Building LLM Applications: Sentence Transformers (Part 3)

Vipra Singh · Follow

16 min read · Jan 13, 2024

Learn Large Language Models (LLM) through the lens of a Retrieval Augmented Generation (RAG) Application.

Posts in this Series

1. *Introduction*
2. *Data Preparation*
3. *Sentence Transformers (This Post)*
4. *Vector Database*
5. *Search & Retrieval*
6. *LLM*
7. *Open-Source RAG*

8. *Evaluation*
9. *Serving LLMs*
10. *Advanced RAG*

. . .

Table of Contents

Greetings!

In the last blogs, we learned about Data Preparation for RAG which involved Data Ingestion, Data Preparation and Chunking.

As we need to search for relevant contextual chunks during RAG, we have to convert the data from textual format to vector embeddings.

Thus, we will be exploring the most efficient way to convert the text via Sentence Transformers.

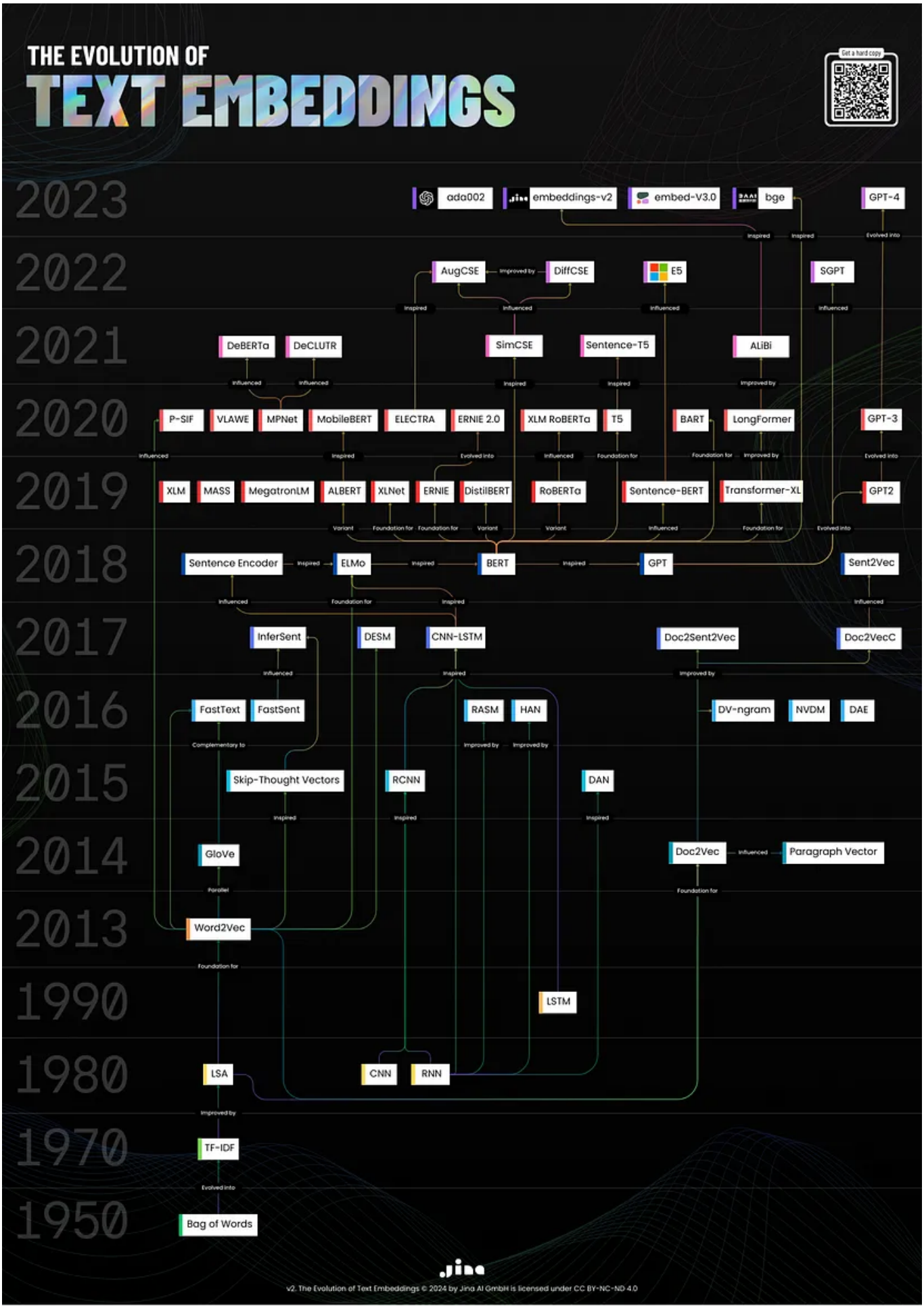

THE EVOLUTION OF
TEXT EMBEDDINGS
Get a hard copy
2023
2022
2021
2020
2019
2018
2017
2016
2015
2014
2013
1990
1980
1970
1950
ada002
embeddings-v2
embed-V3.0
bge
GPT-4
AugCSE
DiffCSE
E5
SGPT
DeBERTa
DeCLUTR
SimCSE
Sentence-T5
ALiBi
P-SIF
VLAWE
MPNet
MobileBERT
ELECTRA
ERNIE 2.0
XLM RoBERTa
T5
BART
LongFormer
GPT-3
XLM
MASS
MegatronLM
ALBERT
XLNet
ERNIE
DistilBERT
RoBERTa
Sentence-BERT
Transformer-XL
GPT2
Sentence Encoder
ELMo
BERT
GPT
Sent2Vec
InferSent
DESM
CNN-LSTM
Doc2Sent2Vec
Doc2VecC
FastText
FastSent
RASM
HAN
DV-ngram
NVDM
DAE
Skip-Thought Vectors
RCNN
DAN
GloVe
Doc2Vec
Paragraph Vector
Word2Vec
LSTM
LSA
CNN
RNN
TF-IDF
Bag of Words
v2. The Evolution of Text Embeddings © 2024 by Jina AI GmbH is licensed under CC BY-NC-ND 4.0

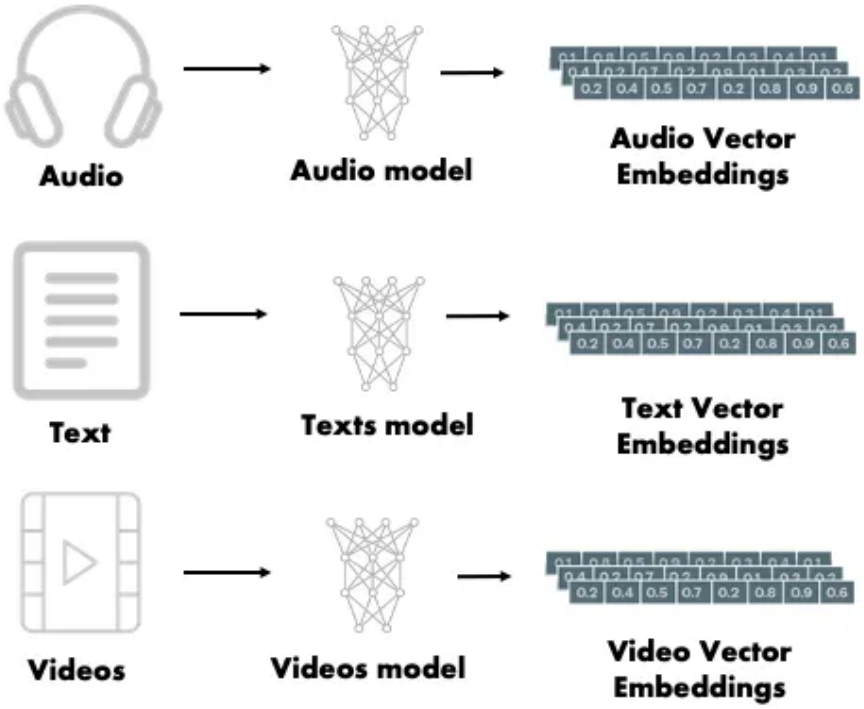
Image by Jina AI

Let's get started with some most commonly used embedding models.

1. Embedding Models

Embeddings are a type of word representation (with numerical vectors) that allows words with similar meanings to have a similar representation.

These vectors can be learned by a variety of machine-learning algorithms and large datasets of texts. One of the main roles of word embeddings is to provide input features for downstream tasks like text classification and information retrieval.



The process of creating vector embeddings from different types of data: Audio, Text, Video

Several word embedding methods have been proposed in the past decade, here are some of them.

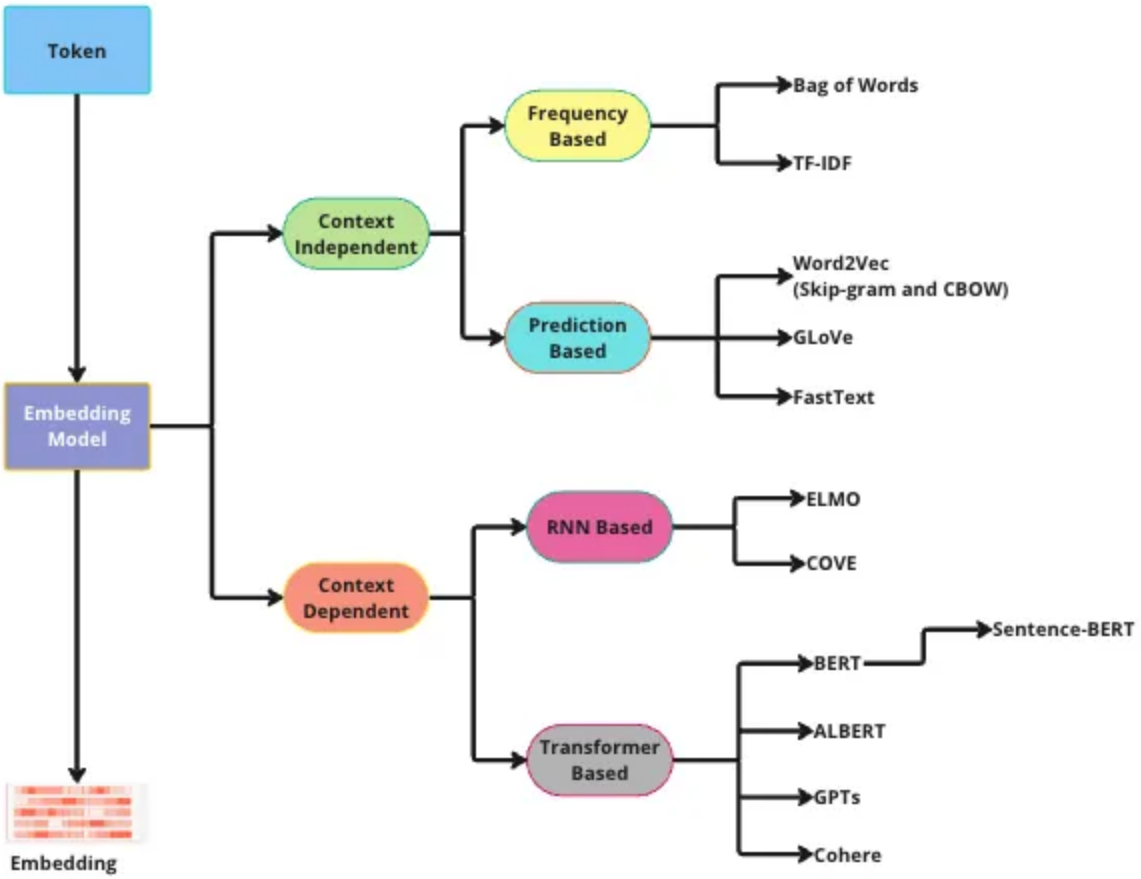


Image by Author

1.1. Context-independent Embeddings

Context-independent embeddings redefine word representations, assigning unique vectors regardless of contextual variations. This concise exploration focuses on the implications for homonym disambiguation.

- Context-independent models allocate distinct vectors to each word, irrespective of context.
- Homonyms like “duck” receive a single vector, blending diverse meanings without contextual cues.
- The approach yields a comprehensive map of word vectors, capturing multiple meanings in a fixed representation.

