# Bidirectional Encoder Representations from Transformers (BERT)

# Table of Contents

## Table of Figures

# Abbreviations

| | |
|---|---|
| BERT | Bidirectional Encoder Representations from Transformers |
| CUDA | Compute Unified Device Architecture |
| DL | Deep Learning |
| FFN | Feed-Forward Network |
| LSTM | Long Short Term Memory |
| MHA | Multi-Head Attention |
| MLM | Masked Language Model |
| MMHA | Masked Multi-Head Attention |
| NLI | Natural Language Inference |
| NLP | Natural Language Processing |
| NN | Neural Network |
| NSP | Next Sentence Prediction |
| PE | Positional Encoding |
| QA | Question Answering |
| RNN | Recurrent Neural Network |
| SQuAD | Stanford Question Answering Dataset |

## 1. Introduction

Bidirectional Encoder Representations from Transformers(BERT) is an open-sourced Natural Language Processing(NLP) technique developed by researchers at Google in 2018(Khalid,2020). Several reasons made BERT a breakthrough in the Deep Learning(DL) space, including better state-of-the-art performance and pre-trained on a bidirectional language representation model enabling fine-tuning of specific NLP tasks more efficiently than its predecessors(Devlin et al.,2019;Khalid,2020). BERT can be used as an embedding to extract features from data and for finetune models for NLP tasks such as text summarisation, question answering, language translation and sentimental analysis (Khalid,2020; ProjectPro,2023).

## 2. Relevance of BERT

DL-based NLP models require huge volumes of data on specific task training to perform well, however, there is no sufficient human-annotated data available for training these kinds of models(Khalid,2020). This is one of the major challenges faced in this field and this problem can be resolved using pre-training and fine-tuning models(ibid). Researchers have developed various pre-training techniques that can leverage unlabelled data on the internet for training general-purpose language models and then fine-tuning those models for specific tasks allowing them to perform well for those tasks without training them from scratch(ibid).

Furthermore, the standard language models before BERT were trained unidirectional, restricting them from understanding the context of the text from both directions which is crucial for fine-tuning based models for token-level tasks such as question answering (Devlin et al.,2019). BERT leverages a Masked Language Model(MLM) that enables it to understand the context from both directions(ibid). MLM masks the tokens randomly from the input text and attempts to predict the masked vocabulary id correctly by understanding the context of the text(ibid). This feature of BERT also makes it suitable for fine-tuning the models more efficiently than predecessor models(ibid).

## 3. BERT Architecture and Working

Having understood the relevance of BERT, this section covers the architecture and its working in more detail. BERT employs a multi-layered bidirectional encoder based

on the architecture of Transformers for processing text (Devlin et al.,2019; Ravichandran,2022). Therefore, it is good to have an understanding of Transformers before diving into BERT architecture.

## 3.1. Transformers

Transformers are widely used Neural Network(NN) architectures because they overcame the limitations of its predecessors including Recurrent Neural Networks(RNNs) and Long Short Term Memory(LSTM) (Giacaglia,2019). Fig.1 depicts the limitations of RNNs and LSTM architectures.

| Limitations | RNN and LSTM | Solution using Transformers |
|---|---|---|
| Sequential Transduction | RNNs and LSTM process text in sequence, token by token, restricting these models from exploiting the advantage of parallel processing. | Transformers can process text more efficiently by parallel computation. |
| Long-term Dependency | RNNs are not effective in storing long sequences of text which help to understand the context of the text to predict the next token. LSTM was introduced to fix this problem of RNN through selectively remembering information, however, it is not able to handle long-term dependencies when the sequence is very long. | Transformers can handle this problem effectively by using self-attention mechanisms which enable them to capture long-term dependencies from long sequences of text. |
| Understanding Context | RNN and LSTM process data from left to right in one direction. Therefore, they are not able to understand the context of the previous token before predicting the next token. | Transformers are trained bidirectional enabling them to understand the context of the tokens more efficiently and the self-attention mechanism helps them to find the keywords that need to be focused on for predicting the next token. |
| Positional Dependency | RNN and LSTM consider the position of the tokens. The result of these models changes significantly if the order or position of the tokens is changed. | Transformers leverage positional encoding of the tokens. Hence, it is very little affected by the rearrangement of token order. |

*Figure 1.Limitations of RNN and LSTM Architectures(Giacaglia,2019)*

It's good to look at the architecture to get clear understanding on underlying architecture of BERT and how transformers overcome the limitations of its predecessors. As illustrated in Fig.2, the architecture of transformers consists of two components: Encoder and Decoder(Jain,2022). The Encoder extracts features from the input sequence and Decoder uses the features to generate an output(KiKaBeN,2021). The transformer model comprises a stack of Encoders and Decoders as demonstrated in Fig.3. Each component is covered in the next sections.

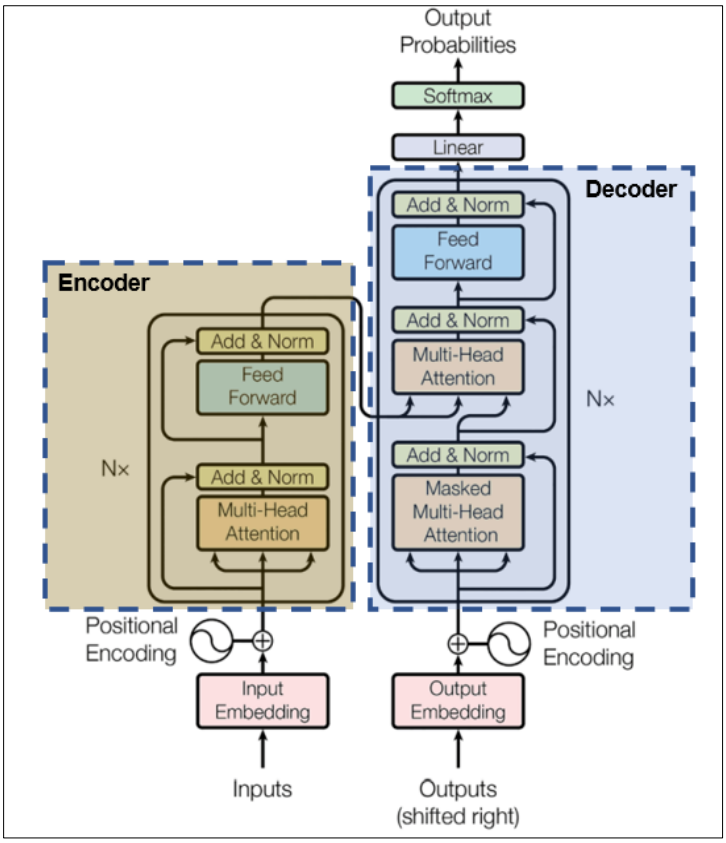



*Figure 2.High-level Architecture of Transformers(Vaswani et al.,2017)*

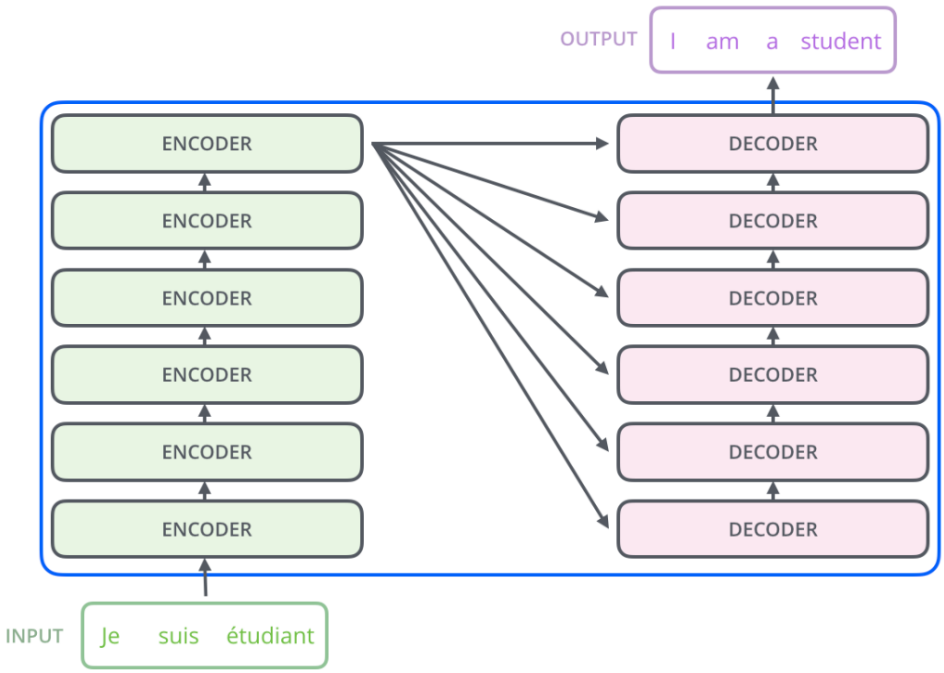



*Figure 3.Encoder-Decoder Architecture of Transformers(Alammar,2018b)*

### 3.1.1. Encoder

Transformer encoder comprises a stack of identical layers where each layer has two main sub-layers: Multi-Head Attention(MHA) and Feed-Forward Network(FFN) (Cristina,2023). The input embedding layer will pre-process the input sequence before it is processed for self-attention(Jain,2022).

#### 3.1.1.1. Input Embedding

The input embedding layer maps each word or token in the input text sequence to a learned vector representation through tokenisation and vectorisation(Jain,2022). NNs learn through numerical values, so each word is represented using a continuous value in the vector(ibid). This process is also known as Word Embedding(ibid). As illustrated in Fig.4, each word in the input sentence is converted into tokens during the tokenisation process and then each token is converted into a vector which is a numerical representation of the word using Word2Vec or GloVe encoding during the vectorisation process(ibid).

