement and fine-tune compared to a straightforward divide-and-conquer approach.
In the case of the paper "Recursively Summarizing Enables Long-Term Dialogue Memory in Large Language Models" by Wang et al. [41], the chunks are dialogue segments, which may or may not fit within a model's context window. Due to the sequential nature of the recursive strategy used, parallelization is not feasible either. This can lead to slower execution times and limited scalability.

The choice between Divide-and-Conquer and Recursion techniques seems to depend on the specific requirements of the summarization task, including text length, desired summary depth, computational resources, and the importance of coherence.

When writing a literature review, it is important to achieve a high level of abstraction while maintaining coherence across the entire summary. This requires a holistic consideration of the text at every stage to ensure all key points are retained and connected. Given these requirements, the recursive approach seems to be more suitable, as it allows the entire text to be summarized iteratively, refining the content in each round. This method ensures that the final summary is both comprehensive and cohesive, meeting the specific needs of a literature review. Given that we are utilizing a large language model (LLM) with a limited context window for summarizing long documents, the recursive technique offers greater flexibility and adaptability for our specific use case. It is also essential to develop an appropriate chunking strategy to ensure that the input text conforms to the model's token limitations.

# Chapter 3

# Design

This project aims to address the challenge of summarizing long research papers using an LLM with a small-context window, and finetuning it to the specific use case.
Given the inherent limitations of token size in standard LLMs, summarization strategies involving recursion are designed and explored to ensure that the entire content of the document is processed without truncation or significant information loss.

## 3.1 Model Selection

The *facebook/bart-large-cnn* model [4], which was pre-trained on the English language, and fine-tuned on CNN Daily Mail is selected as the base model. The *abisee/cnn_dailymail* dataset [11] is an English-language dataset containing over 300k unique news articles as written by journalists at CNN and the Daily Mail. The reason this model is chosen for this project is due to its strong performance in abstractive summarization tasks. It was particularly created for machine reading and comprehension and abstractive question-answering.

The BART model itself was introduced in the paper BART: Denoising Sequence-to-Sequence Pre-training for Natural Language Generation, Translation, and Comprehension by Lewis et al. [18]. BART, a denoising autoencoder for sequence-to-sequence tasks, is particularly well-suited for generating coherent and concise summaries. The BART model can handle a maximum input token size of 1,024 tokens, which is a lot more than models like GPT-2 and BERT models (512 tokens).

The CNN variant of BART makes it a robust starting point (due to being pre-trained on a summarization task) for further fine-tuning on the specific domain of research papers.

## 3.2 Dataset

The arXiv [2] and PubMed [30] datasets are widely used in the field of natural language processing (NLP) for tasks like summarization, particularly focusing on research papers. These datasets consist of scientific articles, with arXiv covering a broad range of subjects in fields like physics, computer science, and mathematics, while PubMed is specialized in biomedical and life sciences literature.

They offer both the full text of articles and their corresponding abstracts, which serve as reference summaries, guiding the models to produce concise, coherent summaries that capture the essential information from the full text. These datasets also provide extensive text corpora with rich technical content, enabling large language models (LLMs) to learn domain-specific language patterns, terminology, and the structure of scientific writing. The paired full texts and abstracts facilitate training on sequence-to-sequence tasks, allowing models to generate accurate and concise summaries. These datasets form the basis of the data for our project, supporting the development of models that efficiently process and summarize complex scientific literature.

## 3.3 Fine-Tuning Process

To generate high-quality summaries that align with the structure and style of research papers, the *facebook/bart-large-cnn* model is fine-tuned on a domain-specific dataset. This dataset consists of research papers and their corresponding handwritten summaries, which provide a rich source of training examples for the model. The fine-tuning process involves:

**Dataset Preparation:** The dataset is curated to include diverse research papers from various fields, ensuring the model can generalize across different types of content. The handwritten summaries are carefully selected to represent high-quality examples of concise and coherent summarization.

**Training Configuration:** The fine-tuning process is configured with a lower learning rate to prevent overfitting and ensure the model learns to generate summaries that closely match the handwritten examples. The training process is monitored with validation on a subset of the dataset to tune hyperparameters and avoid overfitting.

**Evaluation and Iteration:** After fine-tuning, the model is evaluated on a separate test set of research papers to assess its summarization quality. Based on the results, further adjustments to the training dataset may be made to improve performance.

## 3.4 Handling Long Documents

Given the token limit of BART (1024 tokens), summarizing lengthy research papers directly with the model would lead to significant truncation and potential loss of critical information. To address this, a dynamic recursive summarization approach is considered, incorporating memory optimization and efficient token management techniques to ensure comprehensive document coverage without information loss.

Using a recursive strategy with memoization - an optimization technique used to improve the efficiency of functions by storing the results of expensive function calls and reusing them when the same inputs occur again - for the summarization of research papers can effectively balance the need for detailed, coherent summaries with the challenges posed by long and complex texts. Unlike simple sequential or the Divide-and-Conquer approach (section-based chunking), which processes text in isolated chunks and can lead to coherence issues and loss of important context, the recursive approach reconsiders the entire text in each iteration. This iterative process allows for better preservation of key details and ensures that the summary remains cohesive and interconnected, which is crucial for accurately reflecting the structure and arguments of research papers.

**Document Segmentation:** The long research paper is divided into smaller segments that fit within the token limit of the BART model (e.g., 900 tokens). To maintain continuity and context across segments, text overlap is used.

**Recursive Summarization:** A recursive function is used to handle cases where the concatenated summaries from initial segments still exceed the model's token limit. This function progressively reduces the length of the input text through iterative summarization until the final summary fits within the specified token limit. The function dynamically adjusts the maximum summary length and segment size based on the length of the input text and the expected number of chunks, ensuring efficient handling of very long documents.

**Dynamic Adjustments:** The model dynamically adjusts the segmentation length and overlap between segments based on the content, allowing for better handling of varying document lengths and complexities. This ensures that the contextual flow of the document is maintained across segments, reducing the risk of losing important information that might span across segment boundaries.

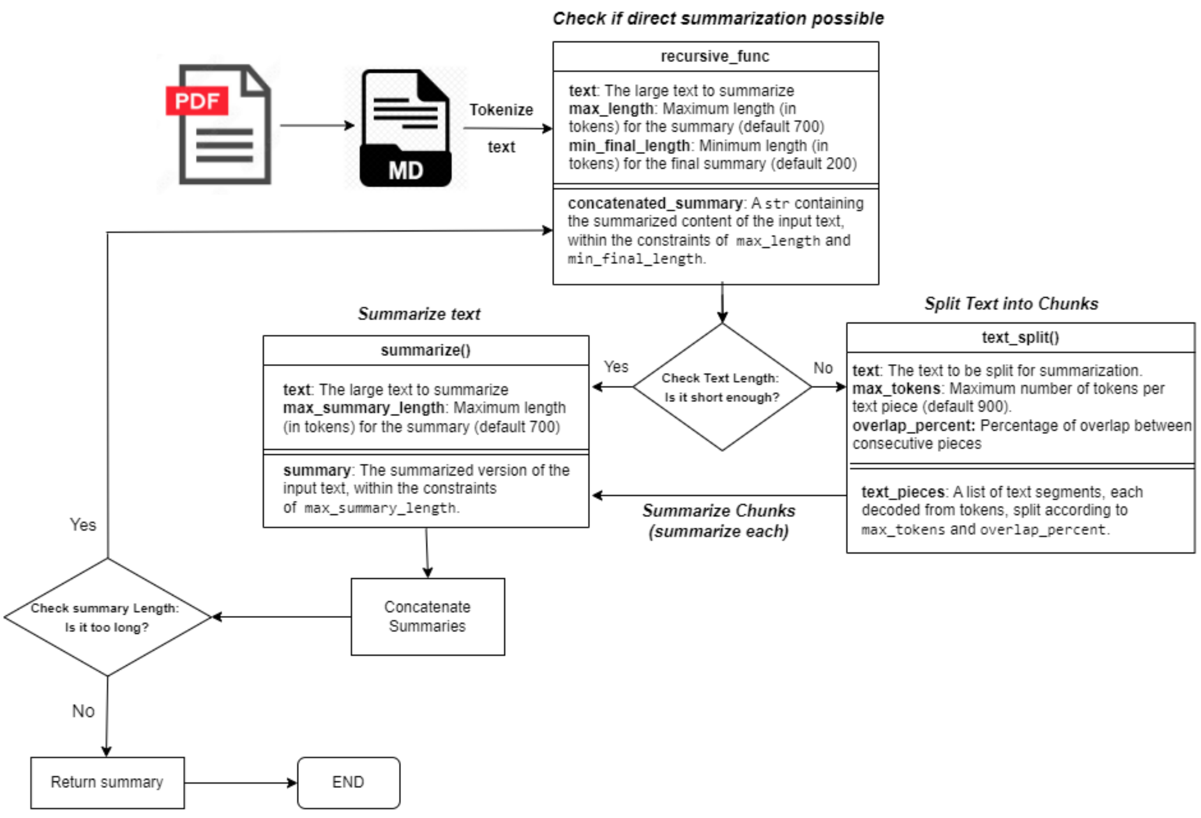

Check if direct summarization possible
recursive_func
text: The large text to summarize
max_length: Maximum length (in tokens) for the summary (default 700)
min_final_length: Minimum length (in tokens) for the final summary (default 200)
concatenated_summary: A str containing the summarized content of the input text, within the constraints of max_length and min_final_length.
PDF
MD
Tokenize text
Summarize text
summarize()
text: The large text to summarize
max_summary_length: Maximum length (in tokens) for the summary (default 700)
summary: The summarized version of the input text, within the constraints of max_summary_length.
Yes
Check Text Length: Is it short enough?
No
Split Text into Chunks
text_split()
text: The text to be split for summarization.
max_tokens: Maximum number of tokens per text piece (default 900).
overlap_percent: Percentage of overlap between consecutive pieces
text_pieces: A list of text segments, each decoded from tokens, split according to max_tokens and overlap_percent.
Summarize Chunks (summarize each)
Yes
Check summary Length: Is it too long?
Concatenate Summaries
No
Return summary
END

Figure 3.1: Adaptive Recursion Strategy

**Final Summary Generation:** The recursive function continues summarizing until the output summary is concise enough to fit within the model's token limit while still capturing the essential content of the document. If the final summary length is shorter than a minimum threshold, the model performs an additional summarization pass to enhance the output, ensuring that the final summary is both comprehensive and well-structured.

