



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 of 2024.

All source files associated with this thesis are found at: <https://github.com/asgersong/BSc>

Asger Poulsen, May 30, 2024

Abstract

Acknowledgements

Contents

1	Introduction	9
2	Theoretical Foundations	11
2.1	Linear Algebra in Deep Learning	11
2.1.1	Vectors, Matrices, and Tensors	11
2.1.2	Matrix Operations	11
	Flops	12
2.1.3	Singular Value Decomposition	12
2.1.4	Low-Rank Approximation	12
	Concept of Low-Rank Approximation	12
	Reduction in Storage	13
	Reduction in Computation for Matrix-Vector Multiplications .	13
	Reduction in Computation for Matrix-Matrix Multiplications .	14
2.1.5	Neural Networks in Deep Learning	14
	Architecture of Neural Networks	15
	Learning Process	15
	Optimization and Regularization	15
2.2	Understanding LLMs	15
2.2.1	What are LLMs?	15
2.2.2	Architecture of LLMs	16
	Encoder	16
	Decoder	17
	Attention	17
	Multi-head Attention	19
	Why Transformers?	20
2.2.3	Training and Fine-Tuning	20
3	Literature Review	23
3.1	Overview of Large Language Models	23
3.2	Previous Studies on LLM Compression and Optimization	23

3.2.1	Low-Memory Optimization	23
3.2.2	Optimization by Prompting	24
3.3	Evaluating Summarization with ROUGE	25
4	Literature Review	27
4.1	Overview of Large Language Models	27
4.2	Previous Studies on LLM Compression	27
4.3	Introduction to BART	28
4.3.1	Architecture and Training	28
4.3.2	Efficacy and Applications	29
4.4	Evaluating Summarization with ROUGE	29
5	Methodology	31
5.1	LLM Optimization	31
5.1.1	Optimization Techniques for Large Language Models	31
Model Pruning	31
Parameter Sharing	31
Focus on Low-Rank Approximation	32
5.2	Case Study: Low-Rank Approximation	32
5.2.1	Implementation Steps	32
5.2.2	Evaluation Metrics	33
5.2.3	Appropriate Rank Selection	34
6	Implementation	37
6.1	Case Study	37
6.1.1	Importing the Model	37
6.1.2	Dataset Preprocessing	37
Importing the Dataset	37
Setting Maximum Lengths	38
Preprocessing Function	38
Applying the Preprocessing Function	38
6.1.3	Finetuning the Model	39
6.1.4	Custom Layer Implementation	39
Traversing the Model and Applying Low-Rank Approximation	40
7	Evaluation and Results	43
7.1	Case Study	43
7.1.1	Results and Analysis	43
7.1.2	Conclusion	44
8	Discussion	45
8.1	Interpretation of Results	45
8.2	Theoretical and Practical Implications	45
8.3	Limitations and Challenges	45
9	Conclusion and Future Work	47
9.1	Summary of Key Findings	47
9.2	Contributions to the Field	47
9.3	Recommendations for Future Research	47

Bibliography	49
A Educational Synergy in Teaching and Research	51
B Poster	55
C Chatbot	57
D Architecture of BART	59

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.

Chapter 2

Theoretical Foundations

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:

$$\mathbf{y} = A\mathbf{x} + \mathbf{b} \quad (2.1)$$

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:

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

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.

Flops

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 $2mnp$.

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:

$$A = U\Sigma V^T \quad (2.3)$$

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 a powerful technique in linear algebra that provides 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 . 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.

¹While 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.

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.
- 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

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

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 requires $O(mn)$ operations.

- **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: $O(kn)$ operations.
 - Σ_k multiplication: $O(k^2)$ operations.
 - U_k multiplication: $O(mk)$ operations.
- **Total:** $O(kn) + O(k^2) + O(mk) = O(k(k + m + n))$ operations.

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

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

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 $O(mnp)$ operations.
- **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: $O(knp)$ operations.
 - Σ_k multiplication: $O(k^2p)$ operations.
 - U_k multiplication: $O(mkp)$ operations.
- **Total:** $O(knp) + O(k^2p) + O(mkp) = O(k(k + m + n)p)$ operations.

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 , these operations can be performed much more efficiently compared to 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:

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

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:

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

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.

