Assignment 10:Understanding the BERT Model

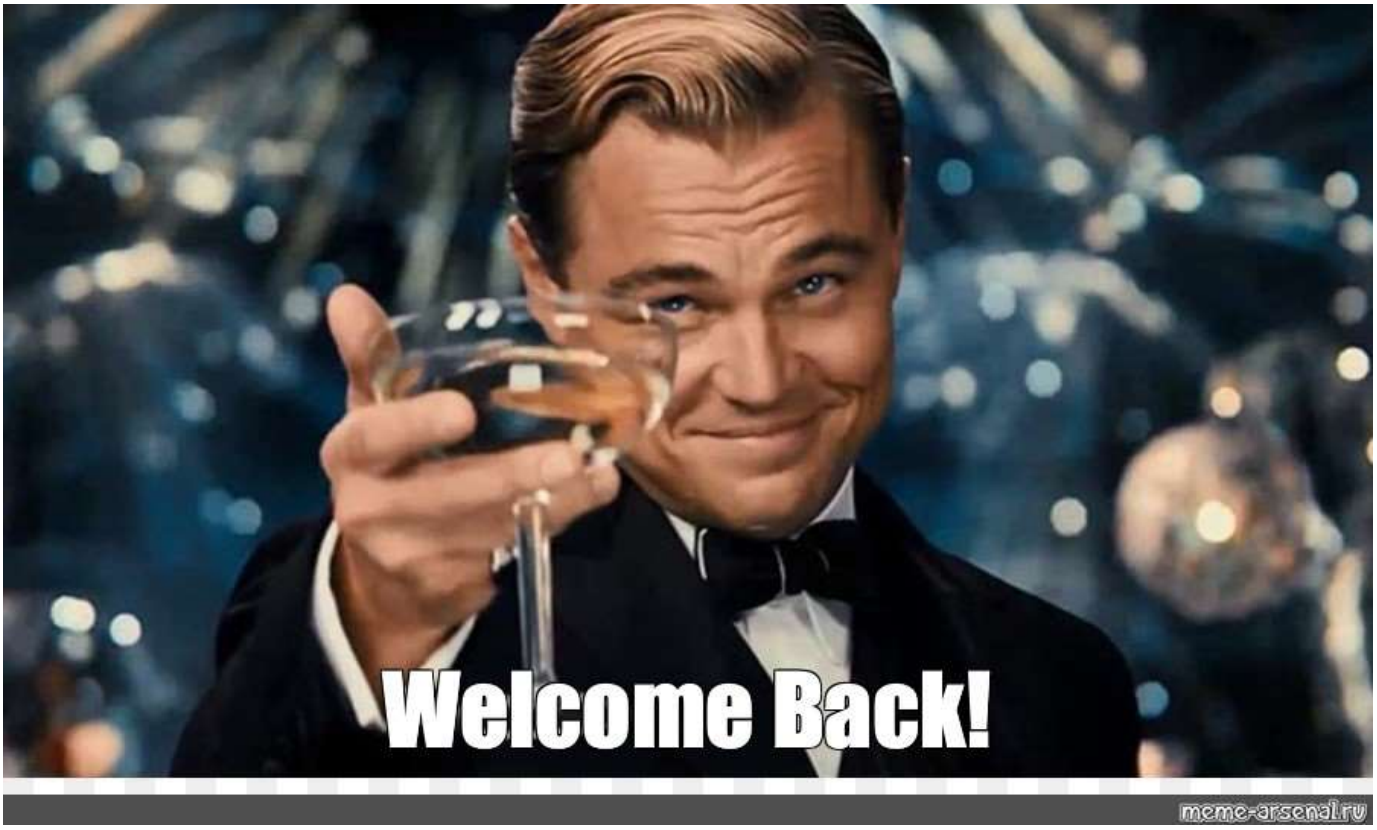


Table of contents:

Basic idea of BERT

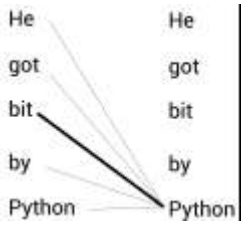
We will get started with one of the most popularly used state-of-the-art text embedding models called BERT. BERT has revolutionized the world of NLP by providing state-of-the-art results on many NLP tasks. We will begin the assignment by understanding what BERT is and how it differs from the other embedding models. We will then look into the working of BERT and its configuration.

BERT stands for **Bidirectional Encoder Representation from Transformer** . It is the state-ofthe-art embedding model published by Google. It has created a major breakthrough in the field of NLP by providing greater results in many NLP tasks, such as question answering, text generation, sentence classification, and many more besides. One of the major reasons for the success of BERT is that it is a context-based embedding model, unlike other popular embedding models, such as word2vec, which are context-free.

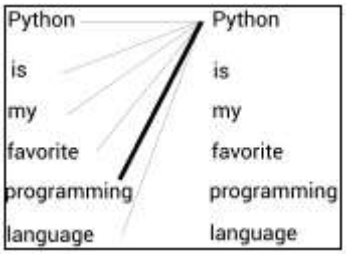
let's understand the difference between context-based and context-free embedding models. Consider the following two sentences:
Sentence A: `He got bit by Python`
Sentence B: `Python is my favorite programming language.`
By reading the preceding two sentences, we can understand that the meaning of the word '`Python`' is different in both sentences. In sentence A, the word '`Python`' refers to the snake, while in sentence B, the word '`Python`' refers to the programming language.

Now, if we get embeddings for the word '`Python`' in the preceding two sentences using an embedding model such as word2vec, the embedding of the word '`Python`' would be the same in both sentences, and so this renders the meaning of the word '`Python`' the same in both sentences. This is because word2vec is the context-free model, so it will ignore the context and always give the same embedding for the word '`Python`' irrespective of the context.

BERT, on the other hand, is a context-based model. It will understand the context and then generate the embedding for the word based on the context. So, for the preceding two sentences, it will give different embeddings for the word 'Python' based on the context.
Let's take sentence A: `He got bit by Python`. First, BERT relates each word in the sentence to all the other words in the sentence to understand the contextual meaning of every word.



let's take sentence B: Python is my favorite programming language. Similarly, here BERT relates each word in the sentence to all the words in the sentence to understand the contextual meaning of every word.



