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, March 30, 2024

Abstract

Acknowledgements

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.

Chapter 2

Literature Review

2.1 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.

2.2 Previous Studies on LLM Compression and Optimization

- A Survey on Model Compression for Large Language Models
- The complete guide to LLM compression

Chapter 3

Theoretical Foundations

3.1 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.

3.1.1 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:

$$\mathbf{y} = A\mathbf{x} + \mathbf{b} \tag{3.1}$$

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.

3.1.2 Eigenvalues and Eigenvectors

Eigenvalues and eigenvectors play a critical role in understanding the behavior of linear transformations characteristic of deep learning models. For matrix A, if there exists a vector $\mathbf{v} \neq 0$ such that:

$$A\mathbf{v} = \lambda\mathbf{v} \tag{3.2}$$

then λ is called an eigenvalue of A, and $\mathbf{v}$ is the corresponding eigenvector. This concept is pivotal in techniques like Principal Component Analysis (PCA), which reduces dimensionality for data visualization and preprocessing.

3.1.3 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:

$$A = U \Sigma V^T \tag{3.3}$$

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.

3.1.4 Principal Component Analysis (PCA)

Principal Component Analysis (PCA) is a statistical procedure that uses an orthogonal transformation to convert a set of observations of possibly correlated variables into a set of values of linearly uncorrelated variables called principal components. This transformation is defined in such a way that the first principal component has the largest possible variance (that is, accounts for as much of the variability in the data as possible), and each succeeding component, in turn, has the highest variance possible under the constraint that it is orthogonal to the preceding components. The resulting vectors (principal components) are an uncorrelated orthogonal basis set. PCA is sensitive to the relative scaling of the original variables.

Given a data matrix $X \in \mathbb{R}^{n \times m}$, where n is the number of observations and m is the number of variables, PCA seeks to identify the matrix W that maps X to a new set of variables Y, the principal components, such that the variance of Y is maximized. Mathematically, PCA solves for the eigenvalue decomposition of the covariance matrix $X^T X$ or the singular value decomposition (SVD) of X itself.

The covariance matrix C of X is given by:

$$C = \frac{1}{n-1} X^T X \tag{3.4}$$

where X^T is the transpose of X. The eigenvalue decomposition of C is then performed to find the principal components:

$$C = W \Lambda W^T \tag{3.5}$$

where W is the matrix of eigenvectors (principal components) and Λ is the diagonal matrix of eigenvalues (variance explained by each principal component). The columns of W are sorted by decreasing eigenvalues to ensure the principal components are ordered by the amount of variance they explain.

The principal components Y can be obtained by projecting the original data X onto the space defined by the principal components:

$$Y = XW \tag{3.6}$$

PCA is widely used in data analysis and for making predictive models. It is commonly used as a tool in exploratory data analysis and for making predictive models. It is also employed to visualize genetic distance and relatedness between populations.

3.2 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.

3.2.1 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.

3.2.2 Architecture of LLMs

The architecture of most LLMs is based on the Transformer model, introduced by Vaswani et al., 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, on word at a time.

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

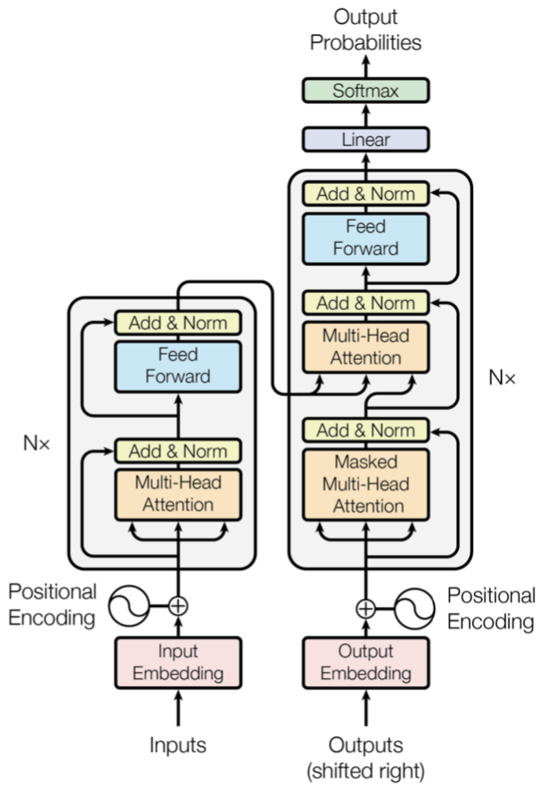

Figure 3.1: Transformer model architecture - Taken from “Attention Is All You Need“

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 crucial component of the Transformer model, allowing it to weigh the importance of different words in the input sequence when generating the output. The attention mechanism is defined as follows:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V \tag{3.7}$$

where Q, K, and V represent the query, key, and value matrices, respectively, and d_k is the dimension of the key vector. This self-attention mechanism allows the model to focus on relevant parts of the input sequence when performing a task, enabling superior handling of long-range dependencies.

Multi-head attention

Multi-head attention is an extension of the attention mechanism that allows the model to focus on different parts of the input sequence simultaneously. It achieves this by, instead of performing a singel attention function with d_{model}-dimensional keys, values and queries, linearly projecting the query, key, and value matrices h times into multiple subspaces with dimensionality d_k, d_k, and d_v, respectively, before applying the attention function. The output of each of these h attention heads is then concatenated and linearly projected to produce the final output. This mechanism allows the model to capture different aspects of the input sequence in parallel, enhancing its ability to learn complex patterns in the data.

