

Preparing Text Data for Transformers: Tokenization, Mapping and Padding



Ganesh Lokare · Follow

11 min read · Feb 10, 2023

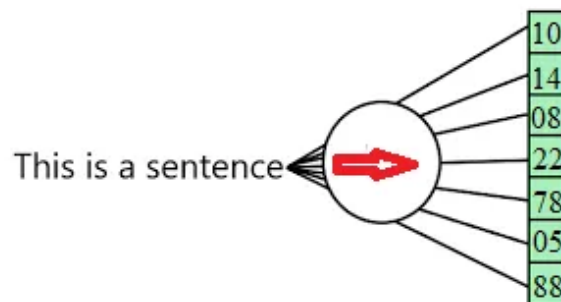


Listen



Share

TEXT PREPROCESSING



In this blog we will be introducing our next section of the series which is how to fine tune a transformer models on a custom dataset.

Why this is different from previous section, previous section was easy because it only involved using models that someone else trained i.e. we only used the model for inference / prediction. The code was very simple because the library we were using provides a nice universal interface for different types of models. So we have the same interface for different kinds of task.

From now, we won't just be making predictions but training our own models on our datasets as well.

What We Will Cover in This Blog

1. Review text preprocessing for transformers

2. Tokenization, token to integer mapping, padding

Text Preprocessing

The steps:

1. Tokenization
2. Map tokens to integers (i.e. "I like Dogs" : [312, 820, 123])
3. Padding / Truncation (to process batches of different length sequences)

1. Tokenization

If you've already studied ML / DL for NLP, you've probably tokenized documents into words.

Tokenization is the process of dividing text into smaller units called tokens, which can be words, phrases, subwords, or characters. In the context of Transformer models, tokenization is a crucial step in preprocessing text data for use in natural language processing tasks.

Tokenization helps the model to identify the underlying structure of the text and process it more efficiently. It enables the model to handle variations in language and handle words with multiple meanings or forms. The choice of tokenization method depends on the task and language being processed and can be a major factor in determining the effectiveness of the model.

Tokenization is an essential step in the text preprocessing pipeline for Transformer models and plays a crucial role in the overall performance of the model.

Types of tokenization:

a) word-level tokenization

Word-level tokenization is a method of dividing text into smaller units called tokens, where each token is a single word.

For example, consider the following sentence:

"The quick brown fox jumps over the lazy dog."

In word-level tokenization, this sentence would be split into the following tokens:

```
"The"  
"quick"  
"brown"  
"fox"  
"jumps"  
"over"  
"the"  
"lazy"  
"dog"
```

Each token is treated as a separate entity and can be processed individually. Word-level tokenization is simple and straightforward, but it can result in difficulties in handling words with multiple meanings or forms. It is also limited by the vocabulary size and cannot handle out-of-vocabulary words.

b) Character-level tokenization

Character-level tokenization is a method of dividing text into smaller units called tokens, where each token is a single character.

For example, consider the following sentence:

```
"I like Dogs."
```

In character-level tokenization, this sentence would be split into the following tokens:

```
"I"  
"l"  
"i"  
"k"  
"e"  
"D"  
"o"  
"g"  
"s"  
"."
```

Each token is treated as a separate entity and can be processed individually.

c) Subword-level tokenization

Subword-level tokenization is a method of dividing text into smaller units called tokens, where each token is a subword unit, typically a sequence of characters. This method of tokenization is used when processing natural language text and aims to capture both the meaning of individual words and the relationships between subword units within words.

For example, consider the following sentence:

"The quick brown fox jumps over the lazy dog."

In subword-level tokenization, this sentence would be split into the following tokens:

"The"
"quic"
"k"
"brown"
"fox"
"jump"
"s"
"over"
"the"
"lazy"
"dog"

In this example, subword-level tokenization has divided words like “quick” and “jump” into subword units, capturing both the meaning of individual words and the relationships between subword units within words.

Subword-level tokenization is a popular method of tokenization in NLP and is used in many state-of-the-art NLP models, including Transformer models. It provides a trade-off between the fine-grained representation of character-level tokenization and the simplicity of word-level tokenization, and can lead to improved performance in many NLP tasks.

