Member-only story

The Ultimate Guide to Training BERT from Scratch: The Tokenizer

From Text to Tokens: Your Step-by-Step Guide to BERT Tokenization

Dimitris Poulopoulos · Following

Published in Towards Data Science · 13 min read · Sep 6

Photo by Glen Carrie on Unsplash

Did you know that the way you tokenize text can make or break your language model? Have you ever wanted to tokenize documents in a rare language or a specialized domain? Splitting text into tokens, it's not a chore; *it's a gateway to transforming language into actionable intelligence*. This story will teach you everything you need to know about tokenization, not only for BERT but for any LLM out there.

In my last story, we talked about BERT, explored its theoretical foundations and training mechanisms, and discussed how to fine-tune it and create a questing-answering system. Now, as we go further into the intricacies of this

groundbreaking model, it's time to spotlight one of the unsung heroes: *tokenization*.

The Ultimate Guide to Training BERT from Scratch: Introduction

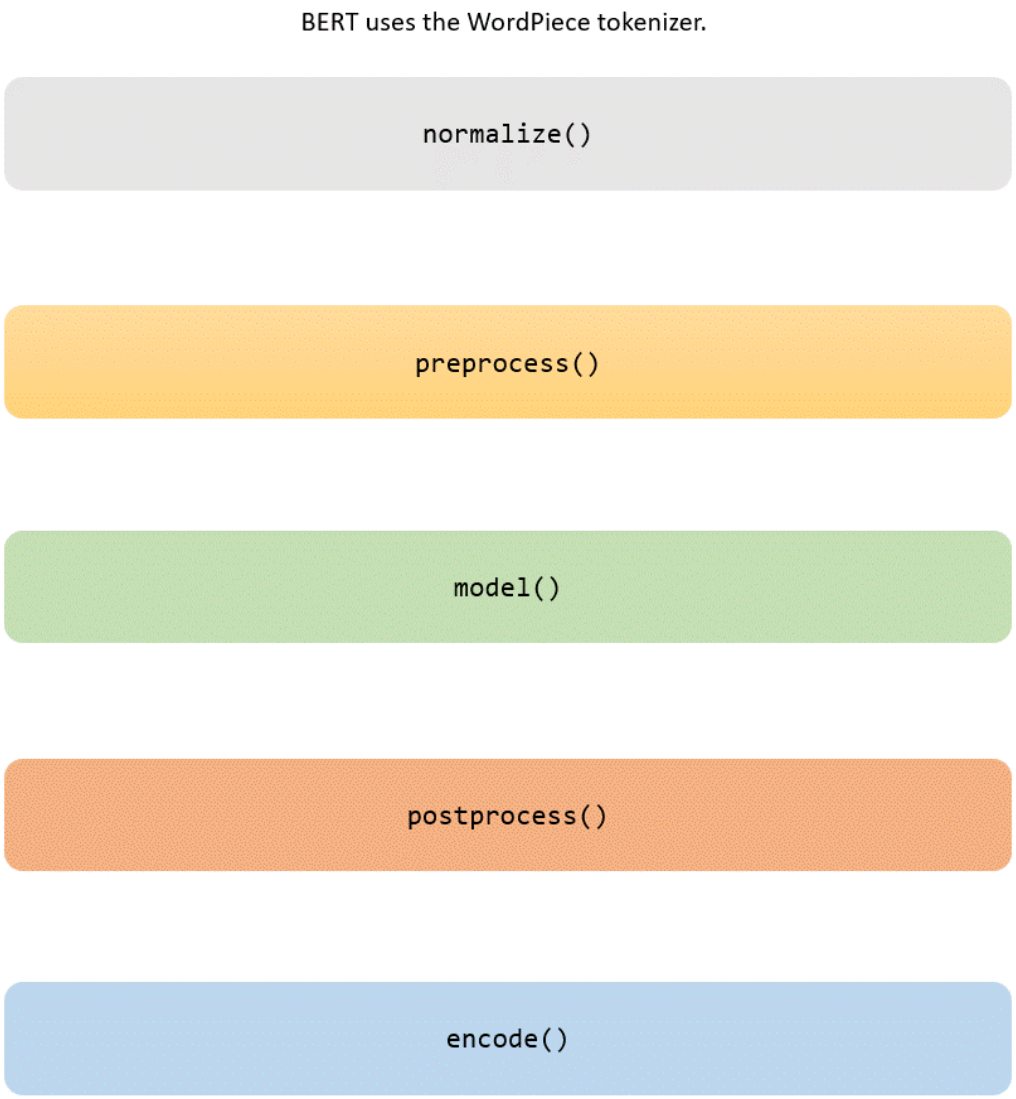
Demystifying BERT: The definition and various applications of the model that changed the NLP landscape.