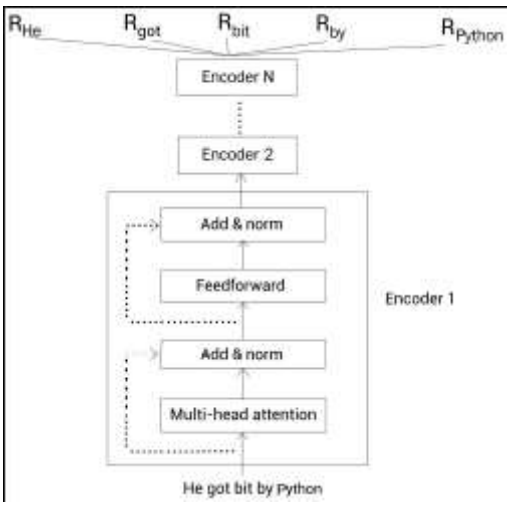

```
###Refer Video
from IPython.display import YouTubeVideo
YouTubeVideo('ioGry-89gqE', width=600, height=300)
```

Working of BERT

Bidirectional Encoder Representation from Transformer (BERT), as the name suggests, is based on the transformer model. We can perceive BERT as the transformer, but only with the encoder. The encoder of the transformer is bidirectional in nature since it can read a sentence in both directions. Thus, BERT is basically the Bidirectional Encoder Representation obtained from the Transformer.

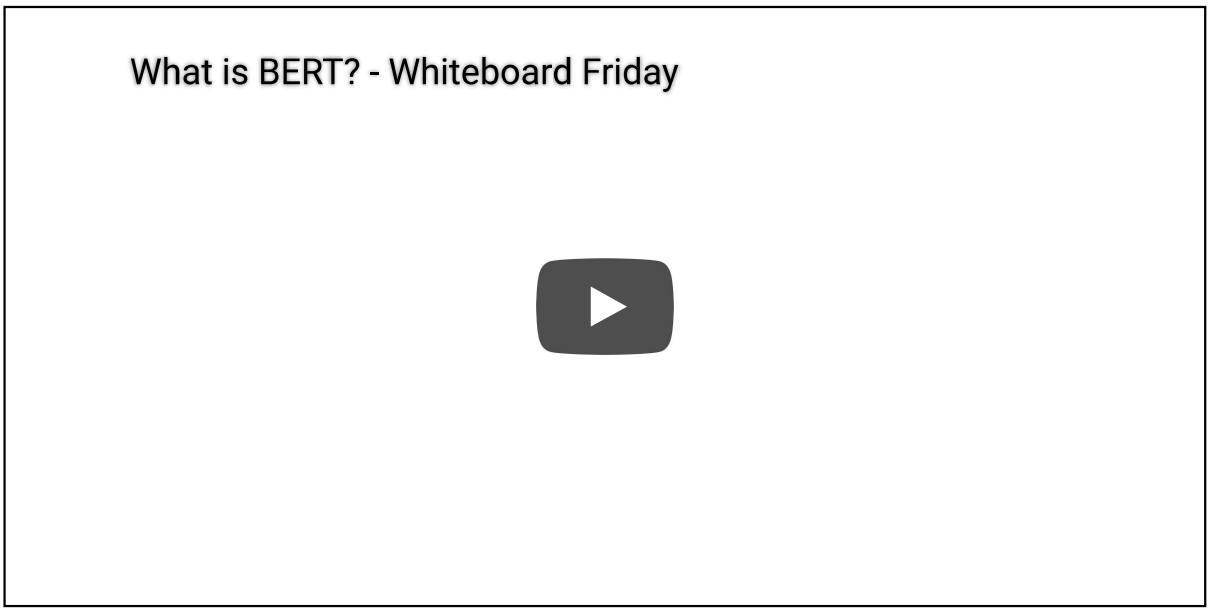
we have a sentence A: 'He got bit by Python'. Now, we feed this sentence as an input to the transformer's encoder and get the contextual representation (embedding) of each word in the

sentence as an output. Once we feed the sentence as an input to the encoder, the encoder understands the context of each word in the sentence using the multi-head attention mechanism (relates each word in the sentence to all the words in the sentence to learn the relationship and contextual meaning of words) and returns the contextual representation of each word in the sentence as an output. As shown in the following diagram, we feed the sentence as an input to the transformer's encoder and get the representation of each word in the sentence as an output. We can stack up N number of encoders.



Thus, with the BERT model, for a given sentence, we obtain the contextual representation (embedding) of each word in the sentence as an output. Now that we understand how BERT generates contextual representation.

```
###Refer Video
YouTubeVideo('Owo36iI6hLA', width=600, height=300)
```

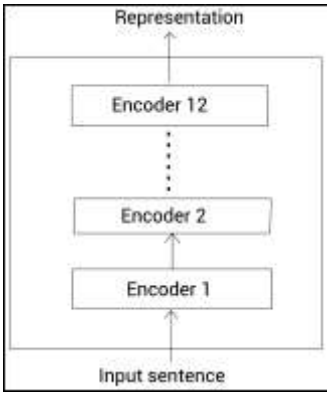


Configurations of BERT The researchers of BERT have presented the model in two standard configurations:

1. BERT-base
2. BERT-large

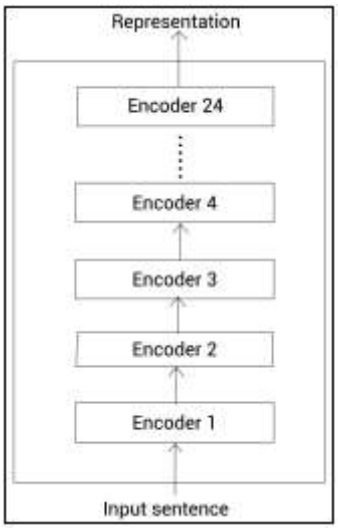
BERT-base

BERT-base consists of 12 encoder layers, each stacked one on top of the other. All the encoders use 12 attention heads. The feedforward network in the encoder consists of 768 hidden units. Thus, the size of the representation obtained from BERT-base will be 768.