2. Map tokens to integers

Mapping tokens to integers in transformers refers to the process of encoding text into numerical representations that can be processed by machine learning models. In NLP, tokens are usually words, subwords, or characters, and mapping them to integers

involves assigning a unique integer value to each token. This integer representation is used to represent the text data in a numerical format that can be used by deep learning models, such as transformers.

The mapping is typically performed by creating a vocabulary of the tokens in the text corpus and assigning each token an integer value based on its frequency of occurrence in the corpus. The most common tokens are assigned lower integer values, while less frequent tokens are assigned higher values. This mapping process allows the model to process the text data efficiently and make predictions based on the underlying patterns in the data.

Here's a simple example of mapping tokens to integers in transformers:

Suppose we have a text corpus consisting of the following sentences:

```
sentence_1 : "The cat sat on the mat"  
sentence_2 : "The dog chased the cat"  
sentence_3 : "The mouse ran away from the cat"
```

Open in app ↗

Sign up

Sign in



Search



```
["The", "cat", "sat", "on", "the", "mat", "dog", "chased", "mouse", "ran", "away"]
```

Next, we can assign integer values to each token in the vocabulary based on their frequency of occurrence. For example, the most frequent token, “the”, could be assigned the integer value 1, the next most frequent token, “cat”, could be assigned the integer value 2, and so on. The final mapping would look like this:

```
{  
  "The": 1,  
  "cat": 2,  
  "sat": 3,  
  "on": 4,  
  "the": 5,  
  "mat": 6,
```

```
"dog": 7,  
"chased": 8,  
"mouse": 9,  
"ran": 10,  
"away": 11,  
"from": 12  
}
```

Finally, we can use this mapping to encode the text data into numerical representations. For example, the first sentence “The cat sat on the mat” would be encoded as:

```
[1, 2, 3, 4, 1, 6]
```

This encoded representation of the text data can then be fed into a transformer model for processing and prediction.

3. Padding / Truncation

Padding and truncation are preprocessing techniques used in transformers to ensure that all input sequences have the same length.

Padding refers to the process of adding extra tokens (usually a special token such as [PAD]) to the end of short sequences so that they all have the same length. This is done so that the model can process all the sequences in a batch simultaneously. The padded tokens do not carry any semantic meaning and are just used to fill up the extra space in the shorter sequences.

Truncation, on the other hand, refers to the process of cutting off the end of longer sequences so that they are all the same length. This is done to ensure that the model is not overwhelmed by very long sequences and to reduce the computational overhead of processing large sequences.

In transformers, padding and truncation are usually performed before feeding the input sequences into the model, and the maximum length for the sequences is set based on the specific task and the available computational resources. The choice of the maximum length is a trade-off between keeping enough information from the original sequence and reducing computational overhead.

Here's a simple example of padding and truncation in transformers:

Suppose we have a batch of sequences with varying lengths:

```
Sequence 1: "The cat sat on the mat"  
Sequence 2: "The dog chased the cat"  
Sequence 3: "The mouse ran away from the cat and the dog"
```

And let's say that we want to set the maximum length for all the sequences to be 10 tokens.

For padding, we would add the special token [PAD] to the end of each sequence until it reaches the maximum length of 10 tokens. The padded sequences would look like this:

```
Sequence 1: "The cat sat on the mat [PAD] [PAD] [PAD] [PAD]"  
Sequence 2: "The dog chased the cat [PAD] [PAD] [PAD] [PAD] [PAD]"  
Sequence 3: "The mouse ran away from the cat and the dog"
```

```
unpadded = [  
  [1, 2, 3],  
  [4, 5],  
  [6, 7, 8, 9, 10],  
]
```



```
padded = [  
  [1, 2, 3, 0, 0],  
  [4, 5, 0, 0, 0],  
  [6, 7, 8, 9, 10],  
]
```

Padding

For truncation, we would cut off the end of each sequence so that it fits within the maximum length of 5 tokens. The truncated sequences would look like this:

```
Sequence 1: "The cat sat on the"  
Sequence 2: "The dog chased the cat"  
Sequence 3: "The mouse ran away from"
```

```
toolong = [  
    [1, 2, 3, ..., lim],  
    [4, 5],  
    [6, 7, ..., lim, ..., 8],  
]
```



```
justright = [  
    [1, 2, 3, ..., lim],  
    [4, 5, 0, ..., lim],  
    [6, 7, ..., lim],  
]
```

Truncation

Note that the choice of padding or truncation, or a combination of both, will depend on the specific task and the desired trade-off between preserving information from the original sequence and reducing computational overhead.