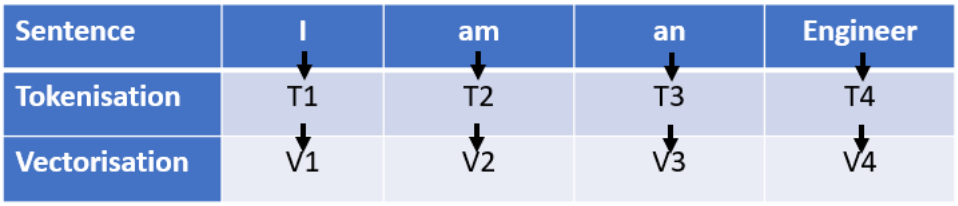



*Figure 4.Tokenisation and Vectorisation(Jain,2022)*

#### 3.1.1.2. Positional Encoding(PE)

The order and position of the words in a sentence are important in human language to interpret its meaning and context(Saeed,2023). The NLP using the RNN model leverages techniques to keep track of the order of words in the input sentence(ibid). However, the Transformers model considers each word as an independent token and adds PE to retain information about the position of the token within a text sequence(ibid).

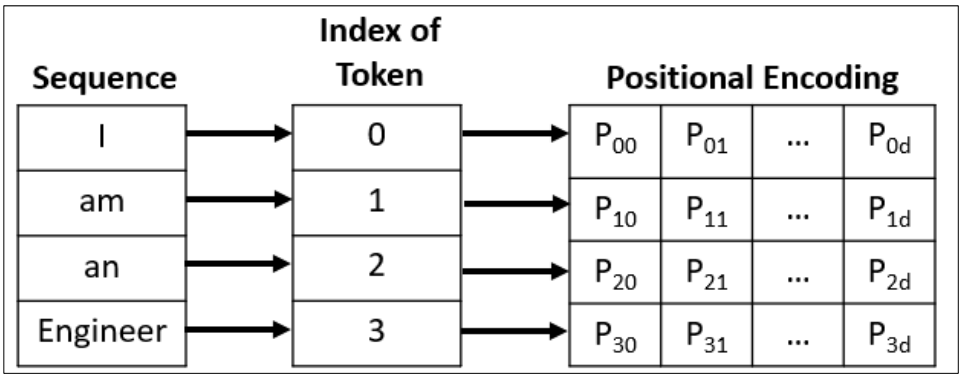



*Figure 5.Positional Encoding Matrix for 'I am an Engineer'(Saeed,2023)*

Instead of using numerical values such as index values, PE assigns a unique numerical representation that identifies the position of each token in a given sequence(Saeed,2023). One reason for not using index value is that index value can be a large number for long text sequences and normalising these indices between 1 and 0 might not provide expected results for variable length sequences(ibid). Hence, PE leverages a positional matrix in which each row is the summation of the encoded object with its positional information as illustrated in Fig.5(ibid). This transformation is achieved through sine and cosine functions as depicted in Fig.6 and an example is illustrated in Fig.7.

**Equations:**

$$P(k,2i) = \sin\left(\frac{k}{n^{2i/d}}\right)$$

$$P(k,2i+1) = \cos\left(\frac{k}{n^{2i/d}}\right)$$

| | |
|---|---|
| L | Length of the input sequence |
| k | Position of the token within the sequence, $0 \leq k \leq L/2$ |
| d | Dimension of the sequence |
| n | User-defined scalar value. It is assigned to 10000 by the authors of Transformers whitepaper. |
| i | Used to map column index, $0 \leq i \leq d/2$. The even index positions are mapped to the sine function whereas odd index positions are mapped to the cosine function. |

*Figure 6.Positional Encoding Equations(Jain,2022;Saeed,2023)*

| Sequence<br>*d=4*<br>*n=100* | Index of Token<br>*k* | Positional Encoding | | | |
|---|---|---|---|---|---|
| | | *i=0* | *i=0* | *i=1* | *i=1* |
| I | 0 | $P_{00}$<br>sin(0/100(0/4))<br>= 0 | $P_{01}$<br>cos(0/100(0/4))<br>= 1 | $P_{02}$<br>sin(0/100(2/4))<br>= 0 | $P_{03}$<br>cos(0/100(2/4))<br>= 1 |
| am | 1 | $P_{10}$<br>sin(1/100(0/4))<br>= 0.84 | $P_{11}$<br>cos(1/100(0/4))<br>= 0.54 | $P_{12}$<br>sin(1/100(2/4))<br>= 0.10 | $P_{13}$<br>cos(1/100(2/4))<br>= 1.0 |
| an | 2 | $P_{20}$<br>sin(2/100(0/4))<br>= 0.91 | $P_{21}$<br>cos(2/100(0/4))<br>= -0.42 | $P_{22}$<br>sin(2/100(2/4))<br>= 0.20 | $P_{23}$<br>cos(2/100(2/4))<br>= 0.98 |
| Engineer | 3 | $P_{30}$<br>sin(3/100(0/4))<br>= 0.14 | $P_{31}$<br>cos(3/100(0/4))<br>= -0.99 | $P_{32}$<br>sin(3/100(2/4))<br>= 0.30 | $P_{33}$<br>cos(3/100(2/4))<br>= 0.96 |

*Figure 7.Positional Encoding Matrix for the sequence 'I am an Engineer'(Jain,2022;Saeed,2023)*

The Input Embedding layer and PE will create a vector space where the words with similar semantics occur closer to each other as demonstrated in the Fig.8(Jain,2022). This vector space helps the model to easily learn to pay attention to the words represented in it(ibid).

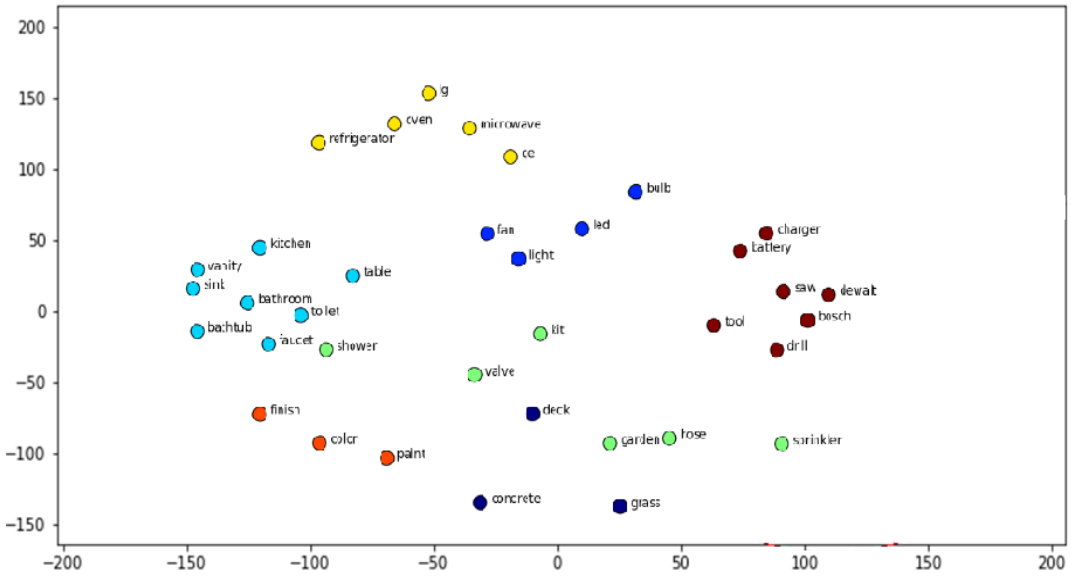



*Figure 8.Word Embedding Space(Padmanabhan,2022)*

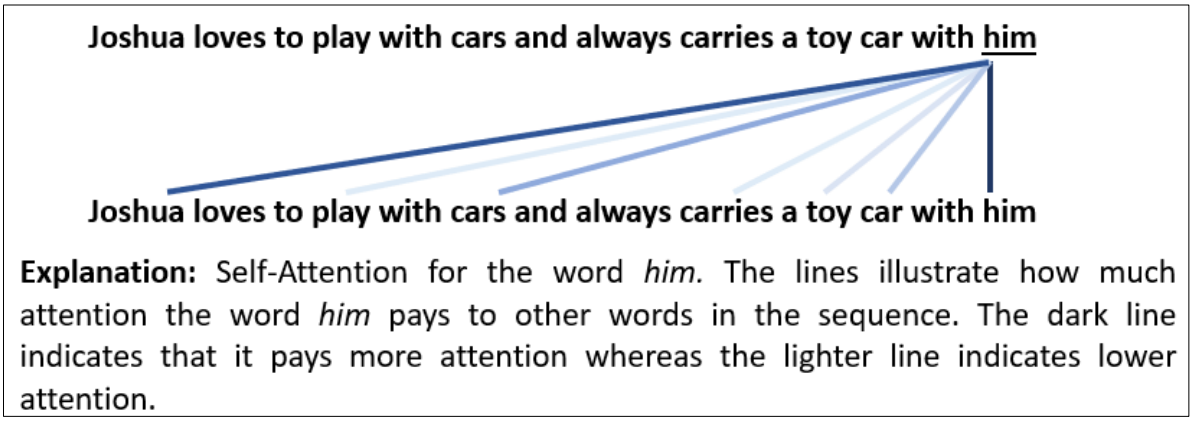
### 3.1.1.3. Multi-Head Attention(MHA)

MHA layer capture the context of the input sequence using self-attention mechanism which assist model to pay attention to different parts of the input sequence in parallel(Jain,2022;Karim,2023). An example of self-attention and its working are illustrated in Fig.9 and Fig.10.

**Input Sequence:** "*Joshua loves to play with cars and always carries a toy car with him*"

**Working:** Self-Attention helps to pay attention to the word *Joshua* whenever it encounters the word *him* in the input sequence. If the model doesn't pay attention to the word *Joshua* from the beginning of the sequence, it might not be able to understand the meaning of the pronoun.