towardsdatascience.com

> *Part III of this story is now live.*

I get it; tokenization might seem like the last boring obstacle between you and the thrilling process of training your model. Believe me, I used to think the same. But I'm here to tell you that tokenization is not just a "necessary evil"— *it's an art form in its own right.*

In this story, we'll examine every part of the tokenization pipeline. Some steps are trivial (like normalization and pre-processing), while others, like the modeling part, are what make each tokenizer unique.



Tokenization pipeline — Image by Author

By the time you finish reading this article, you'll not only understand the ins and outs of the BERT tokenizer, but you'll also be equipped to train it on your own data. And if you're feeling adventurous, you'll even have the tools to customize this crucial step when training your very own BERT model from scratch.

> Splitting text into tokens, it's not a chore; it's a gateway to transforming language into actionable intelligence.

So, why is tokenization so critical? *At its essence, tokenization is a translator;* it takes in human language and translates it to the language machines can understand: numbers. But there's a catch: During this translation process, the tokenizer must keep a crucial balance, finding the sweet spot between meaning and computational efficiency. So, you see, it's not just about crunching numbers; *it's about efficiently capturing the essence of language in a form that machines can comprehend.*

Now that we have our premise let's start by becoming acquainted with the different types of tokenizers. The first hurdle in this journey is settling on a definition for what constitutes a 'word' — and it might not be as straightforward as what you or I typically consider when we discuss language.

> *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!*

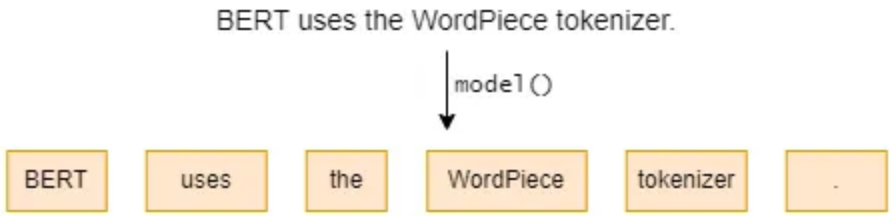
Types of Tokenization

Since the modeling step of the tokenization pipeline is the more demanding one, it's crucial to approach it while the day is still young.

There are various approaches to building a translator between human and machine language. Let's explore the three main types and weigh their pros and cons to understand why subword tokenization is often the go-to method in today's NLP landscape.

Word-based Tokenizers

The word-based approach is the simplest, essentially slicing up raw text into words based on whitespace or other delimiters like punctuation. Think of Python's `split()` function as a classic example.



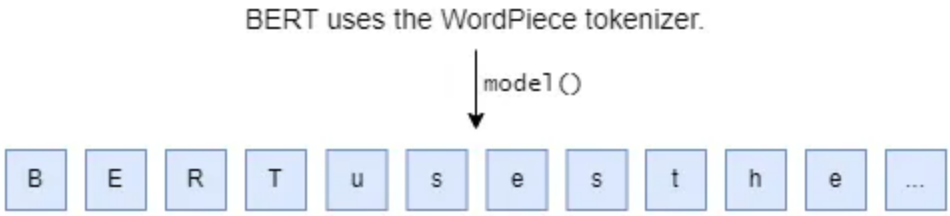
Word-based tokenization-Image by Author

However, this method has its limitations. To truly grasp a language in all its complexity, you'd need to manage a vast token vocabulary—English alone boasts over half a million words. Moreover, this method struggles with inflected forms; words like "dog" and "dogs" are treated as completely separate entities. And if you choose to limit your vocabulary size, the "unknown" token, a placeholder for any word that's not in there, will be all over the place, polluting your data's meaning.