Figure 3.1 shows the flow chart of the operation of the adaptive recursion strategy. This approach effectively addresses the challenge of summarizing long research papers with a small-context LLM, ensuring that the entire document is processed and summarized without truncation, while also optimizing memory usage and computation time.

## 3.5 Evaluation Metrics

When evaluating summarization models, particularly for research papers and other lengthy texts, ROUGE scores (Recall-Oriented Understudy for Gisting Evaluation) are often preferred due to their specific advantages in measuring summary quality.

Comparison with Other Metrics:

BLEU Scores: While BLEU scores (Papineni et al. 2002 [31]) are commonly used for machine translation evaluation, they focus on precision and may not fully capture the recall aspect necessary for summarization tasks.

METEOR: This metric accounts for synonyms and paraphrasing but is less commonly used for summarization compared to ROUGE, and it may not be as effective in evaluating the completeness of content coverage (Banerjee et al., 2005 [4]).

CIDER: Designed for evaluating image captioning, CIDER focuses on content relevance but is not as directly applicable to summarization as ROUGE (Vedantan et al. [40]).

ROUGE scores assess the overlap between the generated summary and reference summaries (typically human-written). This overlap is indicative of how well the generated summary captures the essential content and key phrases of the original text. ROUGE's focus on n-gram overlap aligns well with human judgments of content relevance and coverage, making it a reliable measure of summary quality. Additionally, ROUGE metrics are primarily recall-oriented, which emphasizes the importance of including as much relevant content from the original text as possible.

By using various ROUGE metrics, a comprehensive evaluation of the summary's quality is achieved. The following ROUGE metrics will be used for model evaluation in this project:

1. **ROUGE-1 score:** To measure the overlap of unigrams (single words) between the generated summary and the reference summary. It reflects how much of the original content is captured.

2. **ROUGE-2 score:** Measures the overlap of bigrams (two consecutive words) between the generated summary and reference summaries. This metric helps assess how well the summary retains more complex and specific phrases, contributing to a more detailed understanding of the content.

3. **ROUGE-L score:** Evaluates the longest common subsequence between the generated and reference summaries. It is useful for assessing the summary's coherence and its ability to maintain the logical flow of the original text, ensuring that the generated summary reflects the main structure and sequence of the content.

This design framework ensures that the challenges posed by long document summarization are effectively addressed, resulting in high-quality, coherent summaries that can assist researchers and practitioners in quickly understanding the essence of lengthy academic papers.

# Chapter 4

# Implementation

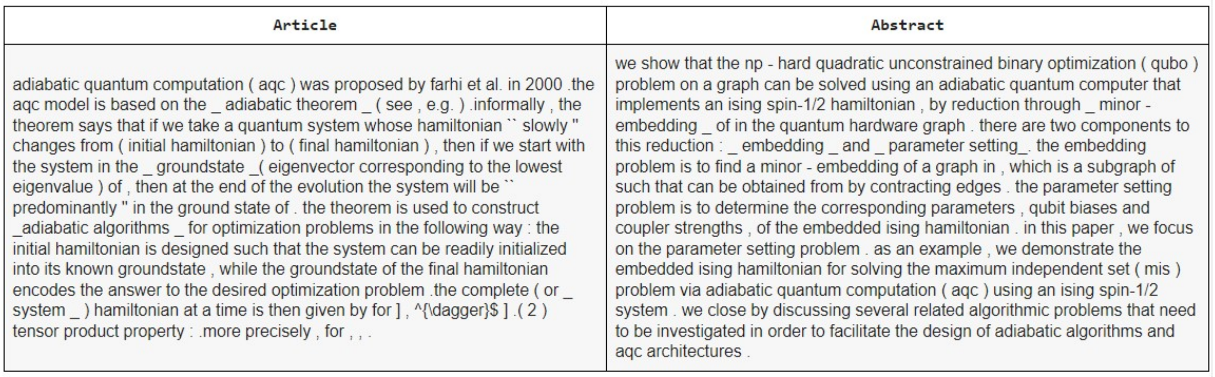
The implementation of this project involved fine-tuning a pretrained large language model (LLM) using a research paper dataset sourced from Hugging Face, a leading platform that provides a wide range of machine learning models, datasets, and tools for natural language processing tasks. The code was written in Python and executed on a Google Colab notebook, which provides a convenient environment for running machine learning experiments with the necessary computational resources. The fine-tuning process required a GPU to handle the intensive computations involved in training the LLM. Additionally, the user interface for the summarization application was developed using Streamlit, a Python framework that simplifies the creation of interactive web apps.

## 4.1 Loading Pretrained Model and Tokenizer

The 'facebook/bart-large-cnn' model and it's rerspective tokenizer is loaded using Hugging Face's transformers library. The bart-large-cnn model is basically the bart-large model which is finetuned on CNN news articles to generate summaries. The tokenizer is responsible for converting text into tokens that the model can process.

## 4.2 Dataset Preparation

The dataset comprises a curated collection of research papers, presented in the column labeled 'article,' alongside their corresponding expert-authored summaries, which are contained within the 'abstract' column. This dataset is intended to facilitate the supervised training of a transformer model, providing a robust foundation for the development and evaluation of natural language processing tasks related to text summarization. The inclusion of expert-generated summaries ensures the quality and relevance of the training data, thereby enhancing the model's capacity to learn and generalize from the provided examples. Figure 4.1 presents a sample of the dataset used in this study. The "Article" column contains the full text of the research papers, while the "Abstract" column provides the corresponding summaries. For the purpose of illustration, the text in the 'Article' column has been truncated to fit within the confines of the figure.



| Article | Abstract |
|---|---|
| adiabatic quantum computation ( aqc ) was proposed by farhi et al. in 2000 .the aqc model is based on the _ adiabatic theorem _ ( see , e.g. ) .informally , the theorem says that if we take a quantum system whose hamiltonian `` slowly " changes from ( initial hamiltonian ) to ( final hamiltonian ) , then if we start with the system in the _ groundstate _( eigenvector corresponding to the lowest eigenvalue ) of , then at the end of the evolution the system will be `` predominantly " in the ground state of . the theorem is used to construct _adiabatic algorithms _ for optimization problems in the following way : the initial hamiltonian is designed such that the system can be readily initialized into its known groundstate , while the groundstate of the final hamiltonian encodes the answer to the desired optimization problem .the complete ( or _ system _ ) hamiltonian at a time is then given by for ] , ^{\dagger}$ ] .( 2 ) tensor product property : .more precisely , for , , . | we show that the np - hard quadratic unconstrained binary optimization ( qubo ) problem on a graph can be solved using an adiabatic quantum computer that implements an ising spin-1/2 hamiltonian , by reduction through _ minor - embedding _ of in the quantum hardware graph . there are two components to this reduction : _ embedding _ and _ parameter setting_. the embedding problem is to find a minor - embedding of a graph in , which is a subgraph of such that can be obtained from by contracting edges . the parameter setting problem is to determine the corresponding parameters , qubit biases and coupler strengths , of the embedded ising hamiltonian . in this paper , we focus on the parameter setting problem . as an example , we demonstrate the embedded ising hamiltonian for solving the maximum independent set ( mis ) problem via adiabatic quantum computation ( aqc ) using an ising spin-1/2 system . we close by discussing several related algorithmic problems that need to be investigated in order to facilitate the design of adiabatic algorithms and aqc architectures . |

Figure 4.1: Sample record from dataset

The dataset preparation process involves several steps to ensure that the data is appropriately filtered and formatted for fine-tuning a language model.

### 4.2.1 Filter Function for Dataset

A custom filter function is defined to preprocess the dataset. The function checks two conditions for each example (research paper) in the dataset:

- **Empty Article Check:** The function first checks if the 'article' field is empty by removing any surrounding whitespace and evaluating the length of the string.

- **Token Length Check:** If the article is not empty, it is tokenized using the previously loaded tokenizer. The function then checks if the length of the tokenized article (in terms of the number of tokens) is 1024 or fewer. This ensures that the articles fed into the model are of manageable length and fit within the model's maximum input size.

Since the max input token length of the BART model is 1024, only the records where the 'articles' were within the token limit were kept, and the rest were discarded. Removing records that exceed the token limit from the dataset helps in maintaining the quality and coherence of summaries, avoids the pitfalls of truncation, and ensures that the model can be trained efficiently and effectively. This approach aligns with best practices for fine-tuning transformer models for summarization tasks.