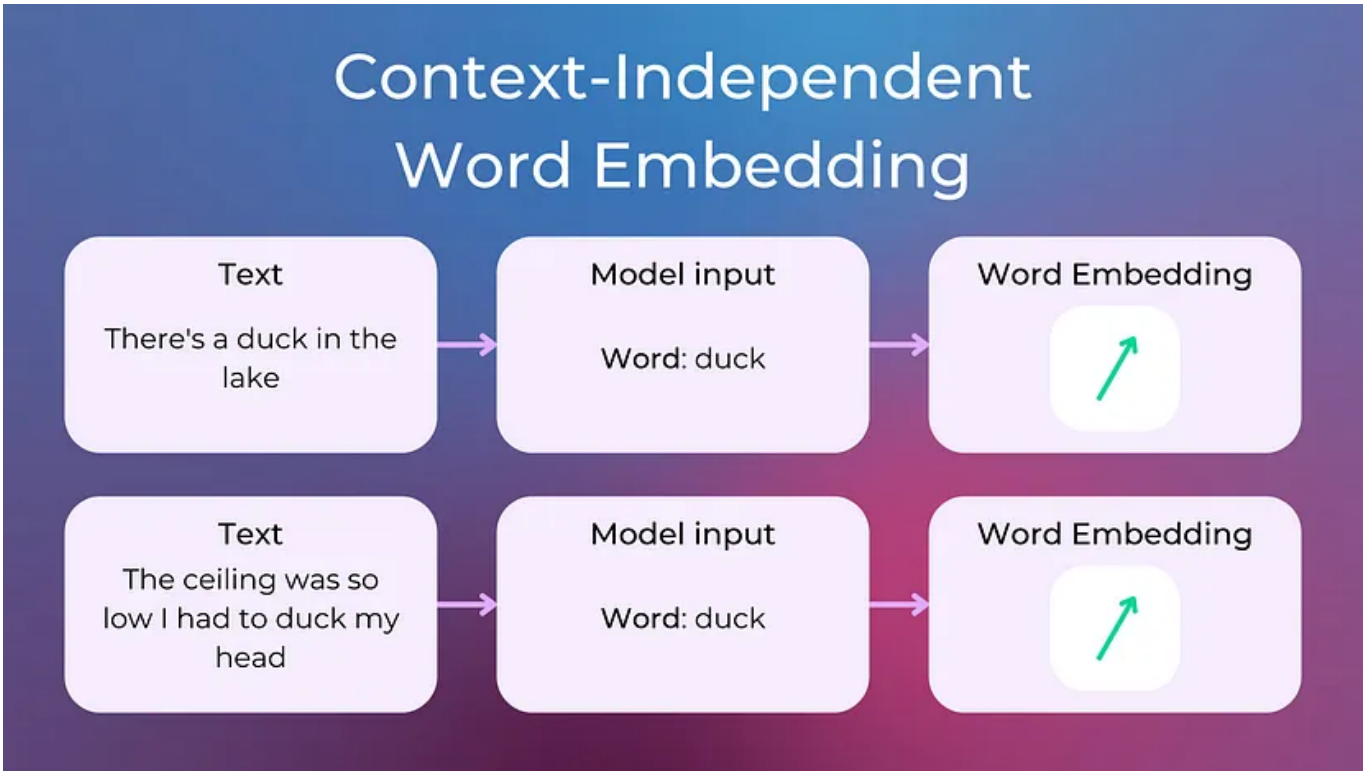


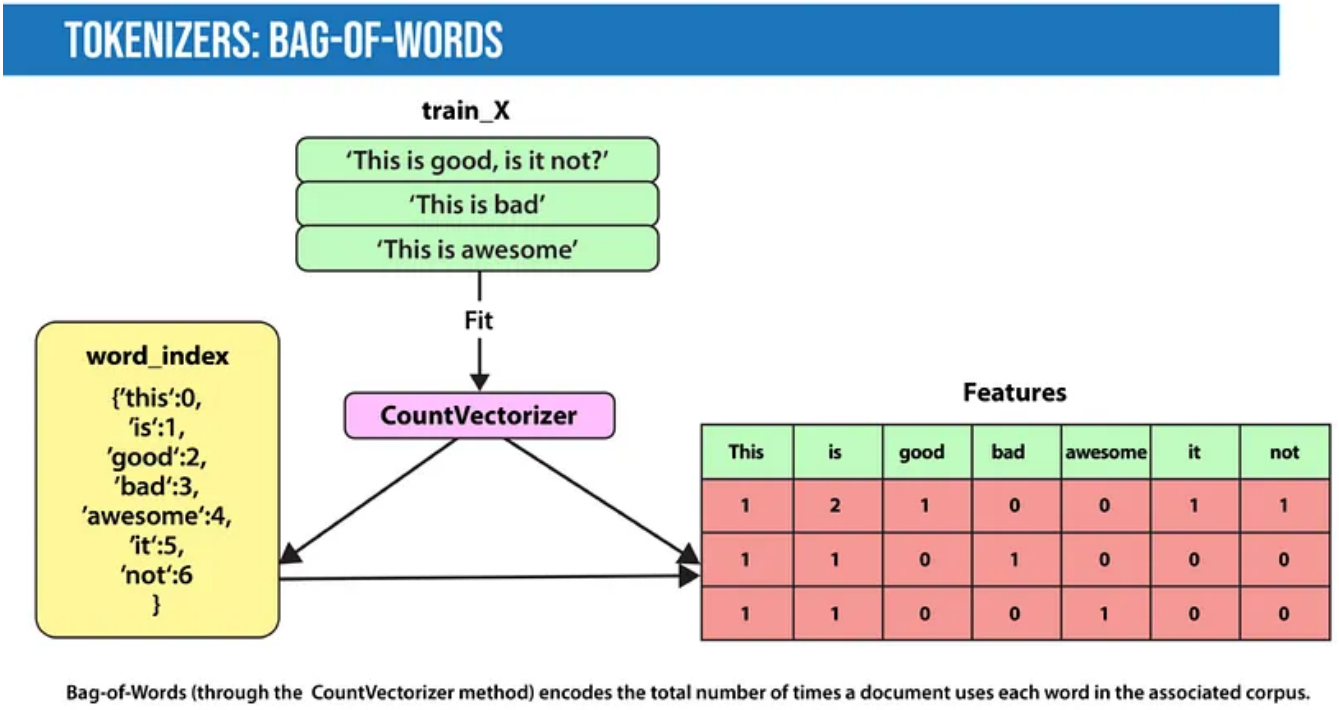
Image by NLP Planet

Context-independent embeddings offer efficiency but pose challenges in nuanced language understanding, especially with homonyms. This paradigm shift prompts a closer look at the trade-offs in natural language processing.

Some common **frequency-based** context-independent embeddings:

- **Bag of Words**

Bag of words will create a dictionary of the most common words in all the sentences and then encode the sentences as shown below.



Bag of Words

- **TF-IDF**

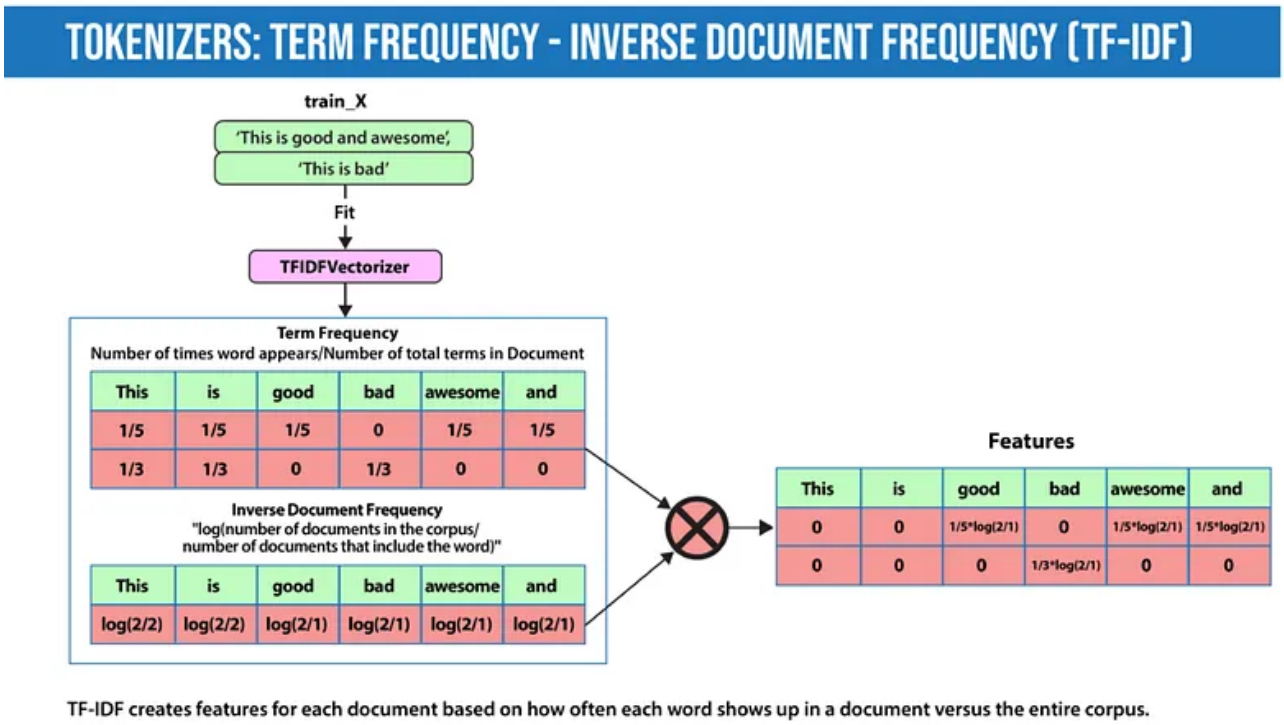
TF-IDF is a simple technique to find features from sentences. While in Count features we take count of all the words/ngrams present in a document, with TFIDF we take features only for the significant words. How do we do that? If you think of a document in a corpus, we will consider two things about any word in that document:

- **Term Frequency:** How important is the word in the document?

$$\text{TF(word in a document)} = \frac{\text{No of occurances of that word in document}}{\text{No of words in document}}$$

- **Inverse Document Frequency:** How important the term is in the whole corpus?

$$\text{IDF(word in a corpus)} = -\log(\text{ratio of documents that include the word})$$



TF-IDF

Some common **prediction-based** context-independent embeddings:

- **Word2Vec**:
  Word embeddings in Word2Vec are learned through a two-layer neural network, which inadvertently captures linguistic contexts during the training process. The embeddings serve as a byproduct of the algorithm's primary objective, showcasing the efficiency of this approach. Word2Vec provides flexibility through two distinct model architectures: CBOW and continuous skip-gram.

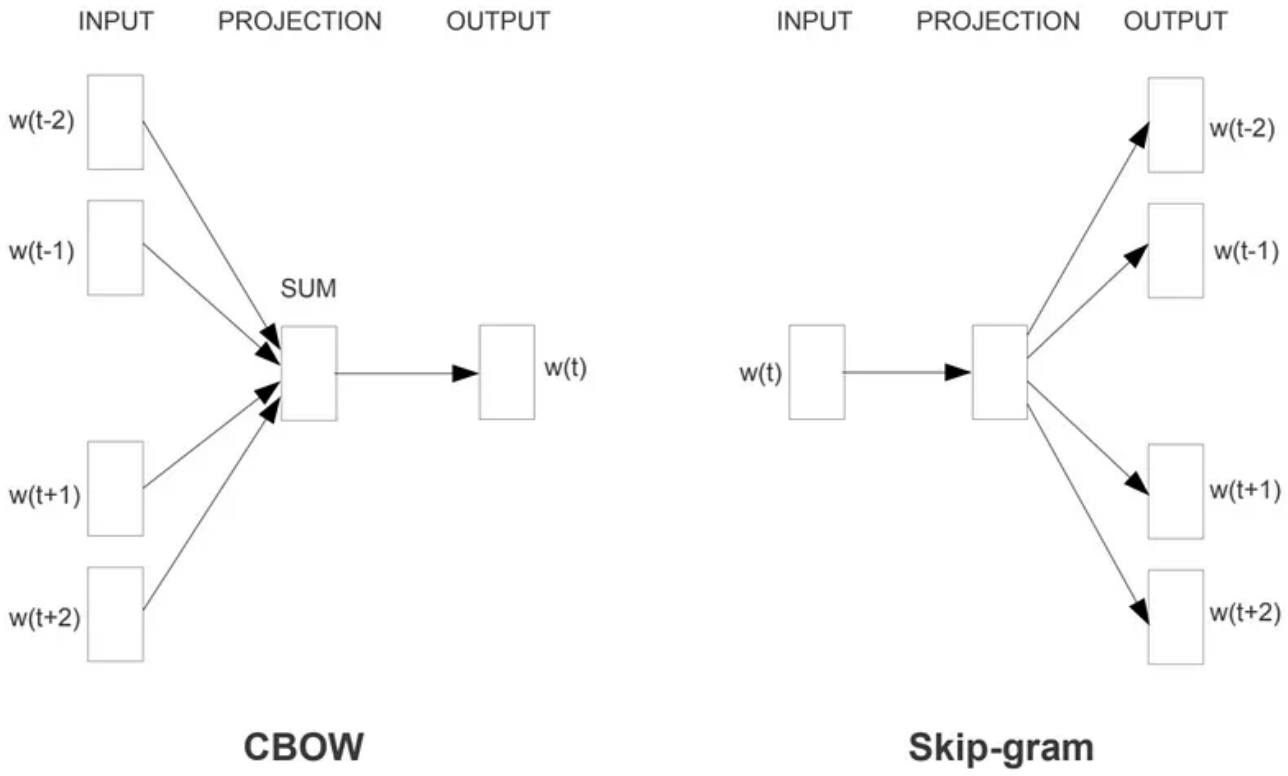
Continuous Bag-of-Words (CBOW):

- Predicts the current word from a window of surrounding context words.

