Search Medium Write

Member-only story

NLP FOR SEMANTIC SEARCH

Fine-Tuning Sentence Transformers with MNR Loss

Next-Gen Sentence Embeddings with Multiple Negatives Ranking Loss

James Briggs · Following
Published in Towards Data Science · 8 min read · Feb 3

38 2

Photo by Lars Kienle on Unsplash. Article originally published on pinecone.io where the author is employed.

Transformer-produced sentence embeddings have come a long way in a very short time. Starting with the slow but accurate similarity prediction of BERT cross-encoders, the world of sentence embeddings was ignited with the introduction of SBERT in 2019 [1]. Since then, many more sentence transformers have been introduced. These models quickly made the original SBERT obsolete.

How did these newer sentence transformers manage to outperform SBERT so quickly? The answer is *multiple negatives ranking (MNR) loss*.

This article will cover what MNR loss is, the data it requires, and how to implement it to fine-tune our own high-quality sentence transformers.

Implementation will cover two training approaches. The first is more involved and outlines the exact steps to fine-tune the model. The second approach makes use of the `sentence-transformers` library's excellent utilities for fine-tuning.

NLI Training

Our article on softmax loss explains that we can fine-tune sentence transformers using **N**atural **L**anguage **I**nference (NLI) datasets.

These datasets contain many sentence pairs, some that *imply* each other and others that *do not imply* each other. As with the softmax loss article, we will use two datasets: the Stanford Natural Language Inference (SNLI) and Multi-Genre NLI (MNLI) corpora.

These two corpora total 943K sentence pairs. Each pair consists of a `premise` and `hypothesis` sentence, which are assigned a `label`:

- **0** — *entailment,* e.g., the `premise` suggests the `hypothesis`.
- **1** — *neutral*, the `premise` and `hypothesis` could both be true, but they are not necessarily related.
- **2** — *contradiction,* the `premise` and `hypothesis` contradict each other.

When fine-tuning with MNR loss, we will drop all rows with *neutral* or *contradiction* labels — keeping only the positive *entailment* pairs.

We will feed sentence A (the `premise`, known as the *anchor*) followed by sentence B (the `hypothesis`, when the label is **0**, this is called the *positive*) into BERT on each step. Unlike softmax loss, we do not use the `label` feature.

These training steps are performed in batches. Meaning several anchor-positive pairs are processed at once.

The model is then optimized to produce similar embeddings between pairs while maintaining different embeddings for non-pairs. We will explain this in more depth soon.

Data Preparation

Let's look at the data preparation process. We first need to download and merge the two NLI datasets. We will use the `datasets` library from Hugging Face.

Because we are using MNR loss, we only want anchor-positive pairs. We can apply a filter to remove all other pairs (including erroneous `-1` labels).

The dataset is now prepared differently depending on our training method. We will continue preparation for the more involved PyTorch approach. If you'd rather just train a model and care less about the steps involved, feel free to skip ahead to the next section.

For the PyTorch approach, we must tokenize our own data. To do that, we will be using a `BertTokenizer` from the `transformers` library and applying the `map` method on our `dataset`.