### 4.2.2 Combining Datasets

Two separate datasets are loaded using Hugging Face's datasets library:

- *bakhitovd/ML_arxiv* dataset [25] containing research papers related to machine learning.
- *ccdv/pubmed-summarization* dataset [8] containing medical research papers.

These datasets provide a diverse range of articles from different domains, which will be useful for training a robust summarization model.
The two datasets into a single dataset using the 'concatenate_datasets' function. This merged dataset contains articles from both the machine learning and medical domains, broadening the scope of the data for fine-tuning the model.

### 4.2.3 Train Dataset

The combined dataset is then filtered using the custom filter function. This filtering is applied to all data splits (train, validation, test) within the combined dataset to ensure consistency across the dataset. The filtering process removes any articles that are either empty or exceed the 1024-token limit.

The final output is a filtered dataset that only includes articles that meet the specified criteria. This prepared dataset is pushed to Hugging Face (*shruti28062000/combined_filtered_dataset* [38]), and is now ready to be used for fine-tuning the pretrained summarization model.

### 4.2.4 Validation Dataset

This newly created dataset [38] has a total of 6088 records in the train split, 294 records in the validation split, and 314 in the test split. Since the validation and test sets are too small individually to provide reliable metrics, the validation and test splits are combined together to be the validation split. This will lead to better generalization during validation, as it provides more data to validate the model's performance.

### 4.2.5 Test Dataset

The records present in the Train dataset are removed from the original combined dataset (which has not been filtered according to token length) to create the Test dataset. Consequently, the Test dataset, called *shruti28062000/test_dataset* [37], consists of unseen data, ensuring that the model is evaluated on entirely new instances. This will also allow us to test the small-context model on very long text that exceeds the token limits, providing a rigorous assessment of its ability to summarize research papers.

## 4.3 Supervised Training

The process begins with transfer learning since a pretrained model, facebook/bart-large-cnn, is used. This model has been trained on a large, general dataset (news articles for summarization) and already possesses a strong understanding of language structure and summarization techniques.
However, in the code, not just a few layers or a new layer on top of the pretrained model are being trained. Instead, all the parameters of the model are being updated using the specific dataset of research papers. This means that every part of the model is being adjusted to better suit the task of summarizing research papers, making it a full fine-

tuning process.

During this supervised fine-tuning process, the model learns to generate concise summaries (abstracts) from longer inputs (articles) by seeing many examples of such pairs in the training data. This approach ensures that the model not only retains its general summarization abilities but also becomes more specialized in handling the structure and language of academic papers.

### 4.3.1 Model Configuration

This section outlines the architecture and structural components of the model. The model configuration directly influences the model's capacity to learn and represent complex patterns in the data. A poor configuration may lead to underfitting or overfitting. Model configuration is usually decided based on the problem at hand and may involve experimentation, but it's often guided by domain knowledge or prior research.

| Configuration | Values |
|---|---|
| Attention type | $n^2$ |
| Max. encoder input length | 1024 |
| Max. decoder input length | 1024 |
| Num. of heads | 16 |
| Num. layers | 12 |
| Activation function | GeLU |

Table 4.1: Model Configuration Parameters

Table 4.1 shows the model configurations used for training the model, detailing the architectural parameters and settings that define the structure and functionality of the model during the training process.

**Attention Type: $\mathbf{n}^2$**

The $n^2$ attention mechanism is a standard self-attention mechanism used in transformers, where each token attends to every other token in the sequence. This results in a quadratic computational complexity with respect to the sequence length. Despite the high computational cost, this method was chosen because it captures comprehensive dependencies between tokens, crucial for tasks requiring contextual understanding, such as natural language processing. $n^2$ is the traditional self-attention method, ensuring the model can fully capture context across sequences.

**Max. Encoder Input Length: 1024**

This defines the maximum number of tokens that the encoder can process in a single sequence. The BART model has a token size limit of 1024 tokens.

**Max. Decoder Input Length: 1024**

This parameter limits the maximum number of tokens the decoder can handle during sequence generation. Matching the encoder's input length ensures consistency and allows the model to generate outputs as long as the inputs, which is particularly useful for tasks like text generation or translation.

**Number of Heads: 16**

This refers to the number of attention heads in the multi-head attention mechanism, allowing the model to focus on different parts of the input sequence simultaneously. Increasing the number of heads improves the model's ability to capture various aspects of the input at different positions, enhancing the model's overall expressiveness. 16 heads offer a good balance between model capacity and computational efficiency, providing robust attention without excessive overhead.

**Number of Layers: 12**

This is the number of transformer layers in the model, where each layer consists of multi-head attention and feedforward networks. 12 layers are a commonly used configuration in models like BART, providing a well-established balance between depth and performance.

**Activation Function: GeLU**

The Gaussian Error Linear Unit (GeLU) is a non-linear activation function that is smoother than the standard ReLU, allowing for more nuanced learning. GeLU is preferred in transformer architectures because it improves the model's capacity to learn complex patterns without introducing hard thresholds, as seen in ReLU. GeLU has been empirically proven to perform well in many NLP tasks, making it a standard choice in transformers.

### 4.3.2 Hyperparameters

Hyperparameters play a crucial role in shaping the training dynamics of the model, directly influencing its performance and convergence. Hyperparameters are set before training begins and can be tuned throughout the training process. They are often optimized through search techniques

In this configuration, specific hyperparameters, as shown in Table 4.2, are chosen to collectively guide the model's learning process, ensuring robust and effective training.

| Hyperparameter | values |
|---|---|
| Batch size | 2 |
| Gradient accumulation steps | 2 |
| Optimizer | AdamW |
| Learning rate | 2e-5 |
| Epochs | 4 |
| Weight decay | 0.01 |
| Evaluation strategy | Epoch |
| Eval accumulation steps | 2 |

Table 4.2: Hyperparameter Settings for Model Training

**Batch size: 2**

The batch size determines the number of samples the model processes before updating its weights. A smaller batch size means lower memory usage.

**Gradient accumulation steps: 2**

Gradient accumulation steps allow the model to simulate a larger batch size by accumulating gradients over several steps before performing an update. With a batch size of 2, accumulating gradients over 2 steps effectively simulates a batch size of 4, enabling the model to benefit from a larger effective batch size without needing extra memory.

**Optimizer: AdamW**

AdamW is a variant of the Adam optimizer that includes weight decay to prevent overfitting. It is widely used for its efficiency and performance on large-scale models. AdamW is chosen for its ability to handle sparse gradients and to maintain performance while controlling overfitting through weight decay.

**Learning rate: 2e-5**

The learning rate controls how much the model's weights are adjusted during training. A smaller learning rate, like 2e-5, allows for more fine-grained updates. This learning rate is typical for fine-tuning pre-trained models like Transformers, ensuring stability in training while still making meaningful updates to the model weights.

**Epochs: 4**

The number of epochs determines how many times the model sees the entire dataset during training. More epochs can lead to better learning but also increase the risk of overfitting.

The initial fine-tuning of the model was conducted over 2 epochs, which resulted in mediocre performance. To improve these outcomes, the model was subsequently trained for an additional 2 epochs, bringing the total to 4 epochs. This extended training significantly enhanced the model's performance, yielding very good results. However, when the model was fine-tuned for an additional epoch beyond this, an increase in validation loss was observed, indicating potential overfitting.

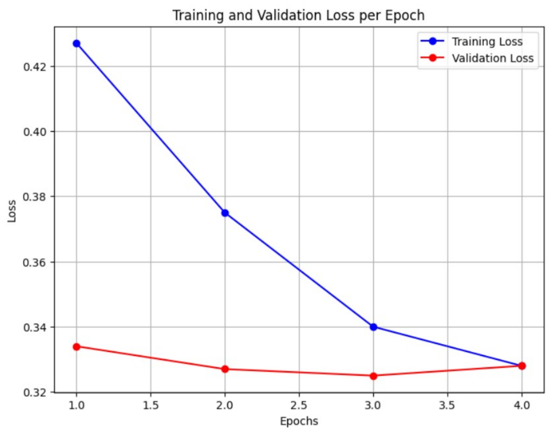
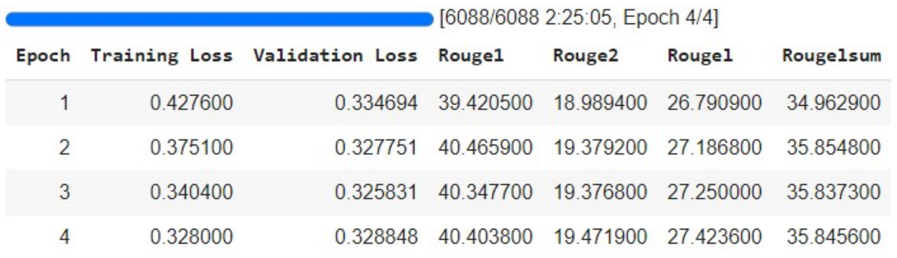


Figure 4.2: Training and Validation Loss per Epoch

Figure 4.2 illustrates the Training and Validation Loss per Epoch. The decreasing training loss indicates effective learning by the model. The initial decrease in validation loss suggests good generalization.