- Emphasizes the collaborative influence of context words in predicting the target word.

Continuous Skip-Gram:

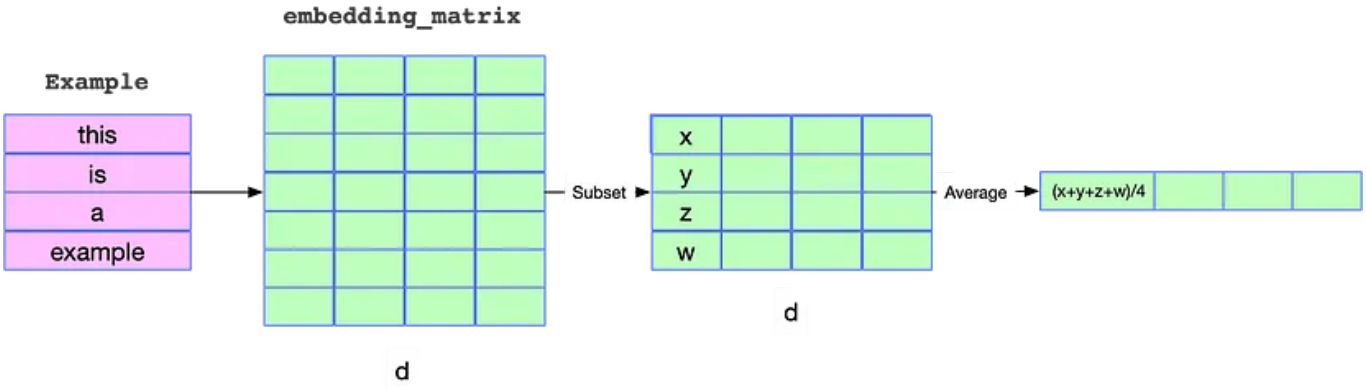
- Uses the current word to predict the surrounding window of context words.
- Focuses on the predictive power of the target word in generating the context words.



CBOW & Skip-gram

Word2Vec's dual model architectures offer versatility in capturing linguistic nuances, allowing practitioners to choose between CBOW and continuous skip-gram based on the specific requirements of their natural language

processing tasks. Understanding the interplay between these architectures enhances the application of Word2Vec in diverse contexts.



Word-2-Vec

- **GloVe (Global Vectors for Word Representation)**: GloVe's strength lies in its utilization of aggregated global word-word co-occurrence statistics from a corpus during training. The resulting representations not only encapsulate semantic relationships but also unveil intriguing linear substructures within the word vector space, adding depth to the understanding of word embeddings.
- **FastText**: In contrast to GloVe, FastText takes a novel approach by treating each word as composed of character n-grams. This distinctive feature allows FastText to not only learn rare words but also handle out-of-vocabulary words with finesse. The emphasis on character-level embeddings empowers FastText to capture morphological nuances, offering a more comprehensive representation of the vocabulary.

1.2. Context-Dependent Embeddings

Context-Dependent methods learn different embeddings for the same word based on its context.

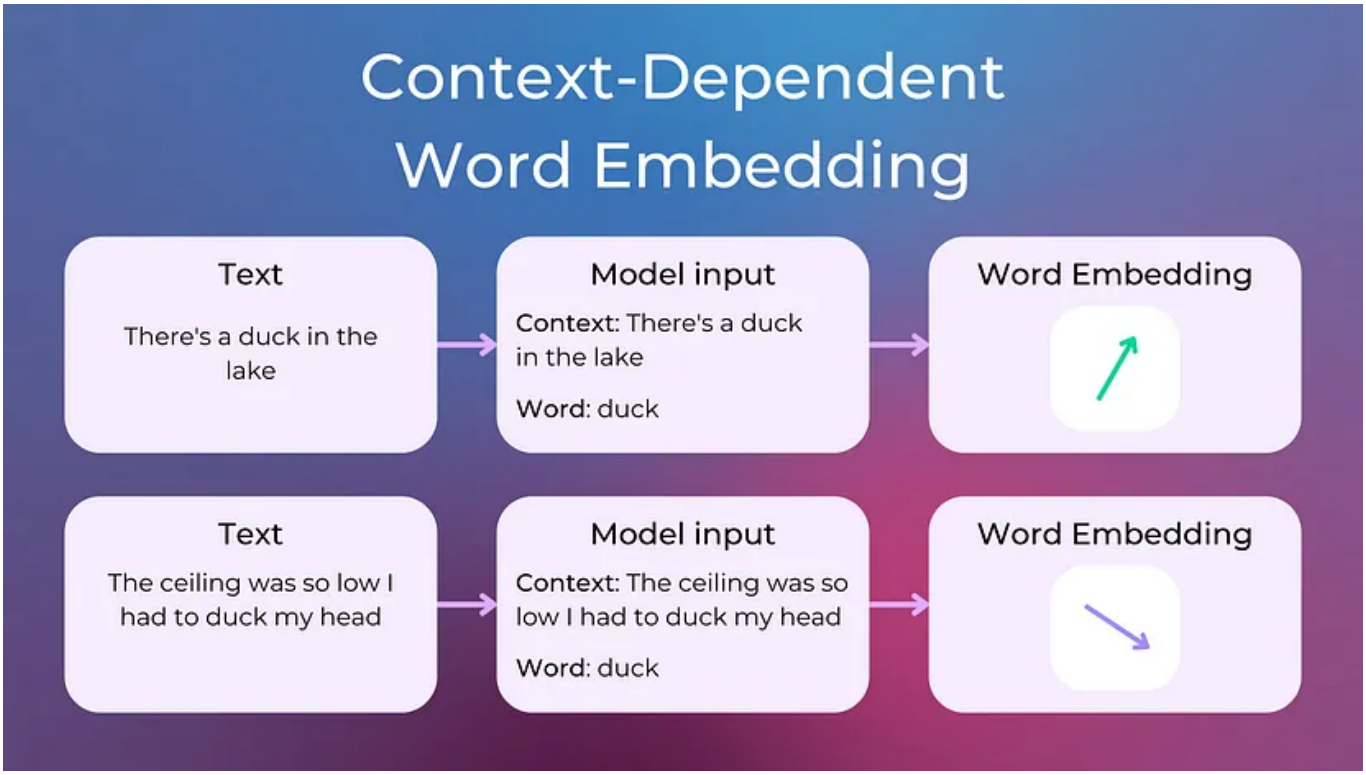


Image by NLP Planet

RNN based

- **ELMO (Embeddings from Language Model):** Learns contextualized word representations based on a neural language model with a character-based encoding layer and two BiLSTM layers.
- **CoVe (Contextualized Word Vectors):** Uses a deep LSTM encoder from an attentional sequence-to-sequence model trained for machine translation to contextualize word vectors.

Transformer-based

- **BERT (Bidirectional Encoder Representations from Transformers):** Transformer-based language representation model trained on a large cross-domain corpus. Applies a masked language model to predict words that are randomly masked in a sequence, and this is followed by a next-

sentence-prediction task for learning the associations between sentences.

- **XLM (Cross-lingual Language Model)**: It's a transformer pre-trained using next token prediction, a BERT-like masked language modeling objective, and a translation objective.
- **RoBERTa (Robustly Optimized BERT Pretraining Approach)**: It builds on BERT and modifies key hyperparameters, removing the next-sentence pretraining objective and training with much larger mini-batches and learning rates.
- **ALBERT (A Lite BERT for Self-supervised Learning of Language Representations)**: It presents parameter-reduction techniques to lower memory consumption and increase the training speed of BERT.

2. BERT

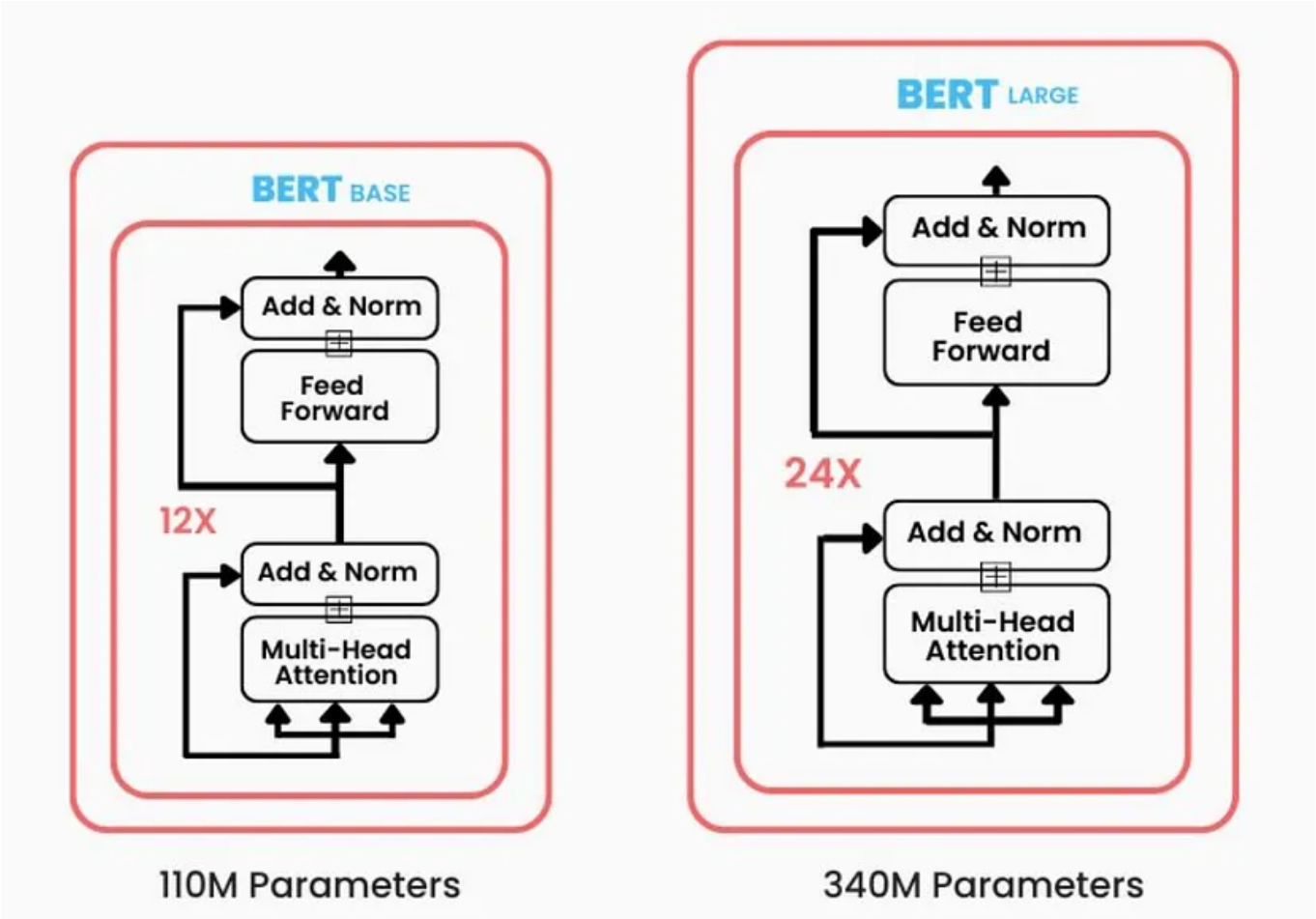
BERT (Bidirectional Encoder Representations from Transformers), a powerhouse in natural language processing developed by Google AI, has reshaped the landscape of language models. This exploration delves into the pre-training methodology and the intricacies of its bi-directional architecture.

- ***Pre-training:*** BERT is pre-trained for two unsupervised tasks — M*asked Language Modeling(MLM)* and *Next Sentence Prediction(NSP).* The *MLM* randomly masks some(15%) of the tokens from the input at random, and the objective is to predict the original *vocabulary ID* of the masked word based only on its context.
- In addition to the masked language model, BERT uses a *NSP* task that jointly pre-trains text-pair representations. Many important downstream tasks such as Question Answering(QA) and Natural Language

Inference(NLI) are based on understanding the relationship between two sentences, which is not directly captured by language modeling.

- ***Pre-training data:*** The pre-training procedure largely follows the existing literature on language model pre-training.For the pre-training corpus we use the Books Corpus (800M words) and English Wikipedia (2,500M words).
- ***Bi-directional*:** Unlike left-to-right language model pre-training, the MLM objective enables the representation to fuse the left and the right context, which allows us to pre-train a deep bidirectional Transformer.

The architecture of BERT is structured with multiple encoder layers, each applying self-attention to the input and passing it to the subsequent layer. Even the smallest variant, BERT BASE, boasts 12 encoder layers, a feed-forward neural network block with 768 hidden units, and 12 attention heads.



BERT Structures

2.1. Input representations

BERT takes as input sequences that are composed of sentences or pairs of sentences (e.g.,<Question, Answer>) in one token sequence for question-answering tasks.