Enough theory!!! Let's code...

```
# install transformers  
!pip install transformers
```

The command `pip install transformers` is used to install the `transformers` package, which provides access to state-of-the-art Transformer-based models for NLP tasks.

Once the `transformers` package is installed, you can import and use the Transformer-based models in your own projects.

Import `AutoTokenizer` class from `transformers` and build tokenizer object. There are different types of tokenizers but `AutoTokenizer` is a generic tokenizer that can handle various types of pre-trained models, including BERT, GPT-2, RoBERTa, XLNet, etc.

```
# Import AutoTokenizer and create tokenizer object  
from transformers import AutoTokenizer  
checkpoint = 'bert-base-cased'  
tokenizer = AutoTokenizer.from_pretrained(checkpoint)
```

The code is using the `AutoTokenizer` class from the `transformers` library to load a pre-trained tokenizer for the BERT model with the "base" architecture and the "cased" version. The pre-trained tokenizer will be used to convert input sequences of text into numerical representations (tokens) that can be fed into the model. The `checkpoint` variable specifies the name of the pre-trained tokenizer to use, and the

`from_pretrained` method is used to load the tokenizer from the `transformers` library's pre-trained models.

Print tokenizer object

```
tokenizer
```

```
tokenizer
```

```
BertTokenizerFast(name_or_path='bert-base-cased', vocab_size=28996, model_max_length=512,
is_fast=True, padding_side='right', truncation_side='right', special_tokens={'unk_token': '[UNK]',
'sep_token': '[SEP]', 'pad_token': '[PAD]', 'cls_token': '[CLS]', 'mask_token': '[MASK]'})
```

tokenizer object

The `BertTokenizerFast` class is a tokenizer that is specifically designed for BERT-based models and provides tokenization functionalities for text data. The `vocab_size` attribute of the `BertTokenizerFast` instance is set to 28996, which means the vocabulary of the tokenizer consists of 28996 unique tokens.

The `model_max_length` attribute is set to 512, which indicates that the maximum length of the input sequence that the BERT model can handle is 512 tokens. This means that if the input sequence is longer than 512 tokens, it will be truncated to fit the maximum length.

The `is_fast` attribute is set to `True`, which means that the tokenization process is optimized for speed and is faster than other tokenization methods.

The `padding_side` attribute is set to "right", which means that the tokenizer will add padding tokens to the right side of the input sequence if it is shorter than the maximum length. The `truncation_side` attribute is set to "right", which means that the tokenizer will truncate the input sequence from the right side if it is longer than the maximum length.

The `special_tokens` attribute is a dictionary that contains the special tokens used in the BERT model, such as the unknown token ([UNK]), the separator token ([SEP]), the padding token ([PAD]), the class token ([CLS]), and the mask token ([MASK]). These special tokens play a crucial role in the BERT model's input and output encoding.

Test our tokenizer with simple string

```
tokenizer('hello world')
```

Output: {

input_ids: [101, 19082, 1362, 102],

token_type_ids: [0, 0, 0, 0],

attention_mask: [1, 1, 1, 1] }

The code is applying the `tokenizer` object, which is an instance of the `AutoTokenizer` class, to the input text `'hello world'`. The `tokenizer` object will convert the input text into a numerical representation that can be fed into the pre-trained model.

The output of the code is a dictionary with three keys:

1. `input_ids` : This is a list of integers that represent the numerical representation of the input text. Each integer corresponds to a token in the vocabulary of the pre-trained model.
2. `token_type_ids` : This is a list of integers that indicate the type of each token in the input sequence. For example, in a sequence classification task, the first token of the input sequence could be marked as type 0, and the second token as type 1.
3. `attention_mask` : This is a list of 1's and 0's that indicate which tokens should be attended to by the pre-trained model and which should be ignored. A 1 indicates that the token should be attended to, while a 0 indicates that the token should be ignored.

input_ids: [101, 19082, 1362, 102] Note that there are 4 input ids because the input has been converted to: [CLS] hello world [SEP]

As we can see tokenizer's output is a dictionary of lists, So we can not feed lists directly to the model. If we pass lists directly model will raise an error. It required tensors.

