AARHUS
UNIVERSITY
DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

Bridging the Gap

Integrating Linear Algebra in the Development and Understanding of Large Language Models for Software Engineering Applications.

by

Asger Poulsen

202106630

Bachelor's Thesis

in

Computer Engineering

Supervisor: Hugo Daniel Macedo

Aarhus University, Department of Electrical and Computer Engineering

8 June 2024

Preface

This bachelor's thesis was written for the Department of Electrical and Computer Engineering, Aarhus University. It is part of the Computer Engineering study program, and was written in the spring semester of 2024.
Special thanks goes to my supervisor Hugo Daniel Macedo for suggesting the initial problem domain, connecting me with relevant contacts, and supervising this thesis.
All source files associated with this thesis are found at: `https://github.com/asgersong/BSc`

Asger Poulsen, June 8, 2024

Abstract

This bachelor's thesis delves into the integration of linear algebra concepts in the development and optimization of Large Language Models (LLMs) for software engineering applications. The thesis presents the theoretical underpinnings of LLMs, particularly focusing on the transformer architecture and its components. Furthermore, it explores practical applications through the development of a Retrieval-Augmented Generation (RAG) chatbot prototype, designed to assist students by providing contextually accurate and detailed responses. The chatbot leverages state-of-the-art models and integrates user-uploaded documents to enhance interaction quality. Additionally, a case study on the low-rank approximation of the BART model assesses the effectiveness of reducing computational complexity and storage requirements while maintaining performance on summarization tasks. The results, evaluated through ROUGE scores, underscore that low-rank approximation can significantly optimize model efficiency, but also highlight the trade-offs between performance and resource optimization. This work contributes to bridging the gap between theoretical linear algebra concepts and their practical applications in modern natural language processing tasks (NLP), offering insights for future research and development in the field.

Keywords: Large Language Models, Linear Algebra, Retrieval-Augmented Generation, Chatbot, BART, Low-Rank Approximation, Summarization, ROUGE Scores

Resume

Denne bacheloropgave dykker ned i integrationen af lineære algebra-koncepter i udviklingen og optimeringen af store sprogmodeller (LLMs) til anvendelser inden for software engineering. Opgaven præsenterer de teoretiske fundamenter for LLM'er, med særlig fokus på transformer-arkitekturen og dens komponenter. Derudover udforskes praktiske anvendelser gennem udviklingen af en prototype på en Retrieval-Augmented Generation (RAG) chatbot, designet til at assistere studerende ved at levere kontekstuelt præcise og detaljerede svar. Chatbotten udnytter avancerede modeller og integrerer bruger-uploadede dokumenter for at forbedre interaktionskvaliteten. Yderligere vurderes en case-studie om lav-rangs-approksimering af BART-modellen, der undersøger effektiviteten af at reducere beregningskompleksitet og lagerkrav, samtidig med at ydeevnen på summariseringsopgaver opretholdes. Resultaterne, evalueret gennem ROUGE-scores, understreger, at lav-rangs-approksimering kan optimere modeleffektivitet markant, men fremhæver også afvejningerne mellem ydeevne og ressourceoptimering. Dette arbejde bidrager til at bygge bro mellem teoretiske lineære algebra-koncepter og deres praktiske anvendelser i moderne opgaver inden for naturlig sprogbehandling (NLP), og tilbyder indsigt til fremtidig forskning og udvikling på området.

Contents

Chapter 1

Introduction

Large Language Models (LLMs) have become a cornerstone in the field of natural language processing (NLP) and artificial intelligence (AI), driving significant advancements and innovations. These models are designed to understand, generate, and interpret human language at a level that is increasingly indistinguishable from that of a human being. The development and evolution of LLMs mark a pivotal shift in how machines can learn from and interact with textual data, enabling a plethora of applications ranging from automated text generation to sophisticated conversational agents known as chatbots. Despite the growing use of AI in software- and computer engineering, there is still a notable disconnect between the core mathematical principles and their practical use in software development. Linear algebra, while fundamental to LLMs, is often not directly linked to its real-world applications for said engineers.

1.1 Thesis Objectives

This thesis aims to bridge the gap between linear algebra concepts taught in the classroom, LLMs, and computer engineering applications. The focus is on acquiring a deeper understanding of the specific linear algebra concepts that underpin LLMs, gaining hands-on experience in working with LLMs, and exploring the integration of classroom-taught linear algebra techniques with the development and optimization of LLMs.

The specific objectives of this bachelor's thesis are as follows:

1. **Theoretical Exploration:** Investigate the mathematics and concepts behind LLMs, emphasizing linear algebraic operations such as matrix multiplication and the Transformer Model architecture that forms the basis for today's state-of-the-art models.

2. **Computer Engineering Application:** Develop a Retrieval-Augmented Generation (RAG) chatbot prototype to gain practical experience with LLMs and explore their applications in software engineering education.

3. **Research Integration:** Evaluate the effectiveness of low-rank approximation techniques, such as Singular Value Decomposition (SVD), in reducing the

computational complexity and storage requirements of the BART model while maintaining performance on a downstream task, such as summarization.

To achieve these objectives, the thesis will explore the theoretical foundations of linear algebra as they apply to LLMs, understand how LLMs are developed and optimized, and assess the impact of low-rank approximation on performance by using objective metrics and qualitative analysis.

1.2 Outline

This thesis is structured as follows. Chapter 2 provides an overview of the fundamental concepts in linear algebra and deep learning, detailing their relevance to the development of LLMs. Chapter 3 reviews existing research on LLMs, focusing on optimization techniques and the role of linear algebra in enhancing model performance. Chapter 4 describes the methodologies employed in the development and evaluation of the RAG chatbot prototype and the application of low-rank approximation to the BART model. Chapter 5 details the implementation steps of the RAG chatbot and the low-rank approximation case study, highlighting key technical aspects and challenges. Chapter 6 presents the results of the empirical evaluations, discussing the impact of the optimization techniques on model performance and computational efficiency. Chapter 7 interprets the findings, considering the broader implications for the field of NLP and AI, and identifies potential areas for future research. Chapter **??** summarizes the key contributions of the thesis, reflects on the research objectives, and provides recommendations for future work.

Chapter 2

Theoretical Foundations

This chapter delves into the mathematical and theoretical underpinnings essential for understanding the development and application of large language models (LLMs). The discussion begins with an exploration of linear algebra, which forms the cornerstone of deep learning and is instrumental in defining the operations and transformations within neural networks. It further elucidates how vectors, matrices, and tensors serve as fundamental building blocks in representing data and parameters in neural networks.

Subsequently, the chapter examines the role of matrix operations, emphasizing the importance of matrix multiplication in transforming data across layers of a neural network. It also covers Singular Value Decomposition (SVD), a powerful technique for decomposing matrices, which is crucial for various machine learning tasks such as data compression and noise reduction.

The concept of low-rank approximation is introduced, highlighting its significance in reducing computational complexity and storage requirements, particularly in the context of deep learning where large matrices are prevalent. This section provides detailed insights into how low-rank approximation achieves efficiency gains without significantly compromising the quality of neural network outputs.

Additionally, the chapter discusses neural networks, focusing on their architecture, learning processes, optimization, and regularization techniques. It explains how neural networks are designed to capture and transmit abstract features through layers of interconnected nodes, and how they are trained to perform specific tasks through optimization algorithms and regularization methods.

Lastly, the chapter provides an understanding of cosine similarity, a metric used to measure the similarity between vectors, and its relevance in text analysis and NLP applications. The chapter culminates with an overview of large language models, detailing their architecture based on the Transformer model, the attention mechanisms that enable them to capture context, and the training and fine-tuning processes that enhance their performance on various NLP tasks.

2.1 Linear Algebra in Deep Learning

Linear algebra forms the cornerstone of deep learning, providing the necessary mathematical framework to model and understand complex relationships within

data. It is instrumental in defining the operations and transformations that occur within deep neural networks, including those underlying LLMs.

2.1.1 Vectors, Matrices, and Tensors

Vectors and matrices are fundamental to representing data and parameters in neural networks. A vector $\mathbf{v} \in \mathbb{R}^n$ can represent a point in n-dimensional space or a single data instance with n features. Matrices $A \in \mathbb{R}^{m \times n}$ facilitate linear transformations from $\mathbb{R}^n$ to $\mathbb{R}^m$, and tensors generalize these concepts to higher dimensions, accommodating the multi-dimensional data structures processed by neural networks.

A typical case, representing a basic neural network operation, can be expressed as:

$$\mathbf{y} = A\mathbf{x} + \mathbf{b} \tag{2.1}$$

where A is the weight matrix, $\mathbf{x}$ is the input vector, $\mathbf{b}$ is the bias vector, and $\mathbf{y}$ is the output vector of the transformation.

2.1.2 Matrix Operations

Matrix operations such as addition, multiplication, and transposition are essential in neural network computations. Matrix multiplication, in particular, plays a crucial role in transforming data between layers, capturing the relationships between input and output features. The dot product of two matrices $A \in \mathbb{R}^{m \times n}$ and $B \in \mathbb{R}^{n \times p}$ is defined as:

$$C = AB = \sum_{i=1}^{n} A_{ij} B_{jk} \tag{2.2}$$

where $C \in \mathbb{R}^{m \times p}$ is the resulting matrix. Matrix multiplication is a key operation in neural networks, enabling the transformation of input data through multiple layers of weights and biases.

Floating-Point Operations

Floating-point operations (FLOPs) are a measure of the computational complexity of matrix operations. The number of FLOPs required for matrix multiplication is proportional to the product of the dimensions of the matrices involved. For example, the number of FLOPs for multiplying two matrices $A \in \mathbb{R}^{m \times n}$ and $B \in \mathbb{R}^{n \times p}$ is $2mnp$ (Swann, 2022). Furtherly, the number of FLOPs required for a matrix-vector multiplication of matrix $A \in \mathbb{R}^{m \times n}$ and vector $\mathbf{x} \in \mathbb{R}^n$ is $2mn$ (Swann, 2022).

2.1.3 Singular Value Decomposition

Singular Value Decomposition (SVD) is a powerful technique for decomposing a matrix into singular vectors and singular values, providing insight into the structure of the data. For any matrix $A \in \mathbb{R}^{m \times n}$, SVD is given by:

$$A = U\Sigma V^T \tag{2.3}$$

where U and V are orthogonal matrices containing the left and right singular vectors, respectively, and Σ is a diagonal matrix with singular values. SVD is essential in many machine learning tasks, including noise reduction, data compression, and the analysis of neural network layers.

2.1.4 Low-Rank Approximation

Low-rank approximation is an effective technique in linear algebra that can provide significant benefits in terms of computational efficiency and storage requirements. This method is particularly useful in the context of deep learning, where large matrices are frequently encountered, and computational resources are often a limiting factor.

Concept of Low-Rank Approximation

Given a matrix A of dimensions $m \times n$, low-rank approximation aims to approximate A by another matrix A_k of lower rank k, where k is much smaller than both m and n. A version of this approximation leverages the Singular Value Decomposition (SVD) of A:

$$A \approx A_k = U_k \Sigma_k V_k^T$$

Here:

- U_k is an $m \times k$ matrix containing the top k left singular vectors.
- Σ_k is a $k \times k$ diagonal matrix containing the top k singular values.[1]
- V_k^T is a $k \times n$ matrix containing the top k right singular vectors.

This decomposition allows A_k to capture the most significant components of A, effectively reducing its rank while preserving its essential characteristics.

Reduction in Storage

The primary benefit of low-rank approximation is the significant reduction in storage requirements. Instead of storing the original matrix A with $m \times n$ elements, the low-rank approximation stores the three smaller matrices U_k, Σ_k, and V_k^T:

- U_k with $m \times k$ elements.
- Σ_k with $k \times k$ elements.
- V_k^T with $k \times n$ elements.

Thus, making the total number of elements required to store these matrices:

$$m \times k + k \times k + k \times n = k(m + k + n)$$

For k much smaller than m and n, this results in a substantial reduction in storage. For example, consider a matrix A of size 1000×1000. If we approximate A with rank $k = 50$ we have:

- Original Matrix: $1000 \times 1000 = 1000000$ elements.

[1] While it is possible to represent Σ_k as a k-element vector, choosing a $k \times k$ matrix allows for more flexibility in mathematical operations and maintains consistency in the representation of the decomposition, which can simplify implementation and theoretical analysis.

- Low-rank approximation: $1000 \times 50 + 50 \times 50 + 50 \times 1000 = 102500$ elements.

which is a substantial reduction in storage requirements.
But if we choose $k = 500$, the low-rank approximation would require

$$1000 \times 500 + 500 \times 500 + 500 \times 1000 = 1250000 \text{ elements},$$

which is more than the original matrix A. Thus, the choice of k is crucial in achieving storage reduction.
More specifically, we want

$$\begin{gathered} k(m+k+n) = k^2 + k(m+n) < mn \\ \Updownarrow \\ k^2 + k(m+n) - mn < 0 \\ \Updownarrow \\ k < \frac{\sqrt{4mn + (m+n)^2} - m - n}{2} \end{gathered}$$

This inequality provides a guideline for selecting an appropriate rank k to achieve storage reduction.

Reduction in Computation for Matrix-Vector Multiplications

Low-rank approximation can also reduce the computational complexity of various matrix operations. In the case of a matrix-vector multiplication we have:

- **Original Matrix**: Multiplying A (size $m \times n$) with a vector (size n) requires $2mn$ FLOPs.
- **Low-Rank Approximation**: Multiplying $A_k = U_k \Sigma_k V_k^T$ with a vector involves three steps:
  - V_k^T with the vector: $2kn$ FLOPs,
  - Σ_k multiplication: $2k^2$ FLOPs.
  - U_k multiplication: $2mk$ FLOPs.
- **Total**: $2kn + 2k^2 + 2mk = 2k(k+m+n)$ FLOPs.

Therefore, the appropriate choice of rank k to achieve computational efficiency for matrix-vector multiplications is

$$\begin{gathered} 2kn + 2k^2 + 2mk = 2k(k+m+n) < 2mn \\ \Updownarrow \\ k < \frac{\sqrt{4mn + (m+n)^2} - m - n}{2} \end{gathered}$$

Reduction in Computation for Matrix-Matrix Multiplications

In the case of a matrix-matrix multiplication we have:

- **Original Matrix**: Multiplying A (size $m \times n$) with another matrix of size $n \times p$ requires $2mnp$ FLOPs.
- **Low-Rank Approximation**: Multiplying $A_k = U_k \Sigma_k V_k^T$ with a matrix of size $n \times p$ involves:
  - V_k^T with the matrix: $2knp$ FLOPs.
  - Σ_k multiplication: $2k^2p$ FLOPs.
  - U_k multiplication: $2mkp$ FLOPs.
- **Total**: $2knp + 2k^2p + 2mkp = 2k(k + m + n)p$ FLOPs.

Therefore, the appropriate choice of rank k to achieve computational efficiency for matrix-matrix multiplications is also

$$k < \frac{\sqrt{4mn + (m+n)^2} - m - n}{2}$$

By choosing a sufficiently small k, the amount of FLOPs can be reduced compared to when using the full-rank matrix A.
Thus, making low-rank approximation via SVD a powerful technique that offers significant benefits in terms of storage reduction and computational efficiency. By focusing on the most important components of a matrix, low-rank approximation makes it feasible to handle large datasets and complex computations more effectively, which is crucial in the field of deep learning and beyond.

2.1.5 Neural Networks in Deep Learning

A Neural network consist of interconnected nodes or "neurons" arranged in layers, with each layer designed to perform specific transformations on its inputs to capture and transmit increasingly abstract features to subsequent layers.

Architecture of Neural Networks

The architecture of a neural network is defined by layers, each comprising a set of neurons connected by weights. These weights are adjusted during the training process to minimize the difference between the actual output of the network and the desired output. A typical feedforward neural network can be mathematically represented as:

$$\mathbf{h}^{(l)} = f(W^{(l)}\mathbf{h}^{(l-1)} + \mathbf{b}^{(l)}) \tag{2.4}$$

where $W^{(l)}$ and $\mathbf{b}^{(l)}$ are the weight matrix and bias vector for the l-th layer, $\mathbf{h}^{(l-1)}$ is the output from the previous layer, and f is a non-linear activation function such as ReLU or sigmoid.

Learning Process

The learning process in neural networks involves adjusting weights and biases to reduce a loss function, commonly through backpropagation. This method efficiently computes gradients using calculus' chain rule:

$$W^{(l)} \leftarrow W^{(l)} - \eta \frac{\partial \mathcal{L}}{\partial W^{(l)}} \tag{2.5}$$

where η is the learning rate and $\mathcal{L}$ is the loss function.

Optimization and Regularization

Optimization algorithms like Stochastic Gradient Descent (SGD), Adam, and RMSprop are crucial for weight updates. Regularization techniques such as dropout and weight decay help prevent overfitting, ensuring the network generalizes well to new data.

2.1.6 Cosine Similarity

The cosine similarity between two vectors $\mathbf{a}$ and $\mathbf{b}$ is defined as:

$$S_c(\mathbf{a}, \mathbf{b}) = \frac{\mathbf{a} \cdot \mathbf{b}}{\|\mathbf{a}\|\|\mathbf{b}\|}$$

where $\mathbf{a} \cdot \mathbf{b}$ is the dot product of $\mathbf{a}$ and $\mathbf{b}$, and $\|\mathbf{a}\|$ and $\|\mathbf{b}\|$ are the magnitudes (or norms) of vectors $\mathbf{a}$ and $\mathbf{b}$, respectively.

