Transformers in NLP

Enroll. No. (s) - 9920103216, 9920103205
Name of Student (s) - Ayushi Bawari, Nipun Singh Rathore
Name of supervisor - Ms. Akanksha Mehndiratta

May-2024

Submitted in partial fulfillment of the Degree of

Bachelor of Technology

In

Computer Science Engineering

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING AND INFORMATION TECHNOLOGY

JAYPEE INSTITUTE OF INFORMATION TECHNOLOGY, NOIDA

(I)

TABLE OF CONTENTS

(II)

DECLARATION

I/We hereby declare that this submission is my/our own work and that, to the best of my knowledge and belief, it contains no material previously published or written by another person nor material which has been accepted for the award of any other degree or diploma of the university or other institute of higher learning, except where due acknowledgment has been made in the text.

Place:Jaypee Institute of Information Technology
Date: 20th April 2024

Signature:	Signature:
Name: Nipun Singh Rathore	Name: Ayushi Bawari
Enrollment No: 9920103205	Enrollment No.: 9920103216

(III)

CERTIFICATE

This is to certify that the work titled **"Transformers in NLP"** submitted by **"Nipun Singh Rathore, Ayushi Bawari"** in partial fulfillment for the award of degree of B.Tech of Jaypee Institute of Information Technology, Noida has been carried out under my supervision. This work has not been submitted partially or wholly to any other University or Institute for the award of this or any other degree or diploma.

Signature of Supervisor

Name of Supervisor	Ms. Akanksha Mehndiratta
Designation	Assistant Professor (Grade-II)
Date	20th April 2024

(IV)

ACKNOWLEDGEMENT

I would like to place on record my deep sense of gratitude to Ms. Akanksha Mehndiratta, Assistant Professor (Grade-II), Jaypee Institute of Information Technology, India for her generous guidance, help and useful suggestions.

I express my sincere gratitude to the faculty, Dept. of Computer Science & Engineering, India, for his/her stimulating guidance, continuous encouragement and supervision throughout the course of present work.

I also wish to extend my thanks to other classmates for their insightful comments and constructive suggestions to improve the quality of this project work.

Signature of the Student		Signature of the Student	
Name of Student	Nipun Singh Rathore	Name of student	Ayushi Bawari
Enrollment Number	9920103205	Enrollment Number	9920103216
Date	20th April 2024	Date	20th April 2024

(V)
SUMMARY

The Transformer Project is an ambitious endeavor focused on advancing natural language processing (NLP) capabilities through the development of a comprehensive framework based on Transformer architecture. Our goal is to create a versatile system that leverages state-of-the-art techniques to address a wide range of NLP tasks, including text classification, summarization, question answering, and more. We have embarked on this project with a multifaceted approach, beginning with the implementation of fundamental Transformer components such as attention mechanisms, tokenizers, embeddings, encoder, and decoder. Integrating pre-trained models from Hugging Face allows us to harness the power of existing Transformer architectures and fine-tune them for specific tasks.

Furthermore, we aim to develop a user-friendly web application interface that empowers users to interact with the Transformer model seamlessly. This includes functionalities for inputting text data, selecting NLP tasks, visualizing model outputs, and more. Overall, the Transformer Project represents a significant step forward in advancing NLP capabilities, with the potential to drive innovation and facilitate breakthroughs in various domains such as healthcare, finance, education, and beyond.

Signature of the student		Signature of Supervisor	
Name	Nipun Singh Rathore	Name	Ms.Akanksha Mehndiratta
Date	20th April 2024	Date	20th April 2024

Signature of the student	
Name	Ayushi Bawari
Date	20th April 2024

(VI)

LIST OF FIGURES

(VII)

LIST OF TABLES

(VIII)

LIST OF SYMBOLS & ACRONYMS

	Acronym		Page
1.	BERT	Bidirectional Encoder Representations from Transformers	12
2.	NLP	Natural Language Processing	12
3.	GPT	Generative Pre-Trained Transformers	12
4.	MHSA	Multi Head Self Attention	15
5.	FFN	Feed Forward Neural Network	15
6.	MLM	Masked Language Model	17
7.	NSP	Next Sentence Prediction	21
8.	SQuAD	Stanford Question Answering Dataset	25
9.	BPE	Byte Pair Encoder	30
10.	ReLU	Rectified Linear Unit	38
11.	AdamW	Adam Weight Decay Optimiser	46

CHAPTER 1: INTRODUCTION

1.1 General Introduction:

The project at hand revolves around the implementation and optimization of a Transformer-based language model, with a primary focus on the BERT (Bidirectional Encoder Representations from Transformers) architecture[1]-[5]. Transformers have emerged as a revolutionary neural network architecture, excelling in natural language processing tasks by capturing intricate contextual relationships within sequences. The code snippets provided encompass fundamental components of the Transformer model, including scaled dot-product attention, multi-head self-attention, encoder and decoder blocks, positional embeddings, and various optimization techniques. The aim is to delve into the intricacies of the Transformer architecture, unraveling its mechanisms through hands-on implementation. Beyond mere replication, the project explores optimization strategies such as quantization, pruning, and knowledge distillation, aiming to enhance the model's efficiency and applicability. This endeavor not only facilitates a deeper understanding of Transformer models but also serves as a foundation for experimenting with advanced language models and adapting them to specific use cases.

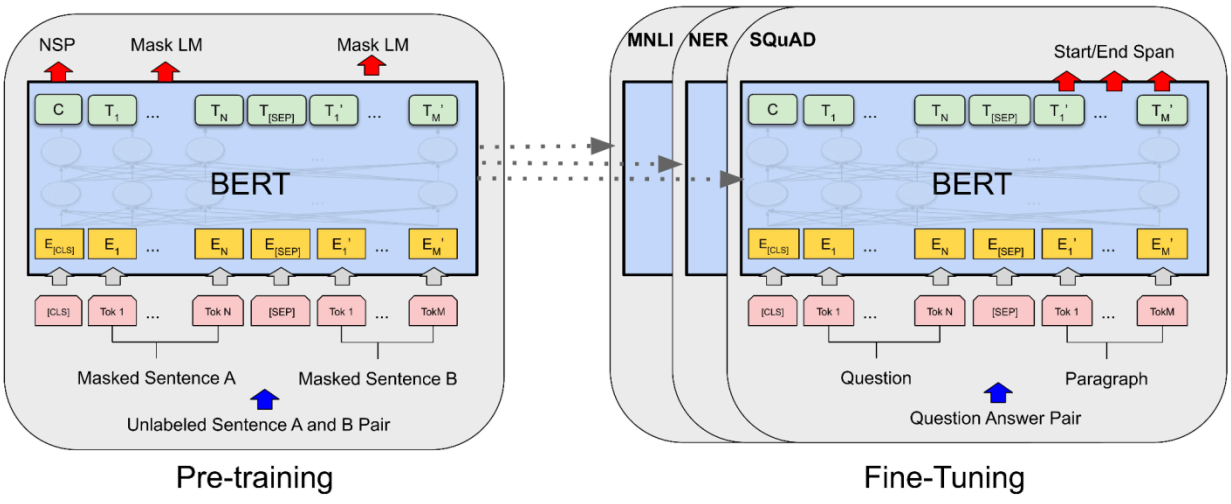

Figure 1.1 : BERT[4]

1.2 Problem Statement:

This project is dedicated to conducting a comprehensive and in-depth study of transformer models in the realm of natural language processing. The primary objective of this project includes unraveling the intricate architectural design, investigating the nuances of pre-training methodologies, exploring fine-tuning strategies, dissecting all constituent layers, and undertaking the ambitious task of pre-training a transformer model from scratch. By understanding the core concepts behind Transformers, such as self-attention[2] mechanisms and positional encoding, we aim to harness their capabilities to achieve superior performance in tasks like text classification, sentiment analysis, language translation, and more. The aim is to gain profound insights into transformers' inner workings and their practical implications for various NLP applications.

1.3 Significance of the Project:

The significance of this project lies in its contributions to natural language processing (NLP) research and practical applications -:

1. **Advancement in NLP:** The project involves training and utilizing transformer-based models for tasks such as question answering, text classification, and sentiment analysis. These models represent the state-of-the-art in NLP and contribute to advancing the field by exploring new architectures and training methodologies.
2. **Development of Custom Models:** By pre-training custom transformer models, the project expands the range of available NLP tools and resources. These custom models can be fine-tuned for specific domains or tasks, providing tailored solutions for various applications.
3. **Practical Applications:** The web application developed in the project demonstrates the practical use of transformer models for question answering tasks. Such applications have real-world implications in fields like customer support, information retrieval, and education, where accurate and efficient question answering systems are highly valuable.
4. **Innovation in Model Architectures:** Through experiments with different model architectures, training techniques, and datasets, the project contributes to innovation in

NLP model design. Insights gained from these experiments can lead to the development of more efficient and effective models in the future.

5. **Educational Value:** The project serves as a learning resource for students and researchers interested in NLP and machine learning. It provides insights into model training, evaluation, and deployment processes, along with practical examples and code implementations.

Model Architectures: Investigate different Transformer architectures, including BERT, GPT, T5, and their variations, to understand their strengths and weaknesses.

Pre-training and Fine-tuning: Explore the process of pre-training Transformer models on large corpora and fine-tuning them for specific downstream tasks relevant to our project goals.

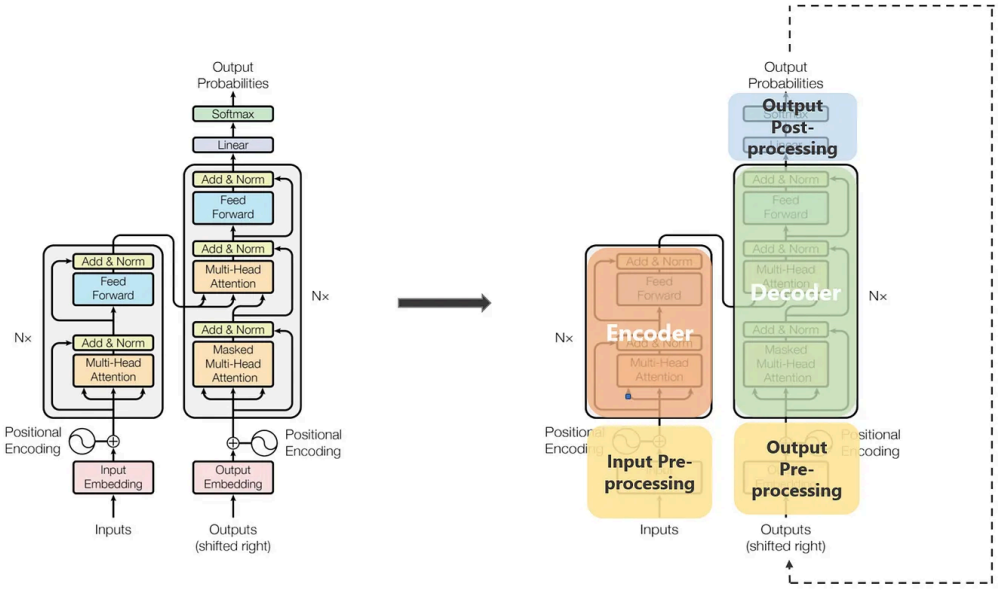

Figure 1.2 : BERT Architecture[4]

Interpretability: Examine techniques for interpreting and visualizing the inner workings of Transformer models to gain insights into how they process and represent language.

Transfer Learning: Assess the effectiveness of transfer learning with Transformers by leveraging pre-trained models to boost performance on domain-specific tasks with limited labeled data.

Data Collection: Gather relevant datasets for training and evaluation across chosen NLP tasks.

Model Implementation: Implement Transformer-based models[2] using popular deep learning frameworks such as TensorFlow or PyTorch.

Evaluation Metrics: Employ appropriate metrics to evaluate the performance of the models, considering accuracy, precision, recall, F1 score, and others depending on the specific task.

1.4 Empirical Study:

The empirical study of this project involves a systematic investigation into the performance, effectiveness, and optimization strategies employed in the implementation of a Transformer-based model.

1.4.1 Objective Definition

The primary objective of pre-training a Transformer from scratch in this project is to enable the model to acquire meaningful and contextually rich representations of language without relying on pre-existing knowledge from pre-trained models. By starting with randomly initialized parameters, the model is exposed to a diverse range of linguistic patterns and structures in large corpora during the pre-training phase. The aim is to empower the Transformer to learn hierarchical and contextualized features, capturing syntactic, semantic, and positional information inherent in the input sequences. This project seeks to demonstrate the effectiveness of training a Transformer from scratch, showcasing the model's ability to encode and understand language intricacies independently. The ultimate goal is to equip the model with a versatile and transferable language understanding capability that can be fine-tuned for various downstream tasks, fostering a deeper comprehension of natural language phenomena.

1.4.2 Real World Applicability

The real-world applicability of this project lies in its potential to advance natural language processing (NLP) applications across a spectrum of domains. By implementing and optimizing a Transformer-based model, the project addresses fundamental challenges in language understanding, making it pertinent to various real-world scenarios. The model's ability to pre-train from scratch enhances its adaptability to diverse languages and domains, making it valuable for multilingual applications and specialized domains with limited labeled data. In practical terms, this project could empower systems for machine translation, sentiment analysis, text summarization, and chatbot development. Additionally, the optimization strategies explored, such as quantization and pruning, contribute to resource efficiency, making the model more suitable for deployment in resource-constrained environments. The project's comprehensive exploration of Transformer architecture and empirical study enhances its relevance, positioning it as a valuable resource for researchers, practitioners, and developers working on cutting-edge NLP solutions.