BERT-large

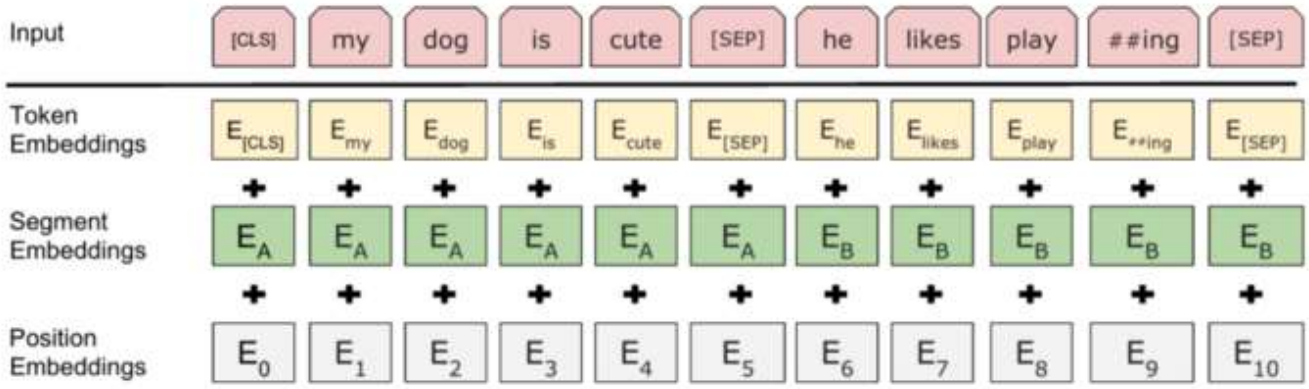
BERT-large consists of 24 encoder layers, each stacked one on top of the other. All the encoders use 16 attention heads. The feedforward network in the encoder consists of 1,024 hidden units. Thus, the size of the representation obtained from BERT-large will be 1,024.



There are two steps in BERT:

pre-training and fine-tuning. During pre-training, the model is trained on unlabeled data over different pre-training tasks.
For fine-tuning, the BERT model is first initialized with the pre-trained parameters, and all of the parameters are fine-tuned using labeled data from the downstream tasks.

BERT Representations:



we'll have a quick glance through the input/output representations used in BERT.

1. **CLS:** This token is called as the classification token. It is used at the beginning of a sequence.
2. SEP: This token indicates the separation of 2 sequences i.e. it acts as a delimiter.
3. **MASK:** Used to indicate masked token in the MLM task.
4. **Segment Embeddings** are used to indicate the sequence to which a token belongs ie. if there are multiple sequences separated by a SEP token in the input, then along with the Positional Embeddings(from Transformers), these are added to the original Word Embeddings for the Model to identify the sequence of the token.

Pre-training the BERT model

The BERT paper proposes 2 pre-training tasks:

1. Masked Language Modeling (MLM)
2. Next Sentence Prediction (NSP)

Fine-tuning on downstream tasks such as:

1. Sentiment analysis
2. Natural language inference
3. Named Entity Recognition (NER)
4. Question-answering

The Transformer Encoder gives BERT its Bidirectional Nature, as it uses tokens from both the directions to attend a given token.

1. Masked Language Model (MLM)
   Masked language modeling does not require training a model with a sequence of visible words followed by a masked sequence to predict.
   BERT introduces the bidirectional analysis of a sentence with a random mask on a word of the sentence.

A potential input sequence could be:

`The cat sat on it because it was a nice rug.`

The decoder would mask the attention sequence after the model reached the word "it":

`"The cat sat on it <masked sequence>."`

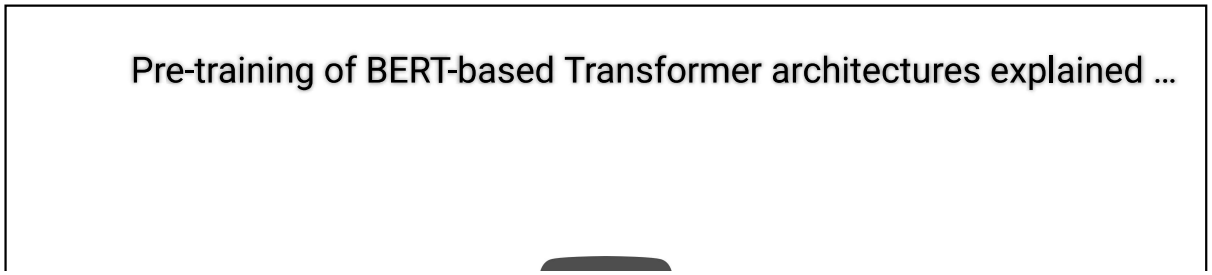
But the BERT encoder masks a random token to make a prediction:

`"The cat sat on it [MASK] it was a nice rug."`

The fine-tuning task is in no way going to see the MASK token in its input. So, for the model to adapt these cases, 80% of the time, the 15% tokens are masked; 10% of the time, 15% tokens are replaced with random tokens; and 10% of the time, they are kept as it is i.e. untouched.

- 80% of the time: Replace the word with the [MASK] token, e.g., `my dog is hairy` → `my dog is [MASK]`
- 10% of the time: Replace the word with a random word, e.g., `my dog is hairy` → `my dog is apple`
- 10% of the time: Keep the word unchanged, e.g., `my dog is hairy` → `my dog is hairy`. The purpose of this is to bias the representation towards the actual observed word.

```
###Refer Video
YouTubeVideo('dabFOBE4eZI', width=600, height=300)
```



2. Next Sentence Prediction (NSP):
   The second method found to train BERT is Next Sentence Prediction (NSP). The input contains two sentences.
   Two new tokens were added:
3. **CLS** is a binary classification token added to the beginning of the first sequence to predict if the second sequence follows the first sequence. A positive sample is usually a pair of consecutive sentences taken from a dataset. A negative sample is created using sequences from different documents.
4. **SEP** is a separation token that signals the end of a sequence.

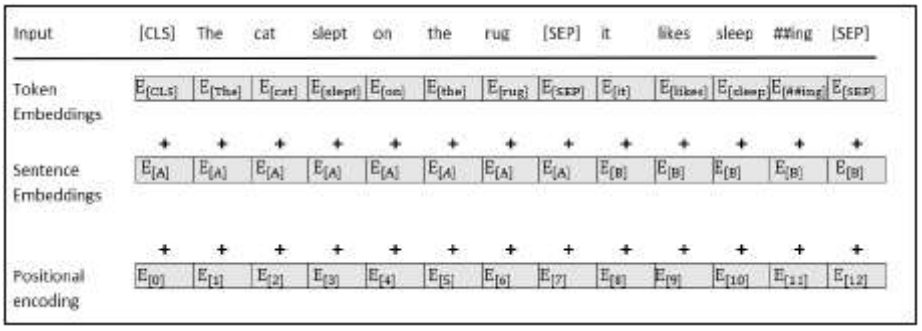
For example, the input sentences taken out of a book could be:

```
"The cat slept on the rug. It likes sleeping all day."
```

These two sentences would become one input complete sequence:

```
[CLS] the cat slept on the rug [SEP] it likes sleep ##ing all day[SEP]
```

If we put the whole embedding process together, we obtain:



