UNIVERSITÀ DEGLI STUDI DI ROMA TOR VERGATA

MACROAREA DI INGEGNERIA

CORSO DI STUDIO IN

Mechatronics Engineering

TESI DI LAUREA IN

Electronics of IOT and Embedded Systems

TITOLO

Analysis of Large Language Models – Tokenization, Embedding and Fine Tuning.

Relatore:

Prof. Giancarlo Cardarilli

Laureando:

Antariksh Milind

Correlatore:

Prof. Sergio Spano

Anno Accademico 2023/24

ACKNOWLEDGEMENTS

I would like to dedicate my work to the most beloved people in my life, my parents, Jyoti and Milind Shreshthi. No words of gratitude can truly do justice to their love, care, support, sacrifice, and blessings – I owe everything to them. I AM BECAUSE THEY ARE!

I would also like to thank my professors, Prof. Giancarlo Cardarilli and Prof. Sergio Spano for their valuable guidance and support throughout my thesis work. It has been a pleasure working with them.

To my dear sister, Saachi, who loves me unconditionally – I love you to the moon and back!

ABSTRACT

Large Language Models (LLMs) are a class of transformer-based artificial intelligence models designed to process and generate human-like text. With the rise of models like GPT, BERT, and their variants, LLMs have significantly transformed various fields, particularly natural language processing (NLP), and have become integral to many applications of machine learning and artificial intelligence. These models are pivotal in tasks such as sentiment analysis, translation, text summarization, and more, reshaping how we interact with technology in both personal and professional contexts.

This thesis explores the foundational concepts and practical implementations of LLMs, focusing on three key components: tokenization, embeddings, and fine-tuning. It begins with an overview of the transformer model architecture and its various applications. The thesis then delves into the processes and types of tokenization and embeddings, followed by the implementation of tokenization and embedding algorithms in code. The final chapter includes fine-tuning a smaller LLM, improving its performance to match that of an industry scale LLM. The chapter demonstrates how an effectively fine-tuned smaller model can replace larger models like GPT for various NLP tasks, while maintaining high performance and reducing operational costs.

Table of Contents

1. THE TRANSFORMER – MODEL OVERVIEW AND APPLICATIONS

Introduction

Natural language processing involves algorithms that help us comprehend, manipulate and produce human language. For such natural language tasks, large language models demonstrate exceptional performance and flexibility owing to the knowledge they learn in the pre-training. Large language models or "LLMs" are neural networks that utilize the transformer architecture. This architecture will be further studied as the chapter progresses.

1.1 Transformer architecture

The transformer is the standard architecture for building large language models. It includes an encoder which takes in the input sequence and passes it on to the decoder through its feedforward network. The input sequence is first tokenized using a tokenizer followed by the conversion of these tokens into vectors using embeddings [1]. The final input embedding is the sum of token embedding and positional embedding. The various parts of the transformer model can be seen in the figure below.

The transformer feeds these input embeddings into a layer known as the multi-head attention. This layer is made up of multiple self-attention layers. The process of assigning weights (parameters) to every word of the input relative to every other word is carried out in the self-attention layer. This reveals the relevance of the word in the current context [2]. The parameters of every self-attention layer are based on different factors, such as, people's relationship, performed activities, words that rhyme, etc. Depending on the factors considered, the self-attention layers are able to establish which words in the input sequence have a closer relationship to one another. For example, in the sentence "The cat was sleeping", the weight between 'The' and 'cat' will have more importance, since the article 'The' is referring to the noun 'cat' and not 'was'.

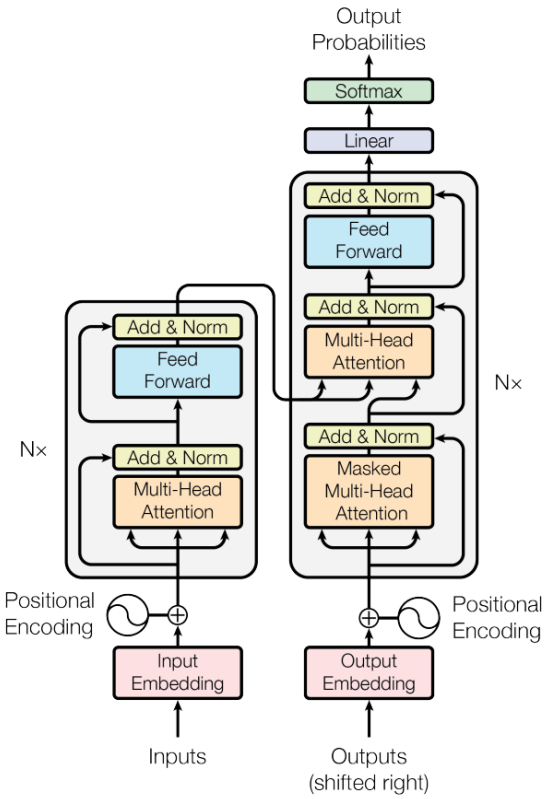

FIGURE 1.1 - TRANSFORMER ARCHITECTURE [1]

The feed forward part predicts the next word by trying to assign a score for each word. The weights from the previous step are taken as input and processed. Based on the training data, it tries to predict what the next word would be (it goes through all words in its vocabulary). Let's say the model is trying to predict the next word in the sentence "The Kangaroo...". It would ideally assign a very high score to 'jumped', maybe a medium score to 'slept', and definitely a low score to 'swam'. The output of this is scores (logits) and not probabilities.

These scores are then converted into probabilities using the Softmax layer. A mathematical function 'softmax' is used for the conversion such that all of the probabilities add up to 1. Eventually, the word with the highest probability is chosen as the next word. The subsequent sections will look into tokenization and embedding in more detail.

1. 2 Tokenization –

In LLMs, tokens are not words but a smaller unit, such as, a character or a part of a word, or even an entire phrase. The size of a token depends entirely on the algorithm used. Since the transformer cannot read words and sentences, they must be first converted into tokens. There is a de-tokenization step at the output of a transformer so that we can interpret the output produced.

The process of tokenization first involves establishing the vocabulary (pre-training). Large text data is gathered from which the model learns and after applying preliminary tokenization methods, the text is split into words, subwords and characters. To generate a set of tokens, a tokenization algorithm such as Byte Pair Encoding (BPE), WordPiece or SentencePiece is used. The algorithm is run and a set of subword tokens or characters is created. Each such token in the established vocabulary is assigned an integer ID. During real time processing, the text is converted into tokens based on the vocabulary and the tokens are mapped to its respective ID.

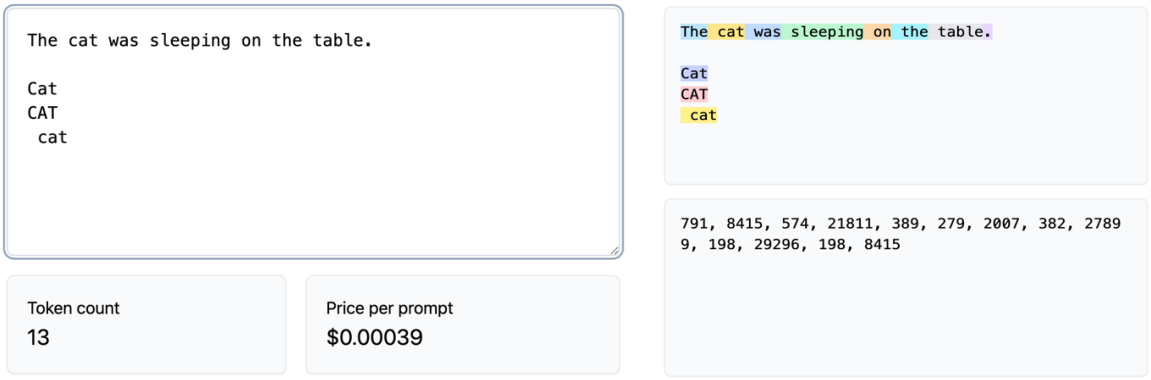

FIGURE 1.2A – TOKENIZATION (GPT-4) [3]

In the above example, gpt-4's tokenization method (BPE) is demonstrated. Observe how each word is assigned a different token. Also, tokenization is case sensitive. The word cat, depending on where it appears in the sentence, whether it has a capital C, or whether all the letters of the word are capital, it has been assigned a unique token.

The Byte Pair Encoding method is the most widely used tokenization method in the transformer models. This method starts off by assigning each character a single token, followed by merging the most frequently appearing adjacent pairs with a two-character long token and so on. In the following example, the byte pair "aa" occurs more than once and hence we can replace it with a byte that is not used in the data ("Z"). We then replace the pair "ab" with "Y". Now the two new bytes are also forming a pair and are appearing twice, and hence, they can be replaced with a new byte "X". This is known as recursive byte pair encoding.

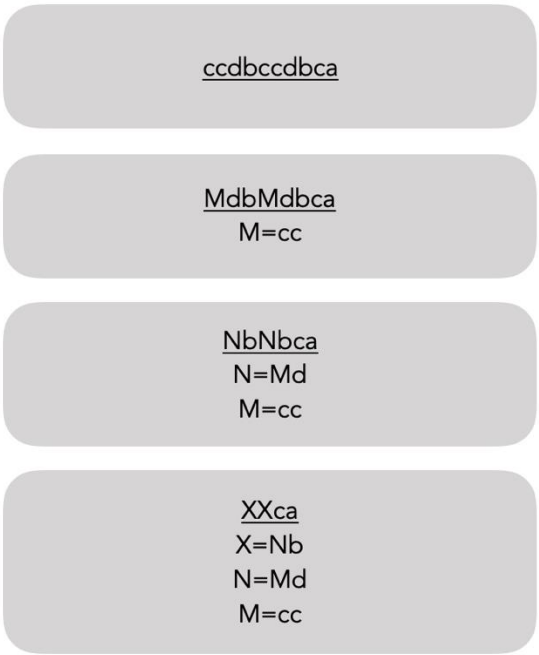

FIGURE 1.2B – BYTE PAIR ENCODING

The data cannot be compressed further since no pair of bytes are occurring more than once. In order to decompress the data, we can the replacements can be performed in reverse order.

1.3 Embeddings –

The subsequent step after tokenization is token embedding. Token embeddings is a high dimensional space where the tokens from the previous step are mapped to a unique position called a vector. Not just text, but also objects like images and audio can be represented using embeddings. Such objects are translated to a mathematical form depending upon the category they belong to and the factors of traits that they may or may not have. Embeddings are used to find similarities between objects and a machine learning model can find a similar image or a document using embeddings. For example, look at following image which is representation of a two-dimensional space.

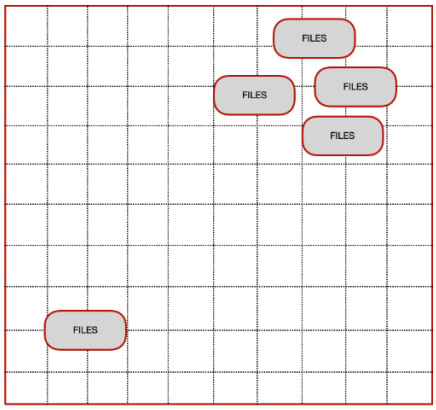

FIGURE 1.3A - EMBEDDING (2-D EXAMPLE)

The files in the top right corner are more relevant to each other and are hence placed closer. The file in the bottom left is not similar to any of the other documents and is therefore, placed far apart. Look at the following three-dimensional space to further understand this concept.

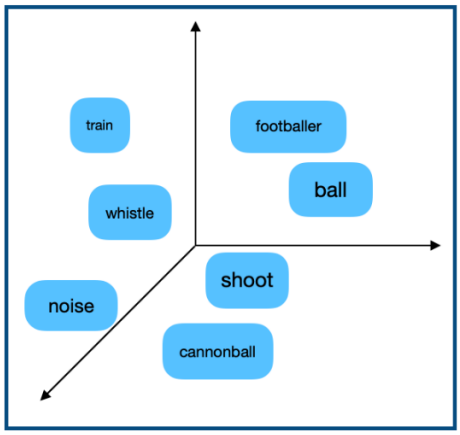

FIGURE 1.3B - EMBEDDING (3-D EXAMPLE)

The words “footballer” and “ball” are placed very close to each other compared to the words “train” and “whistle”. Meanwhile the word “noise” is close to “whistle” and “cannonball”. Similarly, the word “whistle” sits between “noise” and “train”.

These were examples of only two and three-dimensional embeddings, but the number of dimensions could be in the ranges of thousands as in the case of gpt models [4]. Each number in the vector indicates where the object is along that direction. For example, the city of Calgary, Canada can be represented with the longitudinal coordinates {51.0447° N, 114.0719° W}. This is a simple vector with two numbers. If the model wanted to find a city close to Calgary, it will just look for cities with similar coordinates to that of Calgary and conclude that the city of Red Deer (closest city to Calgary) is the closest. Now suppose we wanted to find a city that was not only close but also as big (population wise) as Calgary. For this task, we will have to include another dimension that represents the population of the cities. The vector of Calgary will now become {51.0447° N, 114.0719° W, 1600000}. This extra dimension will allow the model to not only find cities that are closer to each other (based on the coordinates) but also cities with similar population size (the third dimension). The model will conclude that the city of Edmonton, Canada with vector {53.5461° N, 113.4937° W, 1200000} is close and of similar size. In this way more and more dimensions can be added to find the similarities between two or more things based on certain parameters.

Movie	Origin	Year	Genre	Duration
Back to The Future	US	1985	Sci-fi	116
ZNMD	India	2011	Drama	153

FIGURE 1.3C - EMBEDDINGS (4-D EXAMPLE)

In the above example, the vector for the Movie “Back to The Future” would look like {[US], 1985, [Sci-fi], 116} and for “ZNMD” it would look like {[India], 2011, [Drama], 153}. In none of the dimensions are these moives similar and hence when the model is asked for movies similar to “Back to The Future”, it will definitely not recommend “ZNMD” but will recommend something like “The Terminator” with the vector {[US], 1984, [Sci-fi], 107}.

After we have created our token embeddings, we can use these embeddings to calculate the position of every token in the sequence of tokens. This process is known as positional embedding. In this step, a vector is created for every single token. This vector represents the position of each word in a prompt relative to every other word and which helps in understanding the context.

Word	Token Embedding	Positional Embedding	Final Embedding (Input + Positional)
whistle	〔0.75 1.5 -3.2 0.11〕	〔0.58 1.96 0.73 5.2〕	〔0.75+0.58 1.5+1.96 ⋯〕
noise	〔0.83 1.7 -3.1 0.14〕	〔0.54 1.98 0.69 5.4〕	〔0.83+0.54 1.7+1.98 ⋯〕

FIGURE 1.3D – POSITIONAL EMBEDDING

The final input embedding, which moves on into the attention layer, is a sum of token embedding and positional embedding.

1.4 Transformer Applications -

Transformer models have found their applications in solving all kinds of natural language processing tasks. However, they are not restricted to NLPs. Transformers are used in computer vision for image classification and object detection. Transformer have also achieved proficiency in in predicting protein folding structures. Apart from these, speech processing, code generation and recommender systems are some of the many fields in which transformers are being used extensively. Let's take a look at some important NLP tasks that transformers are able to perform. For the following examples, we will be using the `pipeline()` function from the transformers library.

Sentiment Analysis – It is used to analyze text in order to determine whether the emotional tone of the message is positive, negative or neutral [5].

```
from transformers import pipeline

classifier = pipeline("sentiment-analysis")
classifier("I've been waiting for a HuggingFace course my whole life.")
```

```
[{'label': 'POSITIVE', 'score': 0.9598047137260437}]
```

FIGURE 1.4A - SENTIMENT ANALYSIS [6]

A model is selected by the pipeline that has been fine tuned for sentiment analysis. The model assigned a very high score for the label 'positive' and hence the emotional tone of this particular text is positive according to the model.

Text Generation – After we provide a prompt, which could be an incomplete sentence, the model auto-completes the sentence.

```
from transformers import pipeline

generator = pipeline("text-generation")
generator("In this course, we will teach you how to")
```

```
ed_text': 'In this course, we will teach you how to understand and use '
          'data flow and data interchange when handling user data. We '
          'will be working with one or more of the most commonly used '
          'data flows — data flows of various types, as seen by the '
          'HTTP'}]
```

FIGURE 1.4B – TEXT GENERATION.1 [6]

If we are not happy with the result, we can also choose a model of our choice as follows.

```
from transformers import pipeline

generator = pipeline("text-generation", model="distilgpt2")
generator(
    "In this course, we will teach you how to",
    max_length=30,
    num_return_sequences=2,
)
```

FIGURE 1.4C – TEXT GENERATION.2 [6]

Translation – One of the most used feature/application of transformer models is translation. Text from any language can be translated to any target language. An appropriate model has to be chosen in order to facilitate the translation. In the figure below, the Helsinki model translates French to English.

```
from transformers import pipeline

translator = pipeline("translation", model="Helsinki-NLP/opus-mt-fr-en")
translator("Ce cours est produit par Hugging Face.")
```

```
[{'translation_text': 'This course is produced by Hugging Face.'}]
```

FIGURE 1.4D – TRANSLATION [6]

An appropriate model has to be chosen in order to facilitate the translation. In the figure above, the Helsinki model translates French to English.

Along with the above-mentioned NLP tasks, transformers are also capable of many other tasks such as question-answering, summarization, feature-extraction, fill-mask, NER and zero-shot-classification.

2. EMBEDDINGS – EXPLANATION AND TYPES

Introduction –

As we have seen in the previous chapter, embeddings are an important component of the transformer model. They enable the model to establish relationships between tokens and thus help in the prediction of the next word in the output sequence. In this chapter we will look at the embeddings in more details while highlighting the different types and their functions. Two of the most important once that we will focus on are token embeddings and positional embeddings.

2.1 Token Embeddings –

These capture the meaning and characteristics of each word and establish its relationship with other words. They can be further classified as word, sub-word or character embeddings. Word embeddings map individual words to vectors whereas sub-word embeddings are used when the text is tokenized into sub-words (breaking each word into smaller parts). Byte-Pair encoding and SentencePiece are examples of sub-word embeddings. Pre-trained embeddings such as Word2vec, GloVe and FastText belong to the group of word embeddings. When fine linguistic details are to be captured, character embeddings are used which are a character level representation of the text.

2.2 Positional Embeddings –

Without positional embeddings, the transformer is unaware of the order of the tokens. Positional embeddings help the transformer understand the order/position of tokens in a sequence. The transformer model works on each token individually and concurrently, which makes it completely unaware of the order and therefore these embeddings become important as they encode the position of each token in the sequence.

a) Absolute Sinusoidal Positional Embeddings –

A popular method to encode such positional information is sinusoidal positional embeddings.

As discussed in Vaswani et. al [1], the sinusoidal positional embeddings are given by,

$$PE(pos, 2i)=\sin\left(\frac{pos}{10000^{\frac{2i}{d}}}\right)$$

$$PE(pos, 2i+1)=\cos\left(\frac{pos}{10000^{\frac{2i}{d}}}\right)$$

where:

- *pos* is the token's position in the sequence
- *i* represents the index of a specific dimension within the embedding vector
- *d* represents the total number of dimensions for each token in the embedding vector
-

A model is thus able to differentiate between tokens based on their position in the input sequence as these sinusoidal embeddings create a positional embedding for each position in the sequence. Look at the image below to understand this phenomenon.