2.2 Understanding LLMs

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

2.2.1 What are 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.

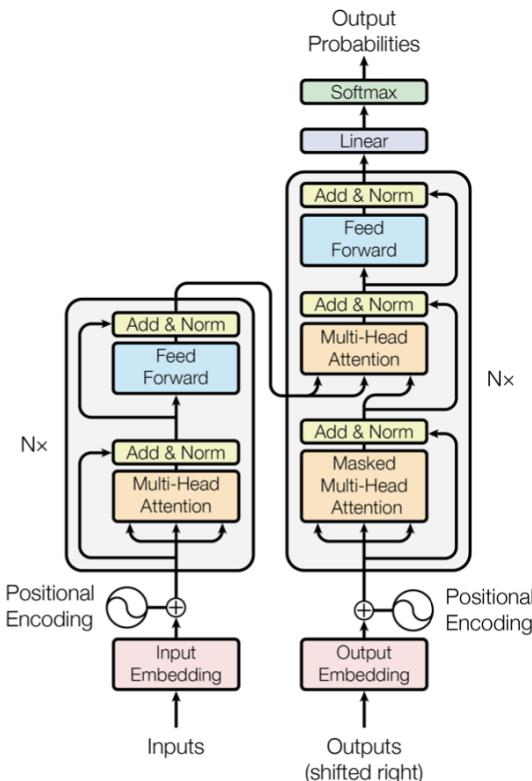


Figure 2.1: Transformer model architecture - Taken from “Attention Is All You Need”

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. Furthermore, 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:

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

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 \vec{E}_i , we compute query (\vec{Q}_i), key (\vec{K}_i), and value (\vec{V}_i) vectors by multiplying the token's embedding with learned matrices W^Q , W^K , and W^V :

$$\vec{Q}_i = W^Q \vec{E}_i, \quad \vec{K}_i = W^K \vec{E}_i, \quad \vec{V}_i = W^V \vec{E}_i$$

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.

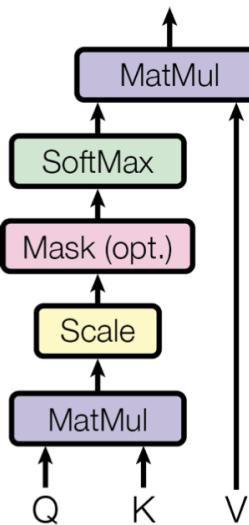


Figure 2.2: Illustration of the Attention Mechanism - Taken from "Attention Is All You Need"

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.

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:

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

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

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:

- Linear Projections:** Project the input embeddings into h different subspaces using learned weight matrices to create multiple sets of queries, keys, and values:

$$Q_i = QW_i^Q, \quad K_i = KW_i^K, \quad V_i = VW_i^V$$

- Parallel Attention Heads:** Apply the attention mechanism to each set of projections in parallel, producing multiple attention outputs:

$$\text{head}_i = \text{Attention}(Q_i, K_i, V_i)$$

- 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 :

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

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.

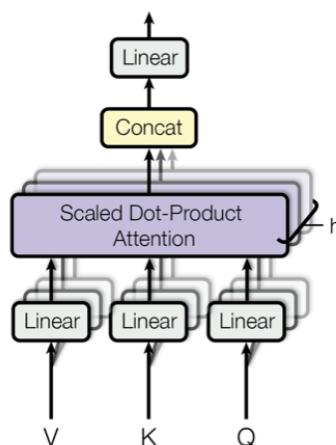


Figure 2.3: Illustration of Multi-head Attention - Taken from "Attention Is All You Need"

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 natural language processing tasks.

Why Transformers?

The Transformer model has several advantages over traditional RNNs and 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:

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

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:

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

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

3.1 Overview of Large Language Models

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 models like Google's BERT (Bidirectional Encoder Representations from Transformers) and OpenAI's GPT (Generative Pre-trained Transformer) series that marked a significant leap in the capabilities of language models. Each iteration of these models has brought about 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.

3.2 Previous Studies on LLM Compression and Optimization

Over the past few years, the increasing demand for Large Language Models (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. Key areas of investigation have included model compression, prompt engineering, and other innovative approaches, all geared towards optimizing resource utilization and facilitating wider accessibility and deployment of LLMs in real-world scenarios. This section aims to shed light on seminal contributions to the optimization of LLMs, detailing the inventive methodologies adopted and the significant advancements realized in making these models more efficient and adaptable to the ever-evolving demands of technology and society.

3.2.1 Low-Memory Optimization

In addressing the optimization of LLMs with constrained resources, Lv et al., 2023 presents a pioneering approach named **LOw-Memory Optimization (LOMO)**. This method innovates by fusing gradient computation and parameter update into

a single operation, significantly reducing the memory requirements for full model fine-tuning. Notably, LOMO enables the comprehensive fine-tuning of models with up to 65 billion parameters on hardware as accessible as a single setup equipped with 8 RTX 3090 GPUs.

