Search Medium Write

Member-only story

The Ultimate Guide to Training BERT from Scratch: Prepare the Dataset

Data Preparation: Dive Deeper, Optimize your Process, and Discover How to Attack the Most Crucial Step

Dimitris Poulopoulos · Follow

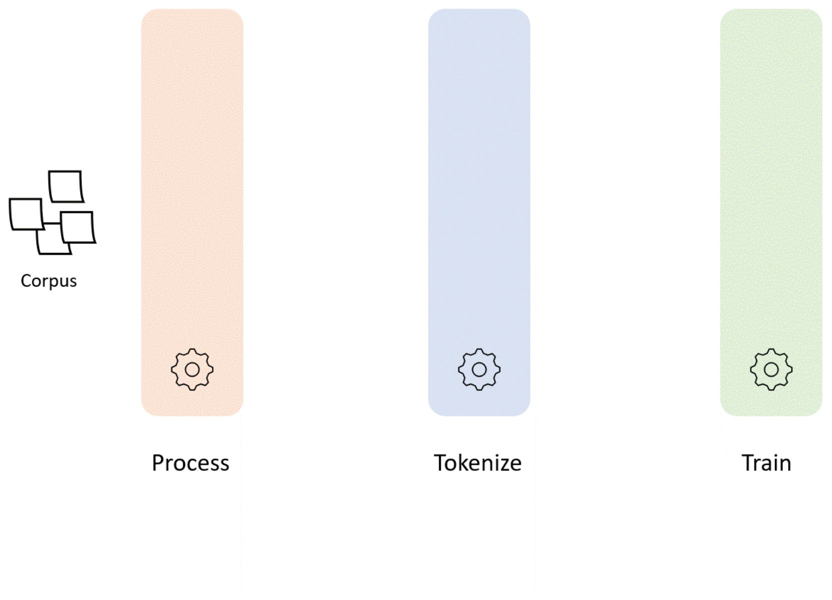
Published in Towards Data Science · 13 min read · Sep 14

101 1

Photo by Patrick Tomasso on Unsplash

Imagine investing a full day fine-tuning BERT, only to hit a performance bottleneck that leaves you scratching your head. You dig into your code and discover the culprit: you just didn't do a good job preparing your features and labels. Just like that, ten hours of precious GPU time evaporates into thin air.

Let's face it, *setting up your dataset isn't just another step — it's the engineering cornerstone of your entire training pipeline*. Some even argue that once your dataset is in good shape, the rest is mostly boilerplate: feed your model, calculate the loss, perform backpropagation, and update the model weights.



The training pipeline — Image by Author

In this story, **we'll get into the process of preparing your data for BERT, setting the stage for the ultimate goal: training a BERT model from scratch.**

Welcome to the third installment of our comprehensive BERT series! In the first chapter, we introduced BERT — breaking down its objectives and demonstrating how to fine-tune it for a practical question-answering system:

The Ultimate Guide to Training BERT from Scratch: Introduction

Demystifying BERT: The definition and various applications of the model that changed the NLP landscape.

towardsdatascience.com

Then, in the second chapter, we dived deep into the world of tokenizers, exploring their mechanics and even creating a custom tokenizer for the Greek language:

The Ultimate Guide to Training BERT from Scratch: The Tokenizer
From Text to Tokens: Your Step-by-Step Guide to BERT Tokenization
towardsdatascience.com

Now, we're tackling one of the most pivotal stages of building a high-performing BERT model: *dataset preparation*. This guide will be a technical one, providing Python snippets and links to the GitHub repositories of popular open-source projects. Okay, we're losing the light; let's start!

> *Learning Rate is a newsletter for those who are curious about the world of ML and MLOps. If you want to learn more about topics like this subscribe here. You'll hear from me on the last Sunday of every month with updates and thoughts on the latest MLOps news and articles!*

Download & Explore

First things first, we should choose a dataset. For the purpose of this tutorial, we'll be working with the `wikitext` dataset, specifically the `wikitext-2-raw-v1` subset.

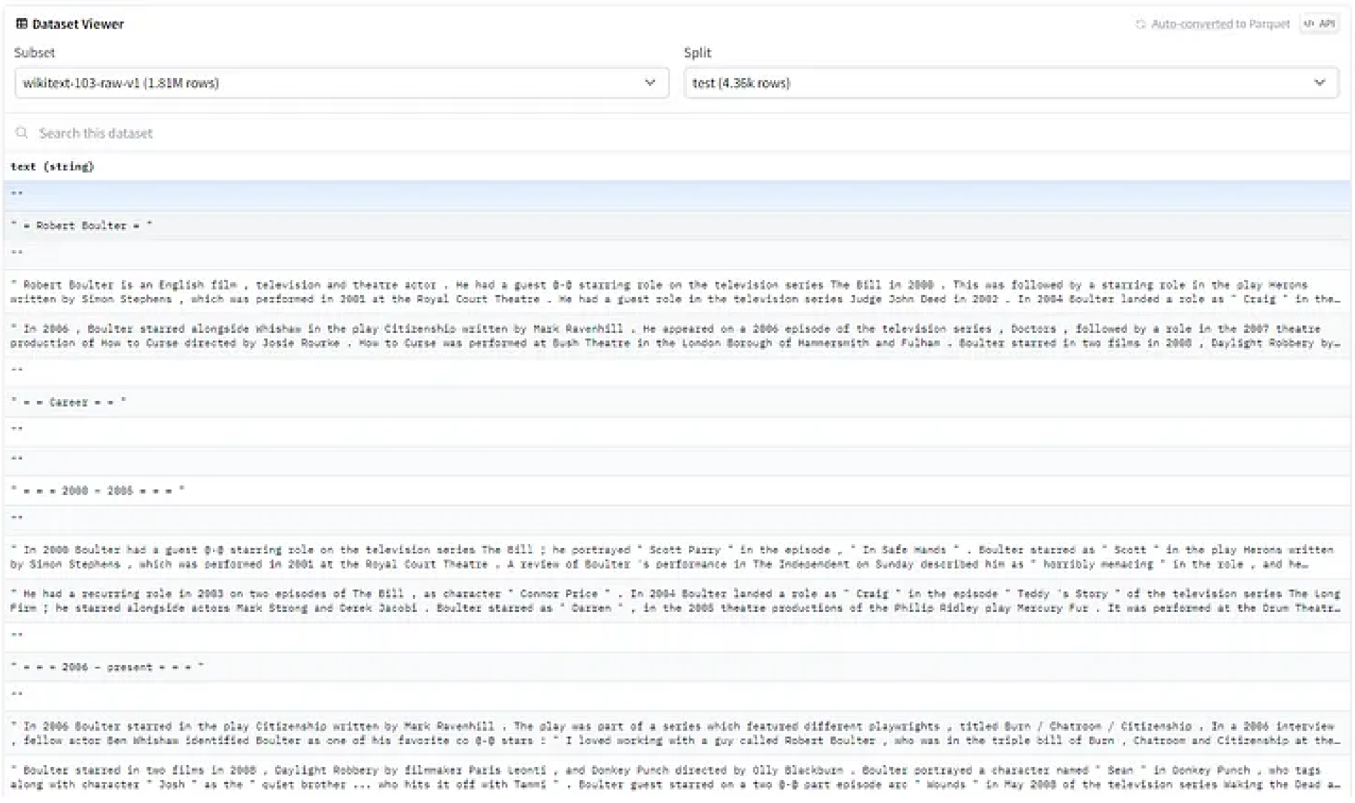
According to its documentation, this dataset gathers over 100 million tokens extracted from Wikipedia's verified articles. It's the ideal playground for our BERT experiments, and thanks to Hugging Face, accessing it is a breeze:

```python
from datasets import load_dataset

datasets = load_dataset('wikitext', 'wikitext-2-raw-v1')
```