*Figure 9.Example of Self-Attention Mechanism-Long Dependencies in a Sequence(Kortschak,2022)*




**Explanation:** Self-Attention for the word *him.* The lines illustrate how much attention the word *him* pays to other words in the sequence. The dark line indicates that it pays more attention whereas the lighter line indicates lower attention.

*Figure 10.Example of Self-Attention Mechanism for the word 'him'(Kortschak,2022)*

MHA comprises multiple self-attention modules focusing on various attention types(Jain,2022).Self-attention receives an input vector and converts it into three representational input vectors – Query, Key and Value, explained in Fig.11(Kortschak,2022). The three vectors help in calculating scores for determining how much attention should be given to other words in the sequence as shown in Fig.10(ibid).

| Vector | Description |
|---|---|
| Query | Representation of the word under consideration for which we need to calculate self-attention. |
| Key | Representation of each word in the sequence which is used against query to calculate self-attention. |
| Value | Actual representation of each word in the sequence. |

*Figure 11.Self-Attention Representational Vectors(Kortschak,2022)*

| Step Count | Description | Input | Output |
|---|---|---|---|
| 1 | Prepare Inputs for calculating self-attention. | Input Sequence | Input#1<br>Input#2<br>...<br>Input#n |
| 2 | Initialise the set of weights to calculate Query, Key and Value vectors. | Input Dimension | Wk, Wv, Wq |
| 3 | Multiply Input values with the respective weights to obtain the values for Query, Key and Value vectors. | Input#1 * Wk, Input#1 * Wv, Input#1 * Wq<br>Input#2 * Wk, Input#2 * Wv, Input#2 * Wq<br>...<br>Input#n * Wk, Input#n * Wv, Input#n * Wq | K1, V1, Q1<br>K2, V2, Q2<br>...<br>Kn, Vn, Qn |
| 4 | Multiply the Key and Query vectors to get the Score value for Input1. | K1*Q1 | S1 |
| 5 | Divide Score value by the square root of the dimension of the Key vector, dk to get scaled dot product, SDP for Input1. | S1/Vdk1 | SDP1 |
| 6 | Apply Softmax function to get scaled scores, SS for Input1. This function ensures that SDP lies between 0 and 1. It set higher values for relevant tokens and set lower values for less relevant tokens. | Softmax(SDP1) | SS1 |
| 7 | Multiply scaled Scores, SS with the Value vector to get weighted values, WV for Input1. | SS1*V1 | WV1 |
| 8 | Sum weighted values, WV to get the output for Input1. | ∑WV1 | Output#1 |
| 10 | Repeat Steps 4 to 8 for all the inputs i.e. Input2 to Inputn to obtain respective outputs. | | |

*Figure 12.Steps for Self-Attention(Jain,2022;Karim,2023; Saeed,2022)*

A set of steps are followed during the working of self-attention as illustrated in Fig.12 and all mathematical calculations followed in this process are vectorised(Karim,2023). Fig.13 depicts an example for the self-attention mechanism. The mechanism is explained with the help of the example demonstrated in Fig.13.

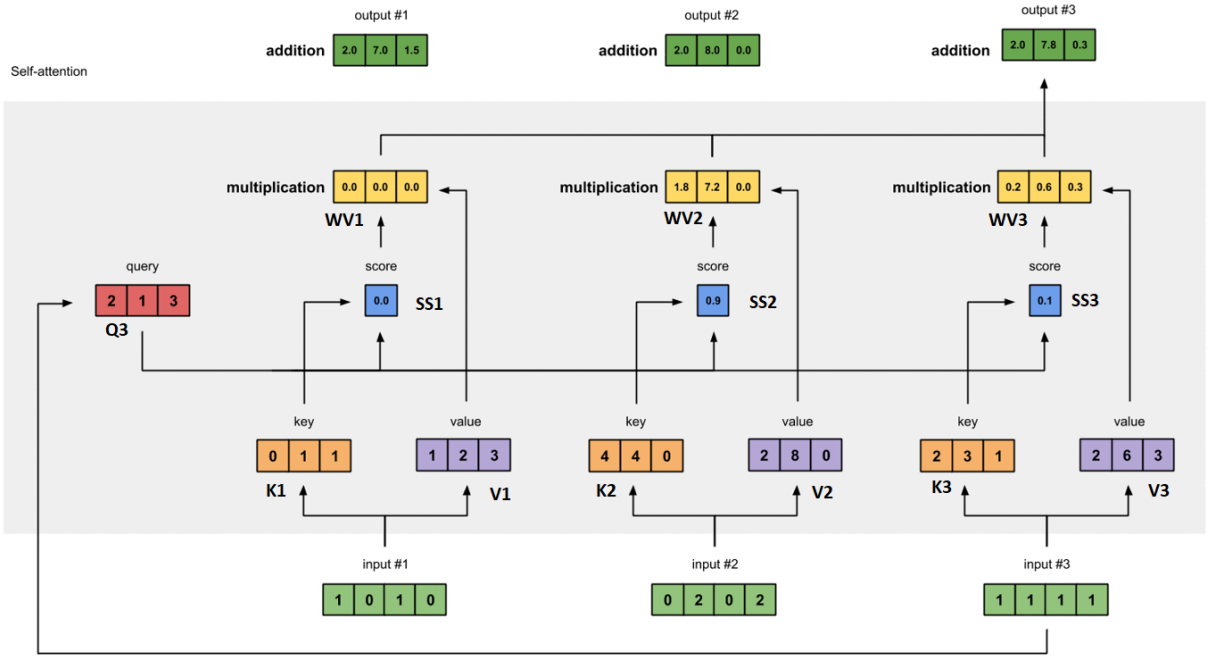



*Figure 13.Example of Multi-Head Attention Mechanism(Karim,2023)*

Firstly, the inputs for calculating self-attention are prepared(Karim,2023). In the example shown in Fig.13, inputs are highlighted as green boxes and the dimension of each input is 4(ibid). Secondly, the weights for calculating Query, Key and Value vectors are initialised(ibid). Each set of weights is set to 4x3 since every input dimension is 4 and it is illustrated in Fig.14(ibid). Thirdly, the three representational vectors for Query, Key and Value are obtained by multiplying it with each input and the results are highlighted in Fig.13 using yellow, purple and red boxes(ibid). The dimensions of the Query and Key vectors must be the same for performing multiplication operation in the upcoming step whereas the dimension of the Value vector can be different from the other two vectors(ibid).

| Weight for Key (Wk) | Weight for Value (Wv) | Weight for Query (Wq) |
|---|---|---|
| [[0, 0, 1],<br>[1, 1, 0],<br>[0, 1, 0],<br>[1, 1, 0]] | [[0, 2, 0],<br>[0, 3, 0],<br>[1, 0, 3],<br>[1, 1, 0]] | [[1, 0, 1],<br>[1, 0, 0],<br>[0, 0, 1],<br>[0, 1, 1]] |

*Figure 14.Weights for Key,Value and Query vectors(Karim,2023)*

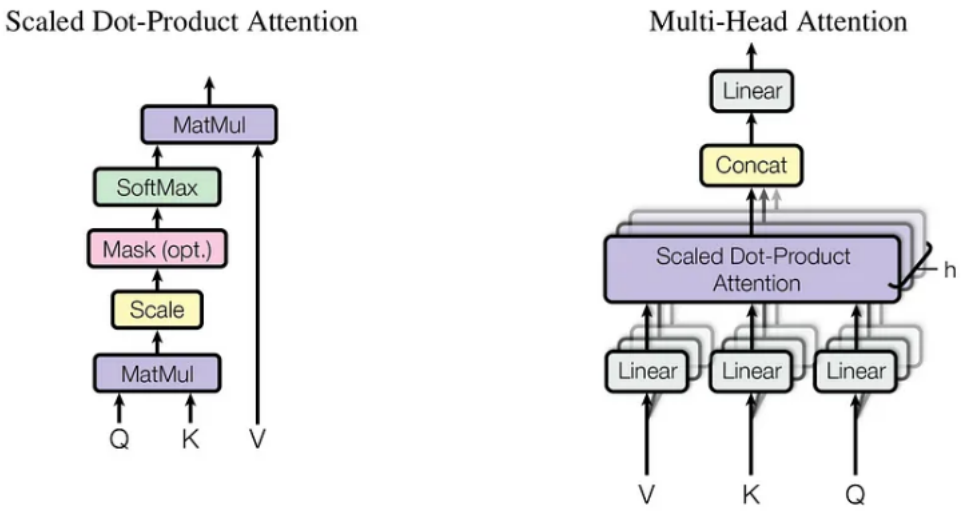



*Figure 15.(a)Scaled Dot-Product Attention(left) (b)Multi-Head Attention with Multiple Attention Layers(right)(Maxime,2019)*

Fourthly, the attention score is calculated by applying the score function on the Query and Key vectors and the dot product function is used to obtain score values in the above example(Jain,2022;Karim,2023). Fifthly, the attention score value is

divided by the square root of the dimension of the Key vector, dk to get scaled dot product as illustrated in Fig.15(a)(Jain,2022). Sixthly, the softmax function is applied to the updated scores to normalise the value and it lies between 0 and 1(Saeed,2022). In the example, the scores were [2,4,4] and after applying softmax function, it became [0,0.5,0.5](Karim,2023). It exaggerates relevant value and lowers the non-relevant value(Jain,2022;Saeed,2022).

Seventhly, the normalised attention scores are multiplied with the Value vector to get the weight values(Karim,2023). Eighthly, calculate the sum of all the weighted values in the yellow boxes to get the output of the first input. Similarly, steps 4 to 8 are repeated for all other inputs(ibid). Thus MHA performs multiple self-attentions in parallel as illustrated in Fig.15(b), each focusing on different parts of the input sequence that helps the model to capture the context of the input(Jain,2022). As discussed earlier, transformers overcome the limitation of its predecessors in handling long-term dependencies and capturing context of input using MHA.