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \ldots, \text{head}_h)W^O \tag{3.8}$$

$$\text{where head}_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V) \tag{3.9}$$

Where $W_i^Q \in \mathbb{R}^{d_{\text{model}} \times d_k}$, $W_i^K \in \mathbb{R}^{d_{\text{model}} \times d_k}$, $W_i^V \in \mathbb{R}^{d_{\text{model}} \times d_v}$, and $W^O \in \mathbb{R}^{hd_v \times d_{\text{model}}}$ are learned linear projections, and d_{model} is the dimension of the model.

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.

3.2.3 Training and Fine-Tuning

The training of LLMs involves two main phases: pre-training and fine-tuning. During pre-training, the model is exposed to a large corpus of text and learns to predict missing words or sentences, acquiring a broad understanding of language. The fine-tuning phase adjusts the pre-trained model to specific tasks by training on smaller, task-specific datasets. This two-phase approach allows LLMs to achieve remarkable performance across a wide range of NLP tasks with minimal task-specific model modifications.

3.2.4 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:

$$\mathcal{L}_{\text{MLM}} = -\sum_{i \in \mathcal{M}} \log p(x_i | x_{\setminus \mathcal{M}}) \tag{3.10}$$

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:

$$\mathcal{L}_{\text{fine-tune}} = -\sum_{(x,y) \in \mathcal{D}_{\text{task}}} \log p(y | x; \theta_{\text{pre-train}} + \Delta\theta) \tag{3.11}$$

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. s

Chapter 4

Methodology

4.1 Educational Synergy Development

4.2 Software Engineering Application

4.3 Evaluation Method

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 Adaptation (LoRA) stands out as a significant innovation, offering a balance between model efficiency and task performance.

4.3.1 General Optimizations for LLMs

Optimization techniques for LLMs can be broadly categorized into two: model pruning and parameter sharing. Model pruning involves systematically removing parameters or connections within the model that contribute the least to its output, thereby reducing its size and complexity. Parameter sharing, on the other hand, reuses the model's parameters across different parts of the model or across different tasks, efficiently leveraging the model's capacity.

4.3.2 Low-Rank Adaptation (LoRA)

Low-Rank Adaptation (LoRA) introduces an efficient technique for adapting large pre-trained models like GPT-3 to specific tasks without the need for extensive retraining of all model parameters. This approach leverages the observation that despite the high parameter count in large neural models, their effective operational space often exhibits a significantly lower intrinsic dimensionality.

Theoretical Foundation At the core of LoRA is the adaptation of weight matrices $W \in \mathbb{R}^{d\times d}$ through the introduction of low-rank matrices $A \in \mathbb{R}^{d\times r}$ and

$B \in \mathbb{R}^{r\times d}$, where $r \ll d$. This results in the adapted weight matrix W' being represented as:

$$W' = W + BA \tag{4.1}$$

Here, r denotes the rank and serves as a crucial parameter that balances the efficiency and expressiveness of the adaptation. This formulation ensures that the pre-trained weights (W) remain unchanged, preserving the foundational knowledge acquired during pre-training, while A and B encapsulate the task-specific adjustments.

Advantages LoRA's methodology brings forth several advantages:

- **Parameter Efficiency:** By optimizing the low-rank matrices A and B, LoRA significantly reduces the number of trainable parameters, leading to efficient storage and faster adaptation processes.
- **Preservation of Pre-trained Knowledge:** The approach ensures that the valuable knowledge captured in the pre-trained model is retained, making it particularly beneficial for adapting computationally expensive models like GPT-3.
- **Flexibility and Scalability:** LoRA's adaptive process is highly scalable and flexible, making it possible to adapt large models to various tasks with minimal computational overhead.

Practical Implementation Implementing LoRA involves selecting the transformer layers most relevant to the target task and applying the low-rank adaptations. The process requires initializing A and B, choosing an appropriate rank r, and training these matrices while keeping the rest of the model parameters fixed. This strategy allows for the efficient adaptation of large-scale models to specific tasks, significantly reducing the need for computational resources compared to traditional full model retraining.

4.3.3 Metrics for Evaluation

Evaluating the effectiveness of optimization techniques like LoRA involves several key metrics:

Model Size Reduction: A primary metric is the reduction in the total number of parameters, indicating the efficiency of the compression technique.

Inference Speed: The impact on the model's inference latency is critical, especially for applications requiring real-time responses.

Performance Retention: The maintenance of task-specific performance, measured through metrics such as accuracy, F1 score, or ROUGE scores for language tasks, is essential to ensure the utility of the compressed model.

Computational Efficiency: The reduction in computational resources required for training and inference reflects the practical benefits of the optimization technique.

These metrics provide a comprehensive framework for assessing the trade-offs involved in LLM optimization, guiding the development and application of techniques like LoRA in enhancing the practical utility of state-of-the-art language models.

Chapter 5

Implementation

5.1 Curriculum Component Implementation

5.2 Chatbot Development

5.2.1 Tools and Libraries Used

5.2.2 Integration of LLM

Chapter 6

Evaluation and Results

6.1 Curriculum Effectiveness

6.2 Chatbot Performance and Usefulness

6.3 Compression and Optimization of LLM

6.3.1 Methodology Applied

6.3.2 Results and Analysis

Chapter 7

Discussion

7.1 Interpretation of Results

7.2 Theoretical and Practical Implications

7.3 Limitations and Challenges

Chapter 8

Conclusion and Future Work

8.1 Summary of Key Findings

8.2 Contributions to the Field

8.3 Recommendations for Future Research