LLM Embeddings Explained: A Visual and Intuitive Guide

The embedding atlas of 50 random words and their closest tokens in the embedding space of `deepseek-ai/DeepSeek-R1-Distill-Qwen-1.5B`.

AUTHORS

Hesam Sheikh Hassani

AFFILIATION

University of Bologna

PUBLISHED

March 27, 2025

Embeddings are the semantic backbone of LLMs, the gate at which raw text is transformed into vectors of numbers that are understandable by the model. When you prompt an LLM to help you debug your code, your words and tokens are transformed into a high-dimensional vector space where semantic relationships become mathematical relationships.

Reading time: 12-15 minutes.

In this article we go through the fundamentals of embeddings. We will cover what embeddings are, how they evolved over time from statistical methods to modern techniques, check out how they're implemented in practice, look at some of the most important embedding techniques, and how the embeddings of an LLM (DeepSeek-R1-Distill-Qwen-1.5B) look like as a graph representation.

This article includes interactive visualization and hands-on code examples. It also avoids verbosity and focuses on the core concepts for a fast-paced read to get straight to the point. The full code is available on my LLM Mechanics GitHub repository.

To contribute to the article, point out any mistakes, or suggest improvements, please check outCommunity.

What Are Embeddings?

Processing text for NLP tasks requires a numeric representation of each word. Most embedding methods come down to turning a word or token into a vector. What makes embedding techniques different from each other, is how they approach this word → vector conversion.

Embedding is not just for text, they can be applied to images, audio, or even graph data. In a general sense, embedding is the process of converting data [of any type] into vectors. Of course, the embedding methods of each modality is different and unique. In this article, when we talk about "embeddings", we are referring to the text embeddings.

You might have heard embeddings in the context of large language models, but embeddings actually have a much longer history. Here is an overview of various embedding techniques:

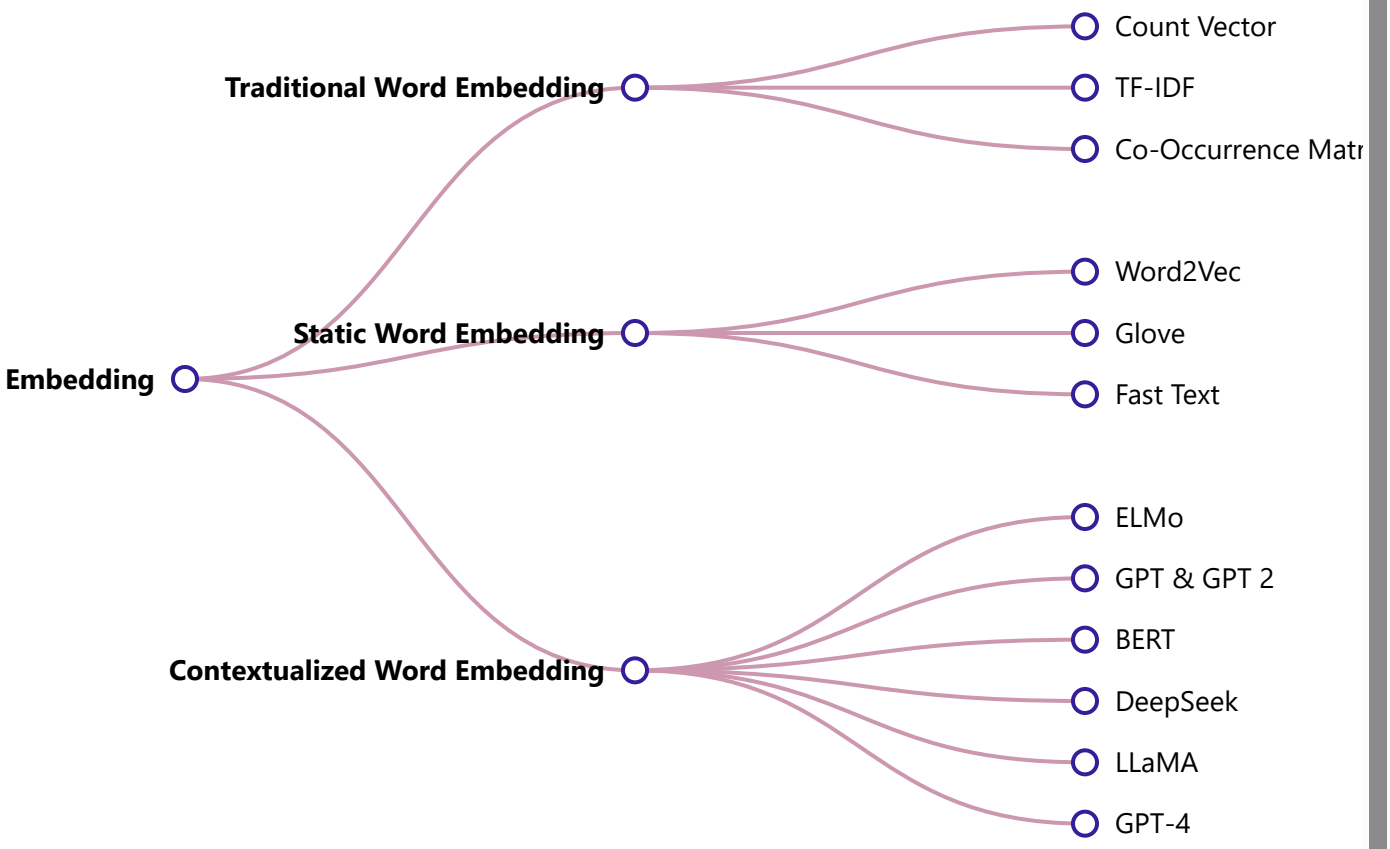


Figure 1: Overview of different word embedding techniques.

When reading about embeddings you may come across static vs. dynamic (or contextualized) embeddings. It's important to distinguish between token embeddings—which are the fixed vectors assigned to input tokens at the very beginning of an LLM—and the contextual representations produced by the deeper layers of the model. While both are technically embeddings, token embeddings are static, whereas the intermediate hidden states evolve as they pass through each layer, capturing the full context of the input. In some literature, these contextual outputs are also referred to as "embeddings," which can be confusing.