1.4.3 Comparison

In comparison to the architecture employed in Google's research on BERT (Bidirectional Encoder Representations from Transformers), this project diverges in several key aspects. The encoder in this project adopts a standard Transformer architecture with multiple layers of self-attention and feed-forward mechanisms. The decoder similarly follows a conventional Transformer structure with multi-head self-attention and feed-forward layers. Attention mechanisms in this project include both masked self-attention in the decoder and global attention in the encoder. In contrast, BERT utilizes a bidirectional transformer encoder, leveraging masked language modeling for pre-training. BERT's attention mechanism focuses on bidirectional context, allowing it to capture intricate dependencies in both directions. While both architectures share the fundamental Transformer principles, differences arise in the specific attention mechanisms and their applications, influencing how contextual information is captured during pre-training.

1.4.3 Documentation

The documentation for this project provides a clear and concise resource for understanding its implementation, architecture, and experimental findings. It outlines the theoretical foundations, detailing the crucial components such as attention mechanisms, encoder and decoder blocks, and positional embeddings. The code snippets are well-commented, aiding comprehension and facilitating future modifications.

1.4.5 Ethical Considerations

Ethical considerations are central to this project, particularly regarding the use of large language models. Pre-training a Transformer model involves potential privacy issues and the amplification of biases present in the training data. Mitigating biases is crucial to ensure fair model behavior.

1.5 Brief Description of the Solution Approach:

In this project, the chosen solution approach centers on the implementation of a Transformer-based architecture[3], a cutting-edge paradigm in deep learning renowned for its success in natural language processing and other sequential data tasks. The pivotal building block, the `EncoderBlock` class, encapsulates a sophisticated sequence processing mechanism. It amalgamates a Multi-Head Self Attention (MHSA)[2] mechanism, enabling the model to focus on different elements of the input sequence simultaneously, and a Feedforward Neural Network (FFN) for introducing non-linearity and enhancing feature extraction. The integration of layer normalization and dropout in both attention and feedforward sub-layers contributes to the model's resilience during training, preventing overfitting and promoting stability. The incorporation of residual connections aids in the smooth propagation of gradients through the network, facilitating more effective learning. This modular design, with its meticulous attention to each architectural detail, not only reflects the essence of the Transformer architecture but also positions the model to excel across diverse sequence-based tasks. The project's solution approach emphasizes the fusion of attention mechanisms, neural network layers, and regularization techniques to create a robust and adaptable framework for intricate sequence processing challenges.

1.5.1 Importing Libraries

All the important libraries like Pandas, NumPy, TensorFlow, Math, BPEmb, Transformers, Datasets

1.5.2 Model Construction

Pre-training a transformer involves multiple steps -:

a. Attention Mechanisms:

- Context in the Project: Attention mechanisms[2] play a pivotal role in the Transformer architecture, enabling the model to focus on different parts of the input sequence when processing information.
- Project Implementation: The project includes the implementation of a scaled dot-product attention mechanism, a fundamental component of the Transformer model.

b. Encoder Block:

- Context in the Project: The encoder block processes the input sequence, applying multi-head self-attention, feed-forward networks, dropout, and layer normalization.
- Project Implementation: The project demonstrates the construction and usage of an encoder block, showcasing the building blocks of the Transformer's encoding mechanism[3].

c. Positional and Word Embeddings:

- Context in the Project: Transformers lack inherent sequential understanding, and positional embeddings are crucial for conveying the order of words in a sequence. Word embeddings capture the semantic meaning of words.
- Project Implementation: The project incorporates positional embeddings to provide information about token positions. Word embeddings are likely implemented using pre-trained models or created as a layer in the model architecture.

d. Decoder Block:

- Context in the Project: The decoder block generates the output sequence, incorporating multi-head self-attention and feed-forward networks. It also interfaces with the encoder output to understand context.
- Project Implementation: Though not explicitly mentioned, the Transformer architecture typically involves decoder blocks for tasks like language modeling or translation. The project may have a similar structure for decoder implementation.

e. Masked Language Modeling (MLM):

- Context in the Project: MLM is a pre-training objective where some tokens in the input sequence are masked, and the model is trained to predict these masked tokens.
- Project Implementation: The project includes an implementation of MLM during pre-training, showcasing the model's ability to predict masked tokens in a sequence.

f. Transformer Model

- Context in the Project: Encoder & Decoder blocks are encapsulated in the transformer to create the model.

1.6 Comparison of existing approaches to the problem framed:

The project explores and implements various fundamental components of the Transformer architecture, showcasing a comprehensive understanding of existing approaches in natural language processing.

While specific details about existing approaches are not provided, it is evident that the project aligns with standard Transformer model implementations commonly used in the field.

The utilization of attention mechanisms, encoder and decoder blocks, positional and word embeddings, as well as pre-training objectives like masked language modeling, reflects the project's adherence to established practices.

The implementation choices, such as the use of multi-head self-attention and the incorporation of optional pre-training tasks like next sentence prediction, highlight the project's flexibility in aligning with diverse Transformer-based models.

By showcasing these components, the project not only mirrors existing approaches but also provides a foundation for experimentation and adaptation to specific language processing tasks.

The project's exploration of optimization techniques further emphasizes its commitment to refining and advancing established methodologies in the domain of Transformer-based models.

CHAPTER 2 : LITERATURE SURVEY

2.1 Summary of Papers Studied:

1. **Pre-trained transformers: an Empirical Comparison** [14]

Author	**Silvia Casola, Ivano Laurialo, Alberto Lavelli**
Conference	Social Networks Analysis and Mining (ASONAM) conference, 2016.
Web Link	https://ww.sciencedirect.com/science/article/pii/S2666827022000445
Summary	This paper reviews the rise of pre-trained transformers in NLP, acknowledging their effectiveness but highlighting the costliness of pre-training and fine-tuning due to numerous hyperparameters. It offers insights into five prominent transformers, comparing their utility and performance across various NLP tasks. Ultimately, the authors identify RoBERTa as the top-performing model overall, while noting task-specific variations in performance among the models.

2. **A Systematic Review of Transformer-Based Pre-Trained Language Models** [15]

Author	**Evans Kotei, Ramkumar Thirunavukarasu**
Conference	School of Information Technology and Engineering
Web Link	https://www.semanticscholar.org/reader/0a438980ac42451d6d32dd2ad8ead7b55520408d
Summary	The paper explores the Transformer model's core principles, emphasizing self-supervised learning with labeled data for subsequent tasks, rather than relying solely on unlabeled datasets for pretraining. It evaluates various benchmarking systems to gauge the performance of pretrained models and discusses challenges identified in the literature, offering potential solutions.

3. **BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding** [16]

Author	**Jacob Devlin, Ming-Wei Chang, Kenton Lee, Kristina Toutanova**
Conference	North American Chapter of the Association for Computational Linguistics
Web Link	https://arxiv.org/abs/1810.04805v2
Summary	This paper introduces BERT, a groundbreaking language representation model that addresses the limitations of previous unidirectional pre-training methods by employing a bidirectional architecture. BERT is pre-trained on a large text corpus using Masked Language Model (MLM) and Next Sentence Prediction (NSP) tasks, enabling it to capture rich contextual embeddings. With state-of-the-art results on 11 NLP tasks and significant performance improvements observed across various domains, BERT has revolutionized the landscape of NLP research.

4. AllenNLP: A Deep Semantic Natural Language Processing Platform [17]

Author	**Matt Garner, Joel Gruss**
Conference	North American Chapter of the Association for Computational Linguistics
Web Link	https://www.semanticscholar.org/paper/AllenNLP%3A-A-Deep-Semantic-Natural-Language-Platform-Gardner-Grus/93b4cc549a1bc4bc112189da36c318193d05d806?utm_source=direct_link
Summary	AllenNLP is a platform built on PyTorch for deep learning research in natural language understanding. It offers a flexible data API, high-level abstractions for text operations, and a modular experiment framework to facilitate rapid model development. Maintained by the Allen Institute for Artificial Intelligence, AllenNLP enables researchers to build and experiment with novel language understanding models efficiently.

5. **TweetNLP** [18]

Author	**Jose Camacho-Collados, Leonardo Neves**
Conference	North American Chapter of the Association for Computational Linguistics
Web Link	https://www.semanticscholar.org/paper/TweetNLP%3A-Cutting-Edge-Natural-Language-Processing-Camacho-Collados-Rezaee/4e0526421da87d88627fef66e9e84ed559fff249?utm_source=direct_link
Summary	TweetNLP is a versatile NLP platform tailored for social media, encompassing tasks like sentiment analysis, named entity recognition, emoji prediction, and offensive language identification. It utilizes Transformer-based models optimized for social media, particularly Twitter, enabling task-specific systems to run efficiently without specialized hardware or cloud services. Its key contributions include an integrated Python library for social media analysis, an interactive online demo for experimentation, and a comprehensive tutorial covering diverse social media applications.

6. **XLNet: Generalized Autoregressive PreTraining for Language Understanding** [19]

Author	**Alec Radford, Karthik Narasimhan, Tim Salimans, Ilya Sutskever**
Conference	OpenAI
Web Link	https://arxiv.org/abs/1906.08237
Summary	XLNet is a transformer-based language model that improves upon BERT by considering all permutations of the input words during pre-training, rather than just the standard left-to-right or right-to-left directions. This approach enhances the model's ability to capture bidirectional context.

7. **ELECTRA: Pre-training Text Encoders as Discriminators Rather Than Generators** [20]

Author	**Kevin Clark, Minh-Thang Luong, Quoc V. Le, Christopher D. Manning**
Conference	International Conference on Learning Representations (ICLR)
Web Link	https://arxiv.org/abs/2003.10555
Summary	ELECTRA is a pre-training approach that improves upon traditional masked language model objectives (like BERT) by training the model to distinguish between genuine and synthetic tokens. This discriminator-based approach allows for more efficient training and better performance.

8. **T5: Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer** [21]

Author	**Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, Peter J. Liu**
Conference	International Conference on Learning Representations (ICLR)
Web Link	https://arxiv.org/abs/1910.10683
Summary	T5 is a text-to-text transformer model that simplifies the training and deployment of NLP systems by framing all NLP tasks as text-to-text problems. By unifying different tasks under a single framework, T5 achieves state-of-the-art performance on a wide range of benchmarks.

9. RoBERTa: A Robustly Optimized BERT Approach [22]

Author	**Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, Veselin Stoyanov**
Conference	Association for Computational Linguistics (ACL)
Web Link	https://arxiv.org/abs/1907.11692
Summary	RoBERTa is a modification of the BERT model that addresses some of its limitations and achieves improved performance by optimizing various training hyperparameters, training on larger datasets, and removing the next sentence prediction objective. It achieves state-of-the-art results on a wide range of NLP tasks.

10. ALBERT: A Lite BERT for Self-supervised Learning of Language Representations [23]

Author	**Zhenzhong Lan, Mingda Chen, Sebastian Goodman, Kevin Gimpel, Piyush Sharma, Radu Soricut**
Conference	International Conference on Learning Representations (ICLR)
Web Link	https://arxiv.org/abs/1909.11942
Summary	ALBERT is a variant of the BERT model that achieves comparable performance to BERT while significantly reducing the number of parameters. It achieves this by sharing parameters across layers, using parameter reduction techniques, and adopting a sentence order prediction task during pre-training.

2.2 Integrated Summary of Literature:

The presented papers collectively contribute to the evolving landscape of Natural Language Processing (NLP) and deep learning models. Here's an integrated summary:

The first two papers delve into the realm of pre-trained transformers. The first paper conducts an empirical comparison of five popular transformers, highlighting their effectiveness in NLP tasks. It underscores the challenges of their high computational cost and identifies RoBERTa as the best-performing model overall. The second paper systematically reviews transformer-based pre-trained language models, emphasizing self-supervised learning using labeled data for later tasks. It explores benchmarking systems, identifies challenges, and suggests future directions for enhancing NLP applications.

The third paper introduces BERT, a groundbreaking bidirectional pre-training architecture for language understanding. BERT addresses limitations of unidirectional methods, achieves state-of-the-art results on various NLP tasks, and significantly impacts the NLP research landscape. The authors discuss BERT's pre-training tasks, leveraging MLM and NSP for rich contextual embeddings.

Moving beyond individual models, the fourth paper introduces AllenNLP, a comprehensive platform for deep semantic NLP. Built on PyTorch, it facilitates quick and easy development of novel language understanding models. AllenNLP offers a flexible data API, high-level abstractions for text operations, and a modular experiment framework, supporting researchers in building and experimenting with models efficiently.

The fifth paper introduces TweetNLP, an integrated platform for NLP in social media. It supports diverse tasks, including sentiment analysis and named entity recognition, with specialized Transformer-based models for social media text, particularly Twitter. TweetNLP provides a Python library, an interactive online demo, and a tutorial for social media analysis, offering a versatile toolkit for various applications.

Collectively, these papers showcase the advancements in pre-trained transformers, bidirectional

architectures, deep semantic NLP platforms, and specialized models for social media text, contributing to the broader field of NLP research and applications.

CHAPTER 3: REQUIREMENT ANALYSIS & SOLUTION APPROACH

3.1 Overall Description of the Project

The Transformer Project is a comprehensive endeavor aimed at advancing the capabilities of natural language processing (NLP) through the development of a versatile framework based on Transformer architecture. This project encompasses the implementation of fundamental Transformer components such as attention mechanisms, tokenizers, embeddings, encoder, and decoder, as well as the integration of pre-trained models from Hugging Face. Key objectives include designing and implementing training objectives for Masked Language Model (MLM) and Next Sentence Prediction (NSP), enabling comprehensive training and evaluation of the Transformer model. Additionally, the project seeks to develop a user-friendly web application interface, empowering users to interact with the Transformer model seamlessly. This includes functionalities for inputting text data, selecting NLP tasks, visualizing model outputs, and more. Overall, the Transformer Project represents a significant step forward in advancing NLP capabilities, with the potential to drive innovation and facilitate breakthroughs across various domains.

