An Explanatory Guide to BERT Tokenizer

Pritesh Prakash
10 Sep, 2021 • 9 min read

This article was published as a part of the Data Science Blogathon

Introduction

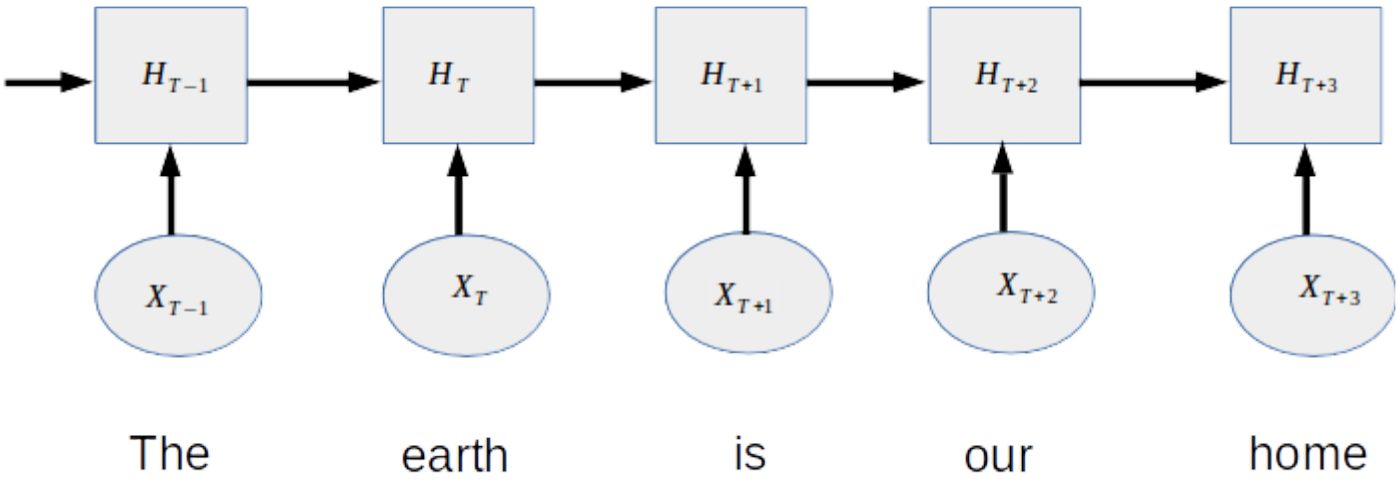
In this article, you will learn about the input required for BERT in the classification or the question answering system development. This article will also make your concept very much clear about the Tokenizer library. Before diving directly into BERT let's discuss the basics of LSTM and input embedding for the transformer.

What is LSTM?

LSTM was used first in sequence to sequence for the NLP solution. In the seq2seq model, the team also took those input samples in which they reversed the source sentences but not the target sentences and they achieved a higher score with this. The dimension of the embedding vector also improves the score for example in English different sizes of Glove embedding are available 50, 100, 200, etc. In short, it always matters the way you feed the data to your network. The LSTM became popular due to its learning capability for long-term sequences. In NLP tasks LSTM can learn the word sequences in the sentence.

Refer to the image below the position of the word 'our' is after the words 'the', 'earth' and 'is' and our neural net is to be capable of learning these sequences. LSTM network sequentially takes input, and