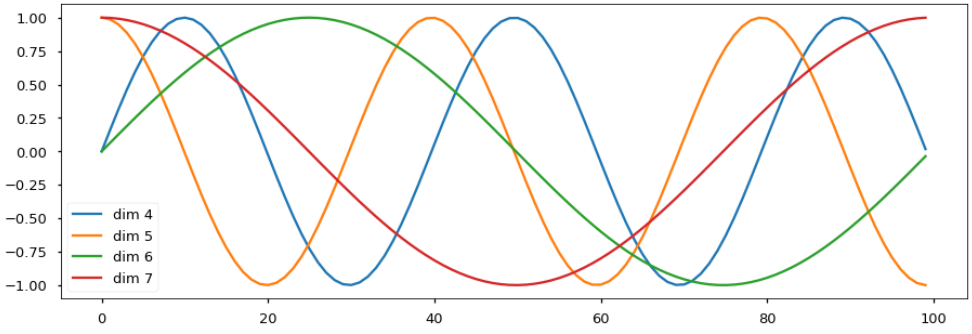

FIGURE 2 - SINUSOIDAL POSITIONAL EMBEDDINGS [11]

Various dimensions are represented by the waves in the image above. The x-axis represents the positions whereas the y axis denotes the corresponding embedding for that position. Suppose, for an input sequence with embedding vectors of dimension 4, the token at position 4 would have an embedding of around 0.25, whereas, a token at position 10 will have a embedding of 1.00. In this way, every position of an input sequence is assigned a unique embedding vector. These embeddings are then added to the token embeddings to get the final embeddings. Let us look at an example to better understand the process. Consider an input sequence "The car drove". After tokenization, we get three embedding vectors for each of the three tokens ['The', 'car', 'drove'] with a dimension d of 4 as follows,

1) 'The' (Ethe): [0.1, 0.2, 0.3, 0.4]
2) 'car' (Ecar): [0.5, 0.6, 0.7, 0.8]
3) 'drove' (Edrove): [0.9, 1.0, 1.1, 1.2]

Computing the positional embeddings involves the following steps,

For pos 0:

$$PE(0,0) = \sin\left(\frac{0}{10000^{\frac{0}{4}}}\right) = \sin(0) = 0$$

$$PE(0,1) = \cos\left(\frac{0}{10000^{\frac{0}{4}}}\right) = \cos(0) = 1$$

$$PE(0,2) = \sin\left(\frac{0}{10000^{\frac{2}{4}}}\right) = \sin(0) = 0$$

$$PE(0,3) = \cos\left(\frac{0}{10000^{\frac{2}{4}}}\right) = \cos(0) = 1$$

Therefore, for pos 0, the positional embedding is [0, 1, 0, 1]

For *pos* 1:

$$PE(1,0) = \sin\left(\frac{1}{10000^{\frac{0}{4}}}\right) = \sin(1) = 0.8415$$

$$PE(1,1) = \cos\left(\frac{1}{10000^{\frac{0}{4}}}\right) = \cos(1) = 0.5403$$

$$PE(1,2) = \sin\left(\frac{1}{10000^{\frac{2}{4}}}\right) = \sin(0.01) = 0.01$$

$$PE(1,3) = \cos\left(\frac{1}{10000^{\frac{2}{4}}}\right) = \cos(0.01) = 0.99995$$

Therefore, for *pos* 1, the positional embedding is [0.8415, 0.5403, 0.01, 0.99995]

For *pos* 2:

$$PE(2,0) = \sin\left(\frac{2}{10000^{\frac{0}{4}}}\right) = \sin(2) = 0.9093$$

$$PE(2,1) = \cos\left(\frac{2}{10000^{\frac{0}{4}}}\right) = \cos(2) = -0.4161$$

$$PE(2,2) = \sin\left(\frac{2}{10000^{\frac{2}{4}}}\right) = \sin(0.02) = 0.02$$

$$PE(2,3) = \cos\left(\frac{2}{10000^{\frac{2}{4}}}\right) = \cos(0.02) = 0.9998$$

Therefore, for *pos* 2, the positional embedding is [0.9093, -0.4161, 0.02, 0.9998]

Consequently, adding these positional embeddings to the token embeddings gives us the final embeddings.

For ‘The’:

$$FE(\text{The}) = TE(\text{The}) + PE(0)$$

$$FE(\text{The}) = [0.1, 0.2, 0.3, 0.4] + [0, 1, 0, 1]$$

$$FE(\text{The}) = [0.1, 1{,}2, 0.3, 1.4]$$

For ‘car’:

$$FE(\text{car}) = TE(\text{car}) + PE(1)$$

$$FE(\text{car}) = [0.1, 0.2, 0.3, 0.4] + [0, 1, 0, 1]$$

$$FE(\text{car}) = [0.1, 1{,}2, 0.3, 1.4]$$

For ‘drove’:

$$FE(\text{drove}) = TE(\text{drove}) + PE(2)$$

$$FE(\text{drove}) = [0.1, 0.2, 0.3, 0.4] + [0, 1, 0, 1]$$

$$FE(\text{drove}) = [0.1, 1{,}2, 0.3, 1.4]$$

Along with the semantic information, these final embeddings now also contain positional information.

b) Absolute Learned Positional Embeddings –

These provide an alternative to sinusoidal positional embeddings and are employed in models like BERT, GPT and RoBERTa. The performance and the results produced by learned embeddings were found to be very similar to sinusoidal embeddings [1]. Learned positional embeddings, instead of using a fixed math function as in sinusoidal embeddings, directly learn the positional embedding from the data, just as it would

learn other parameters of the model. A position vector is learned for every position in the input sequence. A shortcoming of this method is the maximum length that can be represented is bounded. For example, if you are only able to learn sequences up to a length of 512, you will not be able to represent the input sequences with a number of tokens exceeding 512.

c) Relative Postional Embeddings -

Although sinusoidal embeddings are very effective, they fail to provide any information about the closeness of tokens. For example, in sinusoidal embeddings, position 1 and position 2 are no different than position 1 and position 512. In an input sequence of large number of tokens, it is obvious that the tokens at consecutive positions (or tokens close to each other) will be more related to each other compared to tokens that are far apart, such as tokens at position 1 and position 512.

Relative positional embeddings provide a solution by allowing us to encode information about the relative positions of tokens and not just their absolute positions. Instead of representing the absolute position of a token, we represent the relative distance between a pair of tokens. For example, in the sentence "I am a footballer", the position of 'I' is 1 and 'footballer' is 4. Absolute embeddings record these values whereas, relative embedding will record the distance between the two words which is 3. So, if token *a* is at position 3 and token *b* is at position 7, the relative position will be *b-a*, i.e., 4.

In previous methods, we added the vectors containing such positional information to the token embedding vectors. In the case of relative embeddings, the attention scores are modified to include the positional information. Such modifications are made by adding a bias term to the attention scores calculations. Compared to absolute embeddings, relative embeddings are preferred handling longer sequence lengths and generalize better to sequence lengths unseen during training [12].

Let us dive into the math involved in various relative positional encoding methods. Every paper proposes a different method to include relative positional information in the transformer. We will look at three papers, namely, Shaw et.al (2018) [12], Dai et.al (2019) [13] and Raffel et.al (2020) [14].

Firstly, we must know that the query, keys and values of the self-attention block are represented by the following equations.

$$q_m = f_q(x_m, m)$$

$$k_n = f_k(x_n, n)$$

$$v_n = f_v(x_n, m)$$

Here, through the functions f_q, f_k and f_v, the m^{th} and n^{th} positions are comprised in q_m , k_n and v_n. The query and key values are then used to calculate the attentions weights and the final output is the calculated as the weighted sum over the value representation.

$$a_{m,n} = \frac{\exp\left(\frac{q_m^T k_n}{\sqrt{d}}\right)}{\sum_{j=1}^{N} \exp\left(\frac{q_m^T k_n}{\sqrt{d}}\right)}$$

$$o_m = \sum_{n=1}^{N} a_{m,n} v_n$$

Now, in the case of sinusoidal embeddings, the function f_k will be written as, $f_k(x_n, n) = W_k(x_n + p_n)$. Note that 'x' here represents the token embeddings and 'p' represents the positional embeddings. Since, we are discussing the case of sinusoidal embeddings, the positional embeddings include the absolute positional information. The calculation of this 'p' was discussed in the previous section.

For, relative embeddings, as discussed in Shaw et.al [12], the functions are represented by the following equations.

$$f_q(x_m)=W_q x_m$$

$$f_k(x_n,n)=W_k(x_n+\tilde{p}_r^k)$$

$$f_v(x_n,n)=W_v(x_n+\tilde{p}_r^v)$$

The positional embeddings are now represented by a new element 'p_r' which comprises of the relative positional information between two tokens. In this method, there is no positional embedding in query. The 'r' in these equations is the distance relative distance between positions 'm' and 'n'. So, for a query at position m=2 and a key at position n=4, 'r' will be equal to 2. Similarly, also for m=51 and n=53, 'r' will still be equal to 2. This shows that the absolute positions of the tokens lose significance as we are now only focusing in the 'closeness' of the tokens by incorporating the relative distances in the calculations. Notice that they have clipped the relative distance, since after a certain amount of distance, the relative positional information seems to be of no significant benefit.

The Transformer-XL paper [13], emphasized on the expansion of the '$q_m^T k_n$' term of equation 2 as follows,

$$q_m^T k_n = x_m^T W_q^T W_k x_n + x_m^T W_q^T W_k p_n + p_m^T W_q^T W_k x_n + p_m^T W_q^T W_k p_n$$

Along with this paper, many others have worked on the terms p_n and p_m and proposed that they be replaced with certain relative values instead of absolute values. The Transformer-XL paper has made three changes to equation 6. Firstly, p_n is replaced by a relative embedding as seen in the following equation. Subsequently, the term p_m is changed depending on where it is appearing in the equation. Observe the third and fourth terms of equation 6. In the third term, attention is calculated considering position as query and content as key. On the other hand, in the fourth term, attention is calculated using position as query and again position as key. For both cases, we have replaced p_m with different elements. p_m becomes u when key is content (token) and v when key is position. u and v vectors are initialized randomly and learned by backpropagation as we train the model. Lastly, the same vector W_k was being shared with both x_n and p_n. To have different weights for both

content and position, a new W_k was used with the position term p_m. All these changes have resulted in the following equation.

$$q_m^T k_n = x_m^T W_q^T W_k x_n + x_m^T W_q^T \widetilde{W}_k \tilde{p}_{m-n} + u_m^T W_q^T W_k x_n + p_m^T W_q^T \widetilde{W}_k \tilde{p}_{m-n}$$

Authors of the T5 paper [14], while retaining the first term of equation 6, replaced all of the other terms with a trainable bias $b_{i,j}$.

$$q_m^T k_n = x_m^T W_q^T W_k x_n + b_{i,j}$$

They also proposed a new equation, making an addition of an extra term to equation 8. The last term of equation 6 was reintroduced while using different weights.

$$q_m^T k_n = x_m^T W_q^T W_k x_n + p_m^T U_q^T U_k p_n + b_{i,j}$$

Another paper by He et.al [15], focused on the middle two terms of equation 6 and claimed that the relative positions between two tokens can be modelled using only these two terms. They simply replaced p_n and p_m with relative embeddings as follows.

$$q_m^T k_n = x_m^T W_q^T W_k x_n + x_m^T W_q^T W_k \tilde{p}_{m-n} + \tilde{p}_{m-n}^T W_q^T W_k x_n$$

In the following section, a brief comparison is made between the performance of various positional embeddings discussed in this chapter. Relative embeddings require an extra step in the self-attention layer to add the positional matrix to the query-key self-attention matrix. This makes them considerably slower than sinusoidal embeddings [16]. This can be observed in the image below.

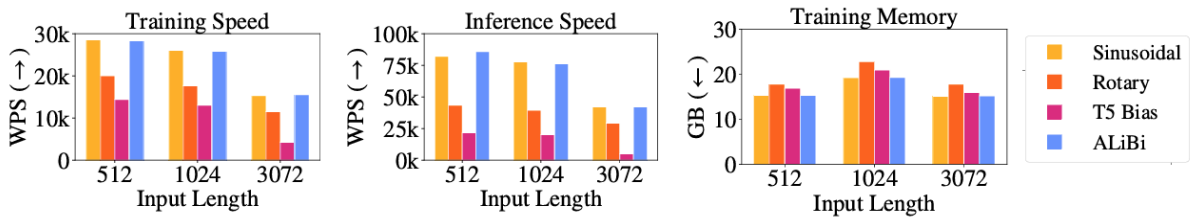

FIGURE 3 - COMPARISON OF TRAINING TIMES, INFERENCE TIMES AND MEMORY USE OF VARIOUS POSITIONAL EMBEDDING METHODS [16].

The T5 Bias here represents the relative embeddings. It can be clearly observed that both training and inference speeds are slower with relative embeddings as compared to sinusoidal or rotary positional embeddings. They also require more training memory than sinusoidal embeddings and just slightly lesser memory than rotary embeddings. Due to such challenges, the use of relative positional embeddings is not very common, especially for larger models.

d) RoPE Rotary Positional Embeddings -

The biggest difference between absolute/relative embeddings and rotary embeddings is that in rotary embeddings we multiply the positional embeddings into the vectors of query and key, instead of adding them as in the case of absolute and relative embeddings. So, instead of adding the vectors, we are actually rotating the vector by a certain angle theta. This angle represents the absolute position of the token in the sequence. The relative positional information is also preserved as the angle between two vectors corresponds to the distance between the tokens they represent. Rotary positional embeddings, therefore, encode both the absolute positional information and the relative positional information of tokens. In the following section, we will look at the math involved in the implementation of rotary embeddings.

The RoFormer paper [17], has discussed about how such rotary positional embeddings can be implemented. We will start off with a case where the dimension of the token embedding vector is 2 and then generalize the formulation for higher dimensional vectors. The query and key vectors are given by the following formulas.

$$q_m = f_q(x_q, m)$$

$$k_n = f_k(x_k, n)$$

These vectors are represented by two functions f_q and f_k which take in two arguments. A x_m and x_n represent the content vectors whereas m and n are the positions at which these content vectors lie. In the subsequent step, we take a dot product of the query and key vectors. This dot product is defined by a function g which has three arguments namely, the content x_m of query, content x_n of key, and the difference $n - m$ in their positions.

$$q_m^T k_n = \langle f_q(x_m, m), f_k(x_n, n) \rangle = g(x_m, x_n, n - m)$$

After derivation, we obtain the following representations of the functions.

$$f_q(x_m, m) = (W_q x_m) e^{im\theta}$$

$$f_k(x_n, n) = (W_k x_n) e^{in\theta}$$

$$g(x_m, x_n, n - m) = Re[(W_q x_m)(W_k x_n) * e^{i(m-n)\theta}]$$

Here, the angle θ is set to a non-zero constant. Real part of the complex number is represented by 'Re' and the complex conjugate number of $(W_k x_n)$ is $(W_k x_n)^*$. We can further represent the above equations in matrix multiplication form using rotation matrix.

$$f_{\{q,k\}}(x_m, m) = \begin{pmatrix} cosm\theta & -sinm\theta \\ sinm\theta & cosm\theta \end{pmatrix} \begin{pmatrix} W_{\{q,k\}}^{(11)} & W_{\{q,k\}}^{(12)} \\ W_{\{q,k\}}^{(21)} & W_{\{q,k\}}^{(22)} \end{pmatrix} \begin{pmatrix} x_m^{(1)} \\ x_m^{(2)} \end{pmatrix}$$

The first matrix is the rotation matrix, the second is the weight matrix and the third is the content vector. We will now scale this formulation to embedding vectors with dimensions higher than two. Let us look at a case with d=512. Following general equation is used for vectors of dimension d and the rotation matrix gets transformed as follows.

$$f_{\{q,k\}}(x_m, m) = R^d_{\Theta,m} W_{\{q,k\}} x_m$$

$$R^d_{\Theta,m} = \begin{pmatrix} cos\, m\theta_1 & -sin\, m\theta_1 & 0 & 0 & \dots & 0 & 0 \\ sin\, m\theta_1 & cos\, m\theta_1 & 0 & 0 & \dots & 0 & 0 \\ 0 & 0 & cos\, m\theta_2 & -sin\, m\theta_2 & \dots & 0 & 0 \\ 0 & 0 & sin\, m\theta_2 & cos\, m\theta_2 & \dots & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \dots & cos\, m\theta_{d/2} & -sin\, m\theta_{d/2} \\ 0 & 0 & 0 & 0 & \dots & sin\, m\theta_{d/2} & cos\, m\theta_{d/2} \end{pmatrix}$$

The above matrix is known as the rotary matrix where every two dimensions are rotated at a time. This phenomenon can be observed in the image below.

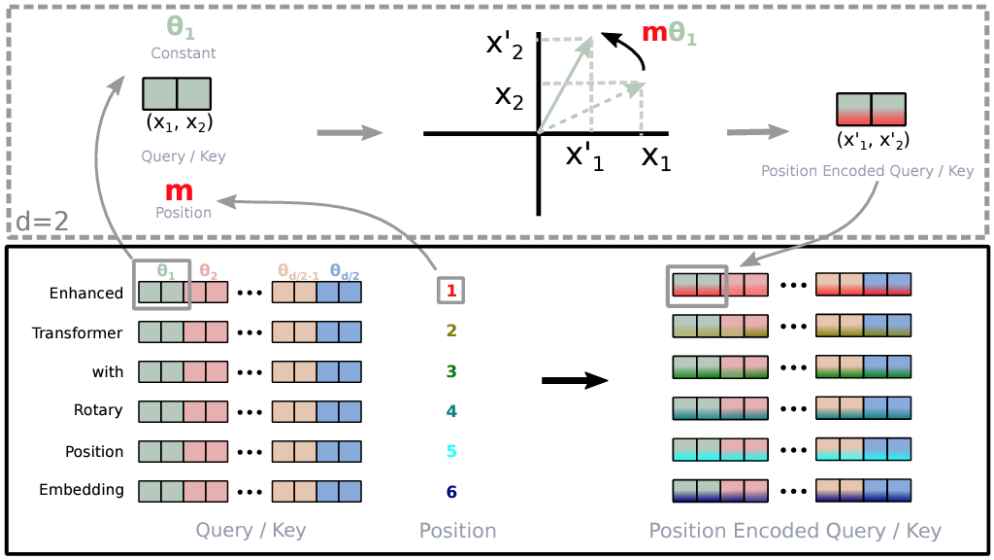

FIGURE 4 - ROPE IMPLEMENTATION [17]

See how each angle θ multiplied by position m represents the position of vectors. Also, it must be observed that only two dimensions are being rotated at a time. Another observation which can be made is that the rotation matrix is extremely sparse. We can overcome this sparsity and improve computational efficiency by transforming our rotation matrix to the following form.

$$R^{d}_{\Theta,m} = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ \vdots \\ x_{d-1} \\ x_d \end{pmatrix} \otimes \begin{pmatrix} cos\, m\theta_1 \\ cos\, m\theta_1 \\ cos\, m\theta_2 \\ cos\, m\theta_2 \\ \vdots \\ cos\, m\theta_{d/2} \\ cos\, m\theta_{d/2} \end{pmatrix} + \begin{pmatrix} -x_2 \\ x_1 \\ -x_4 \\ x_3 \\ \vdots \\ -x_d \\ x_{d-1} \end{pmatrix} \otimes \begin{pmatrix} sin\, m\theta_1 \\ sin\, m\theta_1 \\ sin\, m\theta_2 \\ sin\, m\theta_2 \\ \vdots \\ sin\, m\theta_{d/2} \\ sin\, m\theta_{d/2} \end{pmatrix}$$

RoPE has provided a new way of including positional information in language models. Implementation of rotary positional embeddings over other methods has demonstrated better performance in certain experiments. When tested against BERT, RoFormer showed lesser MLM loss during the pre-training phase [17]. Similarly, when the PerFormer [18] was used with and without RoPE, the training loss was much lower in the case when it was used with RoPE. In the following sections we will look at some other embeddings which are different from the ones we discussed in the previous sections.