Cosine similarity is a metric used to measure the cosine of the angle between two non-zero vectors in an inner product space. It is particularly useful in the context of text analysis and NLP, where it quantifies the similarity between two vectors of an inner product space that represents the words or documents.

By applying cosine similarity, models can effectively capture the similarity in high-dimensional data, making it a valuable tool in various machine learning and deep learning applications.

2.2 Understanding LLMs

LLMs such as GPT (Generative Pre-trained Transformer) and BERT (Bidirectional Encoder Representations from Transformers) have revolutionized the field of NLP by leveraging deep neural networks to understand and generate human-like text unlocking a whole host of new applications.

2.2.1 The Concept of LLMs?

LLMs are deep neural networks trained on vast amounts of text data. They learn to predict the next word in a sentence, understand context, generate text, and perform various NLP tasks with minimal task-specific adjustments. The strength of LLMs lies in their ability to capture intricate patterns in language through extensive pre-training.

2.2.2 Architecture of LLMs

The architecture of most LLMs is based on the Transformer model, introduced by Vaswani et al., 2017, which relies on self-attention mechanisms to weigh the significance of different words in a sentence. The Transformer architecture is composed of two main components: an encoder and a decoder. The encoder takes the input text and produces a sequence of hidden states, which represent the meaning of the text. The decoder then takes the encoder's hidden states and generates the output text, one word at a time.

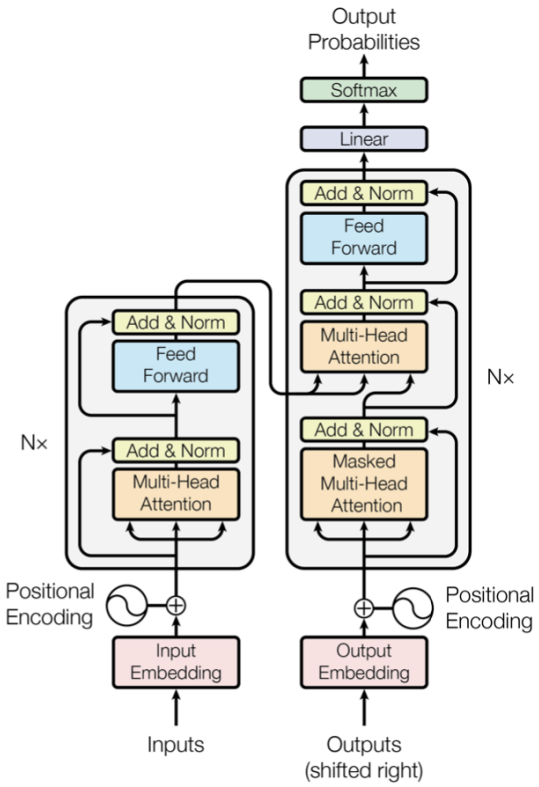


Figure 2.1: Transformer model architecture - Taken from "Attention Is All You Need" by Vaswani et al., 2017.

At a high level, the goal of the Transformer is to take an input text and predict the next word in the sequence. The input text is broken into tokens, which are typically words or parts of words. Each token is then represented as a high-dimensional vector, known as its embedding. Initially, these embeddings encode only the individual meaning of the tokens without any contextual information.

Encoder

The encoder consists of a stack of N identical layers, each containing two sub-layers: a multi-head self-attention mechanism and a feed-forward neural network. Furtherly, a residual connection is employed around each of the two sub-layers, followed by layer normalization. So the output of each sub-layer is $\text{LayerNorm}(x + \text{SubLayer}(x))$, where $\text{SubLayer}(x)$ represents the function implemented by the sub-layer.

The self-attention mechanism allows the model to weigh the importance of different words in the input sequence when generating the output. The feed-forward neural network processes the output of the self-attention mechanism to produce the final hidden states of the encoder. A key feature of the encoder is that it processes all words in the input sequence in parallel, which contributes to the efficiency of the Transformer model. The output of the encoder is a sequence of vectors, each representing an input word in a high-dimensional space.

Decoder

The decoder, on the other hand, also consists of a stack of N identical layers, but with an additional third sublayer in each decoder layer, which performs multi-head attention over the encoder's output. This allows the decoder to focus on different parts of the encoder's output for each word in the output sequence. In the first sublayer of the decoder, self-attention is used, but with a constraint (masking) to prevent positions from attending to subsequent positions. This ensures that the predictions for position i can depend only on the known outputs at positions less than i. The purpose of the decoder is to generate an output sequence one word at a time, using the encoder's output and what it has produced so far as inputs.

Attention

The attention mechanism is a cornerstone of the Transformer model. Its primary function is to dynamically adjust the importance of each word in the input sequence based on its context, thereby refining the embeddings of these words to capture richer meanings.

The attention mechanism allows each token to attend to every other token in the sequence, effectively allowing the model to focus on the most relevant parts of the input when making predictions. This process is mathematically described as follows:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V \tag{2.6}$$

Here, Q, K, and V are the query, key, and value matrices, respectively. These matrices are derived from the input embeddings by multiplying them with learned weight matrices. The dimension d_k is the size of the key vectors and serves to scale the dot products to prevent them from growing too large.

To understand how this works, consider a simple example: The phrases

- "After dinner, they enjoyed a sticky *date* pudding."
- "She circled the *date* of the concert on her calender."
- "They went on their first *date* to the zoo."

The word "date" has different meanings in each phrase, but its initial embedding would be the same. The attention mechanism helps to refine the meaning of "date" based on its context by allowing the embeddings of surrounding words to influence it.

Steps of the Attention Mechanism:

1. **Compute Query, Key, and Value Vectors:** For each embedded token $\overrightarrow{E_i}$, we compute query ($\overrightarrow{Q_i}$), key ($\overrightarrow{K_i}$), and value ($\overrightarrow{V_i}$) vectors by multiplying the token's embedding with learned matrices W^Q, W^K, and W^V:
$$\overrightarrow{Q_i} = W^Q \overrightarrow{E_i}, \quad \overrightarrow{K_i} = W^K \overrightarrow{E_i}, \quad \overrightarrow{V_i} = W^V \overrightarrow{E_i}$$
2. **Compute Attention Scores:** Calculate the dot products of the query vectors with all key vectors to measure how much focus each word should have on every other word, and divide by the square root of the key dimension for numerical stability:
$$\text{Scores} = \frac{QK^T}{\sqrt{d_k}}$$
3. **Apply Softmax Activation:** Normalize these scores using the softmax function to obtain attention weights:
$$\text{Attention Weights} = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)$$
4. **Compute Weighted Sum of Values:** Multiply the attention weights by the value vectors to get the final output for each token:
$$\text{Output} = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

which finally leaves us with Eq. 2.6.

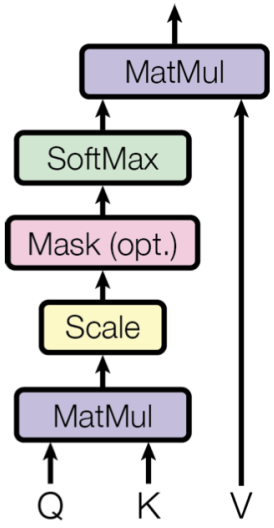


Figure 2.2: Illustration of the Attention Mechanism - Taken from "Attention Is All You Need" by Vaswani et al., 2017.

This mechanism allows the model to dynamically adjust which parts of the input sequence to focus on, thereby capturing the contextual meaning of each word. This whole process is described as a single 'head' of attention, and multiple heads can be used in parallel to capture different aspects of the input sequence, as elaborated in section 2.2.2.

Multi-head Attention

Multi-head attention extends the single-head attention mechanism by allowing the model to attend to different parts of the input sequence simultaneously, capturing a variety of relationships and patterns. Instead of performing a single attention function, multi-head attention projects the queries, keys, and values into multiple subspaces and performs the attention function in each subspace independently.

This process is mathematically described as follows:

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \ldots, \text{head}_h)W^O \tag{2.7}$$

$$\textbf{where } \text{head}_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V) \tag{2.8}$$

Here the projections are the learned parameter matrices W_i^Q, W_i^K, and W_i^V for the i-th head, and W^O is the output matrix.

Steps of Multi-head Attention:

1. **Linear Projections**: Project the input embeddings into h different subspaces using learned weight matrices to create multiple sets of queries, keys, and values:
$$Q_i = QW_i^Q, \quad K_i = KW_i^K, \quad V_i = VW_i^V$$
2. **Parallel Attention Heads**: Apply the attention mechanism to each set of projections in parallel, producing multiple attention outputs:
$$\text{head}_i = \text{Attention}(Q_i, K_i, V_i)$$
3. **Concatenate and Project**: Concatenate the outputs of all attention heads and project them back to the original embedding space using a final learned matrix W^O:
$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \ldots, \text{head}_h)W^O$$

Multi-head attention allows the model to capture different aspects of the input sequence in parallel, which enhances its ability to understand complex patterns and dependencies in the data.

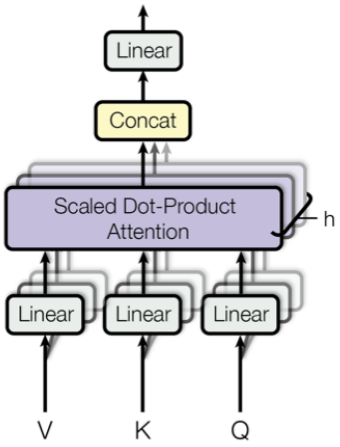


Figure 2.3: Illustration of Multi-head Attention - Taken from "Attention Is All You Need" by Vaswani et al., 2017.

In summary, the attention and multi-head attention mechanisms enable the Transformer model to focus on relevant parts of the input sequence dynamically, improving its ability to capture context and generate accurate predictions. These mechanisms are crucial to the model's success in various NLP tasks.

Why Transformers?

The Transformer model has several advantages over traditional Recurrent Neural Networks (RNNs) and Long Short-Term Memory (LSTMs), including the ability to capture long-range dependencies, parallelize computation, and scale to larger datasets. The self-attention mechanism allows the model to focus on relevant parts of the input sequence, enabling it to learn complex patterns in the data. The multi-head attention mechanism further enhances the model's ability to capture different aspects of the input sequence in parallel, improving its performance on a wide range of NLP tasks such as translation, summarization, and image captioning. This is why Transformers have become the architecture of choice for many state-of-the-art NLP models, including GPT and BERT.

2.2.3 Training and Fine-Tuning

The foundation of an LLM's understanding and generation of human language lies in its pre-training phase. During this stage, the model is exposed to a large corpus of text data, often encompassing a wide range of topics, genres, and styles. The primary objective of pre-training is to enable the model to learn a generalized representation of language.

The pre-training is typically conducted using unsupervised learning techniques, where the model is trained on tasks like Masked Language Modeling (MLM) or Next Sentence Prediction (NSP). In MLM, for example, a percentage of the input tokens are randomly masked, and the model's objective is to predict the original tokens at these masked positions. The MLM objective can be formally represented as:

$$\mathcal{L}_{\text{MLM}} = -\sum_{i \in \mathcal{M}} \log p(x_i | x_{\setminus \mathcal{M}}) \tag{2.9}$$

where $\mathcal{M}$ is the set of masked positions, x_i is the original token at position i, and $x_{\setminus \mathcal{M}}$ represents the input with masked tokens.

Following pre-training, LLMs undergo a fine-tuning phase, wherein the model is specialized to perform specific NLP tasks. This phase involves training the pre-trained model on a smaller, task-specific dataset, allowing the model to adjust its weights to better perform the target task.

The fine-tuning process can be represented as a continuation of the training process, optimizing the following objective:

$$\mathcal{L}_{\text{fine-tune}} = -\sum_{(x,y) \in \mathcal{D}_{\text{task}}} \log p(y | x; \theta_{\text{pre-train}} + \Delta\theta) \tag{2.10}$$

where $\mathcal{D}_{\text{task}}$ is the task-specific dataset, (x, y) are the input-output pairs, $\theta_{\text{pre-train}}$ are the parameters learned during pre-training, and $\Delta\theta$ represents the parameter updates during fine-tuning.

Fine-tuning LLMs presents challenges such as catastrophic forgetting and overfitting, particularly when the task-specific dataset is small. Various strategies, including careful learning rate selection, regularization techniques, and the use of adapters, are employed to mitigate these issues.

Chapter 3

Literature Review

This chapter provides a comprehensive review of the foundational architectures, innovative compression techniques, and specific models that have significantly contributed to the development of LLMs. By examining these areas, this chapter aims to offer insights into the current state and future directions of natural language processing (NLP) research.

Firstly, an overview of LLMs will be presented based on Naveed et al., 2024. This section highlights the rapid evolution of LLMs and the ongoing need for optimization by engineers and researchers. Secondly, previous studies on LLM compression, with a focus on versions of low-rank approximation, will be briefly discussed. The chapter will then introduce the approach of Retrieval-Augmented Generation (RAG) by Lewis et al., 2020, which combines the strengths of pre-trained language models with retrieval mechanisms to enhance performance on knowledge-intensive tasks. Following this, the BART model, which is the focus of this thesis, will be introduced. Finally, the chapter will conclude with a discussion on the evaluation of summarization models using ROUGE metrics.

By providing a detailed examination of these areas, this chapter sets the stage for the subsequent chapters that delve into the practical implementation of a RAG chatbot prototype and the application of low-rank approximation to the BART model.

3.1 Overview of LLMs

The evolution of LLMs can be traced back to earlier models of machine learning that attempted to process and understand language. However, it was the introduction of the Transformer architecture by Vaswani et al., 2017 that revolutionized the field of NLP, enabling the development of large-scale models capable of capturing complex linguistic patterns and generating coherent text. This architectural innovation laid the foundation for today's models like Google's BERT and OpenAI's GPT series that marked a significant leap in the capabilities of language models. Each iteration of these models has brought improvements in understanding context, generating text, and general language comprehension, culminating in state-of-the-art models that are capable of writing essays, composing poetry, and even generating code.

Naveed et al., 2024 gives a comprehensive overview of the evolution of LLMs, highlighting the key milestones and breakthroughs that have shaped the landscape of NLP research and applications. The study delves into the architectural advancements, training methodologies, and performance benchmarks of LLMs, providing a detailed account of the progress made in the field over the past decade. By examining the historical context and the latest developments in LLM research, the study offers valuable insights into the current state-of-the-art and the future directions of large-scale language modeling.

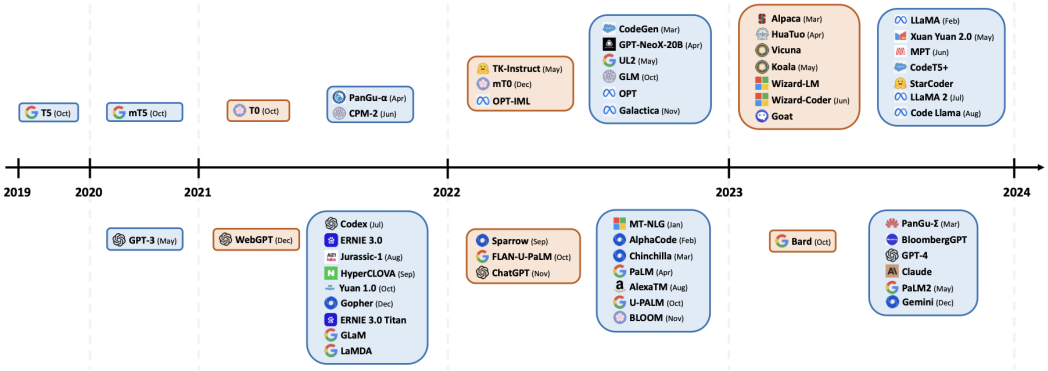


Figure 3.1: Chronological display of LLM releases: blue cards represent 'pre-trained' models, while orange cards correspond to 'instruction-tuned' models. Models on the upper half signify open-source availability, whereas those on the bottom half are closed-source. - Taken from Naveed et al., 2024

Figure 3.1 depicts the chronological development of LLM releases, categorizing them based on their nature as either pre-trained or instruction-tuned, and their availability as either open-source or closed-source. This visual representation provides a thorough overview of the diverse models that have emerged over the years, highlighting the rapid advancement and widespread adoption of LLMs in the NLP domain. The chart also reveals a growing trend towards instruction-tuned and open-source models, illustrating the shifting landscape and emerging trends in natural language processing research.

As newer, larger, and more sophisticated models continue to shape the field of NLP, the demand for optimization strategies that enhance the efficiency and performance of these models become increasingly critical. This has spurred a surge in research (also depicted in Naveed et al., 2024) focused on model compression, parameter sharing, and other optimization techniques aimed at reducing computational overhead and improving the operational characteristics of LLMs. Through the exploration of innovative methodologies and strategies, researchers are striving to make LLMs more accessible, adaptable, and efficient, thereby unlocking their full potential across a wide range of applications. This is a product of the increasing demand for LLMs across a vast spectrum of applications, and as the demand grows, so does the need for researchers and developers to understand and thus, by extension, be able to optimize these models to meet the demands of the applications they are used in.

3.2 Previous Studies on LLM Compression with Low-Rank Approximation

Over the past few year, the increasing demand for LLMs across a vast spectrum of applications, from natural language processing to content generation, has underscored the critical need for optimization strategies tailored to these complex systems. Researchers have delved into a variety of optimization techniques aimed at refining LLMs, with a keen focus on enhancing model efficiency and reducing the computational burden without compromising the specialized performance that these models are known for. A key area of investigation has been model compression, which involves reducing the size and complexity of LLMs while maintaining their functionality and performance.