The `datasets` variable contains three splits: train, validation, and test. Let's focus on the train split first.

The train split contains 36,718 rows of variable-length text. You'll find entries ranging from empty strings to full-blown paragraphs. Here's a peek using the Hugging Face dataset viewer:



Subset: wikitext-103-raw-v1 (1.81M rows)
Split: test (4.36k rows)

```
text (string)
""
" = Robert Boulter = "
""
" Robert Boulter is an English film , television and theatre actor . He had a guest @-@ starring role on the television series The Bill in 2000 . This was followed by a starring role in the play Herons
written by Simon Stephens , which was performed in 2001 at the Royal Court Theatre . He had a guest role in the television series Judge John Deed in 2002 . In 2004 Boulter landed a role as " Craig " in the…
" In 2006 , Boulter starred alongside Whishaw in the play Citizenship written by Mark Ravenhill . He appeared on a 2006 episode of the television series , Doctors , followed by a role in the 2007 theatre
production of How to Curse directed by Josie Rourke . How to Curse was performed at Bush Theatre in the London Borough of Hammersmith and Fulham . Boulter starred in two films in 2008 , Daylight Robbery by…
""
" = = Career = = "
""
""
" = = = 2000 – 2005 = = = "
""
" In 2000 Boulter had a guest @-@ starring role on the television series The Bill ; he portrayed " Scott Parry " in the episode , " In Safe Hands " . Boulter starred as " Scott " in the play Herons written
by Simon Stephens , which was performed in 2001 at the Royal Court Theatre . A review of Boulter 's performance in The Independent on Sunday described him as " horribly menacing " in the role , and he…
" He had a recurring role in 2003 on two episodes of The Bill , as character " Connor Price " . In 2004 Boulter landed a role as " Craig " in the episode " Teddy 's Story " of the television series The Long
Firm ; he starred alongside actors Mark Strong and Derek Jacobi . Boulter starred as " Carlen " , in the 2005 theatre productions of the Philip Ridley play Mercury Fur . It was performed at the Drum Theatr…
""
" = = = 2006 – present = = = "
""
" In 2006 Boulter starred in the play Citizenship written by Mark Ravenhill . The play was part of a series which featured different playwrights , titled Burn / Chatroom / Citizenship . In a 2006 interview
, fellow actor Ben Whishaw identified Boulter as one of his favorite co @-@ stars : " I loved working with a guy called Robert Boulter , who was in the triple bill of Burn , Chatroom and Citizenship at the…
" Boulter starred in two films in 2008 , Daylight Robbery by filmmaker Paris Leonti , and Donkey Punch directed by Olly Blackburn . Boulter portrayed a character named " Sean " in Donkey Punch , who tags
along with character " Josh " as the " quiet brother ... who hits it off with Tammi " . Boulter guest starred on a two @-@ part episode arc " Wounds " in May 2008 of the television series Waking the Dead a…
```

Wikitext dataset — Image by Author

Note to self: Consistency in token lengths is non-negotiable. This is crucial as our model knows how to handle sequences of specific length during training. We should keep that in mind during the data processing phase.

Now, what about the dataset's features?

```
datasets["train"]
```

The output reveals a rather minimalistic structure:

```
Dataset({
    features: ['text'],
    num_rows: 36718
})
```

That's right, it's just a `text` feature. So how do we shape this raw data into a robust training set for BERT? Get ready; we're about to get our hands dirty

Just like Halloween

To create our dataset preparation blueprint, let's take a moment to revisit BERT's objectives. BERT training is a two-phase process, but today, we're focusing only on the first phase: pre-training. The goal here is to teach the model what language is and how context changes the meaning of words.

The pre-training phase has two key tasks: i) Masked Language Modeling (MLM) and ii) Next Sentence Prediction (NSP). During MLM, we purposely mask certain tokens in a sequence and train BERT to accurately predict them. With NSP, we present BERT with two adjacent sequences, asking it to predict whether the second naturally follows the first.

So, our goal is to create a dataset that will support these tasks and teach BERT the concepts we want. Let’s begin!

The foundations

Okay, first things first: we need to do two things before moving on to masking a sequence: We must tokenize the sentences and then create token sequences of equal length. Do you remember the mental note we made earlier?