In some literature, vector embeddings in the form of hidden states are also referred to as "embeddings," which can be confusing.

What Makes a Good Embedding?

When it comes to LLMs, embeddings can be thought of as the dictionary of their language. Better embeddings allow these models to understand the human language and communicate with us. But what makes an embedding technique good? In other words, what makes an embedding ideal? Here are two major properties of an embedding technique:

Semantic Representation

Some types of embeddings capture the semantic relationship between words. This means that words with closer meanings or relationships are closer in the vector space than words that are less related. For example, the vectors of "cat" and "dog" must be more similar than "dog" and "strawberry".

Dimensionality

What should be the size of an embedding vector, 15, 50, 300? Striking the right balance is key. Smaller vectors (lower dimensions) are more efficient to keep in memory or to process, while bigger vectors (higher dimensions) can capture intricate relationships, but are prone to overfitting. For reference, GPT-2 model family has an embedding size of at least 768.

For reference, DeepSeek-V3 and R1 have an embedding size of 7168.

Traditional Embedding Techniques

Almost every embedding technique relies on a large corpus of text data to extract the relationship of the word. Previously, embedding methods relied on statistic methods based on the occurence or co-occurence of words in a text. This was based on the assumption that if a pair of words often appear together then they must have a closer relationship. These are simple methods that are not as computation-heavy as other techniques. One of such methods is:

TF-IDF (Term Frequency-Inverse Document Frequency)

The idea of TF-IDF is to calculate the importance of a word in a document by considering two factors [1]:

1. **Term-Frequency (TF)**: How frequent a term appears in a document. A higher TF shows that a term is more important to the document.

2. **Inverse Document Frequency (IDF)**: How rare a term is across the documents. This is based on the assumption that terms appearing in multiple documents are less important than terms that are unique to fewer documents.

The formula for TF-IDF consists of two parts. First, the term frequency (TF) is calculated as:

$$tf(t, d) = \frac{\text{count of term } t \text{ in document } d}{\text{total number of terms in document } d}$$

For example, if a document has 100 words and the word "cat" appears 5 times, the term frequency for "cat" would be 5/100 = 0.05. This gives us a simple numerical representation of how prevalent that term is in the document.

Then, the inverse document frequency (IDF) is calculated as:

$$idf(t) = \log\left(\frac{\text{total number of documents}}{\text{number of documents containing term } t}\right)$$

This component gives higher weight to terms that appear in fewer documents. Common words that appear in many documents (like "the", "a", "is") will have a lower IDF, while rare, more informative words will have a higher IDF.

Finally, the TF-IDF score is calculated by multiplying these two components:

$$tfidf(t, d) = tf(t, d) \times idf(t)$$

Let's look at a concrete example:

Suppose we have a corpus of 10 documents, and the word "cat" appears in only 2 of these documents. The IDF for "cat" would be:

$$idf(\text{"cat"}) = \log\left(\frac{10}{2}\right) = \log(5) \approx 1.61$$

If in one particular document, "cat" appears 5 times out of 100 total words, its TF would be 0.05. Therefore, the final TF-IDF score for "cat" in this document would be:

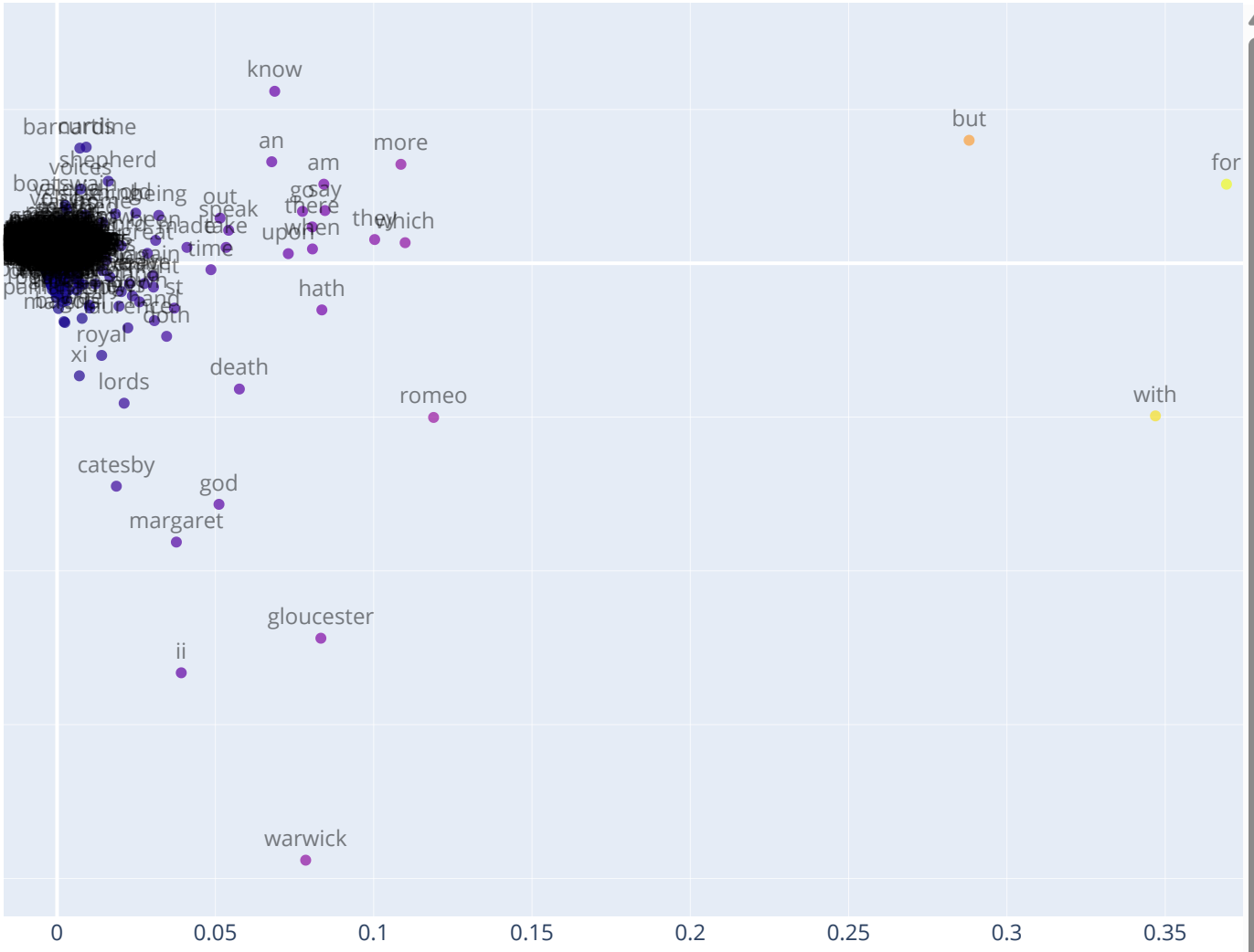
$$tfidf(\text{"cat"}) = 0.05 \times 1.61 \approx 0.08$$