A critical innovation within LOMO is the application of Stochastic Gradient Descent (SGD) as the optimizer, distinguished by its lack of reliance on intermediate state storage, thereby contributing to reduced memory consumption. Specifically, LOMO simplifies the traditional two-step process of gradient descent ($\text{grad} = \frac{\partial \mathcal{L}}{\partial p}$, $p = p - lr * \text{grad}$) into a single-step process ($p = p - lr * \frac{\partial \mathcal{L}}{\partial p}$). This optimization not only minimizes memory usage but also maintains, or in some cases, enhances the fine-tuning efficacy for downstream tasks.

The empirical evaluation of LOMO, particularly on the SuperGLUE benchmark, showcases its effectiveness in lowering memory requirements while preserving or improving model performance on various NLP tasks. The advancements brought forth by LOMO significantly democratize the process of fine-tuning LLMs, reducing the computational barrier and facilitating broader research and application possibilities in the field of Natural Language Processing.

3.2.2 Optimization by Prompting

Optimization by PROmpting (OPRO), as introduced by Yang et al., 2023, represents a novel methodology for leveraging Large Language Models (LLMs) as optimizers. The core concept involves using the generative capabilities of LLMs to iteratively generate candidate solutions to optimization problems described in natural language. This process is guided by a meta-prompt, which provides the LLM with a concise description of the optimization task, historical performance data of past solutions, and a directive to generate new solutions aimed at optimizing the given objective function.

The OPRO framework's approach to generating solutions is iterative and dynamic, enabling a detailed exploration and exploitation of the solution space. Each round of solution generation takes into account the performance trajectory of previous solutions, allowing the LLM to refine its approach continuously and hone in on more promising solution regions. This method allows for a sophisticated balance between exploring new solution avenues and exploiting known high-potential areas, a balance that is crucial for the success of any optimization process.

Despite its innovative approach and promising results across various optimization tasks, the study acknowledges several limitations of the OPRO framework. The effectiveness of the optimization process is highly dependent on the quality and detail of the meta-prompt and the relevance and robustness of the initial solutions provided. Moreover, the computational demands of the iterative solution generation process, particularly for complex problems requiring numerous iterations for convergence, present significant challenges. These factors underscore the need for further research and development to refine the OPRO framework, focusing on improving prompt design, solution evaluation mechanisms, and computational efficiency to broaden its applicability and effectiveness in solving a wide range of optimization problems.

3.3 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, 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 Large Language Models, 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

Literature Review

4.1 Overview of Large Language Models

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 models like Google's BERT (Bidirectional Encoder Representations from Transformers) and OpenAI's GPT (Generative Pre-trained Transformer) series that marked a significant leap in the capabilities of language models. Each iteration of these models has brought about 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.

4.2 Previous Studies on LLM Compression

Over the past few years, the increasing demand for Large Language Models (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.

Low Rank Approximation

In the realm of large language models (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 LORA (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 performance of LLMs without extensive resource demands.

A 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 effi-

ciently. 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.

4.3 Introduction to BART

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.

4.3.1 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 D). 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.

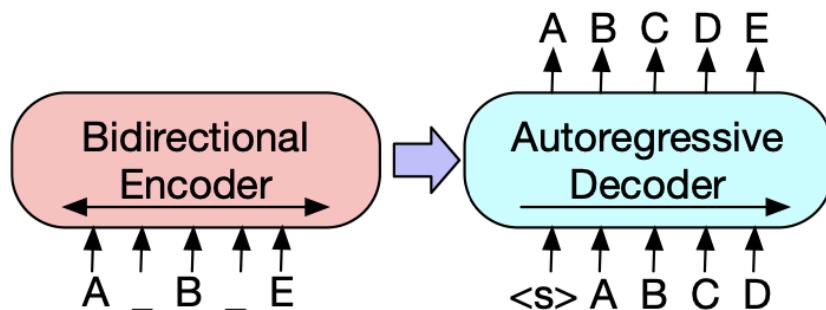


Figure 4.1: BART model’s processing mechanism - Taken from Lewis et al., 2019

Figure 4.1 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.

4.3.2 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.