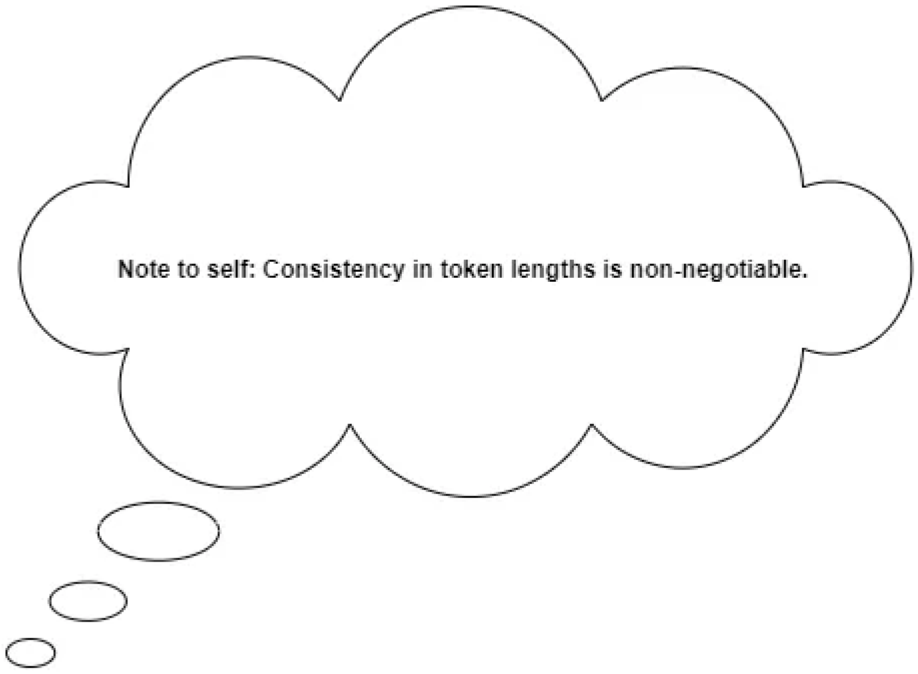

Note to self: Consistency in token lengths is non-negotiable.

Image by Author

Let’s first load the pre-trained BERT tokenizer:

```python
from transformers import AutoTokenizer

model_checkpoint = "bert-base-uncased"
tokenizer = AutoTokenizer.from_pretrained(model_checkpoint, use_fast=True)
```

Next, tokenizing a text sentence is really simple: We come up with a function that accepts an example and passes it through the tokenizer. Then, we map this function across the dataset. This is a common practice when working with Hugging Face datasets, and we'll see this pattern again and again.

```
def tokenize_function(examples):
    return tokenizer(examples["text"], add_special_tokens=False)

tokenized_datasets = datasets.map(
    tokenize_function, batched=True, num_proc=4, remove_columns=["text"])
```

Note the `num_proc` and `batched` arguments. These are essential to make the process run faster. They will make a huge difference as your datasets are getting bigger. For more information about the `map` function, check out the `datasets` library docs. To understand more about these arguments specifically, take a closer look at the multiprocessing and batch-processing sections.

Also, be mindful that we asked the tokenizer to omit adding any special tokens (`add_special_tokens=False`). You'll see later why.

The next step is to create sequences of equal length. BERT usually accepts sequences of `512` tokens. This is mostly defined by the shape of the positional encodings matrix in the model's architecture.

Thus, in this step, we will create sequences of `255` tokens. Why? Because later, we want to concatenate two sentences to create a dataset that supports the NSP tasks. This process will create sequences of `510` tokens (`255 + 255`). However, we should also add two special tokens: `[CLS]` to indicate the start of a sequence and `[SEP]` to mark where the first sentence ends and the second one begins.

As always, let's create a helper function that does that:

```
def group_texts(examples, block_size=255):
    # Concatenate all texts.
    concatenated_examples = {k: sum(examples[k], []) for k in examples.keys()}
    total_length = len(concatenated_examples[list(examples.keys())[0]])
    # We drop the small remainder. We can use padding later.
    total_length = (total_length // block_size) * block_size
    # Split by chunks of `block_size`.
    result = {
        k: [t[i : i + block_size] for i in range(0, total_length, block_size)]
        for k, t in concatenated_examples.items()
    }
    return result
```

This function performs two transformations: First, it brings all the sequences together and then splits them again into small blocks of `255` tokens. Let's map it over our tokenized dataset:

```
lm_datasets = tokenized_datasets.map(
    group_texts, batched=True, batch_size=1000, num_proc=4)
```

That concludes the preliminary transformations. Next, let's concatenate two sequences together to create the effect we want for NSP.

Nest Sentence Prediction

As a reminder, the NSP task tries to predict if two given sentences are adjacent in the original text or not. Thus, we need a dataset where each entry should comprise a pair of sentences along with a label that indicates if they are related.

To create such a dataset, we will, once again, define a helper function. Admittedly, this function will be a bit more intricate than what we've worked with before, but don't let that intimidate you. The most effective way to tackle complexity is often to dive in head first.

```
def create_nsp_entry(example, idx, dataset, total_examples):
    """
    Create a Next Sentence Prediction entry using the given example and its inde
    """
    import random

    first_sentence = example['input_ids']
    attention_mask = [1] * 512
    next_sentence_label = 0

    # Decide the second sequence based on the index
    if idx % 4 < 2:  # Use subsequent sequences half of the time
        next_idx = idx + 1
        try:
            next_sentence = dataset[next_idx]['input_ids']
        except IndexError:
            # If the index is out of bounds
            # (e.g., the example is the last in the dataset),
            # wrap around to the start.
            # In this case, set the label to 1.
            next_idx = next_idx % total_examples
            next_sentence = dataset[next_idx]['input_ids']
            next_sentence_label = 1  # Indicate that the sentences are not conse
        # Set attention mask accordingly.
        attention_mask = [1] + example['attention_mask'] + [1] + dataset[next_id
    else:
        # Get a random sentence to create a negative example.
        rand_idx = random.randint(0, total_examples - 1)
        while rand_idx == idx:  # Ensure we don't pick the same example.
            rand_idx = random.randint(0, total_examples - 1)
        next_sentence = dataset[rand_idx]['input_ids']
```