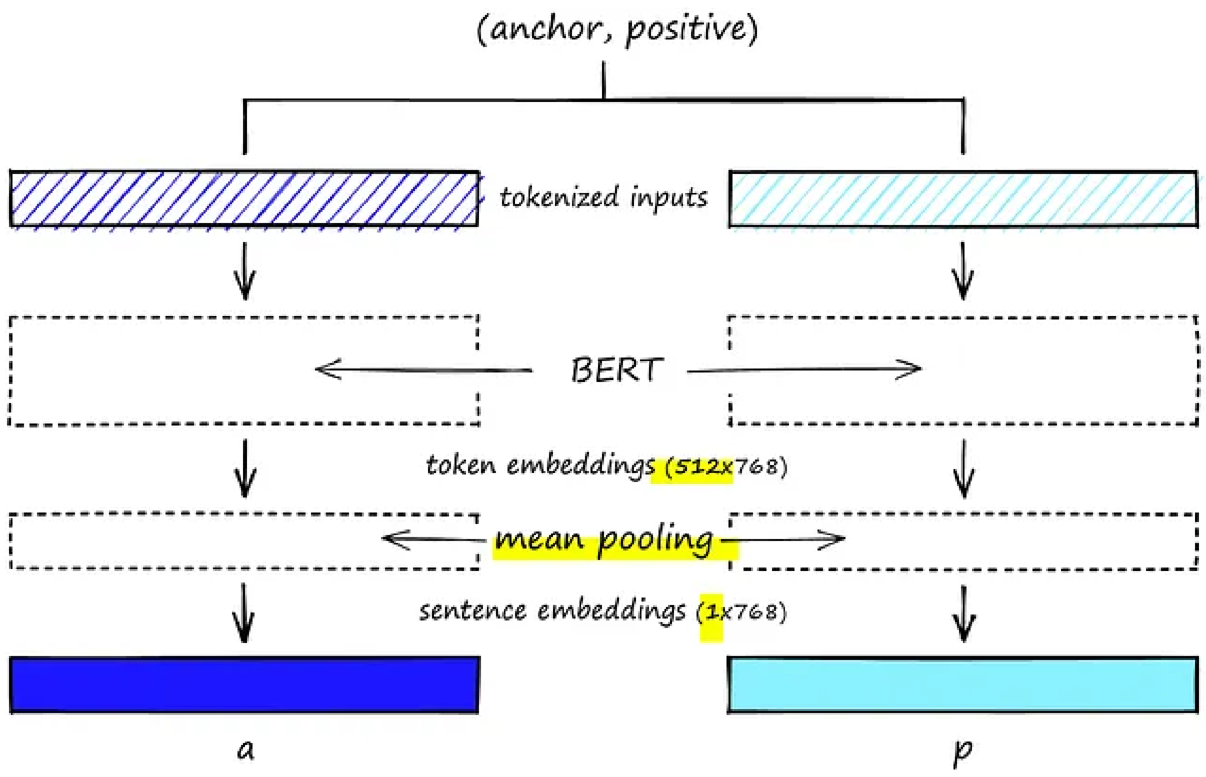
After that, we're ready to initialize our `DataLoader` , which will be used to load data batches into our model during training.

And with that, our data is ready. Let’s move on to training.

PyTorch Fine-Tuning

When training SBERT models, we don’t start from scratch. Instead, we begin with an already pretrained BERT — all we need to do is *fine-tune* it for building sentence embeddings.