#### 3.1.1.4. Feed-Forward Network(FFN)

The output from the self-attention layer is forwarded to a fully connected neural network called the FFN layer which is applied independently to each position of the sequence(Alammar,2018b). It comprises two linear layers-GeLu and Dropout layers and a non-linear function, ReLu in between (Kortschak,2022;Kosar,2021;Suresh,2022). The goal of this layer is to transform the output vectors from the self-attention layer to an acceptable input form for the next encoder or decoder layer(Ankit,2020). Since each self-attention layer can work independently in parallel to generate output, FFN is the most scaled-up component because it can take only one input at a time(Ankit,2020;Suresh,2022). As we discussed in the previous section, this is how transformers overcome the limitation of its predecessors which follows sequential processing(Ankit,2020).

### 3.1.2. Decoder

The decoder has a similar architecture of encoder. However, unlike encoder, the decoder leverages Masked MHA(MMHA)(Kortschak,2022). Since BERT leverages only the encoder of transformer for its implementation, the decoder section is not covered in detail. However, the next section will give a brief overview on MMHA.

#### 3.1.2.1. Masked Multi-Head Attention(MMHA)

In MMHA, some parts of the decoder input is masked before it is forwarded to self-attention layer which helps to train the model for better predictions(Cristina,2022; Kortschak,2022). The MMHA is designed to predict the word at a given position based on the known outputs of the words that are at preceding positions in the input sequence without knowing the words at succeeding positions as shown in Fig.16(ibid).

**Input Sequence 1:** *"Joshua loves to play with cars and always carries a toy____with him"*
**Input Sequence 2:** *"Joshua loves to play with cars and always carries a toy____*[MASKED] [MASKED]"

**Explanation:** It is more difficult to make predictions for input sequence 2 than the input sequence 1 because it doesn't know the words that come after it. The word *car* could be easily predicted if the model knows the next two words *with him* in the sequence. This will help the decoder to make better predictions on the unseen data through this training.

*Figure 16.Example of Masked Multi-Head Attention(Kortschak,2022)*

### 3.1.3. Linear and Softmax Layer

To obtain the output predictions, the last decoder output is transformed into words using a linear function layer, generating a logit vector of vocabulary size(Kortschak,2022). Then softmax function is applied to obtain the probability scores of each word as illustrated in Fig.17, and the word with the highest probability is chosen as prediction as it will be the word closer in the word embedding space(ibid).

**Input Sequence:** *Joshua loves to ____* *carries 0.16*
*play 0.37*
*car 0.02*

**Explanation:** The next word in the sequence will be *play* as it has the highest probability score.

*Figure 17.Example of probability scores for the next word prediction(Kortschak,2022)*

Having understood the transformer architecture in detail, the next section will cover the difference between BERT and Transformers.

## 3.2. BERT Architecture

BERT is a pretrained model built on transformer encoder stack(Kana,2019). While original transformer model used a 6 encoders stack, BERT's base version is 12 encoders stack whereas its large version is 24 encoders stack as illustrated in Fig.18(ibid).

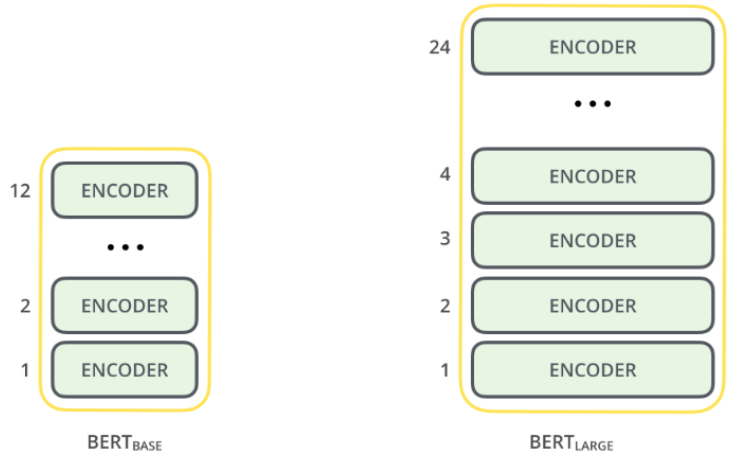



*Figure 18.BERT- Base and Large Versions(Alammar,2018a;Kana,2019)*

### 3.2.1. Differences between BERT and Transformers

The key differences and component size differences of BERT and Transformer are depicted in Fig.19 and Fig.20 respectively.

| Category | Transformer | BERT |
|---|---|---|
| Architecture | Encoder-Decoder | Encoder only |
| Design Objective | Generate output sequences from input sequences | Generate high-quality representations of text that can be used for different NLP tasks. |
| Training Process | Trained on Supervised Learning task | Trained on Unsupervised Learning task |
| Encodings | Word Embeddings and Positional Encodings | Segment Embeddings, Word Embeddings and Positional Encodings |
| Pre-training Approach | Pretrained on various tasks including language modelling and | Pretrained specifically on Masked Language Modeling (MLM) and Next Sentence Prediction (NSP) |
| Bidirectionality | Unidirectional model which processes sequence in left-to-right or right-to-left fashion. | Bidirectional model which processes sequence left and right to capture more deep contextual information. |
| Fine-tuning | Can be fine-tuned for different tasks by appending task specific layer on the pre-trained model. | Can be fine-tuned for specific tasks such as text classification, question answering etc. |

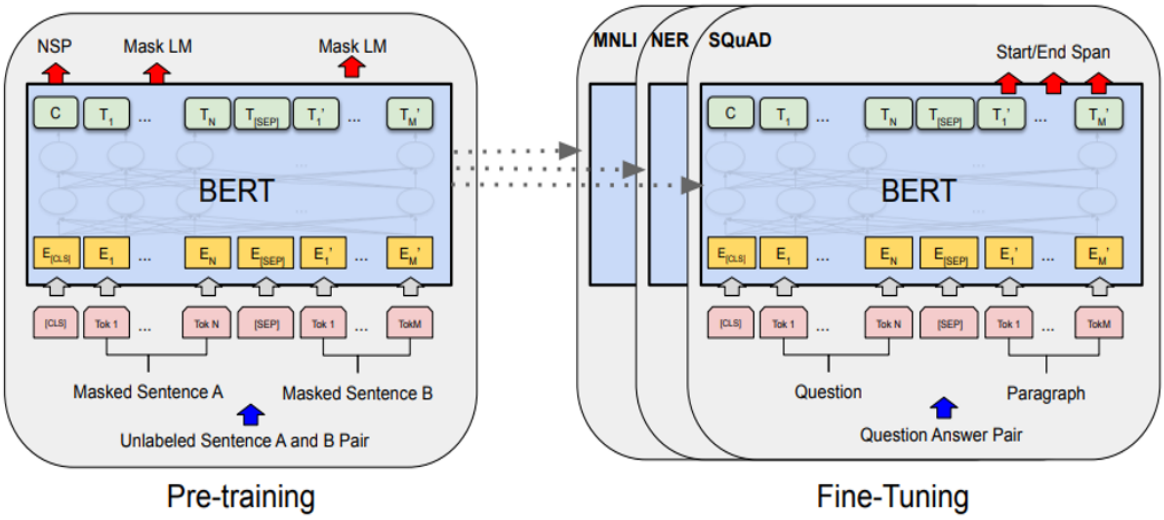
*Figure 19.Key Differences between BERT and Transformers(Alammar,2018a ;DeLucia,2023;Hui,2019;StackExchange,2020)*

| Component | Transformer | BERT Base Version | BERT Large Version |
|---|---|---|---|
| Encoder Stack Size | 6 | 12 | 24 |
| FFN (Hidden Units) | 512 | 768 | 1024 |
| Attention Heads | 8 | 12 | 16 |

*Figure 20.Component Size Differences of Transformers and BERT(Alammar,2018a)*

### 3.2.2. BERT Framework Steps

BERT undergoes through two phases-Pretraining and Fine-tuning(Devlin *et al.*,2019) as depicted in Fig.21.




**Note:** BERT framework has two phases-pretraining and fine-tuning. Both phases follow the same architecture except for the output layer. Also, the same set of parameters is used to initialise the model for different NLP tasks. Every input is prefixed with a special symbol [CLS] to indicate the starting and every input is suffixed with a separator token [SEP] to indicate the end of the sentence.

*Figure 21.BERT Framework Phases(Devlin et al.,2019)*

#### 3.2.2.1. BERT Pretraining

BERT is the first pretrained model on unlabelled data and two NLP tasks are performed during this phase-MLM and NSP which will be covered in the upcoming sections(Hui,2019). Before diving into sub-tasks, it's good to understand the inputs and expected outputs of the pretraining.

##### 3.2.2.1.1. Inputs and Outputs

BERT requires one or two text sequences as input and as mentioned in Fig.21 input is prefixed with [CLS] and suffixed with [SEP](Hui,2019). The model will generate N output tokens for N input tokens including appended tokens as shown in Fig.22(ibid).