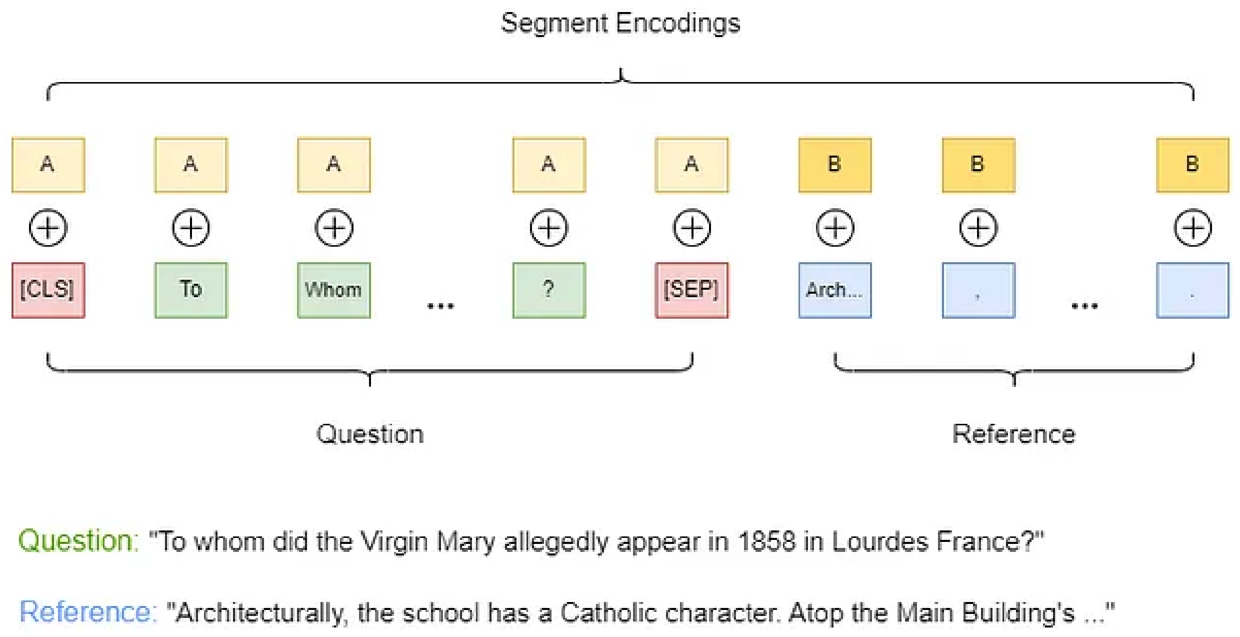
```
            next_sentence_label = 1
            attention_mask = [1] + example['attention_mask'] + [1] + dataset[rand_id

        # Create combined input IDs
        combined_input_ids = [tokenizer.cls_token_id] + first_sentence + [tokenizer.

        # Create token type IDs - the first sentences + the [SEP] are set to 0
        token_type_ids = [0] * (257) + [1] * (255)

        return {
            'input_ids': combined_input_ids,
            'token_type_ids': token_type_ids,
            'attention_mask': attention_mask,
            'next_sentence_label': next_sentence_label
        }
```

Well, that seems a lot! So, let's visualize what this function does to make the code more digestible. This function aims to create something similar to the QnA scheme we saw in the first introductory blog post:


Segment Encodings
A
A
A
A
A
B
B
B
[CLS]
To
Whom
...
?
[SEP]
Arch...
,
...
.
Question
Reference
Question: "To whom did the Virgin Mary allegedly appear in 1858 in Lourdes France?"
Reference: "Architecturally, the school has a Catholic character. Atop the Main Building's ..."

BERT QnA Dataset Preparation — Image by Author

The `if/else` block works as follows:

- Half of the time, create sequences of subsequent sentences, except if the sequence at hand is the last one, in which case, wrap around and create a sequence of two random sentences.
- The other half of the time, create a sequence of two random sentences.
- Set the next sentence label accordingly: `0` for subsequent sequences, `1` if they have no connection.
- Combine the two sentences using the `[CLS]` and `[SEP]` special tokens.
- Create the `token_type_ids` list to mark the two segments.

Now, how did I come up with the names `token_type_ids` and `next_sentence_label`? Are they random names, and you can use whatever variable name you want? No! If you look at the model's forward method signature, you'll find exactly these names:

```
def forward(
        self,
        input_ids: Optional[torch.Tensor] = None,
        attention_mask: Optional[torch.Tensor] = None,
        token_type_ids: Optional[torch.Tensor] = None,  # Different segments
        position_ids: Optional[torch.Tensor] = None,
        head_mask: Optional[torch.Tensor] = None,
        inputs_embeds: Optional[torch.Tensor] = None,
        labels: Optional[torch.Tensor] = None,
        next_sentence_label: Optional[torch.Tensor] = None,  # labels for NSP
        output_attentions: Optional[bool] = None,
        output_hidden_states: Optional[bool] = None,
        return_dict: Optional[bool] = None,
    ) -> Union[Tuple[torch.Tensor], BertForPreTrainingOutput]:
   ...
```

Pay close attention to this fact; you need to use exactly these names. If you don't, the library will remove those features from your dataset before passing it to the model. Why? Because this is what it is supposed to do by default.

Finally, as always, let's map this method. But first, let's separate the different dataset splits so we have them ready later for training and map the function to each one separately:

```
train_dataset = lm_datasets['train']
validation_dataset = lm_datasets['validation']
test_dataset = lm_datasets['test']

nsp_train_dataset = train_dataset.map(
    lambda example, idx: create_nsp_entry(example, idx, train_dataset, total_exa
    with_indices=True)

nsp_validation_dataset = validation_dataset.map(
    lambda example, idx: create_nsp_entry(example, idx, validation_dataset, tota
    with_indices=True)

nsp_test_dataset = test_dataset.map(
    lambda example, idx: create_nsp_entry(example, idx, test_dataset, total_exam
    with_indices=True)
```

There is a subtle difference in how we map the function now: We are creating a lambda function that uses the one we defined, and we set `with_indices=True` so that our helper function has a handle on the index of each example that it operates on.

Great! We have prepared our dataset to handle the NSP task. What about MLM? In the first blog, we've seen that we train BERT on the two tasks simultaneously. So, let's tackle the MLM transformation next.