Input sequences are prepared before being fed to the model using *WordPiece Tokenizer* with a 30k vocabulary size token. It works by splitting a word into several *subwords (Tokens)*.

Special tokens are:

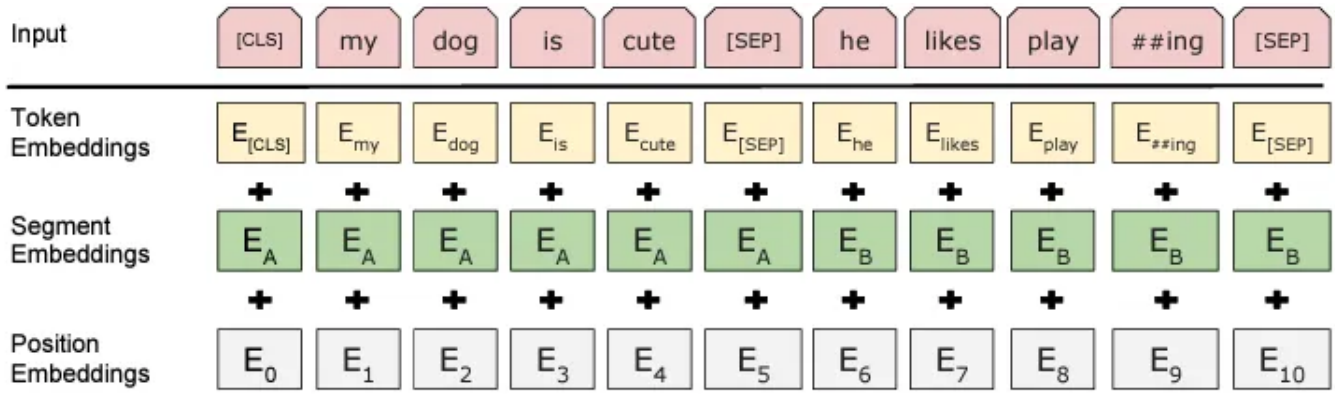
- **[CLS]** used as the first token of each sequence. The final hidden state corresponding to this token is used as the aggregate sequence

representation for classification tasks.

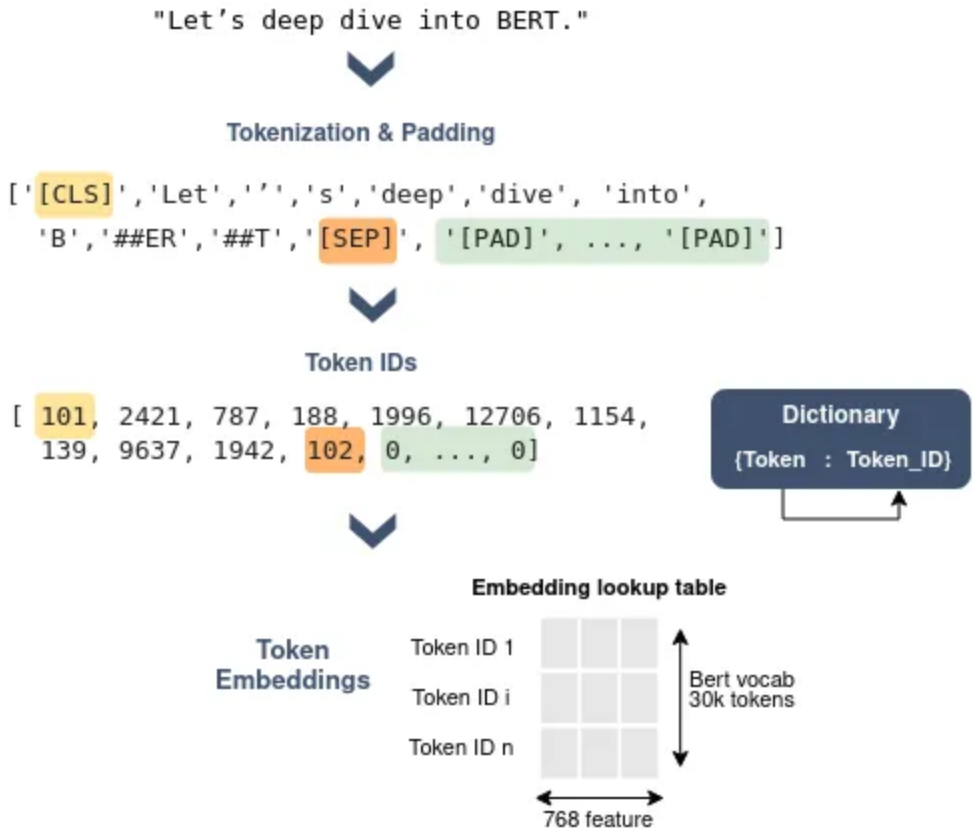
- **[SEP]** Sentence pairs are packed together into a single sequence. We differentiate the sentences in two ways. First, we separate them with a special token([SEP]). Second, we add a learned embedding to every token indicating whether it belongs to sentence A or sentence B.
- **[PAD]** used to represent paddings in the input sentences (empty tokens). The model expects fixed-length sentences as input. A maximum length is thus fixed depending on the dataset. Shorter sentences are padded, whereas longer sentences are truncated. To explicitly differentiate between real tokens and [PAD] tokens, we use an attention mask.

Segmentation embedding is introduced to indicate if a given token belongs to the first or second sentence. *Positional embedding* indicates the position of tokens in a sentence. By contrast to the original Transformer, BERT learns positional embeddings from absolute ordinal position, instead of using trigonometric functions.

For a given token, its input representation is constructed by summing the corresponding token, segment, and position embeddings.



BERT input representation. The input embeddings are the sum of the token embeddings, the segmentation embeddings and the position embeddings.



Sentence encoding using BERT

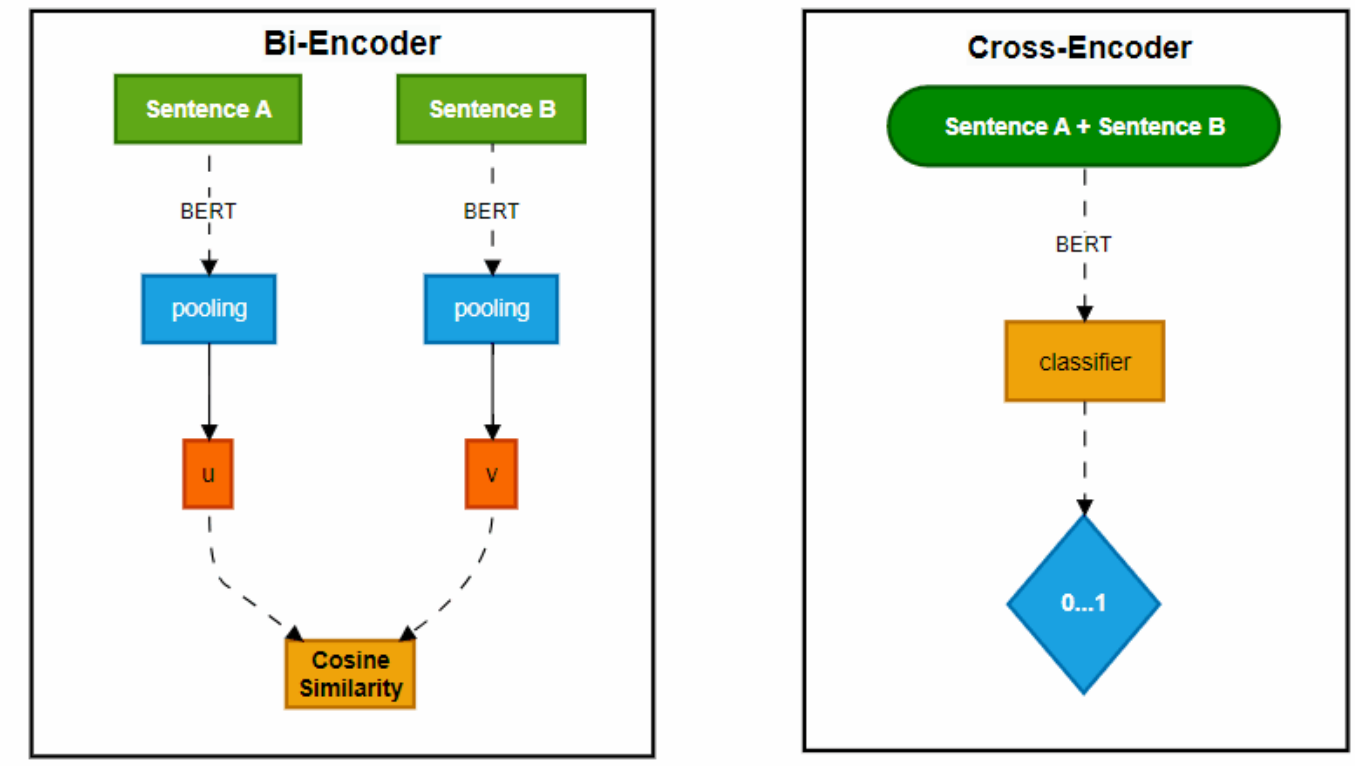
To get the token embedding, an embedding *lookup table* is used at the embedding layer (as illustrated in Figure above), where rows represent all possible token IDs in the vocabulary (30k rows for instance) and columns represent the token embedding size.

2.2. Why Sentence BERT (S-BERT) Over BERT?

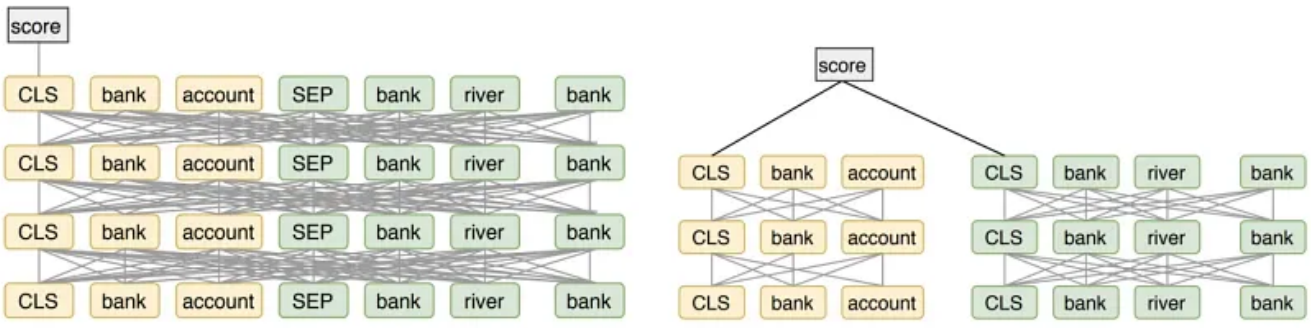
So far, so good, but these transformer models had one issue when building sentence vectors: Transformers work using word or *token*-level embeddings, *not* sentence-level embeddings.

Before sentence transformers, the approach to calculating *accurate* sentence similarity with BERT was to use a **cross-encoder structure.** This meant that we would pass two sentences to BERT, add a classification head to the top of BERT — and use this to output a similarity score.

The BERT cross-encoder architecture consists of a BERT model which consume sentences A and B. Both are processed in the same sequence, separated by a **[SEP]** token. All of this is followed by a feedforward NN classifier that outputs a similarity score.



Siamese (bi-encoder) architecture is shown on the left, and the Non-Siamese (cross-encoder) architecture is on the right. The principal difference is that on the left the model accepts both inputs at the same time. On the right, the model accepts both inputs in parallel, so both outputs are not dependent on each other.