4.4 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 Large Language Models, 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 5

Methodology

5.1 LLM Optimization

Optimizing Large Language Models (LLMs) for efficiency and performance without compromising their effectiveness is a critical area of research in the field of artificial intelligence. Various techniques have been developed to address this challenge, each employing unique strategies to reduce computational resources, decrease model size, and maintain, if not improve, the model's performance. This section explores the general methodologies applied in the optimization of LLMs, focusing particularly on model compression techniques. Among these, Low-Rank Approximation stands out as a highly effective approach. This technique leverages the mathematical properties of matrices to approximate the original model parameters with fewer components, thereby possibly reducing the model's computations and resource requirements.

5.1.1 Optimization Techniques for Large Language Models

Optimization strategies for Large Language Models (LLMs) are crucial for improving computational efficiency and model efficacy. These strategies are generally divided into two primary types: model pruning and parameter sharing.

Model Pruning

Model pruning aims to reduce the model's complexity and size effectively without substantial loss in performance. It involves systematically eliminating parameters or connections within the model that are least consequential to the output, thereby enhancing operational efficiency and making the model more adaptable for use in environments with limited resources.

Parameter Sharing

Conversely, parameter sharing utilizes the model's existing parameters across various parts of the model or different tasks. This method optimizes the use of the model's capacity, enabling multifunctional performance without an increase in parameter count.

Focus on Low-Rank Approximation

This thesis will specifically focus on model pruning techniques, with a particular emphasis on low-rank approximation. This approach approximates large matrices or tensors with ones of a lower rank, reducing the number of components and computational demands while preserving the model's critical information processing capabilities.

5.2 Case Study: Low-Rank Approximation

To illustrate the application of low-rank approximation in compressing LLMs, we consider Facebook's the BART-Base model (Lewis et al., 2019) as a case study. BART is a transformer-based LLM that has been widely used for various natural language processing tasks such as summarization.

This thesis focuses on applying low-rank approximation specifically to the attention matrices. This choice stems from their pivotal role in the transformer architecture. 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-base model, we aim to explore the possibility in reducing computational complexity and storage requirements while preserving its performance on a summarization task.

5.2.1 Implementation Steps

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

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

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

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

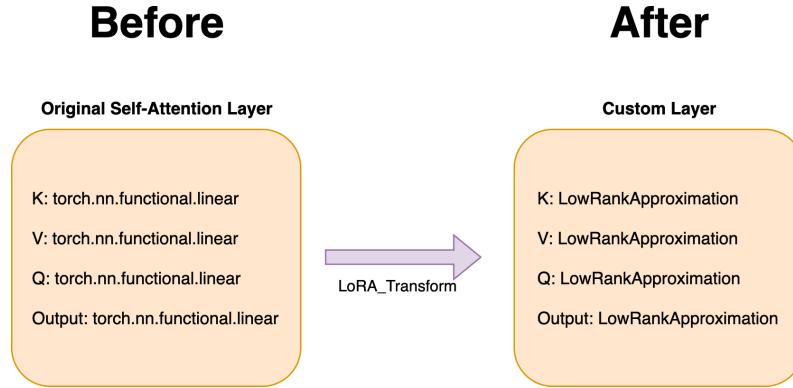


Figure 5.1: Components in attention layers replaced with their custom low-rank approximation

5.2.2 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 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 model in terms of computational resources required?
3. Cosine similarity: How well does the approximated model maintain the semantic properties of the original model?
4. Comparing the summaries generated by the compressed model with the original BART-Base model and the reference summaries using the Samsum dataset (Gliwa et al., 2019).

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

Figure 5.2: SamSum Dataset `test[0]` dialogue and reference summary

5.2.3 Appropriate Rank Selection

In Section 2.1.4, we established the criterion for selecting an appropriate rank k to achieve computational efficiency and storage reduction. 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 we wish to approximate with a low-rank representation.

BART-base Model:

For the BART-base model, the attention matrices have dimensions $m = n = 768$. We know this by inspecting the model's architecture (Appendix D). 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.

To determine the feasibility of low-rank approximation for the BART-Base model, we will evaluate the approximated model at various ranks and observe the impact on the ROUGE scores. This evaluation will help us understand the trade-offs between rank reduction and model performance.

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:

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

Chapter 6

Implementation

6.1 Case Study

6.1.1 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 1: Importing the BART-base model

6.1.2 Dataset Preprocessing

The SamSum dataset was imported and preprocessed to facilitate the training and evaluation of the BART-base model. The dataset consists of dialogues extracted from the Ubuntu Dialog Corpus, along with human-generated summaries. The preprocessing steps included tokenization, truncation, and padding to ensure uniform input sizes for the model.