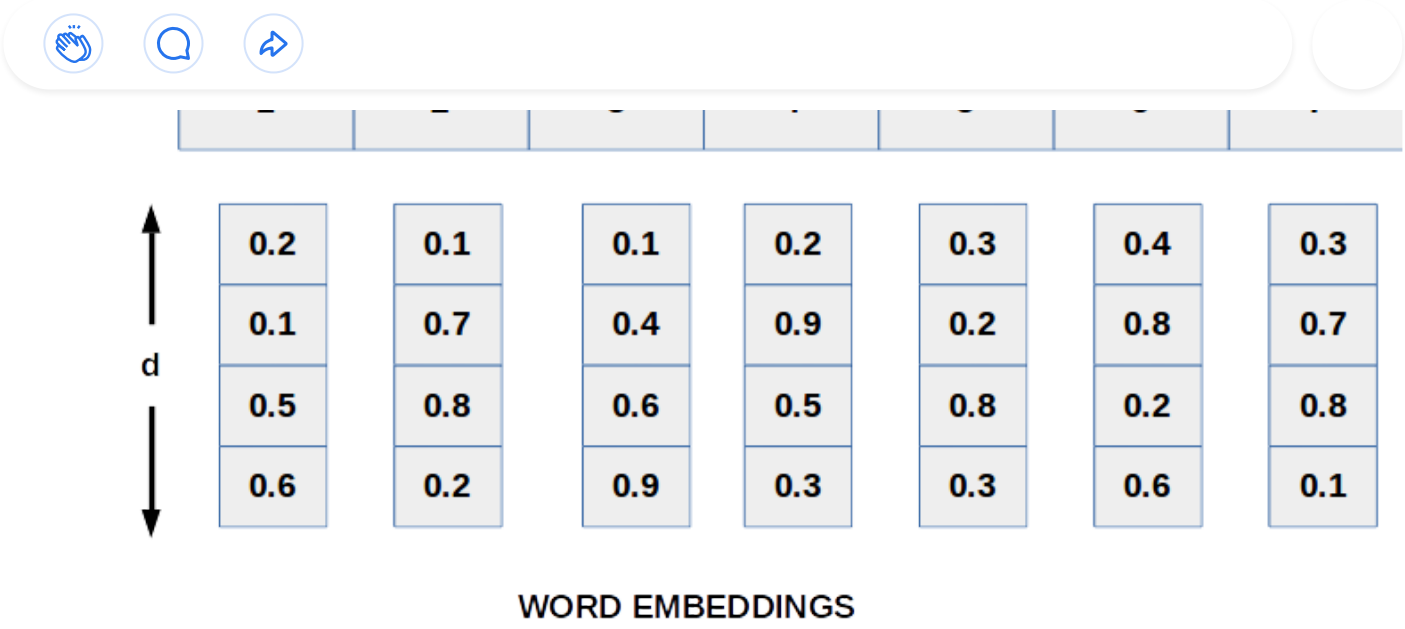
this is also a problem that we can not feed the entire sentence all at once. If we do so in any customized network then preserving the positional information is also a challenge.



Sequential Learning of RNN

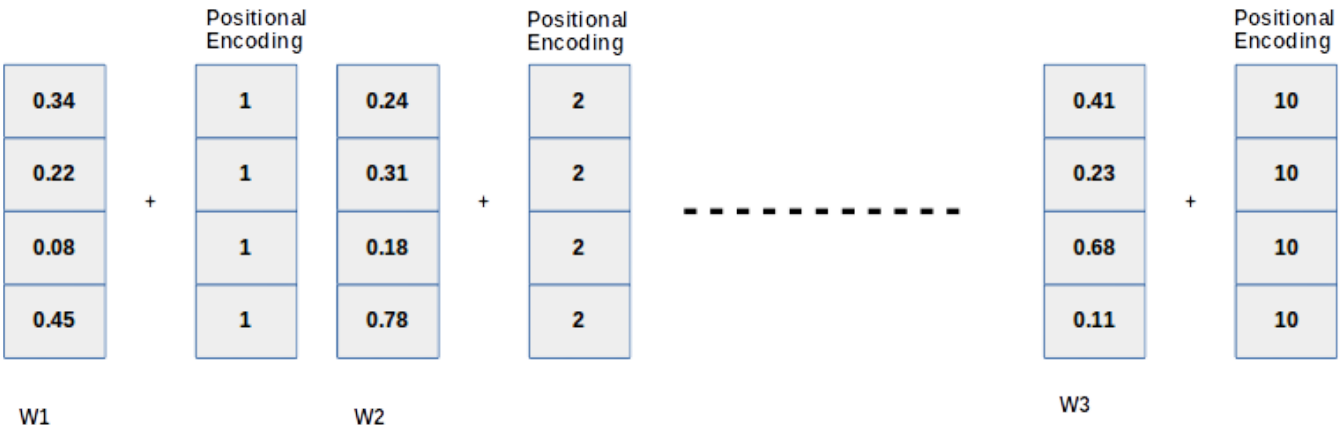
What is Embedding?

A neural network can work only with digits so the very first step is to assign some numerical values to each word. Suppose you have 10000 words dictionary so you can assign a unique index to each word up to 10000. Now all words can be represented by indices. And embedding is a d-dimensional vector for each index. Refer to the figure below just for a basic idea of word embedding each word has a unique index and has an embedding vector.