This score tells us how important the word "cat" is to this specific document relative to the entire corpus. A higher score indicates that the term is both frequent in this document and relatively rare across all documents, making it potentially more meaningful for characterizing the document's content.

Let's use TF-IDF on the TinyShakespeare dataset. To simulate multiple documents, we chop off the document into ten chunks.

▶ TF-IDF example using TinyShakespeare

This gives us a 10 dimensional embedding, each for a document we have. Now to get a better idea of the TF-IDF embeddings, we use PCA to map the 10d space to 2d space so we can visualize it better.



TF-IDF example showing word embeddings plotted in 2D space after dimensionality reduction

There are two things noticeable about this embedding space:

1. The majority of the words are concentrated into one particular area. This means that the embedding of most words are similar in this approach. It signals a lack of expressiveness and uniqueness about these embeddings.
2. The embeddings have no semantic connection. The distance between words has nothing to do with their meaning.

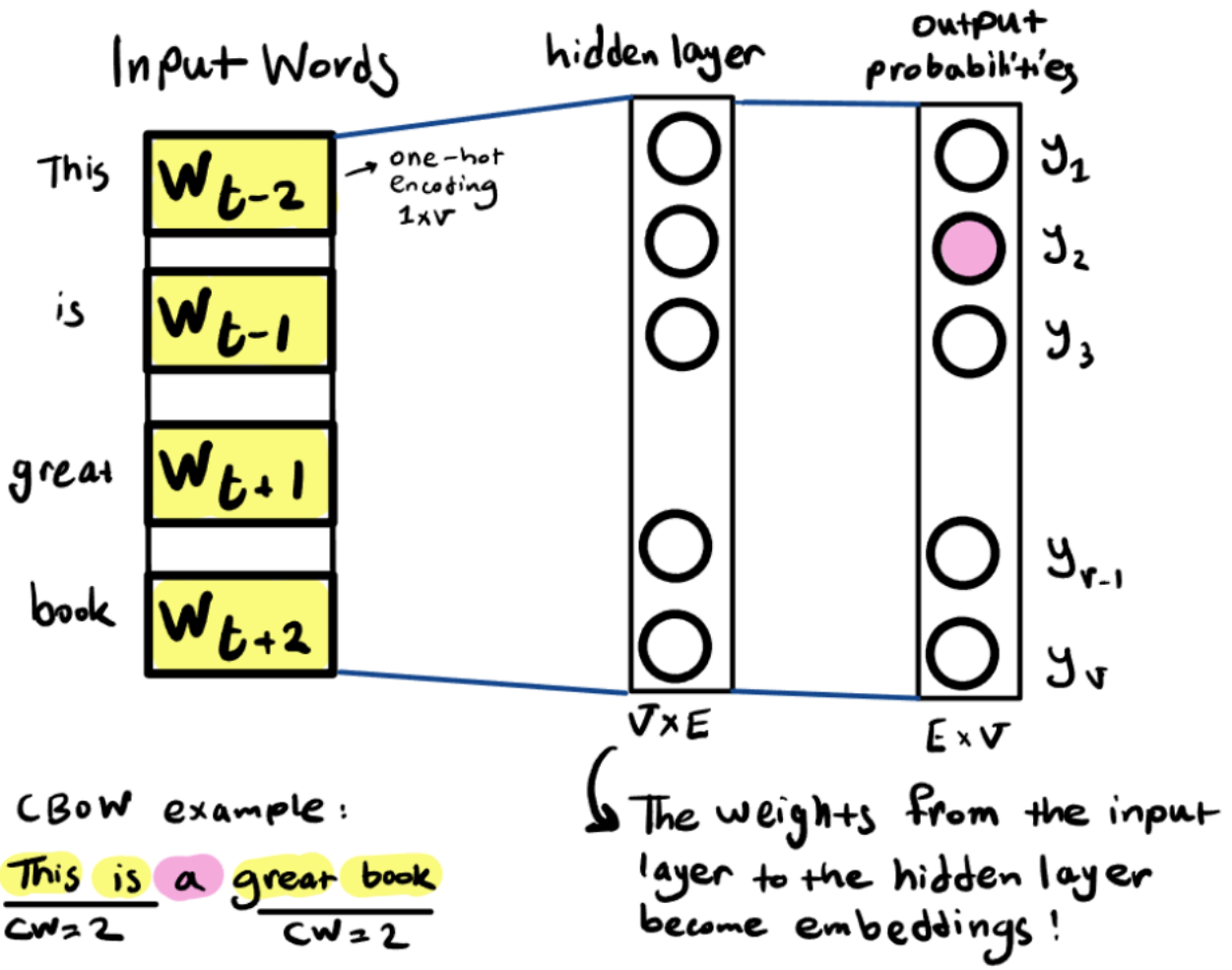
Because TF-IDF is based on the occurrence frequency of terms in the document, words that are semantically close (such as numbers) have no relation in the vector space. The simplicity of TF-IDF and similar statistical methods is what makes them useful in applications such as information retrieval, keyword extraction, and basic text analysis. You can read about some of these methods in [2].

word2vec

Originally proposed in [3], is a more modern technique than TF-IDF. As can be assumed by the name, it

is a network that aims to convert words into embedding vectors. It achieves this by defining a side goal, something to optimize the network for. For example, in CBOW (continuous bag of words), the word2vec network is trained to predict a missing word when its given the neighbors of that word as input. The intuition is that you can infer the embeddings of a word given the words around it.