Cross-encoder(left) and bi-encoder(right)

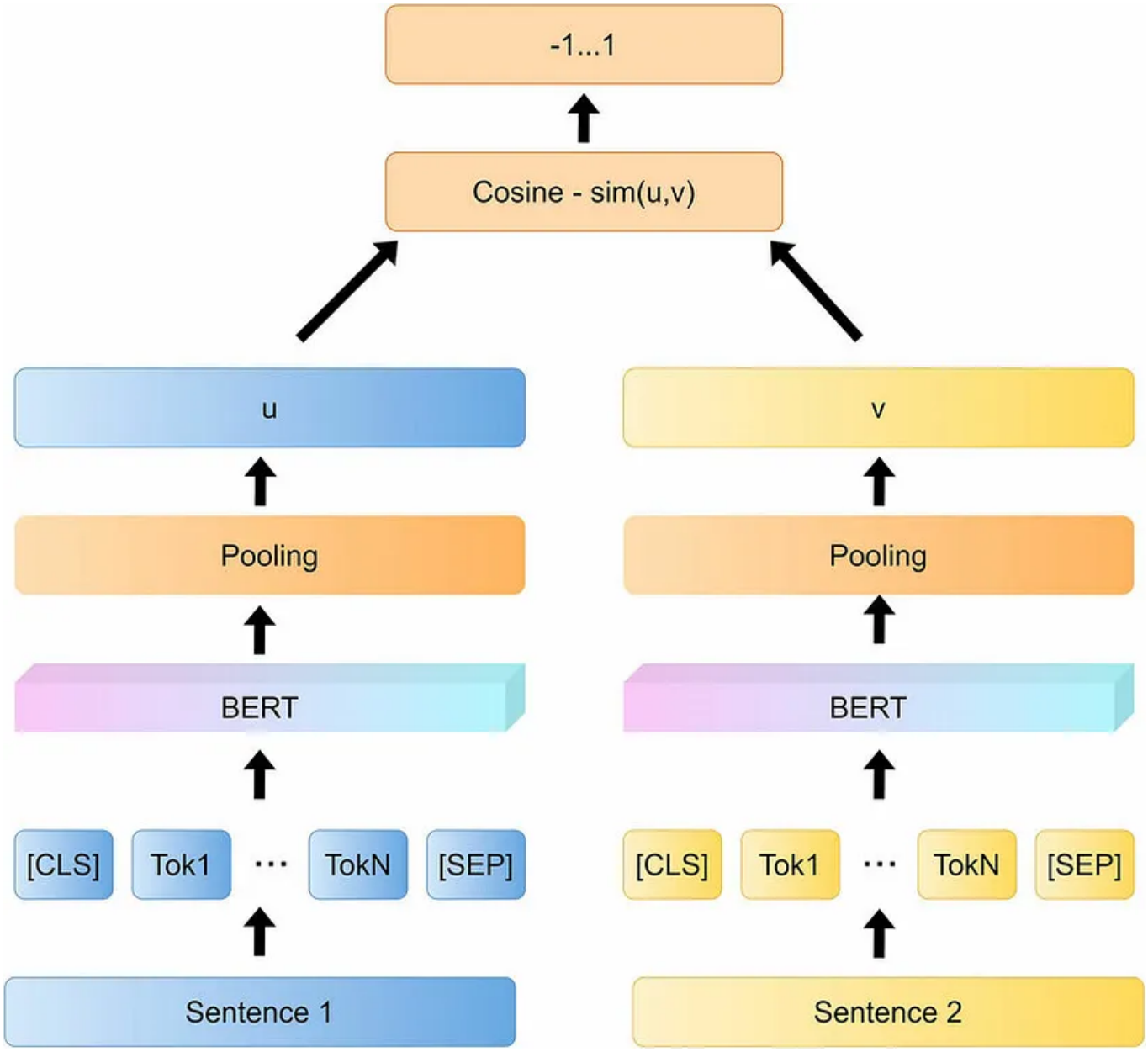
The cross-encoder network does produce very accurate similarity scores (better than SBERT), but it's not scalable. If we wanted to perform a similarity search through a small 100K sentence dataset, we would need to complete the cross-encoder inference computation 100K times.

To cluster sentences, we would need to compare all sentences in our 100K dataset, resulting in just under 500M comparisons — this is simply not realistic.

Ideally, we need to pre-compute sentence vectors that can be stored and then used whenever required. If these vector representations are good, all we need to do is calculate the cosine similarity between each. With the original BERT (and other transformers), we can build a sentence embedding by averaging the values across all token embeddings output by BERT (if we input 512 tokens, we output 512 embeddings). [Approach — 1]

Alternatively, we can use the output of the first [CLS] token (a BERT-specific token whose output embedding is used in classification tasks). [Approach — 2]

Using one of these two approaches gives us our sentence embeddings that can be stored and compared much faster, shifting search times from 65 hours to around 5 seconds. However, the accuracy is not good, and is worse than using averaged GloVe embeddings (which were developed in 2014)



Sentence BERT (Bi-Encoder)

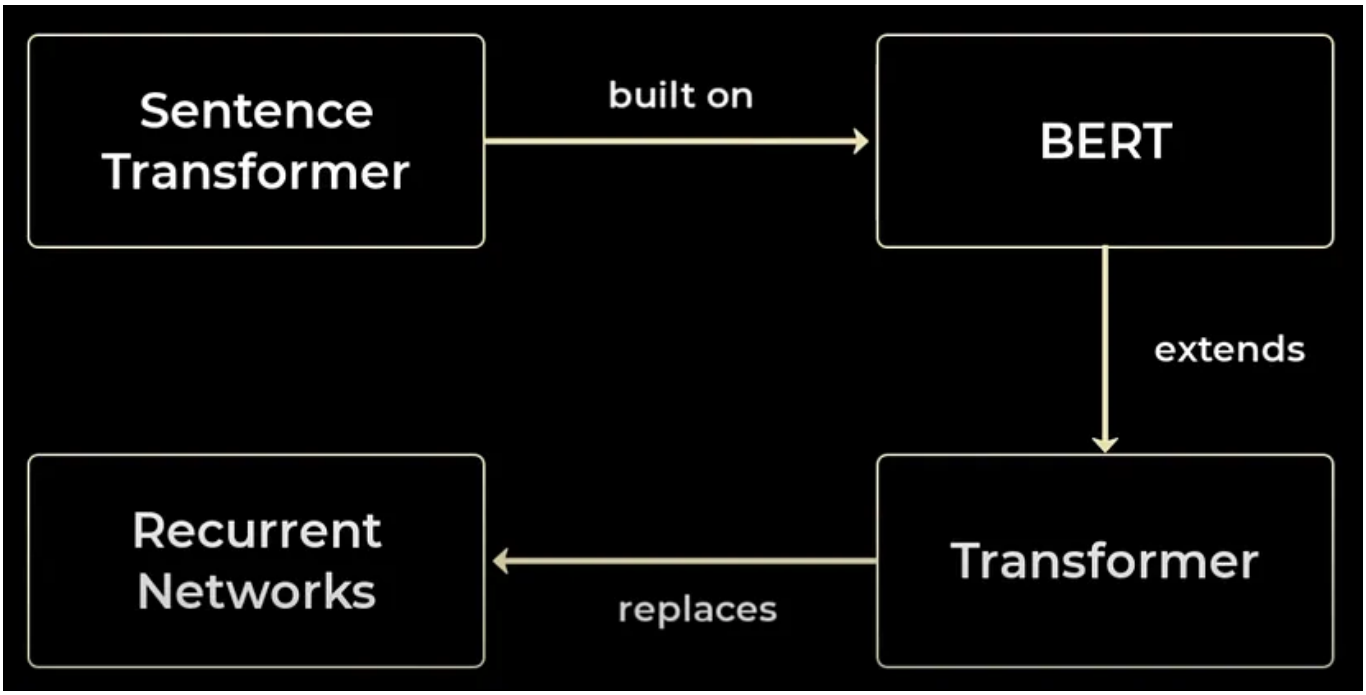
Thus, finding the most similar sentence pair from 10K sentences took 65 hours with BERT. With SBERT, embeddings are created in ~5 seconds and compared with cosine similarity in ~0.01 seconds.

Since the SBERT paper, many more sentence transformer models have been built using similar concepts that went into training the original SBERT. They're all trained on many similar and dissimilar sentence pairs.

Using a loss function such as **softmax loss**, **multiple negatives ranking loss**, or **MSE margin loss** these models are optimized to produce similar embeddings for similar sentences, and dissimilar embeddings otherwise.

> Deriving independent sentence embeddings is one of the main problems of BERT. To alleviate this aspect, SBERT was developed.

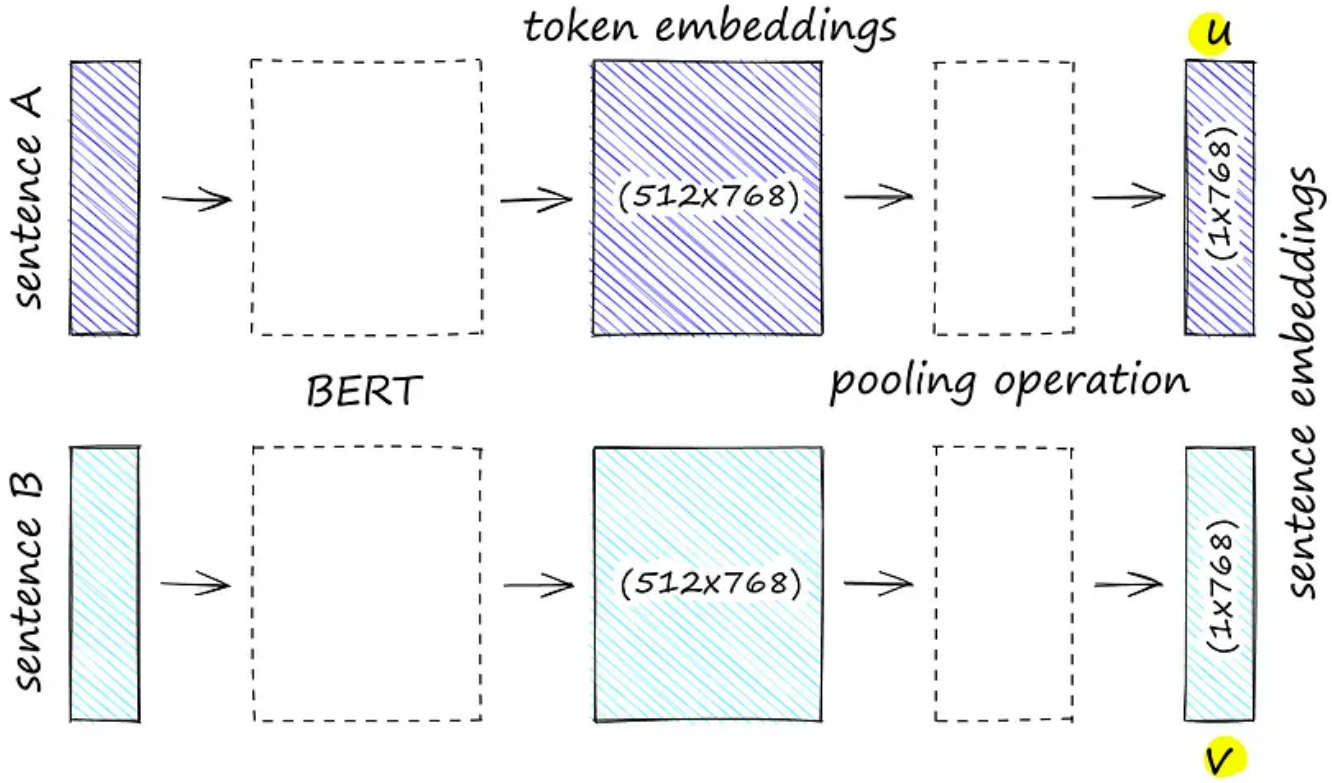
3. Sentence Transformers



Sentence Transformer

We explained the cross-encoder architecture for sentence similarity with BERT. SBERT is similar but drops the final classification head, and processes one sentence at a time. SBERT then uses mean pooling on the final output layer to produce a sentence embedding.

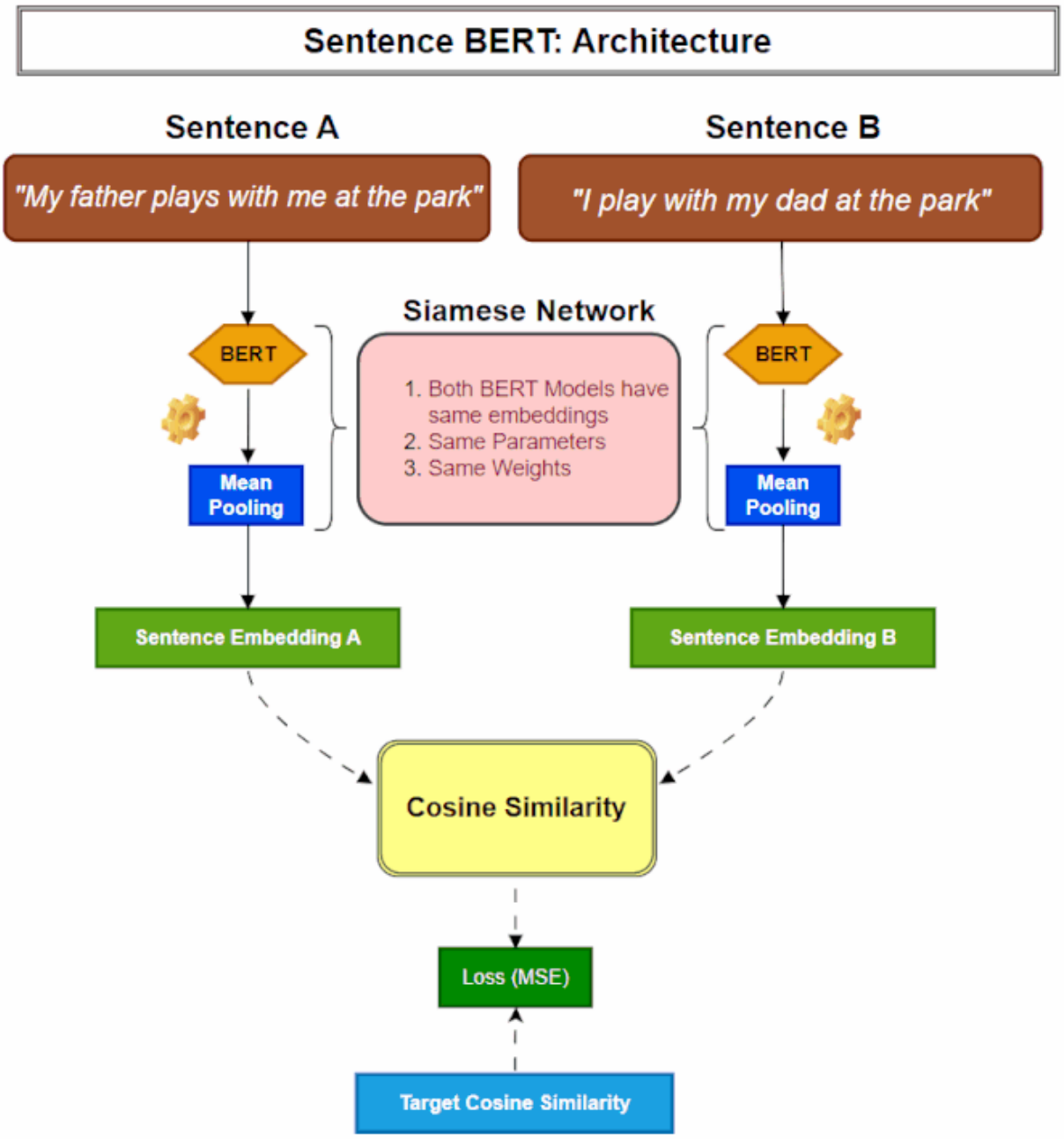
Unlike BERT, SBERT is fine-tuned on sentence pairs using a *siamese* architecture. We can think of this as having two identical BERTs in parallel that share the exact same network weights.