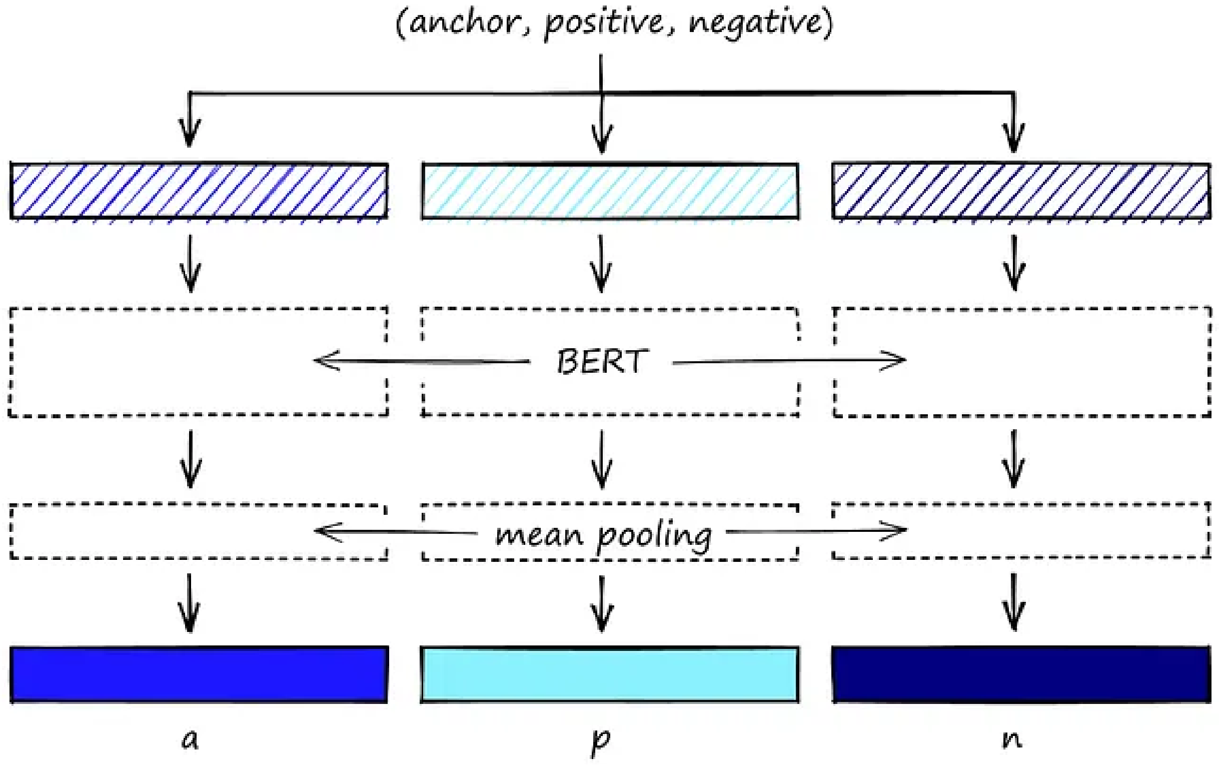
MNR and softmax loss training approaches use a * ‘siamese’*-BERT architecture during fine-tuning. Meaning that during each step, we process a *sentence A* (our *anchor*) into BERT, followed by *sentence B* (our *positive*).



Siamese-BERT network, the anchor and positive sentence pairs are processed separately. A mean pooling layer converts token embeddings into sentence embeddings. Sentence A is our anchor and sentence B the positive.

Because these two sentences are processed *separately*, it creates a *siamese*-like network with two identical BERTs trained in parallel. In reality, there is only a single BERT being used twice in each step.

We can extend this further with *triplet*-networks. In the case of triplet networks for MNR, we would pass three sentences, an *anchor*, it's *positive*, and it's *negative*. However, we are *not* using triplet-networks, so we have removed the negative rows from our dataset (rows where `label` is `2`).



Triplet networks use the same logic but with an added sentence. For MNR loss this other sentence is the negative pair of the anchor.

BERT outputs 512 768-dimensional embeddings. We convert these into *averaged* sentence embeddings using *mean-pooling*. Using the siamese approach, we produce two of these per step — one for the *anchor* that we will call `a`, and another for the *positive* called `p`.