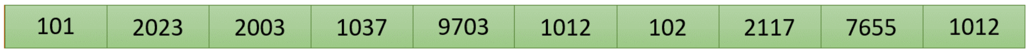
Masked Language Modeling

Luckily, the `transformers` library offers a streamlined solution for the Masked Language Modeling (MLM) task. Simply instantiate the `DataCollatorForLanguageModeling` class with the appropriate parameters, and voila — most of the work is done for you.

But wait, you didn't come here for the easy way out, did you? This guide is all about diving deep, so we won't tolerate any black boxes. We're going to pull back the curtain and explore how exactly to create a custom dataset for MLM.

Here's how we'll tackle it: First, we'll paint a mental picture by visualizing the core concepts. Think of it as a virtual storyboard that brings everything into focus. Next, we'll dissect the code, breaking it down step by step to get a grasp of each component.

Our first order of business? Crafting labels for our input tokens. this is an easy one; all we need to do is clone the input array.

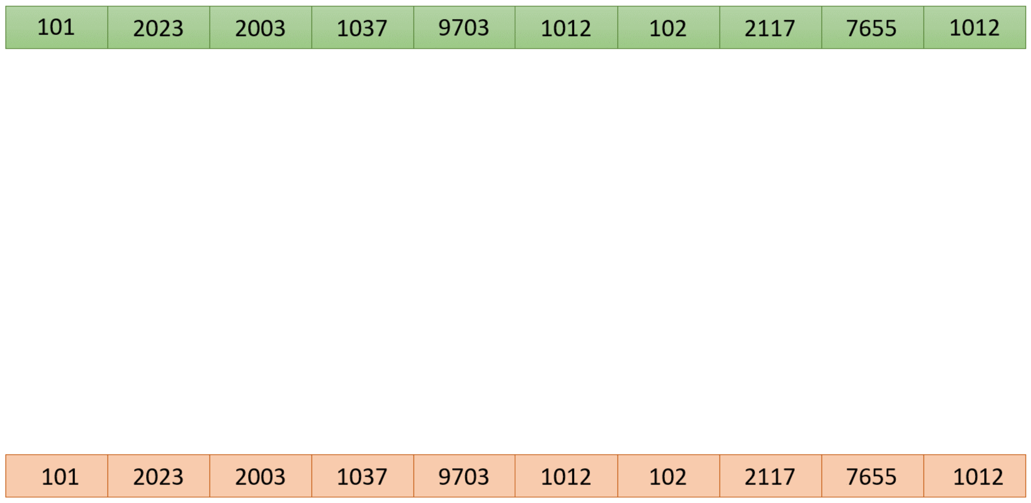


101	2023	2003	1037	9703	1012	102	2117	7655	1012

Labels generation — Image by Author

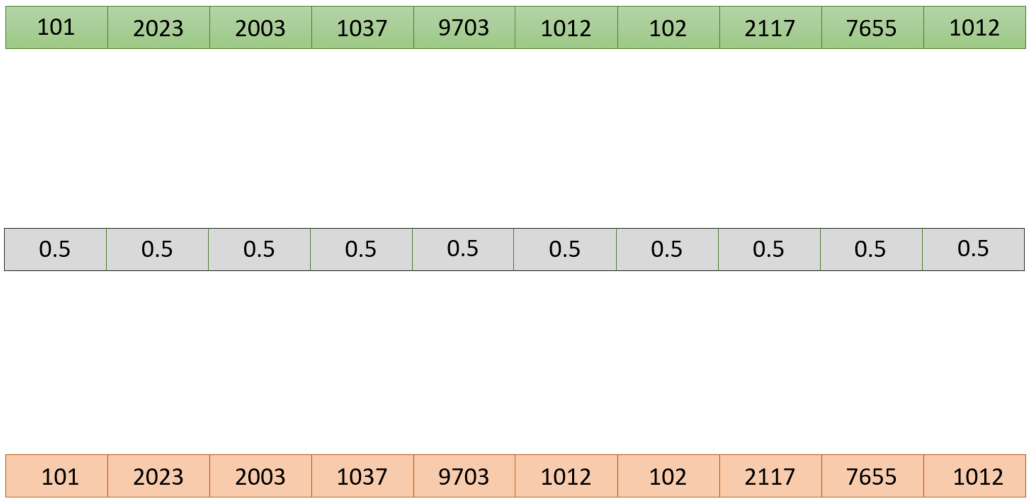
Next up, we forge the “mask” array, which mirrors the shape of our inputs and labels. Each entry in this mask array represents a probability value, dictating the likelihood that a toke at the corresponding position will be masked.

For the sake of demonstration, we’ll use a masking probability of `0.5` , even though in real-world applications, it’s usually set closer to `0.15` :



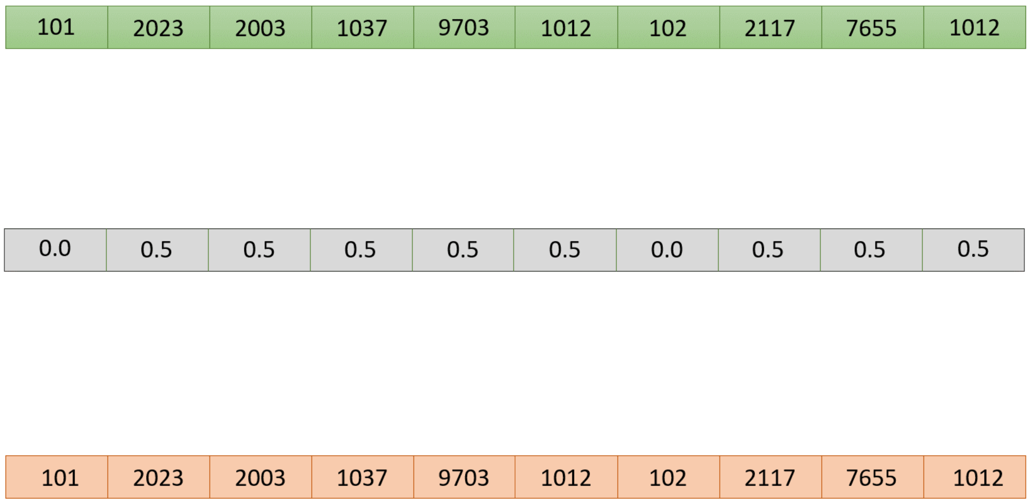
Mask generation — Image by Author

Now, we know that there are some tokens we surely do not want to mask: the special `[CLS]` and `[SEP]` tokens. So, let's set their corresponding probability values in the mask array to zero, effectively ruling out any chance of them being masked.