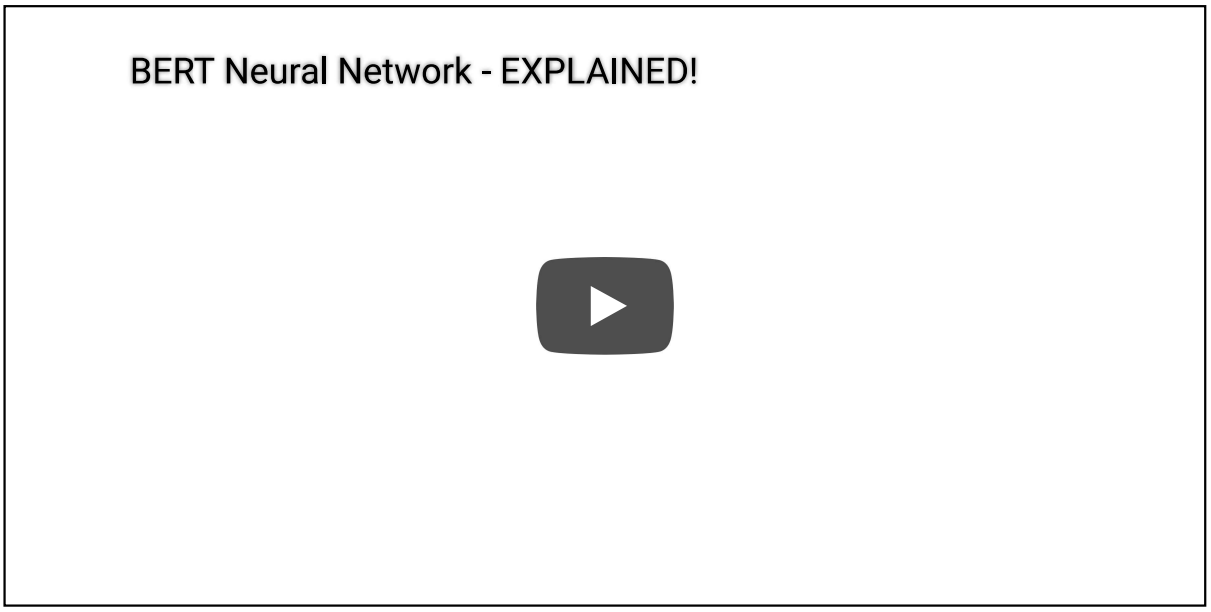
The input embeddings are obtained by summing the token embeddings, the segment (sentence, phrase, word) embeddings, and the positional encoding embeddings.

The input embedding and positional encoding sub-layer of a BERT model can be summed up as follows:

1. A sequence of words is broken down into WordPiece tokens.
2. A [MASK] token will randomly replace the initial word tokens for masked language modeling training.
3. A [CLS] classification token is inserted at the beginning of a sequence for classification purposes.
4. A [SEP] token separates two sentences (segments, phrases) for NSP training.

5. Sentence embedding is added to token embedding, so that sentence A has a different sentence embedding value than sentence B.
6. Positional encoding is learned. The sine-cosine positional encoding method of the original Transformer is not applied.

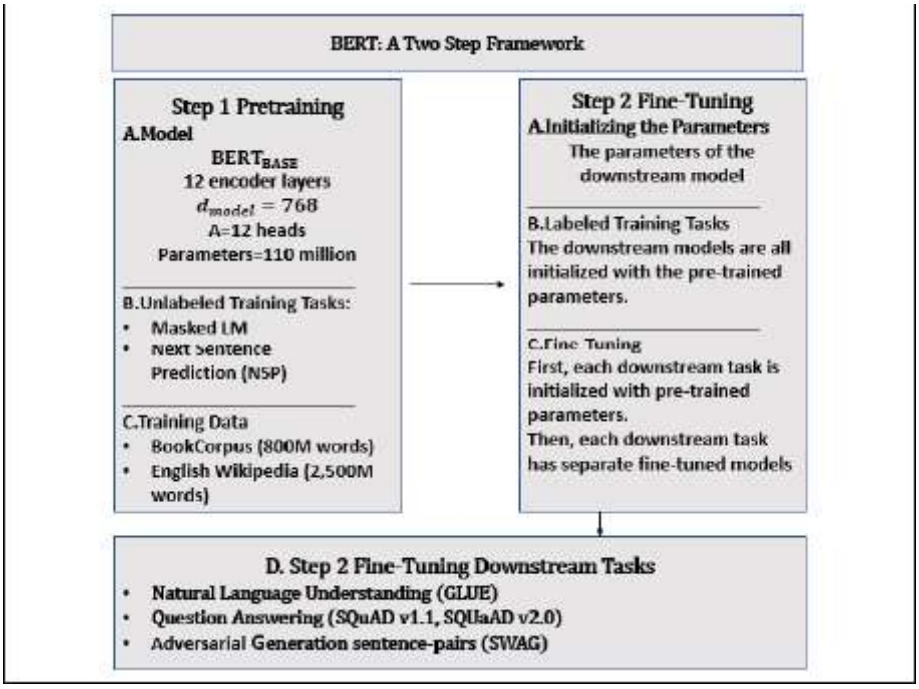
```
###Refer Video
YouTubeVideo('xI0HHN5XKDo', width=600, height=300)
```



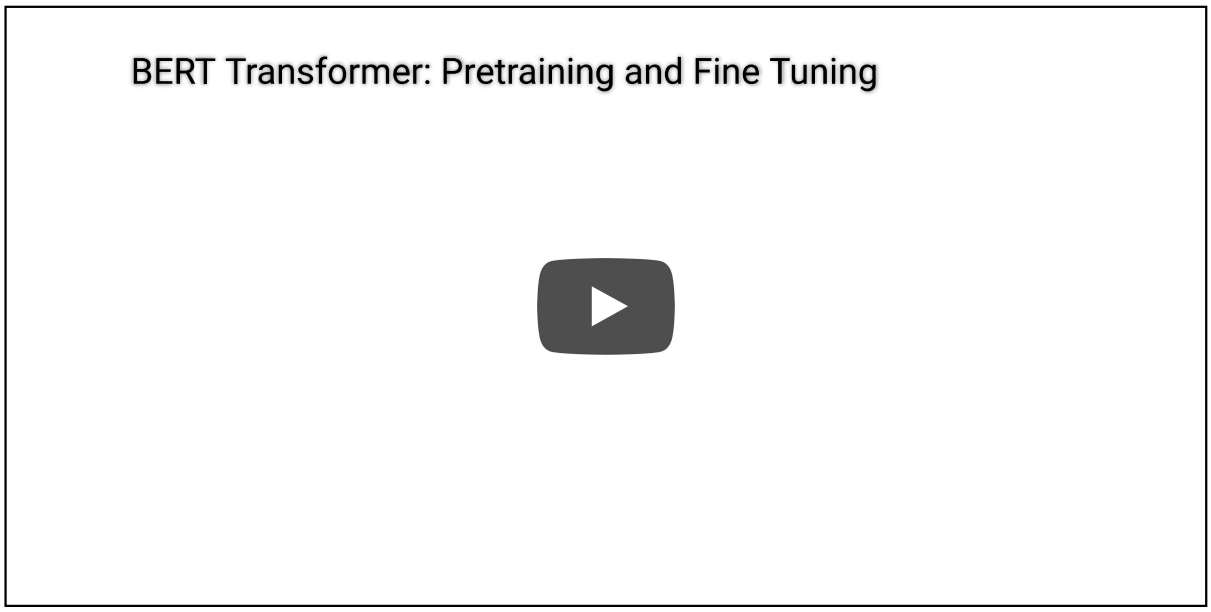
Pretraining and fine-tuning a BERT model