[6088/6088 2:25:05, Epoch 4/4]

| Epoch | Training Loss | Validation Loss | Rouge1 | Rouge2 | Rougel | Rougelsum |
|---|---|---|---|---|---|---|
| 1 | 0.427600 | 0.334694 | 39.420500 | 18.989400 | 26.790900 | 34.962900 |
| 2 | 0.375100 | 0.327751 | 40.465900 | 19.379200 | 27.186800 | 35.854800 |
| 3 | 0.340400 | 0.325831 | 40.347700 | 19.376800 | 27.250000 | 35.837300 |
| 4 | 0.328000 | 0.328848 | 40.403800 | 19.471900 | 27.423600 | 35.845600 |

Figure 4.3: Training Progress and Performance Metrics

Figure 4.3 is a snapshot of the Training Progress and Performance Metrics. It can be observed that ROUGE score are improving, and both training loss and validation loss decrease consistently until the 4th epoch, at which point the validation loss begins to rise. This increase in validation loss indicates that the model is starting to overfit the training data, as it performs well on the training set but struggles to generalize to unseen validation data. Therefore, the optimal model performance was achieved after 4 epochs of fine-tuning. The entire training process took 2 hours and 25 minutes.

**Weight decay: 0.01**

Weight decay adds a penalty to the loss function for large weights, helping to regularize the model and prevent overfitting. A value of 0.01 is a common choice that provides sufficient regularization without overly penalizing the model's weights.

**Evaluation strategy: Epoch**

This parameter defines how often the model is evaluated on the validation set. Evaluating after each epoch allows for periodic checks on model performance.

**Eval accumulation steps: 2**

Similar to gradient accumulation, this allows the model to accumulate predictions over several steps before computing the evaluation metrics. Accumulating over 2 steps can help in managing memory usage during evaluation, which might be necessary if the evaluation involves large batches or complex operations.

**Predict with generate: True**

This indicates that the model uses a generation mechanism (such as beam search) during prediction, which is common in sequence generation tasks. Enabling this ensures that the evaluation reflects the model's performance in actual use cases, like text generation.

**Mixed Precision (fp16): True**

Mixed precision training uses half-precision (fp16) instead of full-precision (fp32) for some operations, which can speed up training and reduce memory usage. This option is enabled to improve training efficiency, particularly on modern GPUs, without sacrificing model accuracy.

## 4.4 Dynamic Recursive Functions

### 4.4.1 Adaptive Recursive Chunking strategy

A recursive function is designed to handle the summarization of large texts that exceed the model's maximum input token length. The BART model has a maximum token limit of 1024 tokens, which means that very long texts need to be broken down into smaller pieces. However, simply splitting and summarizing these pieces may not capture the full context of the original text. The recursive approach ensures that the summary is refined iteratively until it meets the desired length constraints.

Functions used:

***summarize***

The summarize function is designed to produce a concise summary of a given text using a trained language model. The function begins by taking a text input and converting it into a sequence of tokens with the *tokenizer.encode* method. This method generates a PyTorch tensor, making the text suitable for processing by the model. To ensure compatibility with the model's constraints, the token count is limited to a maximum of 1024. If the input text exceeds this limit, it is truncated accordingly.

**Algorithm 1** summarize

```
FUNCTION summarize(text, max_summary_length=700):
    Step 1: Encode the input text into tokens
    inputs = tokenizer.encode(text, return_tensors="pt", max_length=1024, truncation=True)

    Step 2: Move the tokenized input to the appropriate device (CPU/GPU)
    inputs = move_to_device(inputs, device)

    Step 3: Generate the summary using the model
    summary_ids = model.generate(inputs,
                                 max_length=max_summary_length,
                                 min_length=(max_summary_length / 5),
                                 length_penalty=8.0,
                                 num_beams=4,
                                 early_stopping=True)

    Step 4: Decode the generated token IDs into readable text
    summary = tokenizer.decode(summary_ids[0], skip_special_tokens=True)

    Step 5: Return the final summarized text
        RETURN summary
END FUNCTION
```

Once the input text has been tokenized, the function employs the *model.generate* method to create the summary. This method utilizes several parameters to control the output: *inputs* represents the encoded text; *max_length* defines the maximum allowed length for the summary, set to a predetermined maximum summary length; *min_length* specifies the minimum length, set to one-fifth of the maximum summary length, to prevent the generation of overly brief summaries. Additionally, a length penalty is set to 8.0 to discourage excessively long outputs and promote brevity.

The *model.generate* method also incorporates the *num_beams* parameter, set to 4, to use beam search in refining the summary. Beam search evaluates multiple potential outputs and selects the most appropriate one, enhancing the overall quality of the summary. Furthermore, the *early_stopping* parameter is activated to halt the search process once an optimal summary has been identified, thereby improving computational efficiency.

After the summary is generated, its token IDs are converted back into human-readable text using the *tokenizer.decode* method, which omits special tokens (e.g., [CLS], [SEP]) that are not relevant for the output summary to maintain clarity. The final output of the function is the generated summary, which is returned to the user in a concise and readable format.

### ***text_split***

The text_split function is designed to partition a lengthy text into smaller, more manageable chunks, facilitating processing in segments. This approach is particularly advantageous for handling texts that exceed the token limit of the model. The function accepts three parameters: *text*, *max_tokens*, and *overlap_percent*. The text parameter represents the input text to be divided, while max_tokens (set to 900) denotes the maximum number of tokens that each chunk may contain. The overlap_percent (set to 15) specifies the percentage of overlap between consecutive chunks, which helps maintain continuity and context across the text splits.

The function begins by tokenizing the input text using the tokenizer.tokenize method, which breaks down the text into individual tokens without converting them into token IDs. This step ensures the text is decomposed into smaller, more manageable components. Subsequently, the number of tokens that overlap between chunks is calculated using the formula

$$token\_overlap = int(max\_tokens * overlap\_percent/100)$$

This calculation determines how much each chunk will overlap with the next, thereby preserving the contextual flow of the original text.

After determining the overlap, the function uses list comprehension to create chunks by iterating over the tokens with a stride that accounts for the overlap, generating chunks of *max_tokens* length. Each chunk is converted to token IDs, decoded into text, skipping special tokens. The function returns a list of text chunks, ready for individual summarization, enabling efficient processing of large texts.

**Algorithm 2** text_split

```
FUNCTION text_split(text, max_tokens=900, overlap_percent=15):

    Step 1: Tokenize the input text into individual tokens
            tokens = tokenizer.tokenize(text)

    Step 2: Calculate the number of overlapping tokens between chunks
            token_overlap = FLOOR(max_tokens * overlap_percent / 100)

    Step 3: Initialize an empty list to store the pieces of text
            pieces = [ ]

    Step 4: Loop to create chunks of tokens with specified overlap
            FOR i FROM 0 TO LENGTH(tokens) STEP (max_tokens - token_overlap):
                chunk = tokens[i : i + max_tokens]
                ADD chunk TO pieces

    Step 5: Convert each chunk of tokens back to text
            text_pieces = [ ]
            FOR each chunk IN pieces:
                chunk_ids = tokenizer.convert_tokens_to_ids(chunk)
                text_chunk = tokenizer.decode(chunk_ids, skip_special_tokens=True)
                ADD text_chunk TO text_pieces

    Step 6: Return the list of text pieces
            RETURN text_pieces

END FUNCTION
```

#### *recursive_func*

The *recursive_func* function is designed to handle the summarization of large texts by recursively breaking them down and summarizing each part until the desired length is achieved. This recursive approach is especially beneficial for processing texts that are significantly longer than the target summary length, ensuring that the final output is both concise and maintains the essential content of the original text. The function takes four parameters: *text*, *max_length*, *min_final_length*, and *recursion_level*. The text parameter is the original text that needs to be summarized, while *max_length* specifies the maximum length of the final summary in tokens, and *min_final_length* defines the minimum acceptable length to retain sufficient detail. The *recursion_level* parameter tracks the depth of recursion, which is useful for monitoring and debugging purposes.

The function begins by incrementing the *recursion_level* to keep track of how deep the recursion goes with each call. It then tokenizes the input text and calculates the *expected_count_of_chunks*, which is the estimated number of chunks needed based on the *max_length*. If the number of chunks is less than one, indicating that the text is already shorter than the desired length, the function directly calls the summarize function to produce a summary of the text. Otherwise, it calculates the *max_chunk_length* to determine the size of each chunk, with a slight buffer to account for rounding issues.

The function then uses the *text_split* function to divide the text into chunks of this length and generates a summary for each chunk using a list comprehension. The generated summaries are concatenated into a single text, which is re-tokenized to check its length. If the concatenated summary exceeds the max_length, the function calls itself recursively with the concatenated summary as the new input. If the summary length is within the acceptable range, the function checks if it is too short. If it is, the summarize function is called again to expand it slightly. Otherwise, the function returns the concatenated summary, ensuring it meets the specified length requirements. This iterative process allows the function to produce a final summary that is appropriately concise while retaining the key points of the original text.