Positional Embedding in Bert Tokenizer

Let's have a little fun here, just think if we add positional indices to our word embedding. Will this help? Refer to the image below, when we add a positional index to the word embedding(W1, W2..etc) then the final embedding for the rightmost words will always be bigger and it will dominate the original word embedding as in this case if we add 10 to W3, then the significance of W3 will be lost. To overcome this problem if we normalized the indices with the total length (divide all indices by 10) then the same word will have very different embedding for different lengths of sentences.



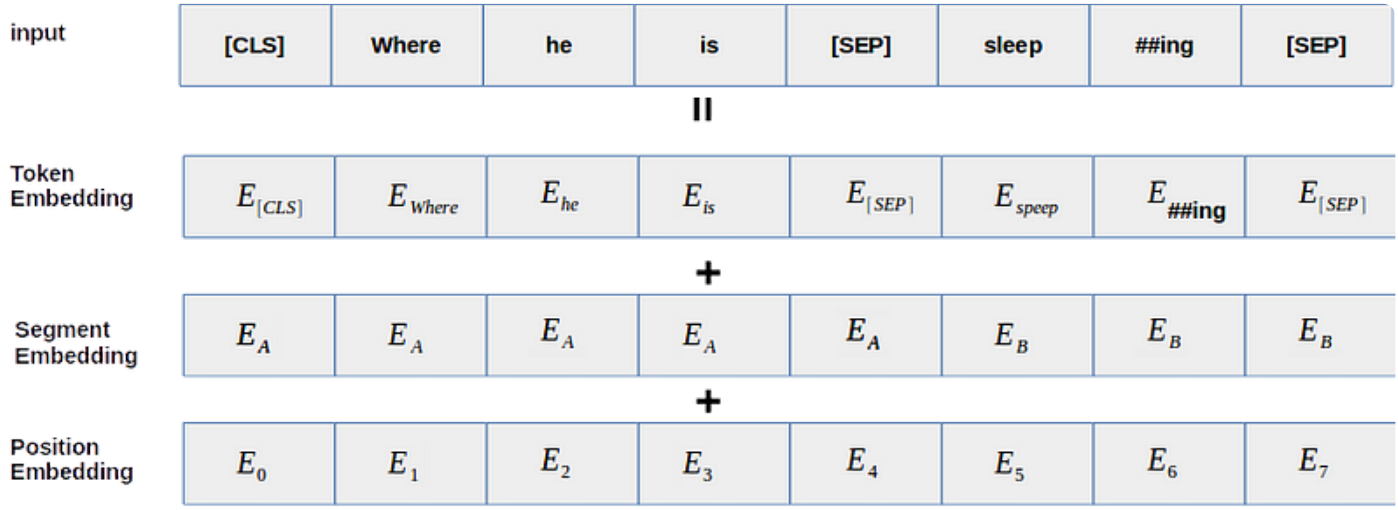
Transformers came up with a beautiful idea for the above problem. They used sinusoidal positional encoding. The formula is written below where pos is positional indices of words in the sentences, d is embedding vector dimension and i is the position of indices in that embedding vector. Using Sin and

$$PE_{(pos,2i)} = \sin(pos \setminus 10000^{2i/d})$$

$$PE_{(pos,2i+1)} = \cos(pos \setminus 10000^{2i/d})$$

Sine cosine formula for positional encoding

BERT Input Embedding



If you have gone through BERT's original paper you must have seen the above figure. If you do not, then do not worry we are here to explore everything. In BERT we do not have to give sinusoidal positional encoding, the model itself learns the positional embedding during the training phase, that's why you will not found the positional embedding in the default library of transformers. BERT came up with the clever idea of using the word-piece tokenizer concept which is nothing but to break some words into sub-words. For example in the above image 'sleeping' word is tokenized into 'sleep' and '##ing'. This idea may help many times to break unknown words into some known words. If I am saying known words I mean the words which are in our vocabulary. We will see this with a real-world example later.