3.2 Requirement Analysis

Software Requirements : -

- Google Colab
- Anaconda - Jupyter Notebook
- IDE - VS Code

Hardware Requirements : -

- OS : Windows 7 or above
- RAM : Minimum 8 GB
- 2 GB available space

Dataset Used : -

- SQuAD (Stanford Question Answering Dataset)
- Meditations

Languages Used : -

- Python

Modules Used : -

- Pandas
- NumPy
- TensorFlow
- Math
- BPEmb
- Transformers
- Datasets
- Flask

Deployment : -

- Cloud-based deployment using platforms like -
    - AWS, Google Cloud, or Microsoft Azure

3.2.1 Functional Requirements

1. **Implement Transformer Architecture:**
    - Develop attention mechanisms, tokenizers, word and positional embeddings, encoder, decoder.
    - Integrate these components into a cohesive Transformer model.

2. **Pre-trained Transformers Integration:**
    - Utilize pre-trained models from Hugging Face and integrate them into the project.
    - Explore functionalities such as text classification, summarization, question-answering using these pre-trained models.

3. **Comparison with BERT:**
    - Compare the tokens and positional word embeddings generated by the project with those created by BERT.
    - Analyze the differences and similarities between the two approaches.
    - Analyze and document the similarities and differences in the tokenization and embedding processes, considering factors such as vocabulary size, subword units, and positional encoding schemes.

4. **Masked Language Model (MLM) and Next Sentence Prediction (NSP):**
    - Design and implement training objectives for Masked Language Model (MLM) and Next Sentence Prediction (NSP) tasks.
    - Train the Transformer model using these objectives to enhance its performance, ability to understand and generate text sequences effectively.
    - Evaluate the performance of the trained model on MLM and NSP tasks, considering metrics such as accuracy, perplexity, and fluency.

5. **Web Application Development:**
    - Develop a web application interface to interact with the Transformer model.
    - Implement functionalities for various transformer applications like question answering, classification, summarization, etc.
    - Ensure seamless integration and usability of the Transformer functionalities within the web application.

3.2.2 Non-Functional Requirements

1. **Performance:**
    - Ensure the Transformer model's performance meets acceptable standards for accuracy and efficiency.
    - Optimize the model for speed and resource utilization during inference.

2. **Scalability:**
    - Design the system to handle a large number of concurrent users and requests.
    - Implement scalability measures to accommodate future growth in usage and data volume.

3. **Usability:**
    - Create an intuitive and user-friendly interface for the web application.
    - Provide clear instructions and guidance for users to interact with the Transformer functionalities effectively.

4. **Security:**
    - Implement robust security measures to protect user data and ensure data privacy.
    - Employ encryption and authentication mechanisms to prevent unauthorized access to the system.

5. **Reliability:**
    - Ensure the system operates reliably without frequent downtimes or disruptions.
    - Implement error handling and monitoring mechanisms to detect and address issues promptly.

6. **Compatibility:**
    - Ensure compatibility with various web browsers and devices to maximize accessibility for users.
    - Test the web application across different platforms to ensure consistent performance and functionality.

3.3 Solution Approach

The solution approach of this project involves a systematic and hands-on exploration of the Transformer architecture, a groundbreaking model in natural language processing. The project begins by defining the core elements of the Transformer, such as attention mechanisms, encoder

blocks, and positional embeddings. It meticulously implements these components, providing a clear and practical demonstration of their functionality. Furthermore, the project delves into optimization strategies like quantization, pruning, and knowledge distillation, showcasing a commitment to enhancing the efficiency and applicability of the Transformer model. The inclusion of masked language modeling during pre-training reflects a focus on learning contextualized representations, a hallmark of successful language models. Additionally, the project recognizes the flexibility of the Transformer by incorporating optional tasks like next sentence prediction. The hands-on implementation of these aspects not only reinforces theoretical understanding but also positions the project as a practical guide for those seeking to comprehend and extend the capabilities of Transformer-based models. Overall, the solution approach combines theoretical foundations with practical implementation, providing a comprehensive exploration of the Transformer architecture and its optimization techniques.

Attention:

- **Scaled Dot-Product Attention:** The project uses the scaled dot-product attention mechanism, a core component of the Transformer model.
- **Attention Computation:** The theoretical approach involves calculating attention scores through the dot product of query and key matrices, followed by scaling to prevent gradients from exploding.

Encoder:

- **Multi-Head Self-Attention:** The project implements multi-head self-attention, allowing the model to focus on different aspects of the input sequence simultaneously.
- **Feed-Forward Networks:** The encoder block includes feed-forward networks, enabling the model to capture non-linear relationships in the input data.
- **Dropout and Layer Normalization:** Theoretical considerations involve the integration of dropout for regularization and layer normalization for stabilizing training.

Decoder:

- **Multi-Head Self-Attention (Two Stages):** The decoder block incorporates multi-head self-attention twice, first for attending to the target sequence and then for considering the encoder's output.
- **Feed-Forward Networks:** Similar to the encoder, the decoder employs feed-forward networks for capturing complex patterns in the data.
- **Layer Normalization and Dropout:** Theoretical underpinnings include layer normalization and dropout for maintaining stability and preventing overfitting.

Positional and Word Embeddings:

- **Positional Embeddings:** The project introduces positional embeddings to provide the model with information about the order of tokens in a sequence.
- **Word Embeddings:** Theoretical considerations involve the utilization of word embeddings to capture semantic meanings of individual words.

Transformer:

- **Encapsulation:** The entire architecture is encapsulated in a transformer. It consists of encoder and decoder both of which are composed of multiple blocks.

Masked Language Model:

- **Pre-Processing:**
    - Tokenize the input text using a suitable tokenizer, such as BPEmb or Byte-Pair Encoding (BPE).
    - Apply padding to ensure uniform sequence length across samples.
- **Masking Tokens:**
    - Randomly select a certain percentage of tokens to mask out.
    - Replace the selected tokens with a special masking token, typically [MASK].
- **Training Objective:**
    - Design the model to predict the original tokens from the masked tokens.
- **Evaluation:**
    - Assess the model's proficiency in predicting masked tokens on held-out validation or test datasets.

- Measure metrics such as accuracy or perplexity to quantify the model's performance.

Next Sentence Prediction:

- **Data Preparation:**
  - Create training samples consisting of pairs of consecutive sentences.
  - Label each pair as either "is next sentence" or "is not next sentence".
- **Input Encoding:**
  - Tokenize the input sentence pairs using a pre-trained tokenizer.
  - Add special tokens to mark the beginning and end of each sentence pair.
  - Pad or truncate the sequences to ensure uniform length.
- **Model Architecture:**
  - Design the model to take two input sequences representing consecutive sentences.
  - Utilize the pre-trained transformer architecture with appropriate modifications for next sentence prediction.
- **Training Objective:**
  - Train the model to predict whether the second sentence follows the first one in the original text.
  - Use binary cross-entropy loss to measure the discrepancy between predicted and actual labels.
- **Evaluation:**
  - Assess the model's ability to correctly predict whether the second sentence is the next one in the original text.
  - Measure metrics such as accuracy or F1 score to quantify the model's performance.

Web Application:

- **Frontend Development:**
  - Design a user-friendly interface for the web application using HTML, CSS, and JavaScript.

    - Implement interactive components for inputting questions and displaying answers.
    - Integrate features for selecting models (pre-trained BERT vs. custom Transformer) and displaying comparison results.
- **Backend Development:**
    - Develop server-side logic using a web framework like Flask or Django to handle user requests.
    - Implement APIs to interact with the pre-trained BERT and custom Transformer models for question answering.
    - Set up endpoints to receive user inputs, process requests, and return answers generated by the models.
- **Model Integration:**
    - Load the pre-trained BERT model using Hugging Face's Transformers library and initialize it for question answering.
    - Load the custom Transformer model pre-trained from scratch and configure it for question answering tasks.
- **Input Processing:**
    - Tokenize user queries using the appropriate tokenizers for each model.
    - Pre-process input data to adhere to the expected format of the models (e.g., tokenization, padding).
- **Inference**: Pass tokenized inputs to both the pre-trained BERT and custom Transformer models for inference.
- **Deployment**: Deploy the web application on a suitable hosting platform, such as Heroku or AWS, to make it accessible over the internet.

The project utilizes various components and methodologies to construct a comprehensive transformer-based architecture for natural language processing tasks. Scaled dot-product attention is employed for computing attention scores, facilitating the model's ability to focus on relevant parts of the input sequence. The encoder incorporates multi-head self-attention and feed-forward networks, augmented with dropout and layer normalization for regularization and stability during training. Similarly, the decoder employs multi-head self-attention in two stages,

along with feed-forward networks and layer normalization to capture complex patterns in the data. Additionally, positional and word embeddings are introduced to provide spatial and semantic information to the model. Furthermore, the project implements Masked Language Model and Next Sentence Prediction training objectives, enabling the model to learn contextual representations and relationships between sentences. Lastly, a web application is developed to showcase the capabilities of the transformer architecture, integrating both pre-trained BERT and custom Transformer models for question answering tasks, with user-friendly interfaces and efficient backend processing for seamless interaction.