2.3 Contextual embeddings –

As opposed to simple word embeddings, in contextual embeddings a word can have different embeddings based on the context in which the word was found. BERT, which was developed by google uses contextual embeddings. Contextual embeddings dynamically adjust the representation of a word depending on its surrounding words. Context is very important for proper functioning of language models.

Consider the following two sentences,

- "Take a right turn."
- "My answer is right."

where the word 'right' has different meanings. In traditional embeddings such as Word2Vec, the word 'right' in both the sentences will have the same token. For the model to accurately comprehend the sentence, it becomes mandatory to take the context into consideration. Contextual embeddings take care of this problem by assigning a different token to same words if they appear in a different context [19]. Some popular models which use contextual embeddings are ELMo (Embeddings from language models, BERT (Bidirectional Encoder Representations from Transformers) and GPT-2 (Generative Pre-trained Transformer) [20]. The following table compares contextual embeddings with the embeddings discussed in section 2.1.

Feature	Traditional Embeddings (Word2Vec, Glove)	Contextual Embeddings (BERT, GPT, ELMo)
Embedding Type	Static	Dynamic
Polysemy Handling	Word senses cannot be distinguished	Can differentiate based on context
Context Awareness	Context is ignored	Considers context of the surrounding
Representation	Same for every occurrence of the word	Different for each occurrence depending in context

TABLE 1 – COMPARISON BETWEEN CONTEXTUAL AND TRADITIONAL EMBEDDINGS

2.4 Multimodal embeddings –

The type of embeddings we have seen so far deal with representation of texts. When data of different modalities such as, images, audio, and video is to be represented, we use multimodal embeddings. The features and characteristics of such modalities is captured and represented in vector format using multimodal embeddings [21]. These embeddings help models to understand both textual descriptions and visual elements simultaneously, which is crucial for tasks such as image captioning. Some other examples, from the many different applications of these embeddings are text-to-speech and speech-to-text tasks, and also visual question answering, where the model answers questions asked about the content of the image or video [22].

Normal embeddings encode and represent relationship between a single data modality such as text or image. Multimodal embeddings go a step beyond by encoding and establishing relationship between more than one type of data modalities in a shared embedding space. Based this idea, many models such as CLIP and ImageBind have been developed. CLIP is a model developed by OpenAI which maps both text and image data in the same embedding space [23]. Similarly, ImageBind by MetaAI brings together six different modalities namely text, images, audio, depth, thermal and IMU data under a joint embedding space [24]. It is able to instantly suggest images based on an audio clip and vice versa [25]. Along with this, ImageBind is capable of many other tasks. Multimodal embeddings have thus enabled us to incorporate multiple modalities under a single embedding space and hence effective perform such cross-modality tasks.

Embeddings are a crucial inseparable part of transformer models. Both content and positional embeddings were studied in the sections above. Numerous types of positional embeddings and the math involved have been discussed in detail. Similarly, content embeddings including contextual and multimodal embeddings have been covered in this chapter.

3. TOKENIZATION – PROCESS AND TYPES

Introduction –

Tokenization is the first and a crucial step of transformer models. It transforms the raw text into a certain form that enables the model to understand human language. The text is broken down into smaller units called tokens (numerical representation of text) which the language model works with. Tokens are not mere numerical representations of text but are also used in machine learning pipeline as features. In the following section we will look at the different types of tokenizations and the various tokenization methods employed by large language models. Let us begin by taking a look at the different types and how the input sequences are broken down into workable parts.

3.1 Types of Tokenization

Tokenization is differentiated based on how the piece of text is broken down into smaller parts. It can either be broken down into words, characters or sentences in case of large input sequences.

a) Word – Level Tokenization

In word level tokenization, the text is broken down into words which are considered as the smallest meaningful units. Methods like this are specifically effective when languages with clear word boundaries such as English are to be processed. Although it is very intuitive, there certain limitations associated with it. Words are differentiated based on spaces or punctuation marks. For example, look at the following sentence.

"The horse jumped over the fence"

The tokenization will produce –

["The", "horse", "jumped", "over", "the", "fence"]

Each word will have its own individual token. Now, since our model can only understand numbers, these tokens are converted to integers know as token IDs. Every token is mapped to an unique integer. Although word level tokenization is very intuitive, there certain limitations associated with it. If it encounters a word that is not available in the model's vocabulary, that word will be treated as unknown and out of vocabulary. With more complex languages, such as German, which forms compound words using smaller words, the model would require large vocabularies increasing training time. In such cases, and even with other complex languages such as Finnish or Turkish, language specific tokenization techniques must be used. Further, languages like Mandarin and Japanese do not use spaces between words, in which case we have to use character level tokenization as follows.

b) Character – Level Tokenization

In this method, text is broken down into individual characters. This method is specifically useful when word boundaries are not known. Since, every character is an individual token, even unknown words are conveniently handled. It is highly flexible but results in longer sequences for the model to process. Observe the following sentence to see how each character represents a token.

"The horse jumped."

The tokenization will produce –

["T", "h", "e", " ", "h", "o", "r", "s", "e", " ", "j", "u", "m", "p", "e", "d", "."]

Observe how spaces and punctuations are also tokenized. In this method, since every character is a valid token, the chances of having out-of-vocabulary (OOV) words are close to zero. This also means that it is very easy to implement as the need to create large vocabularies is eliminated. Another important point, which was a limitation of the previous technique is that character level tokenization works across all scripts (Latin, Mandarin, etc.). An important drawback that must be pointed out is that computational complexity of training and inference is increased as this technique produces much longer sequences than word level tokenization. The model can also experience trouble capturing the meaning as each token comes packed with much less information.

c) Subword – Level Tokenization

This tokenization technique combines the best of both word-level and character-level tokenization techniques. It is a standard and preferred tokenization technique for large language models [26]. As discussed in the previous sections, both the techniques have some limitations. Word-level needs a very large vocabulary and can still have many OOV words. Similarly, character-level produces very long sequences and less meaningful tokens. Subword-level eliminates both of these limitations. Consider the following word,

"Capitalization"

The tokenization will produce (depending on the algorithm) –

["Capital", "ization"]

["Capital", "##ization"]

["Cap", "ital", "ization"]

The two hashes denote that the part 'ization' completes the previous word. The main idea behind sub-word tokenization is that the frequently appearing words should not be split but the rare words should be split into more meaningful subwords. Therefore, this technique has now enabled the model to handle the common words as whole tokens and broken down the rare words into subtokens. The limitations of both the previous types of tokenization techniques are thus eliminated. We will no longer have OOV words as rare words will be decomposed, and the words are still being tokenized as single tokens resulting in smaller sequences than character-level tokenization. Almost all of the currently available models including GPT, BERT, DistilBERT and Electra exclusively use subword tokenization [27]. Look at the following section to see how sub-word tokenization is implemented.

3.2 Algortihms of Tokenization –

In the previous section, we learned about the different types of tokenization methods that are available for NLP task. Now, we will study the various algorithms that are utilized to implement these tokenization techniques in large language models.

a) Byte Pair Encoding (BPE)

One of the most widely used sub-word tokenization algorithms is Byte Pair Encoding. According to Gage et al., (1994), where this algorithm was first introduced, it is a very straightforward data compression method that involves substitution of a new, unused, single byte in the place of a most frequently appearing pair of bytes [28].

We first start with normalization and pre-tokenization. Normalization involves some basic tasks such as eliminating unnecessary whitespaces, removing accents, lowercasing alphabets, etc. [29]. Pre-tokenization focusses on breaking down of the input sequence to word-level. Space tokenization, which separates words

based on spaces between them, or any other method that facilitates the decomposition of the input sequence into words can be used.

Once the text is broken down into words, we move on to create a character level vocabulary. Every character used to form these words is included in this vocabulary. In BPE, frequently appearing pairs are added to this vocabulary one by one. Let us look at examples to understand this algorithm in detail.

Suppose our corpus of text includes the following words with corresponding frequencies,

("flag", 10), ("grab", 6), ("can", 14), ("fan", 2)

First, we will split each word into characters, such that each word will be represented as tokens,

("f", "l", "a", "g", 10), ("g", "r", "a", "b", 6), ("c", "a", "n", 14), ("f", "a", "n", 2)

This is our corpus represented at a character level. Our vocabulary will include all of these characters. Both corpus and vocabulary are shown below.

Corpus – ("f", "l", "a", "g", 10), ("g", "r", "a", "b", 6), ("c", "a", "n", 14), ("f", "a", "n", 2)

Vocabulary – ["f", "l", "a", "g", "r", "b", "c", "n"]

We will add new pairs to this vocabulary that are appearing frequently. The pair ("a", "n") is appearing with the highest frequency of 16. The second highest one is ("a", "g") with 10. Therefore, we choose ("a", "n") and it will be merged as ("a", "n") -> "an". Our vocabulary will be updated accordingly.

Corpus – (“f”, “l”, “a”, “g”, 10), (“g”, “r”, “a”, “b”, 6), (“c”, “an”, 14), (“f”, “an”, 2)

Vocabulary – [“f”, “l”, “a”, “g”, “r”, “b”, “c”, “n”, “an”]

Eventually, we may encounter situations where three characters are being used to form token. Currently, a two-character pair with highest frequency of 10 is (“a”, “g”). We have another pair which is formed by three characters (“c”, “an”) and has a frequency of 14. Therefore, we will merge this three-character pair as (“c”, “an”) -> “can”, and update the vocabulary as follows.

Corpus – (“f”, “l”, “a”, “g”, 10), (“g”, “r”, “a”, “b”, 6), (“can”, 14), (“f”, “an”, 2)

Vocabulary – [“f”, “l”, “a”, “g”, “r”, “b”, “c”, “n”, “an”, “can”]

We finally have a complete word “can” in our vocabulary. Using the same idea, we merge (“a”, “g”) -> “ag” and update the vocabulary.

Corpus – (“f”, “l”, “ag”, 10), (“g”, “r”, “a”, “b”, 6), (“can”, 14), (“f”, “an”, 2)

Vocabulary – [“f”, “l”, “a”, “g”, “r”, “b”, “c”, “n”, “an”, “can”, “ag”]

This process is continued until a required vocabulary size is achieved [29]. The size of vocabulary can be restricted to a certain number. Now, let us see how the vocabulary created will be used to tokenize a sample input sequence.

Suppose our tokenizer encounters the word “bag”. Since all the characters of the said word are included in the vocabulary, it will be tokenized as (“b”, “ag”). Another word “man” will be tokenized differently. Since the character “m” does not appear in the vocabulary, this word will be tokenized as (“[UNK]”, “an”).

The BPE algorithm to implement subword tokenization method is pretty straight forward as seen. Due to this ease and ability to handle OOV words, use of BPE in natural language processing tasks has shown reduction

in the size of language models and improved model performance. This tokenization technique is therefore used by models such as GPT-2 and RoBERTA [30].

b) WordPiece

Word piece is another popular tokenization algorithm used by a bunch of different models. It is quite similar to BPE as this algorithm also focusses on merging pairs. However, as in the previous algorithm, we focussed on merging the most frequent pairs, in this algorithm we will be merging the pairs using a certain formula. This model was first introduced by Schuster et al. back in 2012. According to this paper, the pairs which lead to an increase in the likelihood of the training data, when added to the model, are merged first [31]. A better understanding of this concept can be gained through the following example.

Suppose we have two pairs ("a", "##like") and ("mu", "##tual"). The two parts of the first pair can frequently appear on their own, whereas, in the second the pair, the probability of both the parts appearing alone is quite possibly zero. As a result, this algorithm will first merge the second pair. In short, it first merges the pairs who are more likely to appear in the corpus as pairs and less likely as individual parts. Look at the following formula to understand how this inference was made.

Score = (Frequency_of_Pair) / (Frequency_of_Element_1)] x [Frequency_of_Element_2)]

Pairs with a higher score are merged first. The frequency of the pair is divided by the product of the frequencies of individual parts and doing so, the merging of pairs with less frequent individual parts is prioritized [32].

Let us understand this using the same vocabulary we used in the BPE example.

Corpus – ("f", "##l", "##a", "##g", 10), ("g", "##r", "##a", "##b", 6), ("c", "##a", "##n", 14), ("f", "##a", "##n", 2)

Vocabulary – [“f”, “l”, “a”, “g”, “r”, “b”, “c”, “n”]

Notice how the splits are now represented with a slight difference. The hashes are used to denote that a part is a continuation of the previous part. Here, the pair with highest frequency of 16 is (“a”, “n”) and the frequencies of ‘a’ and ‘n’ are 32 and 16 respectively. The score, therefore, becomes 1/32 which is definitely not the highest. The individual frequency of ‘a’ was very high which led to this score and thus every pair containing the character ‘a’ will give a high score. Keeping all the pairs containing this character out, we get two pairs namely (“f”, “l”) and (“g”, “r”). Using the same formula, the score for (“f”, “l”) turns out to be 1/12 and for (“g”, “r”) is 1/6. Hence, (“g”, “r”) is the pair with the highest score and becomes the first pair to be merged. The vocabulary takes the following shape.

Corpus – (“f”, “##l”, “##a”, “##g”, 10), (“gr”, “##a”, “##b”, 6), (“c”, “##a”, “##n”, 14), (“f”, “##a”, “##n”, 2)

Vocabulary – [“f”, “l”, “a”, “g”, “r”, “b”, “c”, “n”, “gr”]

Note, that the hashes preceding ‘r’ have been removed after the merger with ‘g’. Only one pair without ‘a’ now remains and hence it is merged.

Corpus – (“fl”, “##a”, “##g”, 10), (“gr”, “##a”, “##b”, 6), (“c”, “##a”, “##n”, 14), (“f”, “##a”, “##n”, 2)

Vocabulary – [“f”, “l”, “a”, “g”, “r”, “b”, “c”, “n”, “gr”, “fl”]

Since, ‘a’ is common to all pairs, we can merge any pair. Let’s merge the first pair we encounter (“fl”, “a”).

Corpus – (“fla”, “##g”, 10), (“gr”, “##a”, “##b”, 6), (“c”, “##a”, “##n”, 14), (“f”, “##a”, “##n”, 2)

Vocabulary – [“f”, “l”, “a”, “g”, “r”, “b”, “c”, “n”, “gr”, “fl”, “fla”]

In the next step, the pair (“fla”, “g”) has the highest and hence, we merge it.

Corpus – (“fla”, “##g”, 10), (“gr”, “##a”, “##b”, 6), (“c”, “##a”, “##n”, 14), (“f”, “##a”, “##n”, 2)

Vocabulary – [“f”, “l”, “a”, “g”, “r”, “b”, “c”, “n”, “gr”, “fl”, “fla”, “flag”]

We got our first complete word! This process can be further continued till the desired vocabulary size is achieved. As seen through this example, WordPiece algorithm is largely similar to BPE, except for the fact that we use a different rule to select which pair to merge first. There are, however, some other difference that must also be pointed out.

As opposed to WordPiece, BPE does not save the merge rules learned, instead only the final vocabulary is saved. To tokenize a word, the WordPiece algorithm finds the longest subword of that word in the dictonary and splits it at that point [33]. In our case, if we wanted to tokenize the word “flagr” (random word), the longest subword present in the vocabulary associated with this word would be “flag” and hence we split there and get [“flag”, “##r”]. The final tokenization of “flagr” will be [“flag”, “##r”] since “r” is present in our vocabulary. BPE on the other hand would tokenize this word as [“fla”, “##gr”] as the merges learned in order are applied.

If we had to tokenize the word “flags” in which the character “s” is not present in the vocabulary, Wordpiece algorithm will tokenize this word as [“[UNK]”]. One would imagine it to be [“fla”, “g”, “[UNK]”], as in the case of BPE, but the whole word is tokenized as [“[UNK]”] even if a single character is missing in the vocabulary. In real case scenario, the chances of a “[UNK]” token appearing are very low as all the characters will be included in the vocabulary. WordPiece has proven to be beneficial in decreasing the vocabulary size, which leads to text data being encoded efficiently [34]. It has become a preferred tokenizer for foreign language models such as Japanese, Korean and Arabic as it was originally designed for solving segmentation problem in such languages [32,34]. The BERT model as proposed in Devlin et al., 2018 uses the WordPiece tokenization algorithm [35].

c) Unigram

Unigram is yet another tokenization techniques which is not much similar to BPE or WordPiece. Instead of creating a vocabulary from scratch, as in the two previous algorithms, we start off from a large vocabulary and eliminate the unrequired tokens from it, until an optimal desired size vocabulary is achieved. This algorithm, first proposed by Kudo (2018), has the ability to output multiple word segmentations with their probabilities [35]. This initial vocabulary can be created by any method, even by applying BPE on the original text data.

In Unigram algorithm, while training, we compute a total loss using the current vocabulary over the entire corpus of text data. Subsequently, we focus on individual tokens and identify the token who's removal from the vocabulary will result in the highest loss. Such tokens with high loss contribution to the total loss, are identified as important tokens and kept in the vocabulary. On the other hand, tokens which do not significantly increase the overall loss are eliminated, as their existence in the vocabulary was of less importance.

We don't remove tokens one by one, but eliminate a chunk of tokens, usually 10 to 20 percent (a hyperparameter that can be controlled) of the tokens associated with low increase in loss [36]. We continue such iterations until we achieve the required vocabulary size. Note that the base characters are never eliminated, to ensure the tokenization of any word.

Before implementing the Unigram algorithm, which involves the calculation of total loss followed by observing the changes that occur when we remove certain tokens, we must first understand the tokenization strategy of the Unigram model. Let's start from the corpus used in previous algorithms.

Corpus – ("flag", 10), ("grab", 6), ("can", 14), ("fan", 2)

The initial vocabulary will include all the substrings.