Character-based Tokenizers

On the other end of the spectrum, we have character-based tokenizers, which break down text into individual characters. This drastically reduces the vocabulary size and nearly eliminates the problem of unknown tokens. For example, the English language has 26 characters. If you add punctuation

and other symbols, you're looking at a vocabulary of a size into the hundreds.

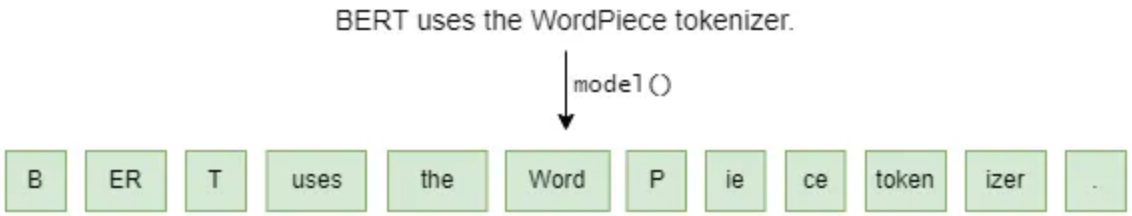


Character-based tokenizers — Image by Author

But it comes with a cost: characters in isolation often lack meaningful context, and this method can result in cumbersome sequences that are computationally expensive to process since the context the model has to keep in its cash grows really fast.

Subword Tokenization

Subword tokenization is the middle ground that often offers the best of both worlds. By dissecting rare or complex words into smaller, meaningful units—like breaking "annoyingly" into "annoy", "ing", and "ly" — this method strikes a balance between efficiency and expressiveness. It offers compact vocabularies without sacrificing the richness of language, making it particularly useful for languages where words can be naturally composed of smaller, meaningful pieces.



Subword tokenizers — Image by Author

> *Note that in the figure, we skip most of the steps in the pipeline for the sake of simplicity. In a real-world scenario, most of the time the tokenizer has already transformed each word to lower case.*

Take the word “annoyingly” as a case in point for the power of subword tokenization. When split into three distinct subtokens — “annoy”, “ing”, and “ly”, each fragment serves as a mini-lesson in language comprehension for the model.

The first subtoken, “annoy,” instructs the tokenizer about the root verb, offering a base layer of meaning. The second subtoken, “ing”, captures a common suffix that serves multiple grammatical functions. It could indicate an ongoing action (as in “running”), the process or result of an action (as in “building”), or even help form nouns (as in “painting”). Finally, the “ly” subtoken is a clue to the tokenizer about how adverbs or adjectives are often constructed in English, helping the model understand that the word modifies an action or describes a characteristic of a noun.

So, why has subword tokenization become the gold standard? *It efficiently navigates the trade-offs of its word-based and character-based counterparts.* You get a vocabulary with almost zero unknown tokens and a system that’s robust enough to capture the nuances of human language, even in its most intricate forms.

BERT uses a subword tokenization method known as “WordPiece”. The rest of this story explores how this method works and how you can train it yourself.

The WordPiece Tokenizer

So, let's build the WordPiece tokenizer from scratch to understand everything that's going under the hood. **Our approach will be twofold: first, we'll construct a mental framework by employing various illustrations to clarify the concepts. Then, we'll put theory into practice by training our very own tokenizer on a custom corpus, utilizing the tokenizers library.**

To kick things off, we need a corpus — a dataset of text that our tokenizer will learn from. Let's consider the following paragraph as our starting point:

> *The WordPiece algorithm serves a crucial role in the architecture of BERT, a state-of-the-art language model. Specifically, WordPiece is responsible for the tokenization process, breaking down text into smaller, manageable units or tokens. This tokenization step is critical during the pre-training phase of BERT, allowing the model to effectively learn the relationships between words or sub-words. By using WordPiece for tokenization, BERT can be more flexible in handling various linguistic constructs and nuances.*

So, the first step is to count the appearances of each word in our corpus and split them into the smallest possible unit:

Image by Author

In our hands-on exercise, you'll notice that we initially split words into individual characters. But that's not the whole story: we prepend the symbol `##` to characters that originally appeared in the middle of a word, setting them apart from those that were initial letters. You'll see later why this is important.