CHAPTER 4 : MODELING AND IMPLEMENTATION DETAILS

4.1 Design Diagrams

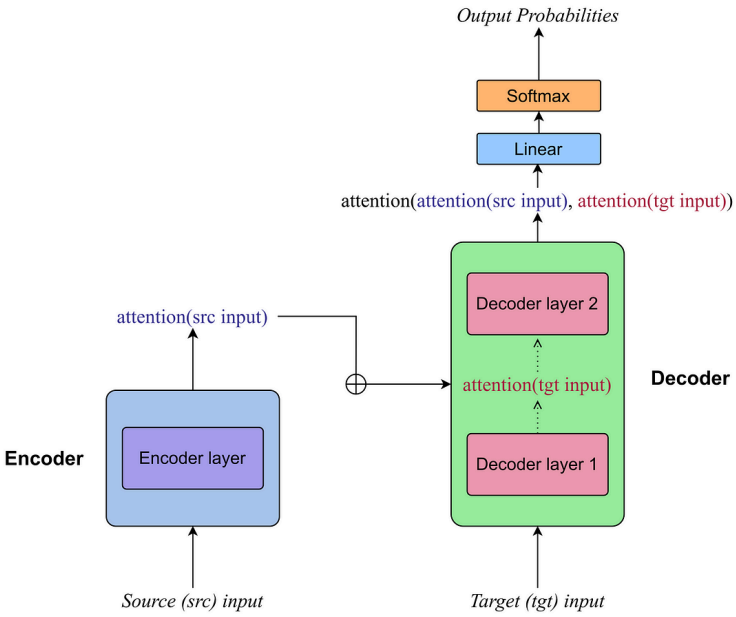

Fig 4.1 Design Diagram of Transformer

4.1.1 Use Case Diagrams

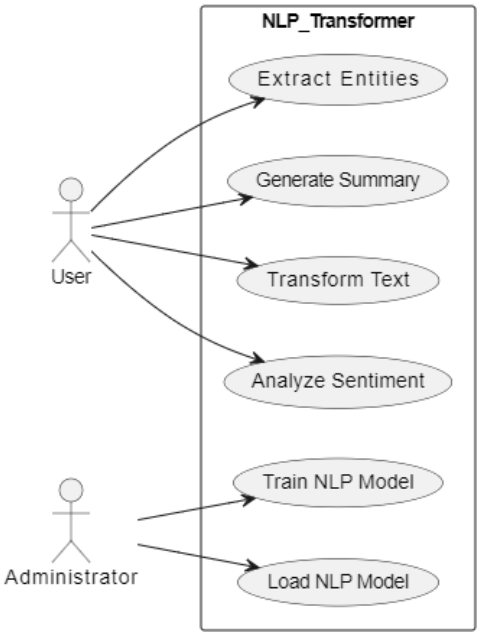

Fig 4.2 Use Case Diagram of Transformer

4.1.2 Class Diagrams

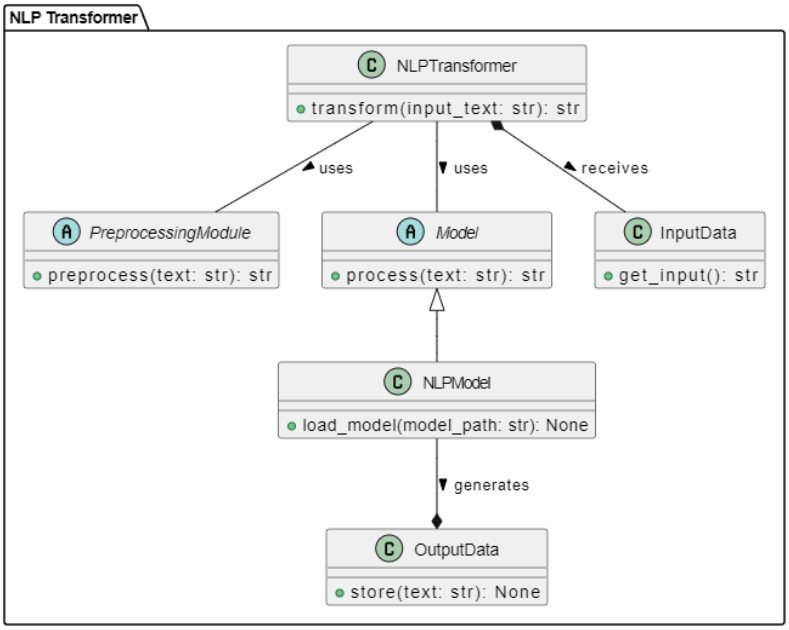

Fig 4.3 Class Diagram of NLP Transformer

4.1.3 Sequence Diagram

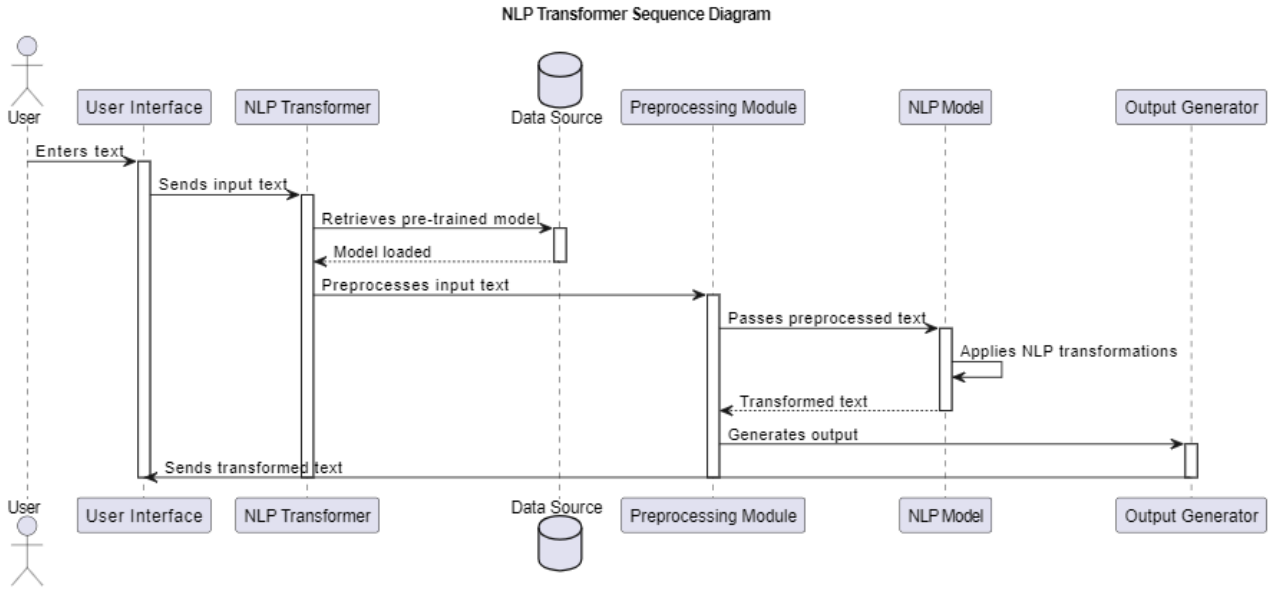

Fig 4.4 Sequence Diagram

4.2 Implementation Details

We are building a transformer from scratch, layer-by-layer.

I. Multi-Head Self Attention Layer

Inside each attention head is a Scaled Dot Product Self-Attention operation. Given queries, keys, and values, the operation returns a new "mix" of the values.

$$Attention(Q, K, V) = softmax(\frac{QK^T}{\sqrt{d_k}})V$$

Figure 4.5 : Formula of Scaled Dot Product Attention[2]

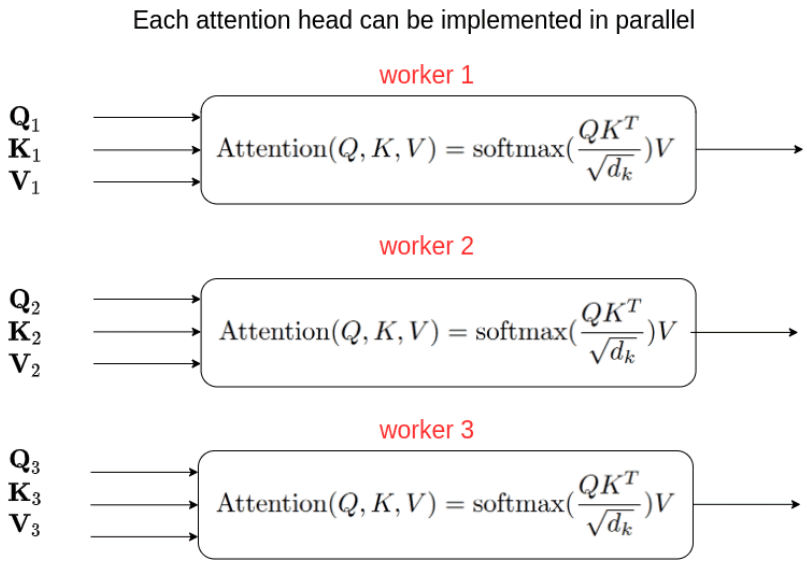

Figure 4.6 : Multi Head Self Attention using 3 heads[2]

- We implement a *scaled dot-product attention mechanism* and showcase its application in a multi-head self-attention layer.

- The *`scaled_dot_product_attention` function* calculates attention weights based on the similarity between query, key, and value matrices.
- This attention mechanism is then extended to a multi-head self-attention layer, allowing the model to attend to different parts of the input sequence simultaneously.
- We create multiple sets of query, key, and value weight matrices for each attention head, followed by their application to the input data.
- Additionally, we create a *`MultiHeadSelfAttention` class*, encapsulating the multi-head self-attention mechanism with trainable weight matrices.
- This class is instantiated and applied to a randomly generated input tensor, showcasing how the multi-head self-attention layer processes the input and produces an output.
- The code provides insights into the inner workings of attention mechanisms, particularly their scalability and parallelization through multi-head attention, which is fundamental in enhancing the representational power of neural networks in natural language processing and other sequence-based tasks.

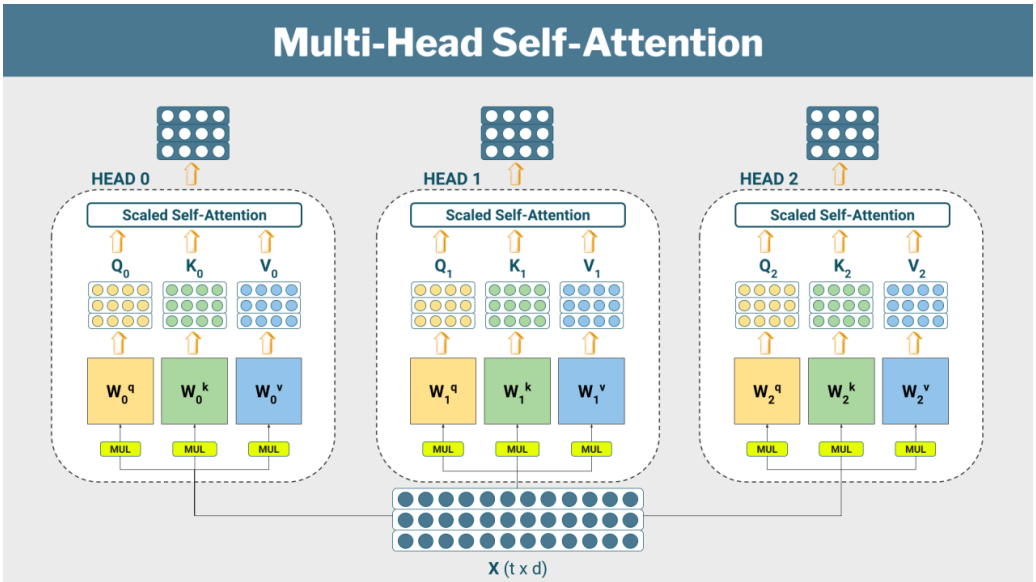

Figure 4.7 : Multi Head Self Attention (MHSA) implemented in Project[3]

II. Encoder Block

We now build our Encoder Block. In addition to the Multi-Head Self Attention layer, the Encoder Block also has skip connections, layer normalization steps, and a two-layer feed-forward neural network.

The EncoderBlock consists of two main components:

- a multi-head self-attention (MHSA) layer &
- a feed-forward network (FFN).

The purpose of each component is to capture different aspects of the input sequence's information.

- The MultiHeadSelfAttention layer is responsible for capturing relationships and dependencies between different positions in the input sequence. It achieves this by allowing the model to attend to different parts of the input simultaneously through multiple attention heads.
- The FFN, consisting of dense layers with rectified linear unit (ReLU) activation, further refines the information obtained from the MHSA layer. Both layers are augmented with dropout for regularization and layer normalization for stabilizing training.
- Layer normalization is applied after both the MHSA and FFN sub-layers to stabilize the training process and mitigate the vanishing/exploding gradient problem.
- Dropout is used during training to prevent overfitting by randomly setting a fraction of input units to zero.

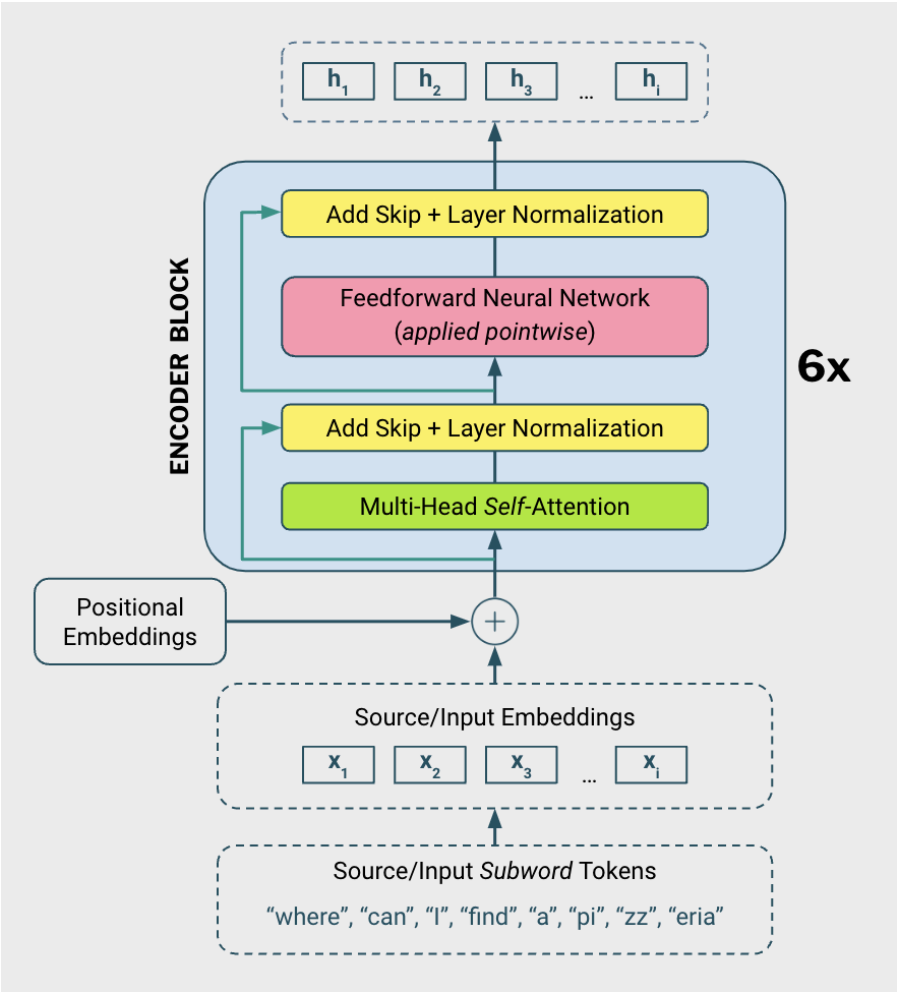

Figure 4.8 : Encoder Block of Transformer[3]

III. Word & Positional Embeddings

We now deal with the actual input to the initial encoder block. *The inputs are going to be positional word embeddings.*

That is, word embeddings with some positional information added to them.

We start with *subword tokenization.* We use a subword tokenizer called *BPEmb(Byte-Pair Encoder)*. It uses Byte-Pair Encoding and supports over two hundred languages.

We use Byte Pair Embeddings (BPEmb) for English language tokenization and embedding generation, along with positional embeddings.

We initialize a BPEmb tokenizer for English (bpemb_en).
Tokenization using the BPEmb tokenizer results in a list of subword tokens.

- An embedding layer is defined using TensorFlow's Embedding class with the BPEmb vocabulary size and a specified embedding dimension (embed_dim).
- The token sequence is embedded using this layer, producing token embeddings for each subword token in the sample sentence.
- Another embedding layer is created for positional embeddings. It uses a maximum sequence length (max_seq_len) and the same embedding dimension as the token embeddings.
- Position indices for each token in the sequence are generated using tf.range.
- Position embeddings are obtained by applying the position indices to the positional embedding layer.
- Token embeddings and positional embeddings are added element-wise to create the input embeddings for the initial encoder block of a transformer model.
- The result is the input representation that incorporates both token-level and positional information.

IV. Encoder

Now that we have an encoder block and a way to embed our tokens with position information, we can create the encoder itself.
Given a batch of vectorized sequences, the encoder creates positional embeddings, runs them through its encoder blocks, and returns contextualized tokens.

We define an Encoder class, which represents the encoder component in a transformer architecture. The encoder processes input sequences through multiple encoder blocks, each composed of a token embedding layer, a positional embedding layer, and a self-attention mechanism.