An SBERT model applied to a sentence pair sentence A and sentence B. Note that the BERT model outputs token embeddings (consisting of 512 768-dimensional vectors). We then compress that data into a single 768-dimensional sentence vector using a pooling function.

In reality, we are using a single BERT model. However, because we process sentence A followed by sentence B as *pairs* during training, it is easier to think of this as two models with tied weights.

3.1. Siamese BERT Pre-Training



Siamese Architecture

There are different approaches to training sentence transformers. We will describe the original process featured most prominently in the original SBERT paper that optimizes on *softmax-loss*.

The softmax-loss approach used the *'siamese'* architecture fine-tuned on the Stanford Natural Language Inference (SNLI) and Multi-Genre NLI (MNLI) corpora.

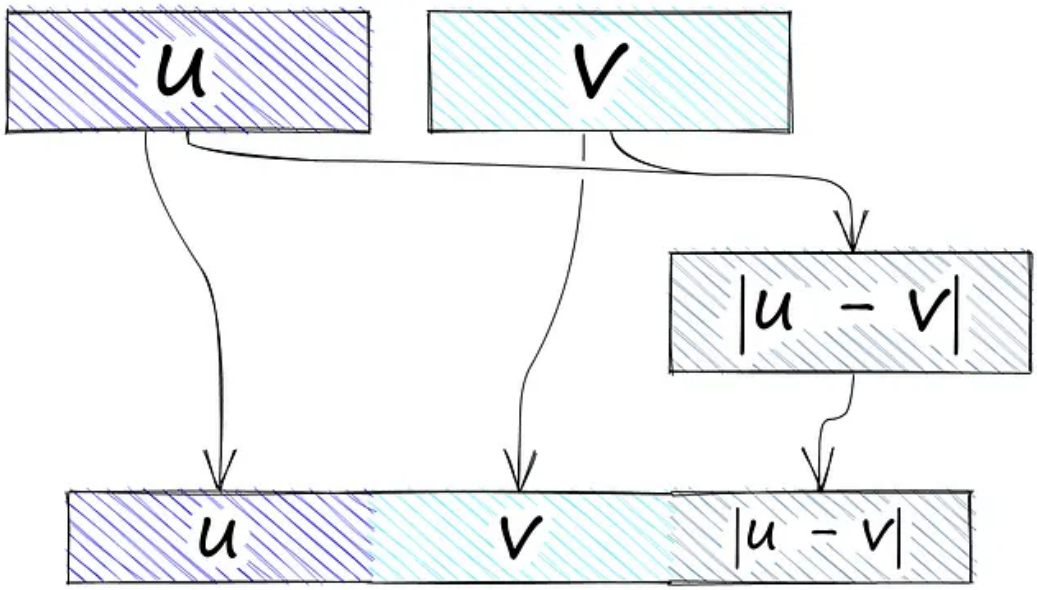
SNLI contains 570K sentence pairs, and MNLI contains 430K. The pairs in both corpora include a **premise** and a **hypothesis**. Each pair is assigned one of three labels:

- **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.

Given this data, we feed sentence A (let's say the **premise**) into siamese BERT A and sentence B (**hypothesis**) into siamese BERT B.

The siamese BERT outputs our pooled sentence embeddings. There were the results of *three* different pooling methods in the SBERT paper. Those are ***mean***, ***max***, and ***[CLS]*-pooling**. The *mean*-pooling approach was best performing for both NLI and STSb datasets.

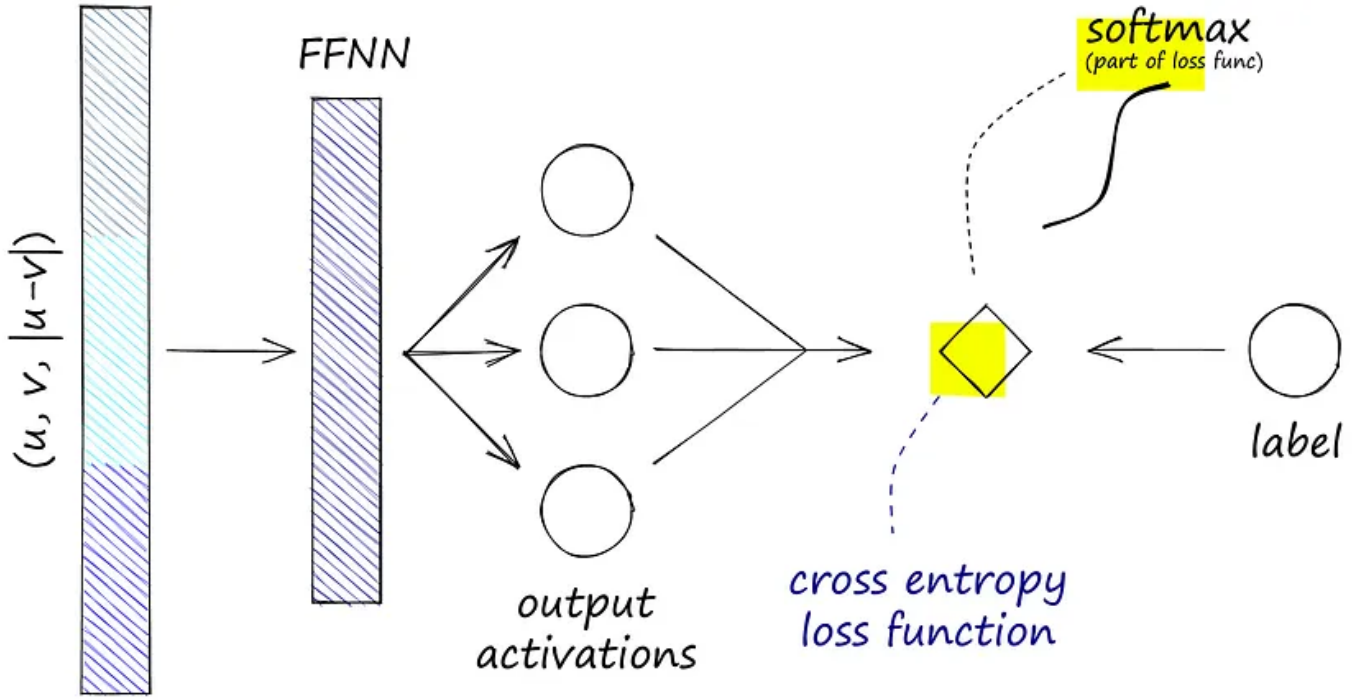
There are now two sentence embeddings. We will call embeddings A as **u** and embeddings B as **v**. The next step is to concatenate **u** and **v**. Again, several concatenation approaches were tested, but the highest performing was a **(u, v, |u-v|)** operation:



We concatenate the embeddings u, v, and |u — v|.

|u-v| is calculated to give us the element-wise difference between the two vectors. Alongside the original two embeddings (**u** and **v**), these are all fed into a feedforward neural net (FFNN) that has *three* outputs.

These three outputs align to our NLI similarity labels **0**, **1**, and **2**. We need to calculate the softmax from our FFNN, which is done within the cross-entropy loss function. The softmax and labels are used to optimize on this *'softmax-loss'*.



The operations were performed during training on two sentence embeddings, u and v. Note that softmax-loss refers cross-entropy loss (which contains a softmax function by default).

The operations were performed during training on two sentence embeddings, **u** and **v**. Note that *softmax-loss* refers cross-entropy loss (which contains a softmax function by default).

This results in our pooled sentence embeddings for similar sentences (label **0**) becoming *more similar*, and embeddings for dissimilar sentences (label **2**) becoming *less similar*.

Remember we are using *siamese* BERTs **not** *dual* BERTs. Meaning we don't use two independent BERT models but a single BERT that processes sentence A followed by sentence B.

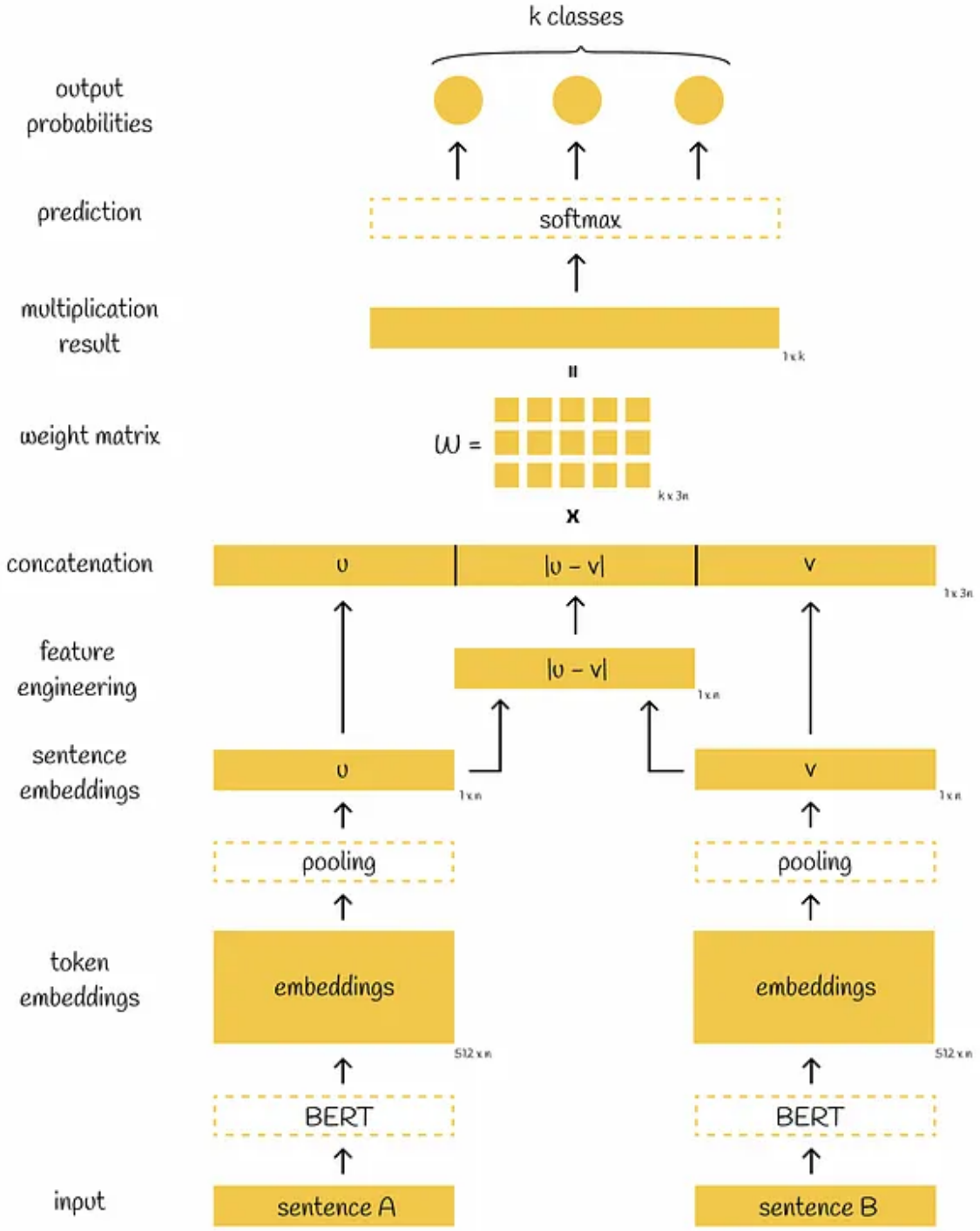
This means that when we optimize the model weights, they are pushed in a direction that allows the model to output more similar vectors where we see an *entailment* label and more dissimilar vectors where we see a *contradiction* label.