BERT is a two-step framework.

The first step is the pretraining, and the second is fine-tuning.



```
###Refer Video
YouTubeVideo('4TcUf9OHGR8', width=600, height=300)
```



Pretraining is the first step of the BERT framework that can be broken down into two sub-steps:

1. Defining the model's architecture: number of layers, number of heads, dimensions, and the other building blocks of the model
2. Training the model on Masked Language Modeling (MLM) and NSP tasks

The second step of the BERT framework is fine-tuning, which can also be broken down into two sub-steps:

1. Initializing the downstream model chosen with the trained parameters of the pretrained BERT model
2. Fine-tuning the parameters for specific downstream tasks such as Recognizing Textual Entailment, Question Answering, and Situations With Adversarial Generations.

Fine Tuning for sentiment analysis

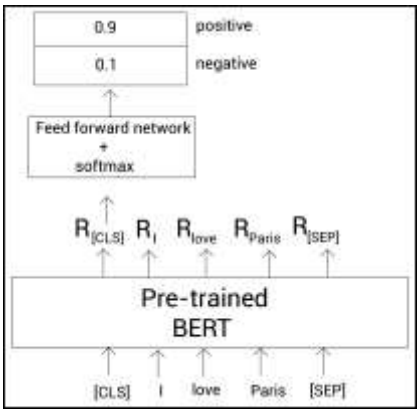
Let's learn how to fine-tune the pre-trained BERT model for a text classification task. Say we are performing sentiment analysis. In the sentiment analysis task, our goal is to classify whether a sentence is positive or negative. Suppose we have a dataset containing sentences along with their labels.

Consider a sentence: `I love Paris.`

First, we tokenize the sentence, add the CLS token at the beginning, and add the SEP token at the end of the sentence. Then, we feed the tokens as an input to the pre-trained BERT model and get the embeddings of all the tokens.

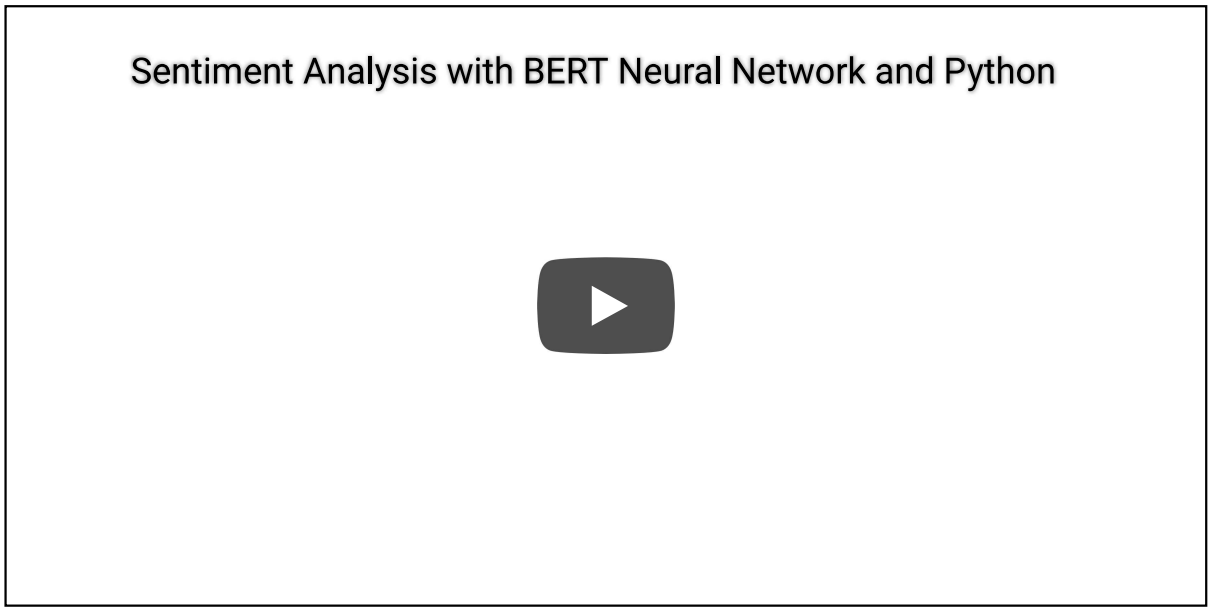
The difference is that when we fine-tune the pre-trained BERT model, we update the weights of

the model along with a classifier. But when we use the pre-trained BERT model as a feature extractor, we update only the weights of the classifier and not the pretrained BERT model.



As we can observe from the preceding figure, we feed the tokens to the pre-trained BERT model and get the embeddings of all the tokens. We take the embedding of the [CLS] token and feed it to a feedforward network with a softmax function and perform classification.

```
###Refer Video
YouTubeVideo('szczpgOEdXs', width=600, height=300)
```



Fine tune for Question-answering(QA):

In a question-answering task, we are given a question along with a paragraph containing an answer to the question. Our goal is to extract the answer from the paragraph for the given question. Now, let's learn how to fine-tune the pre-trained BERT model to perform a question-answering task. The input to the BERT model will be a question-paragraph pair. That is, we feed a question and a paragraph containing the answer to the question to BERT and it has to extract the answer from the paragraph. So, essentially, BERT has to return the text span that contains the answer from the paragraph. Let's understand this with an example – consider the following

question-paragraph pair

'Question = "`What is the immune system?`"

Paragraph = "The immune system is a system of many biological structures and processes within an organism that protects against disease. To function properly, an immune system must detect a wide variety of agents, known as pathogens, from viruses to parasitic worms, and distinguish them from the organism's own healthy tissue."