The word2vec architecture is pretty simple: one hidden layer that we extract the embeddings from, and one output layer which predicts the probabilities of all words in the vocabulary. On the surface, the network is trained to predict the right missing word given its neighbors, but in reality, this is an excuse to train the hidden layer of the network and find the right embeddings for each word. After the network is trained, the last layer can be tossed out the window because figuring out the embeddings is the real goal of the network.



word2vec architecture showing the input layer, hidden layer (embeddings), and output layer

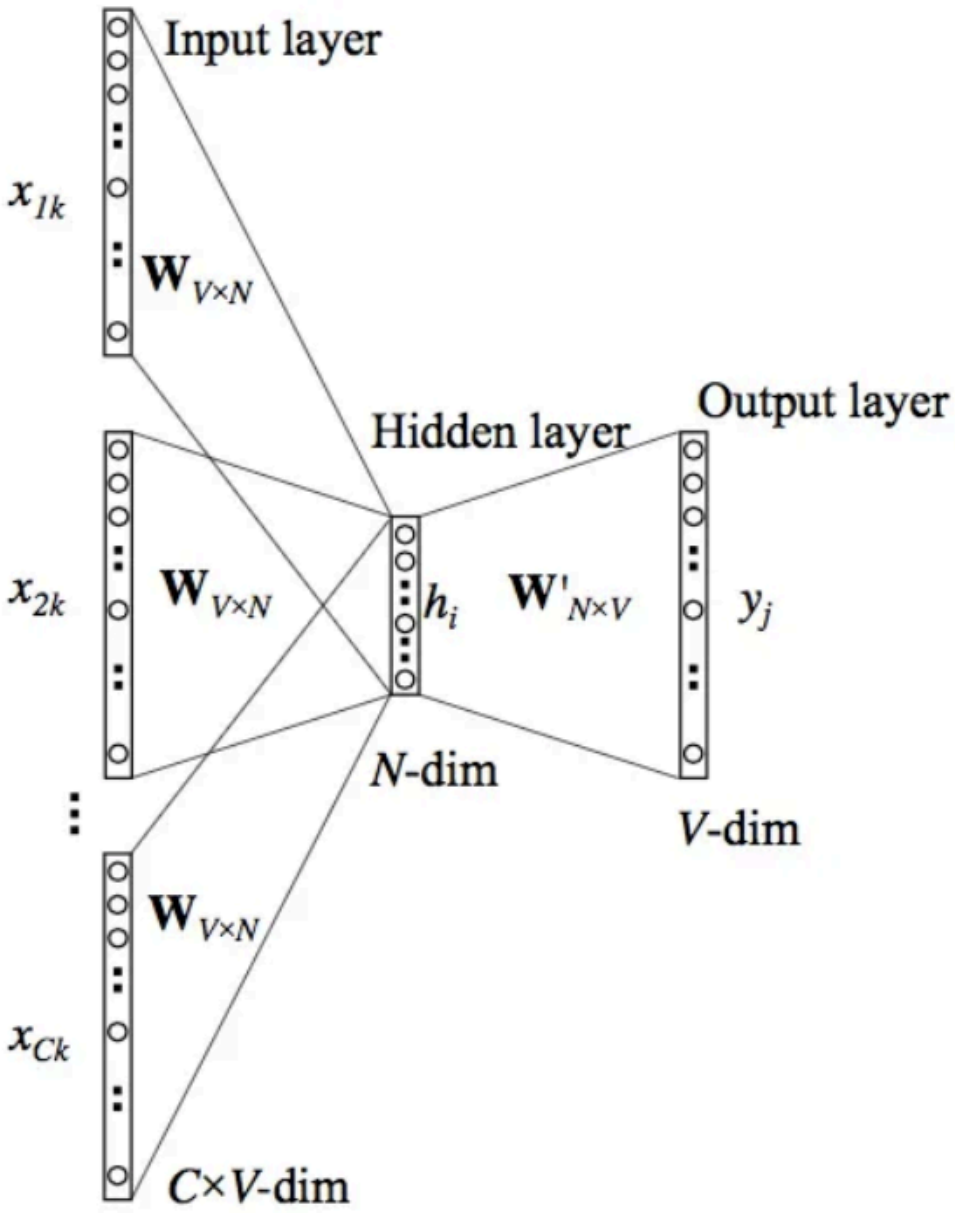
Aside from CBOW, another variant is Skipgram which works completely the opposite: it aims to predict the neighbors, given a particular word as input.

Here's how CBOW word2vec works step by step:

1. Choose a context window (e.g. 2 in the image above)
2. Take the two words before and two words after a particular word as input
3. Encode these four context words as one-hot vectors

4. Pass the encoded vectors through the hidden layer, which has a linear activation function that outputs the input unchanged
5. Aggregate the outputs of the hidden layer (e.g. using a lambda mean function)
6. Feed the aggregated output to the final layer which uses Softmax to predict probabilities for each possible word
7. Select the token with the highest probability as the final output of the network

The hidden layer is where the embeddings are stored. It has a shape of *Vocabulary size x Embedding size* and as we give a one-hot vector (a vector that is all zeros except for one element set to 1) of a word to the network, that specific `1` triggers the embeddings of that word to be passed to the next layers. You can see a cool and simple implementation of the word2vec network in [4].



word2vec embeddings visualization

Since the network relies on the relationship between words in a context, and not on the occurrence or co-occurrence of words as in TF-IDF, it is able to capture **Semantics Relationships** between the words.