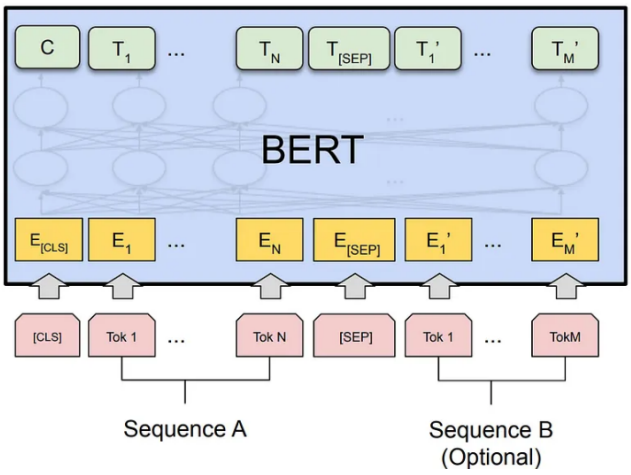
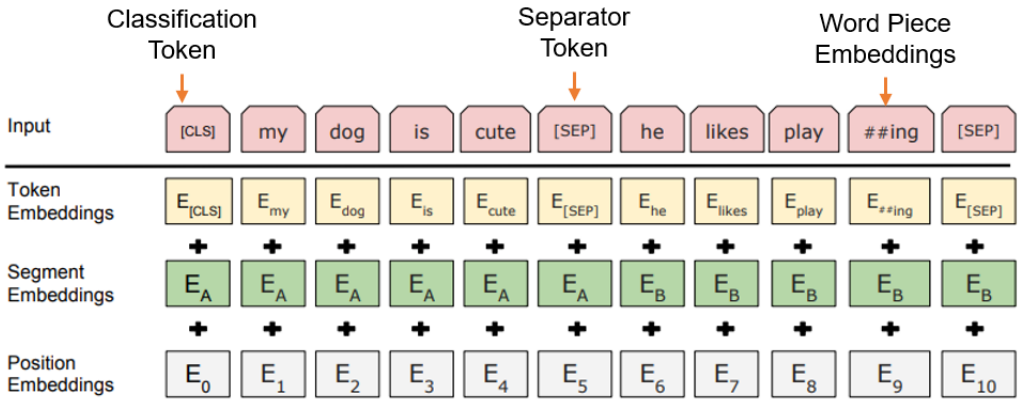



*Figure 22.Inputs and Outputs of BERT(Hui,2019)*

The appended input token [CLS] will generate an output token C which is used only for classification problems and ignored for non-classification problems(Hui,2019). Fig.23 illustrates the input representations used in BERT.




**Note:** BERT input embeddings comprise Word Piece Embeddings, Segment Embeddings and Positional Embeddings.

**Word Piece Embeddings** – BERT leverages Word Piece Embeddings with a 30,000 token vocabulary and denotes the split word with ##. This helps the model to reduce the vocabulary size. For example, word *playing* is split into *play* and *##ing*.

**Token Embeddings** - Every input is prefixed with a special symbol [CLS] to indicate the start and every input is suffixed with a separator token [SEP] to indicate the end of the sentence.

**Positional Embeddings** – It leverages learned Positional Embedding and can support sequences with a maximum length of 512 tokens. It indicates the position of a token in a sentence.

**Segment Embeddings**- A sequence may contain the sentence pairs and [SEP] is used to identify the end of the sentence. An embedding A is added to indicate the first sentence and another embedding B is added to indicate the second sentence in the pair. Only A embedding is used if the input is a single sentence. This will help the model to identify to which sentence the token belongs.

*Figure 23.BERT Input Representation(Devlin et al.,2019;Hui,2019;Khalid,2020)*

### 3.2.2.1.2. Masked Language Modelling(MLM)

In order to train deep bidirectional representation, MLM randomly masks 15% of words in the input sequence by replacing words with [MASK] token and then predict those masked words based on the contextual information obtained from unmasked words in the sequence as illustrated in Fig.24(Devlin *et al.*,2019;Khalid,2020). The drawback of this approach is that the [MASK] token will not be present in fine-tuning phase, resulting in a mismatch with pre-training phase and it can be overcome using different techniques listed in Fig.25.

**Input Sequence:** *"Joshua loves to play with toy cars."*
**After Masking:** *"Joshua loves to* [MASK] *with toy cars."*

**Explanation:** Unidirectional model predicts the word by looking at the words in the preceding locations. While MLM leverages bidirectional processing so it predicts the masked word by looking at the unmasked words on both sides, left and right which will provide more context of the masked word.

*Figure 24.Example of BERT Bidirectional Processing(Khalid,2020)*

| No. | Mitigation Steps |
|---|---|
| 1 | The chosen token at the $i^{th}$ position will be replaced with [MASK] token for 80% of the time. |
| 2 | The chosen token at the $i^{th}$ position will be replaced with a random token for 10% of the time. |
| 3 | The chosen token at the $i^{th}$ position will remain unchanged for 10% of the time. |

*Figure 25.Mitigation Steps for MLM Mask Limitation(Devlin et al.,2019)*

BERT uses a cross entropy loss function which considers only the prediction of masked tokens and ignores the unmasked token predictions(Devlin *et al.*,2019;Khalid,2020).

### 3.2.2.1.3. Next Sentence Prediction(NSP)

The model needs to understand the relationship between sentences in an input in order to perform different NLP tasks such as Question Answering(QA), Natural Language Inference(NLI) etc.(Devlin *et al.*,2019). NSP helps the model to understand the relationship between two sentences in an input sequence and also helps it to predict next sentence(Khalid,2020). The model is trained on inputs with sentence pairs and learns to predict next sentence in the original input(ibid). In this

training, the output token C is considered for classification as mentioned in the previous section(Hui,2019). Fig.26. shows how training samples are selected for NSP and Fig.27 illustrates an example of it.

| No. | NSP Training Examples |
|---|---|
| 1 | When selecting training examples as sentence pairs, 50% of the time B is the original sentence which is followed by A and labelled as *'IsNext'.* |
| 2 | When selecting training examples as sentence pairs, 50% of the time it is a random sentence from the corpus and labelled as *'NotNext'.* |

*Figure 26.NSP Training Examples(Devlin et al.,2019)*

**Sentence A** = The man went to the store.
**Sentence B** = He bought a gallon of milk.
**Label** = IsNextSentence

**Sentence A** = The man went to the store.
**Sentence B** = Penguins are flightless.
**Label** = NotNextSentence

*Figure 27.NSP Training Example(Hui,2019)*

The combination of MLM and NSP approaches during pretraining helps the model to minimise the loss function(Khalid,2020).

### 3.2.2.2. BERT Fine-tuning

Once the model completes the pretraining, it can be finetuned for specific NLP tasks as illustrated in Fig.28(Khalid,2020). In this phase, the model parameters are finetuned end-to-end by providing task related data and corresponding labels(Hui,2019). Fine-tuning phase is relatively cheaper than pre-training phase(Devlin *et al.*,2019).

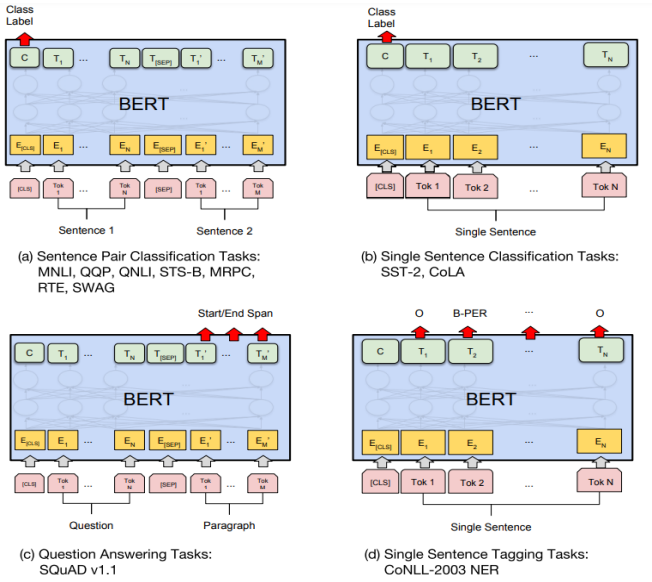



*Figure 28.Fine-tuning of BERT for different NLP tasks(Devlin et al.,2019)*

#### 3.2.2.2.1. BERT Fine-tuning for Question Answering Task

BERT can be finetuned different downstream tasks as illustrated in Fig.28 including sequence-level tasks such as sentence pair classification and single sentence classification, and token-level tasks such as question answering and single sentence tagging tasks(Devlin *et al.*,2019). In this section, BERT finetuning is explained using question answering task.

**Context:** *"Narendra Modi is the Prime Minister of India and was born in Gujarat state."*
**Question:** *"Who is Narendra Modi?"*
**Expected Answer:** *"Prime Minister of India"*

*Figure 29.Example for Question Answering(Hui,2019)*

During finetuning, only two vectors are introduced which is Start(S) vector and End(E) vector(Devlin *et al.*,2019;Hui,2019). $T_i$ is the output token at $i^{th}$ position of the sequence(ibid). The objective is to find whether the token at $i^{th}$ position is the start of answer, for example above, it should determine the position of the word *'Prime'* which is the first word of the answer(ibid). For achieving this goal, the model performs a dot product of S and $T_i$ and the output value is softmaxed over all the tokens in the paragraph. Fig.30-Equation-1 illustrates the mathematical equation for this.

**Equation 1:**

$$P_i = \frac{e^{S \cdot T_i}}{\sum_j e^{S \cdot T_j}} \qquad S \in \mathbb{R}^H \qquad E \in \mathbb{R}^H$$

**Equation 2:**

$$P_i = \frac{e^{E \cdot T_i}}{\sum_j e^{E \cdot T_j}} \qquad S \in \mathbb{R}^H \qquad E \in \mathbb{R}^H$$

*Figure 30.Equations for finding start and end of answer in Question Answering task(Devlin et al.,2019;Hui,2019)*

Similarly, to find the end of the answer, it performs the same equation with End vector as depicted in Fig.30-Equation-2. Then score of each candidate span (phrase or word) is calculated by the equation in Fig.31(Devlin *et al.*,2019). The candidate span with highest score is considered as the answer to the question(ibid). In the above example, the candidate span with highest score will be *"Prime Minister of India".*

$$S \cdot T_i + E \cdot T_j \quad \text{where } j \geq i$$

*i indicates the start position of the candidate span*
*j indicates the end position of the candidate span*

*Figure 31.Candidate Span Score Calculation Equation(Devlin et al.,2019)*

The Stanford Question Answering Dataset(SQuAD) is a collection of crowd-sourced question and answer pairs dataset used for finetuning BERT model for the question answering task(Devlin *et al.*,2019).