Now, our model has to extract an answer from the paragraph; it essentially has to return the text span containing the answer. So, it should return the following:

Answer = "`a system of many biological structures and processes within an organism that protects against disease`"

Now, how do we find the starting and ending index of the text span containing the answer? If we get the probability of each token (word) in the paragraph of being the starting andending token (word) of the answer, then we can easily extract the answer, right? Yes, but how we can achieve this? To do this, we use two vectors called the start vector and the end vector . The values of the start and end vectors will be learned during training.

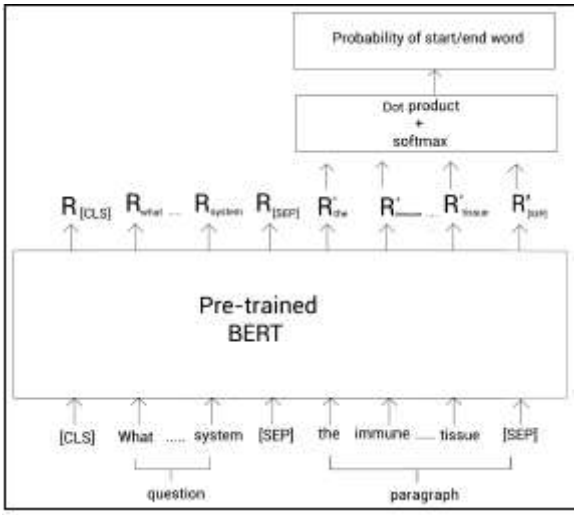
First, we compute the probability of each token (word) in the paragraph being the starting token of the answer. To compute this probability, for each token i, we compute the dot product between the representation of the token Ri and the start vector S. we apply the softmax function to the dot product

$$P_i = \frac{e^{S.R_i}}{\sum_j e^{S.R_j}}$$

we compute the starting index by selecting the index of the token that has a high probability of being the starting token. In a very similar fashion, we compute the probability of each token (word) in the paragraph being the ending token of the answer. To compute this probability, for each token i, we compute the dot product between the representation of the token R and the end vector .

$$P_i = \frac{e^{E.R_i}}{\sum_j e^{E.R_j}}$$

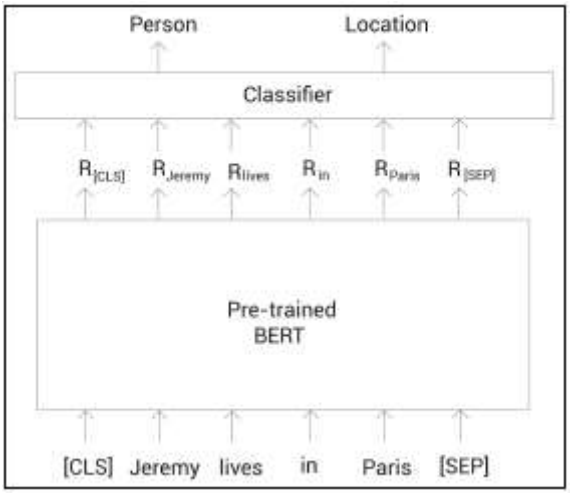
we compute the ending index by selecting the index of the token that has a high probability of being the ending token. Now, we can select the text span that contains the answer using the starting and ending index.



From the preceding figure, we can see how we compute the probability of each token in the paragraph being the start/end word. Next, we select the text span containing the answer using the starting and ending indexes with the highest probability. To get a better understanding of how this works, let's see how to use the fine-tuned question-answering BERT model in the next section.

Fine tune for Named entity recognition:

In NER, our goal is to classify named entities into predefined categories. For instance, consider the sentence Jeremy lives in Paris. In this sentence, "Jeremy" should be categorized as a person, and "Paris" should be categorized as a location.
let's learn how to fine-tune the pre-trained BERT model to perform NER. First, we tokenize the sentence, then we add the [CLS] token at the beginning and the [SEP] token at the end. Then, we feed the tokens to the pre-trained BERT model and obtain the representation of every token. Next, we feed those token representations to a classifier (feedforward network + softmax function). Then, the classifier returns the category to which the named entity belongs.



We can fine-tune the pre-trained BERT model for several downstream tasks. So far, we have learned how BERT works and also how to use the pre-trained BERT model.

We started the chapter by looking at different configurations of the pre-trained BERT model provided by Google. Then, we learned that we can use the pre-trained BERT model in two ways: as a feature extractor by extracting embeddings, and by fine-tuning the pre-trained BERT model for downstream tasks such as text classification, question-answering, and more. We also learned how to use Hugging Face's transformers library to generate embeddings. Then, we learned how to extract embeddings from all the encoder layers of BERT in detail. Moving on, we learned how to fine-tune pre-trained BERT for downstream tasks. We learned how to fine-tune BERT for text classification, NLI, NER, and question-answering in detail.

Question Answering project is very easy to build with BERT because we can use pre-defined file and make our things easy. When we ask some question, it should return the correct answer. Follow these steps:
Step 1: Install Required Libraries
Step 2: Importing required packages
Step 3: Load pre-trained Bert model
Step 4: Define function for question answering
Step 5:Let's use the passage and check the output