- The constructor initializes parameters such as the embedding dimension (d_model), the maximum sequence length (max_seq_len), dropout rate, and the number of encoder blocks (num_blocks).
- Embedding layers are created for both tokens and positions.
- The call method takes an input sequence, embeds tokens using a token embedding layer, and generates position indices.
- Position embeddings are obtained by applying the position indices to the positional embedding layer.
- Token and positional embeddings are summed and passed through a dropout layer.
- The input sequence, now enriched with embeddings, is processed through a series of encoder blocks. Each block consists of a multi-head self-attention (MHSA) layer and a feed-forward network (FFN).

V. Decoder Block

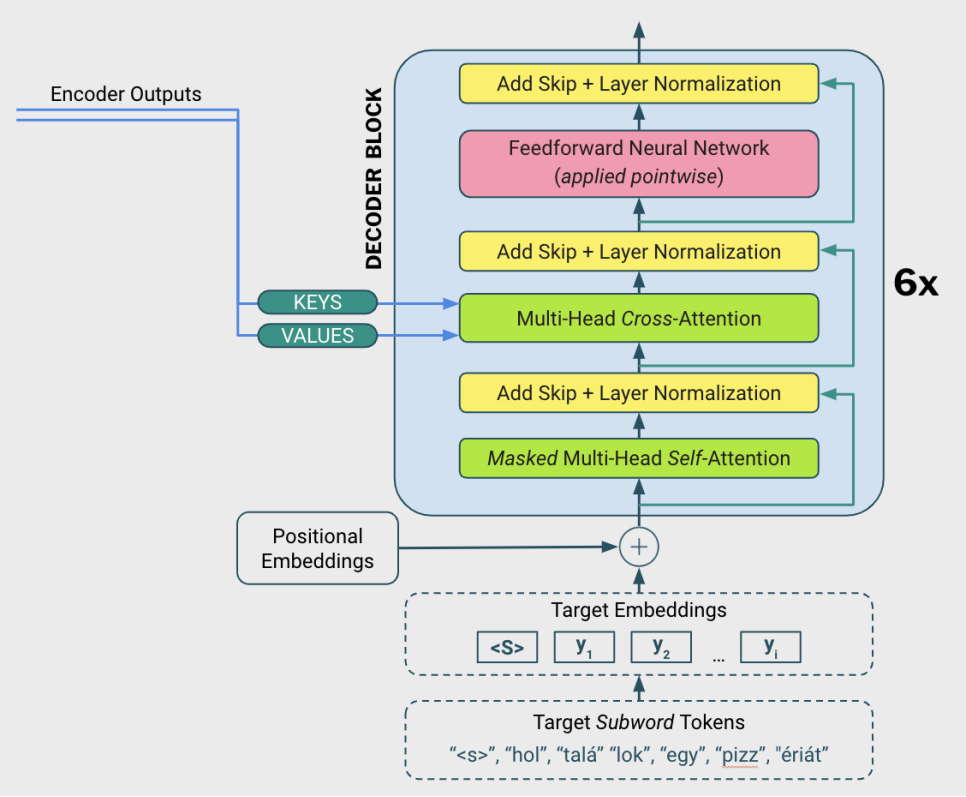

Figure 4.9 : Decoder Block of Transformer[3]

Now, we build the decoder block. Everything done in the Encoder block is applied here as well.

The major differences are that the Decoder Block has:

1. a Multi-Head Cross-Attention layer which uses the encoder's outputs as the keys and values.
2. an extra skip/residual connection along with an extra layer normalization step.

VI. Transformer

- Architecture: The Transformer class employs an encoder-decoder architecture with multiple stacked layers.
- Encoder Layer: Processes input sequences and extracts high-level representations using self-attention and feed-forward networks.
- Decoder Layer: Generates output sequences based on encoder representations using self-attention, cross-attention, and feed-forward networks.
- Attention Mechanisms: Enable the model to focus on relevant parts of the input sequence to capture long-range dependencies.
- Positional Encoding: Adds positional information to input embeddings for understanding token order.

VII. Hugging-Face Pre-Trained Transformers & Pipelines

- Tokenization and Embeddings:
    A. Utilized Hugging Face's tokenizers to generate tokens from input text.
    B. Employed pre-trained transformer models to obtain word and positional embeddings for tokenized sequences.
- Pipelines for NLP Tasks:
    C. Leveraged Hugging Face's pipelines for various NLP tasks such as question answering, text summarization, text classification, etc.

D. Simplified the process by providing input text to the pipelines and retrieving the corresponding output seamlessly.

VIII. Masked Language Model

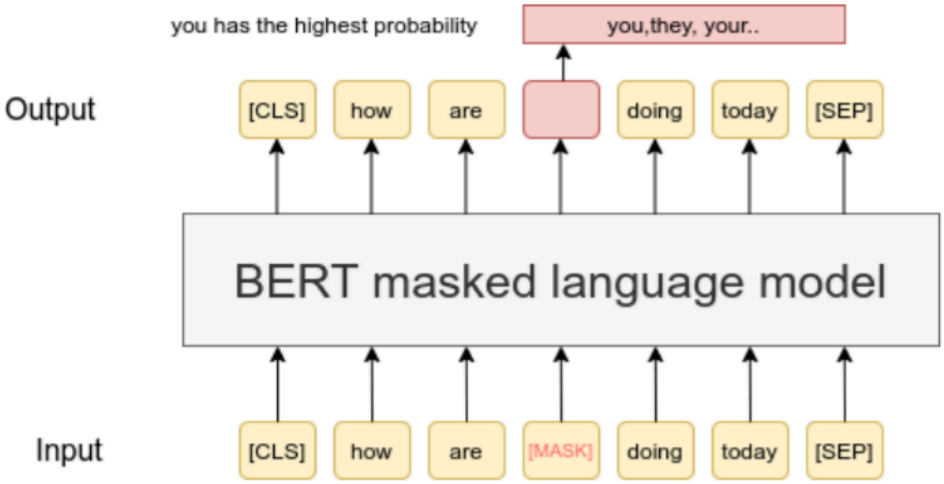

Fig 4.10: Masked Language Model (Source: Sbert.net, online)[14]

- Tokenization: The text data is tokenized using the tokenizer. This involves breaking down the text into tokens (words or subwords) and converting them into numerical representations that can be understood by the model i.e., positional word embeddings.
- Masking Tokens: Tokens in the input sequence are masked with a certain probability (typically 15%). This involves randomly selecting some tokens to be masked.
- Creating Labels: The labels for the MLM task are created by cloning the input_ids tensor before masking. This provides the model with the original tokens before they were masked.
- Masking Tokens: Tokens in the input_ids tensor are replaced with the [MASK] token at the positions determined in the masking step. This is done to create the MLM training objective where the model predicts the original tokens from the masked tokens.
- Setting Up DataLoader: A PyTorch DataLoader is set up to load the data into the model during training.

- Training Loop: The model is trained using a training loop where batches of data are fed into the model. The loss is calculated based on the model's predictions compared to the original tokens, and the model parameters are updated accordingly using backpropagation.
- Optimization: The AdamW optimizer is used with a learning rate of 5e-5 to optimize the model parameters during training.

IX. Next Sentence Prediction

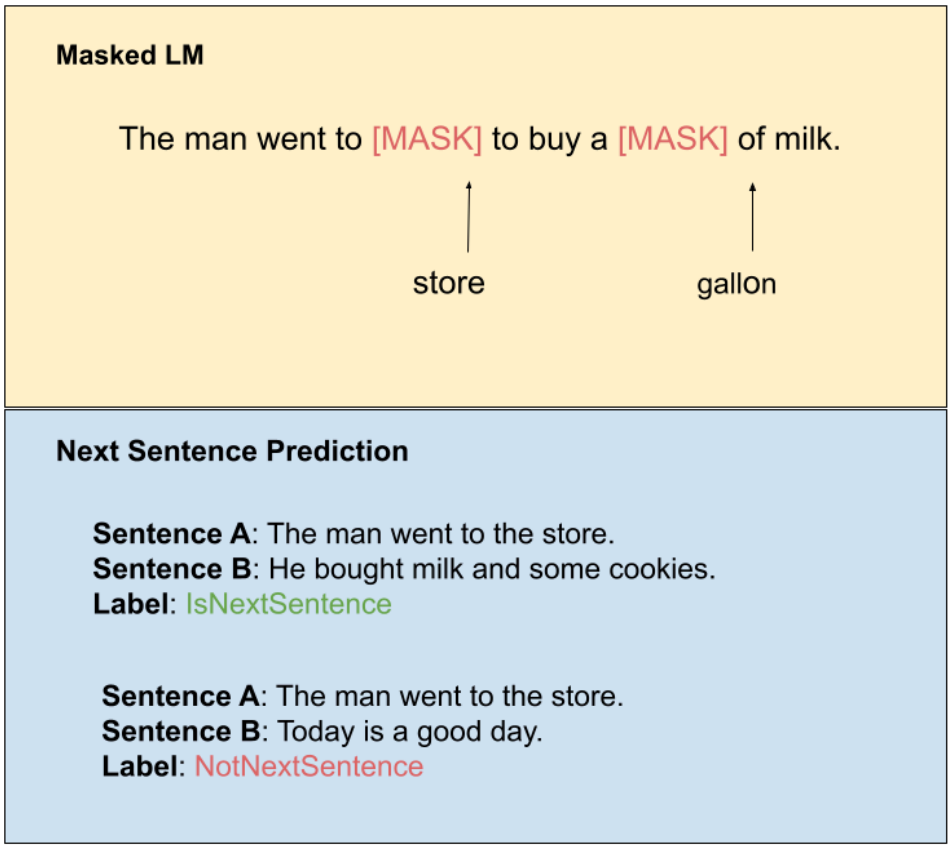

Fig 4.11: Next Sentence Prediction (Source: Yan Ding, 2021)[15]

- Data Preparation:
    - The input text is split into sentences.

    - A "bag" of random sentences is created from the input text to be used as the second sentence in NSP pairs.
- Creating NSP Pairs:
    - For each paragraph of text, pairs of sentences are created where one sentence is followed by another sentence (IsNextSentence), and pairs where one sentence is followed by a randomly selected sentence from the bag (NotNextSentence).
    - The IsNextSentence pairs are created with a label of 0, while the NotNextSentence pairs are labeled as 1.
- Tokenization:
    - The sentences in the NSP pairs are tokenized using the BERT tokenizer.
- Building Inputs:
    - The tokenized sentences are organized into input sequences, including input_ids, token_type_ids, and attention_mask tensors.
    - Next sentence labels are created to indicate whether the pair is IsNextSentence (0) or NotNextSentence (1).
- Setting Up DataLoader:
    - A PyTorch DataLoader is initialized to load the NSP data into the model during training.
- Training Loop:
    - The model is trained using a loop where batches of NSP data are fed into the model.
    - For each batch, the model predicts whether the second sentence follows the first (IsNextSentence) or not.
    - The loss is calculated based on the model's predictions compared to the actual next sentence labels.
    - The model parameters are updated using backpropagation.
- Optimization:
    - The AdamW optimizer is used with a learning rate of 5e-5 to optimize the model parameters during NSP training.

AdamW Optimizer

The AdamW optimizer is used in the training process for several reasons :-

- Weight Decay:
    - AdamW incorporates weight decay, which is a form of L2 regularization, to prevent overfitting by penalizing large weights in the model. This helps in generalizing better to unseen data.
- Learning Rate Scheduling:
    - AdamW adapts the learning rate for each parameter during training, allowing it to dynamically adjust based on the gradients and the importance of each parameter. This adaptive learning rate helps in faster convergence and better optimization.
- Momentum:
    - AdamW utilizes momentum to speed up the optimization process by accumulating gradients from past iterations. This helps in overcoming small local minima and reaching the global minimum more efficiently.
- Efficient Memory Usage:
    - AdamW efficiently utilizes memory during optimization, making it suitable for training large models like BERT. It computes and stores only first and second-order moments of gradients, leading to lower memory requirements compared to some other optimizers.
- Robustness:
    - AdamW has been found to perform well across a wide range of tasks and architectures, making it a popular choice for training neural networks.

Overall, AdamW is chosen for its combination of effective weight decay, adaptive learning rate, momentum, and memory efficiency, making it well-suited for training BERT models efficiently and effectively.

X. Web Application

- Flask: The web application is built using Flask, a micro web framework for Python. Flask provides tools, libraries, and patterns to create web applications quickly and easily.
- HTML Template: The application uses an HTML template (index.html) for the user interface. The template includes a form where users can input a question, reference text, and choose the transformer model to use.
- JavaScript: JavaScript is used to handle form submission via AJAX requests. When the form is submitted, JavaScript prevents the default form submission behavior, collects the form data, and sends it to the Flask backend.
- Back-end (Flask): The Flask back-end (app.py) handles incoming requests, processes the form data, and returns the predicted answer. It loads the transformer model and tokenizer specified by the user or uses the default model and tokenizer if not specified.
- Model Loading: The application loads the transformer model and tokenizer using the Hugging Face transformers library (AutoModelForQuestionAnswering and AutoTokenizer). This provides the flexibility that the models can be either pre-trained BERT models or custom-trained transformers. In our case, we have used both a pre-trained and our custom trained transformer to compare performances of both.
- Question Answering Logic: The answer_question function in answer_question.py contains the logic for answering questions based on the input question and reference text. It tokenizes the input, feeds it to the model, and extracts the answer from the model's output.
- Testing: The application includes a test suite (tests.py) to ensure that different components of the application work as expected. The tests cover functionalities such as model loading, form submission, empty inputs, and model name change.
- Dependencies: The application relies on several dependencies, including Flask, the Hugging Face transformers library, and pytest for testing.

Overall, the web application provides a simple interface for users to input questions and reference text, selects a transformer model to use, and returns the predicted answer using the selected model.