If we collect all the splits, we create our initial tokenizer vocabulary. WordPiece starts from the smallest possible vocabulary you could have (that of single characters) and keeps expanding it to a limit that we set. How does it do that? I'm glad you asked! Let's see; the next step is to identify each possible pair of characters in our vocabulary:

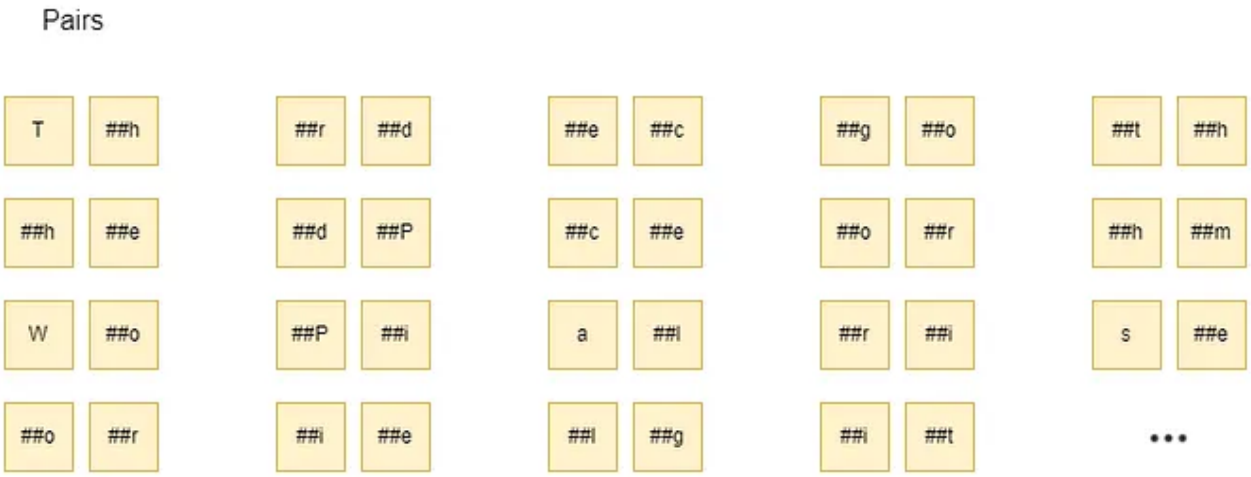


Image by Author

Now, we need to assign a score to each pair. The formula that calculates the score goes as follows:

$$score = \frac{\text{frequency\_of\_pair}}{\text{frequency\_of\_first\_element} \times \text{frequency\_of\_second\_element}}$$

Image by Author

Thus, if we try to calculate the score of the first pair (`T`, `##h`) we get the following:

$$score = \frac{2}{2 \times 12} = 0.0833$$

Image by Author

Using the same approach, we calculate the score for every pair and identify the pair with the highest score. In our case, this is the pair (`##E`, `##R`), so next, we can add this pair to our vocabulary and update the splits by merging these characters. Our new vocabulary has now one more token:

Image by Author

Next, we go back to our splits again, identify every possible pair, score it, and append the pair with the highest score to our vocabulary. What we're doing here is essentially a form of optimization. By identifying commonly

appearing pairs and treating them as single units, we make the tokenizer more efficient.