In the realm of LLMs, low-rank approximation has gained substantial traction as a method to fine-tune and streamline models. It has been utilized in various implementations, notably in Hu et al., 2021 and its subsequent adaptations (Valipour et al., 2023; Zhang et al., 2023; Chavan et al., 2024). These adaptations focus on leveraging low-rank structures to optimize the fine-tuning proces, and in some cases performance of LLMs without extensive resource demands.

Another novel application of this technique is seen in TensorGPT (Xu, Xu, and Mandic, 2023), which employs a low-rank tensor format to manage large embeddings efficiently. By adopting the Tensor-Train Decomposition (TTD), TensorGPT not only reduces the spatial complexity of LLMs but also potentially boosts their deployment on edge devices. This method treats each token embedding as a Matrix Product State (MPS), enabling the compression of the embedding layer by up to a factor of 38.40. Remarkably, this compression is achieved without sacrificing, and in some cases even enhancing, the performance of the model compared to its original configuration.

This focus on low-rank approximation underscores its critical role in advancing the field of LLMs by facilitating more efficient model architectures that maintain high performance while being computationally less demanding.

3.3 Retrieval-Augmented Generation

As LLMs depend on the data they are trained on, they may not always have access to the most up-to-date or specialized information. To address this limitation, Lewis et al., 2020 introduced Retrieval-Augmented Generation (RAG), a novel approach that combines the strengths of pre-trained language models (a *generator*), with retrieval mechanisms to enhance performance on knowledge-intensive tasks (*retriever*). RAG models integrate a dense vector index of external knowledge sources, such as Wikipedia, into the generative process, enabling them to access a vast repository of information beyond their pre-training data.

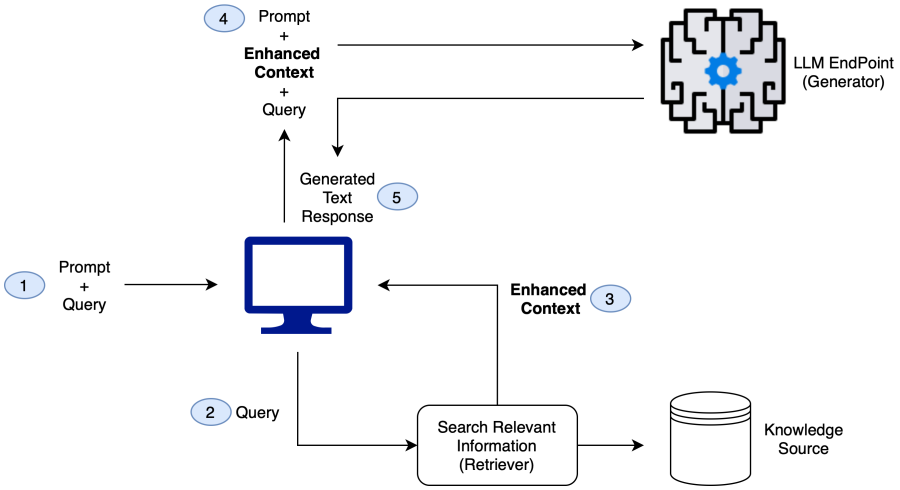


Figure 3.2: Conceptual flow of using RAG with LLMs

Figure 3.2 illustrates the conceptual flow of using RAG with LLMs. After prompting a query, the retriever component fetches relevant passages from the external knowledge source based on the input query, which are then used by the generator to produce the final output. This retrieval mechanism enhances the model's ability to generate contextually rich and accurate responses, particularly in knowledge-intensive tasks. Further benefits of RAG models include:

- **Implementation of Chatbots:** Chatbot applications can benefit from RAG models by providing more accurate and contextually relevant responses to user queries.
- **Staying up to date with Current Information:** Even if the original training data sources for an LLM are suitable for ones needs, maintaining relevancy is challenging. RAG enables developers to keep generative models up-to-date by incorporating the latest research, statistics, and news. By connecting the LLM directly to live social media feeds, news sites, or other frequently updated information sources, RAG can ensure the model's responses remain current and accurate.
- **Enhanced User Trust:** By providing more accurate and contextually relevant responses, RAG models can enhance user trust in the system. This is particularly important in applications where the information provided must be accurate and up-to-date, such as medical advice, legal consultations, or financial services.

3.4 The BART Model

BART: Denoising Sequence-to-Sequence Pre-training for Natural Language Generation, Translation, and Comprehension by (Lewis et al., 2019), introduces BART as a versatile pretraining approach for natural language processing tasks. BART combines the benefits of both autoregressive (like GPT) and autoencoding (like BERT) paradigms into a unified model.

3.4.1 Architecture and Training

BART employs a Transformer-based sequence-to-sequence architecture from (Vaswani et al., 2017), utilizing a bidirectional encoder (similar to BERT) and a left-to-right decoder (similar to GPT) to handle a wide array of tasks from generation to comprehension. For the base model (BART-base) and the large model (BART-large), the encoder and decoder consist of 6 and 12 layers, respectively (Appendix C). It is trained through a novel denoising objective, where the model reconstructs the original text from corrupted versions. This involves techniques such as token masking (Devlin et al., 2019) and text infilling, enhancing the model's ability to understand and generate contextually rich language.

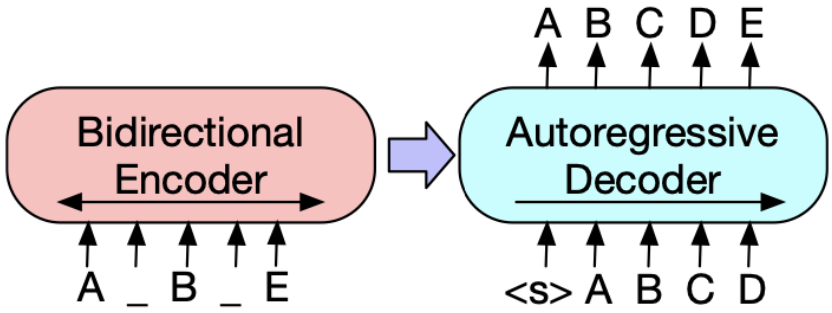


Figure 3.3: BART model's processing mechanism - Taken from Lewis et al., 2019

Figure 3.3 illustrates the BART model's processing mechanism. The original text sequence [ABCDE] is transformed into a masked version [A[MASK]B[MASK]E]. BART's encoder learns bidirectional representations from this altered input, allowing it to handle inputs that are not aligned with decoder outputs. The decoder then attempts to reconstruct the original sequence autoregressively by optimizing the negative log-likelihood. For fine-tuning, the model processes an uncorrupted document to enhance accuracy and adaptability.

3.4.2 Efficacy and Applications

The versatility of BART is demonstrated across various NLP tasks including text generation, comprehension, and translation. Notably, BART outperforms previous work on benchmarks like ROUGE, showcasing substantial improvements particularly in abstractive summarization tasks. The model's ability to fine-tune to specific tasks using pre-trained weights allows it to excel in both generation and comprehension roles, making it a powerful tool for a broad range of applications.

3.5 Evaluating Summarization with ROUGE

The evaluation of automated summarization models is crucial for assessing their efficiency and effectiveness in capturing the essence of text data. ROUGE (Lin, 2004), which stands for Recall-Oriented Understudy for Gisting Evaluation, provides a set of metrics that are indispensable for this purpose. Introduced by Lin, 2004, ROUGE measures compare the overlap between computer-generated summaries and a set of reference summaries typically created by humans.

ROUGE Metrics ROUGE includes several specific metrics, such as ROUGE-N (n-gram overlap) and ROUGE-L (longest common subsequence), each suited for different aspects of summarization. For instance, ROUGE-N focuses on the exactness of content at various n-gram levels, providing insights into the precision of the summarization model in replicating key information. In contrast, ROUGE-L assesses the fluency and structure of the generated summaries by measuring sequence similarity, which is crucial for evaluating the narrative flow of the text.

Application and Relevance The application of ROUGE in evaluation has been extensively validated across various tasks, including the Document Understanding Conferences (DUC), where it has been used to measure the performance of summarization systems in a competitive environment. These metrics have proven to be reliable indicators of human judgment, making them a standard against which the summarization capabilities of LLMs are benchmarked. The relevance of ROUGE scores lies in their ability to provide quantifiable measures that correlate strongly with human evaluations, thus facilitating the improvement and development of more efficient summarization models.

Significance in LLM Research In the context of LLMs, understanding the effectiveness of different compression and optimization techniques often relies on the ability to evaluate how well the reduced models can summarize content. ROUGE metrics serve this purpose by quantifying the trade-offs between model complexity and performance retention, helping researchers and developers optimize LLMs without significant loss in functionality.

Chapter 4

Methodology

This chapter details the methodologies employed in the development and evaluation of the RAG Chatbot and the low-rank approximation of the BART model. The aim is to provide a comprehensive understanding of the techniques, tools, and processes used to achieve the research objectives.

The chapter is structured into two main sections. The first section focuses on the RAG Chatbot, named Study Buddy, which was developed to assist students by providing detailed and context-specific responses. This section outlines the motivation for developing the chatbot, describes the tools and environment used, and explains the implementation steps taken to create the prototype. The second section addresses the low-rank approximation of the BART model. It describes how low-rank approximation was applied to the attention weight matrices of the BART model to reduce computational complexity and storage requirements while maintaining performance on summarization tasks. The implementation steps and evaluation metrics used to assess the effectiveness of this approach are also presented, along with the criteria for selecting an appropriate rank for computational efficiency and storage reduction.

4.1 RAG Chatbot: Study Buddy

A survey conducted (by Hanifi, Cetin, and Yilmaz, 2023) reports that 93% of software engineering students, comprising of mainly 3rd and 4th semester students, use ChatGPT during their course projects. Among them 80.2% were categorized as active users which entails tasks as generating source code. A similar observation can be made by teaching assistants in the SW2PLA course at Aarhus University where students frequently resort to using chatbots like ChatGPT for quick answers to their questions. However, these chatbots often provide generic responses that may not be sufficiently helpful for academic purposes, as they lack the specificity and depth required for effective studying. This observation underscores a gap in the effectiveness of current chatbot solutions in educational contexts.

The RAG model (Lewis et al., 2020) addresses this gap by combining a retriever and a generator, thereby enabling the delivery of more detailed and context-specific information.

To gain practical experience with LLMs and their application in real-world language processing tasks, this thesis aims to develop a prototype for a chatbot inspired by a project at BSS - Aarhus University (Appendix B). This chatbot, utilizises the RAG model Lewis et al., 2020, serves as a 'Study Buddy' to assist students in understanding complex topics, providing quick and relevant answers, and enhancing their learning experience for challenging subjects.

4.1.1 Development and Environment Tools

The implementation of the RAG chatbot was accomplished using the following tools and technologies:

- **OpenAI API:** Utilized to access the GPT-4 model for the chatbot's generation capabilities.
- **Astra DB:** Employed for storing and managing the data used by the retriever component.
- **DataStax RAGstack:** A curated stack of leading open-source software designed to facilitate the implementation of the RAG pattern in production-ready applications using Astra Vector DB as a vector store.
- **Langchain:** An open-source framework used for developing applications powered by LLMs.
- **Streamlit:** An open-source Python framework used for building and deploying data applications with minimal code.

4.1.2 Implementation Steps

The development of the RAG chatbot involves the following steps:

1. **Chatbot Interface:** Designing and creating a user-friendly interface for users to interact with the chatbot.
2. **Generator Component:** Implementing the generator component to provide responses based on the GPT-4 model.
3. **Retriever Component:** Developing the retriever component to search for relevant information in the AstraDB Vector Store based on user queries.
4. **Integration:** Integrating the retriever and generator components to create a functional RAG chatbot.
5. **PDF Uploader:** Implementing a feature that allows users to upload their own materials, enabling more meaningful and contextual responses.
6. **Deployment:** Deploying the chatbot on a cloud platform to make it possible to be publicly accessible.[1]

[1] As the current implementation of the chatbot uses a private OpenAI account and due to the costs associated with this, the current implementation of the chatbot is not publicly accessible. However, access can be provided upon request.

The conceptual workflow of the Study Buddy, is depicted in Figure 3.2, where in Studdy Buddy's case AstraDB is used as the knowledge source and OpenAI's ChatGPT-4 as generator. The process begins with the user inputting a query, which is then processed by the retriever to find relevant information. This information provides context based on the user's query. Subsequently, the generator creates a response using an embedded prompt template, the retrieved information, and the user's query, thereby providing the user with a detailed and specific answer.

4.2 Case Study: Low-Rank Approximation of BART model

To assess the effectiveness of low-rank approximation in compressing Facebook's BART-model (Lewis et al., 2019), both its base- and large model versions are considered as a case study.

This thesis focuses on applying low-rank approximation specifically to the attention matrices. This choice stems from their pivotal role in the transformer architecture Vaswani et al., 2017. These matrices, which help the model assess the relevance of different words within the input data, tend to be large and often encapsulate redundant information (Aghajanyan, Zettlemoyer, and Gupta, 2020).

By applying low-rank approximation to the attention weight matrices of the BART model, the possibility of reducing computational complexity and storage requirements while preserving performance on a summarization task is explored.

4.2.1 Implementation Steps

The implementation of low-rank approximation for BART involves the following steps:

1. **Computing SVD of Attention Weight Matrices:** The attention weight matrices (Key, Query, Value, and Output) of the model are decomposed using SVD to obtain the low-rank approximation.

$$\begin{array}{lcl} Q & & U_q\ \Sigma_q\ V_q^T \\ V & & U_v\ \Sigma_v\ V_v^T \\ K & \xrightarrow[\text{SVD}]{} & U_k\ \Sigma_k\ V_k^T \\ O & & U_o\ \Sigma_o\ V_o^T \end{array}$$

2. **Custom Layer Implementation:** A custom layer is implemented to replace the original attention layers with the low-rank approximation.

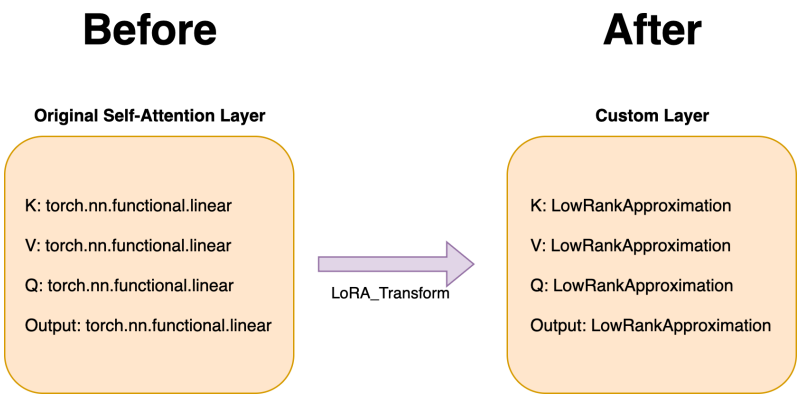


Figure 4.1: Components in attention layers replaced with their custom low-rank approximation

4.2.2 Evaluation Metrics

The approximated model is evaluated based on:

1. ROUGE scores (Lin, 2004): How well does the approximated model perform in comparison to the original BART-Base/Large model on the summarization task? From which rank does the model start to lose performance?

2. Computational efficiency: How does the approximated model compare to the original BART-Base/Large model in terms of computational resources required?

3. Comparing some of the summaries generated by the compressed model with the original BAR-Base/Large model and the reference summaries using the Samsum dataset (Gliwa et al., 2019).

Dialogue
Hannah: Hey, do you have Betty's number? Amanda: Lemme check Hannah: <file_gif> Amanda: Sorry, can't find it. Amanda: Ask Larry Amanda: He called her last time we were at the park together Hannah: I don't know him well Hannah: <file_gif> Amanda: Don't be shy, he's very nice Hannah: If you say so.. Hannah: I'd rather you texted him Amanda: Just text him Hannah: Urgh.. Alright Hannah: Bye Amanda: Bye bye
Summary
Hannah needs Betty's number but Amanda doesn't have it. She needs to contact Larry.

Figure 4.2: SamSum Dataset `test[0]` dialogue and reference summary

4.2.3 Appropriate Rank Selection

In section 2.1.4, the criterion for selecting an appropriate rank k to achieve computational efficiency and storage reduction is established. The criterion is given by:

$$k < \frac{\sqrt{4mn + (m+n)^2} - m - n}{2}$$

where m and n are the dimensions of the original matrix to be approximated with a low-rank representation.

BART-base Model:
For the BART-base model, the attention matrices have dimensions $m = n = 768$. This can be verified by inspecting the model's architecture (Appendix C).
Therefore, the rank k for the approximation should satisfy:

$$k < \frac{\sqrt{4 \times 768 \times 768 + (768+768)^2} - 768 - 768}{2} \approx 318$$

to achieve a reduction in computational complexity and storage requirements.

BART-large Model:
Similarly, for the BART-large model, the attention matrices have dimensions $m = n = 1024$. Therefore, the rank k for the approximation should satisfy:

$$k < \frac{\sqrt{4 \times 1024 \times 1024 + (1024+1024)^2} - 1024 - 1024}{2} \approx 424$$

To determine the feasibility of low-rank approximation for the two BART models, the approximated model at various ranks will be evaluated and then the impact on the ROUGE scores will be observed. This evaluation will illucidate the trade-offs between rank reduction and model performance in the specific case of BART.