Bert Tokenizer in Transformers Library

https://huggingface.co/bert-base-uncased/tree/main and download vocab.txt and config files from here. Install the transformers library using pip.

```
from transformers import BertTokenizer
tokenizer = BertTokenizer.from_pretrained(//path to tokenizers)
sample = 'where is Himalayas in the world map?'
encoding = tokenizer.encode(sample)
print(encoding)
print(tokenizer.convert_ids_to_tokens(encoding))output 1: [101, 2073, 2003, 26779, 1999, 1996, 2088, 4949,
output 2: ['[CLS]', 'where', 'is', 'himalayas', 'in', 'the', 'world', 'map', '?', '[SEP]']
```

In the above code we simply called the encode function and what we get in output1 is indices of the input tokens from the vocab file and output2 is the reverse, a human-readable token of the input_ids. Apart from the input tokens we also got 2 special tokens *'[CLS]'* and *'[SEP]'*. BERT model is designed in such a way that the sentence has to start with the [CLS] token and end with the [SEP] token. If we are working on question answering or language translation then we have to use [SEP] token in between the two sentences to make separation but thanks to the Hugging-face library the tokenizer library does it for us. Now let's play a bit with this example and change the word *'Himalayas to 'Himalayas and see what happens next.*

```
sample = 'where is Himalayass in the world map?'
encoding = tokenizer.encode(sample)
print(encoding)
print(tokenizer.convert_ids_to_tokens(encoding))
output 1: [101, 2073, 2003, 26779, 2015, 1999, 1996, 2088, 4949, 1029, 102]
output 2: ['[CLS]', 'where', 'is', 'himalayas', '##s', 'in', 'the', 'world', 'map', '?', '[SEP]']
```

Did you notice in the above example, the tokenizer brings the word *'Himalayas' back from the dark. This way it can handle most of the unknown words and improve the model accuracy. Now let's dig a bit more and explore one more function of the tokenizer library and understand the concept with a question-answer example again. I am using question-answering data because it has two input pairs of sequences (question and context) on the other side classification has only one sequence.*

```
q1 = 'Who was Tony Stark?'
c1 = 'Anthony Edward Stark known as Tony Stark is a fictional character in Avengers'
encoding = tokenizer.encode_plus( q1, c1)
for key, value in encoding.items():
    print( '{} : {}'.format( key, value ) )
output:
input_ids : [101, 2040, 2001, 4116, 9762, 1029, 102, 4938, 3487, 9762, 2124, 2004, 4116, 9762, 2003, 1037,
token_type_ids : [0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
attention_mask : [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
```

(question and answer) pair as a single input and we are getting a [SEP] token between the question and answer. Token_type_ids are 0s for the first sentence and 1 for the second sentence. Remember if we are doing a classification task then the token_type_ids will not be useful there because the input sequence is not paired(only zeros essentially not required there). To understand attention_mask we have to process data in batches. In a batch, we may have different lengths of inputs. The model always required input data in rectangular format, if we are passing data in a batch. For example, if we have a batch size=3 the batch will look like the below image.