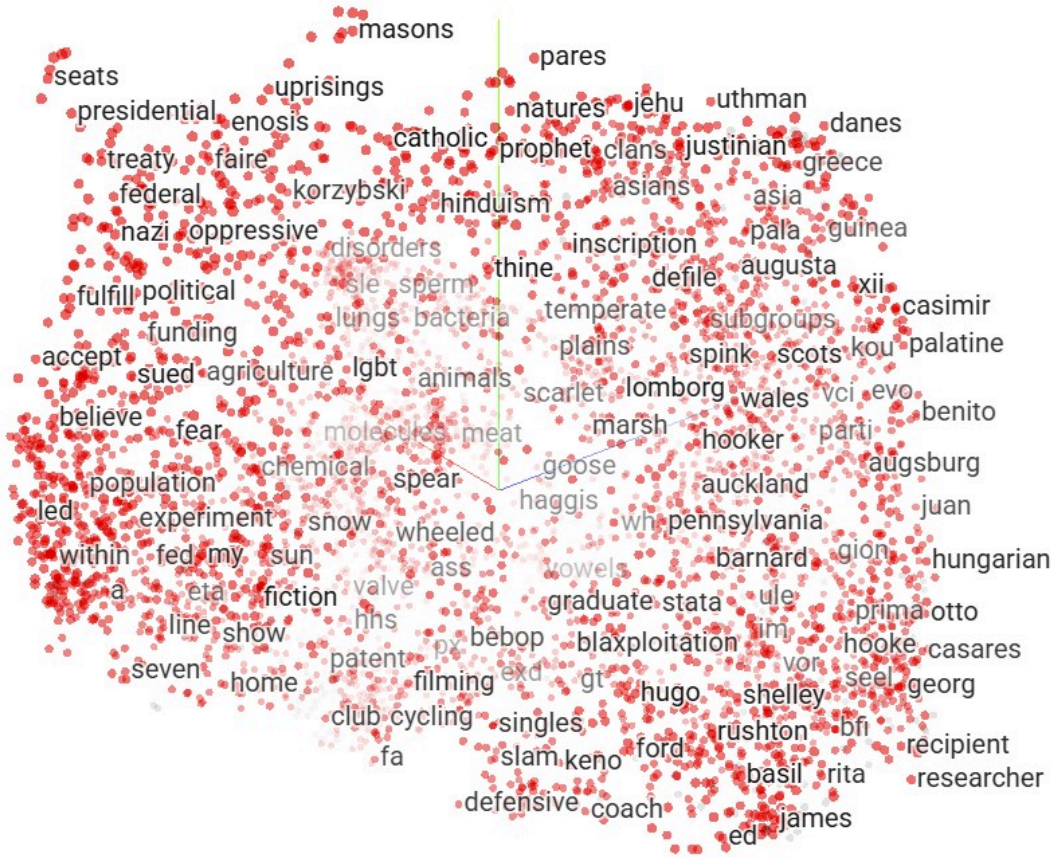
You can download the pretrained version from Google's official page [5]. Here's the code to use word2vec in practice:

▶ word2vec example

To train word2vec efficiently, especially with large vocabularies, an optimization technique called **negative sampling** is used. Instead of computing the full softmax over the entire vocabulary (which is computationally expensive), negative sampling simplifies the task by updating only a small number of negative examples (i.e., randomly selected words not related to the context) along with the positive ones. This makes training faster and more scalable.

The semantic relationship is a fun topic to explore and word2vec is a simple setup for your experiments. You can explore the biases of society or the data, or explore how words have evolved over time by studying the embeddings of older manuscripts.

You can actually visualize and play with word2vec embeddings with Tensorflow Embedding Projector.



TensorFlow Embedding Projector's interactive exploration of word2vec embeddings in 3D/2D space.

BERT (Bidirectional encoder representations from transformers)

Wherever you look in the world of NLP, you will see BERT. It's a good idea to do yourself a favor and learn about BERT once and for all, as it is the source of many ideas and techniques when it comes to LLMs.

Here's a good video to get started on BERT. [6]

In summary, BERT is an encoder-only transformer model consisting of 4 main parts:

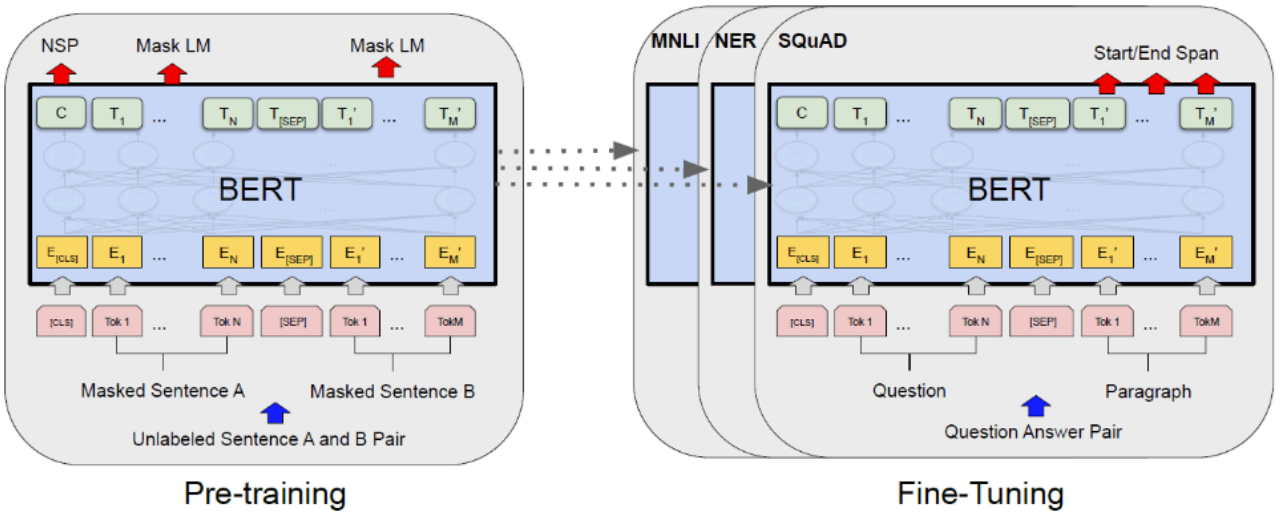
1. Tokenizer: chops up texts into sequences of integers.
2. Embedding: the module that converts discrete tokens into vectors.
3. Encoder: a stack of transformer blocks with self-attention.
4. Task head: when encoder is finished with the representations, this task-specific head handles them for token generation or classification tasks.