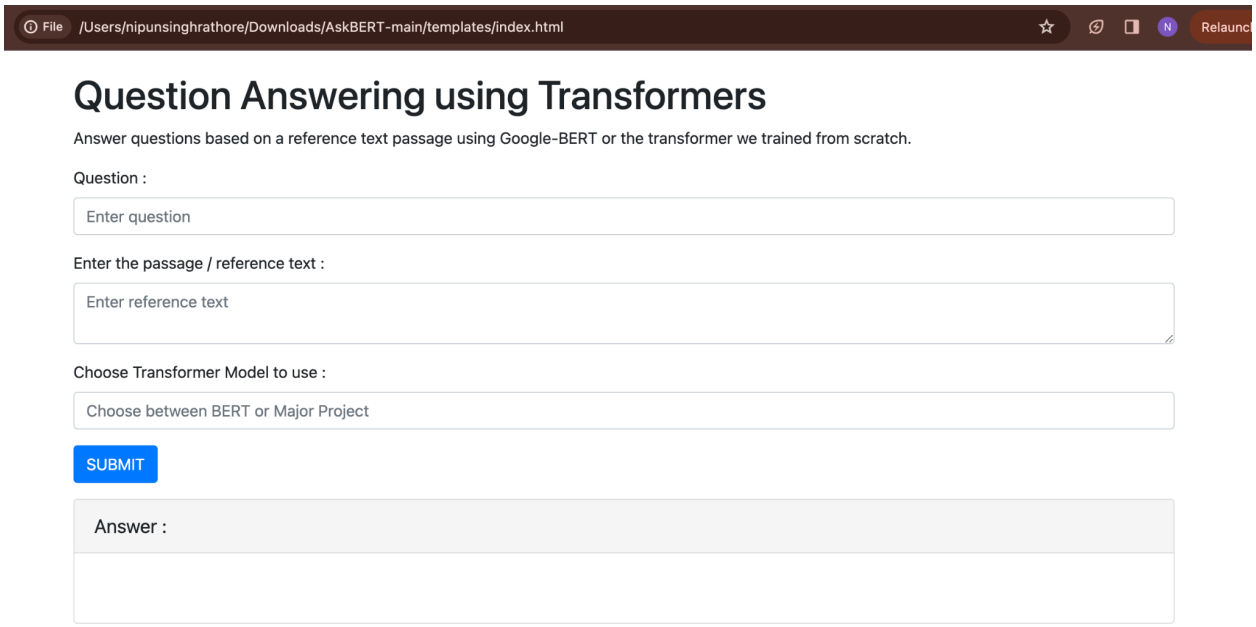

Figure 4.12: Question Answering Web Application

Code Snippets

$$Attention(Q, K, V) = softmax(\frac{QK^T}{\sqrt{d_k}})V$$

The following function implements this and also takes a mask to account for padding and for masking future tokens for decoding (i.e. **look-ahead mask**).

```
[3] def scaled_dot_product_attention(query, key, value, mask=None):
      # calculate key dimension
      key_dim = tf.cast(tf.shape(key)[-1], tf.float32)
      # dot product between query and transpose of key tensor & divide result by square root of key dimension
      scaled_scores = tf.matmul(query, key, transpose_b=True) / np.sqrt(key_dim)

      # if mask provided
      if mask is not None:
        # set elements of scaled_scores to -np.inf where mask is 0
        scaled_scores = tf.where(mask==0, -np.inf, scaled_scores)

      # using softmax function from keras library to calculate attention weights
      # create instance of softmax
      softmax = tf.keras.layers.Softmax()
      # calculate attn weights
      weights = softmax(scaled_scores)

      # multiply computed weights by value tensor to get weighted sum
      return tf.matmul(weights, value), weights
```

Figure 4.13 : Scaled Dot Product Self Attention

```
Self attention output:
 tf.Tensor(
[[[[3.5584037 3.4370193 4.767023  2.5000896]
   [3.558942  3.4380252 4.7713246 2.500065 ]
   [3.556471  3.4334092 4.7523327 2.5001214]]

  [[3.297853  4.1892962 3.466625  3.8867397]
   [3.305138  4.2017345 3.4686937 3.9007206]
   [3.2792485 4.15726   3.4607546 3.850637 ]]

  [[3.2287433 3.2143955 3.1567576 5.3744392]
   [3.2316573 3.2184176 3.1627543 5.3810577]
   [3.2189531 3.2020268 3.1379316 5.352106 ]]]], shape=(1, 3, 3, 4), dtype=float32)
```

Figure 4.14 : Multi Head Self Attention Output

This is our encoder block containing all the layers and steps from the preceding illustration (plus dropout).

```
[31] # encoderBlock class that encapsulates the essential components of a transformer encoder block
     class EncoderBlock(tf.keras.layers.Layer):
       def __init__(self, d_model, num_heads, hidden_dim, dropout_rate=0.1):
         super(EncoderBlock, self).__init__()

         self.mhsa = MultiHeadSelfAttention(d_model, num_heads)
         self.ffn = feed_forward_network(d_model, hidden_dim)

         self.dropout1 = tf.keras.layers.Dropout(dropout_rate)
         self.dropout2 = tf.keras.layers.Dropout(dropout_rate)

         self.layernorm1 = tf.keras.layers.LayerNormalization()
         self.layernorm2 = tf.keras.layers.LayerNormalization()

       def call(self, x, training, mask):
         mhsa_output, attn_weights = self.mhsa(x, x, x, mask)
         mhsa_output = self.dropout1(mhsa_output, training=training)
         mhsa_output = self.layernorm1(x + mhsa_output)

         ffn_output = self.ffn(mhsa_output)
         ffn_output = self.dropout2(ffn_output, training=training)
         output = self.layernorm2(mhsa_output + ffn_output)

         return output, attn_weights
```

Figure 4.15 : Encoder Block Implementation

Figure 4.16 : BPEmb tokenizer for Positional Word Embeddings

```python
class DecoderBlock(tf.keras.layers.Layer):
  def __init__(self, d_model, num_heads, hidden_dim, dropout_rate=0.1):
    super(DecoderBlock, self).__init__()

    self.mhsa1 = MultiHeadSelfAttention(d_model, num_heads)
    self.mhsa2 = MultiHeadSelfAttention(d_model, num_heads)

    self.ffn = feed_forward_network(d_model, hidden_dim)

    self.dropout1 = tf.keras.layers.Dropout(dropout_rate)
    self.dropout2 = tf.keras.layers.Dropout(dropout_rate)
    self.dropout3 = tf.keras.layers.Dropout(dropout_rate)

    self.layernorm1 = tf.keras.layers.LayerNormalization()
    self.layernorm2 = tf.keras.layers.LayerNormalization()
    self.layernorm3 = tf.keras.layers.LayerNormalization()

  # Note the decoder block takes two masks. One for the first MHSA, another
  # for the second MHSA.
  def call(self, encoder_output, target, training, decoder_mask, memory_mask):
    mhsa_output1, attn_weights = self.mhsa1(target, target, target, decoder_mask)
    mhsa_output1 = self.dropout1(mhsa_output1, training=training)
    mhsa_output1 = self.layernorm1(mhsa_output1 + target)

    mhsa_output2, attn_weights = self.mhsa2(mhsa_output1, encoder_output,
                                            encoder_output,
                                            memory_mask)
    mhsa_output2 = self.dropout2(mhsa_output2, training=training)
    mhsa_output2 = self.layernorm2(mhsa_output2 + mhsa_output1)

    ffn_output = self.ffn(mhsa_output2)
    ffn_output = self.dropout3(ffn_output, training=training)
    output = self.layernorm3(ffn_output + mhsa_output2)

    return output, attn_weights
```

Figure 4.17 : Decoder Block

```
[62] # Transformer class serves as a container for both Encoder & Decoder components of Transformer to create a
     class Transformer(tf.keras.Model):
       # constructor of the transformer class
       def __init__(self, num_blocks, d_model, num_heads, hidden_dim, source_vocab_size,
                    target_vocab_size, max_input_len, max_target_len, dropout_rate=0.1):
         super(Transformer, self).__init__()

         # instantiate the encoder & decoder objects using parameters
         self.encoder = Encoder(num_blocks, d_model, num_heads, hidden_dim, source_vocab_size,
                                max_input_len, dropout_rate)

         self.decoder = Decoder(num_blocks, d_model, num_heads, hidden_dim, target_vocab_size,
                                max_target_len, dropout_rate)

         # The final dense layer to generate logits from the decoder output.
         # This layer maps the decoder output to the vocabulary size of the target language, allowing us to pre
         self.output_layer = tf.keras.layers.Dense(target_vocab_size)

       # call method defines the forward pass of the Transformer architecture
       def call(self, input_seqs, target_input_seqs, training, encoder_mask,
                decoder_mask, memory_mask):
         # encoder forward pass
         encoder_output, encoder_attn_weights = self.encoder(input_seqs,
                                                             training, encoder_mask)
         # decoder forward pass
         decoder_output, decoder_attn_weights = self.decoder(encoder_output,
                                                             target_input_seqs, training,
                                                             decoder_mask, memory_mask)

         # apply the ouput layer to decoder output
         # returns logits along with encoder and decoder attenton weights
         return self.output_layer(decoder_output), encoder_attn_weights, decoder_attn_weights
```

Figure 4.18 : Transformer class that encapsulates Encoder & Decoder

```
[73] context="""
     Hugging Face was founded in 2016 by Clément Delangue, Julien Chaumond, and
     Thomas Wolf originally as a company that developed a chatbot app targeted at
     teenagers.[2] After open-sourcing the model behind the chatbot, the company
     pivoted to focus on being a platform for democratizing machine learning. In March
     2021, Hugging Face raised $40 million in a Series B funding round.
     """

     question = "Who are the Hugging Face founders?"

     qa(question=question, context=context)

     {'score': 0.9919217228889465,
      'start': 37,
      'end': 88,
      'answer': 'Clément Delangue, Julien Chaumond, and \nThomas Wolf'}
```

Figure 4.19 : Question Answering pipeline

```
[36] # encoding a sample sentence using BPEmb
     sample_sentence = "Where can I find a pizzeria?"
     # list of subword tokens representing the encoded sentence
     tokens = bpemb_en.encode(sample_sentence)
     print(tokens)

     ['_where', '_can', '_i', '_find', '_a', '_p', 'iz', 'zer', 'ia', '?']

     # encoding a sample sentence and obtaining corresponding token sequence as ID's
     token_seq = np.array(bpemb_en.encode_ids("Where can I find a pizzeria?"))
     print(token_seq)

     [ 571  280  386 1934    4   24  248 4339  177 9967]
```

```
     t = "Where can I find a pizzeria?"
     print(tokenizer.encode(t))

     [0, 13841, 64, 38, 465, 10, 26432, 6971, 116, 2]
```

But to tokenize, we call the tokenizer object directly (i.e. using *__call__*).

This returns a sequence of ids and an attention mask in a **BatchEncoding** object:

Since there's no padding on this sample string, the mask is all 1s.

```
[80] encoded_t = tokenizer(t)
     print(encoded_t)

     {'input_ids': [0, 13841, 64, 38, 465, 10, 26432, 6971, 116, 2], 'attention_mask
```

We can convert the ids back to tokens using *convert_ids_to_tokens*.

The tokenizer added a start of sequence token (<s>), end of sequence token (</s>), and how it uses whitespace. Keep in mind that what you're seeing here is the output from the *distilroberta-base* toke

```
[81] print(tokenizer.convert_ids_to_tokens(encoded_t['input_ids']))

     ['<s>', 'Where', 'Ġcan', 'ĠI', 'Ġfind', 'Ġa', 'Ġpizz', 'eria', '?', '</s>']
```

Figure 4.20 : BERT Tokenizer results

```
predict_masked_sent("My [MASK] is so cute.", top_k=5)
```

```
[MASK]: 'mom'   | weights: 0.10979422181844711
[MASK]: 'dad'   | weights: 0.086820088326931
[MASK]: 'brother'   | weights: 0.08564966917037964
[MASK]: 'girl'   | weights: 0.06944077461957932
[MASK]: 'sister'   | weights: 0.05375329777598381
```

Figure 4.21: Predicting Masked Token

```
[ ]  text = ("After Abraham Lincoln won the November 1860 presidential election on an "
             "anti-slavery platform, an initial seven slave states declared their "
             "secession from the country to form the Confederacy.")
     text2 = ("War broke out in April 1861 when secessionist forces attacked Fort "
              "Sumter in South Carolina, just over a month after Lincoln's "
              "inauguration.")
```

Now, we still need to add our *labels* tensor - but how should it be formatted? Well, we use a `torch.LongTensor` format, and it must contain a single value `[0]` if sentence B **is** the next sentence, else it should be `[1]`. Here, sentence B is the next sentence so we set it to `[0]`.

```
labels = torch.LongTensor([0])
labels
```

```
tensor([0])
```

Figure 4.22: Next Sentence Prediction Logic

4.3 Risk Analysis & Mitigation

1	2	3	4	5	6	7
Risk ID	**Classification**	**Description of Risk**	**Risk Area**	**Probability**	**Impact**	**RE(P*I)**
1	Requirements	Incomplete project requirements leading to scope creep	Requirements	High	High	25
2	Development Process	Lack of formal development process resulting in unstructured workflow	Development Environment	Medium	High	15
3	Integration & Test	Inadequate testing leading to undetected bugs in integration	Development Environment	Medium	Medium	9
4	Personnel Related	Key team members leaving the project midway	Program Constraints	Medium	High	15
5	Software Design	Inadequate performance due to inefficient algorithm design	Design	Medium	High	15
6	Communication	Poor communication among team members resulting in misunderstandings	Work Environment	Medium	Medium	9