Chapter 5

Implementation

This chapter details the implementation of the Retrieval-Augmented Generation (RAG) Chatbot and the low-rank approximation technique applied to the BART model. The implementation process encompasses the development of the chatbot interface, the integration of generator and retriever components, the setup for handling user-uploaded documents, and the deployment of the chatbot. Additionally, the chapter covers the importation and preprocessing of the SamSum dataset, fine-tuning the BART model, implementing custom low-rank layers, and traversing the model to apply low-rank approximations. Each step is meticulously documented with code snippets and explanations to provide a comprehensive understanding of the implementation.

5.1 RAG Chatbot

The development of the RAG Chatbot, named "Study Buddy," aims to address the limitations of current chatbot solutions in educational contexts. Leveraging the Retrieval-Augmented Generation (RAG) model, this chatbot integrates advanced retrieval mechanisms with generative capabilities to provide students with context-specific and detailed responses. This section details the implementation process, including the creation of the user interface, the integration of the generator and retriever components, the setup for handling user-uploaded documents, and the deployment strategy. Through these steps, the Study Buddy chatbot is designed to enhance students' learning experiences by delivering accurate and relevant information efficiently.

5.1.1 Chatbot Interface

The chatbot interface was developed using the `Streamlit` library, which provides a simple and interactive way to create web applications. The interface allows users to interact with the RAG chatbot by entering a question in the input field and receiving the corresponding answer from the model. The chatbot interface was designed to be user-friendly and intuitive, enabling users to easily communicate with the chatbot and obtain relevant information.

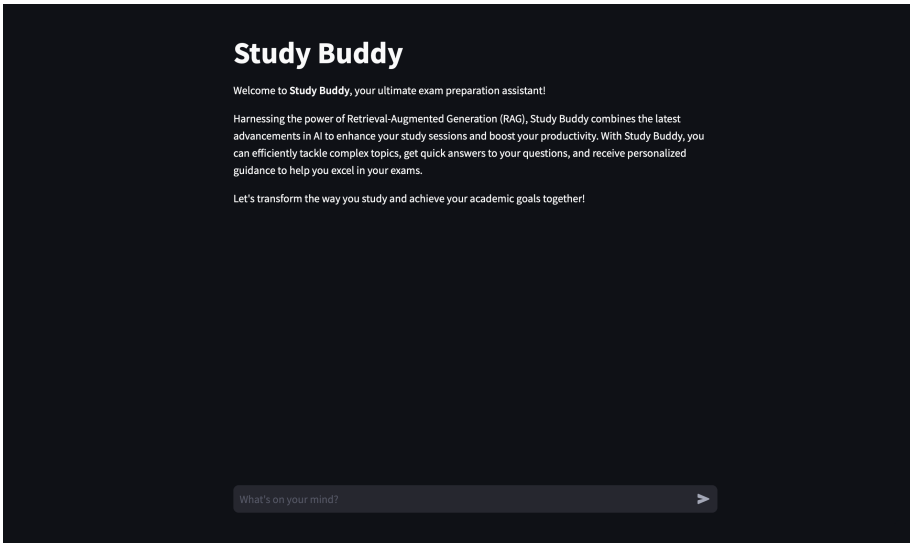


Figure 5.1: RAG Chatbot Interface

The chatbot interface displays a title, description and input field for user queries using the `st.title`, `st.markdown` and `st.chat_input` functions respectively. The title introduces the chatbot as "Study Buddy," while the description provides an overview of the chatbot's capabilities and features. By using the `Streamlit` library, the chatbot interface can be created by just a few lines of code.

5.1.2 Generator Component

The generator component of Study Buddy was implemented using the `Langchain` library, which provides ready-to-use methods to call OpenAI's GPT-4 model. Streamlit reruns the code in the `app.py` file every time a user interacts with the chatbot. The following code snippets demonstrate how the generator calls the GPT-4 model and generates responses to user queries.

```
from langchain_openai import ChatOpenAI

# Cache OpenAI Chat Model for reuse
@st.cache_resource()
def get_chat_model():
    return ChatOpenAI(
        temperature=0.3,
        model='gpt-4',
        streaming=True,
        verbose=True
    )
chat_model_instance = get_chat_model()
```

Listing 1: Caching the OpenAI Chat Model

The code snippet in listing 1 details the implementation of the generator component for the Study Buddy chatbot, leveraging the `Langchain` library to interface with OpenAI's GPT-4 model. The generator component is responsible for producing the responses to user queries.

Key Elements of the Code:

- `ChatOpenAI`: This class from the `langchain_openai` module facilitates interaction with OpenAI's GPT-4 model.
- `@st.cache_resource`: This Streamlit decorator caches the chat model instance, ensuring efficient reuse across multiple interactions without the need to reinitialize it each time.
- `temperature`: A parameter that controls the randomness of the model's responses, with a value of 0.3 indicating a balance between creativity and determinism.
- `model='gpt-4'`: Specifies the use of the GPT-4 model for generating responses.
- `streaming=True`: Enables streaming of the model's output, allowing for real-time response generation.
- `verbose=True`: Provides detailed logs of the model's operations for debugging and transparency.

Explanation of Functionality:

1. **Caching the Chat Model:** The `get_chat_model()` function initializes and caches an instance of the GPT-4 model with specific parameters. By caching this instance, the system ensures that the model does not need to be reloaded each time a user interacts with the chatbot, thereby improving efficiency and response times.
2. **Model Parameters:** The temperature parameter is set to 0.3, which strikes a balance between generating creative responses and maintaining consistency. The `streaming=True` parameter enables the model to stream responses in real-time, providing a more interactive and immediate user experience. The `verbose=True` parameter allows for detailed logging, which is useful for monitoring the chatbot's operations and debugging issues.

This setup ensures that the generator component efficiently interacts with the GPT-4 model to generate responses to user queries. The caching mechanism optimizes performance by reusing the model instance across multiple interactions.

5.1.3 Retriever Component

The retriever component of Study Buddy was implemented using the `Langchain` library to retrieve the $k = 5$ most relevant documents from the Astra DB Vector Store.

```python
from langchain_community.vectorstores import AstraDB
from langchain_openai import OpenAIEmbeddings

# Cache the Astra DB Vector Store for reuse
@st.cache_resource(show_spinner='Connecting to AstraDB')
def get_vector_store():
    vector_store_instance = AstraDB(
        embedding=OpenAIEmbeddings(),
        collection_name="datastax",
        api_endpoint=st.secrets['ASTRA_API_ENDPOINT'],
        token=st.secrets['ASTRA_TOKEN']
    )
    return vector_store_instance
vector_store_instance = get_vector_store()

# Cache the Retriever for reuse
@st.cache_resource(show_spinner='Initializing retriever')
def get_retriever():
    retriever_instance = vector_store_instance.as_retriever(
        search_kwargs={"k": 5}
    )
    return retriever_instance
retriever_instance = get_retriever()
```

Listing 2: Caching the Astra DB Vector Store and Retriever

The code snippet in listing 2 demonstrates the implementation of the retriever component for the Study Buddy chatbot. The `Langchain` library is utilized to connect to the Astra DB Vector Store, which is a database optimized for vector searches. The retriever component is responsible for identifying and retrieving the most relevant documents based on the user's query.

Key Elements of the Code:

- `AstraDB`: This class from the `langchain_community.vectorstores` module is used to create an instance of the vector store.[1] It requires an embedding model, collection name, API endpoint, and a token for authentication.
- `OpenAIEmbeddings`: This class from the `langchain_openai` module provides the embedding model used to convert text into numerical vectors that the vector store can handle.
- `@st.cache_resource`: This Streamlit decorator caches the resources, ensuring that the vector store and retriever instances are reused efficiently across different runs of the application. This caching mechanism helps to optimize performance by avoiding redundant connections and initializations.
- `get_vector_store()`: This function initializes and returns an instance of the vector store by connecting to Astra DB using the provided API endpoint and token.

[1] The vector store was configured in the DataStax Astra DB cloud service to use cosine similarity for semantic searches.

- `get_retriever()`: This function initializes and returns an instance of the retriever, configured to fetch the top $k = 5$ most relevant documents based on the user's query.

Explanation of Functionality:

1. **Connecting to Astra DB:** The `get_vector_store()` function creates a connection to the Astra DB Vector Store using the OpenAI embeddings for document vectorization. The connection details, such as the API endpoint and token, are securely retrieved from the Streamlit secrets configuration.
2. **Initializing the Retriever:** The `get_retriever()` function sets up the retriever to query the vector store. The retriever is configured to return the top 5 relevant documents, which helps in providing contextually accurate responses to user queries.

This setup ensures that the retriever component can efficiently access and retrieve relevant information, enhancing the chatbot's ability to deliver detailed and context-specific answers.

5.1.4 Integration of Generator and Retriever

The generator and retriever components were integrated to provide a comprehensive response to user queries. The following code snippet demonstrates how the generator and retriever components work together to generate responses based on the user's input.

```
from langchain.schema.runnable import RunnableMap

input_data = RunnableMap({
        'context': lambda x: retriever_instance.get_relevant_documents(x['question']),
        'question': lambda x: x['question']
    })
response_chain = input_data | chat_prompt | chat_model_instance
```

Listing 3: Integration of Generator and Retriever

The code snippet in listing 3 illustrates the integration of the generator and retriever components for the Study Buddy chatbot. This integration ensures that user queries are processed using both components to provide contextually accurate and comprehensive responses.

Key Elements of the Code:

- `RunnableMap`: This class is used to map the inputs (context and question) to the necessary functions for processing.
- `retriever_instance.get_relevant_documents`: This function retrieves the most relevant documents from the vector store based on the user's query.
- `chat_prompt`: A predefined template that formats the retrieved context and user question for the generator.

- `chat_model_instance`: The cached instance of the GPT-4 model used for generating responses.

Explanation of Functionality:

1. **Input Data Mapping:** The `RunnableMap` is used to create a mapping of inputs, where the 'context' key is assigned a lambda function that retrieves relevant documents based on the user's question, and the 'question' key simply maps the user's question.

2. **Response Chain Creation:** The `response_chain` is constructed by chaining the input data through the chat prompt and the chat model instance. This chain ensures that the user's question is enriched with relevant context before generating the final response.

3. **Generating Responses:** When invoked, the `response_chain` processes the inputs by first retrieving the relevant documents, then formatting them using the chat prompt, and finally generating a comprehensive response using the GPT-4 model.

This integration of the generator and retriever components ensures that the Study Buddy chatbot can deliver detailed and contextually relevant answers to user queries, enhancing its utility as an exam preparation assistant.

5.1.5 PDF Uploader

The PDF uploader component allows users to upload course materials, lecture notes, or other relevant documents to the Study Buddy chatbot. The uploaded documents are stored in the Astra DB Vector Store, enabling the chatbot to retrieve and reference them when responding to user queries.

```
# Sidebar for document upload
with st.sidebar:
    with st.form('upload_form'):
        uploaded_file = st.file_uploader('Upload a document for enhanced context', type=['pdf'])
        submit_button = st.form_submit_button('Save to AstraDB')
        if submit_button:
            process_and_vectorize_document(uploaded_file, vector_store_instance)
```

Listing 4: PDF Uploader Interface

The code snippet in listing 4 creates an interface within the Streamlit sidebar for uploading PDF documents. This interface allows users to upload course materials, lecture notes, or other relevant documents, which are then processed and stored in the Astra DB Vector Store.

```python
# Function to process and vectorize uploaded documents into Astra DB
def process_and_vectorize_document(uploaded_file, vector_db):
    if uploaded_file is not None:

        # Create a temporary file to store the uploaded document
        temp_dir = tempfile.TemporaryDirectory()
        temp_file_path = os.path.join(temp_dir.name, uploaded_file.name)
        with open(temp_file_path, 'wb') as temp_file:
            temp_file.write(uploaded_file.getvalue())

        # Load the PDF file
        document_pages = []
        pdf_loader = PyPDFLoader(temp_file_path)
        document_pages.extend(pdf_loader.load())

        # Initialize text splitter
        splitter = RecursiveCharacterTextSplitter(
            chunk_size=1500,
            chunk_overlap=100
        )

        # Split the document and add it to the vector store
        split_pages = splitter.split_documents(document_pages)
        vector_db.add_documents(split_pages)
        st.info(f"{len(split_pages)} pages have been loaded into the database.")
```

Listing 5: Processing and Vectorizing Uploaded Documents

The code snippet in 5 defines a function that processes and vectorizes uploaded PDF documents, storing them in the Astra DB Vector Store. This function is crucial for enabling the chatbot to retrieve and reference user-uploaded materials when generating responses.

Key Elements of the Code:

- `tempfile.TemporaryDirectory()`: Creates a temporary directory to store the uploaded file.
- `open(temp_file_path, 'wb')`: Writes the uploaded file to the temporary directory.
- `PyPDFLoader`: Loads the PDF document into memory.
- `RecursiveCharacterTextSplitter`: Splits the document into smaller chunks for vectorization.
- `vector_db.add_documents`: Adds the vectorized document chunks to the Astra DB Vector Store.
- `st.info`: Displays an information message indicating the number of pages processed and loaded into the database.

Explanation of Functionality:

1. **Temporary File Storage:** The uploaded PDF document is stored temporarily to facilitate processing.
2. **Document Loading:** The PDF document is loaded into memory using the `PyPDFLoader` class.
3. **Text Splitting:** The loaded document is split into smaller chunks using the `RecursiveCharacterTextSplitter` class. This step is necessary to manage large documents and prepare them for vectorization.
4. **Vectorization and Storage:** The split document chunks are vectorized and added to the Astra DB Vector Store. This enables efficient retrieval and reference by the chatbot when responding to user queries.
5. **User Feedback:** A message is displayed to inform the user about the number of pages successfully processed and loaded into the database.

This comprehensive approach ensures that the uploaded documents are effectively processed and made available for contextual responses, significantly enhancing the chatbot's utility and relevance.

5.1.6 Deployment

The Study Buddy chatbot was deployed using the Streamlit sharing platform, which provides a simple and efficient way to host and share Streamlit applications. The deployment process involved uploading the application code to the Streamlit sharing platform and configuring the necessary settings to make the chatbot publicly accessible.

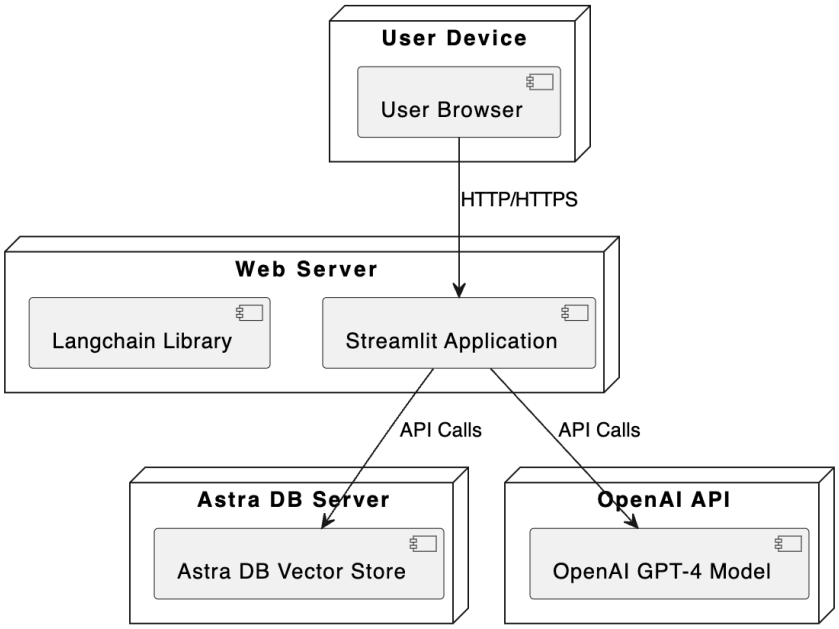


Figure 5.2: Deployment diagram of the Study Buddy RAG chatbot.

The deployment diagram illustrated in Figure 5.2 shows the architecture of the Retrieval-Augmented Generation (RAG) chatbot named *Study Buddy*. This chatbot

is designed to assist students in their studies by providing detailed and contextually relevant answers to their questions.

Components

- **User Device:**
  - *User Browser:* The interface through which users interact with the Study Buddy chatbot. Users enter their queries and receive responses via a web browser.
- **Web Server:**
  - *Streamlit Application:* The front-end of the chatbot, developed using Streamlit. This component handles user interactions, displays the chat interface, and processes user inputs.
  - *Langchain Library:* Integrated within the Streamlit application, this library manages interactions with the OpenAI API and Astra DB. It facilitates the retrieval of relevant documents and the generation of responses.
- **Astra DB Server:**
  - *Astra DB Vector Store:* A highly optimized vector database provided by DataStax, used to store and manage the contextual data required by the chatbot for generating precise and relevant answers.
- **OpenAI API:**
  - *OpenAI GPT-4 Model:* The backend service provided by OpenAI, which generates human-like responses based on the input queries and contextual information retrieved from the Astra DB.

5.1.7 Connections

- **User Device to Web Server:** The user's browser communicates with the Streamlit application over HTTP/HTTPS protocols. This connection allows the user to send queries and receive responses.
- **Web Server to Astra DB Server:** The Streamlit application makes API calls to the Astra DB Vector Store to retrieve relevant documents based on the user's query.
- **Web Server to OpenAI API:** The Streamlit application also makes API calls to the OpenAI GPT-4 Model to generate detailed and contextually enriched responses.