#### 3.2.2.2.2. Recommended Hyper Parameter Values

Although most model parameters are same as that of pretraining, there are some exceptions such as batch size, learning rate, epoch count and dropout probability(Devlin *et al.*,2019). The authors of BERT recommend a set of hyperparameter values that works well across all tasks based on their experiments are listed in Fig.32.

| Hyper Parameter | Optimal Value Range (Recommended by the authors of BERT) |
|---|---|
| Batch Size | 16, 32 |
| Learning Rate (Adam) | 5e-5, 3e-5,2e-5 |
| Epoch Count | 2,3,4 |
| Dropout Probability | 0.1 |

*Figure 32.Recommended Optimal Hyper Parameter Values(Devlin et al.,2019)*

## 4. BERT Real-World Applications

As discussed in the previous sections, BERT has several applications and a few are listed in Fig.33.

| Applications of BERT | |
|---|---|
| Text Summarisation | BERT provides a common framework for extractive and abstractive summarisation.<br>The extractive summarisation technique identifies the important sentences in the input and generates a summary using those sentences. The neural encoder transforms the input into sentence representations and the classifier chooses the important sentences from it.<br>Unlike extractive summarisation, the abstractive summarisation technique generates summaries by rephrasing the words in the input or using new words. |
| Question Answering | As discussed in the fine-tuning section, BERT can be used for question answering and chatbots. |
| Google Search | BERT can identify the intent of the search and return relevant search results for users by analysing multiple tokens and sentences with maximum attention in parallel. |
| Sentiment Analysis | BERTweet is the first pre-trained language model which is used for sentiment analysis of Tweets on Twitter. It can identify whether a tweet is positive or negative by calculating POS and NEG scores respectively. |
| Language Translation | BERT is pre-trained in multiple languages and can be used for language translations. |
| Named-Entity Recognition | BERT can be used for word classification based on different categories such as organisation, person, location, time etc. |
| Fact Checking | BERT can be used to classify whether the input is a fact or not. |
| Sentence Classification | BERT can be used to classify input sentences into different categories based on topics |

*Figure 33.Applications of BERT(Alammar,2018a;Chakraborty,2020;Jain,2022;Persson,2021)*

## 5. Benefits and Limitations of BERT

As we discussed in the previous sections, there are several benefits of using BERT as shown in Fig.34. Also, it has some drawbacks and a few are listed in Fig.35.

| Benefits of BERT | |
|---|---|
| Multi-lingual support | BERT is pretrained on more languages than other models |
| Easily Fine-tuned | BERT can be easily and quickly fine-tuned for specific NLP tasks |
| Accuracy | BERT is more accurate than other models since its being updated regularly |
| Bidirectional | BERT process data in both directions to capture deep contextual information |
| Long-term dependencies | BERT can capture long-term dependencies and relationships between words in the input sequence since it's built using Transformer Encoder architecture |
| Contextual Word Embeddings | BERT can perform well over other models by capturing deep contextual information using contextual word embeddings |
| State-of-the-art Performance | BERT has achieved several NLP performance benchmarks including GLUE(General Language Understanding Evaluation) over other models |

*Figure 34.Benefits of BERT(DeLucia,2023;Ravichandran,2022)*

| Limitations of BERT | |
|---|---|
| Expensive | BERT requires a huge amount of computational power during training and it is very expensive. |
| Time-consuming | Training of BERT is very time-consuming as it is computation intensive. |
| Large Model | The model is large due to the corpus and training structure. |
| Long Sequences | BERT can not handle long sequences due to memory constraints and long sequences may get truncated which will result in potential loss of contextual information. |
| Document-level tasks | BERT can handle only sentence-level or token-level tasks effectively. It is less effective for large text-level tasks. |
| Out Of Vocabulary(OOV) Words | Although BERT handles OOV words using word piece encoding, there are chances of encountering words in input sequence which are not in the vocabulary. In that case, BERT might not be able to produce the expected outcomes. |

*Figure 35.Limitations of BERT(Devlin et al.,2019; McCormick,2019;Ravichandran,2022)*

## 6. BERT Model for Spam Message Classification

This section will cover step by step tutorial to build a model using BERT for classifying the message whether it is spam or not.

### 6.1. Step 1: Import Libraries and Dataset

Firstly, to setup the environment, install required libraries and import as needed.

```
!pip install transformers
import numpy as np
import pandas as pd
import torch
import torch.nn as nn
from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report
import transformers
from transformers import AutoModel, BertTokenizerFast
```

```
from torch.utils.data import TensorDataset, DataLoader, RandomSampler, SequentialSampler
```

Once required libraries are imported, specify GPU as CUDA(Jain,2021). CUDA(Compute Unified Device Architecture) is developed by NVIDIA to provide a platform to accelerate the GPU capabilities(Turing,2022).

```
# specify GPU
device = torch.device("cuda")
```

To check the availability of the GPU, the following code is executed and if it is not found it will throw an error(Kana,2019).

```
# verify GPU availability
import tensorflow as tf

device_name = tf.test.gpu_device_name()
if device_name != '/device:GPU:0':
  raise SystemError('GPU device not found')
print('Found GPU at: {}'.format(device_name))
```

Then, read the data from the local file and load data from the dataset into a dataframe(Jain,2021). The dataset contains two columns label and text, where label 1 indicates spam and 0 indicates genuine(ibid)

```
df = pd.read_csv("/content/spamdata_v2.csv")
df.head()
```

| | label | text |
|---|---|---|
| **0** | 0 | Go until jurong point, crazy.. Available only ... |
| **1** | 0 | Ok lar... Joking wif u oni... |
| **2** | 1 | Free entry in 2 a wkly comp to win FA Cup fina... |
| **3** | 0 | U dun say so early hor... U c already then say... |
| **4** | 0 | Nah I don't think he goes to usf, he lives aro... |

Then, check the label class distribution as shown below and it will show the percentage of spam and genuine messages in the dataset(Jain,2021).

```
# check class distribution
df['label'].value_counts(normalize = True)
```

```
0    0.865937
1    0.134063
Name: label, dtype: float64
```

## 6.2. Step 2: Divide the data for testing and training

In this step, data is divided for testing and training purposes(Jain,2021).

```
# split train dataset into train, validation and test sets
train_text, temp_text, train_labels, temp_labels = train_test_split(df['text'], df['label'],
                                                                    random_state=2018,
                                                                    test_size=0.3,
                                                                    stratify=df['label'])


val_text, test_text, val_labels, test_labels = train_test_split(temp_text, temp_labels,
                                                                random_state=2018,
                                                                test_size=0.5,
                                                                stratify=temp_labels)
```

## 6.3. Step 3: Import BERT and Load Tokenizer

In this step, hugging face BERT model, uncased version is imported and load the BERT tokenizer for preparing input representations(Jain,2021).

```
# import BERT-base pretrained model
bert = AutoModel.from_pretrained('bert-base-uncased')

# Load the BERT tokenizer
tokenizer = BertTokenizerFast.from_pretrained('bert-base-uncased')
```

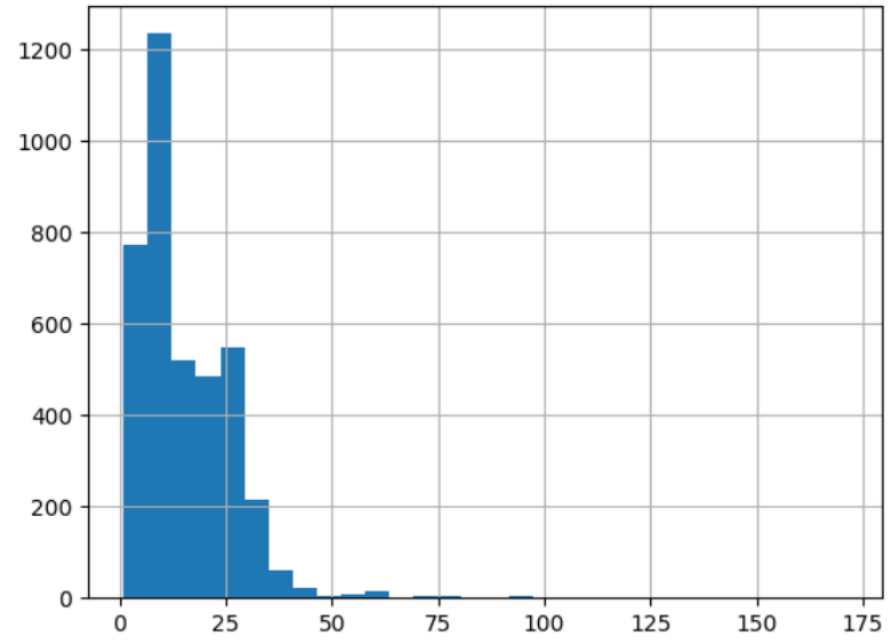
```
Downloading (…)lve/main/config.json: 100% 570/570 [00:00<00:00, 9.28kB/s]
Downloading pytorch_model.bin: 100% 440M/440M [00:05<00:00, 136MB/s]
Some weights of the model checkpoint at bert-base-uncased were not used when initializing BertModel: ['c
- This IS expected if you are initializing BertModel from the checkpoint of a model trained on another t
- This IS NOT expected if you are initializing BertModel from the checkpoint of a model that you expect
Downloading (…)okenizer_config.json: 100% 28.0/28.0 [00:00<00:00, 1.02kB/s]
Downloading (…)solve/main/vocab.txt: 100% 232k/232k [00:00<00:00, 1.65MB/s]
Downloading (…)/main/tokenizer.json: 100% 466k/466k [00:00<00:00, 7.50MB/s]
```