7.	Performance	System underperforming due to inefficiencies	Design	Medium	High	15
8.	Budget	Project exceeding allocated budget or timeline	Program Constraints	High	High	25
9..	Project Scope	Scope of the project expanding beyond initial boundaries	Requirements	High	High	25

Table 4.1: Risk Analysis

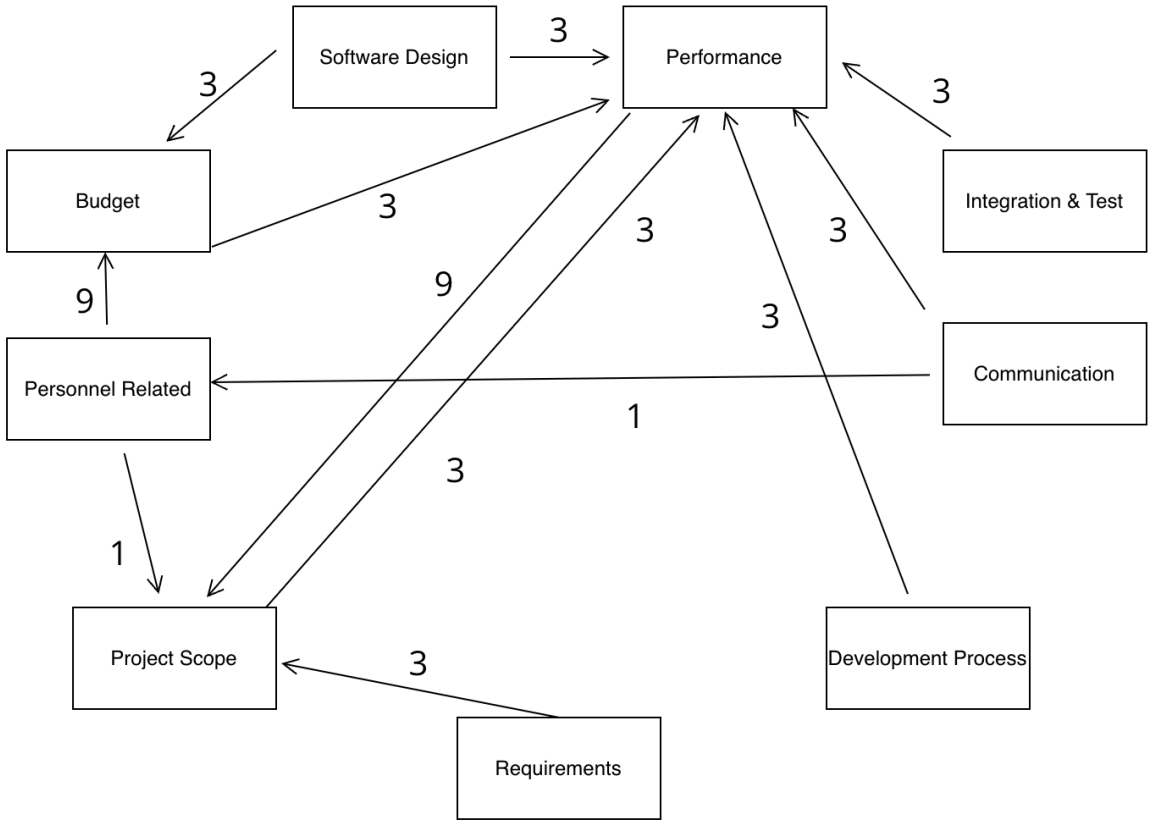

Figure 4.23: Weighted Interrelationship Graph

S.No.	Risk Area	No. of risk statements	Weights(In + Out)	Total Weight	Priority
1.	Performance	7	9+3+3+3+3+3+3	27	1
2.	Project Scope	4	1+9+3+3	16	2
3.	Budget	3	9+3+3	15	3
4.	Personnel Related	3	9+1+1	11	4
5.	Software Design	2	3+3	6	5
6.	Requirements	1	3	3	6
7.	Communication	1	3	3	7
8.	Development Process	1	3	3	8
9.	Integration & Test	1	3	3	9

Table 4.2: Risk Area Wise Total Weighting Factor

Risk Statement	Risk Area	Priority of Risk Area in IG
Risk of Suboptimal Performance Arising from Inefficient Algorithm Design	Software Design	5
Risk of Unstructured Workflow Due to Absence of Formal Development Process	Development Process	8
Risk of delayed integration of custom model due to time constraints	Project Scope	7

Table 4.3: Risks that actually occurred

Risk Response & Effectiveness

“Risk of Suboptimal Performance Arising from Inefficient Algorithm Design”

• Response: Algorithm redesign and optimization efforts were initiated.

• Effectiveness: Moderate, as it improved performance but did not fully address all inefficiencies.

“Risk of Unstructured Workflow Due to Absence of Formal Development Process”

• Response: Implementation of project in a structured manner.

• Effectiveness: High, as it brought structure and improved workflow efficiency significantly.

“Risk of delayed integration of custom model due to time constraints”

• Response: Prioritization of essential features and parallel development of integration modules.

• Effectiveness: Moderate, as it helped manage time constraints but integration still faced delays.

Top Risk (based on priority)	Mitigation Approaches
Performance	Improve algorithm efficiency through optimization and redesign.
Project Scope	Prioritize critical features to streamline project scope.
Budget	Review and adjust budget allocation to accommodate unexpected expenses.

Table 4.4: Top Risks to be mitigated

Some potential risks & their mitigation would be as follows -:

1. **Limited Dataset for Pre-training:**
    - Risk: Using only the Meditations and SQuAD datasets for pre-training may limit the model's ability to generalize to a wider range of text types and tasks.
    - Mitigation: Augment the training data with additional diverse datasets to improve the model's robustness and generalization capabilities.
2. **Integration Challenges:**
    - Risk: Difficulty in integrating the custom-trained transformer model with the web application may delay the deployment process.
    - Mitigation: Allocate sufficient time for integration and testing, and ensure compatibility between the model and the application's infrastructure.
3. **Performance Issues:**

- Risk: The deployed model may face performance issues such as slow inference times or high resource consumption.
- Mitigation: Optimize the model architecture, implement efficient inference techniques, and use hardware acceleration (e.g. GPU) and optimisers if necessary to improve performance.

4. **Error Handling and Exception Management:**
    - Risk: Inadequate error handling and exception management may lead to unexpected crashes or incorrect behavior of the application.
    - Mitigation: Implement comprehensive error handling mechanisms at critical points in the application code to gracefully handle errors and provide informative error messages to users.
5. **Security Vulnerabilities:**
    - Risk: The web application may be vulnerable to security threats such as SQL injection, cross-site scripting (XSS), or unauthorized access.
    - Mitigation: Conduct thorough security audits, implement security best practices (e.g., input validation, parameterized queries), and use encryption techniques to protect sensitive data.
6. **Dependency Management:**
    - Risk: Dependency conflicts or outdated libraries may cause compatibility issues and affect the stability of the application.
    - Mitigation: Regularly update dependencies, use virtual environments or containerization (e.g., Docker) to isolate dependencies, and maintain a record of dependencies to facilitate troubleshooting.

CHAPTER 5: TESTING

5.1 Testing Plan

Type of Test	Test Performed(Y/N)	Explanation	Component
Requirements Testing	Yes	Necessary to ensure that the software meets the specified requirements.	Entire application, including frontend and backend.
Unit	Yes	Essential for verifying the functionality of individual units or components in isolation.	Functions and methods within the application, such as answer_question.
Integration	Yes	Required to test the interaction and integration between different modules or components.	Interaction between frontend and backend components, as well as external APIs.
Performance	Yes	Important to evaluate the system's responsiveness and efficiency under normal conditions.	Application servers.
Security	Yes	Crucial to identify and mitigate potential security vulnerabilities and protect against unauthorized access or data breaches.	Entire application, including input validation, authentication, and data encryption.
Volume	Yes	Necessary to assess the system's scalability and ability to handle large volumes of data or transactions.	Database servers and storage.
Custom	Yes	Custom testing based on specific project	Specific components or features identified

		requirements or unique functionalities.	during development, including pre-training of custom transformers, MLM, and NSP objectives.

Table 5.1 : Testing Plan

5.2 Component decomposition and type of testing required

Based on the components involved in the web application, we can decompose the system into the following components:

1. Frontend:
    - Responsible for user interaction and presentation of the application.
    - Includes HTML, CSS, and JavaScript code for the user interface.
    - Requires requirements testing, integration testing with backend APIs, and security testing for input validation and XSS prevention.
2. Backend API:
    - Handles HTTP requests from the frontend and performs question answering tasks.
    - Includes Flask routes and functions for processing requests and returning responses.
    - Requires unit testing for individual functions, integration testing with the model loading and prediction components, and security testing for input validation and authorization.
3. Model Loading and Prediction:
    - Loads transformer models and tokenizers based on user input.
    - Performs question answering tasks using the loaded model.
    - Requires unit testing for model loading and prediction functions, integration testing with the backend API, and performance testing to measure inference speed.
4. Custom Transformer Pre-training:
    - Involves pre-training custom transformer models using MLM and NSP tasks.
    - Includes training logic, data preprocessing, and model evaluation components.

- Requires unit testing for training and evaluation functions, integration testing with the model loading component, and performance testing to measure training time and model performance.

5. MLM and NSP Training:
    - Involves training MLM and NSP tasks separately.
    - Includes training logic, data preprocessing, and evaluation components for each task.
    - Requires unit testing for training and evaluation functions, integration testing with the model loading component, and performance testing to measure training time and task performance.

Based on this decomposition, the following types of testing are required:

1. Requirements Testing:
    - Frontend
    - Backend API
2. Unit Testing:
    - Frontend (for individual components if applicable)
    - Backend API
    - Model Loading and Prediction
    - Custom Transformer Pre-training
    - MLM and NSP Training
3. Integration Testing:
    - Frontend with Backend API
    - Backend API with Model Loading and Prediction
    - Model Loading and Prediction with Custom Transformer Pre-training
    - Custom Transformer Pre-training with MLM and NSP Training
4. Performance Testing:
    - Backend API
    - Model Loading and Prediction
    - Custom Transformer Pre-training
    - MLM and NSP Training

5. Security Testing:
    - Frontend
    - Backend API
6. Volume Testing:
    - Backend API
    - Model Loading and Prediction
    - Custom Transformer Pre-training
    - MLM and NSP Training
7. Custom Testing:
    - Frontend
    - Backend API
    - Model Loading and Prediction
    - Custom Transformer Pre-training
    - MLM and NSP Training

S.No	**List of Various Components (modules) that require testing**	**Type of Testing Required**	**Technique for writing test cases**
1.	Frontend	Requirements, Integration, Security	Black Box, White Box
2.	Backend API	Requirements, Unit, Integration, Security	Black Box, White Box
3.	Model Loading	Unit, Integration, Performance	Black Box, White Box
4.	Custom Transformer Pre-Training	Unit, Integration, Performance	Black Box, White Box
5.	MLM & NSP objectives	Unit, Integration, Performance	Black Box, White Box

Table 5.2: Component Decomposition and Identification of Tests required

5.3 List all test cases in prescribed format

Test Case ID	Input	Expected Outcome	Status
TC-FE-01	Valid input question and reference	Response containing the expected answer	Pass
TC-FE-02	Empty question field	Response indicating missing question	Pass
TC-FE-03	Empty reference field	Response indicating missing reference	Pass
TC-FE-04	Different model name	Response with updated model and successful model loading	Pass

Table 5.3: Test Cases for Component 1

Test Case ID	Input	Expected Outcome	Status
TC-BA-01	Valid HTTP request with question and reference	Response containing the expected answer	Pass
TC-BA-02	HTTP request with empty question field	Response indicating missing question	Pass
TC-BA-03	HTTP request with empty reference field	Response indicating missing reference	Pass
TC-BA-04	HTTP request with different model name	Response with updated model and successful model loading	Pass

Table 5.4: Test Cases for Component 2

Test Case ID	Input	Expected Outcome	Status
TC-MLP-01	Model loading function call	Loaded model and tokenizer objects	Pass
TC-MLP-02	Valid input question and reference	Response containing the expected answer	Pass

TC-MLP-03	Invalid input causing exception	Error response indicating failure to load model	Pass
TC-MLP-04	Performance testing with large input	Response time within acceptable limits	Pass

Table 5.5: Test Cases for Component 3

Test Case ID	Input	Expected Outcome	Status
TC-CTP-01	Training data input	Trained custom transformer model	Pass
TC-CTP-02	Evaluation data input	Evaluation metrics (e.g., accuracy, loss)	Pass
TC-CTP-03	Performance testing with large dataset	Training time within acceptable limits	Pass

Table 5.6: Test Cases for Component 4

Test Case ID	Input	Expected Outcome	Status
TC-MNT-01	MLM training data input	Trained MLM model	Pass
TC-MNT-02	NSP training data input	Trained NSP model	Pass
TC-MNT-03	Evaluation data input	Evaluation metrics (e.g., accuracy, loss)	Pass
TC-MNT-04	Performance testing with large dataset	Training time within acceptable limits	Pass

Table 5.7: Test Cases for Component 5

5.4 Error and Exception Handling

Relatively basic error and exception handling has been used but covers various scenarios.