**Algorithm 3** recursive_func

```
FUNCTION recursive_func(text, max_length=700, min_final_length=200, recursion_level=0):

    Step 1: Increment the recursion level to track the depth
            recursion_level = recursion_level + 1
    Step 2: Tokenize the input text to get the total number of tokens
            tokens = tokenizer.tokenize(text)
    Step 3: Calculate the expected number of chunks based on the max_length
        expected_count_of_chunks = LENGTH(tokens) / max_length
    Step 4: If the text is already short enough, summarize it directly
            IF expected_count_of_chunks < 1 THEN
                RETURN summarize(text, max_summary_length=max_length)
            END IF
    Step 5: Calculate the maximum length for each chunk
      max_chunk_length = FLOOR(LENGTH(tokens) / expected_count_of_chunks) + 2
    Step 6: Split the text into chunks using the text_split function
            pieces = text_split(text, max_tokens=max_chunk_length)
    Step 7: Generate summaries for each chunk
            summaries = [ ]
            FOR each piece IN pieces DO
                summary = summarize(piece,
                max_summary_length=(max_chunk_length// 3 * 2))
                ADD summary TO summaries
            END FOR
    Step 8: Concatenate all summaries into a single text
            concatenated_summary = JOIN(summaries, " ")
    Step 9: Re-tokenize the concatenated summary to check its length
            tokens = tokenizer.tokenize(concatenated_summary)
    Step 10: Check if the concatenated summary exceeds the max_length
            IF LENGTH(tokens) > max_length THEN
                RETURN recursive_func(concatenated_summary,
                max_length=max_length,
                min_final_length=min_final_length,
                recursion_level=recursion_level)
            ELSE
    Step 11: If the concatenated summary is too short, re-summarize it
                IF LENGTH(tokens) < min_final_length THEN
                RETURN summarize(concatenated_summary,
                max_summary_length=max_length)
                ELSE
    Step 12: Return the concatenated summary if it meets length requirements
                    RETURN concatenated_summary
                END IF
            END IF
END FUNCTION
```

This adaptive recursive strategy implements a sophisticated approach for summarizing large texts. The code has many aspects such as:

- **Dynamic Length Adjustment:** The summarize function dynamically adjusts the maximum length of the summary. It also sets a minimum length dynamically based on a fraction of *max_summary_length* to ensure summaries are concise but informative.
- **Adaptive Recursion:** The *recursive_func* function adapts its behavior based on the length of the input text and the resulting summaries. It decides whether to summarize directly or split the text into smaller chunks for recursive summarization, demonstrating dynamic adaptation to different text sizes.
- **Dynamic Chunk Size Calculation:** In the *recursive_func* function, the chunk size is calculated based on the total number of tokens and the desired number of chunks. This ensures efficient processing by avoiding excessively large or small chunks.
- **Recursive Summarization:** The *recursive_func* function calls itself recursively to handle texts that are too large for a single summarization pass. It breaks down the text into chunks, summarizes each chunk, and then attempts to summarize the concatenated summaries.

### 4.4.2 Adaptive Recursive Text Summarization with Memoization and Parallel Processing

Several modifications were made to the adaptive recursive function to enhance its performance, efficiency, and scalability in handling text summarization tasks.

#### Memoization

It is a technique used to speed up function calls by storing the results of expensive function calls and returning the cached result when the same inputs occur again. This is particularly useful in recursive functions or functions that are called multiple times with the same arguments.

The *@lru_cache* decorator is used to enable memoization for the summarize function. lru_cache stands for Least Recently Used Cache. It caches the results of summarization for specific text inputs. The idea is to avoid recalculating summaries for repeated inputs.

**Algorithm 4** summarize

```
FUNCTION summarize(text, max_summary_length=700):

    Step 1: Check if the result for this input is already cached
            IF result exists in cache for (text, max_summary_length) THEN
                RETURN cached result

    Step 2: Encode the input text into tokens and prepare input tensor
            inputs = Tokenizer.encode(text, return_tensors="pt", max_length=1024,
truncation=True)
            inputs = Move inputs to the appropriate device (CPU/GPU)

    Step 3: Generate summary using the model
            summary_ids = Model.generate(
                inputs,
                max_length=max_summary_length,
                min_length=INT(max_summary_length / 5),
                length_penalty=8.0,
                num_beams=4,
                early_stopping=True
            )

    Step 4: Decode the generated tokens into text
            summary = Tokenizer.decode(summary_ids[0], skip_special_tokens=True)

    Step 5: Store the result in the cache
            Cache the result for (text, max_summary_length) as summary

    Step 6: Return the generated summary
            RETURN summary

END FUNCTION
```

### Parallel Processing

The use of *ThreadPoolExecutor* enables parallel processing of text chunks during summarization. This allows multiple chunks to be processed simultaneously rather than sequentially

.

**Algorithm 5** parallel_summarize

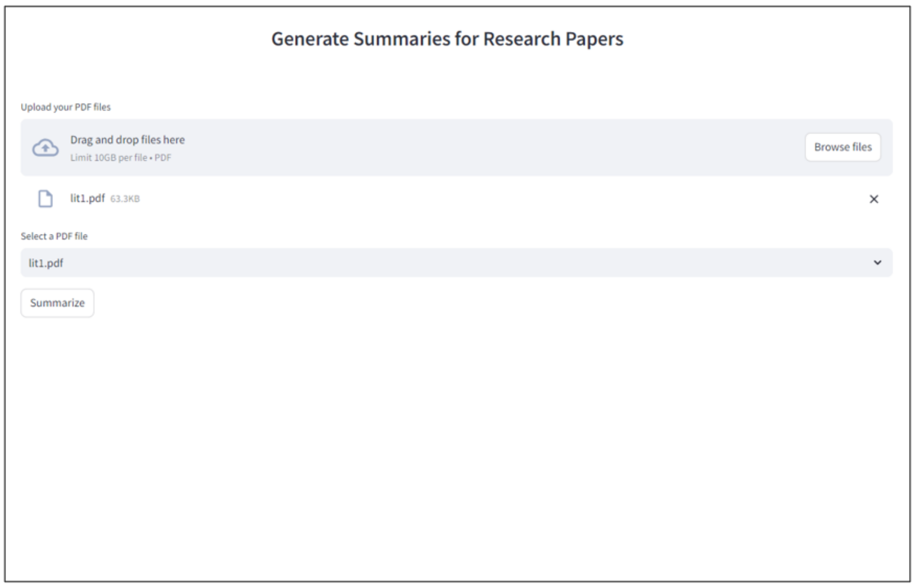
```
Function parallel_summarize(text, max_summary_length=700, max_tokens=900, overlap_percent=15)

    Step 1: Split the text into chunks with overlap
      pieces = text_split(text, max_tokens, overlap_percent)
    Step 2: Initialize a ThreadPoolExecutor for parallel processing
      Initialize ThreadPoolExecutor as executor
    Step 3: Use executor to map each text chunk to the summarize function in parallel
      summaries = executor.map(summarize_function, each piece in pieces, max_summary_length)
    Step 4: Convert the executor's output to a list of summaries
      summaries = Convert executor output to list
    Step 5: Concatenate the individual summaries into a single string
      concatenated_summary = Join summaries with space separator
    Step 6: Return the concatenated summary
      Return concatenated_summary
End Function

Function summarize_function(piece, max_summary_length)
    Generate a summary for the given text piece
      summary = summarize(piece, max_summary_length)
    Return summary
End Function

Function text_split(text, max_tokens, overlap_percent)
    Tokenize the input text into individual tokens
      tokens = Tokenizer.tokenize(text)
    Calculate the number of overlapping tokens between chunks
      token_overlap = Floor(max_tokens * overlap_percent / 100)
    Initialize an empty list to store the pieces of text
      pieces = [ ]
    Loop to create chunks of tokens with specified overlap
      For i from 0 to Length(tokens) step (max_tokens - token_overlap)
          chunk = tokens[i : i + max_tokens]
          Add chunk to pieces
    Convert each chunk of tokens back to text
      text_pieces = [ ]
      For each chunk in pieces
          chunk_ids = Tokenizer.convert_tokens_to_ids(chunk)
          text_chunk = Tokenizer.decode(chunk_ids, skip_special_tokens=True)
          Add text_chunk to text_pieces
    Return text_pieces

End Function
```

## 4.5 PDF Processing

The process of PDF text extraction and conversion begins with reading the PDF file to extract its content. The library used for this process is PyMuPDF, which is accessed through the Python module fitz. The specific function used to extract text from the PDF is *get_text(”text”)*. This function reads the text from each page of the PDF and returns it in readable format. The extracted text from all pages is then concatenated into a single string to create a comprehensive plain-text version of the entire document. Non-text features like images, tables, equations etc. get removed during this cleaning process.
After the text is extracted, it is converted into a more manageable format. This involves taking the combined text and preparing it for further analysis or processing. The conversion process ensures that the text is in a suitable format for tasks such as summarization, text cleaning, or other forms of text analysis, providing a clear and straightforward way to handle large volumes of information from PDF files.

## 4.6 User Interface

The user interface (UI) of this research paper summarizer application, developed using Streamlit, is structured to facilitate the uploading of PDF files and the generation of text summaries for research papers. Users can upload multiple research papers, which are then listed by name in a dropdown menu that allows the selection of a specific PDF file for summarization. The process is initiated by a ”Summarize” button, which triggers the text extraction and summarization function for the chosen PDF file. Figure 4.4 illustrates the UI of the application.
Upon activation of the summarization process, the page is organized into two columns: the left column displays the uploaded PDF file using an iframe, enabling users to view the document directly within the application via a base64-encoded data source, while the right column presents the generated summary (as illustrated in Figure 4.5).