Install Required Libraries

First we will install transformers and torch using the pip install command.

```
#install transformers and torch
```

```
Collecting transformers
  Downloading transformers-4.15.0-py3-none-any.whl (3.4 MB)
     |████████████████████████████████| 3.4 MB 27.9 MB/s
Collecting tokenizers<0.11,>=0.10.1
  Downloading tokenizers-0.10.3-cp37-cp37m-manylinux_2_5_x86_64.manylinux1_x86_64.mar
     |████████████████████████████████| 3.3 MB 48.9 MB/s
Collecting huggingface-hub<1.0,>=0.1.0
  Downloading huggingface_hub-0.4.0-py3-none-any.whl (67 kB)
     |████████████████████████████████| 67 kB 4.2 MB/s
Requirement already satisfied: numpy>=1.17 in /usr/local/lib/python3.7/dist-packages
Collecting pyyaml>=5.1
  Downloading PyYAML-6.0-cp37-cp37m-manylinux_2_5_x86_64.manylinux1_x86_64.manylinux_
     |████████████████████████████████| 596 kB 55.2 MB/s
Requirement already satisfied: importlib-metadata in /usr/local/lib/python3.7/dist-pa
Collecting sacremoses
  Downloading sacremoses-0.0.47-py2.py3-none-any.whl (895 kB)
     |████████████████████████████████| 895 kB 44.3 MB/s
Requirement already satisfied: tqdm>=4.27 in /usr/local/lib/python3.7/dist-packages (
Requirement already satisfied: requests in /usr/local/lib/python3.7/dist-packages (fr
Requirement already satisfied: packaging>=20.0 in /usr/local/lib/python3.7/dist-packa
Requirement already satisfied: filelock in /usr/local/lib/python3.7/dist-packages (fr
Requirement already satisfied: regex!=2019.12.17 in /usr/local/lib/python3.7/dist-pac
```

```
Requirement already satisfied: typing-extensions>=3.7.4.3 in /usr/local/lib/python3.7
Requirement already satisfied: pyparsing!=3.0.5,>=2.0.2 in /usr/local/lib/python3.7/d
Requirement already satisfied: zipp>=0.5 in /usr/local/lib/python3.7/dist-packages (f
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/python3.7/dist-pa
Requirement already satisfied: chardet<4,>=3.0.2 in /usr/local/lib/python3.7/dist-pac
Requirement already satisfied: urllib3!=1.25.0,!=1.25.1,<1.26,>=1.21.1 in /usr/local/
Requirement already satisfied: idna<3,>=2.5 in /usr/local/lib/python3.7/dist-packages
Requirement already satisfied: six in /usr/local/lib/python3.7/dist-packages (from sa
Requirement already satisfied: click in /usr/local/lib/python3.7/dist-packages (from
Requirement already satisfied: joblib in /usr/local/lib/python3.7/dist-packages (from
Installing collected packages: pyyaml, tokenizers, sacremoses, huggingface-hub, trans
  Attempting uninstall: pyyaml
    Found existing installation: PyYAML 3.13
    Uninstalling PyYAML-3.13:
      Successfully uninstalled PyYAML-3.13
Successfully installed huggingface-hub-0.4.0 pyyaml-6.0 sacremoses-0.0.47 tokenizers-
Requirement already satisfied: torch in /usr/local/lib/python3.7/dist-packages (1.10
Requirement already satisfied: typing-extensions in /usr/local/lib/python3.7/dist-pac
```

Next we will import the required pacakges from the installed libraries.

- BertForQuestionAnswering: to implement bert model for question answering task.
- BertTokenizer: for tokenization of text
- torch: for tensor computation
- numpy: for array and matrix operations

```
#import above mentioned libraries
```

Now we will create the model from the pretrained bert model for question answering. Use the from_pretrained function to do so. We will be using the 'bert-large-uncased-whole-word-masking-finetuned-squad' pretrained model.
Refer: https://huggingface.co/bert-large-uncased-whole-word-masking-finetuned-squad.

```
#load the pretrained model
```

```
Downloading: 100%                    443/443 [00:00<00:00, 6.77kB/s]
Downloading: 100%                    1.25G/1.25G [00:29<00:00, 44.1MB/s]
```

Next load the tokenizer for the same model using the BertTokenizer.from_pretrained function.
Refer:
https://www.programcreek.com/python/example/124529/pytorch_pretrained_bert.tokenization.BertTokenizer.from_pretrained

```
#load the bert tokenizer
```

```
Downloading: 100%  226k/226k [00:00<00:00, 4.53MB/s]
Downloading: 100%  28.0/28.0 [00:00<00:00, 790B/s]
Downloading: 100%  455k/455k [00:00<00:00, 6.17MB/s]
```

Our main function starts from here. We have created a function that takes question, passage and max length as parameters. The max length is set to 500 aa default that means our passage length would be 500 or less than that. We have a given question and short passage example,

```
question: What is the name of YouTube Channel
passage: Watch complete playlist of Natural Language Processing. Don't forget to like, share and subscribe my channel CloudyML.
```

We will pass the question and passage when we will call the function. First in the function, we have to tokenize the inputs question and passage and then we will enocde them that means representing each token by thier Ids. After that we have to calculate length of question and passage. Before this we have to get index of SEP token then our question will be sep_index+1 because index always starts from 0 and remaining will be length of passage.

Next we need to seperate the question and passage, we can provide segment_id of question as 0 and passage as 1. We can also convert ids into tokens. So it will return CLS,my question and SEP.

Next step is to calculate start and end token scores so this score will encapsulate the answer. For this we have to pass input Ids and segment IDs to our model. From start_token_scores we can find the start position of our answer and end_token_scores we will get end of our answer so this score is calculated from pre trained model.

In next steps we have to convert score from tensor to numpy array because we will apply some function of numpy, Now we have to remove ## hash sign from the question and passage as we have seen eariler.