This deployment architecture ensures that the chatbot can efficiently handle user queries by leveraging advanced natural language processing capabilities and a robust database for contextual data retrieval. The integration of these components enables *Study Buddy* to provide accurate and helpful study assistance to users.

5.2 Case Study: Low-Rank Approximation of BART model

This section explores the application of low-rank approximation techniques to the BART model, focusing on both the base and large versions. By reducing the dimensionality of the attention weight matrices, the goal is to decrease computational complexity and storage requirements while maintaining performance in summarization tasks. The implementation process includes importing and preprocessing the SamSum dataset, fine-tuning the BART model, creating custom low-rank layers, and traversing the model to apply these approximations. Evaluation metrics, such as ROUGE scores and computational efficiency, are used to assess the effectiveness of this approach. This case study provides in the effectiveness of low-rank approximation of the BART model for dialogue summarization tasks.

5.2.1 Importing the Model

The BART-base model was imported from the Hugging Face platform using the `transformers` library.

```
model_checkpoint = "facebook/bart-base"
model = AutoModelForSeq2SeqLM.from_pretrained(model_checkpoint)
```

Listing 6: Importing the BART-base model

5.2.2 Dataset Preprocessing

The SamSum dataset was imported and preprocessed to facilitate the training and evaluation of the BART-base model. The dataset comprises conversations crafted and documented by linguists proficient in English, which were subsequently annotated with summaries. The preprocessing steps included tokenization, truncation, and padding to ensure uniform input sizes for the model.

Importing the Dataset

First, the necessary libraries was imported and then the SamSum dataset was loaded using the `datasets` library:

```
from datasets import load_dataset

raw_datasets = load_dataset("samsum")
```

Listing 7: Loading the SamSum dataset

The `load_dataset` function fetches the SamSum dataset, which contains dialogues and their corresponding summaries.

Setting Maximum Lengths

To ensure that the input and target sequences fit within the model's constraints, we set the maximum lengths for input and target sequences:

```
max_input_length = 512
max_target_length = 128
```

Listing 8: Setting maximum lengths for inputs and targets

Here, `max_input_length` is set to 512 tokens and `max_target_length` is set to 128 tokens, which are reasonable lengths for dialogues and summaries respectively.

Preprocessing Function

Next, a preprocessing function is defined to tokenize the inputs and targets. This function truncates and pads the sequences to the specified maximum lengths:

```
def preprocess_function(examples):
    inputs = [doc for doc in examples["dialogue"]]
    model_inputs = tokenizer(inputs, max_length=max_input_length, truncation=True)

    # Setup the tokenizer for targets
    with tokenizer.as_target_tokenizer():
        labels = tokenizer(examples["summary"], max_length=max_target_length, truncation=True)

    model_inputs["labels"] = labels["input_ids"]
    return model_inputs
```

Listing 9: Defining the preprocessing function

This function performs the following steps:

- Tokenizes the dialogue texts.
- Sets up the tokenizer for the target summaries.
- Tokenizes the summaries and adds them to the `model_inputs` dictionary as labels.

Applying the Preprocessing Function

Finally, we applied the preprocessing function to the entire dataset using the `map` method, which processes the dataset in batches:

```
tokenized_datasets = raw_datasets.map(preprocess_function, batched=True)
```

Listing 10: Applying the preprocessing function to the dataset

The `map` method applies the `preprocess_function` to each example in the dataset, ensuring that all dialogues and summaries are properly tokenized, truncated, and padded. This results in a tokenized dataset ready for training and evaluation with the BART-base model.

5.2.3 Fine-Tuning the Model

To establish a baseline for dialogue summarization tasks the model was then fine-tuned on the SamSum dataset using an L4 GPU provided by Google Colab Pro. This was done by defining the training arguments, setting up the trainer, and training the model on the tokenized dataset.

```
batch_size = 4
args = Seq2SeqTrainingArguments(
    "test-dialogue-summarization",
    evaluation_strategy = "epoch",
    learning_rate=2e-5,
    per_device_train_batch_size=batch_size,
    per_device_eval_batch_size=batch_size,
    gradient_accumulation_steps=2,
    weight_decay=0.01,
    save_total_limit=2,
    num_train_epochs=10,
    predict_with_generate=True,
    fp16=True
)

data_collator = DataCollatorForSeq2Seq(tokenizer, model=model)

trainer = Seq2SeqTrainer(
    model,
    args,
    train_dataset=tokenized_datasets["train"],
    eval_dataset=tokenized_datasets["validation"],
    data_collator=data_collator,
    tokenizer=tokenizer,
    compute_metrics=compute_metrics
)

trainer.train()
```

Listing 11: Fine-tuning the BART-base model

The same approach was used to fine-tune the BART-large model. This model was fine-tuned on the same dataset using the same training arguments and setup, with the only difference being the number of epochs due to the larger model size.

5.2.4 Custom Layer Implementation

A custom low-rank layer was implemented to replace the Q, K, V and output matrices typically found in the attention mechanisms of transformers. This implementation utilizes the SVD from the PyTorch library to decompose and subsequently truncate the weight matrices, preserving only the k most significant components.

```python
class LowRankLayer(nn.Module):
    """given a linear layer find low rank decomposition"""
    def __init__(self, rank, full_rank_layer):
        super().__init__()
        self.rank = rank
        self.bias = full_rank_layer.bias
        U, S, Vh = torch.linalg.svd(full_rank_layer.weight, driver = 'gesvd')
        S_diag = torch.diag(S)
        self.U = U[:, :self.rank]
        self.S = S_diag[:self.rank, :self.rank]
        self.Vh = Vh[:self.rank, :]
        self.weight = full_rank_layer.weight

    """forward pass through the low-rank layer"""
    def forward(self, x):
        output_t1 = F.linear(x, self.Vh)
        output_t2 = F.linear(output_t1, self.S)
        output = F.linear(output_t2, self.U, self.bias)
        return output
```

Listing 12: Custom Low-Rank Layer Implementation

The class `LowRankLayer` inherits from `nn.Module`, indicating it is a PyTorch neural network layer. The class contains two main components: the `init` method, which initializes the layer, and the `forward` method, which defines the forward pass.

Initialization In the `init` method, the low-rank decomposition is performed:

- The rank k and the full-rank layer are passed as parameters.
- The bias from the original layer is retained.
- Singular Value Decomposition (SVD) is performed on the weight matrix of the full-rank layer using `torch.linalg.svd`. This decomposes the weight matrix into U, S, and Vh.
- The diagonal matrix S is converted into a diagonal tensor using `torch.diag`.
- The top k singular vectors and values are retained:
  - `self.U` contains the first k columns of U.
  - `self.S` is the top $k \times k$ diagonal part of S.
  - `self.Vh` contains the first k rows of Vh.

Forward Pass The `forward` method performs the forward pass through the low-rank layer:

- The input x is first multiplied by Vh using `F.linear`.
- The result is then multiplied by the diagonal matrix S.
- Finally, the intermediate result is multiplied by U and the bias is added.

This approach maintains the key structural properties of the original layer. By focusing on the most significant singular values and vectors, the low-rank approximation retains the most important features of the weight matrices, thus aiming to preserve the performance of the model to a large extent.

5.2.5 Traversing the Model and Applying Low-Rank Approximation

The BART-base model was traversed to identify the attention layers that could be replaced with the low-rank approximation. The model's architecture was examined to determine the layers that could benefit from the low-rank decomposition.

```
@dataclass
class LowRankConfig:
    rank:int
    target_modules: list[str]

# find the module that ends target suffix
def get_submodules(model, key):
    parent = model.get_submodule(".".join(key.split(".")[:-1]))
    target_name = key.split(".")[-1]
    target = model.get_submodule(key)
    return parent, target, target_name

# this function replaces a target layer with low rank layer
def recursive_setattr(obj, attr, value):
    attr = attr.split('.', 1)
    if len(attr) == 1:
        setattr(obj, attr[0], value)
    else:
        recursive_setattr(getattr(obj, attr[0]), attr[1], value)

# Traversing and modifying the BART model
def loRA_Transform(model, config):
  for key, module in model.named_modules():
      target_module_found = (
        any(key.endswith("." + target_key) for target_key in config.target_modules)
      )
      if target_module_found:
          low_rank_layer = LowRankLayer(config.rank, module)
          #replace target layer with low rank layer
          recursive_setattr(model, key, low_rank_layer)
```

Listing 13: Traversing the BART-base model for low-rank approximation

The `loRA_Transform` function traverses the BART-base model and replaces the target layers with low-rank layers. The function iterates over the model's modules and identifies the layers that match the target layer names specified in the configuration. For each target layer found, a low-rank layer is created using the `LowRankLayer` class and replaces the original layer in the model.

```
"""
Evaluate compressed BART and save metrics
"""
for rank in range(start_rank, end_rank - 1, -step_size):
    gc.collect()  # Garbage collect Python objects
    torch.cuda.empty_cache()  # Clear CUDA cache

    compressed_model = copy.deepcopy(model).to(device)
    config = LowRankConfig(rank=rank, target_modules=["out_proj", "q_proj", "v_proj", "k_proj"])
    loRA_Transform(compressed_model, config)

    # Move the model to GPU if available for evaluation
    if torch.cuda.is_available():
        compressed_model.to('cuda')

    # Define a trainer for evaluation
    trainer = Seq2SeqTrainer(
        model=compressed_model,
        args=args,
        train_dataset=tokenized_datasets["train"],
        eval_dataset=tokenized_datasets["test"],
        data_collator=data_collator,
        tokenizer=tokenizer,
        compute_metrics=compute_metrics
    )

    # Perform evaluation
    eval_results = trainer.evaluate()

    # Store metrics for the new experiment
    for key in metrics_keys:
        if key in eval_results:
            metrics[key].append(eval_results[key])
        else:
            metrics[key].append(None)  # Append None if the key doesn't exist in results
```

Listing 14: Evaluating the low-rank approximated BART models

5.2.6 Evaluation

The compressed BART models were evaluated using the ROUGE metrics to assess their performance in generating summaries. The evaluation involved iterating over different ranks, applying the low-rank approximation, and computing the ROUGE scores for each compressed model by evaluating the summaries generated on the test dataset from the SamSum dataset.

Chapter 6

Results

This chapter presents the outcomes of the two primary areas of investigation in this thesis: the development of the Study Buddy RAG chatbot prototype, and the assesment of the implementation of low-rank approximation on the BART model. This chapter is structured to provide a comprehensive overview of the findings, highlighting the impact of document insertion on the chatbot's performance and the effectiveness of low-rank approximation in optimizing the BART model.

The first section focuses on the Study Buddy RAG chatbot prototype, detailing the internal functionality tests and case studies that assess the impact of inserting relevant documents on the chatbot's response quality. The evaluation emphasizes the improvements in accuracy, relevance, and response quality that result from enhancing the chatbot with additional context.

The second section addresses the low-rank approximation of the BART model. It examines the fine-tuning process, presents ROUGE scores to evaluate summarization performance, and discusses the computational efficiency gains achieved through low-rank approximation. Additionally, the section includes a comparison of generated summaries at various approximation ranks to illustrate the trade-offs between model compression and performance retention.

6.1 RAG Chatbot

The primary goal of developing the Study Buddy RAG chatbot prototype was to gain hands-on experience with working with LLMs and their application in real-world language processing tasks.

The evaluation of the Study Buddy chatbot involved querying the chatbot both before and after inserting relevant documents. The aim was to observe how the inclusion of additional context impacts the chatbot's performance in terms of accuracy, relevance, and response quality. This internal testing was done to verify that the chatbot was ready for potential future scalability and subsequent testing by multiple users.

6.1.1 Functionality Testing

The functionality of the chatbot was tested by ensuring that all components (retriever, generator, and PDF uploader) worked seamlessly together. This involved

verifying the following:

- **PDF Uploader:** Successfully uploading PDF documents and storing them in the Astra DB Vector Store.
- **Retriever Component:** Accurately retrieving relevant documents based on user queries.
- **Generator Component:** Generating coherent and contextually relevant responses using the GPT-4 model.
- **Integration:** Ensuring that the retriever and generator components integrate smoothly to provide comprehensive responses.

The Study Buddy chatbot successfully passed all functionality tests, demonstrating that the components worked together seamlessly to provide accurate and contextually relevant responses to user queries when relevant documents were inserted. Section 6.1.2 provides examples of how the chatbot was tested before and after inserting relevant documents.

6.1.2 Case Study: Impact of Document Insertion on Response Quality

A series of specific queries were made to the Study Buddy chatbot before and after inserting a set of relevant documents. The goal was to observe changes in the quality of responses. The queries were related to complex topics that required detailed and specific information.

Example Query 1: The first query was related to the topic of "orthogonal matrices" and involved asking the chatbot to explain the properties of orthogonal matrices.

- **Before Document Insertion:** The chatbot provided a general and correct overview of the topic.

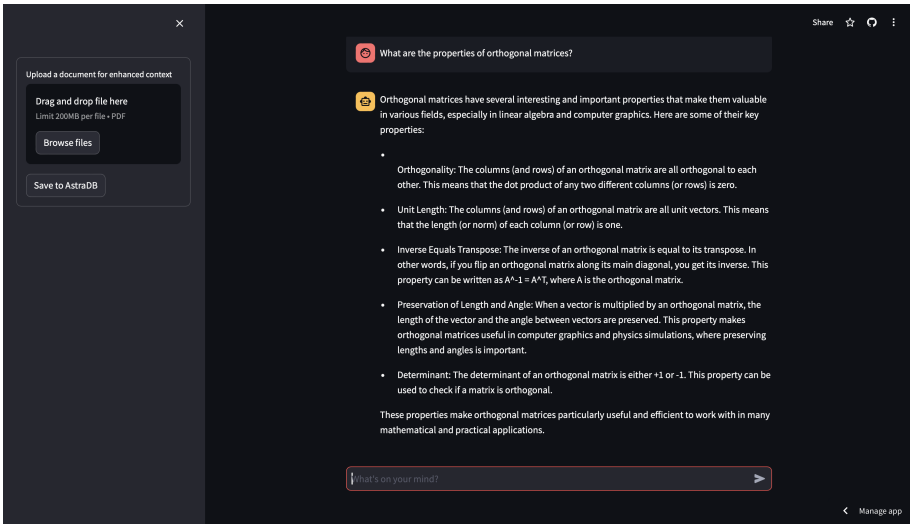


Figure 6.1: Response before insertion of lecture slides on orthogonal matrices

- **After Document Insertion:** The chatbot referenced specific information from the uploaded documents, while providing an accurate response.

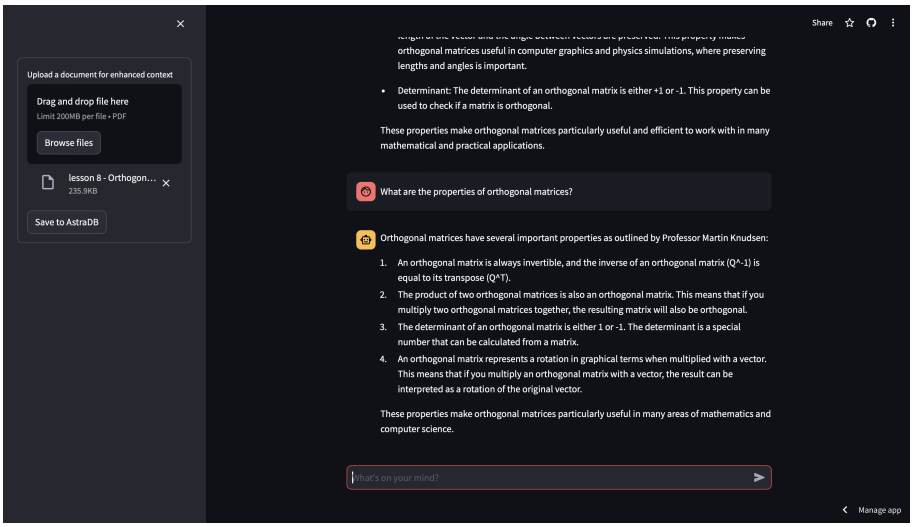


Figure 6.2: Response after insertion of lecture slides on orthogonal matrices

In this case we see that the professor who created the slides was referenced in the response, which was not present in the response before the document insertion. Furthermore, the response is almost identical in both structure and wording to one of the slides in the uploaded document.

Example Query 2: The second query was related to an uncommon topic, e.g. a project I did earlier in my studies, and involved asking the chatbot to explain the purpose of the system developed during the project. This query was chosen to test the chatbot's ability to provide information on topics that OpenAI's GPT-4 model is not familiar with.

- **Before Document Insertion:** A response was produced but lacked detailed information and context about the project.