Proper model input

```
tokenizer('hello world', return_tensors = 'pt')
```

Output: {

‘input_ids’: tensor([[101, 19082, 1362, 102]]),

‘token_type_ids’: tensor([[0, 0, 0, 0]]),

‘attention_mask’: tensor([[1, 1, 1, 1]])}

The `return_tensors` argument is set to `'pt'`, which indicates that the output should be in PyTorch tensor format.

The available options for `return_tensors` are:

1. `'pt'` : Returns PyTorch tensors.
2. `'tf'` : Returns TensorFlow tensors.
3. `None` : Returns Python lists. This is the default option if `return_tensors` is not specified.

You can choose the appropriate option based on the deep learning framework you are using for your NLP model. If you are using PyTorch, you can set `return_tensors` to `'pt'`. If you are using TensorFlow, you can set `return_tensors` to `'tf'`. If you are not using a deep learning framework, you can use the default option, which returns Python lists.

Multiple Inputs

```
data = ["I like cat",  
        "Do you like cat too?"]  
  
tokenizer(data, padding = True, truncation=True, return_tensors='pt')
```

Output: {

‘input_ids’: tensor([[101, 146, 1176, 5855, 102, 0, 0, 0],

```
[ 101, 2091, 1128, 1176, 5855, 1315, 136, 102]])),
```

```
'token_type_ids': tensor([[0, 0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0, 0, 0]]),
```

```
'attention_mask': tensor([[1, 1, 1, 1, 1, 0, 0, 0], [1, 1, 1, 1, 1, 1, 1, 1]])}
```

As we can see, we have inputs of different lengths, so it is necessary to apply padding or truncation. If we haven't applied padding and truncation model will not accept input of different lengths.

By setting `padding` to `True`, the tokenized inputs will be padded with a special padding token (usually represented by 0 in the vocabulary) so that all inputs have the same length. This is often necessary in NLP tasks, as many models expect a fixed-length input.

By setting `truncation` to `True`, if any of the input texts exceed the maximum length of the pre-trained model, they will be truncated to that maximum length. This is necessary to ensure that the inputs can be processed by the pre-trained model.

In Transformers, we only require the above text preprocessing steps, which is called Tokenization. After that, we can directly feed this preprocessed data to the model.

That's it for the text preprocessing.

Next, I will demonstrate how to train and fine-tune transformer models on custom datasets.

If you enjoyed this post, please follow me and be sure to check out some of my other blog posts for more insights and information. I'm constantly exploring new topics and writing about my findings, so there's always something new and interesting to discover.

Ganesh Lokare - Medium

Read writing from Ganesh Lokare on Medium. Data Scientist
Github:<https://github.com/GaneshLokare>. Every day, Ganesh...

medium.com

Transformers

Hugging Face

NLP

Tokenization

Deep Learning



Follow

Written by Ganesh Lokare

119 Followers

Data Scientist Github:<https://github.com/GaneshLokare>

More from Ganesh Lokare

Fine-Tuning Transformers



Ganesh Lokare

Fine-Tuning Transformers with custom dataset: Classification task

While pre-trained transformer models have many benefits, there are also several drawbacks to using them on a custom dataset compared to...