Importing the Dataset

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

```
from datasets import load_dataset
raw_datasets = load_dataset("samsum")
```

Listing 2: 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 3: 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, we defined a preprocessing function 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 4: 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 5: 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.

6.1.3 Finetuning the Model

The model was then fine-tuned on the SamSum dataset for dialogue summarization tasks.

Epoch	Training Loss	Validation Loss	Rouge1	Rouge2	Rougel	Rougelsum	Gen Len
0	1.81	1.54	47.39	24.45	40.03	43.77	18.21
2	1.48	1.49	47.98	24.7740	40.66	44.21	18.06

Table 6.1: Training and validation results across epochs.

6.1.4 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 Singular Value Decomposition (SVD) from the PyTorch library to decompose and subsequently truncate the weight matrices, preserving only the k most significant components.

```
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 6: 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.

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 7: 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.

Chapter 7

Evaluation and Results

7.1 Case Study

7.1.1 Results and Analysis

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

ROUGE scores: The low-rank approximation technique achieves comparable ROUGE scores until rank $r = 510$, 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:

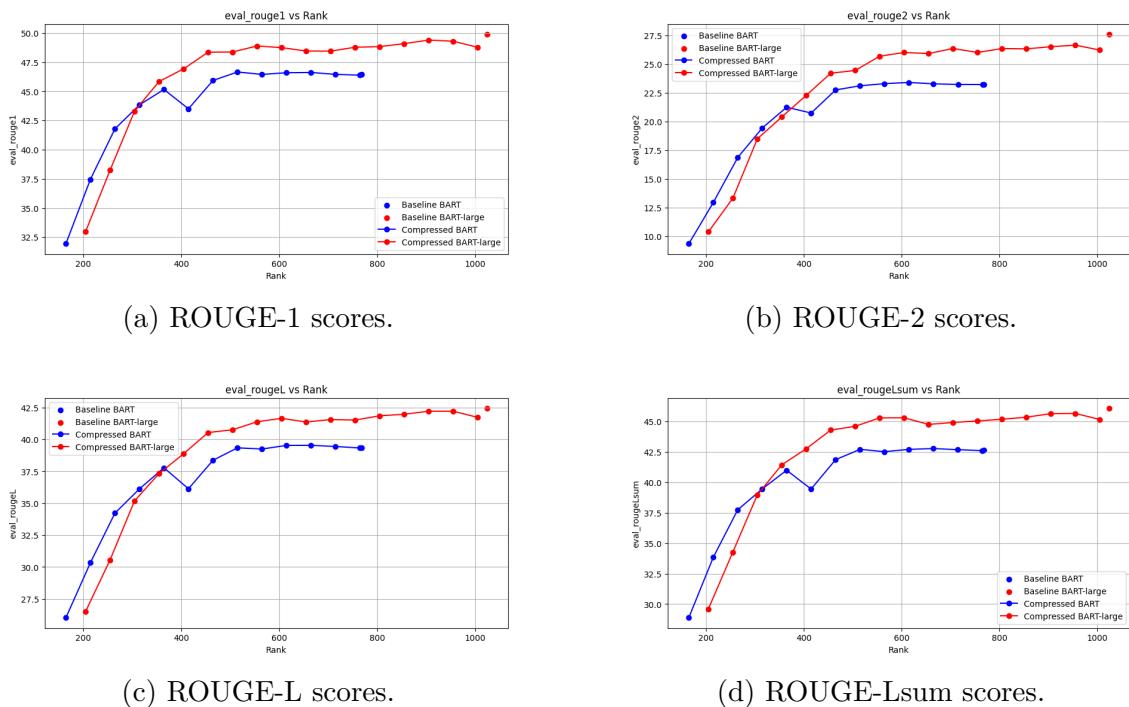


Figure 7.1: ROUGE scores for different ranks.

As depicted in Figure 7.1, the ROUGE-1, ROUGE-2, ROUGE-L, and ROUGE-Lsum scores maintain a high level of performance up to a rank of $r = 510$. However,

it is important to note that for the BART-large model, there is a slight decline in performance even when the model is low-rank approximated at full rank. Beyond the rank of $r = 510$, a noticeable degradation in scores is observed, indicating the limitations of the low-rank approximation in preserving the model’s quality.

Cosine Similarity:

Comparing Summaries:

Computational Complexity and Storage Requirements: Recall from Section 5.2.3 that we needed to low-rank approximate the self-attention matrices with at most rank $r = 318$ and rank $r = 424$ for BART-base and BART-large respectively to achieve a significant reduction in computational complexity and storage requirements. However, the results show that the summarization quality declines significantly before this threshold.

7.1.2 Conclusion

The low-rank approximation of the BART-Base model does not present a viable approach to compressing large language models for summarization tasks. Although the technique can achieve significant reductions in computational complexity and storage requirements, the summarization quality declines significantly before reaching a practically useful rank. These findings highlight the limitations of low-rank approximations for preserving the performance of large language models in summarization tasks.

In summary, while low-rank approximation effectively reduces the computational burden and storage requirements of the BART-Base model, the associated decline in summarization quality limits its practical applicability. This work underscores the need for alternative approaches to optimize and deploy large language models in resource-constrained environments without compromising performance.