We continue this process of identifying, scoring, and appending high-scoring pairs until we hit the target number of tokens we wish to include in our vocabulary. So, by setting the desired number of tokens to `60`, we get the following vocabulary:

> *'##E', '##P', '##R', '##T', '##a', '##b', '##c', '##d', '##e', '##f', '##g', '##h', '##i', '##k', '##l', '##m', '##n', '##o', '##p', '##r', '##s', '##t', '##u', '##v', '##w', '##x', '##y', '##z', ',', '-', '.', 'B', 'S', 'T', 'W', 'a', 'b', 'c', 'd', 'e', 'f', 'h', 'i', 'l', 'm', 'n', 'o', 'p', 'r', 's', 't', 'u', 'v', 'w', '##ERT'*

Finally, to calculate the subtokens of a word, we work from the beginning of the string, trying to identify the longest possible sequence that appears in our vocabulary. So, for example, the word `BERT` is split into the following subwords: `B`, `##ERT`:

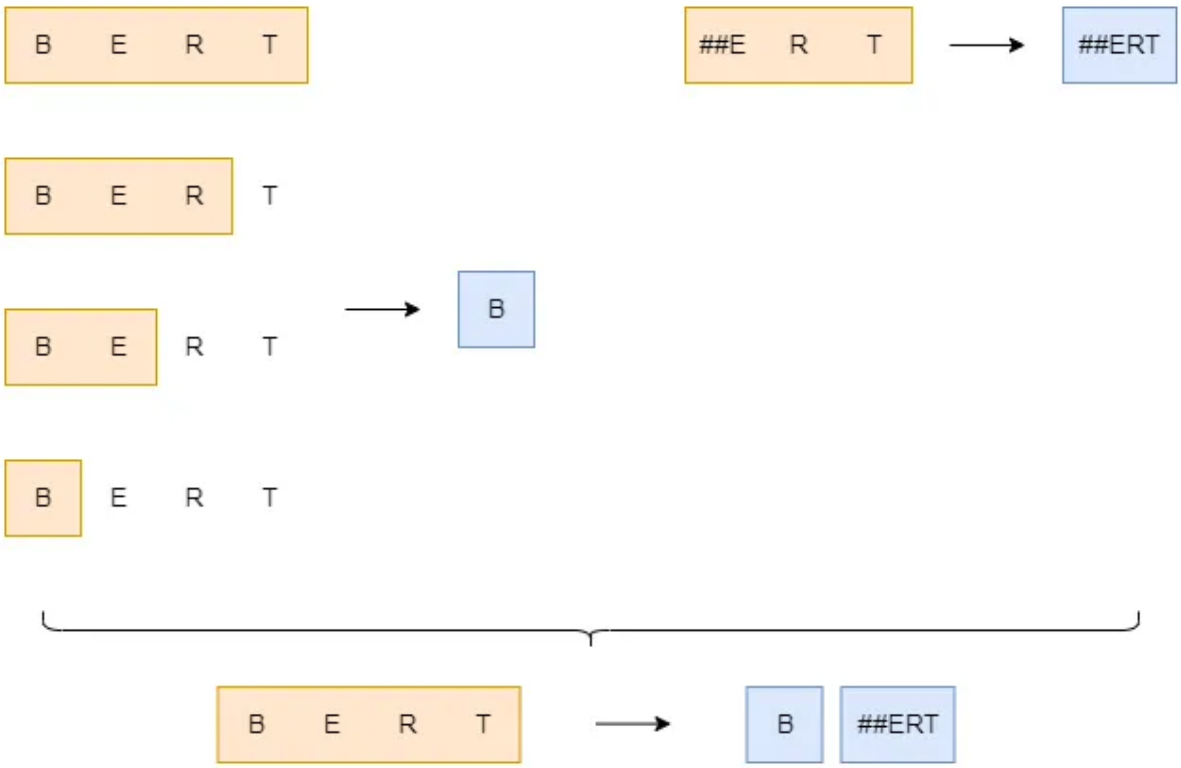


Image by Author

As the finishing touch to our tokenizer's vocabulary, we introduce a set of special tokens that serve specific functions within the BERT model and serve as linguistic markers. Generally, these special tokens include: `[[PAD], [UNK], [CLS], [SEP], [MASK]]`.

Here's a quick rundown of what each special token is designed to do:

- `[PAD]` : Padding token used to equalize the lengths of different input sequences.
- `[UNK]` : Stands for "unknown" and is used to handle words that are not in the vocabulary.
- `[CLS]` : Short for "classification", this token is prepended to the input when the model is used for classification tasks.