14 min read · Feb 11, 2023



156



2



Ganesh Lokare

Text Summarization with Hugging Face Transformers: A Beginner's Guide

What We Will Cover in This Blog

7 min read · Feb 8, 2023



Transformers

 Ganesh Lokare

Effortless Sentiment Analysis with Hugging Face Transformers: A Beginner's Guide


Sentiment Analysis

9 min read · Feb 4, 2023



THIS AI CAN ANSWER QUESTIONS



 Ganesh Lokare

Question Answering with Hugging Face Transformers: A Beginner's Guide

What We Will Cover in This Blog

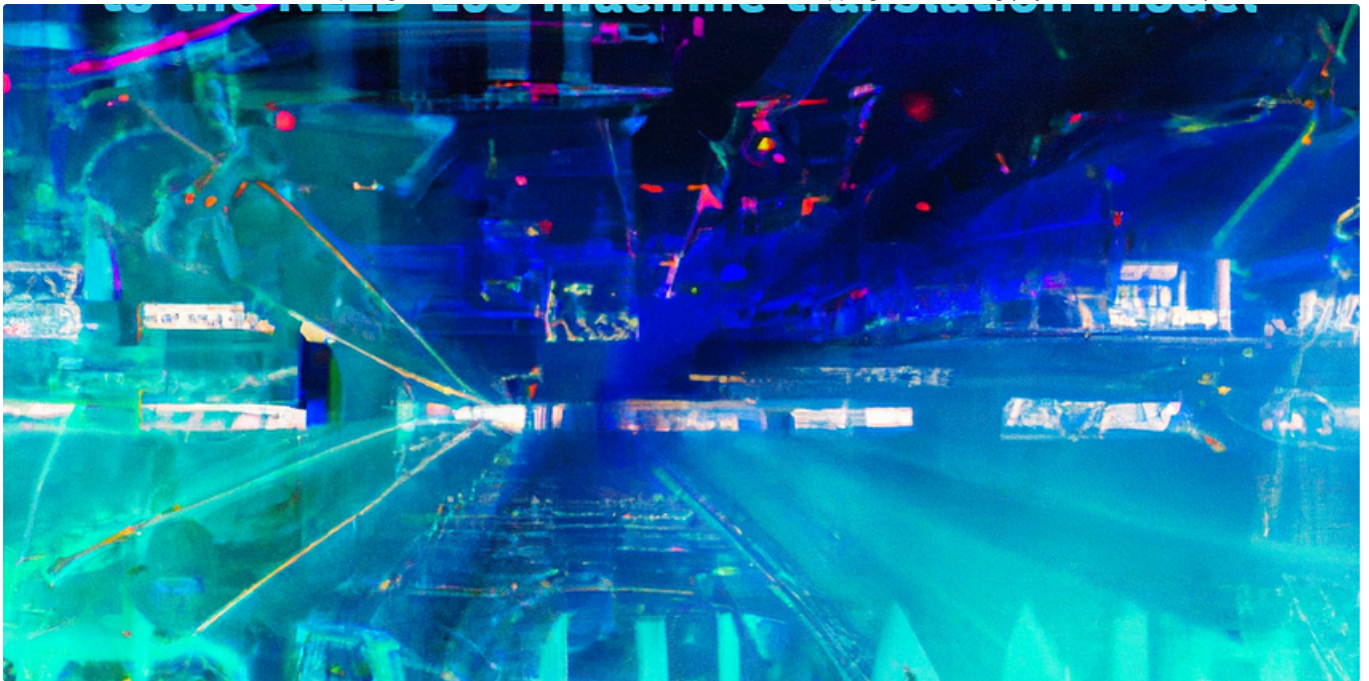
9 min read · Feb 9, 2023

 66  1



See all from Ganesh Lokare

Recommended from Medium



David Dale

How to fine-tune a NLLB-200 model for translating a new language

NLLB is a translation model that supports 200 languages. I teach it one more language, Tyvan, and explain the code behind this update.

25 min read · Oct 17, 2023



202



4



Ahmed Mellit

Text Similarity Implementation using BERT Embedding in Python

Unlocking the Power of BERT and Cosine Similarity: Enhancing Text Similarity Analysis