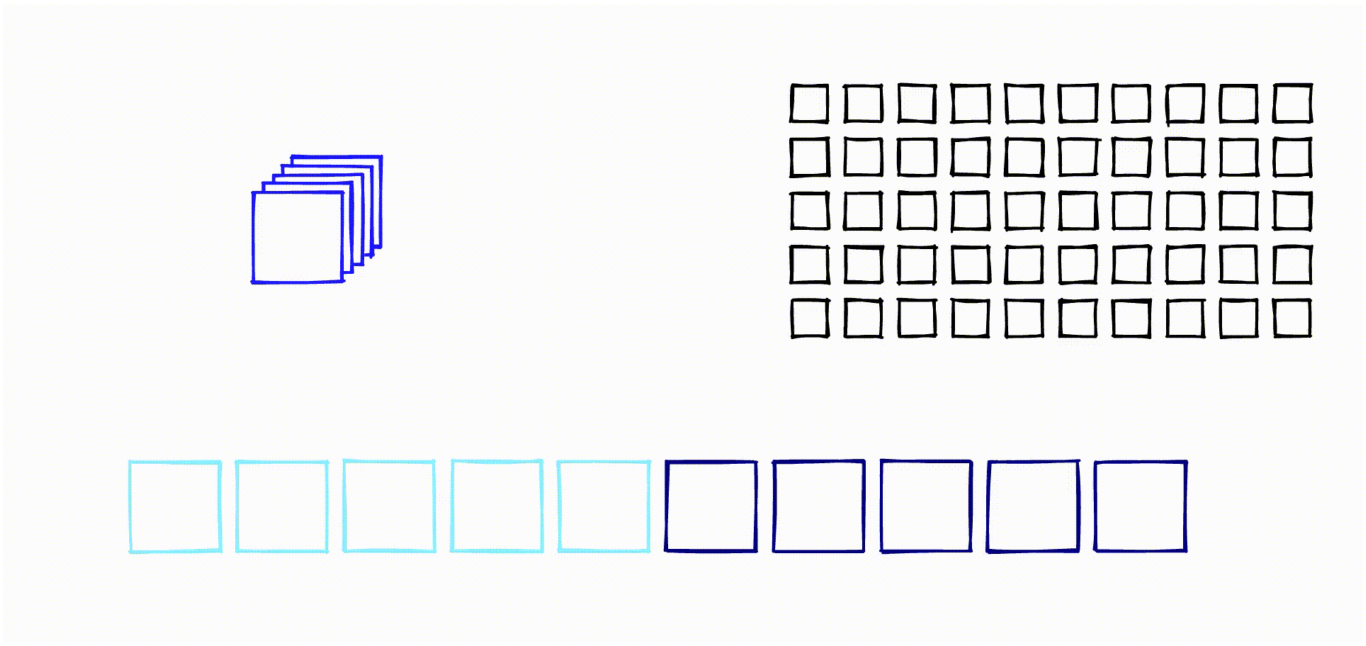
In the `mean_pool` function, we're taking these token-level embeddings (the 512) and the sentence `attention_mask` tensor. We resize the `attention_mask` to match the higher `768` -dimensionality of the token embeddings.

The resized mask `in_mask` is applied to the token embeddings to exclude padding tokens from the mean pooling operation. Mean-pooling takes the average activation of values across each dimension but *excluding* those padding values, which would reduce the average activation. This operation transformers our token-level embeddings (shape `512*768`) to sentence-level embeddings (shape `1*768`).

These steps are performed in *batches*, meaning we do this for many *(anchor, positive)* pairs in parallel. That is important in our next few steps.

First, we calculate the cosine similarity between each anchor embedding (`a`) and *all* of the positive embeddings in the same batch (`p`).

From here, we produce a vector of cosine similarity scores (of size `batch_size`) for each anchor embedding `a_i` *(or size* `2 * batch_size` *for triplets)*. Each anchor should share the highest score with its positive pair, `p_i` .



Cosine similarity scores using five pairs/triples in a triplet network (with `(a, p, n)`). A siamese network is the same but excludes the dark blue `n` blocks (`n`).

To optimize for this, we use a set of increasing label values to mark where the highest score should be for each `a_i` , and categorical cross-entropy loss.

And that’s every component we need for fine-tuning with MNR loss. Let’s put that all together and set up a training loop. First, we move our model and layers to a CUDA-enabled GPU *if available*.

Then we set up the optimizer and schedule for training. We use an Adam optimizer with a linear warmup for 10% of the total number of steps.

And now, we define the training loop using the same training process that we worked through before.

With that, we’ve fine-tuned our BERT model using MNR loss. Now we save it to file.

And this can now be loaded using either the `SentenceTransformer` or HF `from_pretrained` methods. Before we move on to testing the model performance, let's look at how we can replicate that fine-tuning logic using the *much simpler* `sentence-transformers` library.

Fast Fine-Tuning

As we already mentioned, there is an easier way to fine-tune models using MNR loss. The `sentence-transformers` library allows us to use pretrained sentence transformers and comes with some handy training utilities.

We will start by preprocessing our data. This is the same as we did before for the first few steps.

Before, we tokenized our data and then loaded it into a PyTorch `DataLoader`. This time we follow a *slightly different format*. We * don't* tokenize; we reformat into a list of `sentence-transformers` `InputExample` objects and use a slightly different `DataLoader`.