- `[SEP]` : Separator token, often used to indicate the end of one sentence and the beginning of another in multi-sentence tasks.
- `[MASK]` : Used in the training process for the BERT model to indicate positions where the model should predict a missing word.

That's it! You now know how WordPiece, BERT's tokenizer, learns to split each work into subwords. Next, let's train a custom one on our corpus.

Training WordPiece

As a native Greek speaker, I'm particularly excited about the prospect of training BERT's WordPiece tokenizer for the Greek language. And I would strongly encourage you to do the same if your native tongue isn't English. *Localizing these powerful models not only contributes to the diversification of NLP technologies but also offers a chance to tailor them to the nuances of your own language or dialect.*

But if you are an English speaker, consider honing your tokenizer for a specific domain or industry. For example, you could train it on legal documents to capture the idiosyncrasies of legal jargon. Or how about gearing it towards understanding Python code?

In my case, I need to find and download the dataset. These days getting the dataset you need is easier than ever, thanks to the Hugging Face Hub. I'll work with the `greek_legal_code` dataset:

```
from datasets import load_dataset

raw_datasets = load_dataset("greek_legal_code", "chapter")
```

Then, let's quickly create our training corpus. For this, we'll use only the first `1000` examples of the `train` split:

```
training_corpus = (
    raw_datasets["train"][i : i + 1000]["text"]
    for i in range(0, len(raw_datasets["train"]), 1000)
)
```

Next, let's get a handle on the pre-trained BERT tokenizer:

```
from transformers import AutoTokenizer

old_tokenizer = AutoTokenizer.from_pretrained("bert-base-uncased")
```

To compare the before and after, let's see how the original BERT tokenizer would split the Greek intro of Wikipedia on Machine Learning:

```
example = ("Μηχανική μάθηση είναι υποπεδίο της επιστήμης των υπολογιστών,"
           " που αναπτύχθηκε από τη μελέτη της αναγνώρισης προτύπων και της"
           " υπολογιστικής θεωρίας μάθησης στην τεχνητή νοημοσύνη. Το 1959,"
           " ο Άρθουρ Σάμουελ ορίζει τη μηχανική μάθηση ως 'Πεδίο μελέτης που"
           " δίνει στους υπολογιστές την ικανότητα να μαθαίνουν, χωρίς να έχουν"
           " ρητά προγραμματιστεί'")
```


The result is shown below. In total, we get 274 different tokens:

```
['μ', '##η', '##χ', '##α', '##ν', '##ι', '##κ', '##η', 'μ', '##α', '##θ', '##η',
274
```

Now, let's train the tokenizer to understand Greek better. We'll use a vocabulary of `50000` words:

```
tokenizer = old_tokenizer.train_new_from_iterator(training_corpus, 50000)
```

Finally, repeat the test:

```
tokens = tokenizer.tokenize(example)
print(tokens); print(len(tokens))
```

The results are amazing; we reduced the number of tokens to `67`, while the tokenizer seems to understand the nuances of the Greek language better now:

```
['μηχανικη', 'μαθη', '##ση', 'ειναι', 'υποπε', '##διο', 'της', 'επιστημης', 'των
67
```

And there you have it! You're now fully equipped to train a tokenizer tailored to your own corpus. But don't stop there — feel free to experiment with

different algorithms altogether! Want to switch lanes and explore another approach? You can easily load the GPT tokenizer, which employs the Byte Pair Encoding (BPE) algorithm instead of WordPiece. The sky really is the limit when it comes to fine-tuning and customizing your tokenizer.

Conclusion

What a journey it's been! From diving into the intricate details of tokenization to building our very own WordPiece tokenizer, we've covered a lot of ground.

If you've followed along, you're now equipped not just with theoretical knowledge but with the hands-on experience to create a tokenizer tailored to your needs. Maybe you'll adapt it for your native language, perhaps for an industry-specific application, or even for something as intricate as source code analysis.