Chapter 8

Discussion

8.1 Interpretation of Results

8.2 Theoretical and Practical Implications

8.3 Limitations and Challenges

Chapter 9

Conclusion and Future Work

9.1 Summary of Key Findings

9.2 Contributions to the Field

9.3 Recommendations for Future Research

Bibliography

- Vaswani, Ashish et al. (2017). *Attention Is All You Need*. arXiv: 1706 . 03762 [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].
- 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].
- Zhu, Xunyu et al. (2023). *A Survey on Model Compression for Large Language Models*. arXiv: 2308.07633 [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 Computational 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].

- Lewis, Mike et al. (2019). *BART: Denoising Sequence-to-Sequence Pre-training for Natural Language Generation, Translation, and Comprehension*. arXiv: 1910.13461 [cs.CL].
- Devlin, Jacob et al. (2019). *BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding*. arXiv: 1810.04805 [cs.CL].

Appendix A

Educational Synergy in Teaching and Research

Introduction

This appendix outlines the dual role I embraced during my semester as both a Teaching Assistant (TA) in the course "SW2PLA: Practical Linear Algebra for Software Engineers" and doing research for my Bachelor's thesis. My unique position allowed me to bridge theoretical understanding and practical application, fostering an educational environment where concepts in linear algebra, that are 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, my responsibilities included assisting students with practical exercises, grading assignments, and engaging in discussions about linear algebra during lectures. These activities were aimed at deepening students' understanding of linear algebra's application in modern computational technologies, particularly within the realms of basic AI and elementary machine learning. Simultaneously, my 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. Which gave me 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 it's 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, I also had the opportunity to present the methodologies and intermediate findings of my bachelor's thesis to the class. This presentation served as a practical demonstration of how linear algebra is utilised 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. My presentation alongside the students' highlighted the reciprocal nature of teaching/learning and research.

External Presentation and Recognition

In addition to sharing my findings within the SW2PLA course, I had the opportunity to present the methodologies and intermediate results of my bachelor's thesis at a poster competition held during the Center for Language Generation and

AI (CLAI) workshop. This presentation allowed me to engage with a broader audience of experts and peers in the field, receiving valuable feedback that furthered my research endeavors.

The highlight of this event was the recognition of the work, as the 3rd place in the competition was secured. This achievement not only affirmed the quality and relevance of my 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 I was delivering in the SW2PLA course, enhancing the overall teaching and learning experience.