Our `InputExample` contains just our `a` and `p` sentence pairs, which we then feed into the `NoDuplicatesDataLoader` object. This data loader ensures that each batch is duplicate-free — a helpful feature when ranking pair similarity across *randomly* sampled pairs with MNR loss.

Now we define the model. The `sentence-transformers` library allows us to build models using *modules*. We need just a transformer model (we will use `bert-base-uncased` again) and a mean pooling module.

We now have an initialized model. Before training, all that's left is the loss function — MNR loss.

And with that, we have our data loader, model, and loss function ready. All that's left is to fine-tune the model! As before, we will train for a single epoch and warmup for the first 10% of our training steps.

And a couple of hours later, we have a new sentence transformer model trained using MNR loss. It goes without saying that using the `sentence-transformers` training utilities makes life *much easier*. To finish off the article, let’s look at the performance of our MNR loss SBERT next to other sentence transformers.

Compare Sentence Transformers

We’re going to use a semantic textual similarity (STS) dataset to test the performance of *four models*; our *MNR loss* SBERT (using PyTorch and `sentence-transformers`), the *original* SBERT, and an MPNet model trained with MNR loss on a 1B+ sample dataset.

The first thing we need to do is download the STS dataset. Again we will use `datasets` from Hugging Face.

STSb (or STS benchmark) contains sentence pairs in features `sentence1` and `sentence2` assigned a similarity score from *0* -> *5*.

Three samples from the validation set of STSb.

Because the similarity scores range from 0 -> 5, we need to normalize them to a range of 0 -> 1. We use `map` to do this.

We're going to be using `sentence-transformers` evaluation utilities. We first need to reformat the STSb data using the `InputExample` class — passing the sentence features as `texts` and similarity scores to the `label` argument.

To evaluate the models, we need to initialize the appropriate evaluator object. As we are evaluating continuous similarity scores, we use the `EmbeddingSimilarityEvaluator`.

And with that, we're ready to begin evaluation. We load our model as a `SentenceTransformer` object and pass the model to our `evaluator`.

The evaluator outputs the * Spearman's rank correlation* between the cosine similarity scores calculated from the model's output embeddings and the similarity scores provided in STSb. A high correlation between the two values outputs a value close to *+1*, and no correlation would output *0*.

For the model fine-tuned with `sentence-transformers`, we output a correlation of *0.84*, meaning our model outputs good similarity scores according to the scores assigned to STSb. Let's compare that with other models.

The top two models are trained using MNR loss, followed by the original SBERT.

These results support the advice given by the authors of `sentence-transformers`, that models trained with MNR loss outperform those trained with softmax loss in building high-performing sentence embeddings [2].

Another key takeaway here is that despite our best efforts and the complexity of building these models with PyTorch, *every* model trained using the easy-to-use `sentence-transformers` utilities far outperformed them.

In short, fine-tune your models with MNR loss, and do it with the `sentence-transformers` library.

That's it for this walkthrough and guide to fine-tuning sentence transformer models with multiple negatives ranking loss — the current best approach for building high-performance models.

We covered preprocessing the two most popular NLI datasets — the Stanford NLI and multi-genre NLI corpora — for fine-tuning with MNR loss. Then we delved into the details of this fine-tuning approach using PyTorch before taking advantage of the excellent training utilities provided by the `sentence-transformers` library.

Finally, we learned how to evaluate our sentence transformer models with the semantic textual similarity benchmark (STSb) — identifying the highest-performing models.