5 min read · Nov 9, 2023



58



Lists



Natural Language Processing

1244 stories · 724 saves



data science and AI

40 stories · 89 saves



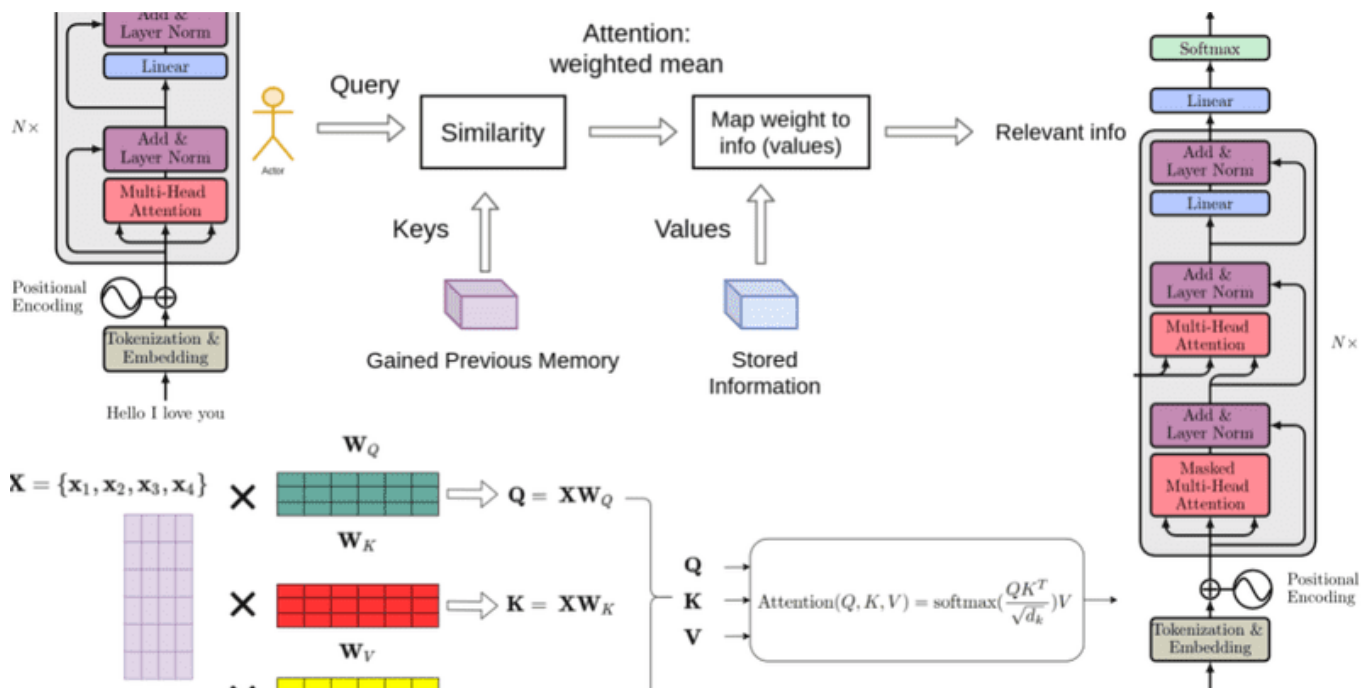
The New Chatbots: ChatGPT, Bard, and Beyond

12 stories · 320 saves



Practical Guides to Machine Learning

10 stories · 1128 saves



Amanatullah

Transformer Architecture explained

Transformers are a new development in machine learning that have been making a lot of noise lately. They are incredibly good at keeping...