Check the length of messages in training data set(Jain,2021).

```
# get length of all the messages in the train set
seq_len = [len(i.split()) for i in train_text]

pd.Series(seq_len).hist(bins = 30)
```

```
<Axes: >
```




## 6.4. Step 4: Tokenize and Encode the Input Sequence

As the fourth step, tokenize the input for testing, training and validation datasets(Jain,2021)

```
# tokenize and encode sequences in the training set
tokens_train = tokenizer.batch_encode_plus(
    train_text.tolist(),
    max_length = 25,
    pad_to_max_length=True,
    truncation=True
)

# tokenize and encode sequences in the validation set
tokens_val = tokenizer.batch_encode_plus(
    val_text.tolist(),
    max_length = 25,
    pad_to_max_length=True,
    truncation=True
)

# tokenize and encode sequences in the test set
tokens_test = tokenizer.batch_encode_plus(
    test_text.tolist(),
    max_length = 25,
    pad_to_max_length=True,
    truncation=True
)
```

```
/usr/local/lib/python3.10/dist-packages/transformers/tokenization_utils_base.py:2364: FutureWar
  warnings.warn(
```

As we discussed in previous sections, BERT uses Word Piece tokenization and maximum sequence length of the sequence can be upto 512(Jain,2021). In this example, the max_length is set to 25 so that input token length can be upto 25. Also, the pad_to_max_length is set to true for padding the tokens that are less than the max_length. The results of the tokenization is stored into respective lists.

## 6.5. Step 5: Convert Lists to Tensors

The model expects the input datatype as torch tensors(Kana,2019). Therefore, all the data which is stored in the lists during the previous step is converted into tensors(Jain,2021).

```
## convert lists to tensors for training data
train_seq = torch.tensor(tokens_train['input_ids'])
train_mask = torch.tensor(tokens_train['attention_mask'])
train_y = torch.tensor(train_labels.tolist())

## convert lists to tensors for validation data
val_seq = torch.tensor(tokens_val['input_ids'])
val_mask = torch.tensor(tokens_val['attention_mask'])
val_y = torch.tensor(val_labels.tolist())

## convert lists to tensors for testing data
test_seq = torch.tensor(tokens_test['input_ids'])
test_mask = torch.tensor(tokens_test['attention_mask'])
test_y = torch.tensor(test_labels.tolist())
```

## 6.6. Step 6: Hyper Parameter Optimisation

In this step, optimal values are assigned to hyper parameters. As discussed in the previous section, recommended optimal values can be used. However, in this example, learning rate and epoch is selected differently.

```
batch_size =32
dropout_probability =0.1
adam_learning_rate= 1e-5
epochs = 10
```

Also, automated hyper parameter optimisation can be done using libraries such as GridSearchCV for better accuracy(Brownlee,2022).

## 6.7. Step 7: Initialise Data Loader

As seventh step, the data loaders are initialised for training and validation datasets(Jain,2021). This will create iterators required for fine-tuning the model using the data during BERT fine-tuning phase(Kana,2019).

```
# wrap tensors
train_data = TensorDataset(train_seq, train_mask, train_y)

# sampler for sampling the data during training
train_sampler = RandomSampler(train_data)

# dataLoader for train set
train_dataloader = DataLoader(train_data, sampler=train_sampler, batch_size=batch_size)

# wrap tensors
val_data = TensorDataset(val_seq, val_mask, val_y)

# sampler for sampling the data during training
val_sampler = SequentialSampler(val_data)

# dataLoader for validation set
val_dataloader = DataLoader(val_data, sampler = val_sampler, batch_size=batch_size)
```

## 6.8. Step 8: BERT Model for Pretraining

In this step, all the layers involved in the pretraining phase are defined including attention layers, FFN layer, linear and softmax layers. This code will freeze the layers of the model and the parameter values will not change for the frozen layers(Jain,2021).

```
# freeze the model layers so that the parameters won't change for those frozen layers
for param in bert.parameters():
    param.requires_grad = False
```

In this example, BERT_Arch class defined with two functions – init() and forward(). This function will initialise and set layers required for pretraining.

```
class BERT_Arch(nn.Module):

    def __init__(self, bert):
        super(BERT_Arch, self).__init__()

        self.bert = bert

        # dropout layer
        self.dropout = nn.Dropout(dropout_probability)

        # relu activation function
        self.relu =  nn.ReLU()

        # dense layer 1
        self.fc1 = nn.Linear(768,512)

        # dense layer 2 (Output layer)
        self.fc2 = nn.Linear(512,2)

        #softmax activation function
        self.softmax = nn.LogSoftmax(dim=1)
```

```
#define the forward pass
def forward(self, sent_id, mask):

    #pass the inputs to the model
    _, cls_hs = self.bert(sent_id, attention_mask=mask, return_dict=False)

    x = self.fc1(cls_hs)

    x = self.relu(x)

    x = self.dropout(x)

    # output layer
    x = self.fc2(x)

    # apply softmax activation
    x = self.softmax(x)

    return x
```

```
# pass the pre-trained BERT to our define architecture
model = BERT_Arch(bert)

# push the model to GPU
model = model.to(device)
```

## 6.9. Step 9: BERT Fine-tuning

Once pretraining is completed, the next step is fine-tuning the values(Jain,2021). Adam optimiser is used tune the learning rate to update the individual network weights(Brownlee,2021).

```
# optimizer from hugging face transformers
from transformers import AdamW

# define the optimizer
optimizer = AdamW(model.parameters(),lr = adam_learning_rate)
```

```
from sklearn.utils.class_weight import compute_class_weight

#compute the class weights
class_weights = compute_class_weight(
                                        class_weight = "balanced",
                                        classes = np.unique(train_labels),
                                        y = train_labels
                                    )

print("Class Weights:",class_weights)
```

```
Class Weights: [0.57743559 3.72848948]
```

It also define the cross entropy loss function with updated weights(Jain,2021).

```
# converting list of class weights to a tensor
weights= torch.tensor(class_weights,dtype=torch.float)

# push to GPU
weights = weights.to(device)

# define the loss function
cross_entropy  = nn.NLLLoss(weight=weights)
```

```
# function to train the model
def train():

    model.train()
    total_loss, total_accuracy = 0, 0

    # empty list to save model predictions
    total_preds=[]

    # iterate over batches
    for step,batch in enumerate(train_dataloader):

        # progress update after every 50 batches.
        if step % 50 == 0 and not step == 0:
            print('  Batch {:>5,}  of  {:>5,}.'.format(step, len(train_dataloader)))
```

```
        # push the batch to gpu
        batch = [r.to(device) for r in batch]

        sent_id, mask, labels = batch

        # clear previously calculated gradients
        model.zero_grad()

        # get model predictions for the current batch
        preds = model(sent_id, mask)

        # compute the loss between actual and predicted values
        loss = cross_entropy(preds, labels)

        # add on to the total loss
        total_loss = total_loss + loss.item()
```

```
    # backward pass to calculate the gradients
    loss.backward()

    # clip the the gradients to 1.0. It helps in preventing the exploding gradient problem
    torch.nn.utils.clip_grad_norm_(model.parameters(), 1.0)

    # update parameters
    optimizer.step()

    # model predictions are stored on GPU. So, push it to CPU
    preds=preds.detach().cpu().numpy()

# append the model predictions
total_preds.append(preds)

# compute the training loss of the epoch
avg_loss = total_loss / len(train_dataloader)

  # predictions are in the form of (no. of batches, size of batch, no. of classes).
  # reshape the predictions in form of (number of samples, no. of classes)
total_preds  = np.concatenate(total_preds, axis=0)

#returns the loss and predictions
return avg_loss, total_preds
```

The model is trained with training dataset and after each iteration(epoch), the model accuracy is evaluated(Jain,2021). The progress report is displayed after every 50 batches for monitoring the model(ibid).

```
# function for evaluating the model
def evaluate():

    print("\nEvaluating...")

    # deactivate dropout layers
    model.eval()

    total_loss, total_accuracy = 0, 0

    # empty list to save the model predictions
    total_preds = []

    # iterate over batches
    for step,batch in enumerate(val_dataloader):

        # Progress update every 50 batches.
        if step % 50 == 0 and not step == 0:

            # Calculate elapsed time in minutes.
            elapsed = format_time(time.time() - t0)

            # Report progress.
            print('  Batch {:>5,}  of  {:>5,}.'.format(step, len(val_dataloader)))

        # push the batch to gpu
        batch = [t.to(device) for t in batch]

        sent_id, mask, labels = batch
```

```
      # deactivate autograd
      with torch.no_grad():

          # model predictions
          preds = model(sent_id, mask)

          # compute the validation loss between actual and predicted values
          loss = cross_entropy(preds,labels)

          total_loss = total_loss + loss.item()

          preds = preds.detach().cpu().numpy()

          total_preds.append(preds)

  # compute the validation loss of the epoch
  avg_loss = total_loss / len(val_dataloader)

  # reshape the predictions in form of (number of samples, no. of classes)
  total_preds  = np.concatenate(total_preds, axis=0)

  return avg_loss, total_preds
```

Also, it calculates the training and validation loss after each iteration(Jain,2021).

```
# set initial loss to infinite
best_valid_loss = float('inf')

# empty lists to store training and validation loss of each epoch
train_losses=[]
valid_losses=[]

#for each epoch
for epoch in range(epochs):

    print('\n Epoch {:} / {:}'.format(epoch + 1, epochs))

    #train model
    train_loss, _ = train()

    #evaluate model
    valid_loss, _ = evaluate()

    #save the best model
    if valid_loss < best_valid_loss:
        best_valid_loss = valid_loss
        torch.save(model.state_dict(), 'saved_weights.pt')

    # append training and validation loss
    train_losses.append(train_loss)
    valid_losses.append(valid_loss)

    print(f'\nTraining Loss: {train_loss:.3f}')
    print(f'Validation Loss: {valid_loss:.3f}')
```