BERT inspired from the Transformer architecture introduced in "Attention is all you need", to become an encoder-only transformer that can produce meaningful representations and understand language. The idea was that depending on specific problems to solve, BERT is fine-tuned to learn about that task. These specific tasks can be Q&A (question + passage -> answer), text summarization, classification, etc.

In the pretraining phase, BERT is trained to learn two tasks simultaneously:

1. **Masked Language Modeling**: is to predict masked words in a sentence (I [MASKED] this book before -> read)
2. **Next Sentence Prediction**: given two sentences, predict if A came before B or not. The special [SEP] token separates the two sentences and the task is similar to binary classification.

Note the other special token, [CLS]. This special token helps with classification tasks. As the model processes input layer by layer, [CLS] becomes an aggregation of all the input tokens, which can later be used for classification purposes.



BERT architecture overview. (Image source: [7])

So why is BERT important?

BERT is among the first instances of Transformer-based **contextualized, dynamic embeddings**. When given a sentence as input, the layers of the BERT model use self-attention and feed-forward mechanisms to update and incorporate context from all other tokens in the sentence. The final output of each Transformer layer is a contextualized representation of the word.

Embeddings in Modern LLMs

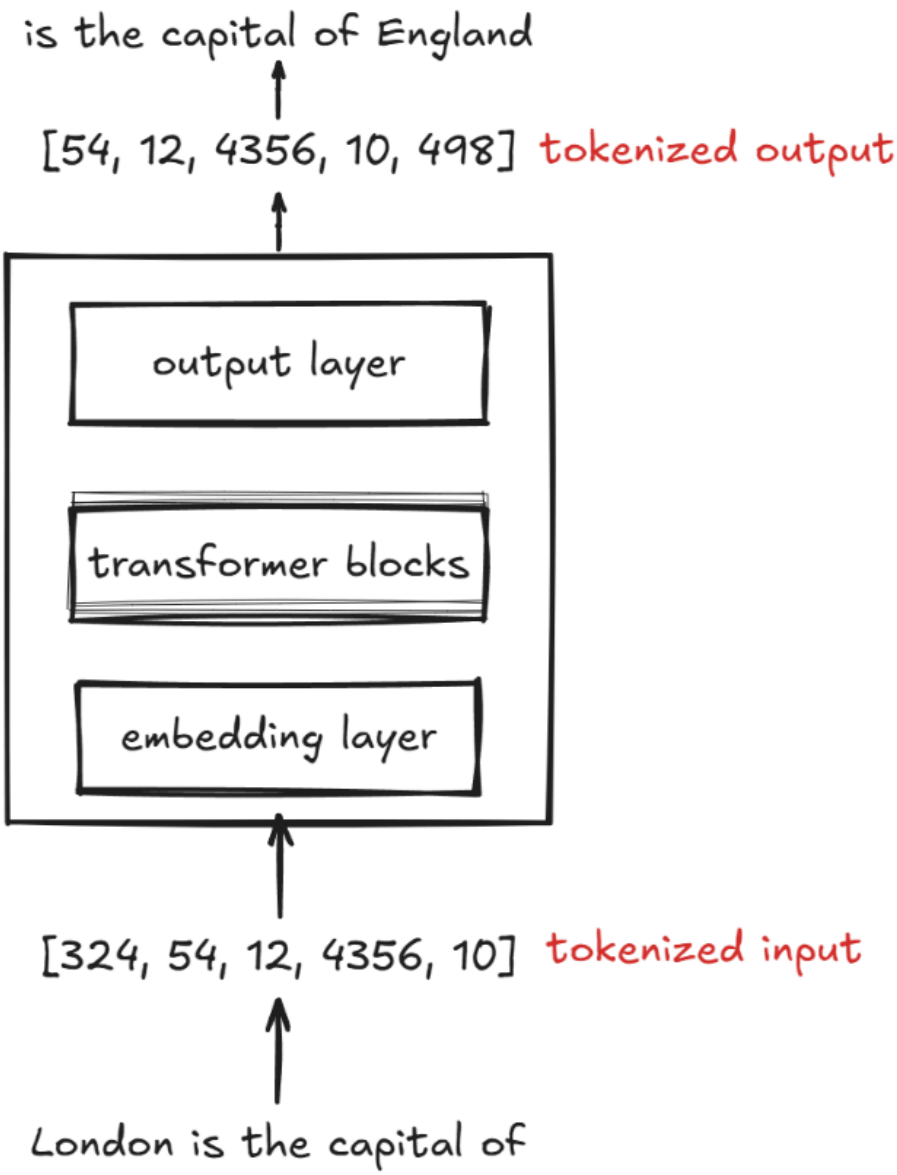
Embeddings are a foundational component in large language models and also a broad term. For the purpose of this article, we focus on "embeddings" as the module that transforms tokens into vector representations as opposed to the latent space in the hiddent layers.

Where does the embedding fit into LLMs?

In transformer-based models, the term "embedding" can refer to both static embeddings and dynamic contextual representations:

1. **Static Embeddings** generated in the first layer and combine token embeddings (vectors representing tokens) with positional embeddings (vectors encoding a token's position in the sequence).
2. **Dynamic Contextual Representations**. As input tokens pass through the self-attention and feed-forward layers, their embeddings are updated to become contextual. These dynamic representations capture the meaning of tokens based on their surrounding context. For example, the word "bank" appears both as "river bank" and "bank robbery", and while the **token embedding** of the word bank

is the same in both cases, the transformations it goes through in the layers of the network take into account the context of which the word "bank" appears in.