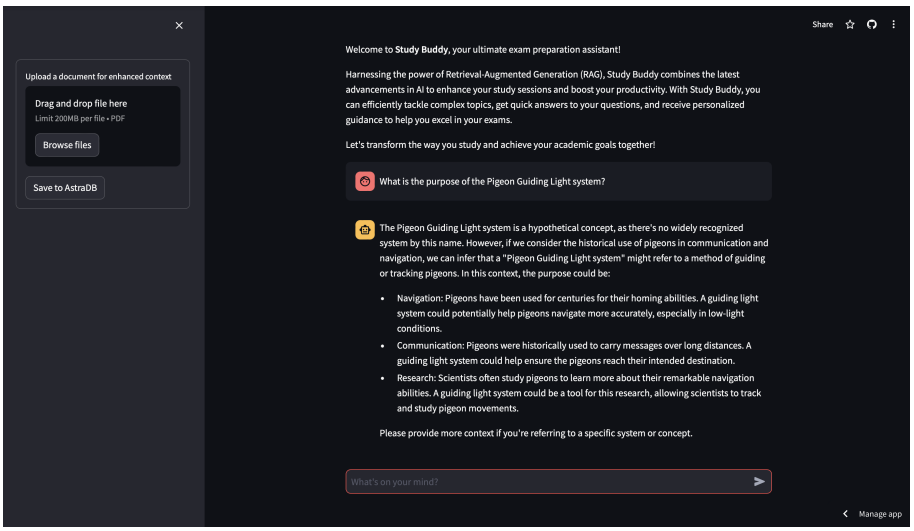


Figure 6.3: Response before insertion of project documentation

- **After Document Insertion:** The response included a detailed explanation in bullet points of the project, providing fast insight to a project that is not accesible for ChatGPT-4.

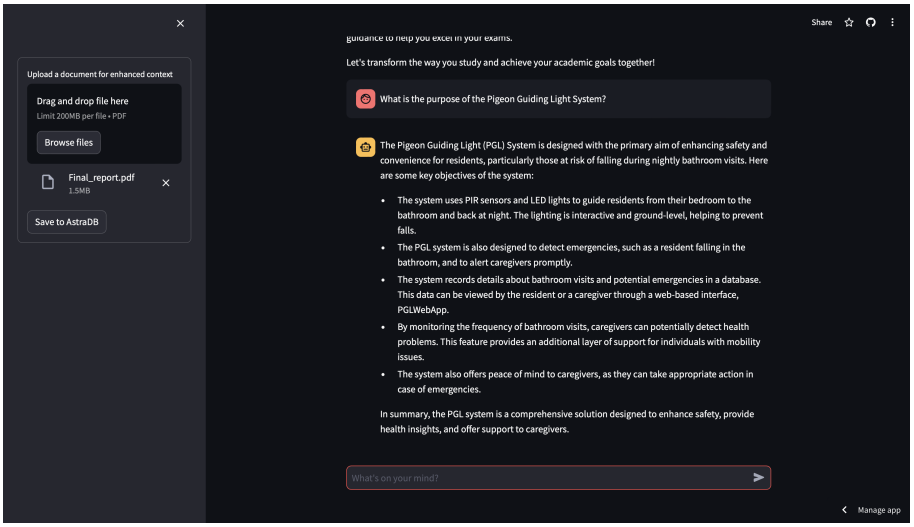


Figure 6.4: Response after insertion of project documentation

6.1.3 Case Study: Impact of Document Insertion on Unrelated Query Response

- **Document Insertion of a Single Document:** After inserting only a document on "orthogonal matrices," a query was made on a different topic, e.g. "eigenvectors". The chatbot was not able to provide a relevant response, indicating that the document insertion did impact the chatbot's ability to respond to unrelated queries.

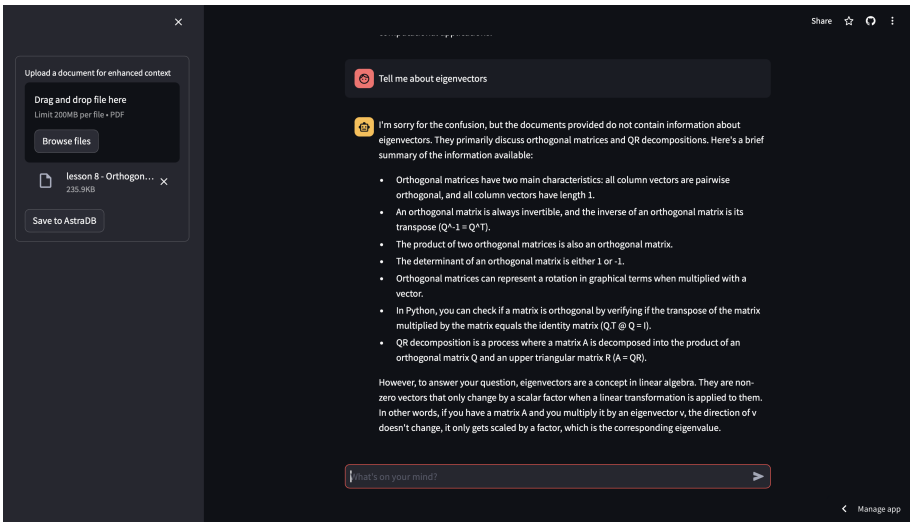


Figure 6.5: Response to an unrelated query after insertion of orthogonal matrices slides

- **Document Insertion of Multiple Documents:** After inserting another document on "eigenvectors," the chatbot was able to provide a relevant re-

sponse to the same query on "eigenvectors." This demonstrates that the chatbot's ability to respond to queries on different topics can be improved by inserting the relevant documents.

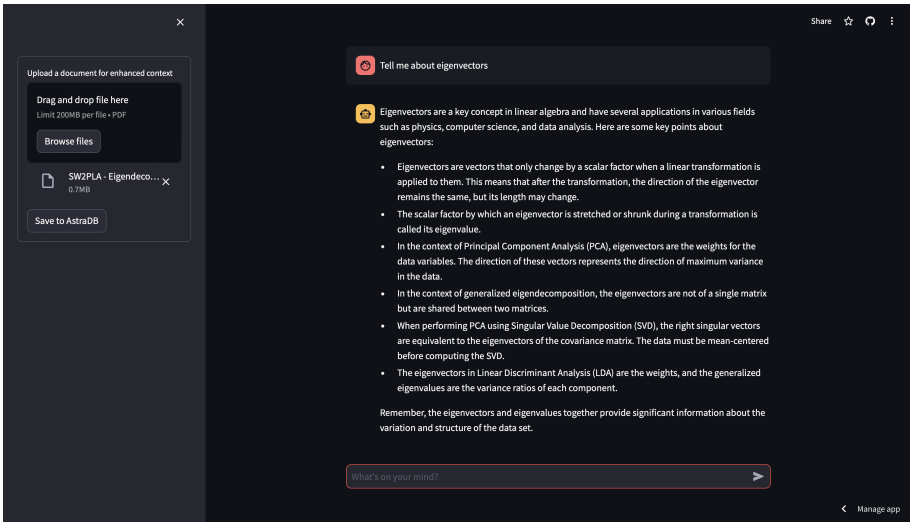


Figure 6.6: Response to query after insertion of eigenvectors slides

It should be noticed that the response is tailored after the inserted document, such that it also covers PCA and SVD, which are specifically stated as related topics in the slides.

6.2 Case Study: Low Rank Approximation of BART Model

The evaluation of fine-tuning and then compressing the BART-base and BART-large model using low-rank approximation reveals the following:

6.2.1 Fine-tuning the BART Model

The fine-tuning of the BART model on the SamSum dataset provided the following results: The fine-tuning of BART-base with 10 epochs took approximately

Epoch	Training Loss	Validation Loss	Rouge1	Rouge2	Rougel	Rougelsum	Gen Len
0	1.815700	1.543361	47.514900	24.586800	40.399100	43.972500	18.091700
2	1.458700	**1.489950**	48.360500	25.598600	41.272800	44.942500	17.880200
4	1.266000	1.495493	48.338000	25.252300	41.116400	44.727300	18.094100
6	1.123200	1.523768	48.887200	25.618700	41.335700	**45.155400**	18.206600
8	1.041000	1.528558	**49.063600**	**25.955900**	**41.537300**	45.311800	**18.343500**
9	**1.019900**	1.539690	48.923000	25.675400	41.485100	45.143400	18.327600

Table 6.1: Training and Validation Metrics of Fine-Tuning BART-base across Epochs

1 hour. It can be observed from Table 6.1 that the BART-base model achieved the best training loss, ROUGE-1, ROUGE-2, and ROUGE-L scores at epoch 8. The

generated length of the summaries also peaked at epoch 8, indicating that the model performed best at this stage. The ROUGE-2 score, however, was highest at epoch 9, suggesting that the model peaked in bigram overlap at this point. The training loss decreased consistently across epochs, indicating that the model was learning from the training data.

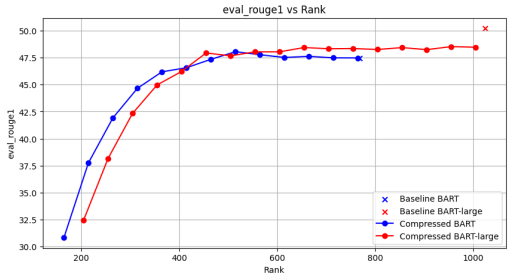
Epoch	Training Loss	Validation Loss	Rouge1	Rouge2	Rougel	Rougelsum	Gen Len
0	1.022900	1.489733	48.960100	26.305700	41.343800	44.952700	**19.270200**
2	1.010700	**1.386865**	**50.152900**	**27.821400**	**43.000900**	**46.561300**	18.358200
4	**0.812600**	1.429855	50.003400	26.972100	42.172400	45.977900	18.588000

Table 6.2: Training and Validation Metrics of Fine-Tuning BART-large across Epochs

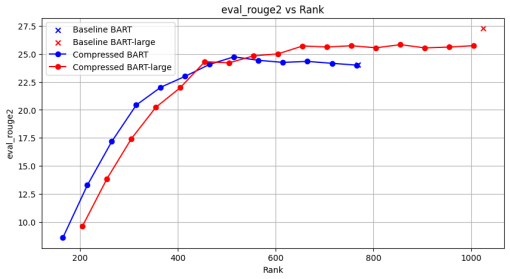
The fine-tuning of BART-large with 5 epochs took approximately 1 hour and 50 minuters. Table 6.2 shows that the BART-large model achieved the best validation loss, ROUGE-1, ROUGE-2, ROUGE-L, and ROUGE-Lsum scores at epoch 2 indicating that the model performed best at this stage. The training loss decreased consistently across epochs, also indicating that the model was learning from the training data.

6.2.2 ROUGE scores

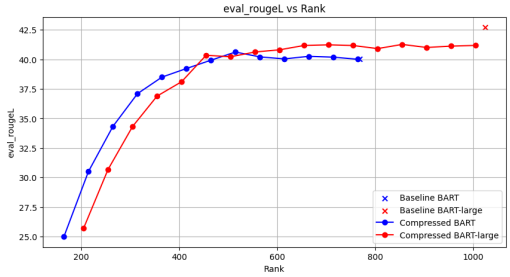
The low-rank approximations upholds comparable ROUGE scores until a certain rank, after which the scores begin to decline. This indicates that the model's performance is preserved up to a certain rank, beyond which the approximation starts to impact the summarization quality. The following figures illustrate the ROUGE scores for various ranks:



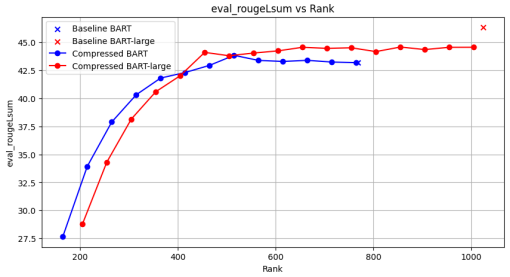
(a) ROUGE-1 scores.



(b) ROUGE-2 scores.



(c) ROUGE-L scores.



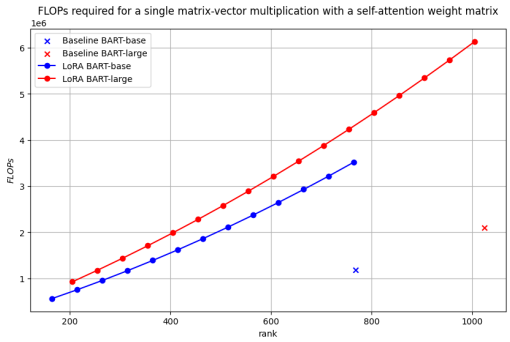
(d) ROUGE-Lsum scores.

Figure 6.7: ROUGE scores for different ranks.

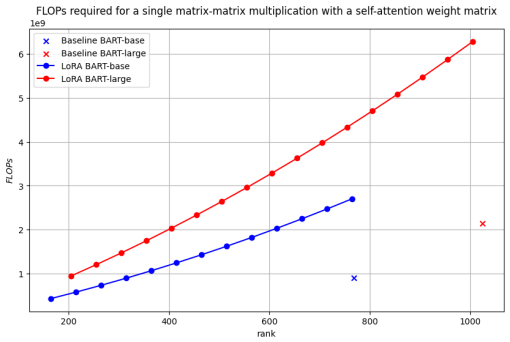
As depicted in Figure 6.7, the ROUGE-1, ROUGE-2, ROUGE-L, and ROUGE-Lsum scores for BART-base maintain similar level of performance, compared to its original fine-tuned model, up to a rank of around $r = 510$. However, it is important to note that for the BART-large model, there is an immediate decline in performance even when the model is low-rank approximated near full rank. Beyond that, similarly for BART-large, after the rank of around $r = 510$, a rapid degradation in scores is observed, indicating the limitations of the low-rank approximation in preserving the model's quality.

6.2.3 Computational Efficiency

The low-rank approximation of the BART model succeeds in reducing the computational complexity and storage requirements of the model. The following figures illustrate the computational efficiency of the low-rank approximations:



(a) Vector-Matrix Multiplication FLOPs.



(b) Matrix-Matrix Multiplication FLOPs.

Figure 6.8: FLOPs required for a vector-matrix multiplication and matrix-matrix multiplication of a single self-attention weight matrix in BART.

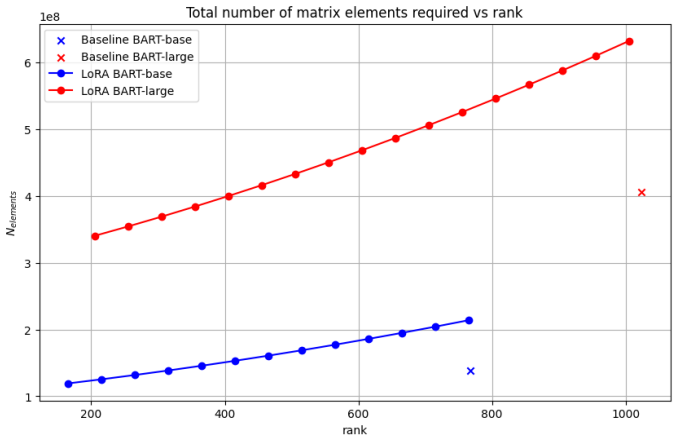


Figure 6.9: Number of elements in the weight matrices of BART-base and BART-large depending on rank.

Figure 6.8 show that the low-rank approximation can significantly reduce the number of FLOPs required for vector-matrix and matrix-matrix multiplications,

indicating a substantial improvement in computational efficiency after a certain threshhold of $r = 318$ and $r = 424$ for BART-base and BART-large, respectively. The number of elements in the weight matrices of BART-base and BART-large also decreases with the reduction in rank after the same threshhold, further demonstrating the efficiency gains achieved through low-rank approximation as depicted in figure 6.9.

6.2.4 Comparing Summaries

The analysis of summaries produced by the BART-base model at various ranks demonstrates how low-rank approximation affects the quality of generated summaries.
The following example illustrates summaries generated at different ranks:
At full rank ($r_{\text{out}} = 768, r_v = 767, r_k = r_q = 766$), both with and without the application of `LoRA_Transform`, the summary is as follows:

> *Hannah is looking for Betty's number. Amanda can't find it. Larry called her last time they were at the park together.*

While this summary differs from the reference summary (see Figure 4.2), it still captures the essence of the dialogue. For simplicity let us call this the *original phrasing* as we wish to compare the summaries to this as we reduce the rank.

The quality of the generated summaries demonstrates a noticeable decline as the rank of the low-rank approximation decreases. Initially, at higher ranks close to full rank, the summaries remain somewhat coherent and consistent, closely matching the original summary at full rank. However, as the rank decreases to $r = 510$, the summaries start to lose details and exhibit increased errors and inconsistencies.

At ranks 765 through 730, the summary remains consistent:

> *Hannah is looking for Betty's number. Amanda can't find it. Larry called her last time they were at the park together.*

However, a shift is noticed at rank 725 through 720:

> *Hannah doesn't know Betty's number. She texted Larry last time they were at the park together.*

This slight change introduces minor variations in the narrative.

The original phrasing returns from rank 715 through 655:

> *Hannah is looking for Betty's number. Amanda can't find it. Larry called her last time they were at the park together.*

By rank 650 through 490, the summary shifts between losing a critical detail at times and the original phrasing:

> *Hannah is looking for Betty's number. Larry called her last time they were at the park together.*

As the rank decreases further, the summary begins to diverge more substantially:

> Rank 485: *Amanda can't find Betty's number. Hannah doesn't know Larry.*
>
> Rank 470: *Betty's number is not in Hannah's phone.*

By the time the rank reaches around 450, the summaries are severely affected:

> Rank 435: *Hannah has Betty's number. She texted Larry last time they were at the park together.*

Further reduction in rank leads to highly inconsistent and erroneous summaries:

> Rank 325: *Amanda can't find Betty's number. Hannah and Amanda don't know each other.*
>
> Rank 310: *Amanda and Hannah don't know each other.*

At the lowest ranks, the summaries become almost nonsensical:

> Rank 205: *Amanda, Hannah and Amanda are going to meet up with Larry.*
>
> Rank 175: *Amanda and Amanda don't know each other person in the park.*
>
> Rank 150: *Amanda and Amanda are going to the park.*

These observations further indicates that the low-rank approximation method can maintain summarization quality up to a certain rank, beyond which the coherence and accuracy of the summaries degrade significantly. This highlights the trade-off between computational efficiency and model performance in the context of low-rank approximation of the BART model.