Vocabulary – [“f”, “l”, “a”, “g”, “fl”, “la”, “ag”, “fla”, “lag, “flag”, “r”, “b”, “gr”, “ra”, “ab”, “gra”, “rab”, “c”, “n”, “ca”, “an”, “fa”]

Every token in the Unigram model is considered independent from the token before it. To tokenize a give word, we examine the probabilities of all the different segmentations that can be used to form the word. The segmentations that give the highest probability are chosen as the tokens of that word. This probability is actually the frequency of that particular token divided by the summation of frequencies of all tokens in the vocabulary. Take a look at the following example. All the tokens with their frequencies are listed below

(“f”, 12) (“l”, 10) (“a”, 32) (“g”, 6) (“fl”, 10) (“la”, 10) (“ag”, 10) (“fla”, 10) (“lag, 10) (“flag”, 10)

(“r”, 6) (“b”, 6) (“gr”, 6) (“ra”, 6) (“ab”, 6) (“gra”, 6) (“rab”, 6) (“c”, 14) (“n”, 16) (“ca”, 14)

(“an”, 16) (“fa”, 2)

Token “ra”, for example, will have a probability of 6/224 (sum of all freq. = 224). To tokenize a given word, we will look at all the possible segmentations of that word and calculate the probabilities. As all tokens in this method are considered independent, the probability is just the product of probabilities of all tokens used to form that word. Note that different combinations of tokens can be used to form the same word. Therefore, through calculations we choose the combination that leads to the highest probability and tokenize the word using that combination of tokens.

Suppose the word “can” is to be tokenized. It can be segmented as [“c”, “a”, “n”], [“c”, “an”] or [“ca”, “n”]. In each of these case, the probabilities are as follows.

P([“c”, “a”, “n”]) = P(“c”) x P(“a”) x P(“n”) = (14/224) x (32/224) x (16/224) =0.00063

P([“c”, “an”]) = P(“c”) x P(“an”) = 0.0044

P([“ca”, “n”]) = P(“ca”) x P(“n”) = 0.0044

It can be seen that tokenizations with the least number of tokens give out higher probability (hence these will be selected). The word “can” will be tokenized as either [“c”, “an”] or [“ca”, “n”] depending on which is encountered first in the corpus. Observe how we were able to tokenize the word with least number of tokens (2 instead of 3). These calculations are fairly straightforward and easy to implement, however, in real case scenarios, the calculations can become quite tedious, and thus an algorithm known as Viterbi algorithm is used. This was tokenization process in Unigram model. We will now jump back to the loss calculations part of the Unigram algorithm.

During every iteration, we tokenize every word in the corpus and calculate the loss. Every word is associated with a score (probability) and the loss is equal to the negative log likelihood of this score. Summing losses for all the words gives us the total loss. Observe the following example for a better understanding. Our corpus was as follows.

Corpus – (“flag”, 10), (“grab”, 6), (“can”, 14), (“fan”, 2)

Furthermore, the tokenizations for each word were calculated using the steps mentioned previously. The scores for every word in our corpus are as follows.

“flag” – [“f”, “lag”] = 0.0024

“grab” – [“gra”, “b”] = 0.0007

“can” – [“ca”, “n”] = 0.0044

“fan” – [“f”, “an”] = 0.0038

Using these scores, the loss is calculated as follows.

Formula [27,35]–

$$\mathcal{L} = -\sum_{i=1}^{N} \log \left(\sum_{x \epsilon S(x_i)} p(x) \right)$$

Where, $x_1, x_2, \ldots x_N$ are all the words in the training corpus and $S(x_i)$ represents all the possible tokenizations for the word x.

10 * (-log(0.0024)) + 6 * (-log(0.0007)) + 14 * (-log(0.0044)) + 2 * (-log(0.0038)) = 191.02

Subsequently, we are required to check how the removal of a token from the vocabulary affects this loss. Such calculations are quite cumbersome but can be easily implemented using PyTorch. For the sake of demonstration and understanding how tokens are eliminated. Look at the following example.

If we eliminated the token “an”, the word “fan” will have to be tokenized as [“fa”, “n”] which will result in a score of 0.00063. This will cause the loss to rise by,

-2 * (-log(0.0038) + 2 * (-log(0.00063) = 3.59

Similarly, if we remove the token “lag” from the word “flag”, the word will be tokenized as [“fla”, “g”] and the loss would rise by 2.33. On the other hand, if you look at tokens such as “ca” and “gra”, the removal of these tokens would cause absolutely no effect on the loss. The words associated with these tokens, which are “can” and “grab”, can be tokenized as [“c”, “an”] and [“g”, “rab”] respectively, with the exact same scores. Hence, the loss would be unaffected. The Unigram algorithm will thus eliminate these tokens (and other 10 to 20 percent of such tokens) and leave the ones whose removal will cause a hike in the loss such as “lag” and “an”. The same process is continued until a desired vocabulary size is achieved.

According to Kudo (2018), not only does Unigram have the same benefits as the BPE algorithm, but it is also more flexible and thanks to its probabilistic language model origins, it is capable of outputting multiple segmentations along with their probabilities [35]. Due to such benefits, it is used by models such as T5, mBART, AlBERT, Big Bird and XLNet [36].

d) SentencePiece

SentencePiece is not really a tokenization algorithm but it is a tokenization tool used in unison with other algorithms such as BPE and Unigram to tokenize and detokenize text. First introduced by Google in Kudo & Richardson (2018), SentencePiece eliminates the need of pre-tokenization which segments the input into word sequences. Therefore, we can directly train models frow raw sentences, thus enabling us to create an end-to-end language independent system [37]. This is especially useful for languages such as Japanese and Thai which do not use spaces to separate words. Each character is separated using a special character.

SentencePiece comprises of four components namely, normalization, trainer, encoder and decoder [37]. Semantically equivalent Unicode characters are normalized into canonical forms using normalizer. The trainer employs a subword segmentation algorithm such as BPE or Unigram to train from the normalized corpus. The function of encoder is to utilize the normalizer to normalize the input text and tokenize it into subword sequences using the trainer. Decoder on the other hand converts such subword sequences into normalized text.

Encoder and decoder are basically tokenizing and detokenizing the text. In the case of SentencePiece, however, instead of saying tokenization and detokenization, we say encoding and decoding as it is capable of directly converting text to an id sequence. SentencePiece has shown higher BLEU scores over other models, for translations between Japanese and English, while using a significantly smaller vocabulary [13]. Another important advantage is that it performs lossless tokenization. No information is lost between tokenization and detokenization. The exact same normalized text, which was used as input before tokenization, is obtained at the end of detokenization step. Due to these benefits, especially when foreign languages are to be processed, SentencePiece automatically becomes an ideal option. Models such as AlBERT, XLnet, Marian and T5 currently use SentencePiece [27].

4. EXPLANATION OF BYTE PAIR ENCODING IMPLEMENTATION IN MATLAB

In this section, we will look at a detailed explanation of the code used to implement the BPE tokenization algorithm in Matlab. The code is as follows.

4.1 BPE Code

```
clear

clc


function vocab = get_vocab(data)

    % Converts the input data into a vocabulary where each word is split into individual characters

    % and the frequency of each word is counted.

    vocab = containers.Map;


    for i = 1:length(data)

        line = data{i};

        words = strsplit(line);  % Split the sentence into words


        for j = 1:length(words)

            word = words{j};


            % Split word into characters and add '</w>' to signify end of word

            word_chars = cellstr(reshape(char(word), 1, [])');  % Convert to cell array of characters
```

```
            word_chars{end + 1} = '</w>';  % Add the end-of-word marker

            % Join the characters with spaces
            word_bpe = strjoin(word_chars, ' ');

            % Update frequency in vocab
            if isKey(vocab, word_bpe)
                vocab(word_bpe) = vocab(word_bpe) + 1;
            else
                vocab(word_bpe) = 1;
            end
        end
    end
end

function pairs = get_stats(vocab)
    % Returns the frequency count of pairs of symbols (characters) in the vocabulary.
    pairs = containers.Map;

    vocabKeys = keys(vocab);
    for i = 1:length(vocabKeys)
        word = vocabKeys{i};
        freq = vocab(word);
        symbols = strsplit(word);  % Split the word into individual symbols

        % Iterate through symbol pairs
```

```
        for j = 1:(length(symbols) - 1)
            pair = sprintf('%s %s', symbols{j}, symbols{j + 1});
            if isKey(pairs, pair)
                pairs(pair) = pairs(pair) + freq;
            else
                pairs(pair) = freq;
            end
        end
    end
end

function vocab_out = merge_vocab(pair, vocab_in)
    % Merges the given pair of characters in the vocabulary
    vocab_out = containers.Map;

    % Create a regular expression for the bigram (pair of symbols)
    bigram = strrep(pair, ' ', '');  % Merge the two symbols (remove space between them)
    pattern = ['\<' pair '\>'];  % Look for exact matches (using word boundaries)

    vocabKeys = keys(vocab_in);
    for i = 1:length(vocabKeys)
        word = vocabKeys{i};
        new_word = regexprep(word, pattern, bigram);  % Merge the pair in the word
        vocab_out(new_word) = vocab_in(word);
    end
end
```

```
function vocab = byte_pair_encoding(data, n)
    % Performs Byte Pair Encoding on a given dataset and returns the final vocabulary
    vocab = get_vocab(data);

    for i = 1:n
        % Get the current pair frequencies
        pairs = get_stats(vocab);

        % Check if there are no pairs to merge
        if isempty(pairs)
            disp('No more pairs to merge');
            break;
        end

        % Extract keys and values from pairs (this is the frequency map)
        pairKeys = keys(pairs);
        pairValues = cell2mat(values(pairs));

        % Find the most frequent pair
        [~, bestIdx] = max(pairValues);
        best = pairKeys{bestIdx};  % The most frequent pair

        % Merge the best pair in the vocabulary
        vocab = merge_vocab(best, vocab);
    end
```

```
    % Display the final vocabulary after all merges
    disp('Final Vocabulary after all merges:');
    vocabKeys = keys(vocab);        % Extracting the final tokens
    vocabValues = values(vocab);    % Extracting the token frequencies

    % Display each token with its frequency
    for i = 1:length(vocabKeys)
        fprintf('%s -> %d\n', vocabKeys{i}, vocabValues{i});
    end
end

corpus = ['In this code we are implementing the byte pair encoding tokenization. BPE merges the most frequent pairs. It is a preferred tokenization method in many language models'];

data = strsplit(corpus, '.');  % Split the text into sentences
n = 150;  % Perform 10 BPE merges
vocab = byte_pair_encoding(data, n);

disp(vocab);
```

4.2 Code Explanation

Let us start with the different functions used. We have used four functions in this code namely, get_vocab, get_stats, merges_vocab and the main function which executes the algorithm byte_pair_encoding. A detailed explanation of all of these functions is given below.

a) function get_vocab(data)

This function is used to create a vocabulary from the input data. The input data in our case will be taken from the corpus in the main script. Vocabulary here refers to the collection of words and their corresponding frequencies.

```
function vocab = get_vocab(data)
```

We define a new function get_vocab. 'data' is the input parameter which contains text from the corpus and 'vocab' will be the output of the function.

```
vocab = containers.Map;
```

Here, we create a map object which stores the words as keys and their frequencies as values.

```
for i = 1:length(data)
```

This loop iterates over each sentence in the input data. Each sentence is treated as a string and processed to extract words.

```
line = data{i};
```

Each sentence (line) is extracted one by one from the data.

```
words = strsplit(line);
```

The line is split into individual words by spaces using 'strsplit()'. The sentence is converted into a cell array of words.

```
for j = 1:length(words)
```

The inner loop iterates through each word in the sentence.

```
word = words{j};
```

Variable ‘word’ stores the current word being processed from the cell array of words.

```
word_chars = cellstr(reshape(char(word), 1, [])');
```

This line coverts the word into individual characters. ‘char(word)’ coverts the string into a character array. ‘reshape(char(word), 1, [])’ reshapes the word into a column of characters. ‘cellstr()’ converts this character column into a cell array where each cell contains one character. The result is a cell array ‘word_chars’ that contains each character of the word as a separate element.

```
word_chars{end + 1} = '</w>';
```

A special token ‘<\w>’ is added to the end of ‘word_chars’ to indicate the end of the word.

```
word_bpe = strjoin(word_chars, ' ');
```

Characters in ‘word_chars’ are joined in a new string ‘word_bpe’ with spaces separating each character. For example the word “text” in ‘word_chars’ will be {‘t’, ‘e’, ‘x’, ‘t’, ‘<\w>}. The same word in ‘word_bpe’ becomes ‘t e x t <\w>’.

```
if isKey(vocab, word_bpe)
    vocab(word_bpe) = vocab(word_bpe) + 1;
else
```

```
vocab(word_bpe) = 1;
```

This loop checks if the processed version of the word exists in the vocab map. If it does, its frequency is incremented by 1. If it doesn't the word is added to the map with the current frequency.

b) `function get_stats(vocab)`

This function is used to compute and return the frequency count of pairs of characters from the input vocabulary. It identifies and counts all the adjacent pairs in the vocabulary.

```
pairs = containers.Map;
```

A map is created to store the frequency of pairs of characters in the vocabulary. The pairs(bigrams) will be the keys and their frequencies will be values.

```
vocabKeys = keys(vocab);
```

Extracts all keys (words which are stored as strings of individual characters separated by spaces) from the vocab.

```
for i = 1:length(vocabKeys)
```

This loop iterates over each word (key) in the vocab map.

```
word = vocabKeys{i};
```

'word' holds the current word being processed.

```
freq = vocab(word);
```

Retrieves the frequency of the current word from the vocab map. This value indicates how many times the word has appeared in the dataset.

```
symbols = strsplit(word);
```

The current word is split into individual symbols using 'strsplit(word)'. This converts the string into cell array of symbols.

```
for j = 1:(length(symbols) - 1)
```

This loop iterates through the characters (symbols) in the word, looking at pairs of adjacent symbols. The loop runs until 'length(symbols) – 1', to avoid going out of bounds when accessing symbols{j+1}.

```
pair = sprintf('%s %s', symbols{j}, symbols{j + 1});
```

A pair of adjacent symbols (characters), meaning two consecutive characters, are joined using a space between them.

```
if isKey(pairs, pair)
    pairs(pair) = pairs(pair) + freq;
else
    pairs(pair) = freq;
```

This loop increments the frequency of a pair if it exists or keeps it the same if it does not.

c) function merge_vocab(pair, vocab_in)

This function merges the pair of characters into one symbol across all the words in the vocabulary.

```
function vocab_out = merge_vocab(pair, vocab_in)
```

'vocab_in' is the input vocabulary. Each key in the map is a word and each word is split into individual characters separated by spaces. 'pair' is the pair of characters to be merged. 'vocab_out' is the ouput vocabulary after the pairs are merged.

```
bigram = strrep(pair, ' ', '');
```

This removes the space between the two characters.

```
pattern = ['\<' pair '\>'];
```

Creates a regular expression pattern to find exact matches of the pair in a word. If the pair to be merged is 'a n', only the full pair 'a n' is merged, not the other occurences of 'a' and 'n' in the word.

```
vocabKeys = keys(vocab_in);
```

Retrieves all the keys in the input vocabulary.

```
for i = 1:length(vocabKeys)
```

This loop iterates through every word in the vocabulary.

```
word = vocabKeys{i};
```

Retrieves current word from the vocabulary.

```
new_word = regexprep(word, pattern, bigram);
```

This line replaces the pairs of symbols with their merged version. 'regexprep' is a matlab function that is used to perform search and replace operation.

```
vocab_out(new_word) = vocab_in(word);
```

This line updates the new vocabulary 'vocab_out' with the transformed word 'new_word' and assigns the same frequency as the original word in 'vocab_in'.

d) `function byte_pair_encoding(data, n)`

This is the main function that performs our algorithm when it is called from the main script. It processes the given dataset, applies BPE algorithm, and returns the updated vocabulary after merging the most frequent pairs.

```
vocab = get_vocab(data);
```

We call the 'get_vocab' function. 'vocab' stores the initial vocabulary generated from the input data.

```
for i = 1:n
```

This loop iterates 'n' times. 'n' is the number of merges to be executed.

```
pairs = get_stats(vocab);
```

We call the 'get_stats' function. 'pairs' stores the pairs of characters as keys and their frequency as values.

```
if isempty(pairs)
    disp('No more pairs to merge');
    break;
end
```

This loop checks if any character pairs are found and displays the the print statement when none are found.

```
pairKeys = keys(pairs);
```

'pairKeys' stores all of the keys from 'pairs'. 'keys' is a matlab function that retrieves the keys from the pairs dictionary.

```
pairValues = cell2mat(values(pairs));
```

Values from 'pairs' are stored in 'pairValues'. 'cell2mat' coverts values from cell format to a numeric matrix.

```
[~, bestIdx] = max(pairValues);
```

The 'max' function returns the maximum value in 'pairValues'. The frequency of the maximum value is ignored using '~' and only the index of the maximum value is stored using 'bestIdx'.

```
best = pairKeys{bestIdx};
```

Using 'bestIdx', this line retrieves the corresponding pair from 'pairkeys'. 'best' is now the most frequent pair.

```
vocab = merge_vocab(best, vocab);
```

We call the merge vocab function. This function merges the most frequent 'best' pair in all the words of the vocabulary.

```
disp('Final Vocabulary after all merges:');

vocabKeys = keys(vocab);        % Extracting the final tokens

vocabValues = values(vocab);
```

The statement within disp is printed and the lists of keys and values from the vocabulary are extracted and stored.

```
for i = 1:length(vocabKeys)

    fprintf('%s -> %d\n', vocabKeys{i}, vocabValues{i});

end
```

This loop runs through all the keys in the final vocabulary and the output is printed.

Main Script.

```
corpus = ['In this code we are implementing the byte pair encoding tokenization. BPE merges the most frequent pairs. It is a preferred tokenization method in many language models'];

data = strsplit(corpus, '.');  % Split the text into sentences

n = 150;  % Perform 150 BPE merges

vocab = byte_pair_encoding(data, n);

disp(vocab);
```

Our main script is pretty straightforward and simple. The input text in 'corpus' is split into sentences using 'strsplit' and stored in 'data'. 'n' denotes the number of times we perform the merging of the most frequent pairs. Finally, we call the 'byte_pair_encoding' function and display the final vocabulary.

4.3 Experimental Results

In the following section we will see how our code is executing the Byte Pair Encoding tokenization method on out given corpus.

I will not go into the theoretical details of how the merging is performed, as it is already discussed in a previous chapter. After 10 merges or 10 iterations of our code, the vocabulary takes the following form:

```
Final Vocabulary after 10 merges:
 </w> -> 2
B P E </w> -> 1
I n</w> -> 1
I t</w> -> 1
a </w> -> 1
a re</w> -> 1
b y t e</w> -> 1
c od e</w> -> 1
en c od i ng </w> -> 1
f re q u en t</w> -> 1
i m p l e m en ti ng </w> -> 1
i n</w> -> 1
i s</w> -> 1
```

l a ng u a g e</w> -> 1

m a n y </w> -> 1

m e r g e s</w> -> 1

m e th od </w> -> 1

m o s t</w> -> 1

m od e l s</w> -> 1

p a i r </w> -> 1

p a i r s</w> -> 1

p re f e r re d </w> -> 1

t o k en i z a ti o n</w> -> 2

th e</w> -> 2

th i s</w> -> 1

w e</w> -> 1

It can be observed that certain pairs with high frequencies have been merged already. Pairs such as “re”, “od”, “th” have been merged at the end of 10 iterations. The pairs “od” and “th” appear in many words of the corpus, so they must have had a high frequency which led to their early merging. Other pairs with a lower frequency will follow in the subsequent iterations. It must also be noted that pairing is not restricted to single characters but a single character (or multiple characters) can also be paired to other single or multiple characters. For example, “a” can be paired with “re” if this is the pair with highest frequency.