4. SBERT Objective Functions

By using these two vectors **u** and **v**, three approaches for optimizing different objectives are discussed below.

4.1. Classification

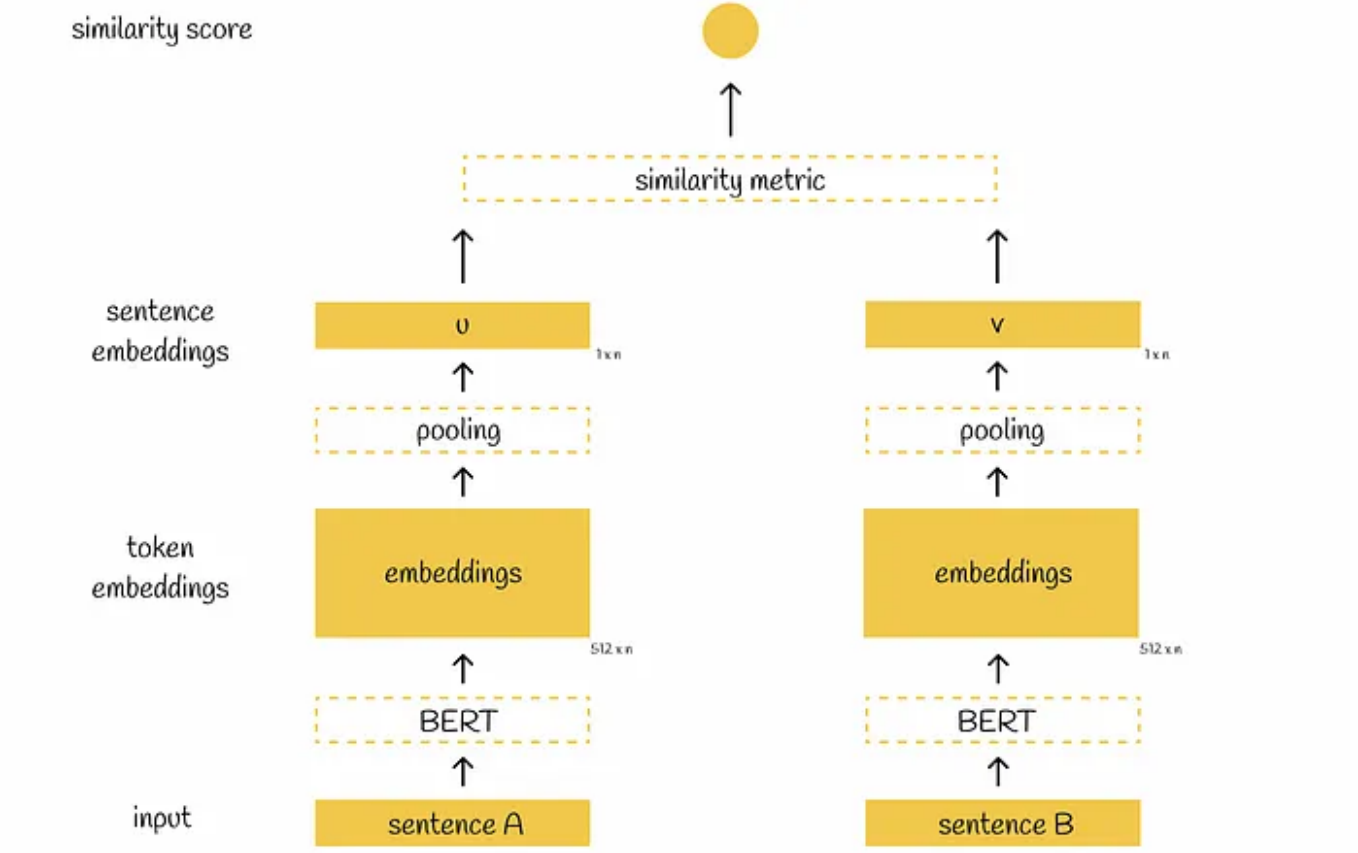
The three vectors u, v and $|u\text{-}v|$ are concatenated, multiplied by a trainable weight matrix W and the multiplication result is fed into the softmax classifier which outputs normalized probabilities of sentences corresponding to different classes. The cross-entropy loss function is used to update the weights of the model.



SBERT architecture for classification objective. Parameter n stands for the dimensionality of embeddings (768 by default for BERT base) while k designates the number of labels.

4.2. Regression

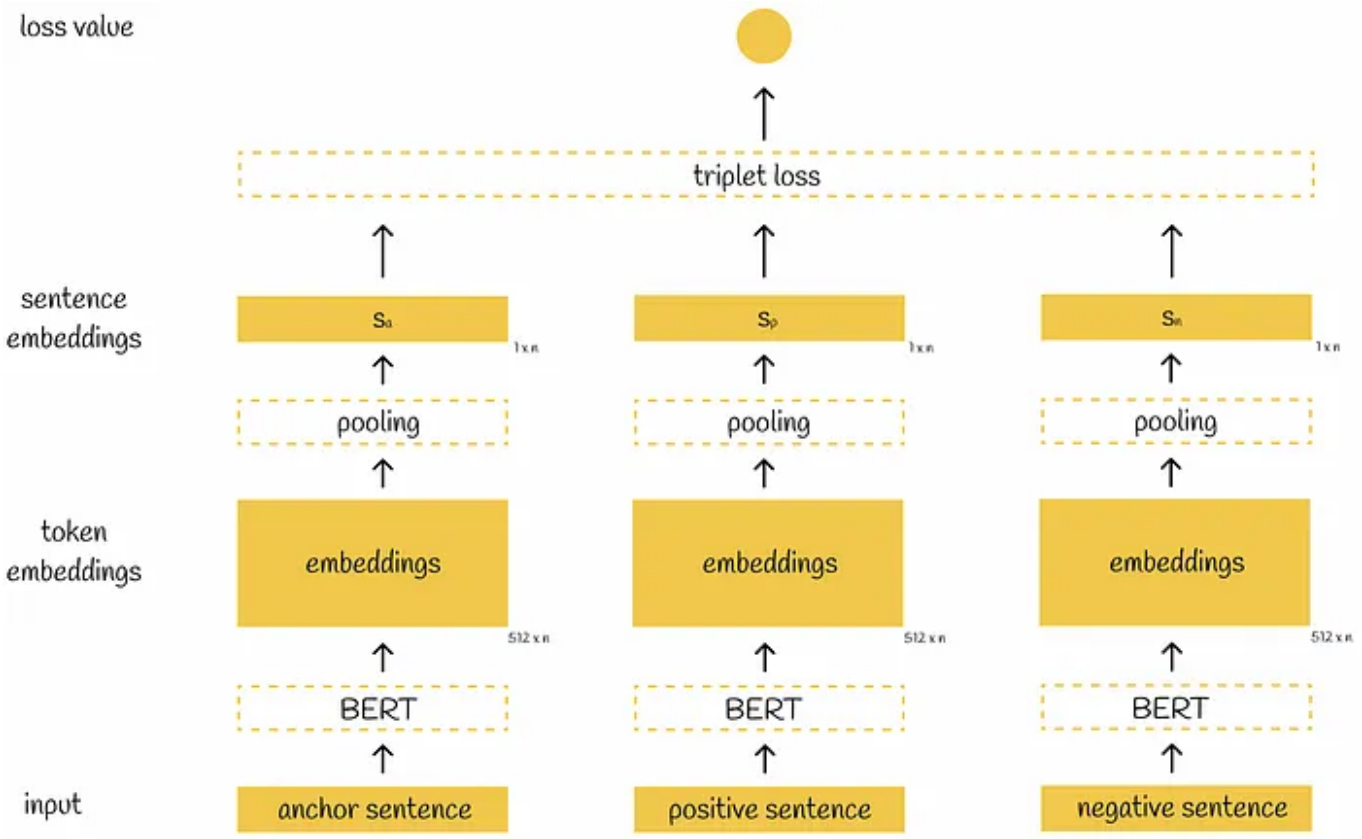
In this formulation, after getting vectors **u** and **v**, the similarity score between them is directly computed by a chosen similarity metric. The predicted similarity score is compared with the true value and the model is updated by using the MSE loss function.



SBERT architecture for regression objective. Parameter n stands for the dimensionality of embeddings (768 by default for BERT base).

4.3. Triplet Loss

The triplet objective introduces a triplet loss which is calculated on three sentences usually named *anchor*, *positive* and *negative*. It is assumed that *anchor* and *positive* sentences are very close to each other while *anchor* and *negative* are very different. During the training process, the model evaluates how closer the pair *(anchor, positive)* is, compared to the pair *(anchor, negative)*.



Triplet SBERT architecture

For now, let's look at how we can initialize and use these sentence-transformer models.

5. Hands-On with Sentence Transformers

The fastest and easiest way to begin working with sentence transformers is through the **sentence-transformers** library created by the creators of SBERT. We can install it with **pip**.

```
!pip install sentence-transformers
```

We will start with the original SBERT model **bert-base-nli-mean-tokens**. First, we download and initialize the model.

```
from sentence_transformers import SentenceTransformer

model = SentenceTransformer('bert-base-nli-mean-tokens')

model
```

Output:

```
SentenceTransformer(
  (0): Transformer({'max_seq_length': 128, 'do_lower_case': False}) with Transfo
  (1): Pooling({'word_embedding_dimension': 768, 'pooling_mode_cls_token': False
)
```

The **output** we can see here is the **SentenceTransformer** object which contains *three* components:

- The **transformer** itself, here we can see the max sequence length of **128** tokens and whether to lowercase any input (in this case, the model does *not*). We can also see the model class, **BertModel**.
- The **pooling** operation, here we can see that we are producing a **768**-dimensional sentence embedding. We are doing this using the *mean pooling* method.

Once we have the model, building sentence embeddings is quickly done using the **encode** method.

```
sentences = [
    "the fifty mannequin heads floating in the pool kind of freaked them out",
    "she swore she just saw her sushi move",
    "he embraced his new life as an eggplant",
    "my dentist tells me that chewing bricks is very bad for your teeth",
    "the dental specialist recommended an immediate stop to flossing with constr
]

embeddings = model.encode(sentences)

embeddings.shape
```

Output:

```
(5, 768)
```

6. Which Embedding Model to Choose?

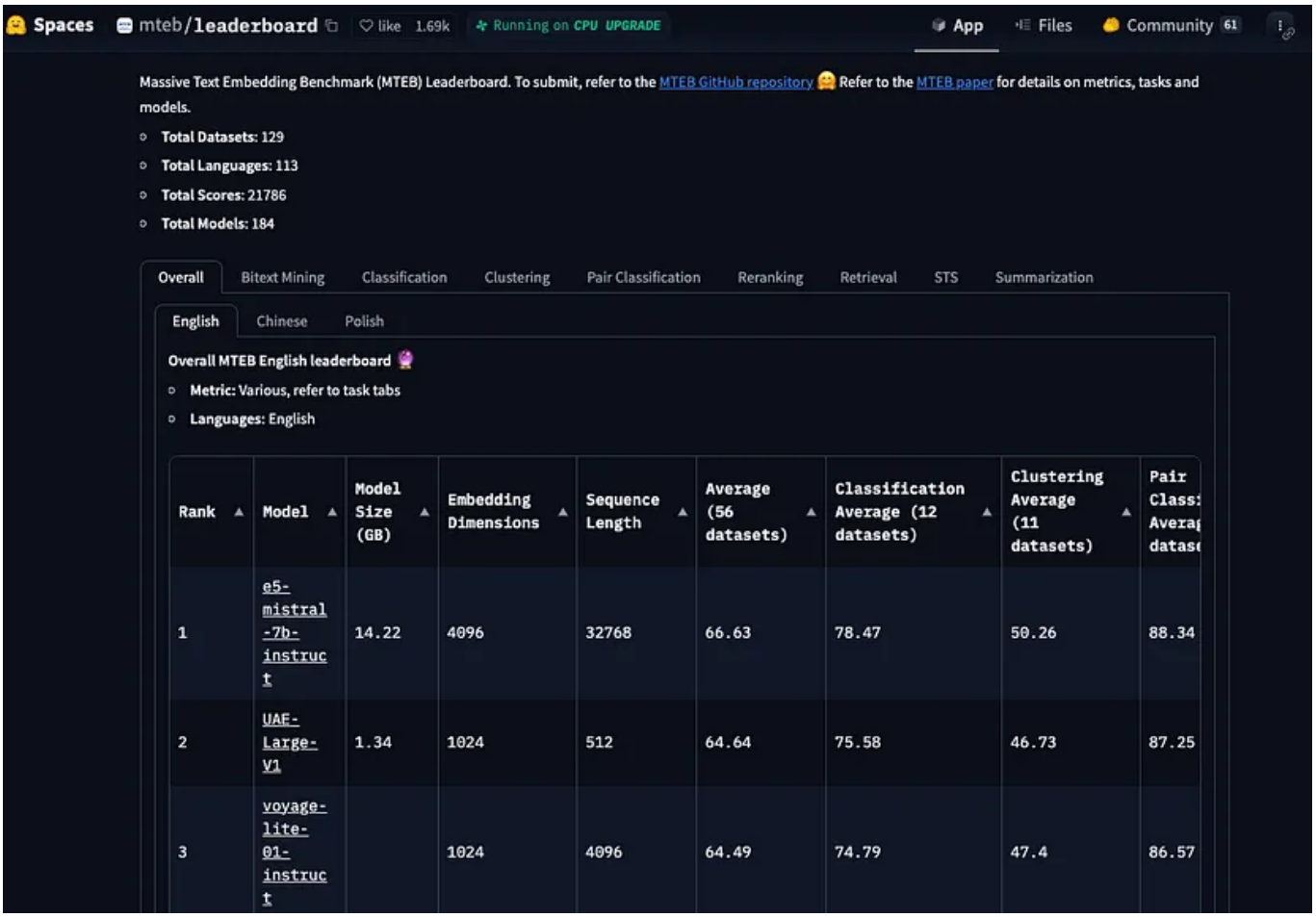
However, as we will soon discover, the majority of the currently employed embedding models belong to the transformer category. These models are offered by various providers, with some being open source and others proprietary, each tailored to specific objectives:

- Some are best suited for coding tasks.
- Others are designed specifically for the English language.
- And there are also embedding models that excel in handling multilingual datasets.