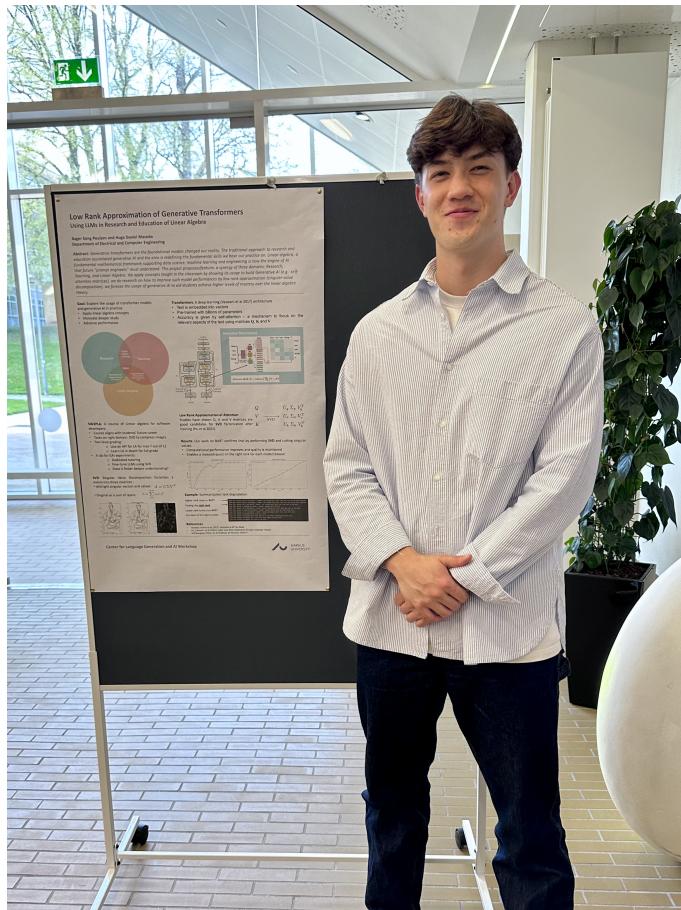


Figure A.1: 3rd place in the CLAI Poster Competition

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 a good practice for me to present research findings to a broader audience, including students and faculty members from various disciplines.

Conclusion

My involvement in the SW2PLA course as a TA, while simultaneously conducting research for my bachelor's thesis, created a rich educational synergy. This experience

not only enhanced my understanding and teaching of linear algebra but also allowed me to directly apply and evaluate theoretical concepts in practical, research-based applications. It underscores the importance of an integrated approach in education, where teaching responsibilities and academic research complement and enrich each other, preparing students, and myself included, for future challenges in technology and engineering.

Appendix B

Poster

The following page contain the poster presented at the CLAI Workshop 2024.

Low Rank Approximation of Generative Transformers Using LLMs in Research and Education of Linear Algebra

Asger Song Poulsen and Hugo Daniel Macedo
Department of Electrical and Computer Engineering

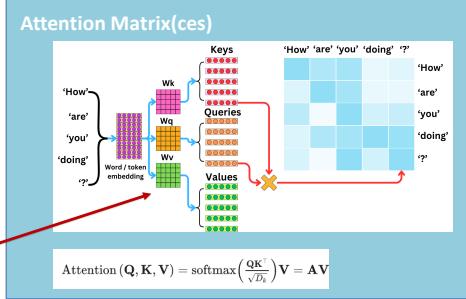
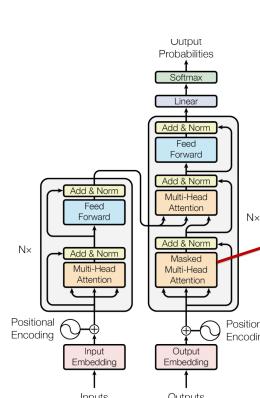
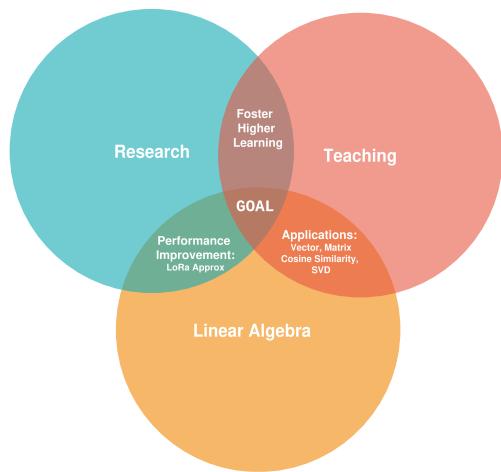
Abstract: Generative transformers are the foundational models changed our reality. The traditional approach to research and education assimilated generative AI and the area is redefining the fundamental skills we base our practice on. Linear algebra, a fundamental mathematical framework supporting data science, machine learning and engineering is now the engine of AI that future "prompt engineers" must understand. This project proposes/features a synergy of three domains: Research, Teaching, and Linear Algebra. We apply concepts taught in the classroom by showing its usage to build Generative AI (e.g.: self-attention matrices), we do research on how to improve such model performances by low rank approximation (singular-value decomposition), we foresee the usage of generative AI to aid students achieve higher levels of mastery over the linear algebra theory.

Goal: Explore the usage of transformer models and generative AI in practice:

- Apply linear algebra concepts
- Motivate deeper study
- Advance performance

Transformers: A deep learning (Vaswani et al 2017) architecture

- Text is embedded into vectors
- Pre-trained with billions of parameters
- Accuracy is given by self-attention – a mechanism to focus on the relevant aspects of the text using matrices \mathbf{Q} , \mathbf{K} and \mathbf{V}

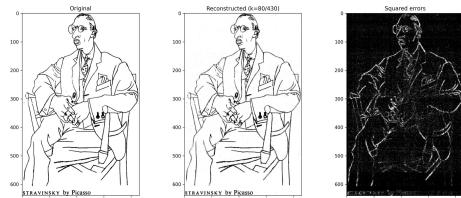


SW2PLA: A course of Linear algebra for software developers:

- Course aligns with students' future career
- Tasks on right domain: SVD to compress images
- Two level grading:
 - Use an API for LA for max 7 out of 12
 - Learn LA in depth for full grade
- A lab for GAI experiments:
 - Dedicated tutoring
 - Fine-tune LLMs using SVD
 - Does it foster deeper understanding?

SVD: Singular Value Decomposition factorizes a matrix into three matrices :

- left/right singular vectors and values: $A = U\Sigma V^T$
- Original as a sum of layers: $A = \sum_{i=1}^{\text{rank}} u_i \sigma_i v_i^T$