Remember, the principles we've explored here extend beyond any single language or domain. They're the foundations upon which all Natural Language Processing tasks are built. By understanding and mastering them, you're setting yourself up for success in a field that's only going to become more central to our lives.

It's time to gear up for the next exciting phase — preparing your dataset for model training. Trust me, if you found tokenization enthralling, dataset preparation will be equally eye-opening. This will be where all the pieces we've assembled — the tokenizer, the special tokens, and your domain or language-specific nuances — come together to fuel the BERT model's learning process.

Stay tuned!

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 | NLP | Tokenization | Editors Pick

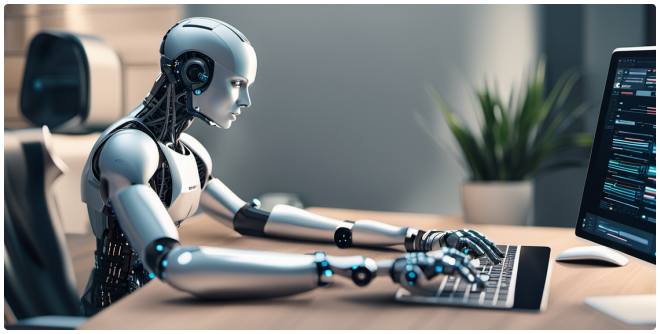
Written by Dimitris Poulopoulos

12.8K Followers · Writer for Towards Data Science

Following

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

Breaking Boundaries: Exploring Function Calling for LLMs

How function calling paves the way for seamless integration of Large Language...

✦ · 8 min read · Aug 10

80

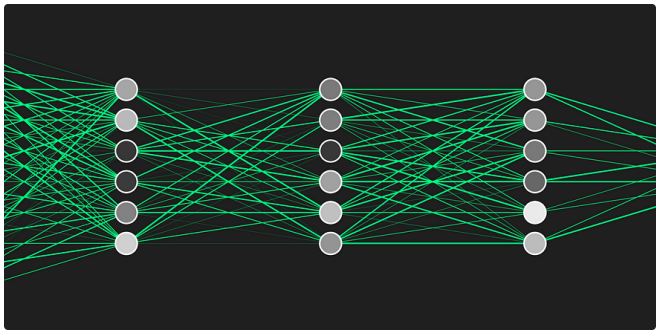
Mike Shakhomirov in Towards Data Science

How to Become a Data Engineer

A shortcut for beginners in 2024

✦ · 17 min read · Oct 7

746 11



Callum Bruce in Towards Data Science

How to Program a Neural Network

A step-by-step guide to implementing a neural network from scratch

✦ · 14 min read · Sep 24

Dimitris Poulopoulos in Towards Data Science

The Power of Linux Cgroups: How Containers Take Control of Their...

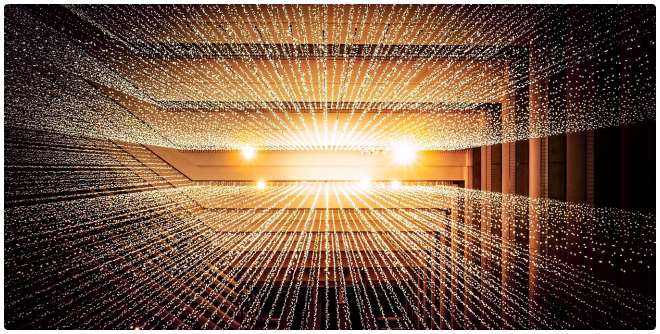
Optimizing Container Resource Allocation with Linux Control Groups

✦ · 8 min read · Jan 10

489 4 111 1

See all from Dimitris Poulopoulos See all from Towards Data Science

Recommended from Medium



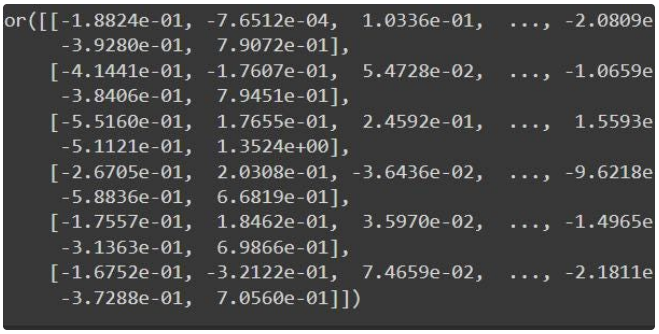
Beatriz Stollnitz in Towards Data Science

Add Your Own Data to an LLM Using Retrieval-Augmented...