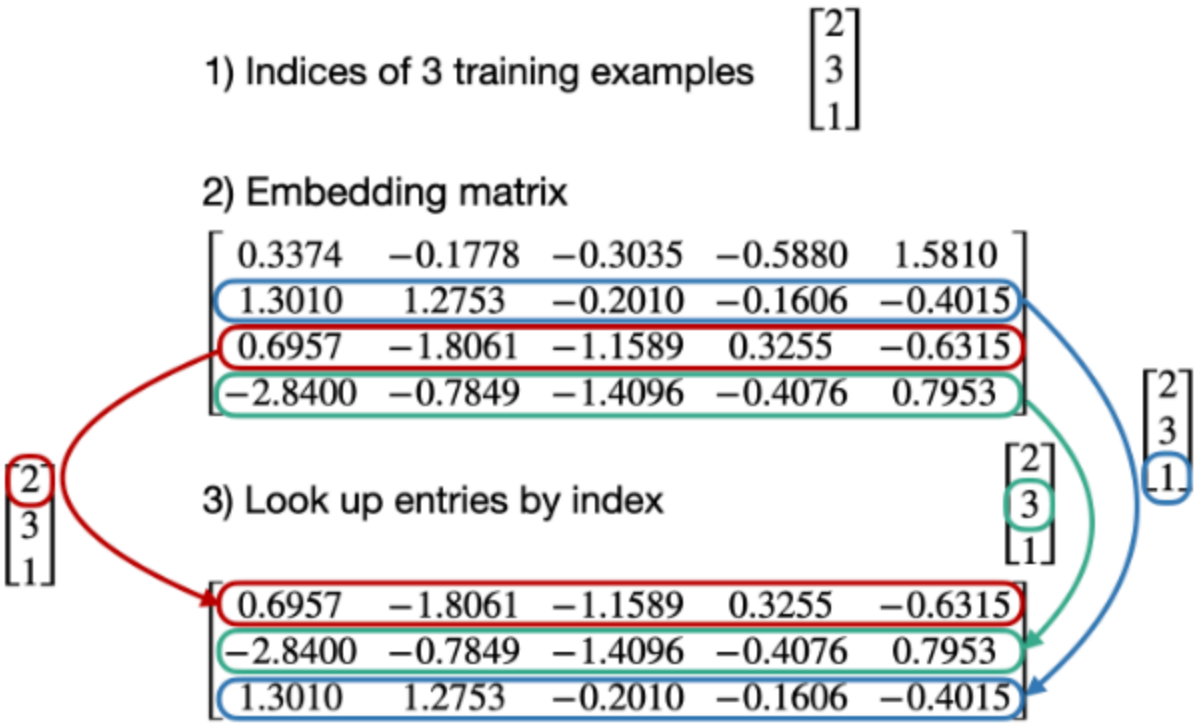
Overview of how embeddings fit into the LLM architecture

LLM Embeddings are Trained

LLM embeddings are optimized during the training process. Borrowing from Sebastian Raschka's **Build a Large Language Model (From Scratch)** [8], " *While we can use pretrained models such as Word2Vec to generate embeddings for machine learning models, LLMs commonly produce their own embeddings that are part of the input layer and are updated during training. The advantage of optimizing the embeddings as part of the LLM training instead of using Word2Vec is that the embeddings are optimized to the specific task and data at hand.*"

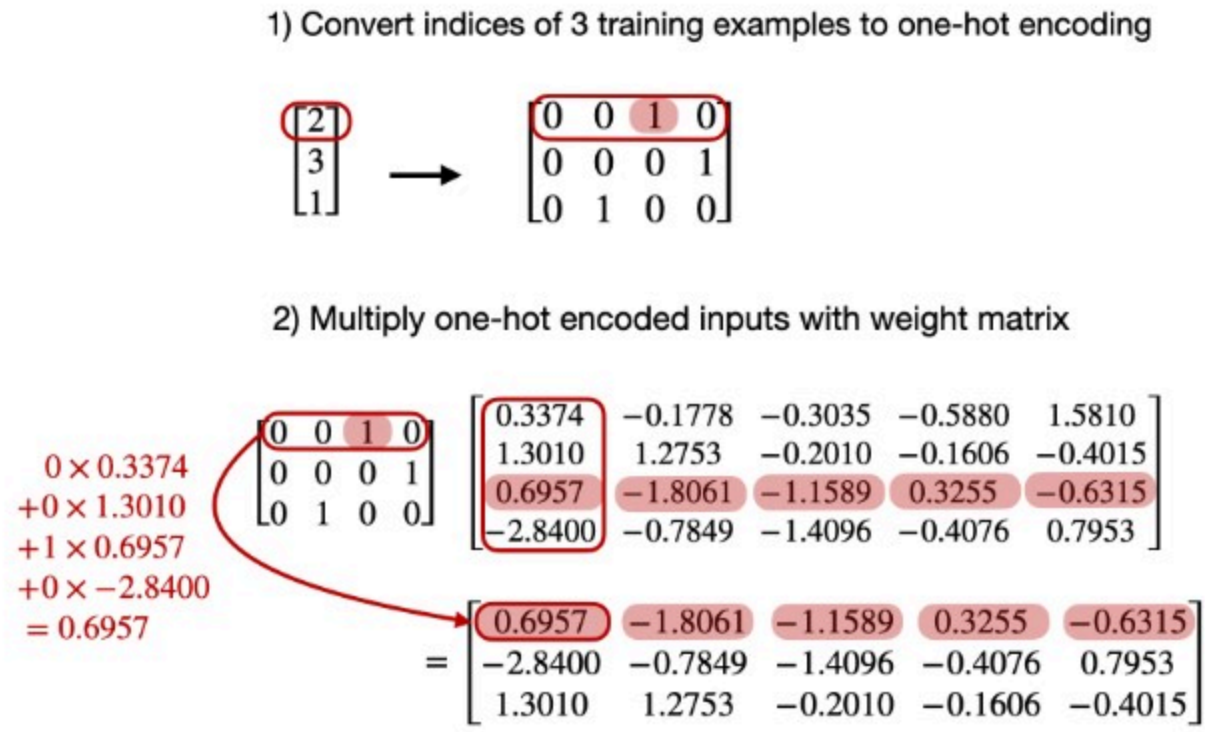
torch.nn.Embedding

The embedding layer in LLMs works as a look-up table. Given a list of indices (token ids) it returns their embeddings. Build a Large Language Model (From Scratch) [8] shows this concept comperhensively.



Visualization of how the embedding layer works as a lookup table. (Image source: [8])

The code implementation of an embedding layer in PyTorch is done using `torch.nn.Embedding` which acts as a simple look-up table. There is nothing more special about this layer than a simple Linear layer, other than the fact that it can work with indices as input rather than one-hot encoding inputs. The Embedding layer is simply a Linear layer that works with indices.