Sample output is shown below.

```
 Epoch 1 / 10
  Batch    50  of    122.
  Batch   100  of    122.

Evaluating...

Training Loss: 0.675
Validation Loss: 0.650

 Epoch 2 / 10
  Batch    50  of    122.
  Batch   100  of    122.

Evaluating...
```

Once the fine-tuning process is completed, the model is loaded with the best weights for performing predictions(Jain,2021).

```
#load weights of best model
path = 'saved_weights.pt'
model.load_state_dict(torch.load(path))
```

```
<All keys matched successfully>
```

### 6.10. Step 10: Model Prediction

In this phase, model predicts whether a message is spam or not on unseen data(Jain,2021).

```
# get predictions for test data
with torch.no_grad():
    preds = model(test_seq.to(device), test_mask.to(device))
    preds = preds.detach().cpu().numpy()
```

### 6.11. Step 11: Model Evaluation

In this step, the accuracy of the model is evaluated(Jain,2021)

```
# model's performance
preds = np.argmax(preds, axis = 1)
print(classification_report(test_y, preds))
```

```
              precision    recall  f1-score   support

           0       0.97      0.88      0.92       724
           1       0.51      0.81      0.63       112

    accuracy                           0.87       836
   macro avg       0.74      0.85      0.77       836
weighted avg       0.91      0.87      0.88       836
```

The recall score shows 0.88 indicates the percentage of correctly predicting spam messages by the model. The accuracy score indicates the overall correctness of model predictions. Overall, the model performance is good.

## 7. Conclusion

This tutorial provided an overview on BERT model. It covered the relevance of the model. It discussed the architecture of Transformers which is used to built BERT. Then, it explained the BERT architecture, the two phases- Pretraining and Fine-tuning. Furthermore, this tutorial briefly explained the fine-tuning of BERT in question

answering task. Then provided a brief overview of the real-world applications, benefits and limitations of BERT. Finally, it covered step by step tutorial for building a model using BERT for spam message classification.

## 8. References


Alammar, J. (2018a) *The illustrated Bert, Elmo, and Co. (how NLP cracked transfer learning)*, *The Illustrated BERT, ELMo, and co. (How NLP Cracked Transfer Learning) – Jay Alammar – Visualizing machine learning one concept at a time.* Available at: http://jalammar.github.io/illustrated-bert/ (Accessed: 05 June 2023).

Alammar, J. (2018b) *The illustrated transformer*, *The Illustrated Transformer – Jay Alammar – Visualizing machine learning one concept at a time.* Available at: http://jalammar.github.io/illustrated-transformer/ (Accessed: 04 June 2023).

Ankit, U. (2020) *Transformer neural network: Step-by-step breakdown of the beast.*, *Medium*. Available at: https://towardsdatascience.com/transformer-neural-network-step-by-step-breakdown-of-the-beast-b3e096dc857f (Accessed: 04 June 2023).

Brownlee, J. (2021) *Gentle introduction to the adam optimization algorithm for deep learning*, *MachineLearningMastery.com*. Available at: https://machinelearningmastery.com/adam-optimization-algorithm-for-deep-learning/ (Accessed: 06 June 2023).

Brownlee, J. (2022) *How to grid search hyperparameters for deep learning models in python with keras*, *MachineLearningMastery.com*. Available at: https://machinelearningmastery.com/grid-search-hyperparameters-deep-learning-models-python-keras/ (Accessed: 06 June 2023).

Chakraborty, S. (2020) *Practical uses of bert*, *Medium*. Available at: https://sayanchak.medium.com/practical-uses-of-bert-c384ae3a5c2a (Accessed: 05 June 2023).

Cristina, S. (2022) *The Transformer model*, *MachineLearningMastery.com*. Available at: https://machinelearningmastery.com/the-transformer-model/ (Accessed: 05 June 2023).

Cristina, S. (2023) *Implementing the transformer encoder from scratch in tensorflow and keras*, *MachineLearningMastery.com*. Available at: https://machinelearningmastery.com/implementing-the-transformer-encoder-from-scratch-in-tensorflow-and-keras/#:~:text=The%20Transformer%20encoder%20consists%20of,%2Dconnected%20feed%2Dforward%20network. (Accessed: 31 May 2023).

DeLucia, S.-A. (2023) *Unleashing the power of bert: How the Transformer model revolutionized NLP*, *Arize AI*. Available at: https://arize.com/blog-course/unleashing-bert-transformer-model-nlp/#:~:text=One%20of%20the%20main%20differences,wide%20range%20of%20NLP%20tasks. (Accessed: 05 June 2023).

Devlin, J. *et al.* (2019) *BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding* [Preprint]. doi:https://doi.org/10.48550/arXiv.1810.04805.

Giacaglia, G. (2019) *Transformers*, *Medium*. Available at: https://towardsdatascience.com/transformers-141e32e69591 (Accessed: 17 May 2023).

Hui, J. (2019) *NLP - Bert & Transformer*, *Medium*. Available at: https://jonathan-hui.medium.com/nlp-bert-transformer-7f0ac397f524 (Accessed: 05 June 2023).

Jain, H. (2021) *Bert for 'everyone' (tutorial + implementation)*, *Kaggle*. Available at: https://www.kaggle.com/code/harshjain123/bert-for-everyone-tutorial-implementation/notebook (Accessed: 06 June 2023).

Jain, S.M. (2022) 'Introduction to Transformers', in *Introduction to transformers for NLP: With the hugging face library and models to solve problems*. New York, Berkeley: Apress, pp. 19–36.

Kana, M. (2019) *Bert for dummies - step by step tutorial*, *Medium*. Available at: https://towardsdatascience.com/bert-for-dummies-step-by-step-tutorial-fb90890ffe03 (Accessed: 05 June 2023).

Karim, R. (2023) *Illustrated: Self-attention*, *Medium*. Available at: https://towardsdatascience.com/illustrated-self-attention-2d627e33b20a (Accessed: 03 June 2023).

Khalid, S. (2020) *Bert explained: A Complete Guide with Theory and tutorial*, *Medium*. Available at: https://medium.com/@samia.khalid/bert-explained-a-complete-guide-with-theory-and-tutorial-3ac9ebc8fa7c (Accessed: 13 May 2023).

KiKaBeN (2021) *Transformer's encoder-decoder: Let's understand the model architecture*, *KiKaBeN*. Available at: https://kikaben.com/transformers-encoder-decoder/ (Accessed: 04 June 2023).

Kortschak, H. (2022) *Attention and Transformer models*, *Medium*. Available at: https://towardsdatascience.com/attention-and-transformer-models-fe667f958378 (Accessed: 03 June 2023).

Kosar, V. (2021) *Feed-forward, self-attention & key-value*, *Vaclav Kosar Blog*. Available at: https://vaclavkosar.com/ml/Feed-Forward-Self-Attendion-Key-Value-Memory (Accessed: 04 June 2023).

Maxime (2019) *What is a Transformer?*, *Medium*. Available at: https://medium.com/inside-machine-learning/what-is-a-transformer-d07dd1fbec04 (Accessed: 04 June 2023).

McCormick, C. (2019) *Bert word embeddings tutorial*, *BERT Word Embeddings Tutorial · Chris McCormick*. Available at: https://mccormickml.com/2019/05/14/BERT-word-embeddings-tutorial/#21-special-tokens (Accessed: 06 June 2023).

Padmanabhan, A. (2022) *Word embedding*, *Devopedia*. Available at: https://devopedia.org/word-embedding (Accessed: 02 June 2023).

Persson, S. (2021) *Paper summary - bert: Pre-training of deep bidirectional Transformers for language understanding*, *Medium*. Available at: https://medium.com/analytics-vidhya/paper-summary-bert-pre-training-of-deep-

bidirectional-transformers-for-language-understanding-861456fed1f9 (Accessed: 05 June 2023).

ProjectPro (2023) *Bert NLP model explained for complete beginners*, *ProjectPro*. Available at: https://www.projectpro.io/article/bert-nlp-model-explained/558 (Accessed: 15 May 2023).

Ravichandran, A. (2022) *How bert NLP optimization model works*, *How BERT NLP Optimization Model Works*. Available at: https://www.turing.com/kb/how-bert-nlp-optimization-model-works#er:-encoder-representations (Accessed: 17 May 2023).

Saeed, M. (2022) *Transformers: What they are and why they matter - blog*, *Scale Virtual Events*. Available at: https://exchange.scale.com/public/blogs/transformers-what-they-are-and-why-they-matter (Accessed: 04 June 2023).

Saeed, M. (2023) *A gentle introduction to positional encoding in transformer models, part 1*, *MachineLearningMastery.com*. Available at: https://machinelearningmastery.com/a-gentle-introduction-to-positional-encoding-in-transformer-models-part-1/ (Accessed: 01 June 2023).

StackExchange (2020) *How is Bert different from the original transformer architecture?*, *Artificial Intelligence Stack Exchange*. Available at: https://ai.stackexchange.com/questions/23221/how-is-bert-different-from-the-original-transformer-architecture (Accessed: 05 June 2023).

Suresh, S.K. (2022) *Understanding transformers-encoder*, *Medium*. Available at: https://medium.com/mlearning-ai/understanding-transformers-encoder-1f269b1cc943 (Accessed: 04 June 2023).

Turing (2022) *Understanding nvidia cuda: The basics of GPU parallel computing*, *Turing*. Available at: https://www.turing.com/kb/understanding-nvidia-cuda (Accessed: 06 June 2023).

Vaswani, A. *et al.* (2017) ‘Attention is All you Need’, in *Advances in neural information processing systems: 31st annual conference on neural information process ..* Long Beach, CA: NEURAL INFO PROCESS SYS F, pp. 6000–60010.