Chapter 7

Discussion

This chapter presents a detailed discussion of the results obtained from the experiments conducted in this thesis. The primary focus is on the performance and potential applications of a Retrieval-Augmented Generation (RAG) Chatbot and the effects of applying low-rank approximation techniques to the BART model. The discussion is organized into several sections, beginning with the interpretation of results, followed by an analysis of the limitations encountered during the research, and concluding with suggestions for future work. By critically evaluating the outcomes, this chapter aims to provide a comprehensive understanding of the contributions and implications of the research findings.

7.1 Interpretation of Results

The results obtained from the experiments conducted in this thesis offer substantial insights into the potential applications of a Retrieval-Augmented Generation (RAG) Chatbot and the low-rank approximation technique applied to the BART model. This section elaborates on the implications of these findings.

7.1.1 RAG Chatbot

The prototype of the RAG Chatbot, developed as a 'Study Buddy,' demonstrated notable enhancements in response quality, particularly in terms of relevance, following the integration of document retrieval with text generation components. Various scenarios were evaluated to assess the chatbot's functionality and its capacity to handle diverse query types.

Specific Topic Queries

The chatbot's responses to specific topic queries, such as questions about orthogonal matrices, exhibited significant improvements in contextually appropriate detail after the incorporation of relevant documents. This indicates that the retriever component effectively supplements the generator with pertinent content, thereby making the responses more informative and contextually appropriate for students seeking to learn about specific topics.

Unfamiliar Topics

In instances where the topic was not commonly addressed by standard language models, such as a previous student project Thomsen et al., 2022, the chatbot initially provided inferred responses. However, upon the insertion of specific project documentation, the factual correctness and relevance of the responses improved markedly. This demonstrates the RAG model's capability to adapt to specialized and potentially unfamiliar content through effective document retrieval. This feature is particularly beneficial in educational settings where course-specific material, which is not widely available online and thus not covered by pre-trained language models, can be incorporated to enhance the chatbot's responses.

Impact on Unrelated Queries

The insertion of a single document related to a specific topic degraded the chatbot's performance on unrelated queries. Conversely, the presence of multiple relevant documents enhanced the chatbot's ability to provide accurate and relevant responses across various topics covered by the documents. This indicates the robustness and scalability of the RAG architecture in handling diverse queries when equipped with a comprehensive document repository. This also underscores the importance of maintaining a well-curated and diverse document repository to ensure the chatbot's ability to generalize effectively across different topics.

Additionally, if the goal of the chatbot is to provide accurate and relevant responses on a narrow set of topics, the results indicate the chatbot's potential to limit responses to the scope of the inserted documents. This is akin to a domain-specific chatbot, as discussed in Appendix B. Such specialization is particularly useful in educational settings where the chatbot is designed to assist students with course-specific queries.

7.1.2 Case Study: Low-Rank Approximation of BART Model

The application of low-rank approximation to the BART model aimed to assess its effectiveness in reducing computational complexity while maintaining performance. The results were analyzed based on ROUGE score metrics and by comparing the generated summaries of the approximated model with those of the original BART model.

Fine-Tuning and Evaluation

The fine-tuning of the BART models on the SamSum dataset demonstrated that both models could learn from the training data. BART-large outperformed BART-base in terms of ROUGE scores, which was expected due to its larger capacity and more parameters. However, considering that BART-large has more than twice as many parameters as BART-base, and the performance difference was minimal, it suggests that BART-base is a more viable option for summarization tasks when computational resources are limited. Although BART-base was fine-tuned over more epochs than BART-large, the results indicate that the model's performance was not significantly affected by the number of epochs.

Performance Retention

The ROUGE scores remained stable until a rank of approximately 510, beyond which a decline was observed. As discussed in Section 4.2.3, to low-rank approximate the self-attention matrices, a maximum rank of $r = 318$ for BART-base and $r = 424$ for BART-large is needed to achieve a reduction in computational complexity and storage requirements. However, the results indicate that summarization quality declines significantly before this threshold. This suggests that while low-rank approximation can reduce the model's computational complexity and storage requirements, it comes at the cost of summarization quality, thereby limiting its practical utility in this context.

Computational Efficiency

The reduction in the size of attention matrices leads to a noticeable decrease in computational requirements. The approximated model demonstrated improved efficiency in terms of storage and processing time, making it more suitable for deployment in resource-constrained environments. As an academic exercise exploring a technique taught in linear algebra courses for computer and software engineering students, the low-rank approximation of the BART model provided valuable insights into the trade-offs between computational efficiency and performance in large language models. The results underscore the importance of balancing these factors when optimizing LLMs for real-world applications.

7.2 Limitations and Future Work

This section outlines the limitations encountered during the development and evaluation of the RAG Chatbot and the low-rank approximation applied to the BART model. It also proposes directions for future research and development to address these limitations and enhance the performance and applicability of the solutions. The discussion is divided into two main parts: the limitations and future work for the RAG Chatbot, and the limitations and future work for the low-rank approximation of the BART model.

7.2.1 RAG Chatbot

This subsection discusses the limitations specific to the RAG Chatbot, focusing on data quality, conversation context, handling unrelated queries, and deployment scalability. Recommendations for future improvements are also provided.

Ensuring Data Quality in RAG Implementations

The quality of data retrieved by a RAG implementation depends entirely on the data it can access. If the underlying source systems contain outdated, incomplete, or biased information, the RAG implementation cannot detect these flaws. It will merely retrieve and pass this flawed data to the language model, which then generates the final output. Therefore, ensuring that the document repository is comprehensive and up-to-date is crucial for maintaining high performance.

Maintaining Conversation Context

While testing the chatbot, it was observed that it was unable to maintain the context of the conversation across multiple queries. This was due to the lack of a memory component that could store the conversation history and use it to provide more coherent responses. Implementing a memory component or a dialogue manager could enhance the chatbot's conversational abilities and make it more engaging for users. However, this was not the primary focus of the study and thus was not implemented in the current version of the chatbot prototype.

Unrelated Queries

As noted in Section 6.1.3, the chatbot's performance on unrelated queries deteriorated after inserting a single document by keeping on topic with the inserted document. This limitation could be viewed as a feature depending on the desired use case of the chatbot. However, if this is the case, a more robust system for rejecting unrelated queries should be implemented. Currently, the prototype does not have a mechanism for detecting and rejecting queries that are outside the scope of the inserted documents. This is a trade-off between specialization and generalization to consider when taking the prototype to the next development phase.

Deployment Scalability

The current implementation of the RAG chatbot prototype is designed for educational purposes for a single user. Although the Study Buddy demonstrated improved performance in a controlled environment, scaling this solution for broader deployment in real-world educational settings presents additional challenges. These include managing large-scale document repositories and ensuring seamless integration with existing educational platforms. Additionally, the ethical implications of deploying AI-driven chatbots in education, such as data privacy (GDPR), must be considered. Future work should focus on addressing these challenges to enhance the scalability and deployability of the RAG chatbot in real-world educational environments.

7.2.2 Case Study: Low-Rank Approximation

This subsection highlights the limitations of the low-rank approximation applied to the BART model, including summarization quality, comparative studies, dataset and task specificity, and computational resources. Future research directions are suggested to address these issues.

Summarization Quality

The main drawback of applying the low-rank approximation technique to the BART model is the decline in summarization quality as the rank decreases. Although this approach can significantly reduce computational complexity and storage requirements, the performance drop before reaching a practically useful rank limits its practical applicability. Future research should investigate alternative methods to optimize BART for resource-constrained environments without compromising performance.

Comparative Studies

The evaluation of low-rank approximation was limited to the BART model, and the results may not be generalizable to other large language models (LLMs). Future work should conduct comparative studies across different models to assess the effectiveness of low-rank approximation in optimizing various architectures. This would provide a more comprehensive understanding of the trade-offs between computational efficiency and performance in different LLMs.

Dataset and Task Specificity

The evaluation of low-rank approximation was performed using the SamSum dataset for abstractive summarization. Results may differ when applied to other datasets or tasks, such as question answering or text generation. Future work should explore the effects of low-rank approximation on various datasets and tasks to assess its generalizability and effectiveness across different applications.

Computational Resources

The initial fine-tuning and evaluation of large models like BART demand substantial computational resources, posing a barrier for researchers and practitioners with limited access to high-performance computing facilities. Therefore, it is recommended that future work ensures the availability of adequate computational resources for these tasks.

7.3 Conclusion

This discussion highlights the significant contributions of the RAG chatbot and low-rank approximation in advancing the efficiency and applicability of AI-driven solutions. While challenges remain, the promising results pave the way for further exploration and development in these areas, offering substantial potential for optimizing large language models and enhancing their deployment in resource-constrained environments.

Chapter 8

Conclusion

This thesis set out to bridge the gap between theoretical linear algebra concepts and their practical applications in computer engineering, particularly in the context of LLMs. The research aimed to deepen the understanding of the linear algebraic foundations of LLMs, gain practical experience through the development of a RAG chatbot prototype, and assess the effectiveness of low-rank approximation techniques in reducing the computational complexity and storage requirements of the BART model while maintaining performance on summarization tasks.

8.1 Summary of Key Findings

This section summarizes the key findings and contributions of this thesis:

8.1.1 Theoretical Exploration

The investigation into the mathematical principles underpinning LLMs, and especially the Transformer model architecture, highlighted the critical role of linear algebraic operations such as matrix multiplication. This exploration provided a detailed understanding of how concepts like low-rank approximations by SVD can be utilized to enhance model efficiency.

8.1.2 RAG Chatbot Development

The development of the RAG chatbot prototype, Study Buddy, showcased the practical application of LLMs in educational settings. The integration of document retrieval with text generation components significantly improved the relevance and accuracy of responses to specific topic queries. This practical implementation demonstrated the potential of RAG models to supplement traditional educational tools by providing contextually appropriate and detailed information.

8.1.3 Low-Rank Approximation of BART Model

The case study on the low-rank approximation of the BART model underscored the trade-offs between computational efficiency and model performance. While low-rank approximation can significantly reduce computational complexity and storage

requirements, it also led to a decline in summarization quality beyond certain rank thresholds. This study highlighted the importance of balancing performance retention with resource optimization in the application of LLMs.

8.2 Contributions to the Field

This thesis makes several contributions to the field of natural language processing and artificial intelligence:

1. **Integration of Theoretical and Practical Knowledge**: By linking linear algebra concepts with their practical applications in LLMs, this research provides a comprehensive framework for understanding and optimizing these models. This integration is crucial for advancing both educational methodologies and practical implementations in AI.

2. **Development of a Practical Educational Tool**: The "Study Buddy" RAG chatbot serves as a prototype for future educational tools that leverage AI to enhance learning experiences. Its successful implementation demonstrates the feasibility of using advanced AI models in real-world educational settings.

3. **Optimization Techniques for LLMs**: The exploration of low-rank approximation techniques provides valuable insights into optimizing LLMs. The findings contribute to ongoing research aimed at making LLMs more accessible and efficient, especially in resource-constrained environments.

8.3 Recommendations for Future Research

While this thesis has made significant strides in understanding and applying linear algebra in the context of LLMs, several areas warrant further investigation:

1. **Extended Comparative Studies**: Future research should conduct comparative studies across various LLM architectures to assess the generalizability of low-rank approximation techniques. This would provide a more comprehensive understanding of the trade-offs involved in optimizing different models.

2. **Broader Dataset and Task Evaluation**: Evaluating the effectiveness of low-rank approximation on diverse datasets and tasks, beyond summarization, would help establish its applicability across different NLP applications. This includes tasks such as question answering, text generation, and translation.

3. **Enhanced Chatbot Functionality**: Further development of the RAG chatbot should focus on enhancing its scalability and integration with existing educational platforms. Additionally, addressing the ethical implications of AI-driven educational tools, such as data privacy and bias, is crucial for their broader deployment.

In conclusion, this thesis has demonstrated the interconnectedness of theoretical and practical aspects of linear algebra and LLMs. The findings not only advance the understanding of these complex models but also pave the way for future innovations in AI-driven educational tools and optimization strategies for LLMs.

Bibliography

Vaswani, Ashish et al. (2017). *Attention Is All You Need*. arXiv: 1706.03762 [cs.CL].

Naveed, Humza et al. (2024). *A Comprehensive Overview of Large Language Models*. arXiv: 2307.06435 [cs.CL].

Hu, Edward J. et al. (2021). *LoRA: Low-Rank Adaptation of Large Language Models*. arXiv: 2106.09685 [cs.CL].

Valipour, Mojtaba et al. (May 2023). "DyLoRA: Parameter-Efficient Tuning of Pre-trained Models using Dynamic Search-Free Low-Rank Adaptation". In: *Proceedings of the 17th Conference of the European Chapter of the Association for Computational Linguistics*. Ed. by Andreas Vlachos and Isabelle Augenstein. Dubrovnik, Croatia: Association for Computational Linguistics, pp. 3274–3287. DOI: 10.18653/v1/2023.eacl-main.239. URL: https://aclanthology.org/2023.eacl-main.239.

Zhang, Qingru et al. (2023). "Adaptive Budget Allocation for Parameter-Efficient Fine-Tuning". In: *The Eleventh International Conference on Learning Representations*. URL: https://openreview.net/forum?id=lq62uWRJjiY.

Chavan, Arnav et al. (2024). *One-for-All: Generalized LoRA for Parameter-Efficient Fine-tuning*. URL: https://openreview.net/forum?id=K7KQkiHanD.

Xu, Mingxue, Yao Lei Xu, and Danilo P. Mandic (2023). *TensorGPT: Efficient Compression of the Embedding Layer in LLMs based on the Tensor-Train Decomposition*. arXiv: 2307.00526 [cs.CL].

Lewis, Mike et al. (2019). *BART: Denoising Sequence-to-Sequence Pre-training for Natural Language Generation, Translation, and Comprehension*. arXiv: 1910.13461 [cs.CL].

Lv, Kai et al. (2023). *Full Parameter Fine-tuning for Large Language Models with Limited Resources*. arXiv: 2306.09782 [cs.CL].

Yang, Chengrun et al. (2023). *Large Language Models as Optimizers*. arXiv: 2309.03409 [cs.LG].

Hanifi, Khadija, Orcun Cetin, and Cemal Yilmaz (Oct. 2023). "On ChatGPT: Perspectives from Software Engineering Students". In: pp. 196–205. DOI: 10.1109/QRS60937.2023.00028.

Zhu, Xunyu et al. (2023). *A Survey on Model Compression for Large Language Models*. arXiv: 2308.07633 [cs.CL].

Lewis, Patrick et al. (2020). *Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks*. arXiv: 2005.11401 [cs.CL].

Gliwa, Bogdan et al. (Nov. 2019). "SAMSum Corpus: A Human-annotated Dialogue Dataset for Abstractive Summarization". In: *Proceedings of the 2nd Workshop on New Frontiers in Summarization*. Hong Kong, China: Association for Com-

putational Linguistics, pp. 70–79. DOI: 10.18653/v1/D19-5409. URL: https://www.aclweb.org/anthology/D19-5409.

Lin, Chin-Yew (July 2004). "ROUGE: A Package for Automatic Evaluation of Summaries". In: *Text Summarization Branches Out*. Barcelona, Spain: Association for Computational Linguistics, pp. 74–81. URL: https://aclanthology.org/W04-1013.

Aghajanyan, Armen, Luke Zettlemoyer, and Sonal Gupta (2020). *Intrinsic Dimensionality Explains the Effectiveness of Language Model Fine-Tuning*. arXiv: 2012.13255 [cs.LG].

Thomsen, Andreas Kaag et al. (2022). *Pigeon Guiding Light: Final Report*. Computer Engineering Semester Project II Report.

Devlin, Jacob et al. (2019). *BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding*. arXiv: 1810.04805 [cs.CL].

Steen, Heidi and Dan Wahin (2024). *Retrieval Augmented Generation (RAG) in Azure AI Search*. https://learn.microsoft.com/en-us/azure/search/retrieval-augmented-generation-overview [Accessed: 2024-05-22].

Swann, Andrew (2022). *Numerisk Lineær Algebra*.

Appendix A

Educational Synergy in Teaching and Research

Introduction

This appendix outlines the dual role undertaken during the semester as both a Teaching Assistant (TA) in the course "SW2PLA: Practical Linear Algebra for Software Engineers" and conducting research for a Bachelor's thesis. This unique position facilitated bridging theoretical understanding and practical application, fostering an educational environment where concepts in linear algebra, taught in the SW2PLA course, were directly linked to research in Large Language Models (LLMs).

Dual Role and Synergistic Learning

As a Teaching Assistant in the SW2PLA course, responsibilities included assisting students with practical exercises, grading assignments, and engaging in discussions about linear algebra during lectures. These activities aimed to deepen students' understanding of linear algebra's application in modern computational technologies, particularly within the realms of basic AI and elementary machine learning. Simultaneously, the bachelor's thesis focused on integrating linear algebra within the development and optimization of LLMs, specifically assessing the effectiveness of low-rank approximation by SVD of Facebook's BART-model's attention weight matrices. This dual role provided a new perspective on the practical applications of linear algebra in AI research and a further understanding of the linear algebra concepts taught in the SW2PLA course.