Mask special tokens — Image by Author

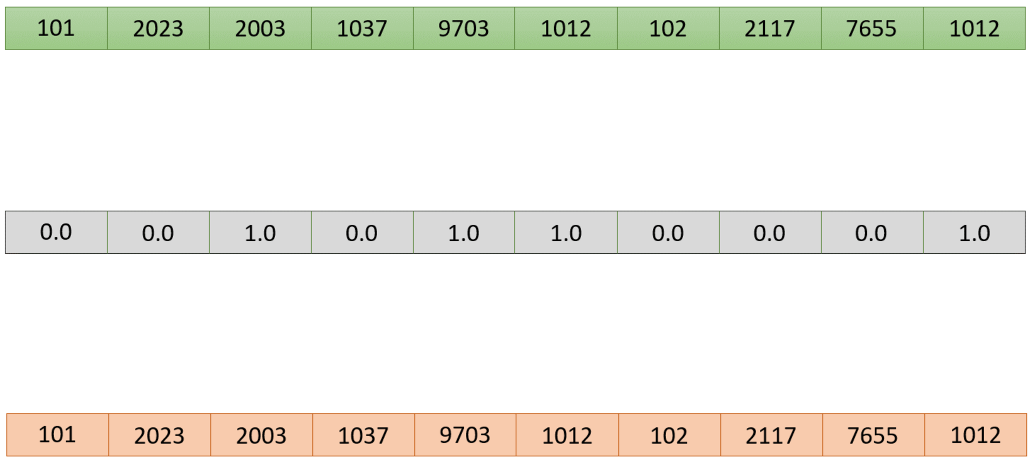
Now, for each position in the mask, let's evaluate the probability and set the value to either `1.0` or `0.0`. Value `1.0` means that we will mask the token in the corresponding position while `0.0` means we will leave it unchanged:



Probability evaluation — Image by Author

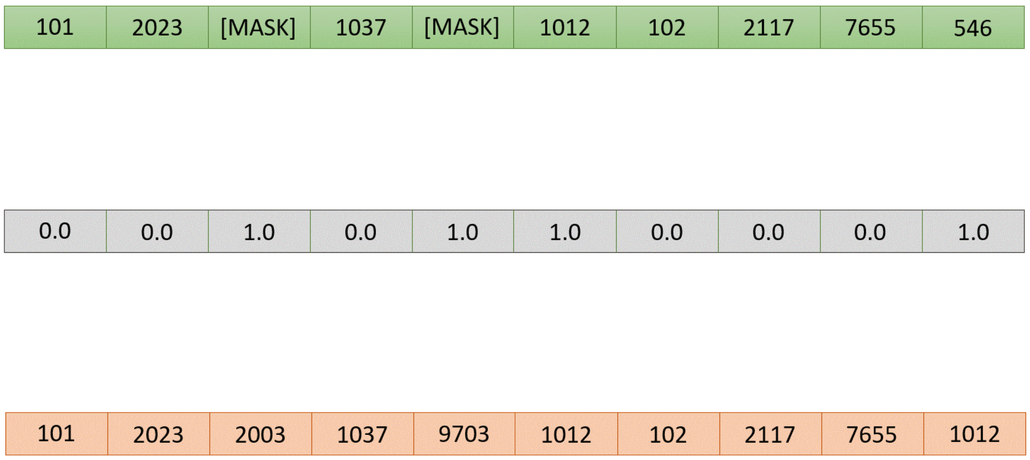
Great, we now have a map that points out which tokens we will mask. But here's a twist: not all masked positions will be replaced by the `[MASK]` token. In fact, we'll employ a more tricky strategy. Here's how it breaks down: 50% of the masked tokens will get the special `[MASK]` token, 25% will be swapped out for a random token, and the remaining 25% will be left as is — though BERT will not know this.

In reality, we use an 80–10–10 scheme, but to make it work with our illustration, let's employ the 50–25–25 strategy:



Mask tokens — Image by Author

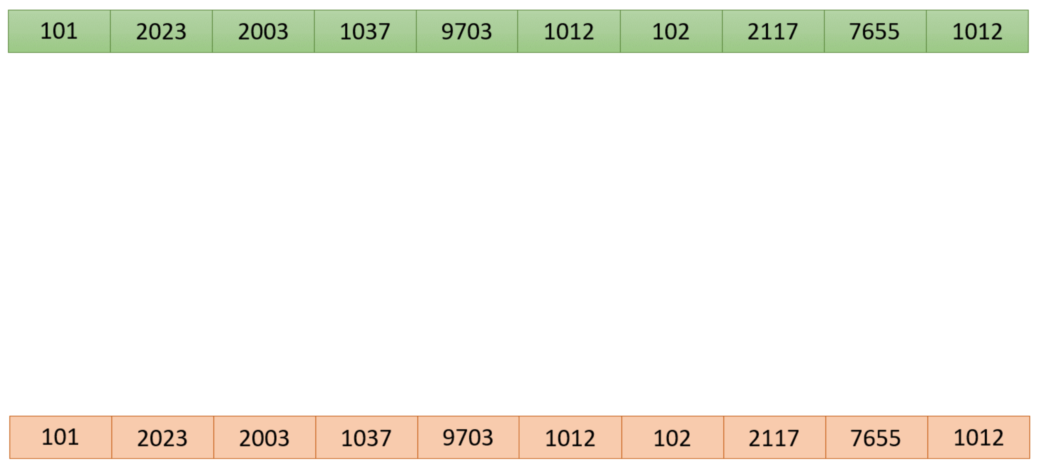
Finally, it's time to treat the labels. The labels at indices that should not contribute to the loss — i.e., the unmasked tokens — are set to `-100`. This is not a random negative number. It's the default value for the `ignore_index` property in PyTorch's CrossEntropyLoss. Thus, Pytorch will ignore these labels when calculating the loss:



101	2023	[MASK]	1037	[MASK]	1012	102	2117	7655	546
0.0	0.0	1.0	0.0	1.0	1.0	0.0	0.0	0.0	1.0
101	2023	2003	1037	9703	1012	102	2117	7655	1012

Process labels— Image by Author

That's it! Let's bring everything together in a final summarizing animation:



Masked Language Modeling — Image by Author

Now that we have a clear mental picture, let's see the code. This is taken directly from `transformers` library:

```
def torch_mask_tokens(self, inputs: Any, special_tokens_mask: Optional[Any] = No
    """
    Prepare masked tokens inputs/labels for masked language modeling: 80% MASK,
    """
    import torch

    labels = inputs.clone()
    # We sample a few tokens in each sequence for MLM training (with probability
    probability_matrix = torch.full(labels.shape, self.mlm_probability)
    if special_tokens_mask is None:
        special_tokens_mask = [
            self.tokenizer.get_special_tokens_mask(val, already_has_special_toke
        ]
        special_tokens_mask = torch.tensor(special_tokens_mask, dtype=torch.bool
    else:
        special_tokens_mask = special_tokens_mask.bool()

    probability_matrix.masked_fill_(special_tokens_mask, value=0.0)
    masked_indices = torch.bernoulli(probability_matrix).bool()
```

```
        labels[~masked_indices] = -100  # We only compute loss on masked tokens

        # 80% of the time, we replace masked input tokens with tokenizer.mask_token
        indices_replaced = torch.bernoulli(torch.full(labels.shape, 0.8)).bool() & m
        inputs[indices_replaced] = self.tokenizer.convert_tokens_to_ids(self.tokeniz

        # 10% of the time, we replace masked input tokens with random word
        indices_random = torch.bernoulli(torch.full(labels.shape, 0.5)).bool() & mas
        random_words = torch.randint(len(self.tokenizer), labels.shape, dtype=torch.
        inputs[indices_random] = random_words[indices_random]

        # The rest of the time (10% of the time) we keep the masked input tokens unc
        return inputs, labels
```

We kick off by cloning the inputs to form the labels — old news by now! Next, we generate a probability matrix that's the same shape as the labels and fill it with a chosen probability value. We then pinpoint the special tokens in the sequence and turn down their masking probabilities to a flat zero. Are you with me? Everything should look familiar!

Here's where it gets spicy: we evaluate the probability matrix using a Bernoulli distribution. Sounds a bit grande, but it's basically a coin flip determining whether each matrix position gets a `1.0` or a `0.0`. Then we translate these ones and zeros to true and false values. Any 'false' hits in the matrix? We set their corresponding label values to `-100`, as we discussed previously.

Finally, we bring into play the 80–10–10 masking scheme we previously touched on, using the same Bernoulli logic. This effectively masks, replaces, or leaves tokens untouched in the input sequence, offering BERT a varied diet of data to chew on.

And that's all! It's a one-liner to go through all these steps using the `datasets` library. Instantiate a `DataCollatorForLanguageModeling` object and then pass it in the `Trainer` (more on the `Trainer` in the next episode):

```
data_collator = DataCollatorForLanguageModeling(tokenizer=tokenizer, mlm_probabi
```

Conclusion

In the journey to train BERT from scratch, preparing the dataset is often the most labor-intensive and difficult step but also one of the most rewarding.

If you've followed along, you've not just learned how to prepare the dataset but also the reason behind each decision in dataset preparation. From selecting the ideal dataset and breaking down its features to the fine art of masking tokens using a varied strategy, we've delved into the details that could make or break your model's performance.

In the next chapter, we'll finally touch on the training process. See you there!

About the Author

My name is Dimitris Poulopoulos, and I'm a machine learning engineer working for HPE. I have designed and implemented AI and software solutions for major clients such as the European Commission, IMF, the European Central Bank, IKEA, Roblox and others.

If you are interested in reading more posts about Machine Learning, Deep Learning, Data Science, and DataOps, follow me on Medium, LinkedIn, or @james2pl on Twitter.

Opinions expressed are solely my own and do not express the views or opinions of my employer.

Technology Artificial Intelligence Deep Learning NLP Editors Pick

---

More from the list: "NLP"

Curated by Himanshu Birla

Jon Gi... in Towards Data ...
Characteristics of Word Embeddings
11 min read · Sep 4, 2021

Jon Gi... in Towards Data ...
The Word2vec Hyperparameters
6 min read · Sep 3, 2021

Jon Gi... in
The Word2ve
15 min rea

View list

Written by Dimitris Poulopoulos

Follow

12.7K Followers · Writer for Towards Data Science

Machine Learning Engineer. I talk about AI, MLOps, and Python programming.
More about me: www.dimpo.me

More from Dimitris Poulopoulos and Towards Data Science

Dimitris Poulopoulos in Towards Data Science

The Ultimate Guide to Training BERT from Scratch: Introduction

Demystifying BERT: The definition and various applications of the model that...

· 10 min read · Sep 2

156

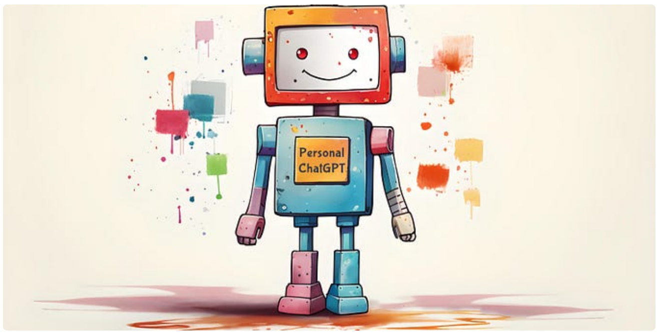
Antonis Makropoulos in Towards Data Science

How to Build a Multi-GPU System for Deep Learning in 2023

This story provides a guide on how to build a multi-GPU system for deep learning and...