Generate Summaries for Research Papers

Upload your PDF files

Drag and drop files here
Limit 10GB per file • PDF

Browse files

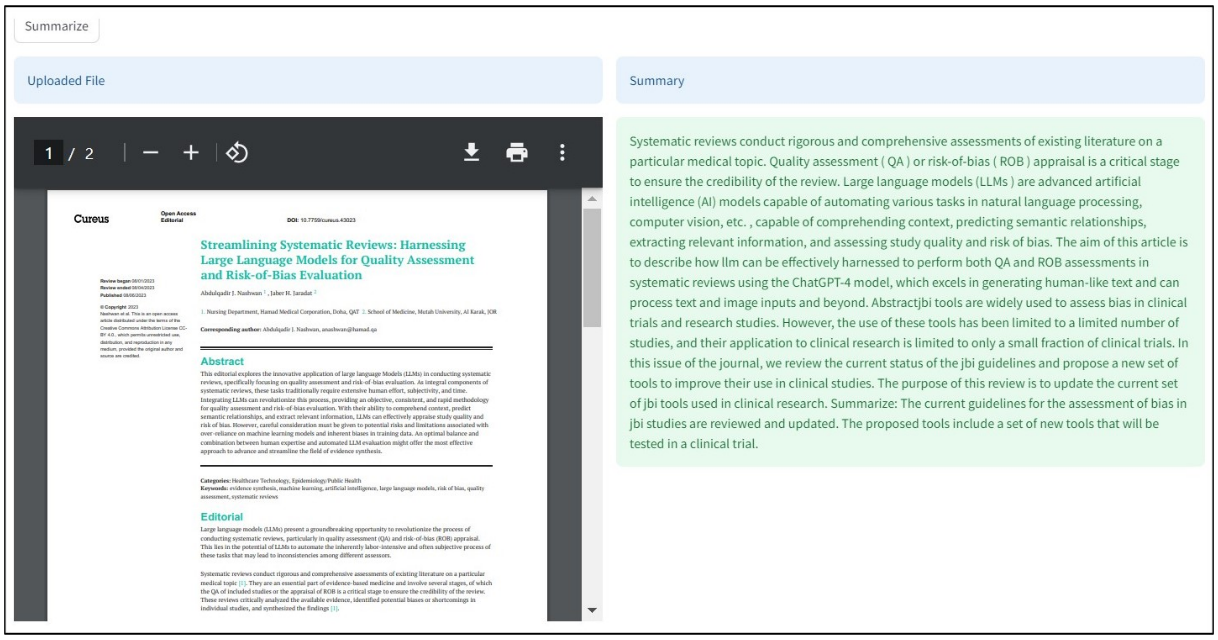
lit1.pdf 63.3KB

Select a PDF file

lit1.pdf

Summarize

Figure 4.4: User Interface



Summarize

Uploaded File

1 / 2

Cureus

Open Access
Editorial

DOI: 10.7759/cureus.43023

Streamlining Systematic Reviews: Harnessing Large Language Models for Quality Assessment and Risk-of-Bias Evaluation

Review began 08/01/2023
Review ended 08/04/2023
Published 08/06/2023

© Copyright 2023
Nashwan et al. This is an open access article distributed under the terms of the Creative Commons Attribution License CC-BY 4.0., which permits unrestricted use, distribution, and reproduction in any medium, provided the original author and source are credited.

Abdulqadir J. Nashwan [1] , Jaber H. Jaradat [2]

1. Nursing Department, Hamad Medical Corporation, Doha, QAT 2. School of Medicine, Mutah University, Al Karak, JOR

Corresponding author: Abdulqadir J. Nashwan, anashwan@hamad.qa

Abstract

This editorial explores the innovative application of large language Models (LLMs) in conducting systematic reviews, specifically focusing on quality assessment and risk-of-bias evaluation. As integral components of systematic reviews, these tasks traditionally require extensive human effort, subjectivity, and time. Integrating LLMs can revolutionize this process, providing an objective, consistent, and rapid methodology for quality assessment and risk-of-bias evaluation. With their ability to comprehend context, predict semantic relationships, and extract relevant information, LLMs can effectively appraise study quality and risk of bias. However, careful consideration must be given to potential risks and limitations associated with over-reliance on machine learning models and inherent biases in training data. An optimal balance and combination between human expertise and automated LLM evaluation might offer the most effective approach to advance and streamline the field of evidence synthesis.

Categories: Healthcare Technology, Epidemiology/Public Health
Keywords: evidence synthesis, machine learning, artificial intelligence, large language models, risk of bias, quality assessment, systematic reviews

Editorial

Large language models (LLMs) present a groundbreaking opportunity to revolutionize the process of conducting systematic reviews, particularly in quality assessment (QA) and risk-of-bias (ROB) appraisal. This lies in the potential of LLMs to automate the inherently labor-intensive and often subjective process of these tasks that may lead to inconsistencies among different assessors.

Systematic reviews conduct rigorous and comprehensive assessments of existing literature on a particular medical topic [1]. They are an essential part of evidence-based medicine and involve several stages, of which the QA of included studies or the appraisal of ROB is a critical stage to ensure the credibility of the review. These reviews critically analyzed the available evidence, identified potential biases or shortcomings in individual studies, and synthesized the findings [1].

Summary

Systematic reviews conduct rigorous and comprehensive assessments of existing literature on a particular medical topic. Quality assessment ( QA ) or risk-of-bias ( ROB ) appraisal is a critical stage to ensure the credibility of the review. Large language models (LLMs ) are advanced artificial intelligence (AI) models capable of automating various tasks in natural language processing, computer vision, etc. , capable of comprehending context, predicting semantic relationships, extracting relevant information, and assessing study quality and risk of bias. The aim of this article is to describe how llm can be effectively harnessed to perform both QA and ROB assessments in systematic reviews using the ChatGPT-4 model, which excels in generating human-like text and can process text and image inputs and beyond. Abstractjbi tools are widely used to assess bias in clinical trials and research studies. However, the use of these tools has been limited to a limited number of studies, and their application to clinical research is limited to only a small fraction of clinical trials. In this issue of the journal, we review the current status of the jbi guidelines and propose a new set of tools to improve their use in clinical studies. The purpose of this review is to update the current set of jbi tools used in clinical research. Summarize: The current guidelines for the assessment of bias in jbi studies are reviewed and updated. The proposed tools include a set of new tools that will be tested in a clinical trial.

Figure 4.5: Summary Display

# Chapter 5

# Evaluation

The evaluation process involves generating summaries for various articles of differing lengths using a text summarization model. The performance is then evaluated using ROUGE-1, ROUGE-2, and ROUGE-L scores to measure the quality of the summaries in terms of content overlap and coherence compared to reference summaries. Additionally, the generation time is recorded to assess the model's efficiency. By examining these metrics across different examples, we can gain insights into the model's strengths and weaknesses, particularly in handling different text lengths and maintaining summary quality while being computationally efficient.

The BART-large-CNN model, which was fine-tuned on the research paper dataset, was evaluated using both the adaptive recursive summarization strategy and the enhanced version incorporating memoization and parallel processing. The performance results of these approaches will be compared with those of an LED Longformer model.

## 5.1 Finetuned BART-large-CNN model

### 5.1.1 Adaptive Recursive approach

Table 5.1 is a snapshot of the generated results that provides a detailed overview of the performance metrics for a text summarization model applied to a sample of 10 research papers from a test dataset. The metrics include ROUGE scores (ROUGE-1, ROUGE-2, and ROUGE-L) and the time taken to generate summaries.

| Example | Article Length (words) | ROUGE-1 | ROUGE-2 | ROUGE-L | Time (s) |
|---|---|---|---|---|---|
| 0 | 3668 | 0.461 | 0.260 | 0.271 | 33.235 |
| 1 | 2694 | 0.192 | 0.081 | 0.112 | 42.209 |
| 2 | 3862 | 0.399 | 0.172 | 0.210 | 39.098 |
| 3 | 1140 | 0.256 | 0.068 | 0.156 | 8.193 |
| 4 | 1607 | 0.181 | 0.073 | 0.120 | 8.899 |
| 5 | 6571 | 0.308 | 0.074 | 0.171 | 115.907 |
| 6 | 1743 | 0.193 | 0.110 | 0.158 | 7.900 |
| 7 | 3329 | 0.359 | 0.142 | 0.189 | 20.631 |
| 8 | 8774 | 0.410 | 0.153 | 0.209 | 62.211 |
| 9 | 880 | 0.377 | 0.120 | 0.182 | 3.834 |
| Total | 34268 | 0.314 | 0.125 | 0.178 | 342.118 |

Table 5.1: Test results using Adaptive Recursion

The results of the fine-tuned BART-large-CNN model implemented with an adaptive recursive function and optimized using the Accelerate library indicate a strong performance in text summarization tasks. The model achieved an average ROUGE-1 score of around 0.314, showing that it effectively captures the main content of the articles.
The ROUGE-2 score of 0.125 and ROUGE-L score of 0.178 suggest moderate performance in maintaining coherence and the logical structure of the summaries.
The generation times, ranging from 3.8 seconds for the shortest article to about 116 seconds for the longest, reflect the model's computational efficiency. The total generation time of 342 seconds for all examples is significantly reduced compared to the 392 seconds without the Accelerate library, demonstrating the library's effectiveness in speeding up model inference. This optimization is crucial for real-time applications and large-scale processing, as it nearly halves the time required for generating summaries while maintaining high-quality outputs.

The results indicate that while the model is effective at generating summaries that capture the main content, it may require further optimization to improve coherence and reduce generation times, especially for longer articles. These findings provide valuable insights into the model's current capabilities and highlight areas for potential enhancement.