After 50 iterations of our code, that is, after 50 merges, our vocabulary takes the following form:

Final Vocabulary after 50 merges:

</w> -> 2

BPE</w> -> 1

In</w> -> 1

It</w> -> 1

a</w> -> 1

are</w> -> 1

byte</w> -> 1

code</w> -> 1

en coding</w> -> 1

f re q u en t</w> -> 1

i m p l e m en ti ng</w> -> 1

i n</w> -> 1

is</w> -> 1

l anguage</w> -> 1

m any</w> -> 1

m e th od </w> -> 1

m er g e s</w> -> 1

m o s t</w> -> 1

m od e l s</w> -> 1

p re f er re d</w> -> 1

pair </w> -> 1

pair s</w> -> 1

th is</w> -> 1

the</w> -> 2

tokenization</w> -> 2

w e</w> -> 1

It can be observed that at the end of 50 merges, many more characters have been paired and all the characters of some words such as “in”, “it”, “tokenization” have been paired to form full words. Therefore, at the end of 50 iterations, we can observe some full words in the vocabulary. We can stop iterating further if this is the vocabulary that we desire or merging can be continued until all possible pairs have been merged.

After 100 iterations of our code, that is after 100 merges, our vocabulary takes the following form:

No more pairs to merge

Final Vocabulary after all merges:

</w> -> 2

BPE</w> -> 1

In</w> -> 1

```
It</w> -> 1
a</w> -> 1
are</w> -> 1
byte</w> -> 1
code</w> -> 1
encoding</w> -> 1
frequent</w> -> 1
implementing</w> -> 1
in</w> -> 1
is</w> -> 1
language</w> -> 1
many</w> -> 1
merges</w> -> 1
method</w> -> 1
models</w> -> 1
most</w> -> 1
pair</w> -> 1
pairs</w> -> 1
preferred</w> -> 1
the</w> -> 2
this</w> -> 1
tokenization</w> -> 2
we</w> -> 1
```

It can be observed that there are no more pairs left to merge. All of the pairs have been merged and our vocabulary now includes full words. The numbers corresponding to the words represent their frequency or the number of times they appear in the corpus. The BPE algorithm was therefore, successfully implemented using this code and how the merging takes place over different iterations was also observed. Finally, a vocabulary with full words obtained after merging all the possible pairs was also achieved.

5. EXPLANATION OF POSITIONAL ENCODING IMPLEMENTATION IN MATLAB

The following matlab script implements positional encoding followed by visualization of the same using a graph. I have defined a function "positionalEncoding" which generates the positional encodings using sine and cosine functions.

It processes the input sentence by splitting it into words and determining the sequence. Then, according to the embedding dimension specified by the user, the positional embeddings are generated. These embeddings are then displayed using a plot that compares the embedding vectors of the first and fourth word. This plot showcases how different positions in a sequence are encoded uniquely.

5.1 Positional Embedding Code

```
clear

clc


function pos_enc = positionalEncoding(sequence_length, embedding_dim)

    % Initialize the positional encoding matrix

    pos_enc = zeros(sequence_length, embedding_dim);


    % Loop through each position in the sequence

    for pos = 1:sequence_length

        for i = 1:embedding_dim

            if mod(i, 2) == 0  % Even index (cosine)

                pos_enc(pos, i) = cos(pos / (10^(i / embedding_dim)));

            else  % Odd index (sine)
```

```
            pos_enc(pos, i) = sin(pos / (10^(i / embedding_dim)));
        end
    end
  end
end

% Input statement
input_statement = 'Positional embeddings include the positional information of a word into the model';

% Step 1: Preprocess the input statement (split into words)
words = strsplit(input_statement);

% Step 2: Determine the sequence length
sequence_length = length(words);  % Number of words in the input statement

% Step 3: Define the embedding dimension
embedding_dim = 128;  % For example, 64-dimensional positional embeddings

% Step 4: Generate positional embeddings based on sequence length and embedding dimension
positional_embeddings = positionalEncoding(sequence_length, embedding_dim);

% Display the positional embeddings
disp(positional_embeddings)

%% Heat map
imagesc(positional_embeddings);
```

```
colorbar;

title('Positional Embeddings Heatmap');

xlabel('Embedding Dimensions');

ylabel('Word Position');



%% Graph

first_word_embedding = positional_embeddings(1, :);

%disp('Positional embedding vector for the 1st word:');

% disp(first_word_embedding);



fourth_word_embedding = positional_embeddings(4, :);

%disp('Positional embedding vector for the 4th word:');

%disp(fourth_word_embedding);



figure;

plot(first_word_embedding, 'b', 'LineWidth', 2);

hold on;

plot(fourth_word_embedding, 'r', 'LineWidth', 2);

legend('1st word', '4th word');

xlabel('Embedding Dimension');

ylabel('Value');

title('Comparison of Positional Embeddings for 1st and 4th Words');
```

5.2 Code Explanation

Let us start with the function used.

```
function pos_enc = positionalEncoding(sequence_length, embedding_dim)
```

This function takes in two arguments, 'sequence_length' and 'embedding_dim'. The first one is an integer representing the number of positions in a sequence (words in the sentence). The second is an integer representing the size of the embedding vector for each position. The function outputs a matrix 'pos_enc,' where each row corresponds to the positional embedding of a word.

```
pos_enc = zeros(sequence_length, embedding_dim);
```

A matrix 'pos_enc' is initialized with zeros. The dimensions of this matrix are 'sequence_length' rows by 'embedding_dim' columns. It stores the positional embeddings for each position in the sequence with one row per position.

```
for pos = 1:sequence_length
```

This is the outer loop that iterates over all positions in the sequence. The variable 'pos' represents the position index, ranging from 1 to 'sequence_length'. For each position, we compute the embedding vector.

```
for i = 1:embedding_dim
```

This is the inner loop that iterates over each dimension of the embedding vector. The variable 'i' represents the dimension index. Ranging from 1 to 'embedding_dim'. Each dimension is calculated differently depending on whether 'i' is odd or even.

```
if mod(i, 2) == 0
```

Checks if the current dimension index 'i' is even using the modulo operator (mod). For even dimensions, the embedding value is computed using the cosine function.

```
pos_enc(pos, i) = cos(pos / (10^(i / embedding_dim)));
```

The cosine value is calculated based on the position 'pos' divided by a scaling factor. The scaling factor is determined by the exponential relationship '10^(i / embedding_dim)'.

```
else
```

This branch is executed when the dimension index 'i' is odd. For odd dimensions, the embedding value is computed using the sine function.

```
pos_enc(pos, i) = sin(pos / (10^(i / embedding_dim)));
```

Similar to the even case, the sine value is calculated using the position 'pos' and a scaling factor derived from the dimension index.

```
input_statement = 'Positional embeddings include the positional information of a word into the model';
```

A string variable 'input_statement' is defined, containing a sentence or text input. This serves as the example input that will be processed to extract positional embeddings.

```
words = strsplit(input_statement);
```

The input string is split into individual words using the matlab function ‘strsplit’. The default delimiter is spaces which results in a cell array ‘words.

```
sequence_length = length(words);
```

The ‘length’ function calculates the length of the ‘words’ array. This value is stored in ‘sequence_length’ which represents the total number of words in the input sentence.

```
embedding_dim = 1024;
```

The variable ‘embedding_dim’ can be set to any value of your choice and is currently set to 1024. This variable specifies the dimensionality of the positional embeddings. Each word or position in the sequence will be represented by 1024 dimensions.

```
disp(positional_embeddings)
```

This ‘disp’ function displays the ‘positional_embeddings’ matrix in the command window. The generated embeddings can be verified here and if not appropriate, necessary changes can be made by changing the embedding dimension or the scaling factor. The scaling factor of 10000, from the original transformers paper, does not produce desirable encodings. Instead a factor of 10 was found to be more effective in producing accurate and usable encodings.

Now let’s look at the code that was used to create the plot that compares the embeddings of two different positions.

```
first_word_embedding = positional_embeddings(1, :);
```

The positional embedding for the first word is extracted from the "positional_embeddings" matrix. This is done by selecting the first row (corresponding to the first word) and all columns (embedding dimensions).

```
fourth_word_embedding = positional_embeddings(4, :);
```

Similarly, the positional embedding for the fourth word is extracted by selecting the fourth row. This is useful for comparing embeddings of words at different positions in the sequence.

```
figure;
```

A new figure window is created for plotting. This ensures the plot is displayed in a separate window, making it easier to analyze.

```
plot(first_word_embedding, 'b', 'LineWidth', 2);
```

The positional embedding of the first word is plotted as a blue line. The 'b' specifies the color (blue), and 'LineWidth' adjusts the thickness of the line for better visibility.

```
hold on;
```

The 'hold on' command allows multiple plots to be drawn on the same figure without overwriting the existing plot.

```
plot(fourth_word_embedding, 'r', 'LineWidth', 2);
```

The positional embedding of the fourth word is plotted as a red line. The 'r' specifies the color (red), and 'LineWidth' ensures the line is clearly visible.

```
legend('1st word', '4th word');
```

A legend is added to the plot to differentiate between the two lines. The labels '1st word' and '4th word' indicate which line corresponds to which word's embedding.

```
xlabel('Embedding Dimension');
```

The x-axis is labeled 'Embedding Dimension'. This indicates that the horizontal axis represents the dimensions of the embedding vector.

```
ylabel('Value');
```

The y-axis is labeled 'Value'. This indicates that the vertical axis represents the numerical values of the embedding components.

```
title('Comparison of Positional Embeddings for 1st and 4th Words');
```

A title is added to the plot, providing a concise description of its content. This makes the plot easier to interpret and understand.

The following lines of code are used to create a heat map that shows the relationship between embedding dimensions and word positions.

```
imagesc(positional_embeddings);
```

The 'imagesc' function generates a heat map visualization of the 'positional_embeddings' matrix. Each cell in the heat map corresponds to a value in the matrix, with color intensity representing the magnitude of the value. The x-axis represents embedding dimensions, and the y-axis represents word positions.

```
colorbar;
```

The ‘colorbar’ function adds a color scale to the heat map. This scale provides a reference for interpreting the magnitude of the values represented by the colors. For instance, darker colors represent smaller values, and brighter colors represent larger values.

```
title('Positional Embeddings Heatmap');
```

The ‘title’ function adds a title to the heat map. The title ‘Positional Embeddings Heatmap’ provides a clear and concise description of the plot, making it easier to understand.

```
xlabel('Embedding Dimensions');
```

The ‘xlabel’ function labels the x-axis as ‘Embedding Dimensions’. This clarifies that the horizontal axis corresponds to the different dimensions of the positional embedding vectors.

```
ylabel('Word Position');
```

The ‘ylabel’ function labels the y-axis as ‘Word Position’. This clarifies that the vertical axis corresponds to the positions of words or tokens in the sequence.

5.3 Experimental Results

In this section we will take a look at the positional embeddings created by our code. We were able to encode each position (word) in the input sequence with a unique positional encoding using absolute sinusoidal encoding method as discussed in the original transformers paper.

The positional embeddings matrix is shown below. Note that the rows represent each position (word) in the input sequence and the columns represent the number of embedding dimension. The following rows and columns are extracted from a much larger positional embedding matrix.

0.8390	0.5478	0.8342	0.5552	0.8293	0.5625	0.8243	0.5697	0.8194	0.5768
0.9130	−0.3998	0.9201	−0.3835	0.9269	−0.3672	0.9333	−0.3509	0.9394	−0.3347
0.1544	−0.9858	0.1808	−0.9810	0.2068	−0.9756	0.2324	−0.9695	0.2577	−0.9628
−0.7449	−0.6803	−0.7207	−0.7059	−0.6958	−0.7304	−0.6702	−0.7537	−0.6440	−0.7760
−0.9650	0.2405	−0.9758	0.1972	−0.9845	0.1539	−0.9912	0.1107	−0.9960	0.0677
−0.3052	0.9438	−0.3556	0.9249	−0.4047	0.9035	−0.4521	0.8799	−0.4979	0.8541
0.6330	0.7936	0.5835	0.8298	0.5322	0.8626	0.4793	0.8918	0.4252	0.9175
0.9939	−0.0743	0.9992	−0.0034	0.9995	0.0669	0.9948	0.1363	0.9854	0.2043

FIGURE 5 - POSITIONAL EMBEDDING MATRIX

It can be clearly observed that every embedding dimension of each position has been assigned with an unique value. The formula mentioned in the transformers paper for sinusoidal embeddings was used. A scaling factor of 10 was used instead of the original 10000. It was observed that the embeddings produced using 10000 as a scaling factor were repetitive. Using 10 as a scaling factor instead, enabled the production of unique encodings across all dimensions and positions.

We can also observe this using graphs and heat maps.

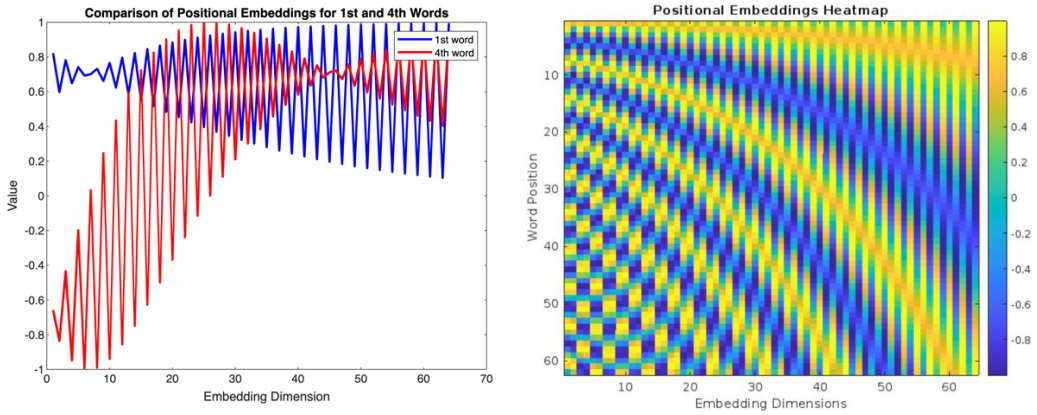

FIGURE 6 - GRAPH & HEAT MAP 62/64

The above images represent a graph and heat map of a sequence length of 62 and embedding dimension of 64. The graph compares the embeddings of 1st and 4th word across 64 dimensions. A sinusoidal pattern along with a unique embedding value for each dimension can be observed.

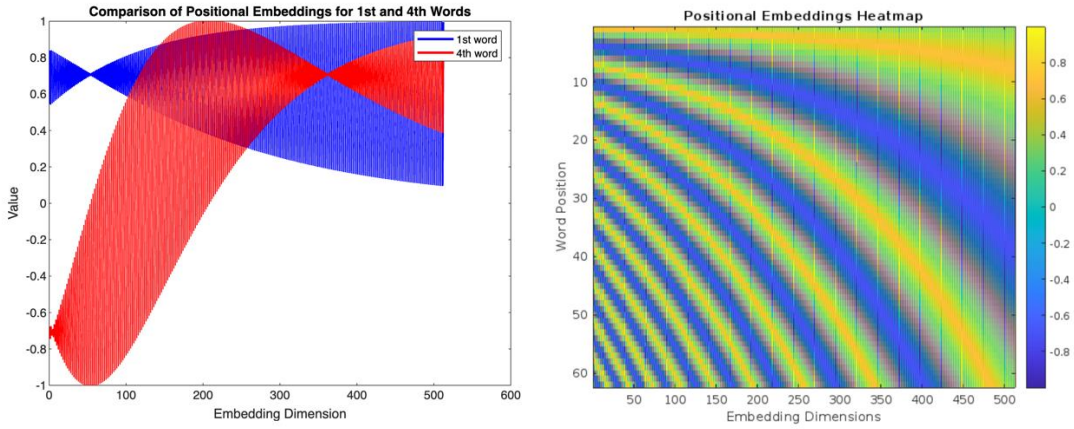

FIGURE 7 - GRAPH & HEAT MAP 62/512

Again, in the above images, a graph and a heat map of a sequence length of 62 and embedding dimension of 512 can be seen. A similar sinusoidal pattern, as compared to the previous graph can be observed.

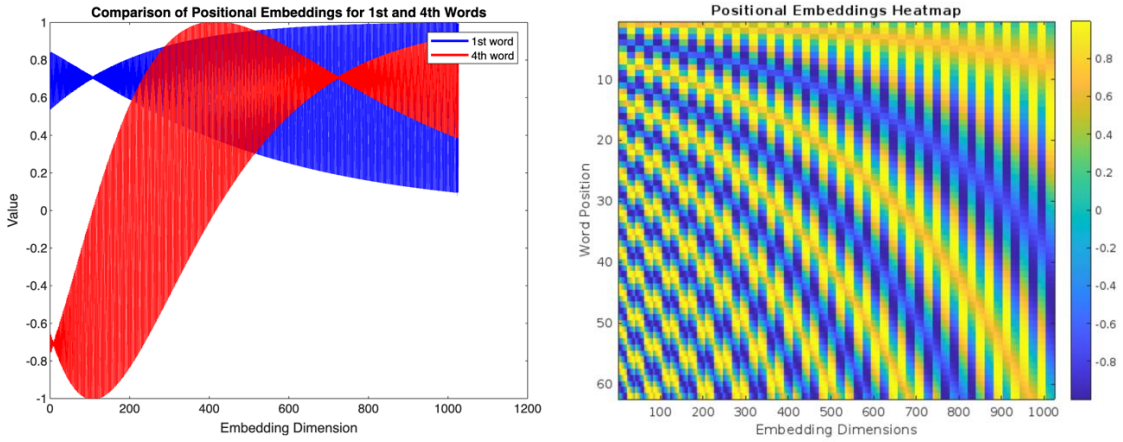

FIGURE 8 - GRAPH & HEAT MAP 62/1024

A graph and a heat map of a sequence length of 62 and embedding dimension of 512 are seen in the above images. All of the heat maps included in this chapter demonstrate similar behaviour regardless of the embedding dimensions. Through this map we can visualize the relationship between embedding dimensions and word positions. The wave like pattern in the map arises from the alternating sine and cosine functions. At lower embedding dimensions, the wave patterns are more tightly packed, due to high frequency oscillations. As the dimensions increase, the waves become more spaced due to the lower frequency oscillations.

This frequency variation ensures that different dimensions capture positional information at varying granularities. Lower embedding dimensions oscillate more rapidly because of their high frequency. This means their values change significantly for small changes in position. As a result, they can represent small, fine-grained distinctions between nearby positions in a sequence. Higher embedding dimensions oscillate more slowly because of their low frequency. This means their values change gradually over a larger range of positions. As a result, they capture more general or smoother relationships between positions that are farther apart.