10 min read · Sep 17

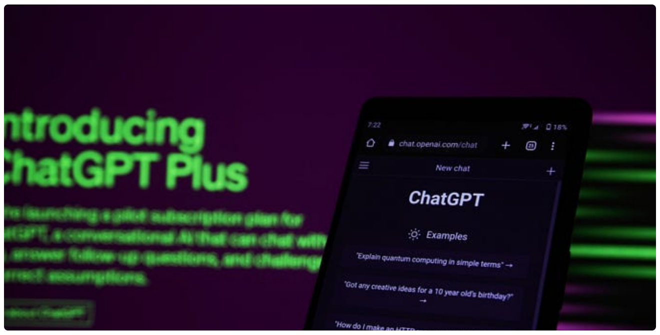
549 11



Robert A. Gonsalves in Towards Data Science

Your Own Personal ChatGPT

How you can fine-tune OpenAI's GPT-3.5 Turbo model to perform new tasks using you...



Dimitris Poulopoulos in Towards AI

Revolutionizing Human-Machine Interaction: The Emergence of...

Decoding the art and science of prompt engineering, the secret sauce for...

· 15 min read · Sep 8

595 7

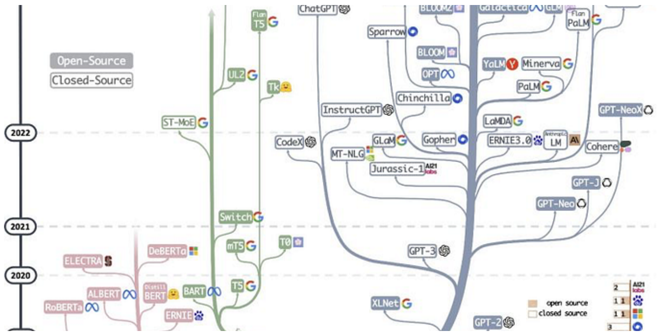
· 5 min read · Aug 30

60

See all from Dimitris Poulopoulos

See all from Towards Data Science

Recommended from Medium



Haifeng Li

A Tutorial on LLM

Generative artificial intelligence (GenAI), especially ChatGPT, captures everyone's…

15 min read · Sep 14

372

Tayyib Ul Hassan Gondal

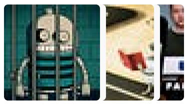
Demystifying Transformers in NLP

Transformers are taking over the world! Well, not really, but they are becoming increasingl…

11 min read · Sep 15

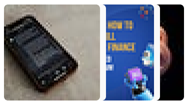
71

Lists



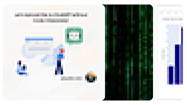
AI Regulation

6 stories · 138 saves



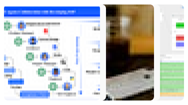
ChatGPT prompts

24 stories · 459 saves



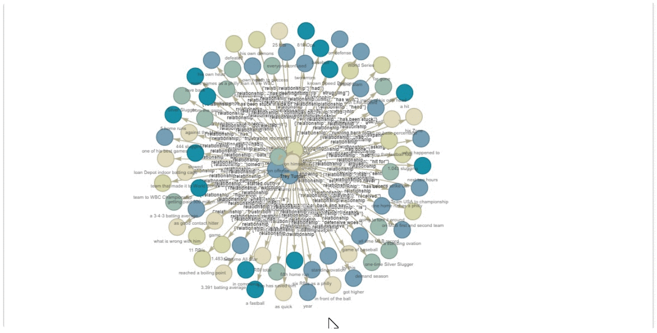
ChatGPT

21 stories · 179 saves



Natural Language Processing

669 stories · 283 saves



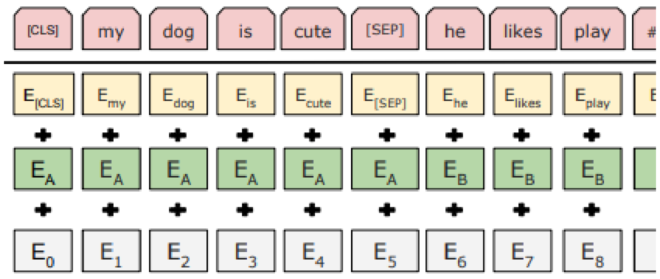
Wenqi Glantz in Better Programming

7 Query Strategies for Navigating Knowledge Graphs With...

Exploring NebulaGraph RAG Pipeline with the Philadelphia Phillies

· 17 min read · 4 days ago

501 4



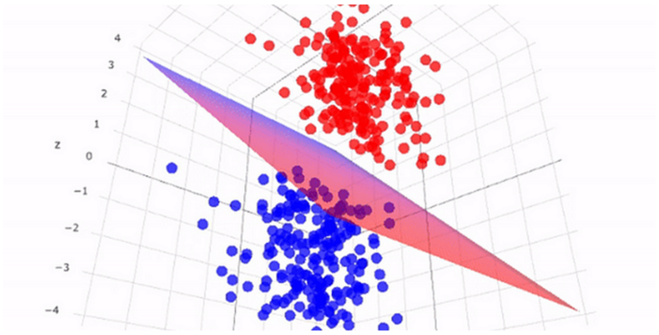
Zain ul Abideen

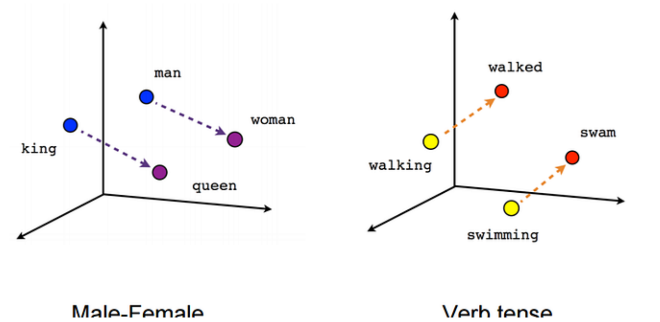
A Comparative Analysis of LLMs like BERT, BART, and T5

Exploring Language Models

6 min read · Jun 26

20 1





Tasmay Pankaj Tibre... in Low Code for Data Scie...

Maninder Singh

Support Vector Machines (SVM): An Intuitive Explanation

Everything you always wanted to know about this powerful supervised ML algorithm

17 min read · Jul 1

732 4

Accelerate Your Text Data Analysis with Custom BERT Word…

One thing is for sure the way humans interact with each other naturally is one of the most…

4 min read · Apr 24

156

See more recommendations