Project Overview in SW2PLA

The final project for the SW2PLA course required students to undertake a practical application of eigendecomposition or SVD. The project aimed to provide hands-on experience with linear algebra's potent applications, offering several avenues for exploration:

1. **Recommender Systems**: Using SVD to predict user preferences based on past interactions.
2. **PageRank Algorithm**: Implementing an eigenvector-based approach to rank web pages.
3. **Image Compression**: Applying SVD to reduce the dimensionality of image data without losing significant information.
4. **Fibonacci Algorithm**: Using eigendecomposition to compute the Fibonacci algorithm without recursion.
5. **Eigenportfolios**: Employing eigendecomposition for optimized stock portfolio selection.
6. **Facial Recognition**: Utilizing PCA (a form of eigendecomposition) to identify and classify facial features in images.
7. **Open Project**: Any project idea involving Eigendecomposition or SVD, whether applying these techniques to optimize algorithms, enhance data processing, or explore new applications, approaches, or innovative use of linear algebra.

Integration with Bachelor's Thesis

During the course, the methodologies and intermediate findings of the bachelor's thesis were presented to the class. This presentation served as a practical demonstration of how linear algebra is utilized within the development and optimization of LLMs—particularly through techniques like low-rank approximations and matrix factorizations—to enhance computational efficiency and model scalability. This further provided students with real-world applications of their theoretical studies.

Student Projects and Feedback

The culmination of the course was the student presentations, where they demonstrated their projects' outcomes. This session provided an interactive platform for peer learning and feedback, where students could showcase their application of linear algebra in various projects. The presentation alongside the students' highlighted the reciprocal nature of teaching/learning and research.

3rd place at the 2024 CLAI Poster Competition

In addition to sharing findings within the SW2PLA course, the methodologies and intermediate results of the bachelor's thesis were presented at a poster competition held during the 2024 CLAI workshop. This presentation allowed engagement with a broader audience of experts and peers in the field, receiving valuable feedback that furthered research endeavors.

The highlight of this event was the recognition of the work, securing 3rd place in the competition. This achievement not only affirmed the quality and relevance of the research in the field of AI but also provided a platform to showcase the practical applications of linear algebra in optimizing Large Language Models. This external validation brought additional perspective to the educational content delivered in the SW2PLA course, enhancing the overall teaching and learning experience.

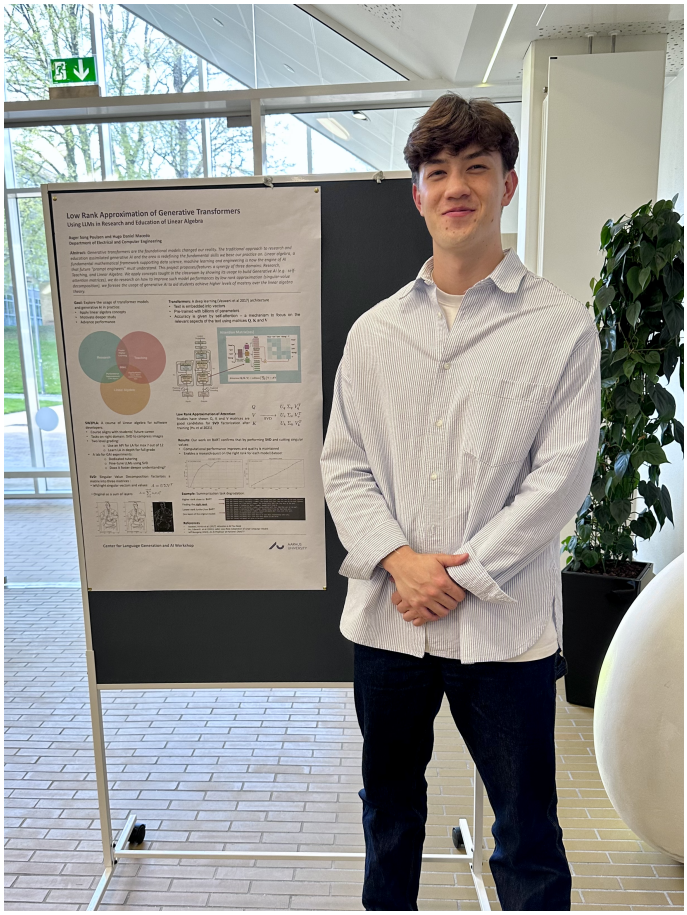


Figure A.1: 3rd place in the CLAI Poster Competition

It was an honor to represent the Department of Electrical and Computer Engineering (ECE) at such a forum. This accolade not only represents a personal achievement but also served as valuable practice for presenting research findings to a broader audience, including students and faculty members from various disciplines.

Conclusion

The involvement in the SW2PLA course as a TA, while simultaneously conducting research for the bachelor's thesis, created a rich educational synergy. This experience not only enhanced the understanding and teaching of linear algebra but also allowed for the direct application and evaluation of theoretical concepts in practical, research-based applications. It underscores the importance of an integrated approach in education, where teaching responsibilities and academic research com-

plement and enrich each other, preparing students for future challenges in technology and engineering.

Appendix B

Interviews with CLAI Members

The Center for Language Generation and AI (CLAI) at Aarhus University is dedicated to researching and developing foundational language models. CLAI explores the implications of these models for understanding language and cognition, and addresses the ethical issues inherent in language generation and AI. Additionally, the center investigates the creative and pedagogical potential of these technologies. CLAI is a cross-disciplinary research program that integrates linguistics, humanities computing, cognitive science, media studies, and aesthetics.

As preparation for this bachelor's thesis, interviews were conducted with members of CLAI to gain insights into the current research and application of LLMs within the university. Furtherly, inspiration was sought for the development of a chatbot, specifally the issues and challenges that arise when implementing such a system in an academic setting. Interviews were conducted with the following members of CLAI:

Customized RAG Chatbot at Aarhus University

Associate professor Carsten Bergenholtz from the Department of Management implemented, in collaboration with associate professor Oana Vuculescu, a RAG chatbot for his Philosophy of Science course at Aarhus University. For inspiration and guidance on the development of a RAG chatbot an informal interview was conducted with professor Bergenholtz. His involvement provided crucial insights that shaped the direction of the implementation of the chatbot related to this thesis.

Interview Summary

During our discussions, concerns were raised about the use of chatbots like ChatGPT-4 in educational environments. While these systems can produce impressive outputs, they also pose risks due to potential inaccuracies and a lack of course-specific knowledge. However, solutions exist to mitigate these challenges. Professor Bergenholtz shared his experience with implementing a customized RAG chatbot for his Philosophy of Science course, which had an enrollment of 550 students.

Chatbot Implementation

The customized chatbot was used approximately 20,000 times by the students, indicating strong engagement and utility. Professor Bergenholtz uploaded about 250 pages of course-relevant documents, from text to subtitles from his online lectures, to create a knowledge base for the chatbot. This setup, based on AU's Microsoft Azure platform, ensured that the chatbot was:

- GDPR compliant, adhering to data protection regulations.
- Based on the ChatGPT-4 model, ensuring advanced conversational capabilities.
- Freely accessible to all students, removing financial barriers.
- Equipped with an appealing user interface, enhancing user experience.
- Integrated within the existing university systems, ensuring seamless access.

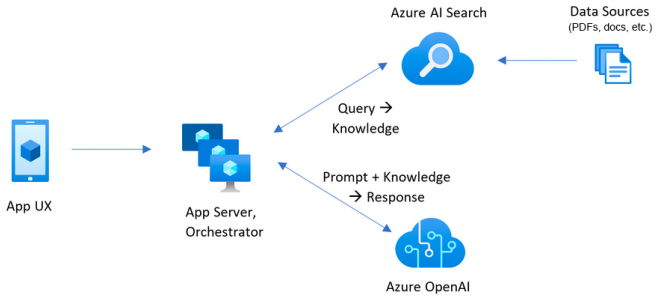


Figure B.1: RAG patterns that include Azure AI Search from which Bergenholtz' chatbot was inspired from - taken from Steen and Wahin, 2024

The RAG chatbot was specifically designed to respond only to queries related to the course content. Questions outside the course scope received a 'cannot answer this' response to maintain the focus and academic integrity of the tool. Additionally, the chatbot provided a link to the source of its answers, enhancing transparency and trust.

Chatbot Evaluation and Costs

The quality of the chatbot's responses was satisfactory, with about 85% of interactions leading to useful answers, and only 2-4.5% of responses being flawed. Notably, the flawed responses were quickly identified by users through follow-up questions. Despite its imperfections, the chatbot was considered a significant improvement over traditional search methods or regular chatbots used by students. The total cost of the chatbot was approximately 600€, with actual running costs around 400€ for 20,000 interactions, showcasing its cost-effectiveness.

Student Feedback and Future Prospects

The chatbot was well-received based on survey responses, with students appreciating its ability to clarify complex concepts, compare texts, and summarize content. This tool proved particularly useful for large classes and when copyright for the necessary materials was held by the course instructor. Plans are in place to continue and enhance this service in future courses, focusing on guiding students to ask more effective questions.

Conclusion

The implementation of the RAG chatbot at Aarhus University exemplifies the practical application of LLMs in enhancing educational experiences. The project set a precedent for future educational tools that leverage AI to support learning and inquiry. This initiative highlights the synergy between innovative technology and traditional educational practices, paving the way for more dynamic and interactive learning environments.

Introduction to Various Tools and Platforms for Development and Research of LLMs

Research assistant Rasmus Hansen from the Centre for Educational Development at Aarhus University specialises in research on students and teachers' practice with learning technology. His current focus is specifically on Generative AI, writing, and feedback. The interview with Rasmus provided valuable insights into some tools and platforms that can be used for developing and researching LLMs.

Interview Summary

After presenting the goal of the thesis and the planned implementation of the chatbot, Rasmus provided an overview of the tools and platforms that could be beneficial for the project. The following tools/platforms were recommended:

- **Hugging Face:** A platform that provides access to pre-trained models, datasets, and training pipelines for NLP tasks.
- **GPT4ALL:** A free-to-use, locally running, privacy-aware chatbot where no GPU or internet is required. Furtherly, a guide on how use RAG in GPT4ALL was provided.[1]
- **LMStudio:** A platform that allows users to create, train, and deploy custom language models locally.

As the project progressed only the Hugging Face platform was used for this thesis. The platform provided access to the pre-trained BART model, and allowed for fine-tuning on custom datasets. The introduction to Hugging Face was instrumental in easy access to the BART model and the implementation of the case study: Low Rank Approximation.

[1] `https://docs.gpt4all.io/gpt4all_chat.html`

Appendix C

Architecture of BART

One can inspect the model architecture by using the `transformers` library by Hugging Face.

BART-Base

```
from transformers import AutoModelForSeq2SeqLM
model_checkpoint = "facebook/bart-base"
model = AutoModelForSeq2SeqLM.from_pretrained(model_checkpoint)
```

Listing 15: Loading pre-trained BART-base model from the Hugging Face model repository

This code snippet fetches the configuration, tokenizer, and weights of the BART model from the Hugging Face model repository. The model can be inspected by printing the model object, which will output the model architecture as shown in Listing 16.

```
BartForConditionalGeneration(
  (model): BartModel(
    (shared): Embedding(50265, 768, padding_idx=1)
    (encoder): BartEncoder(
      (embed_tokens): BartScaledWordEmbedding(50265, 768, padding_idx=1)
      (embed_positions): BartLearnedPositionalEmbedding(1026, 768)
      (layers): ModuleList(
        (0-5): 6 x BartEncoderLayer(
          (self_attn): BartSdpaAttention(
            (k_proj): Linear(in_features=768, out_features=768, bias=True)
            (v_proj): Linear(in_features=768, out_features=768, bias=True)
            (q_proj): Linear(in_features=768, out_features=768, bias=True)
            (out_proj): Linear(in_features=768, out_features=768, bias=True)
          )
          (self_attn_layer_norm): LayerNorm((768,), eps=1e-05, elementwise_affine=True)
          (activation_fn): GELUActivation()
          (fc1): Linear(in_features=768, out_features=3072, bias=True)
          (fc2): Linear(in_features=3072, out_features=768, bias=True)
          (final_layer_norm): LayerNorm((768,), eps=1e-05, elementwise_affine=True)
        )
      )
      (layernorm_embedding): LayerNorm((768,), eps=1e-05, elementwise_affine=True)
    )
    (decoder): BartDecoder(
      (embed_tokens): BartScaledWordEmbedding(50265, 768, padding_idx=1)
      (embed_positions): BartLearnedPositionalEmbedding(1026, 768)
      (layers): ModuleList(
        (0-5): 6 x BartDecoderLayer(
          (self_attn): BartSdpaAttention(
            (k_proj): Linear(in_features=768, out_features=768, bias=True)
            (v_proj): Linear(in_features=768, out_features=768, bias=True)
            (q_proj): Linear(in_features=768, out_features=768, bias=True)
            (out_proj): Linear(in_features=768, out_features=768, bias=True)
          )
          (activation_fn): GELUActivation()
          (self_attn_layer_norm): LayerNorm((768,), eps=1e-05, elementwise_affine=True)
          (encoder_attn): BartSdpaAttention(
            (k_proj): Linear(in_features=768, out_features=768, bias=True)
            (v_proj): Linear(in_features=768, out_features=768, bias=True)
            (q_proj): Linear(in_features=768, out_features=768, bias=True)
            (out_proj): Linear(in_features=768, out_features=768, bias=True)
          )
          (encoder_attn_layer_norm): LayerNorm((768,), eps=1e-05, elementwise_affine=True)
          (fc1): Linear(in_features=768, out_features=3072, bias=True)
          (fc2): Linear(in_features=3072, out_features=768, bias=True)
          (final_layer_norm): LayerNorm((768,), eps=1e-05, elementwise_affine=True)
        )
      )
      (layernorm_embedding): LayerNorm((768,), eps=1e-05, elementwise_affine=True)
    )
  )
  (lm_head): Linear(in_features=768, out_features=50265, bias=False)
)
```

Listing 16: BART-base model architecture

The model has a total of 139 million parameters.

BART-Large

Similarly, for the BART-large model:

```
from transformers import AutoModelForSeq2SeqLM
model_checkpoint = "facebook/bart-large"
model = AutoModelForSeq2SeqLM.from_pretrained(model_checkpoint)
```

Listing 17: Loading pre-trained BART-large model from the Hugging Face model repository

The model architecture is shown in Listing 18.

```
BartForConditionalGeneration(
  (model): BartModel(
    (shared): Embedding(50265, 1024, padding_idx=1)
    (encoder): BartEncoder(
      (embed_tokens): BartScaledWordEmbedding(50265, 1024, padding_idx=1)
      (embed_positions): BartLearnedPositionalEmbedding(1026, 1024)
      (layers): ModuleList(
        (0-11): 12 x BartEncoderLayer(
          (self_attn): BartSdpaAttention(
            (k_proj): Linear(in_features=1024, out_features=1024, bias=True)
            (v_proj): Linear(in_features=1024, out_features=1024, bias=True)
            (q_proj): Linear(in_features=1024, out_features=1024, bias=True)
            (out_proj): Linear(in_features=1024, out_features=1024, bias=True)
          )
          (self_attn_layer_norm): LayerNorm((1024,), eps=1e-05, elementwise_affine=True)
          (activation_fn): GELUActivation()
          (fc1): Linear(in_features=1024, out_features=4096, bias=True)
          (fc2): Linear(in_features=4096, out_features=1024, bias=True)
          (final_layer_norm): LayerNorm((1024,), eps=1e-05, elementwise_affine=True)
        )
      )
      (layernorm_embedding): LayerNorm((1024,), eps=1e-05, elementwise_affine=True)
    )
    (decoder): BartDecoder(
      (embed_tokens): BartScaledWordEmbedding(50265, 1024, padding_idx=1)
      (embed_positions): BartLearnedPositionalEmbedding(1026, 1024)
      (layers): ModuleList(
        (0-11): 12 x BartDecoderLayer(
          (self_attn): BartSdpaAttention(
            (k_proj): Linear(in_features=1024, out_features=1024, bias=True)
            (v_proj): Linear(in_features=1024, out_features=1024, bias=True)
            (q_proj): Linear(in_features=1024, out_features=1024, bias=True)
            (out_proj): Linear(in_features=1024, out_features=1024, bias=True)
          )
          (activation_fn): GELUActivation()
          (self_attn_layer_norm): LayerNorm((1024,), eps=1e-05, elementwise_affine=True)
          (encoder_attn): BartSdpaAttention(
            (k_proj): Linear(in_features=1024, out_features=1024, bias=True)
            (v_proj): Linear(in_features=1024, out_features=1024, bias=True)
            (q_proj): Linear(in_features=1024, out_features=1024, bias=True)
            (out_proj): Linear(in_features=1024, out_features=1024, bias=True)
          )
          (encoder_attn_layer_norm): LayerNorm((1024,), eps=1e-05, elementwise_affine=True)
          (fc1): Linear(in_features=1024, out_features=4096, bias=True)
          (fc2): Linear(in_features=4096, out_features=1024, bias=True)
          (final_layer_norm): LayerNorm((1024,), eps=1e-05, elementwise_affine=True)
        )
      )
      (layernorm_embedding): LayerNorm((1024,), eps=1e-05, elementwise_affine=True)
    )
  )
  (lm_head): Linear(in_features=1024, out_features=50265, bias=False)
)
```

Listing 18: BART-large model architecture

The model has a total of 406 million parameters.