References

[1] N. Reimers, I. Gurevych, Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks (2019), ACL

[2] N. Reimers, Sentence Transformers NLI Training Readme, GitHub

Artificial Intelligence Deep Learning Machine Learning Naturallanguageprocessing

Vector Search

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

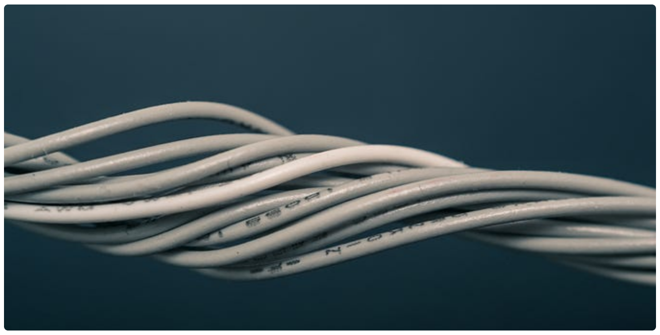
Written by James Briggs

Following

10.9K Followers · Writer for Towards Data Science

Freelance ML engineer learning and writing about everything. I post a lot on YT https://www.youtube.com/c/jamesbriggs

More from James Briggs and Towards Data Science



James Briggs in Towards Data Science

The Right Way to Build an API with Python

All you need to know on API development in Flask

✦ · 7 min read · Sep 11, 2020

1.2K 13

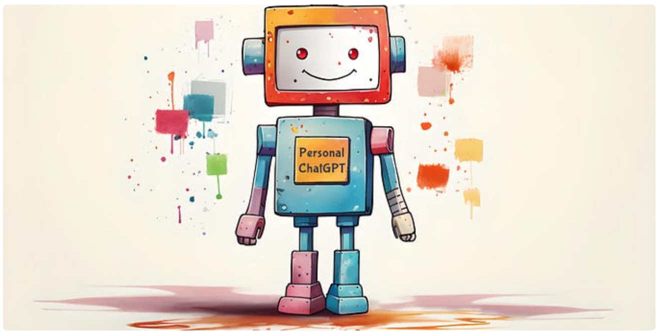
Antonis Makropoulos in Towards Data Science

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

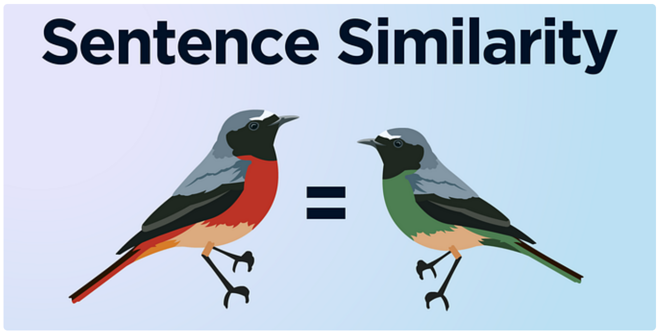
549 11



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…



James Briggs in Towards Data Science

BERT For Measuring Text Similarity

· 15 min read · Sep 8

595 7

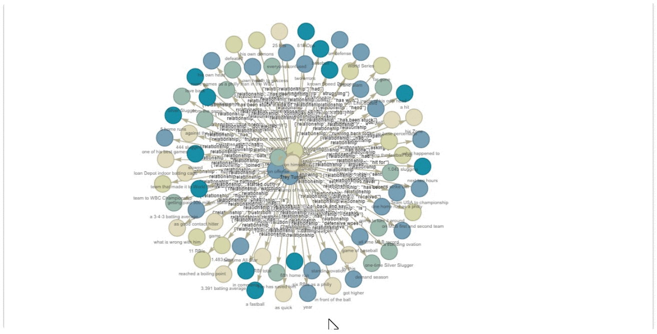
· 5 min read · May 5, 2021

798 8

See all from James Briggs

See all from Towards Data Science

Recommended from Medium



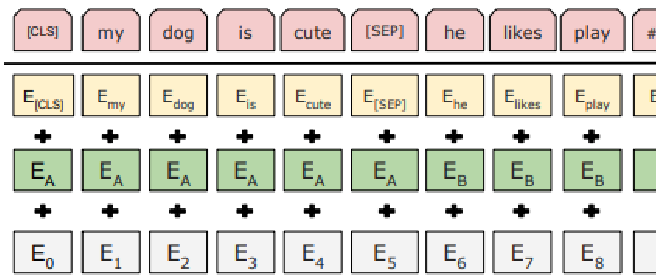
Wenqi Glantz in Better Programming

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

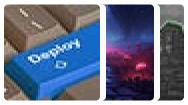
A Comparative Analysis of LLMs like BERT, BART, and T5