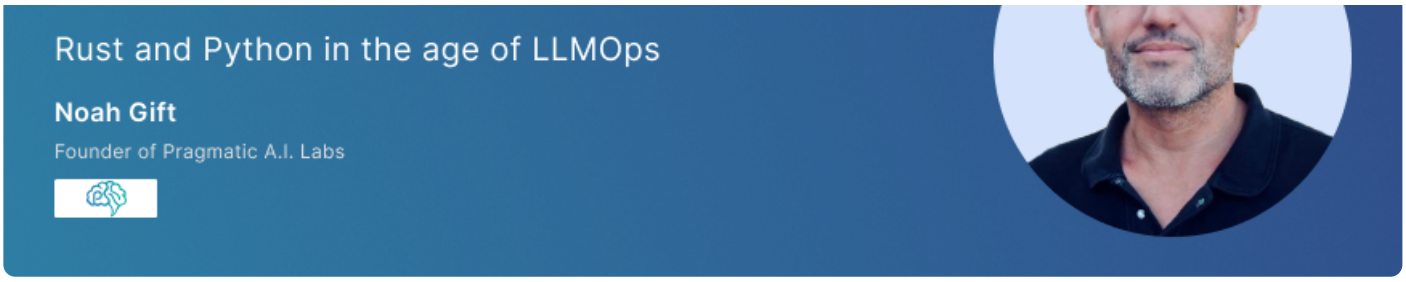
I	like	this	movie
Oh	my	God	
Help	me		

Not a format

I	like	this	movie
Oh	my	God	Pad 0
Help	me	Pad 0	Pad 0

Required format

From the above image, you can visualize that what I was just saying above. We need to make the same length for all the samples in a batch. In that process, some padding value has to be added to the right side of the tokens in shorter sentences and to ensure the model will not look into those padded values attention mask is used with value as zero. Let's have a real data problem for the same question answering data.




```
q1 = 'Who was Tony Stark?'
c1 = 'Anthony Edward Stark known as Tony Stark is a fictional character in Avengers'
q2 = 'Who was Tony in Marvel'
c2 = 'Tony Stark is a fictional character in Marvel Avengers'
encoding = tokenizer([q1,q2], [c1,c2])
for key, value in encoding.items():
    print('1- {}: {}'.format(key, value))
output:
input_ids: [[101, 2040, 2001, 4116, 9762, 1029, 102, 4938, 3487, 9762, 2124, 2004, 4116, 9762, 2003, 1037,
token_type_ids: [[0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1], [0, 0, 0, 0, 0, 0, 0, 1,
attention_mask: [[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1], [1, 1, 1, 1, 1, 1, 1, 1,
```

In the above code, we made two lists the first list contains all the questions and the second list contains all the contexts. This time we received two lists for each dictionary (input_ids, token_type_ids, and attention_mask). If you observed the size of both lists is still different. This happened because we did not use padding as an argument.

```
encoding = tokenizer([q1,q2], [c1,c2],padding=True)
for key, value in encoding.items():
    print('1- {}: {}'.format(key, value))
output:
input_ids: [[101, 2040, 2001, 4116, 9762, 1029, 102, 4938, 3487, 9762, 2124, 2004, 4116, 9762, 2003, 1037,
token_type_ids: [[0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1], [0, 0, 0, 0, 0, 0, 0, 1,
attention_mask: [[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1], [1, 1, 1, 1, 1, 1, 1, 1,
```

After adding the padding argument we got zero paddings for shorter sentences and everything remains the same. Here comes the importance of attention_mask, the model will only focus where the masking is set to true or 1. The same thing can be achieved with another function of the tokenizer library, the only difference is, you have to make a list of pair of a question and a context as below-

```
encoding = tokenizer.batch_encode_plus([[q1,c1], [q2,c2]], padding=True)
for key, value in encoding.items():
    print('{}: {}'.format(key, value))
```

And we will get the output for the above code the same as you get in the previous example. Similarly, different BERTs varients required different input embedding. For example, DistilBERT does not use

```
from transformers import DistilBertTokenizer
tokenizer= DistilBertTokenizer.from_pretrained('//path to tokenizers')encoding = tokenizer.batch_encode_pl
for key, value in encoding.items():
    print('{}: {}'.format(key, value))
Output:
input_ids: [[101, 2040, 2001, 4116, 9762, 1029, 102, 4938, 3487, 9762, 2124, 2004, 4116, 9762, 2003, 1037,
attention_mask: [[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1], [1, 1, 1, 1, 1, 1, 1, 1,
```

Conclusion

There is a specific input type for every BERT variant for example DIstilBERT uses the same special tokens as BERT, but the DIstilBERT model does not use token_type_ids. Thanks to the Hugging-face transformers library, which has mostly all the required tokenizers for almost all popular BERT variants and this saves a lot of time for the developer. BERT model can be applied to 11 different NLP problems and this library will help you to make an input pipeline for all of them. I hope this article made your understanding of the input pipeline much better than before.

Note: All the images used in this article are designed by the author.

The media shown in this article are not owned by Analytics Vidhya and are used at the Author's discretion.

BERT blogathon deep learning NLP

Pritesh Prakash
10 Sep 2021

Senior Research Staff in CRL-BEL

Advanced Data Science Deep Learning Libraries NLP

Responses From Readers

Submit reply

Write, Shine, Succeed

Write, captivate, and earn accolades and rewards for your work

- Reach a Global Audience
- Get Expert Feedback
- Build Your Brand & Audience
- Cash In on Your Knowledge
- Join a Thriving Community
- Level Up Your Data Science Game

Sion Chakrabarti
16

CHIRAG GOYAL
87

Barney Darlington
5

Suvojit Hore
9

Company

About Us

Contact Us

Careers

Discover

Blogs

Expert session

Podcasts

Comprehensive Guides

Learn

Engage