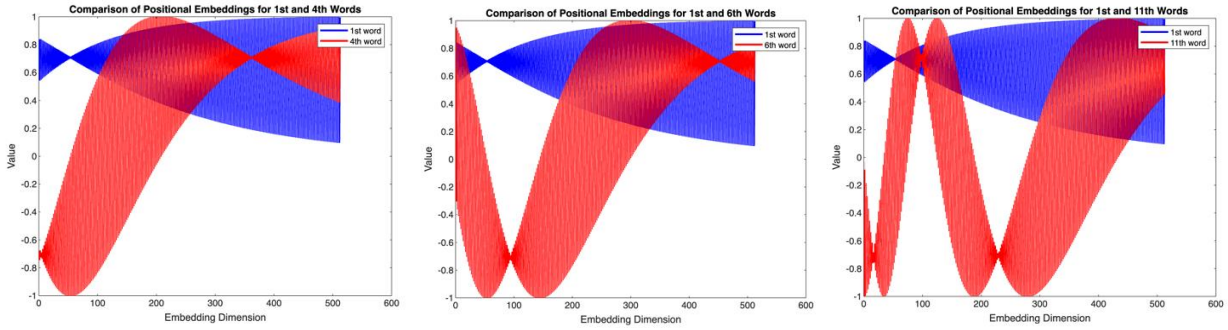

FIGURE 9 – EMBEDDING VALUE COMPARISONS BETWEEN DIFFERENT WORD POSITIONS

In the above images, the embedding values of 4^{th}, 6^{th}, and 11^{th} position are compared with the embedding values of 1^{st} position. A distinct sinusoidal wave is seen in each plot representing unique embedding values for each position.

Thus, a fully functioning positional embedding algorithm was implemented in this section through matlab. Not only were we able to create unique positional encodings, for each position (word), using an embedding vector, but we were also able to visualize these positional embeddings, through plots and maps.

6. FINE TUNING A SMALL LLM TO REPLACE AN INDUSTRY STANDARD LLM

Introduction

Large Language Models demand a significant amount of computation power for their training and inference. Training an LLM can cost millions of dollars depending on its size. Models such as Gemini and GPT-4 have set back their respective companies anywhere from $30 million to over $100 million [38]. Therefore, it is impractical for an average user to create a model from scratch for their own specific use.

Even if we keep the training part aside, using LLMs for inference can also be a costly affair. High quality GPUs are required, which makes running such models locally extremely difficult. These models can be run on cloud, but if the task requires a certain amount of security, downloading them on a local device becomes the only solution. Therefore, instead of using an LLM with a very high number of parameters, we can instead train a much smaller LLM on our specific task.

This process is known as fine tuning. With the rise of models like BERT, GPT, and T5, fine-tuning has enabled users to achieve state-of-art performance across various NLP (Natural Language Processing) tasks such as sentiment analysis, question answering, and text summarization. A fine-tuned model is a pre-trained model trained on a downstream task. The model becomes efficient at solving the problem it was fine-tuned on and, despite its smaller size, can achieve performance comparable to a considerably larger model, if trained correctly.

In this chapter, we will look at the fine-tuning process in detail, and discuss the different types while focusing on parameter efficient methods such as LoRA (Low Rank Adaptation) and quantization. Later, we will fine-tune a small model to match the performance of a significantly bigger industry scale model.

6.1 Fine-Tuning: Concept and Types

Fine-tuning is the process of taking a pre-trained model and adapting it to a specific task by training it on a smaller, task-specific dataset. Unlike training from scratch, fine-tuning leverages the knowledge acquired from large scale corpora and requires fewer resources. There are three main types of LLM fine-tuning methods [39], namely,

a) Unsupervised Fine-tuning

This method eliminates the need for labeled data. The LLM instead processes a large collection of unlabeled text, enhancing its comprehension of language. While beneficial for new domains like law and medicine, it is less accurate for classification and summarization tasks.

b) Supervised Fine-Tuning (SFT)

The LLM is trained with task specific labeled data during SFT. For instance, fine-tuning an LLM for text classification involves a dataset of text snippets paired with class label. Although this approach is effective, it demands a large amount of labeled data, making it both expensive and time consuming.

c) Prompt Engineering (Instruction Fine-Tuning)

The LLM is directly provided with natural language instructions, making it valuable for developing specialized assistants. You can directly prompt the LLM with the guidelines on how you want to receive the outputs. This method minimizes the need for large datasets but its effectiveness depends entirely on the quality of the prompts.

Of the three methods listed above, we will be using supervised fine-tuning (SFT) to fine-tune our model.

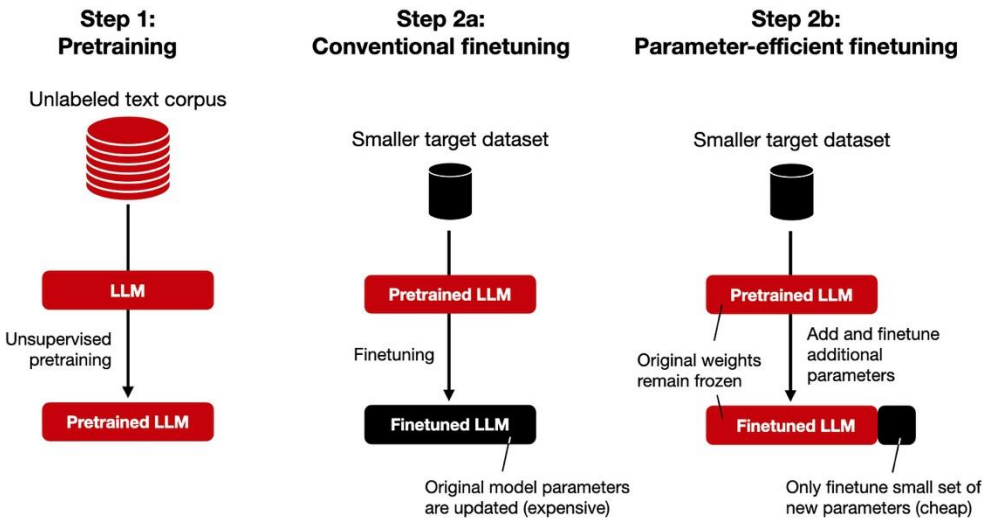

FIGURE 6.1 - TRADITIONAL AND PARAMETER EFFICIENT FINE-TUNING [40]

Of the three methods listed above, we will be using supervised fine-tuning (SFT) to fine-tune our model. In traditional fine-tuning we update every weight and bias of every parameter, which makes it computationally expensive [41]. Instead, we can use a process called parameter-efficient fine tuning, in which only a subset of the parameters are updated. We will look into all of this in subsequent sections.

6.2 Prerequisites to Fine-Tuning

The two main steps that you should carefully consider before fine-tuning are selecting the dataset and choosing the right pre-trained model. Even before that, you must have a solid understanding of the task you plan to train your model on.

a) Task Selection

Transformer models have been successfully applied to various NLP tasks, including text classification, named entity recognition (NER), text summarization, translation, etc. In this study, we will be performing sentiment analysis which involves classifying a given piece of text as positive or negative sentiment. Sentiment analysis is used across various industries to extract insights from textual data.

Two of my favorite examples are financial market analysis and political analysis/opinion mining. Investors can use sentiment analysis to analyze financial news and social media discussions. Thus, stock movement can be predicted based on public sentiment toward a company. Similarly, discussions on online social platforms can also be used for understanding voter sentiment toward a party or a candidate. Other applications include customer service automation, customer feedback and reviews and social media monitoring.

b) Dataset

We have selected the Stanford Sentiment Treebank (SST-2) dataset which is a widely used benchmark for evaluating sentiment classification models. It contains 67,349 phrases extracted from movie reviews, with each phrase labeled as either positive or negative. A well-structured dataset is critical for fine-tuning LLMs effectively. We will preprocess the SST-2 dataset by tokenizing it with the BERT tokenizer and padding/truncating the sequences to ensure correct input lengths. Proper data preprocessing ensures that the model receives structured input, improving training efficiency and generalization performance.

c) Model

For our task, we will be selecting the BERT-base-uncased model. BERT (Bidirectional Encoder Representations from Transformers) is a 12 layer transformer model with 110 million parameters [42].

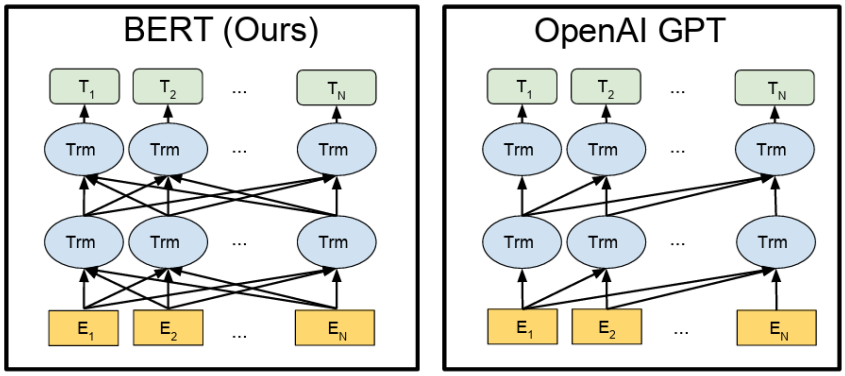

FIGURE 6.2 - BERT VS GPT MODEL ARCHITECTURE [42]

As opposed to GPT, BERT transformer uses bidirectional self-attention [42]. GPT uses constrained self-attention meaning every token can only a attend to context on its left, whereas, tokens in BERT can attend

context in both left and right directions. By fine-tuning BERT on SST-2, we will attempt to increase the performance of BERT on sentiment analysis task.

6.3 Parameter Efficient Fine-Tuning : LoRA and Quantization

Traditional fine-tuning is computationally expensive, requiring significant memory and processing power. Parameter Efficient Fine-Tuning (PEFT) techniques aim to reduce this cost by modifying only a small subset of model parameters while keeping majority of the pre-trained weights frozen. This allows the model to adapt to new tasks efficiently, without needing to update all the parameters.

PEFT methods introduce lightweight trainable components, such as Low-Rank Adaptation (LoRA), to achieve comparable performance with far fewer trainable parameters. Additionally, techniques like quantization help reduce the model's memory footprint, enabling deployment on resource constrained devices.

a) Low-Rank Adaptation (LoRA)

LoRA is a PEFT fine-tuning technique in which we freeze the weights of the pre-trained model and introduce trainable rank decomposition matrices in each layer of the transformer architecture of the model [43]. This significantly reduces the number of trainable parameters for downstream tasks. By using LoRA, we will only be training 50 percent of the models parameters and still achieve a massive increase in performance.

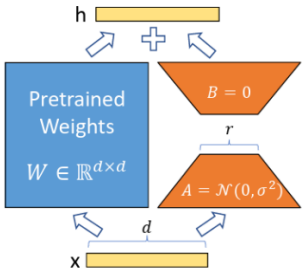

FIGURE 6.10 - LORA (ONLY A AND B IS TRAINED) [43]

In traditional fine-tuning, we update all the weight matrices (Wq, Wk, Wv) of the transformer model which results in high memory usage and computational cost. With LoRA, instead of updating the full weight matrix

W, we freeze it, and introduce two small matrices A and B, which capture task specific adaptations. Since A and B have low rank structures, the number of trainable parameters is significantly reduced.

b) Quantization

Quantization is a technique for optimizing transformer models by reducing the precision of numerical representations. Standard deep learning models use 32-bit floating point precision, but quantization allows for lower precision formats (such as 8 or 4-bit integers). This reduces model size due to the lowering of bit precision and also reduces inference times [44].

In our study, we will use post-training quantization (PTQ) which applies quantization while eliminating the need of retraining the entire pre-trained model weights [44, 45]. This approach reduces memory usage and accelerates inference while preserving accuracy [45]

6.4 The Fine-Tuning Process

In this section, we will fine-tune our model on the previously discussed dataset. The hyperparameters, code, training hardware and all of the other important aspects related to the fine-tuning process will be presented step by step. Our code implementation includes data preprocessing, model loading, evaluation and training. We will use a train-validation split, monitor the loss function, and track accuracy and F1 score during training. Let us begin with the first step, which loading the model and dataset.

a) Installing the Required Libraries

The first step is to install the libraries that are required for our task,

```
pip install transformers -q
```

```
pip install bitsandbytes -q

!pip install datasets -q
```

The transformers library is required for loading and fine-tuning BERT. The bitsandbytes is used for efficient quantization to optimize memory usage and datasets library is used to load the SST-2 dataset, which will be used for sentiment analysis.

b) Setting up Quantization

Before loading the model we must first setup quantization if we wish to use a quantized model.

```
from transformers import BitsAndBytesConfig

bnb_config = BitsAndBytesConfig(

    load_in_4bit=True,

    bnb_4bit_quant_type="nf8",

    bnb_4bit_compute_dtype="float16",

    bnb_4bit_use_double_quant=True,

)
```

By enabling `load_in_4bit=True` , the model's weights are stored in 4-bit format, making fine-tuning feasible on limited hardware. The 'normal float 8' of `bnb_4bit_quant_type="nf8"` is an advanced 4-bit format optimized for better stability with deep learning tasks compared to older formats. `bnb_4bit_compute_dtype="float16",` ensures computations are performed in half-precision. `bnb_4bit_use_double_quant=True` applies quantization twice which is useful for limited ram setups. This should be set to false if you want to avoid the risk of losing accuracy.

c) Loading the Model and Dataset

Now let's load the model, tokenizer and the dataset,

```
from transformers import BertForSequenceClassification

model = BertForSequenceClassification.from_pretrained("google-bert/bert-base-uncased", num_labels=2)



from transformers import BertTokenizer

tokenizer = BertTokenizer.from_pretrained("google-bert/bert-base-uncased")



from datasets import load_dataset

dataset = load_dataset("SetFit/sst2")
```

We load the pre-trained BERT model for sequence classification with number of labels equal to 2. This initialized BERT for binary classification, making it suitable for sentiment analysis where the labels correspond to positive and negative sentiments. Note that we are not quantizing the model. For this study, we had the required memory to perform fine-tuning on an unquantized BERT. We fine-tuned BERT using A100 GPU on google colab pro, therefore quantization was not required.

The appropriate tokenizer for BERT is loaded to convert text into token IDs for processing. The SST-2 dataset is loaded from HuggingFace which contains text and corresponding labels.

d) Preprocessing the Dataset

The next step is to make the dataset ready for processing,

```
def tokenize_function(example):
    return tokenizer(example["text"],
                     padding="max_length",
                     truncation=True,
                     max_length=128)
tokenized_dataset = dataset.map(tokenize_function, batched=True)
```

The above function converts the text into tokens. This function applies padding to ensure that all sequences have the same length (`padding="max_length"`) and truncates longer sequences to 128 token (`max_length=128`). The function is then a applied to the dataset using `dataset.map` which efficiently transforms the text into tokenized inputs suitable for model training.

e) Configuring LoRA for PEFT

We will now define the settings for LoRA and wrap our model with LoRA layers.

```
from peft import get_peft_model
from peft import LoraConfig, TaskType

peft_config = LoraConfig(task_type=TaskType.SEQ_CLS,
                         inference_mode=False,
                         r=16, lora_alpha=32,
                         lora_dropout=0.1)
```

```
lora_model = get_peft_model(model, peft_config)
lora_model.print_trainable_parameters()
```

LoRA is configured using `LoraConfig` , where `TaskType.SEQ_CLS` specifies that the task is sequence classification. The rank `r=16` controls the size of the low rank decompostion matrices. A smaller 'r' results in faster training but can harm the accuracy. A larger 'r' is able to extract more information leading to a better accuracy. The `lora_alpha=32` scales the adapted weights to ensure effective learning. A dropout rate of `lora_dropout=0.1` is used to prevent overfitting. Finally, we wrap our model with LoRA using `get_peft_model(model, peft_config)`. After wrapping, this is the number of available parameters for training,

```
trainable params: 591,362 || all params: 110,075,140 || trainable%: 0.5372
```

f) Calculating Metrics : Accuracy and F1 Score

We will define a function compute metrics which calculate the accuracy and F1 score.

```
from sklearn.metrics import accuracy_score, f1_score
import numpy as np

def compute_metrics(eval_pred):

    logits, labels = eval_pred

    predictions = np.argmax(logits, axis=-1)

    accuracy = accuracy_score(labels, predictions)
```

```
    f1 = f1_score(labels, predictions, average="macro")

    return {"accuracy": accuracy, "f1": f1}
```

This function receives predictions from model and extracts the logits and labels `logits, labels = eval_pred`. Using `np.argmax(logits, axis=-1)` it selects the class with the highest probability as the predicted label. The accuracy is calculated using `accuracy = accuracy_score(labels, predictions)` , while the F1 score is calculated using `f1 = f1_score(labels, predictions, average="macro")`. These metrics will provide insights on how well the model is performing on the sentiment analysis task.

g) Defining Training Arguments

To fine-tune effectively, we define training arguments which specify key training parameters.

```
from transformers import TrainingArguments, Trainer


training_args = TrainingArguments(
    output_dir="sonu/bert-base-uncased/peft-lora",
    learning_rate=3e-4,
    per_device_train_batch_size=64,
    per_device_eval_batch_size=64,
    num_train_epochs=5,
    weight_decay=0.05,
    eval_strategy="steps",
    eval_steps=25,
    save_strategy="steps",
    load_best_model_at_end=True,
```

```
    logging_dir="./logs",
    logging_steps=10,
    optim="paged_adamw_32bit",
    warmup_steps=25,
    lr_scheduler_type="cosine",
    report_to="wandb",
)
```

Majority of the parameters here are selected on trial-and-error basis. There is no right or wrong value, and the correct values are established based on how the model behaves during training. A `learning_rate=3e-4` was chosen as it provided the best accuracy and decently short training time. A smaller learning rate ensures gradual and stable learning but training can be slower. A larger learning rate leads to larger weight updates per step but can cause instability and prevent model from converging.

A larger batch size requires more GPU memory but is faster. A smaller batch size is preferred when less memory is available but the training is considerably slower. I used a large batch size as I had access to a GPU. Model is trained for 5 epochs and a weight decay of 0.5 is chosen to prevent overfitting. Evaluation is set to 25 steps, meaning the models performance is evaluated every 25 steps. The learning rate does not suddenly increase to the specified value. It gradually increases from 0 to 3e-4 in the first 25 steps. After that the learning rate decays following a cosine curve. The AdamW optimizer was used which uses adaptive learning rates and weight decay to prevent overfitting. This optimizer works well with models like BERT. An optimizer is an algorithm that adjusts the model's parameters (weights and biases) during training to minimize the loss function. It does this by calculating gradients (how much each parameter affects the loss) and updating the parameters in a way that reduces the loss over time. The optimizer adjusts the weight in the direction that reduces the loss using a learning rate. Finally, we report the metrics to WandB for visualization.

h) Setting up Trainer and Training the model

We setup trainer

```
trainer = Trainer(
    model=lora_model,
    args=training_args,
    train_dataset=tokenized_dataset["train"],
    eval_dataset=tokenized_dataset["test"],
    tokenizer=tokenizer,
    compute_metrics=compute_metrics,
)


trainer.train()


trainer.evaluate()
```

Using `Trainer` we load the model, tokenizer, training arguments, train and test datasets and the compute metrics function. Finally, using `trainer.train()` we train the model and using `trainer.evaluate()` we evaluate its performance after training.