$$Q \quad \quad \quad U_q \Sigma_q V_q^T$$

$$V \xrightarrow[\text{SVD}]{} U_v \Sigma_v V_v^T$$

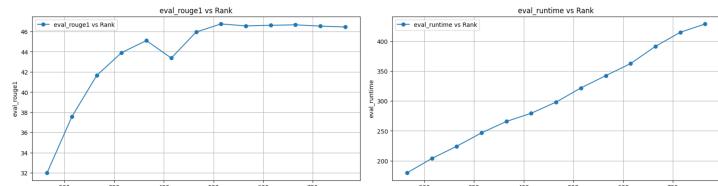
$$K \quad \quad \quad U_k \Sigma_k V_k^T$$

Low Rank Approximation of Attention:

Studies have shown Q, K and V matrices are good candidates for SVD factorization after training (Hu et al 2021).

Results: Our work on BART confirms that by performing SVD and cutting singular values:

- Computational performance improves and quality is maintained
- Enables a research quest on the right rank for each model/dataset



Example: Summarization task degradation:

Higher rank closer to BART

Finding the right rank

Lower rank further from BART
(less layers of the original model)

Rank 765: Hannah doesn't know Betty's number. She texted Larry last time they were at the park together.
Rank 660: Hannah doesn't know Betty's number. She texted Larry last time they were at the park together.
Rank 518: Hannah can't find Betty's number. She texted Larry last time they were at the park.
Rank 410: Hannah is looking for Betty's number. Amanda can't find it.
Rank 370: Amanda can't find Betty's number. Hannah texted Larry last time they were at the park.
Rank 365: Amanda is looking for Betty's number. Hannah doesn't know her boyfriend Larry.
Rank 355: Amanda and Hannah don't know each other.
Rank 315: Amanda and Hannah don't know each other.
Rank 210: Amanda and Amanda don't know what to do with their relationship with Larry.
Rank 190: Amanda and Amanda don't know each other person in the world.
Rank 165: Amanda and Amanda are going to the park.

References

- Vaswani, Ashish et al. (2017). *Attention Is All You Need*
- Hu, Edward J. et al. (2021). *LoRA: Low-Rank Adaptation of Large Language Models*
- Jeff Bussgang (2024). *An AI Professor at Harvard: ChatLTV*

Appendix C

Chatbot

Customized RAG Chatbot at Aarhus University

In the initial phase of the project, I engaged with faculty members at Aarhus University to gain insights into the current research and application of Large Language Models (LLMs) within the university. This collaboration aimed to understand both the potential and the challenges associated with the use of LLMs in academic settings.

Associate Professor Carsten Bergenholz from the Department of Management contributed significantly by sharing his experiences with LLMs. His involvement provided crucial insights that shaped the direction of our project.

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 Bergenholz shared his experience with implementing a customized Retrieval-Augmented Generation (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 Bergenholz 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.

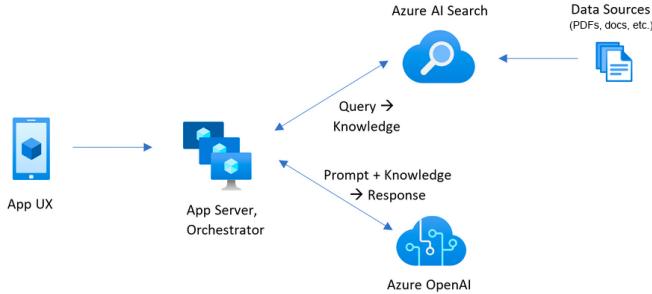


Figure C.1: Figure to put somewhere

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.

Appendix D

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 8: 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 9.

```

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 9: 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 10: Loading pre-trained BART-large model from the Hugging Face model repository

The model architecture is shown in Listing 11.

```

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 11: BART-large model architecture

The model has a total of 406 million parameters.