Finally our function is completed.

```
def bert_question_answer(question, passage, max_len=500):

    """
    question: What is the name of YouTube Channel
    passage: Watch complete playlist of Natural Language Processing. Don't forget to like,
    """

    #Tokenize input question and passage
    #Add special tokens - [CLS] and [SEP]


    """
```

```
[101, 2054, 2003, 1996, 2171, 1997, 7858, 3149, 102, 3422, 3143, 2377, 9863, 1997, 301
2123, 1005, 1056, 5293, 2000, 2066, 1010, 3745, 1998, 4942, 29234, 2026, 3149, 1045, 2
We can see that 101 and 102 is [CLS] and [SEP] indicates that Starting and ending of s
"""

#Getting number of tokens in 1st sentence (question) and 2nd sentence (passage that co




"""
8
9
27
"""

#Need to separate question and passage
#Segment ids will be 0 for question and 1 for passage

"""
[0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1
"""

#Converting token ids to tokens

"""
tokens = ['[CLS]', 'what', 'is', 'the', 'name', 'of', 'youtube', 'channel', '[SEP]', '
'play', '##list', 'of', 'natural', 'language', 'processing', '.', 'don', "'", 't', 'fo
',', 'share', 'and', 'sub', '##scribe', 'my', 'channel', 'i', '##g', 'CLoudyML, '[SEP]
"""

#Getting start and end scores for answer
#Converting input arrays to torch tensors before passing to the model


"""
tensor([[-5.9787, -3.0541, -7.7166, -5.9291, -6.8790, -7.2380, -1.8289, -8.1006,
     -5.9786, -3.9319, -5.6230, -4.1919, -7.2068, -6.7739, -2.3960, -5.9425,
     -5.6828, -8.7007, -4.2650, -8.0987, -8.0837, -7.1799, -7.7863, -5.1605,
     -8.2832, -5.1088, -8.1051, -5.3985, -6.7129, -1.4109, -3.2241,  1.5863,
     -4.9714, -4.1138, -5.9107, -5.9786]], grad_fn=<SqueezeBackward1>)
tensor([[-2.1025, -2.9121, -5.9192, -6.7459, -6.4667, -5.6418, -1.4504, -3.1943,
     -2.1024, -5.7470, -6.3381, -5.8520, -3.4871, -6.7667, -5.4711, -3.9885,
     -1.2502, -4.0869, -6.4930, -6.3751, -6.1309, -6.9721, -7.5558, -6.4056,
     -6.7456, -5.0527, -7.3854, -7.0440, -4.3720, -3.8936, -2.1085, -5.8211,
     -2.0906, -2.2184,  1.4268, -2.1026]], grad_fn=<SqueezeBackward1>)
"""

#Converting scores tensors to numpy arrays


"""
[-5.978666  -3.0541189 -7.7166095 -5.929051  -6.878973  -7.238004
-1.8289301 -8.10058   -5.9786286 -3.9319289 -5.6229596 -4.191908
-7.20684   -6.773916  -2.3959794 -5.942456  -5.6827617 -8.700695
```

```
-4.265001  -8.09874   -8.083673  -7.179875  -7.7863474 -5.16046
-8.283156  -5.108819  -8.1051235 -5.3984528 -6.7128663 -1.4108785
-3.2240815  1.5863497 -4.9714    -4.113782  -5.9107194 -5.9786243]

[-2.1025064 -2.912148  -5.9192414 -6.745929  -6.466673  -5.641759
-1.4504088 -3.1943028 -2.1024144 -5.747039  -6.3380575 -5.852047
-3.487066  -6.7667046 -5.471078  -3.9884708 -1.2501552 -4.0868535
-6.4929943 -6.375147  -6.130891  -6.972091  -7.5557766 -6.405638
-6.7455807 -5.0527067 -7.3854156 -7.043977  -4.37199   -3.8935976
-2.1084964 -5.8210607 -2.0906193 -2.2184045  1.4268283 -2.1025767]
"""
#Getting start and end index of answer based on highest scores


"""
31
34
"""

#Getting scores for start and end token of the answer


"""
1.59
1.43
"""

#Combining subwords starting with ## and get full words in output.
#It is because tokenizer breaks words which are not in its vocab.







# If the answer didn't find in the passage
```

```
#Testing function
bert_question_answer("What is the name of YouTube Channel", " Don't forget to like, share
```

```
 (22, 23, 6.49, 7.19, 'cloudyml')
```

Next lets try with some another passage. Define a passage of your choice and also define questions based on the passage. Apply the above defined function to the questions and display the answers.

```
# Let me define another passage
passage= """NLP is a subfield of computer science and artificial intelligence concerned wi
```

```
computers and human (natural) languages. It is used to apply machine learning algorithms t
example, we can use NLP to create systems like speech recognition, document summarization,
detection, named entity recognition, question answering, autocomplete, predictive typing a
us have smartphones that have speech recognition. These smartphones use NLP to understand
people use laptops which operating system has a built-in speech recognition. NLTK (Natural
leading platform for building Python programs to work with human language data. It provide
to many corpora and lexical resources. Also, it contains a suite of text processing librar
tokenization, stemming, tagging, parsing, and semantic reasoning. Best of all, NLTK is a f
community-driven project. We'll use this toolkit to show some basics of the natural langua
the examples below, I'll assume that we have imported the NLTK toolkit. We can do this lik
Sentence tokenization (also called sentence segmentation) is the problem of dividing a str
its component sentences. The idea here looks very simple. Word tokenization (also called w
is the problem of dividing a string of written language into its component words. In Engli
using some form of Latin alphabet, space is a good approximation of a word divider. Howeve
we only split by space to achieve the wanted results. Some English compound nouns are vari
they contain a space. In most cases, we use a library to achieve the wanted results, so ag
for the details. Stop words are words which are filtered out before or after processing of
learning to text, these words can add a lot of noise. That's why we want to remove these i
Stop words usually refer to the most common words such as "and", "the", "a" in a language,
universal list of stopwords. The list of the stop words can change depending on your appli
a predefined list of stopwords that refers to the most common words. If you use it for you
download the stop words using this code: nltk.download("stopwords"). Once we complete the
the stopwords package from the nltk.corpus and use it to load the stop words."""

#print the length of the passage


     Length of the passage: 433 words
```

Below is the code for accepting question from the user and giving the answer.

```
# Turn off warnings
import transformers
transformers.logging.set_verbosity_error()


#define 1st question
question ="What is full form of NLTK"
print ('\nQuestion 1:\n', question)
#apply bert_question_answer to 1st question

#print the answer



     Question 1:
      What is full form of NLTK

     BERT ANSWER:  natural language toolkit



#define 2nd question
question ="What are stop words "
```

```
print ('\nQuestion 2:\n', question)
#apply bert_question_answer to 2nd question

#print the answer
```

```
Question 2:
 What are stop words

BERT ANSWER:  words which are filtered out before or after processing of text
```

```
#define 3rd question
question ="What is NLP "
print ('\nQuestion 3:\n', question)
#apply bert_question_answer to 3rd question

#print the answer
```

```
Question 3:
 What is NLP

Answer from BERT:  a subfield of computer science and artificial intelligence concerr
```

```
#define 4th question
question ="How to download stop words from nltk"
print ('\nQuestion 4:\n', question)
#apply bert_question_answer to 4th question

#print the answer
```

```
Question 4:
 How to download stop words from nltk

Answer from BERT:  import nltk
```

```
#define 5th question
question ="What is supervised learning"
print ('\nQuestion 6:\n', question)
#apply bert_question_answer to 5th question

#print the answer
```

```
Question 6:
```

```
What is supervised learning

Answer from BERT:  Sorry!, I could not find an answer in the passage.
```

Now let's try with in interface. Now you change the questions and passage as you need :).
For better understanding of model try with your question and passage. So that you will get clear idea of QA models of bert -_-

```
#@title Question-Answering Application { vertical-output: true }
#@markdown ---
#define question
question= "" #@param {type:"string"}
#define passage
passage = "" #@param {type:"string"}
#@markdown ---
#apply bert_question_answer
answer_start_index, answer_end_index, start_token_score, end_token_score,ans
#print the answer
#@markdown Answer:
print(ans)
```

Question-Answering Application

question: " Insert text here "

passage: " Insert text here "

Answer:

```
Enter your inputs Question and Passage
```

Great job!! You have come to the end of this assignment. Treat yourself for this :))

Do fill this feedback form

You may head on to the next project section.

✓ 0s completed at 6:41 PM