6.5 Fine-Tuning Results

In this section we will examine if our model was able to converge and how significantly we were able to increase its performance on the task of sentiment analysis. Let us start by looking at the performance of pretrained BERT before fine tuning on the SST-2 dataset.

```
# bert-base-uncased : Pretrained Performance
trainer.evaluate()
```

```
[29/29 00:10]
{'eval_loss': 0.6911117434501648,
 'eval_model_preparation_time': 0.0082,
 'eval_accuracy': 0.5123558484349259,
 'eval_f1': 0.3931804058252952,
 'eval_runtime': 11.2383,
 'eval_samples_per_second': 162.036,
 'eval_steps_per_second': 2.58}
```

FIGURE 11 - BERT PRETRAINED PERFORMANCE

As it can be seen, the performance is incredibly low with the model getting majority of the predictions wrong. If we wanted to use it for sentiment analysis, this model would need fine-tuning.

Therefore, using the settings we discussed in the previous section, we will train the model for 5 epochs. The model is evaluated every 25 steps and the training loss, validation loss, accuracy and F1 score over 5 epochs can be seen in the following figure.

[545/545 13:29, Epoch 5/5]

Step	Training Loss	Validation Loss	Accuracy	F1
25	0.699300	0.670744	0.656233	0.655214
50	0.434100	0.370269	0.842394	0.841229
75	0.381800	0.292687	0.876991	0.876824
100	0.315300	0.274022	0.887424	0.887196
125	0.226700	0.253867	0.897858	0.897839
150	0.296400	0.258429	0.892916	0.892864
175	0.276800	0.249676	0.899506	0.899466
200	0.307000	0.230128	0.908841	0.908775
225	0.256400	0.226605	0.911587	0.911587
250	0.214200	0.237235	0.905546	0.905515
275	0.257500	0.227603	0.910489	0.910480
300	0.268600	0.237177	0.902252	0.902186
325	0.212900	0.243209	0.901153	0.901063
350	0.222100	0.226948	0.909390	0.909388
375	0.185900	0.227775	0.910489	0.910476
400	0.162200	0.233456	0.909940	0.909920
425	0.162400	0.239464	0.907194	0.907129
450	0.199000	0.224904	0.912685	0.912675
475	0.201200	0.231697	0.909940	0.909917
500	0.185900	0.229556	0.910489	0.910470
525	0.198400	0.230690	0.909390	0.909369

FIGURE 12 - BERT TRAINING FOR 5 EPOCHS

We start off with a training and validation loss of almost 0.7. Both of the losses show gradual and consistent drop over the 5 epochs or 525 steps. After around 250 steps, we see a training loss of 0.21 and validation loss of 0.23. After this point we don't see a significant drop in either losses. The accuracy and f1 scores have also reached 0.9 by this point. We still continue to check is the losses can drop further. As validation loss is not increasing, which means there is no risk of over fitting and hence, we continue this process for 275 more steps. The sweet-point lies between 375 and 400 steps where the training loss has dropped to 0.16 and the accuracy has reached 0.91. By the end of our training we see significant improvement is both the losses and the metrics.

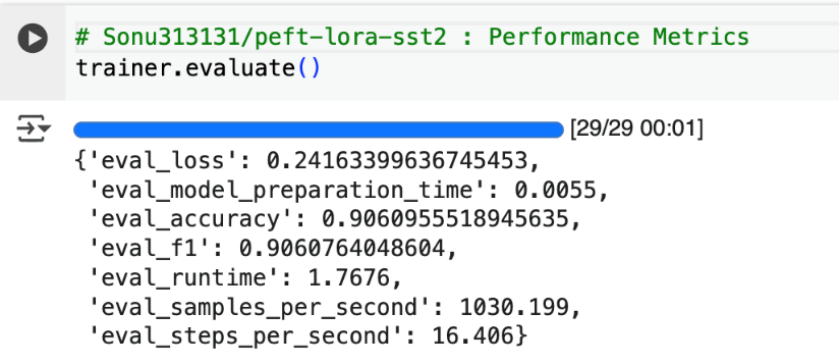

FIGURE 13 - BERT FINE-TUNED PERFORMANCE

Finally, after 5 epochs, evaluating the model gave these results. We are able to improve both the accuracy and f1 scores significantly. Accuracy went up from 0.51 to 0.91 and similarly, f1 score shot up from 0.39 to 0.91. This shows a 78% increase in the accuracy and 133% increase in the f1 score. Therefore, it is safe to say that the fine-tuning process turned out to be a huge success. We can also visualize these improvements with the following graphs which were acquired from Wandb.

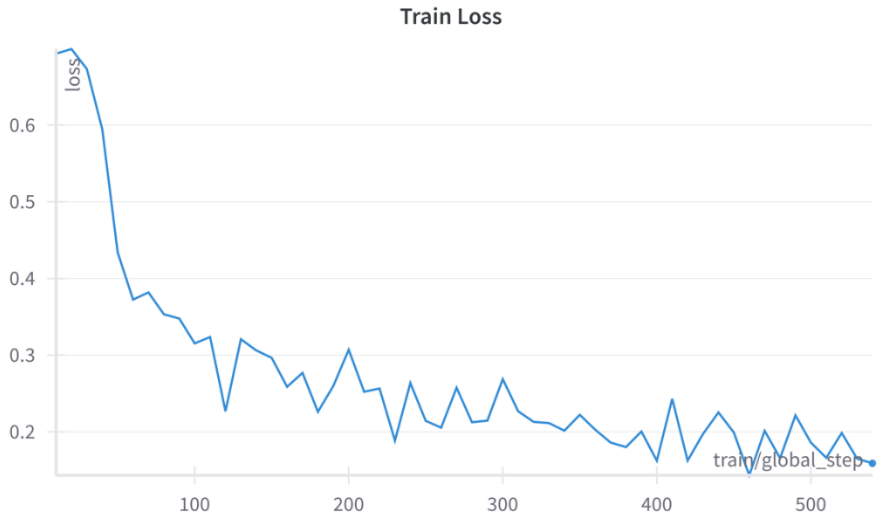

FIGURE 14 - TRAINING LOSS

It can be clearly seen that the loss decreases consistently over 5 epochs or 525 steps. This shows that the model was able to converge. The loss initially decreases sharply and from around step 150, follows a slow decline rate. Lowest value of 0.16 is reached between 400 and 450 steps.

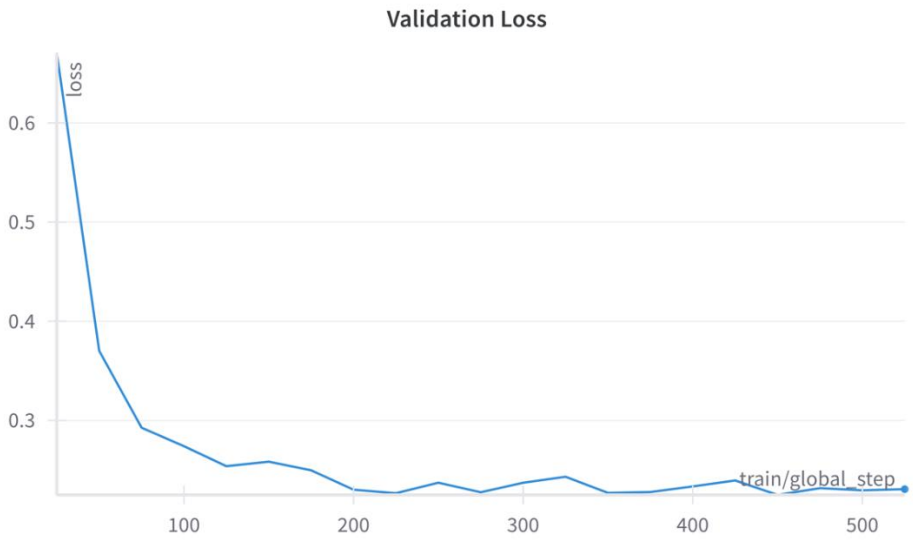

FIGURE 15 - VALIDATION LOSS

The validation loss also consistently decreases overtime. It also does not, at any point show a considerable increase. Through this we can conclude that through our model training strategy, we were successful in avoiding overfitting.

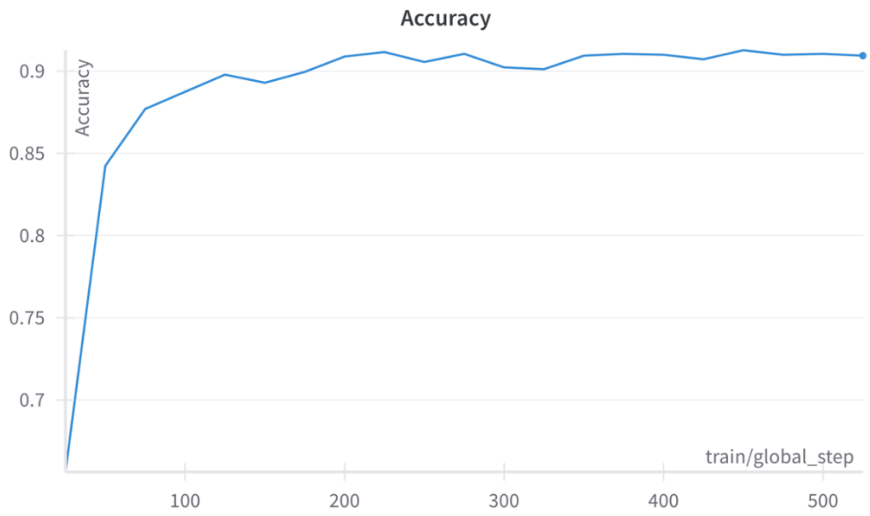

FIGURE 16 - ACCURACY

The accuracy showed a sharp rise in the first 100 steps and continued to increase slowly thereafter. A final accuracy of 0.91 was achieved by the model which is also reflected in the graph.

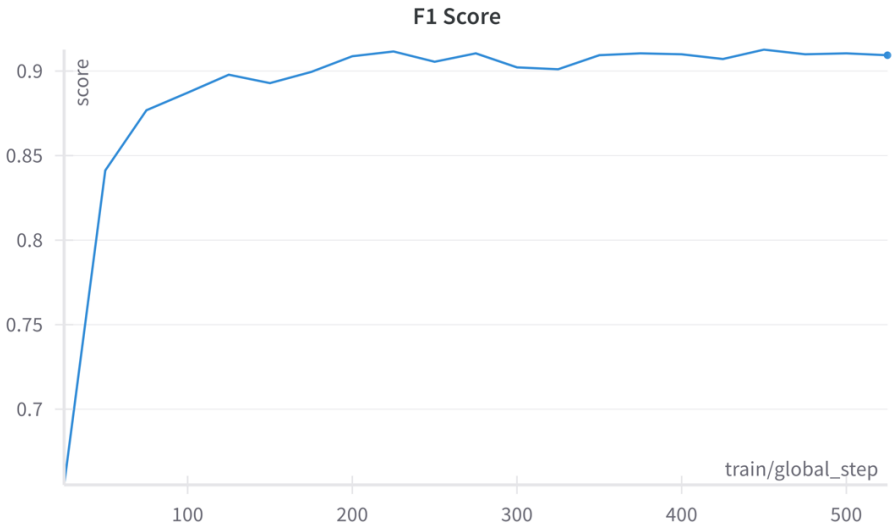

FIGURE 17 - F1 SCORE

The F1 score also mimicked the behavior of accuracy and showed quick rise in the first 100 steps. The growth slowed down after that and the highest F1 score of 0.91 was achieved by the end of the training.

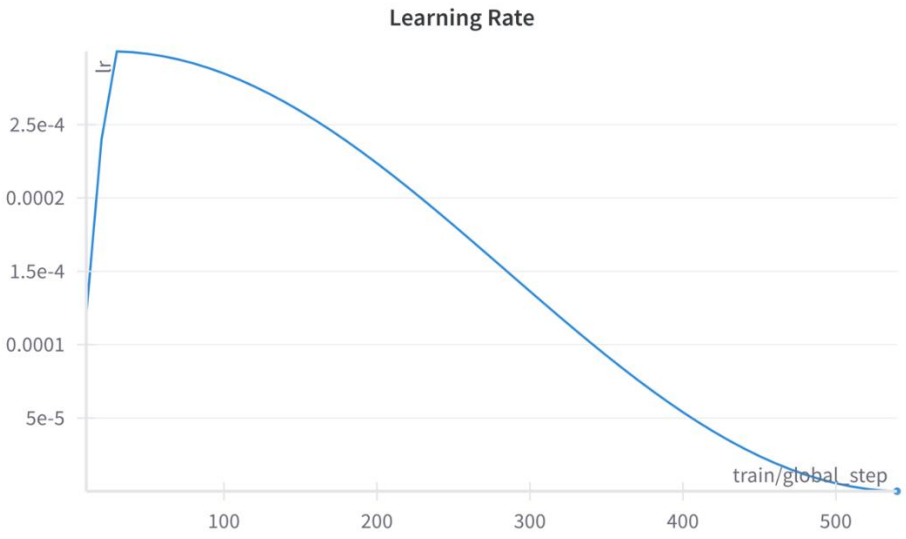

FIGURE 18 - LEARNING RATE

The learning rate followed a path according to the settings in our training arguments. Initially, for the first 25 steps, learning rate increased from 0 to the specified learning rate, as we had set warmup_steps=25. Thereafter, the learning rate followed a decay in the form of a cosine wave. This was set by lr_scheduler_type="cosine".

6.6 Performance Comparison with Larger Models

In this section, we will compare the performance of our fine-tuned model with other well-known industry scale models. To make a direct comparison of the performance, we will evaluate these models on the same dataset and try to compute the accuracy and f1 scores.

The models that I compared our fine-tuned model with are Mistral-7B-v0.3, GPT-2-large and Falcon-7B-instruct. All of these models are 7 to 60 times larger than our model, therefore a considerably higher performance is expected. The accuracy and f1 scores of the models after evaluation are in the graph below.

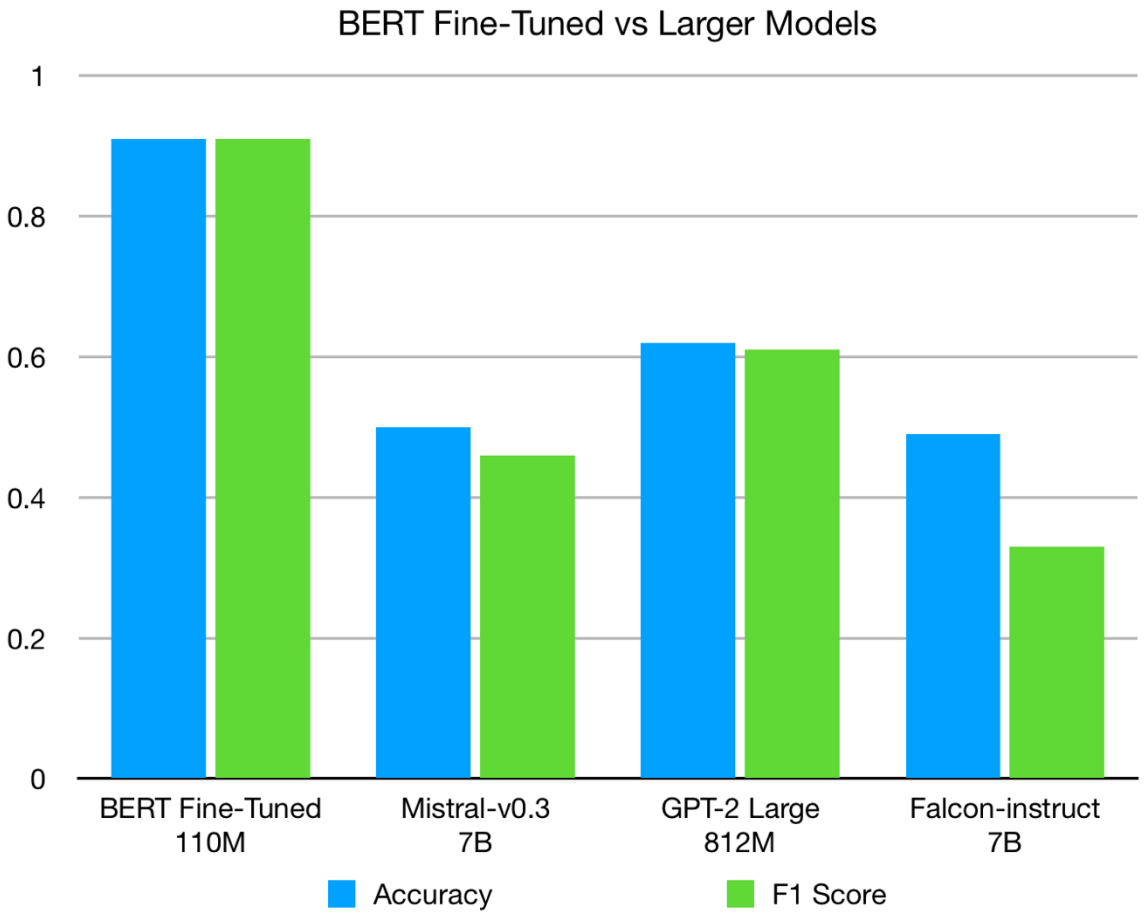

FIGURE 19 - BERT FINE-TUNED VS LARGER MODELS

It can be clearly seen that none of the models were able to outperform our model despite their larger sizes. Both Mistral and Falcon were able to achieve accuracies of only 50% meaning half of their predictions were incorrect. The f1 scores were below 0.5 for both the models.

```
[19] # mistralai/Mistral-7B-v0.3 : Performance Metrics
     trainer.evaluate()
```

```
[29/29 01:51]
{'eval_loss': 4.87389612197876,
 'eval_model_preparation_time': 0.0051,
 'eval_accuracy': 0.500274574409665,
 'eval_f1': 0.4688395893059537,
 'eval_runtime': 115.0939,
 'eval_samples_per_second': 15.822,
 'eval_steps_per_second': 0.252}
```

FIGURE 20 - MISTRAL PERFORMANCE ON SST-2

```
# tiiuae/falcon-7b-instruct : Performance Metrics
trainer.evaluate()
```

```
[29/29 01:28]
{'eval_loss': 3.8714394569396973,
 'eval_model_preparation_time': 0.0042,
 'eval_accuracy': 0.49917627677100496,
 'eval_f1': 0.33296703296703295,
 'eval_runtime': 91.7442,
 'eval_samples_per_second': 19.849,
 'eval_steps_per_second': 0.316}
```

FIGURE 21 – FALCON PERFORMANCE ON SST-2

```
1 # openai-community/gpt2-large : Performance Metrics
2 trainer.evaluate()
```

```
[29/29 00:46]
{'eval_loss': 0.6495855450630188,
 'eval_model_preparation_time': 0.0244,
 'eval_accuracy': 0.627677100494234,
 'eval_f1': 0.6120925620582528,
 'eval_runtime': 48.2017,
 'eval_samples_per_second': 37.779,
 'eval_steps_per_second': 0.602}
```

FIGURE 22 – GPT-2 (LARGE) PERFORMANCE ON SST-2

Of the 3 models that we evaluated, only GPT-2 (large) was able to provide a somewhat satisfactory result. Both the accuracy and f1 scores of this model were over 0.6 but still considerably lower than our model.