Learn how to add your own proprietary data to a pre-trained LLM using a prompt-based...

21 min read · Sep 30

305 3



Abhijat Sarari in Python in Plain English

How to Generate Word Embedding Using BERT?

Introduction

9 min read · Aug 28

2

Lists

AI Regulation

6 stories · 153 saves

ChatGPT prompts

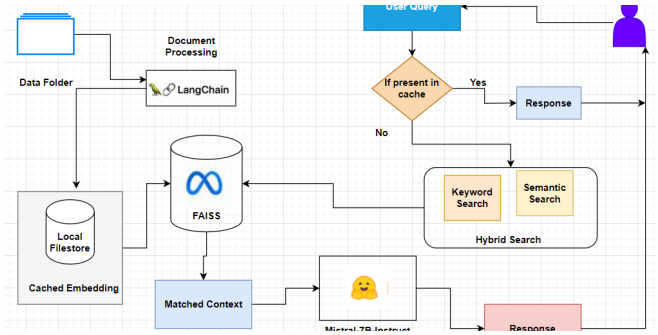
27 stories · 515 saves

ChatGPT

21 stories · 203 saves

Generative AI Recommended Reading

52 stories · 310 saves



Plaban Nayak in AI Planet

Advanced RAG Implementation on Custom Data Using Hybrid Searc...

What is RAG ?

40 min read · Oct 8

96 2

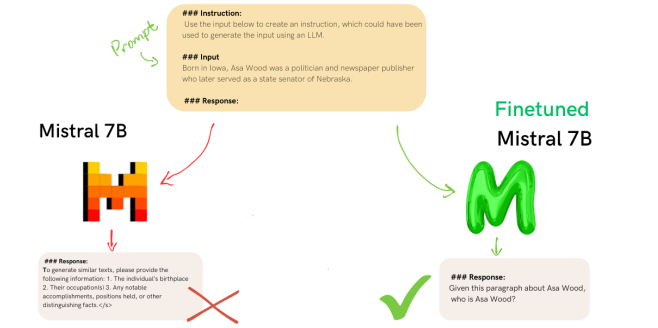
Gathnex

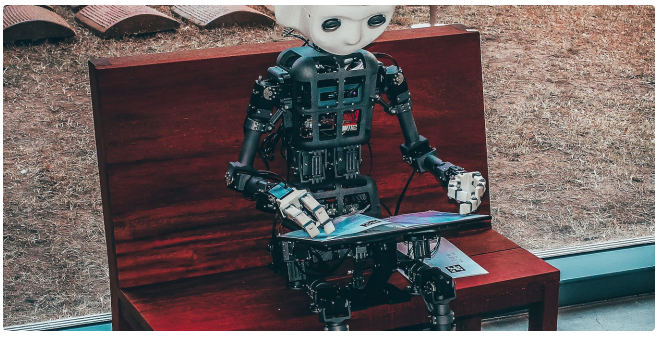
Mistral-7B Fine-Tuning: A Step-by-Step Guide

Introducing Mistral 7B: The Powerhouse of Language Models

5 min read · Oct 4

26 2





Qendel AI in GoPenAI

Fine-tuning Mistral 7B Instruct Model in Colab: A Beginner's Guide

Yanli Liu in Level Up Coding

A Step-by-Step Guide to Runing Mistral-7b AI on a Single GPU wit...

9 Easy Beginner Steps to Fine-tune Mistral 7B Instruct Model on Your Data

✨ · 8 min read · Oct 5

313 4

How to run your AI efficiently through 4-bit Quantization (with Colab notebook...

✨ · 6 min read · Oct 9

317 3

See more recommendations