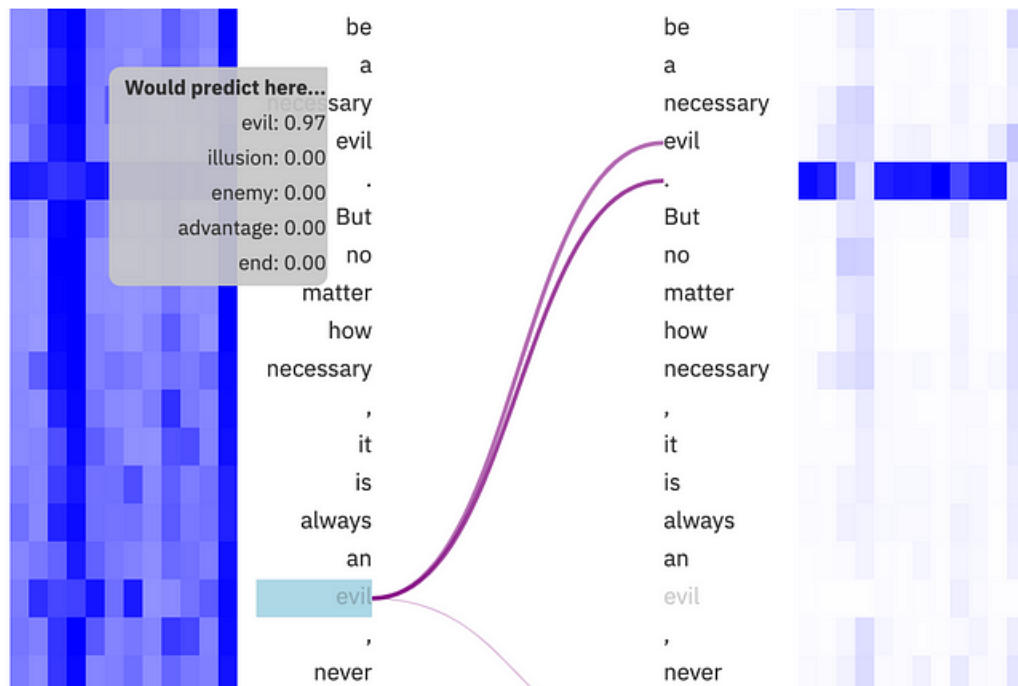
10 min read · Sep 1, 2023



509



5



Ransaka Ravihara in Towards Data Science

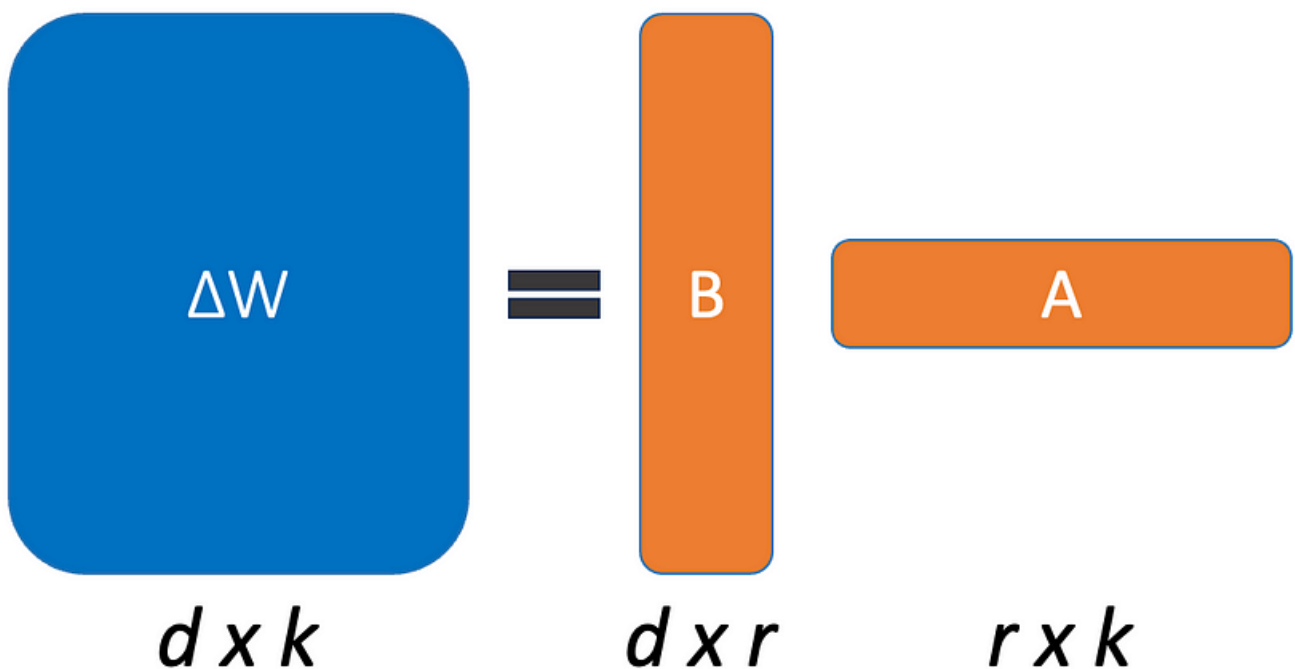
How to Train BERT for Masked Language Modeling Tasks

Hands-on guide to building language model for MLM tasks from scratch using Python and Transformers library

7 min read · Oct 18, 2023



103





Manyi

More about LoraConfig from PEFT

Parameter-Efficient Fine-Tuning (PEFT) enable efficient adaptation of pre-trained models to downstream applications without fine-tuning all...

3 min read · Sep 5, 2023



498



Anirban Sen

Finetuning LLMs using LoRA

Before getting to the meat of the blog i.e. Finetuning an LLM, it will be good to have a brief summary of Why do we even need it ?

8 min read · Sep 30, 2023



322



1



See more recommendations