I was not ready to stop here and was motivated to keep searching for larger models who our model is capable of competing with. Based on these findings, I thought it would be appropriate to evaluate large GPT models which are industry scale models used by millions of users on a daily basis.

The larger GPT models, however, are closed source. Hence, to evaluate them, we have to make api calls to the OpenAI API. Using a python script, we send sentences to OpenAI and ask the model to classify the sentence as positive or negative. We then store these predictions in an empty list and at the end calculate the accuracy and f1 scores. The script that was used to evalute the performance is as follows,

```
import openai
import datasets
import time
from sklearn.metrics import accuracy_score, f1_score

# ✅ Load SST-2 dataset from Hugging Face
sst2 = datasets.load_dataset("SetFit/sst2", split= "test")


# ✅ Initialize OpenAI API
client = openai.OpenAI(api_key="api_key")
```

We import `openai` to interact with OpenAI's API for text classification. `time` is used for adding delays to avoid hitting API rate limits. We then load the dataset and create and OpenAI API client using our API key.

```
# ✅ Define a function to classify sentiment
def classify_sentiment(sentence, model="gpt-4o-mini"):
    """Sends a sentence to GPT-4o-mini (or GPT-3.5) and returns the
predicted sentiment."""
    try:
        response = client.chat.completions.create(
            model=model,
            messages=[
                {"role": "system", "content": "You are a helpful assistant
trained to classify sentiment."},
                {"role": "user", "content": f"Classify the sentiment of
the following sentence as either 'positive' or 'negative':\n\nSentence:
'{sentence}'\nSentiment:"}
            ],
            max_tokens=10,
            temperature=0  # Deterministic output
        )
        prediction = response.choices[0].message.content.strip().lower()
        return 1 if "positive" in prediction else 0  # Map text response
to label
    except Exception as e:
        print(f"Error: {e}")
        return None  # Handle API failures gracefully
```

We define `classify_sentiment` function to send a sentence to OpenAI's GPT model and extract response.

We first create a chat completion request using `client.chat.completions.create` method. First message, with the role "system", sets the assistant's behaviour by instructing to classify the sentiment. The second message, with the role "user", provides the input sentence and asks the model to classify it as either "positive" or "negative". `max_tokens=10` ensures the model's response remains short. The temperature is set to zero to make the model give deterministic outputs.

`response.choices[0].message.content` retrives the text output from the first response choice. The function then returns 1 if the response contains the word "positive" and 0 if "negative".

The `except Exception as e` block is triggered if any error occurs during API request. The error details are displayed using `print(f"Error: {e}")` and instead of crashing the program, it gracefully returns "None" to indicate that the sentiment classification of that sentence was unsuccessful.

```
# Evaluate the model on the SST-2 dataset
true_labels = []
predicted_labels = []

for i, example in enumerate(sst2):
    sentence = example["text"]
    true_label = example["label"]  # 1 = Positive, 0 = Negative

    prediction = classify_sentiment(sentence, model="gpt-4o-mini")

    if prediction is not None:
        true_labels.append(true_label)
        predicted_labels.append(prediction)

    # Print progress every 10 samples
    if (i + 1) % 10 == 0:
        print(f"Processed {i+1}/{len(sst2)} sentences...")

    # Avoid hitting OpenAI rate limits (adjust as needed)
    time.sleep(0.5)
```

We store the ground truth labels and model predictions, and loop through each sentence in the dataset using `for i, example in enumerate(sst2)`. We then extract sentence and label, call the function and add the true and predicted labels to their respective list if the prediction is valid. The progress is printed every 10 sentences and a delay of 0.5 seconds is added between every request so we don't hit OpenAI's rate limits.

```
# Compute Accuracy and F1-score
accuracy = accuracy_score(true_labels, predicted_labels)
f1 = f1_score(true_labels, predicted_labels)

print("\n GPT-4o-mini Sentiment Analysis Results on SST-2:")
print(f" Accuracy: {accuracy:.4f}")
print(f" F1 Score: {f1:.4f}")
```

The accuracy and f1 score are then calculated and printed. I compared the performance of our model with GPT-3.5 and GPT-4o-mini. The results are as follows,

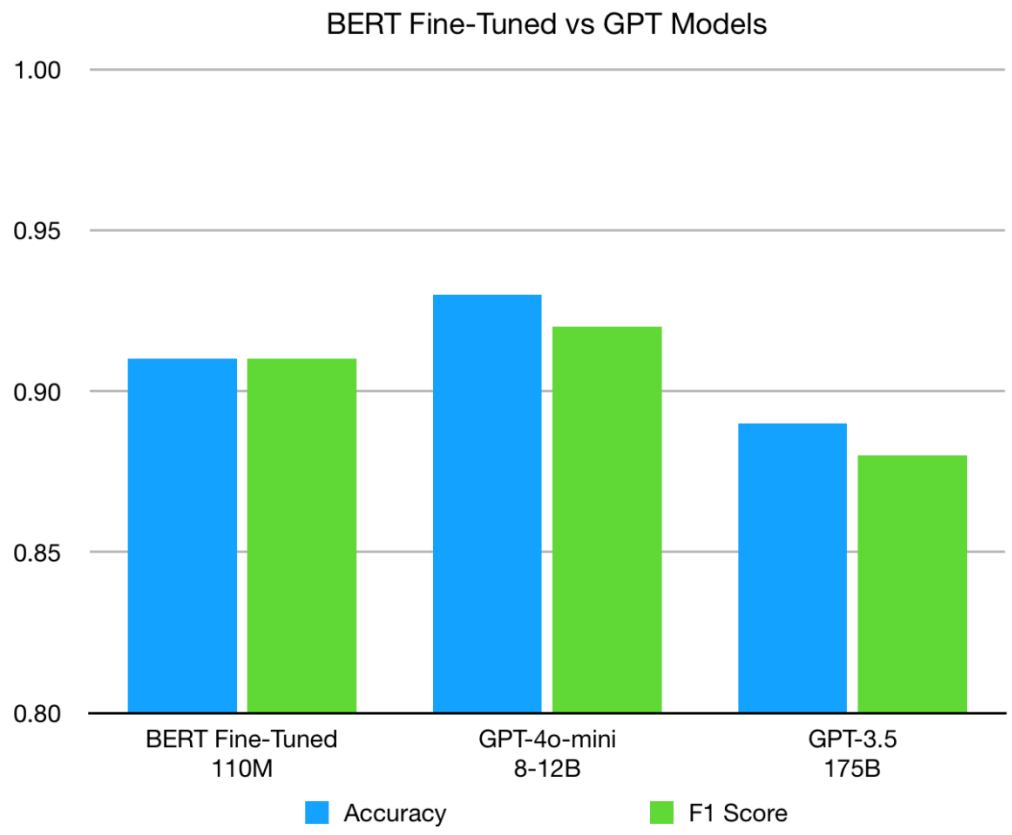

FIGURE 23 - BERT FINE-TUNED VS GPT-4O-MINI & GPT-3.5

Finally, the GPT models were able to provide a comparable performance to our model. GPT-3.5 fell just short whereas GPT-4o-mini was able to outperform us by a tiny margin.

```
GPT-4o-mini Sentiment Analysis Results on SST-2:
Accuracy: 0.9314
F1 Score: 0.9270
```

FIGURE 24 - GPT-4O-MINI PERFORMANCE

```
GPT-4 Sentiment Analysis Results on SST-2:
Accuracy: 0.8935
F1 Score: 0.8827
```

FIGURE 25 - GPT-3.5 PERFORMANCE

In figure 18, the model is incorrectly named as GPT-4, while it is GPT-3.5. GPT 3.5 which is over 1500 times bigger than our model achieved an accuracy of 0.89 and f1 score of 0.88. Although the scores are lower than our model, they are pretty impressive. GPT-4o-mini was able to do better than us with accuracy of 0.93 and f1 score of 0.92.

While we compare the performances, we must not forget that the models that we are comparing our model with are industry scale models used by people around the world on a daily basis. They are trained on world-class data with millions of dollars being spent on the training. The fact that our model was able to compete with them is a huge achievement considering the limited resources that were available to us for fine-tuning. Through the chapter, not only were we able to obtain theoretical knowledge about fine-tuning but were also able to implement this knowledge to make a model perform the task of sentiment analysis at a world-class level of performance. This model is now freely available for use on HuggingFace at Sonu313131/peft-lora-sst. Through this study, we were able to contribute to the field of natural language processing (NLP) and machine learning (ML) by providing a model that despite its smaller size, produces high tier industry standard performance on the task of sentiment analysis.

CONCLUSION

The rapid advancement of Large Language Models (LLMs) has revolutionized the field of Natural Language Processing (NLP), enabling powerful applications across industries. However, the reliance on massive, computationally expensive models poses significant challenges in terms of accessibility, cost and efficiency. This thesis has systematically explored the key components of LLMs – tokenization, embeddings, and fine-tuning.

A crucial aspect of this result was the in-depth analysis of tokenization and embeddings, the foundational blocks of modern LLMs. We examined various tokenization techniques including Byte Pair Encoding (BPE), WordPiece, and Unigram, and implemented these algorithms in code to provide practical insights into their workings. Similarly, embeddings such as token, contextual, multimodal, and positional were also thoroughly explored. The code implementation demonstrated their practical use, offering a comprehensive understanding of how these techniques power modern LLMs.

Through rigorous experimentations, we have shown that a smaller, properly fine-tuned LLM can match the performance of much larger industry-scale models. By leveraging fine-tuning techniques such as LoRA and quantization, we have been able to significantly enhance the capabilities of a compact model, making it a viable alternative to heavyweight models for sentiment analysis. This achievement is more than just a technical optimization – it is a paradigm shift in how we approach model deployment, proving size is not the ultimate determinant of effectiveness.

The implications of this work are profound. A well-tuned smaller model not only reduces inference costs but also democratizes AI by enabling high-performance NLP applications on local machines, without the need for extensive computational infrastructure. This approach opens the door for more sustainable and cost-effective AI solutions.

Looking ahead, the future of AI lies in striking the perfect balance between efficiency and power. As fine-tuning techniques continue to evolve, the gap between smaller and larger models will continue to shrink, making AI more accessible than ever before. This thesis serves as a testament to the potential of fine tuning – not just as a means to improve models, but as a revolutionary approach to redefining the scale at which AI operates.

References –

1. Vaswani A, Shazeer N, Parmar N, Uszkoreit J, Jones L, Gomez AN, et al. Attention is all you need. arXiv.org. 2017. Available from: https://arxiv.org/abs/1706.03762

2. Jurafsky D, James M. Stanford. Large Language Models [Internet]. [cited 2025 Feb 11]. Available from: https://web.stanford.edu/~jurafsky/slp3/10.pdf

3. Tiktokenizer [Internet]. [cited 2025 Feb 11]. Available from: https://tiktokenizer.vercel.app/?model=gpt-4

4. OpenAI. New and improved embedding model [Internet]. [cited 2025 Feb 11]. Available from: https://openai.com/index/new-and-improved-embedding-model/

5. Amazon Web Services, Inc. What is Sentiment Analysis? - Sentiment Analysis Explained – AWS [Internet]. [cited 2025 Feb 11]. Available from: https://aws.amazon.com/what-is/sentiment-analysis/

6. Hugging Face NLP Course. Transformers, what can they do? [Internet]. [cited 2025 Feb 11]. Available from: https://huggingface.co/learn/nlp-course/en/chapter1/3

11) Rush A. Harvard SEAS. The Annotated Transformer [Internet]. [cited 2025 Feb 11]. Available from: https://nlp.seas.harvard.edu/2018/04/03/attention.html

12) Shaw P, Uszkoreit J, Vaswani A. Self-Attention with Relative Position Representations. arXiv.org. 2018. Available from: https://arxiv.org/abs/1803.02155

13) Dai Z, Yang Z, Yang Y, Carbonell J, Le QV, Salakhutdinov R. Transformer-XL: Attentive Language Models Beyond a Fixed-Length Context. arXiv.org. 2019. Available from: https://arxiv.org/abs/1901.02860

14) Raffel C, Shazeer N, Roberts A, Lee K, Narang S, Matena M, et al. Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer. Journal of Machine Learning Research. 2020;21(140):1–67. Available from: http://jmlr.org/papers/v21/20-074.html

15) He P, Liu X, Gao J, Chen W. DeBERTa: Decoding-enhanced BERT with Disentangled Attention. arXiv.org. 2020. Available from: https://arxiv.org/abs/2006.03654

16) Press O, Smith NA, Lewis M. Train Short, Test Long: Attention with Linear Biases Enables Input Length Extrapolation. arXiv.org. 2021. Available from: https://arxiv.org/abs/2108.12409

17) Su J, Lu Y, Pan S, Murtadha A, Wen B, Liu Y. RoFormer: Enhanced Transformer with Rotary Position Embedding. arXiv.org. 2021. Available from: https://arxiv.org/abs/2104.09864

18) Choromanski K, Likhosherstov V, Dohan D, Song X, Gane A, Sarlos T, et al. Rethinking Attention with Performers. arXiv.org. 2020. Available from: https://arxiv.org/abs/2009.14794

19) Liu Q, Kusner MJ, Blunsom P. arXiv.org. 2020. A Survey on Contextual Embeddings. Available from: https://arxiv.org/abs/2003.07278

20) Ethayarajh K. arXiv.org. 2019. How Contextual are Contextualized Word Representations? Comparing the Geometry of BERT, ELMo, and GPT-2 Embeddings. Available from: https://arxiv.org/abs/1909.00512v1

21) Farley P. Microsoft Learn. Multimodal embeddings concepts - Image Analysis 4.0 - Azure AI services. [cited 2025 Feb 11]. Available from: https://learn.microsoft.com/en-us/azure/ai-services/computer-vision/concept-image-retrieval

22) Hugging Face. What is Visual Question Answering? [Internet]. [cited 2025 Feb 11]. Available from: https://huggingface.co/tasks/visual-question-answering

23) OpenAI [Internet]. CLIP: Connecting text and images. [cited 2025 Feb 11]. Available from: https://openai.com/index/clip/

24) Girdhar R, El-Nouby A, Liu Z, Singh M, Alwala KV, Joulin A, et al. ImageBind: One Embedding Space To Bind Them All. arXiv.org. 2023. Available from: https://arxiv.org/abs/2305.05665

25) ImageBind by Meta AI [Internet]. [cited 2025 Feb 11]. Available from: https://imagebind.metademolab.com/demo

26) Wolleb B, Silvestri R, Vernikos G, Dolamic L, Popescu-Belis A. Assessing the Importance of Frequency versus Compositionality for Subword-based Tokenization in NMT. arXiv.org. 2023. Available from: https://arxiv.org/abs/2306.01393

27) Hugging Face Transformers [Internet]. Summary of the tokenizers. [cited 2025 Feb 11]. Available from: https://huggingface.co/docs/transformers/en/tokenizer_summary

28) Gage P. A new algorithm for data compression. Semanticscholar.org. 1994. Available from: https://www.semanticscholar.org/paper/A-new-algorithm-for-data-compression-Gage/1aa9c0045f1fe8c79cce03c7c14ef4b4643a21f8

29) Hugging Face NLP Course [Internet]. Normalization and pre-tokenization. [cited 2025 Feb 11]. Available from: https://huggingface.co/learn/nlp-course/chapter6/4?fw=pt

30) Bostrom K, Durrett G. Byte Pair Encoding is Suboptimal for Language Model Pretraining. arXiv.org. 2020. Available from: https://arxiv.org/abs/2004.03720

31) Schuster M, Nakajima K. Japanese and Korean voice search. In: 2012 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP) [Internet]. IEEE; 2012 [cited 2025 Feb 11]. Available from: https://doi.org/10.1109/icassp.2012.6289079

32) Hugging Face NLP Course. WordPiece tokenization [Internet]. [cited 2025 Feb 11]. Available from: https://huggingface.co/learn/nlp-course/en/chapter6/6

33) Qarah F, Alsanoosy T. A Comprehensive Analysis of Various Tokenizers for Arabic Large Language Models. Applied Sciences. 2024 Jun 29;14(13):5696. Available from: https://www.mdpi.com/2076-3417/14/13/5696

34) Devlin J, Chang MW, Lee K, Toutanova K. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. arXiv.org. 2018. Available from: https://arxiv.org/abs/1810.04805

35) Kudo T. Subword Regularization: Improving Neural Network Translation Models with Multiple Subword Candidates. arXiv.org. 2018. Available from: https://arxiv.org/abs/1804.10959

36) Hugging Face NLP Course [Internet]. Unigram tokenization. [cited 2025 Feb 11]. Available from: https://huggingface.co/learn/nlp-course/en/chapter6/7?fw=pt

37) Kudo T, Richardson J. SentencePiece: A simple and language independent subword tokenizer and detokenizer for Neural Text Processing. arXiv.org. 2018. Available from: https://arxiv.org/abs/1808.06226

38) Buchholz K. The Extreme Cost Of Training AI Models. Forbes [Internet]. 2024 Aug 23 [cited 2025 Feb 11]; Available from: https://www.forbes.com/sites/katharinabuchholz/2024/08/23/the-extreme-cost-of-training-ai-models/

39) The Ultimate Guide to Fine-Tuning LLMs from Basics to Breakthroughs: An Exhaustive Review of Technologies, Research, Best Practices, Applied Research Challenges and Opportunities (Version 1.0) [Internet]. [cited 2025 Feb 11]. Available from: https://arxiv.org/html/2408.13296v1#Ch1.S5

40) Raschka S. Finetuning LLMs Efficiently with Adapters. Ahead of AI [Internet]. 2023 May 20 [cited 2025 Feb 11]; Available from: https://magazine.sebastianraschka.com/p/finetuning-llms-with-adapters

41) LLMs: Fine-tuning, distillation, and prompt engineering. Google for Developers [Internet]. [cited 2025 Feb 11]; Available from: https://developers.google.com/machine-learning/crash-course/llm/tuning

42) Devlin J, Chang MW, Lee K, Toutanova K. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. arXiv.org. 2018. Available from: https://arxiv.org/abs/1810.04805

43) Hu EJ, Shen Y, Wallis P, Allen-Zhu Z, Li Y, Wang S, et al. LoRA: Low-Rank Adaptation of Large Language Models. arXiv.org. 2021. Available from: https://arxiv.org/abs/2106.09685

44) Jeon H, Kim Y, Kim J joon. L4Q: Parameter Efficient Quantization-Aware Fine-Tuning on Large Language Models. arXiv.org. 2024. Available from: https://arxiv.org/abs/2402.04902

45) Xiao G, Lin J, Seznec M, Wu H, Demouth J, Han S. SmoothQuant: Accurate and Efficient Post-Training Quantization for Large Language Models. arXiv.org. 2022. Available from: https://arxiv.org/abs/2211.10438