### 5.1.2 Memoization and Parallel processing

Table 5.2 shows the snapshot of the evaluation metrics for text summarization using adaptive recursion with memoization and parallel processing.

| Example | Article Length (words) | ROUGE-1 | ROUGE-2 | ROUGE-L | Time (s) |
|---|---|---|---|---|---|
| 0 | 3668 | 0.461 | 0.260 | 0.271 | 38.729 |
| 1 | 2694 | 0.192 | 0.081 | 0.112 | 46.188 |
| 2 | 3862 | 0.399 | 0.172 | 0.210 | 45.166 |
| 3 | 1140 | 0.256 | 0.068 | 0.156 | 9.485 |
| 4 | 1607 | 0.181 | 0.073 | 0.120 | 10.155 |
| 5 | 6571 | 0.308 | 0.074 | 0.171 | 133.232 |
| 6 | 1743 | 0.193 | 0.110 | 0.158 | 8.607 |
| 7 | 3329 | 0.359 | 0.142 | 0.189 | 24.589 |
| 8 | 8774 | 0.410 | 0.153 | 0.209 | 75.075 |
| 9 | 880 | 0.377 | 0.120 | 0.182 | 3.764 |
| Total | 34268 | 0.314 | 0.125 | 0.178 | 394.990 |

Table 5.2: Test Results with Memoization and Parallel Processing

The results of the fine-tuned BART-large-CNN model using an adaptive recursive function with memoization and parallel processing demonstrate its capability to produce high-quality summaries, as indicated by the ROUGE metrics.

The the ROUGE metrics remain the same. However, despite the use of memoization aimed at reducing redundant computations by caching previously generated summaries, the total generation time was 394 seconds, which is longer compared to the 342 seconds achieved previously when the model was implemented using only an adaptive recursive function (without memoization). This increase in generation time indicates that, although memoization can save time by avoiding repeated computations, it introduced additional overhead that outweighed its benefits in this context. This is likely due to the complexities of managing the cache in a dynamic, recursive environment with parallel processing, where function inputs are not frequently repeated, making the cache checks and management more costly than the savings from avoided computations.

## 5.2 LED Model

The evaluation of the LED model based on the results table indicates that its summarization performance, as measured by ROUGE scores in Table 5.3, is generally lower than expected.

| Example | Article Length (words) | ROUGE-1 | ROUGE-2 | ROUGE-L | Time (s) |
|---|---|---|---|---|---|
| 0 | 3668 | 0.251 | 0.089 | 0.151 | 8.400 |
| 1 | 2694 | 0.142 | 0.059 | 0.090 | 6.463 |
| 2 | 3862 | 0.235 | 0.091 | 0.147 | 6.098 |
| 3 | 1140 | 0.287 | 0.043 | 0.133 | 4.426 |
| 4 | 1607 | 0.154 | 0.028 | 0.071 | 5.994 |
| 5 | 6571 | 0.064 | 0.007 | 0.064 | 6.963 |
| 6 | 1743 | 0.121 | 0.051 | 0.074 | 6.065 |
| 7 | 3329 | 0.227 | 0.081 | 0.138 | 6.784 |
| 8 | 8774 | 0.110 | 0.011 | 0.088 | 6.060 |
| 9 | 880 | 0.144 | 0.009 | 0.107 | 2.701 |
| Total | 34268 | 0.174 | 0.047 | 0.106 | 59.956 |

Table 5.3: Test Results of the LED Model

The average ROUGE-1 score is 0.173, suggesting that while the model captures some of the key content, it misses a significant portion of the relevant unigrams compared to the reference summaries.

The ROUGE-2 score, with an average of 0.046, reveals that the LED model struggles even more with retaining consecutive word pairs, indicating a challenge in maintaining coherence and fluency within the summaries.

Similarly, the average ROUGE-L score of 0.105 suggests limited overlap with the longest common subsequences, further highlighting deficiencies in capturing the full structure and logical flow of the original texts.

These scores demonstrate that while the LED model is efficient and capable of processing lengthy documents quickly, it sacrifices a considerable amount of detail and accuracy in the summaries it generates, pointing to a trade-off between speed and quality in its performance.

## 5.3 Detailed Results Comparison

### 5.3.1 ROUGE Scores

#### ROUGE-1

**LED Model:** The average ROUGE-1 score is 0.1735

**Finetuned Model:** The average ROUGE-1 score was 0.313

The ROUGE-1 score for the LED model is significantly lower than for the BART model. This suggests that the LED model is less effective at capturing the main content (unigrams) of the original articles. Figure 5.1 compares the ROUGE-1 scores between the finetuned model and the LED model across different test examples. The finetuned model achieves higher ROUGE-1 scores compared to the LED model.

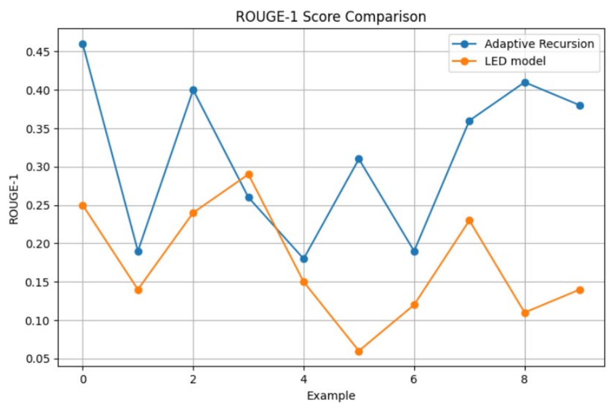


Figure 5.1: Comparison of ROUGE-1 Scores Across Test Examples

#### ROUGE-2

**LED Model:** The average ROUGE-2 score is 0.046

**Finetuned Model:** The average ROUGE-2 score was 0.125

The ROUGE-2 scores for the LED model are also much lower, indicating that it struggles more with maintaining the correct sequence of word pairs (bigrams) than the BART model. Figure 5.2 shows that the finetuned model consistently achieves higher ROUGE-2 scores than the LED model.

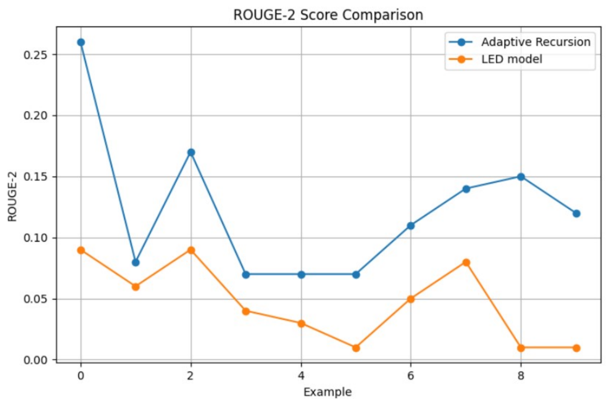


Figure 5.2: Comparison of ROUGE-2 Scores Across Test Examples

**ROUGE-L**

**LED Model:** The average ROUGE-L score is 0.105

**Finetuned Model:** The average ROUGE-L score was 0.177

The LED model's ROUGE-L score is also lower than that of the BART model, implying that the summaries generated by the LED model have less overlap with the longest common subsequence of the reference summaries. Figure 5.3 shows that the finetuned model consistently achieves higher ROUGE-L scores than the LED model.

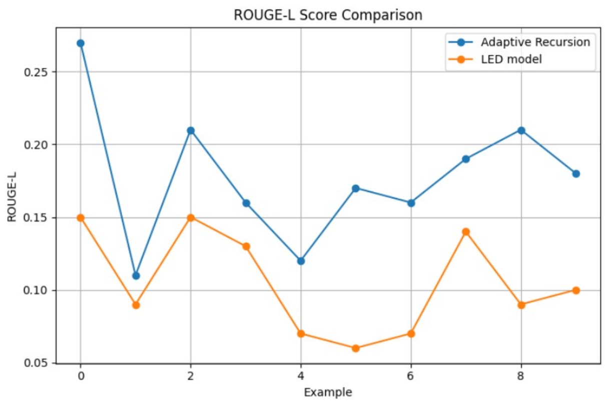


Figure 5.3: Comparison of ROUGE-L Scores Across Test Examples

### 5.3.2 Generation Time (s)

**LED Model:** Total generation time across all examples was 59 seconds.

**Finetuned Model (without memoization):** Total generation time was 394 seconds.

The LED model is significantly faster than the BART model, with a much lower total generation time. This suggests that the LED model is more efficient and better suited for handling longer texts quickly, reflecting the model's design for processing long documents. Figure 5.4 shows that the LED model consistently achieves higher ROUGE-2 scores than the finetuned model.

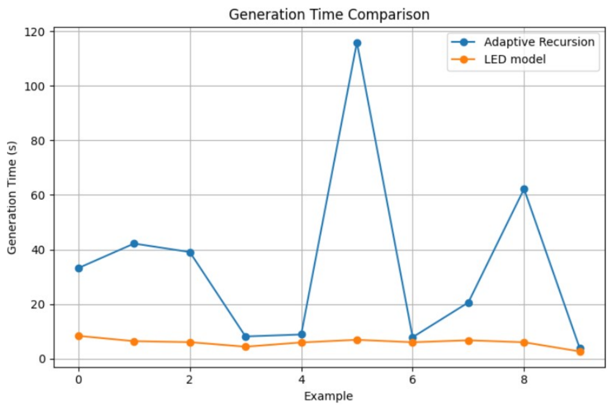


Figure 5.4: Comparison of Generation Time Across Test Examples

### 5.3.3 Overall Performance and Efficiency

**Content Coverage (ROUGE-1):**

The finetuned BART-large-CNN model with the adaptive recursive summarization strategy performs 81.50% better (based on the subsample selected) than the LED model.