Exploring Language Models

6 min read · Jun 26

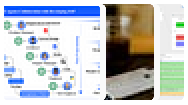
20 1

Lists



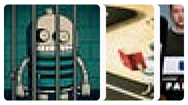
Predictive Modeling w/ Python

20 stories · 452 saves



Natural Language Processing

669 stories · 283 saves

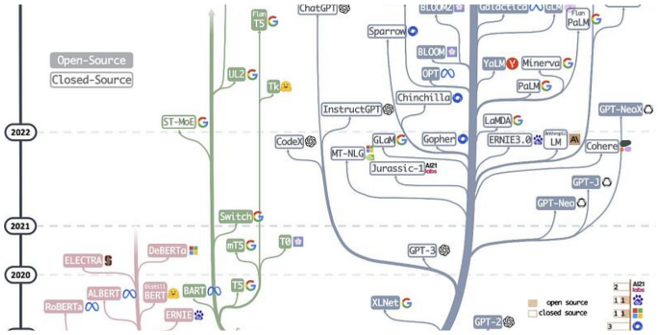


AI Regulation

6 stories · 138 saves

Practical Guides to Machine Learning

10 stories · 519 saves



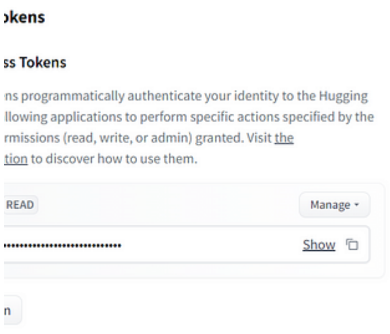
Haifeng Li

A Tutorial on LLM

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

15 min read · Sep 14

372



ɔkens

ss Tokens

ns programmatically authenticate your identity to the Hugging
llowing applications to perform specific actions specified by the
rmissions (read, write, or admin) granted. Visit the
tion to discover how to use them.

READ Manage

•••••••••••••••••••••••••••••• Show

n

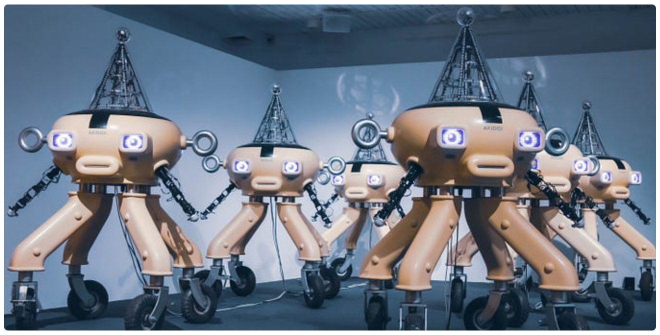
Ankit

Generating Summaries for Large Documents with Llama2 using...

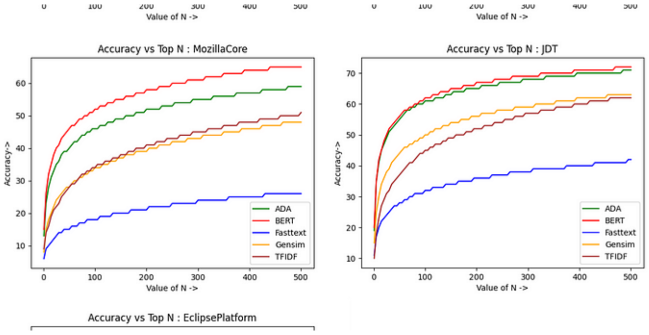
Introduction

11 min read · Aug 28

103 3



AR



Avinash Patil

LoRA: Low-Rank Adaptation from Scratch — Code and Theory

Transformer models can have a lot of parameters which can make fine-tuning the…

5 min read · Aug 3

3

Embeddings: BERT better than ChatGPT4?

In this study, we compared the effectiveness of semantic textual similarity methods for…

4 min read · Sep 19

3 1

See more recommendations