1. Form Data Parsing: When handling form submissions in the Flask route, the code checks if the required fields (question and reference) are present in the form data. If any of these

fields are missing, the application returns an error response with an appropriate message indicating that the required fields are missing.

```
# check if http request method is POST, if true means form has been submitted with data
if request.method == 'POST':
    # Parse input
    # This line extracts the form data submitted with the POST request.
    # In Flask, request.form is a dictionary-like object containing the key-value pairs of form data submitted.
    data = request.form
    # retrieve the value corresponding to the key 'question' from the data dictionary
    question = data['question']
    # retrieve the value corresponding to the key 'reference' from the data dictionary
    reference = data['reference']
    model_name = data.get('model_name', DEFAULT_MODEL_NAME)
```

Figure 5.1: Form Data Parsing

```
# Get the answer to the question
answer = answer_question(question, reference, app.model, app.tokenizer)
answer = answer if answer else 'I do not know the answer to that question 😢'
```

Figure 5.2: Error Message

2. Model Loading: When changing the model used for question answering, the code attempts to load the specified model using the Hugging Face Transformers library. If the specified model name is invalid or if there are any errors during model loading, the application returns an error response with a message indicating the issue.

```
# To avoid loading the model and tokenizer every time, we only do it if the model name has changed
# This condition checks whether the requested model name (model_name) is different from the last use
if model_name != app.last_used_model_name:
    # Load the new model and tokenizer
    app.model = AutoModelForQuestionAnswering.from_pretrained(model_name)
    app.tokenizer = AutoTokenizer.from_pretrained(model_name)
    app.last_used_model_name = model_name
except Exception as e:
    return jsonify({'error': f'Error loading model: {str(e)}'}), 500
```

Figure 5.3: Model Loading Error Handling

3. Tokenization and Inference: During tokenization and inference to generate the answer to the question, there is no explicit error handling. However, if there are any issues during tokenization or inference, the Hugging Face Transformers library may raise exceptions

which would propagate to the calling code. These exceptions would be caught by the Flask application's error handling mechanism and returned as part of the HTTP response.

5.5 Limitations of the solution

- Limited Dataset:
    - The pre-training of the transformer model was performed using only the Meditations dataset and the SQuAD dataset. Using a small dataset for pre-training may result in limited coverage of language patterns and concepts.
    - This can potentially lead to suboptimal performance, especially for questions and contexts outside the scope of these datasets.
- Model Integration:
    - As of now, the custom-trained transformer model has not been integrated into the web application.
    - This limits the application's functionality to only using the pre-trained BERT model. Integration of custom models requires additional development effort and testing to ensure compatibility and performance.
- Model Selection:
    - The web application allows users to select the transformer model to use, but there is no guidance or recommendation provided regarding which model might be most suitable for a given task or context. Users may not be familiar with the available models or their capabilities, leading to suboptimal model selection.
- Dependency on External Services:
    - The application relies on external services such as the Hugging Face model hub for downloading transformer models. Any issues with these services, such as downtime or changes in model availability, could affect the application's functionality.
- Performance:
    - The performance of the application may vary depending on factors such as the complexity of the question-answer task. Response times may be slower for more complex questions or when using larger transformer models.
- Scalability:

    - The current implementation of the application may not be easily scalable to handle a large number of concurrent users or heavy traffic. Scaling the application would require additional infrastructure setup and optimization.
- Model Fine-tuning:
    - The pre-trained BERT model used in the application is fine-tuned on the SQuAD dataset, which is primarily focused on question-answering tasks.
    - Fine-tuning on a broader range of datasets or domains may improve the model's performance for specific use cases.

CHAPTER 6: FINDINGS, CONCLUSION & FUTURE WORK

6.1 Findings

1. Successful Implementation of Web Application:
    - The project has resulted in the successful development of a web-based question-answering application.
    - The application allows users to input questions and reference text and provides answers based on pre-trained transformer models.
2. Integration with BERT Model:
    - The application is currently integrated with the BERT model and provides accurate answers for a wide range of questions.
3. Model Loading and Switching Functionality:
    - The application allows users to switch between different pre-trained transformer models by specifying the model name in the input form.
    - Model loading and switching functionalities are implemented to ensure seamless integration with various transformer models.
4. Error and Exception Handling:
    - Error and exception handling mechanisms are implemented to provide meaningful error messages to users in case of invalid inputs or model loading failures.
    - Appropriate error responses are returned to users with details of the encountered errors.
5. Effective Tokenization:
    - BPemb tokenizer proved to be effective for tokenizing text in multiple languages, including languages with complex morphological structures.
    - BPemb tokenizer utilizes subword tokenization, which allows it to handle out-of-vocabulary tokens by breaking down words into smaller subword units.
6. Optimal Balance of Masked Tokens:
    - Through experimentation and analysis, it was found that masking approximately 15% of tokens in the input text provided a balance between preserving contextual

information and promoting robustness in the model's ability to predict masked tokens.

- Masking too few tokens might lead to insufficient exposure to masked language modeling objectives, while masking too many tokens could disrupt the natural flow of text and hinder learning.

7. Use of Optimizers:
    - AdamW optimizer was chosen for its ability to handle large-scale transformer models efficiently.
    - AdamW's adaptive learning rate and weight decay helped in stabilizing and accelerating the training process for transformer-based architectures.
8. Overfitting Considerations:
    - Pre-training on small datasets may lead to overfitting on specific patterns present in the training data.
    - Regularization techniques such as dropout, weight decay, and early stopping are essential to prevent overfitting and ensure generalization to unseen data.
9. Custom Transformer Pre-training:
    - The custom transformer model shows promising results in capturing domain-specific language patterns and knowledge from the training data.
    - Fine-tuning on specific tasks may further improve the model's performance in domain-specific question answering tasks.

6.2 Conclusion

The project has been an enriching journey into the realm of transformer models and their applications in natural language processing (NLP). While significant progress has been made in understanding and implementing various components of the transformer architecture, including attention mechanisms, tokenization, embeddings, and training objectives like Masked Language Model (MLM) and Next Sentence Prediction (NSP), there are still areas where further exploration and development are needed.

Despite the challenges encountered in fully implementing and deploying a custom transformer model from scratch, valuable insights have been gained. The comparison between custom tokenization and embeddings with those of pre-trained models like BERT provided valuable insights into their efficacy and performance. Additionally, the experience of working with Hugging Face's pre-trained transformers and pipelines for tasks like text classification, summarization, and question answering has enhanced understanding and proficiency in leveraging existing NLP tools and resources.

Moving forward, efforts will continue towards refining and completing the implementation of the custom transformer model, with a focus on addressing the challenges encountered. Further experimentation and exploration are planned to optimize the model's performance and extend its capabilities. Additionally, the development and deployment of the web application for NLP tasks will be pursued with a continued emphasis on user-friendly interfaces and seamless integration with the transformer model.

Overall, while the project has encountered some obstacles, it has been a valuable learning experience, laying the foundation for future research and development in the field of NLP and transformer-based models.

6.3 Future Work

The project has promising avenues for future exploration and development. Some of the key areas for future scope include :-

1. **Completion of Custom Transformer Model:** Efforts will continue towards completing and refining the implementation of the custom transformer model from scratch. This includes addressing any existing challenges and optimizing the model's architecture and training procedures.

2. **Integration with Web Application:** The ultimate goal of integrating the custom transformer model with the web application for NLP tasks remains a priority. This will involve incorporating the model's functionality into the application and ensuring seamless performance and user experience.

3. **Performance Comparison:** Once the custom transformer model is fully functional, it will be important to conduct a comprehensive performance comparison with existing pre-trained transformers like BERT. This will provide valuable insights into the effectiveness and efficiency of the custom model compared to established benchmarks.

4. **Exploration of Additional NLP Tasks:** The project can be extended to explore a wider range of NLP tasks beyond the initial scope. Tasks such as sentiment analysis, named entity recognition, and text generation could be implemented and evaluated using the custom transformer model.

5. **Fine-tuning and Optimization:** Further experimentation and optimization of the custom transformer model will be undertaken to enhance its performance across different tasks and datasets. This includes fine-tuning hyperparameters, exploring different training strategies, and incorporating additional training data.

6. **User Feedback and Iterative Improvement:** Continuous feedback from users of the

web application will be solicited and used to iteratively improve the functionality and usability of the custom transformer model. This will ensure that the model meets the needs and expectations of its intended users.

By pursuing these avenues, the project aims to contribute to advancements in transformer-based NLP models and their practical utility in various domains.

REFERENCES

[1] Y. Liu, H. Huang, J. Gao and S. Gai, "A study of Chinese Text Classification based on a new type of BERT pre-training," 2023 5th International Conference on Natural Language Processing (ICNLP), Guangzhou, China, 2023, pp. 303-307, doi: 10.1109/ICNLP58431.2023.00062.

[2] H. Wang and M. Tu, "Enhancing Attention Models via Multi-head Collaboration," 2020 International Conference on Asian Language Processing (IALP), Kuala Lumpur, Malaysia, 2020, pp. 19-23, doi: 10.1109/IALP51396.2020.9310460.

[3] J. Wu, X. Huang, J. Liu, Y. Huo, G. Yuan and R. Zhang, "NLP Research Based on Transformer Model," 2023 IEEE 10th International Conference on Cyber Security and Cloud Computing (CSCloud)/2023 IEEE 9th International Conference on Edge Computing and Scalable Cloud (EdgeCom), Xiangtan, Hunan, China, 2023, pp. 343-348, doi: 10.1109/CSCloud-EdgeCom58631.2023.00065.

[4] Devlin, Jacob et al. “BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding.” North American Chapter of the Association for Computational Linguistics (2019).

[5] Denis Rothman, Transformers for Natural Language Processing: Build innovative deep neural network architectures for NLP with Python, PyTorch, TensorFlow, BERT, RoBERTa, and more , Packt Publishing, 2021.

[6] Salazar, Julian et al. “Masked Language Model Scoring.” Annual Meeting of the Association for Computational Linguistics (2019).

[7] A. F. Adoma, N. -M. Henry and W. Chen, "Comparative Analyses of Bert, Roberta, Distilbert, and Xlnet for Text-Based Emotion Recognition," 2020 17th International Computer Conference on Wavelet Active Media Technology and Information Processing (ICCWAMTIP), Chengdu, China, 2020, pp. 117-121, doi: 10.1109/ICCWAMTIP51612.2020.9317379.

[8] A. Joshy and S. Sundar, "Analyzing the Performance of Sentiment Analysis using BERT, DistilBERT, and RoBERTa," 2022 IEEE International Power and Renewable Energy Conference (IPRECON), Kollam, India, 2022, pp. 1-6, doi: 10.1109/IPRECON55716.2022.10059542

[9] J. von der Mosel, A. Trautsch and S. Herbold, "On the Validity of Pre-Trained Transformers for Natural Language Processing in the Software Engineering Domain" in IEEE Transactions on Software Engineering, vol. 49, no. 04, pp. 1487-1507, 2023.

[10] Kotei, E.; Thirunavukarasu, R. A Systematic Review of Transformer-Based Pre-Trained Language Models through Self-Supervised Learning. Information 2023,14,187. https://doi.org/10.3390/info14030187

[11] T. Wolf et al., "Huggingface's Transformers: State-of-the-art Natural Language Processing," ArXiv preprint arXiv:1910.03771, 2019.

[12] Masked Language Model, https://www.sbert.net/examples/unsupervised_learning/MLM [Online Image]

[13] Next Sentence Prediction, Yan Ding (2021), https://www.shuffleai.blog/blog/gap-sentences-generation-in-pegasus.html [Online Image]

[14] Silvia Casola, Ivano Lauriola, Alberto Lavelli, Pre-trained transformers: an empirical comparison, Machine Learning with Applications, Volume 9, 2022, 100334, ISSN 2666-8270,

[15] Kotei, Evans and Ramkumar Thirunavukarasu. “A Systematic Review of Transformer-Based PreTrained Language Models through Self-Supervised Learning.” Inf. 14 (2023): 187.

[16] BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding,Jacob Devlin and Ming-Wei Chang and Kenton Lee and Kristina Toutanova}, 2019.

[17] Gardner, Matt, Joel Grus, Mark Neumann, Oyvind Tafjord, Pradeep Dasigi, Nelson F. Liu, Matthew E. Peters, Michael Schmitz and Luke Zettlemoyer. “AllenNLP: A Deep Semantic Natural Language Processing Platform.” ArXiv abs/1803.07640 (2018): n. pag.

[18] Camacho-Collados, José, Kiamehr Rezaee, Talayeh Riahi, Asahi Ushio, Daniel Loureiro, Dimosthenis Antypas, Joanne Boisson, Luis Espinosa-Anke, Fangyu Liu, Eugenio Martínez-Cámara, Gonzalo Medina, Thomas Buhrmann, Leonardo Neves and Francesco Barbieri. “TweetNLP: CuttingEdge

Natural Language Processing for Social Media.” Conference on Empirical Methods in Natural Language Processing (2022).

[19] XLNet: Generalized Autoregressive Pretraining for Language Understanding, Zhilin Yang and Zihang Dai and Yiming Yang and Jaime Carbonell and Ruslan Salakhutdinov and Quoc V. Le, 2020.

[20] ELECTRA: Pre-training Text Encoders as Discriminators Rather Than Generators, Kevin Clark and Minh-Thang Luong and Quoc V. Le and Christopher D. Manning, 2020.

[21] Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer, Colin Raffel and Noam Shazeer and Adam Roberts and Katherine Lee and Sharan Narang and Michael Matena and Yanqi Zhou and Wei Li and Peter J. Liu, 2023.

[22] RoBERTa: A Robustly Optimized BERT Pretraining Approach, Yinhan Liu and Myle Ott and Naman Goyal and Jingfei Du and Mandar Joshi and Danqi Chen and Omer Levy and Mike Lewis and Luke Zettlemoyer and Veselin Stoyanov, 201

[23] ALBERT: A Lite BERT for Self-supervised Learning of Language Representations, Zhenzhong Lan and Mingda Chen and Sebastian Goodman and Kevin Gimpel and Piyush Sharma.