**Coherence and Fluency (ROUGE-2 and ROUGE-L):**

The finetuned BART-large-CNN model with the adaptive recursive summarization strategy also shows superior performance in maintaining coherence and fluency, performing 163.83% better than the LED model when it comes to ROUGE-2 scores. It also performs 69.52% better than the LED model when it comes to ROUGE-L scores (this is based on the subsample selected).

**Efficiency (Generation Time):**

The LED model is about 70% faster (based on the subsample selected) in generating summaries, reflecting its ability to handle large documents efficiently. This efficiency might come at the expense of the summary quality, as shown by the lower ROUGE scores. This trade-off suggests that while the LED model is computationally efficient and scalable for longer texts, it may sacrifice accuracy and coherence compared to the finetuned BART-large-CNN model.

# Chapter 6

# Conclusion and Future Work

This study successfully tested and validated the hypothesis that a fine-tuned Large Language Model (LLM) with a small context window can be more effective than long-context LLMs for summarizing research papers, particularly for the purpose of conducting literature reviews. The research demonstrated that by employing techniques such as dynamic recursion and chunking, small-context LLMs can efficiently handle the summarization of long research papers without significant loss of information. This approach not only generates high-quality summaries that align well with the academic rigor and structure of research papers but also provides several advantages over traditional long-context models. The findings suggest that small-context LLMs, when properly fine-tuned and optimized, can be a more resource-efficient, environmentally friendly, and cost-effective alternative for tasks that involve processing extensive academic texts.

Despite the success of the hypothesis, there are several areas for improvement.
Future work could explore more advanced recursion techniques, such as incorporating adaptive recursion thresholds or machine learning-based chunk size optimization, to further improve the model's efficiency and output quality. Incorporating external knowledge bases or citation databases could enhance the model's ability to contextualize and summarize complex research topics more accurately. Additionally, future research should focus on integrating LaTeX equations, tables, and images into research papers. Effectively processing these elements is essential for creating summaries that accurately reflect the

content of scientific documents. Further efforts could also aim at enhancing fine-tuning strategies to capture academic writing nuances more effectively, thereby improving the model's generalizability and robustness across various disciplines.

By pursuing these avenues, future research could further solidify the viability of small-context LLMs for a range of academic and professional summarization tasks, ensuring they remain a competitive alternative to long-context models.

# Bibliography

[1] https://www.turing.ac.uk/news/publications/impact-large-language-models-finance-towards-trustworthy-adoption.

[2] https://www.kaggle.com/datasets/Cornell-University/arxiv.

[3] Banerjee, S., and Lavie, A. Meteor: An automatic metric for mt evaluation with improved correlation with human judgments. In *Proceedings of the acl workshop on intrinsic and extrinsic evaluation measures for machine translation and/or summarization* (2005), pp. 65–72.

[4] https://huggingface.co/facebook/bart-large-cnn.

[5] Beltagy, I., Peters, M. E., and Cohan, A. Longformer: The long-document transformer. *arXiv preprint arXiv:2004.05150* (2020).

[6] Benabderrahmane, S., Mellouli, N., and Lamolle, M. On the predictive analysis of behavioral massive job data using embedded clustering and deep recurrent neural networks. *Knowledge-Based Systems 151* (2018), 95–113.

[7] https://anuj58.medium.com/big-bird-283d64c11ea3.

[8] https://huggingface.co/datasets/ccdv/pubmed-summarization.

[9] Chen, J., Huang, X., Jiang, H., and Miao, X. Low-cost and device-free human activity recognition based on hierarchical learning model. *Sensors 21*, 7 (2021), 2359.

[10] Cho, K. Learning phrase representations using rnn encoder–decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078* (2014).

[11] https://huggingface.co/datasets/abisee/cnn$_d$*ailymail.*

[12] https://medium.com/@bakhitovd/fine-tuning-a-longformer-model-for-summarization-of-machine-learning-articles-fa6a9b31182a.

[13] Elman, J. L. Finding structure in time. *Cognitive science 14*, 2 (1990), 179–211.

[14] Gaskell, A. *On the Summarization and Evaluation of Long Documents.* PhD thesis, Imperial College London, 2020.

[15] Gidiotis, A., and Tsoumakas, G. A divide-and-conquer approach to the summarization of long documents. *IEEE/ACM Transactions on Audio, Speech, and Language Processing 28* (2020), 3029–3040.

[16] He, Y., Li, L., Zhu, X., and Tsui, K. L. Multi-graph convolutional-recurrent neural network (mgc-rnn) for short-term forecasting of transit passenger flow. *IEEE transactions on intelligent transportation systems 23*, 10 (2022), 18155–18174.

[17] Hochreiter, S., and Schmidhuber, J. Long short-term memory. *Neural computation 9*, 8 (1997), 1735–1780.

[18] Lewis, M., Liu, Y., Goyal, N., Ghazvininejad, M., Mohamed, A., Levy, O., Stoyanov, V., and Zettlemoyer, L. Bart: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension. *arXiv preprint arXiv:1910.13461* (2019).

[19] Lin, C.-Y. Rouge: A package for automatic evaluation of summaries. In *Text summarization branches out* (2004), pp. 74–81.

[20] Lin, C.-Y., and Hovy, E. Automatic evaluation of summaries using n-gram co-occurrence statistics. In *Proceedings of the 2003 human language technology conference of the North American chapter of the association for computational linguistics* (2003), pp. 150–157.

[21] https://medium.com/pythoneers/how-llms-evolved-in-the-artificial-intelligence-world-346874d98293.

[22] https://sh-tsang.medium.com/brief-review-longformer-the-long-document-transformer-8ab204d56613.

[23] Mastropaolo, A., Ciniselli, M., Di Penta, M., and Bavota, G. Evaluating code summarization techniques: A new metric and an empirical characterization. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering* (2024), pp. 1–13.

[24] Mikolov, T. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781* (2013).

[25] https://huggingface.co/datasets/bakhitovd/ML$_{a}rxiv$.

[26] Niu, Z., Zhong, G., and Yu, H. A review on the attention mechanism of deep learning. *Neurocomputing 452* (2021), 48–62.

[27] Nosouhian, S., Nosouhian, F., and Khoshouei, A. K. A review of recurrent neural network architecture for sequence learning: Comparison between lstm and gru.

[28] Papineni, K., Roukos, S., Ward, T., and Zhu, W.-J. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting of the Association for Computational Linguistics* (2002), pp. 311–318.

[29] Phang, J., Zhao, Y., and Liu, P. J. Investigating efficiently extending transformers for long input summarization. *arXiv preprint arXiv:2208.04347* (2022).

[30] https://pubmed.ncbi.nlm.nih.gov/.

[31] Rabiner, L. R. A tutorial on hidden markov models and selected applications in speech recognition. *Proceedings of the IEEE 77*, 2 (1989), 257–286.

[32] Radford, A., Wu, J., Child, R., Luan, D., Amodei, D., Sutskever, I., et al. Language models are unsupervised multitask learners. *OpenAI blog 1*, 8 (2019), 9.

[33] Raffel, C., Shazeer, N., Roberts, A., Lee, K., Narang, S., Matena, M., Zhou, Y., Li, W., and Liu, P. J. Exploring the limits of transfer learning with a

unified text-to-text transformer. *Journal of machine learning research 21*, 140 (2020), 1–67.

[34] Spärck Jones, K. A statistical interpretation of term specificity and its application in retrieval. *Journal of documentation 60*, 5 (2004), 493–502.

[35] Strubell, E., Ganesh, A., and McCallum, A. Energy and policy considerations for modern deep learning research. In *Proceedings of the AAAI conference on artificial intelligence* (2020), vol. 34, pp. 13693–13696.

[36] Sutskever, I., Vinyals, O., and Le, Q. V. Sequence to sequence learning with neural networks. *Advances in neural information processing systems 27* (2014).

[37] https://huggingface.co/datasets/shruti28062000/test$_{d}ataset.$

[38] https://huggingface.co/datasets/shruti28062000/combined$_{f}iltered_{d}ataset.$

[39] Vaswani, A. Attention is all you need. *arXiv preprint arXiv:1706.03762* (2017).

[40] Vedantam, R., Zitnick, C. L., and Parikh, D. Consensus-based image description evaluation. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 4566–4575.

[41] Wang, Q., Ding, L., Cao, Y., Tian, Z., Wang, S., Tao, D., and Guo, L. Recursively summarizing enables long-term dialogue memory in large language models. *arXiv preprint arXiv:2308.15022* (2023).

[42] Weizenbaum, J. Eliza—a computer program for the study of natural language communication between man and machine. *Communications of the ACM 9*, 1 (1966), 36–45.

[43] Zaheer, M., Guruganesh, G., Dubey, K. A., Ainslie, J., Alberti, C., Ontanon, S., Pham, P., Ravula, A., Wang, Q., Yang, L., et al. Big bird: Transformers for longer sequences. *Advances in neural information processing systems 33* (2020), 17283–17297.

[44] Zhang, T., Kishore, V., Wu, F., Weinberger, K. Q., and Artzi, Y. Bertscore: Evaluating text generation with bert. *arXiv preprint arXiv:1904.09675* (2019).

[45] Zhu, J., Yang, Z., Mourshed, M., Guo, Y., Zhou, Y., Chang, Y., Wei, Y., and Feng, S. Electric vehicle charging load forecasting: A comparative study of deep learning approaches. *Energies 12*, 14 (2019), 2692.