The simplest approach is to leverage existing academic benchmarks. Nevertheless, it's crucial to bear in mind that these benchmarks may not comprehensively mirror real-world usage of retrieval systems in AI applications.

Alternatively, you can conduct testing with various embedding models and compile a final evaluation table to pinpoint the most suitable one for your specific use case. I highly recommend incorporating a re-ranker into this process as it can significantly enhance retriever performance, ultimately yielding the optimal results.

To simplify your decision-making process, Hugging Face offers the remarkable Massive Text Embedding Benchmark (MTEB) Leaderboard. This resource provides comprehensive information about all available embedding models and their respective scores on various metrics :



HuggingFace MTEB

If you opt for the second approach, there is an excellent Medium blog post available that shows how to utilise the Retrieval Evaluation module from LlamaIndex. This resource can help you efficiently assess and identify the optimal combination of embedding and reranker from an initial list of models.

Embedding	WithoutReranker		bge-reranker-base		bge-reranker-large		Cohere-Reranker	
	Hit Rate	MRR	Hit Rate	MRR	Hit Rate	MRR	Hit Rate	MRR
OpenAI	0.876404	0.718165	0.91573	0.832584	0.910112	0.855805	0.926966	0.86573
bge-large	0.752809	0.597191	0.859551	0.805243	0.865169	0.816011	0.876404	0.822753
llm-embedder	0.814607	0.587266	0.870787	0.80309	0.876404	0.824625	0.882022	0.830243
Cohere-v2	0.780899	0.570506	0.876404	0.798127	0.876404	0.825281	0.876404	0.815543
Cohere-v3	0.825843	0.624532	0.882022	0.806086	0.882022	0.834644	0.88764	0.836049
Voyage	0.831461	0.68736	0.926966	0.837172	0.91573	0.858614	0.91573	0.851217
JinaAI-Small	0.831461	0.614045	0.91573	0.843071	0.926966	0.857303	0.926966	0.868633
JinaAI-Base	0.848315	0.68221	0.938202	0.846348	0.938202	0.868539	0.932584	0.873689
Google-PaLM	0.865169	0.719476	0.910112	0.833708	0.910112	0.85309	0.910112	0.855712

I trust that you now feel better equipped to make an informed decision when selecting the most suitable embedding and reranking models for your RAG architecture!

Conclusion

The blog explores various embedding models for generating vector representations of text, including Bag of Words, TF-IDF, Word2Vec, GloVe, FastText, ELMO, BERT, and more. It delves into the architecture and pre-training of BERT, introduces Sentence BERT (SBERT) for efficient sentence embeddings, and provides a hands-on example using the sentence-transformers library. The conclusion emphasizes the challenge of choosing the right embedding model and suggests leveraging resources like the Hugging Face Massive Text Embedding Benchmark (MTEB) Leaderboard for evaluation.

Credits

In this blog post, we have compiled information from various sources, including research papers, technical blogs, official documentations, YouTube videos, and more. Each source has been appropriately credited beneath the corresponding images, with source links provided.

Below is a consolidated list of references:

1. https://jina.ai/news/the-1950-2024-text-embeddings-evolution-poster/
2. https://partee.io/2022/08/11/vector-embeddings/
3. https://www.nlplanet.org/course-practical-nlp/01-intro-to-nlp/11-text-as-vectors-embeddings
4. https://www.deeplearning.ai/resources/natural-language-processing/

5. https://www.mygreatlearning.com/blog/word-embedding/#sh4
6. https://mlwhiz.com/blog/2019/02/08/deeplearning_nlp_conventional_methods/
7. https://vitalflux.com/bert-vs-gpt-differences-real-life-examples/
8. https://d3mlabs.de/?p=1169
9. https://www.linkedin.com/pulse/why-does-bert-stand-out-sea-sentence-embedding-models-bhaskar-t-bi6wc%3FtrackingId=RKK3MNdP8pugx6iyhwJ2hw%253D%253D/?trackingId=RKK3MNdP8pugx6iyhwJ2hw%3D%3D
10. https://www.amazon.science/blog/improving-unsupervised-sentence-pair-comparison
11. https://www.researchgate.net/figure/Sentence-BERT-model_fig3_360530243
12. https://www.youtube.com/watch?app=desktop&v=O3xbVmpdJwU
13. https://www.pinecone.io/learn/series/nlp/sentence-embeddings/
14. https://towardsdatascience.com/sbert-deb3d4aef8a4
15. https://huggingface.co/spaces/mteb/leaderboard

Thank you for reading!

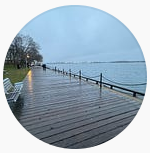
If this guide has enhanced your understanding of Python and Machine Learning:

- Please show your support with a clap 👏 or several claps!
- Your claps help me create more valuable content for our vibrant Python or ML community.

- Feel free to share this guide with fellow Python or AI / ML enthusiasts.
- Your feedback is invaluable — it inspires and guides my future posts.

Connect with me!

Vipra

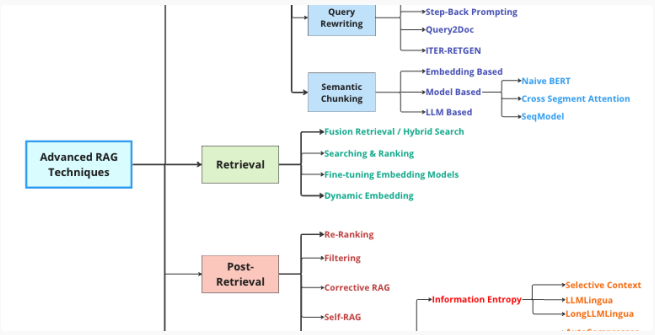


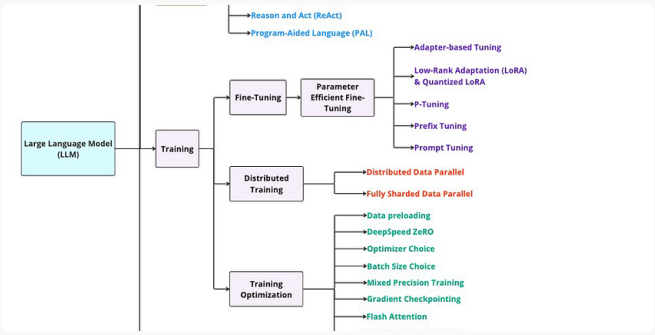
Written by Vipra Singh

1K Followers

Follow

More from Vipra Singh





Vipra Singh

Building LLM Applications: Advanced RAG (Part 10)

Learn Large Language Models (LLM) through the lens of a Retrieval Augmented...

✨ · 48 min read · Apr 27, 2024

384 2

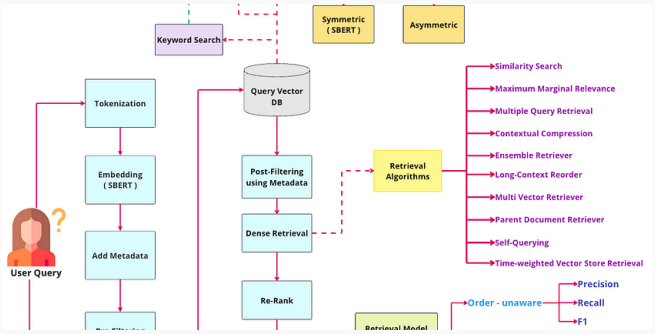
Vipra Singh

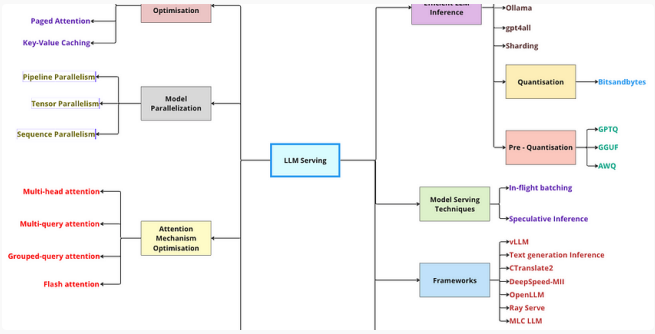
Building LLM Applications: Large Language Models (Part 6)

Learn Large Language Models (LLM) through the lens of a Retrieval Augmented...

38 min read · Feb 10, 2024

390 1





Vipra Singh

Building LLM Applications: Search & Retrieval (Part 5)

Learn Large Language Models (LLM) through the lens of a Retrieval Augmented...

26 min read · Jan 27, 2024

405 1

Vipra Singh

Building LLM Applications: Serving LLMs (Part 9)

Learn Large Language Models (LLM) through the lens of a Retrieval Augmented...

✨ · 50 min read · Apr 17, 2024

546 3

See all from Vipra Singh

Recommended from Medium

Sagar Dubey

Sentence Transformers can be your answer to resource hungry LLMs

There are both cost and latency constraints around using LLMs. What if you can get...

9 min read · Dec 27, 2023

3

Paul Iusztin in Decoding ML

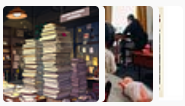
The 4 Advanced RAG Algorithms You Must Know to Implement

Implement from scratch 4 advanced RAG methods to optimize your retrieval and post-...

15 min read · May 4, 2024

1.4K 10

Lists



Staff Picks

656 stories · 1020 saves

Stories to Help You Level-Up at Work

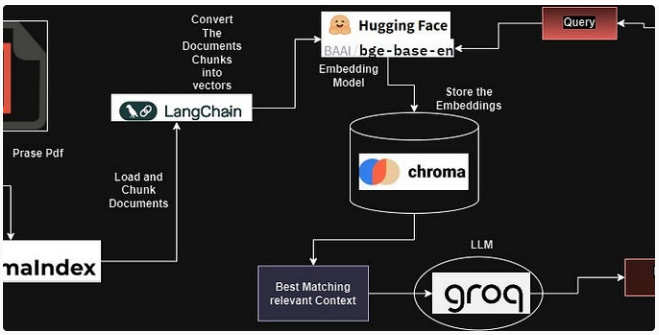
19 stories · 640 saves

Self-Improvement 101

20 stories · 1981 saves

Productivity 101

20 stories · 1791 saves



Plaban Nayak in The AI Forum

RAG on Complex PDF using LlamaParse, Langchain and Groq

Retrieval-Augmented Generation (RAG) is a new approach that leverages Large Languag...

13 min read · Apr 7, 2024

678 9

Lars Wiik

Best Embedding Model 🌟 — OpenAI / Cohere / Google / E5 /...

An In-depth Comparison of Multilingual Embedding Models

12 min read · Apr 7, 2024

478 3

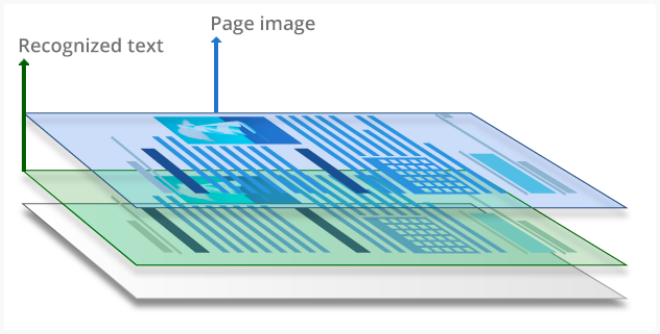
Mahesh

How to Productionize Large Language Models (LLMs)

Understand LLMOps, architectural patterns, how to evaluate, fine tune & deploy...

94 min read · Mar 27, 2024

186 2



Sasha Korovkina in Dev Genius

Building a High Precision Financial PDF Extraction Tool. Part 1.

Parsing Text from PDF Files

10 min read · Apr 29, 2024

209