Visualization of the embedding lookup process. (Image source: [8])

This notebook by Sebastian Raschka explains the Embedding layer in depth [9].

Now let's work with the embedding of a model and see some visuals!

Embeddings in Action (DeepSeek-R1-Distill-Qwen-1.5B)

How does the embedding layer in a large language model look like?

Let's dissect the embeddings of the distilled version of DeepSeek-R1 in the Qwen model. Some parts of the following code is inspired by [10].

We begin by loading the `deepseek-ai/DeepSeek-R1-Distill-Qwen-1.5B` model from Hugging Face and saving the embeddings.

▶ Load the model and save the embeddings

Now let's load the embedding layer and work with it. The goal of separating the embedding from the other parts of the model, saving, and loading it is to get the embeddings of an input much faster and efficiently rather than doing a complete forward pass of the model.

▶ Load the model embeddings

Now let's see how a sentence is tokenized and then converted to embeddings.

▶ Convert prompt to embeddings

In the above code, we tokenize the sentence and print the embeddings of the tokens. The embeddings are 1536-dimensional vectors. Here is a simple example with the sentence: "HTML coders are not considered programmers":

token_id	token	Embedding Vector (1536 dimensions)
151646		-0.027466, 0.002899, -0.005188 ... 0.021606
5835	HTML	-0.018555, 0.000912, 0.010986 ... -0.015991
20329	#cod	-0.026978, -0.012939, 0.021362 ... 0.042725
388	ers	-0.012085, 0.001244, -0.069336 ... -0.001213
525	#are	-0.001785, -0.008789, 0.006195 ... -0.016235
537	#not	0.016357, -0.039062, 0.045898 ... 0.001686
6509	#considered	-0.000721, -0.021118, 0.027710 ... -0.051270
54846	#programmers	-0.047852, 0.057861, -0.069336 ... 0.005280

Example of token embeddings for the sentence "HTML coders are not considered programmers"

Finally, let's see how we can find the most similar embeddings to a particular word. As embeddings are vectors, we can use cosine similarity to find the most similar embeddings to a particular word. Then, we can use the `torch.topk` function to find the top k most similar embeddings.

▶ Find similar embeddings

Embeddings as Graphs: A Network Analysis

How can we view the embeddings? One method is to look at the embedding layer as a network, in which tokens are the nodes; if two token vectors are close then we assume their nodes are connected via an edge.

As an example, if we take the sentence *"AI agents will be the most hot topic of artificial intelligence in 2025."*, tokenize it, convert the tokens to embeddings, find the 20 most similar embeddings to each of the ones we had, the following will be the embedding graph:

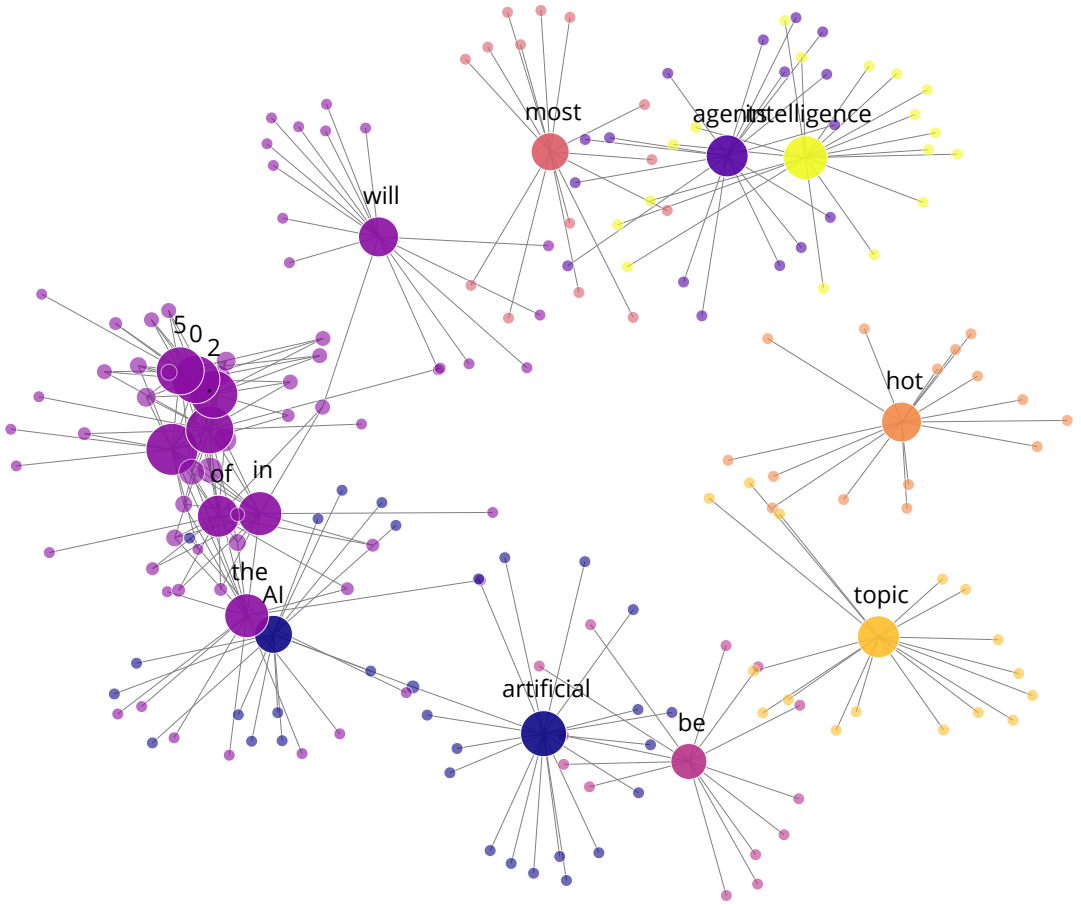


Figure 2: Visualization of the embedding graph for the sentence "AI agents will be the most hot topic of artificial intelligence in 2025."

You can actually see a more comprehensive example at the beginning of the article in which 50 tokens and their closest tokens are mapped out.

In the graph examples, we have ignored the "token variations". A token or word may have many different variations, each with their own embeddings. For example, the token "list" may have many different variations with their own embeddings, such as "_list", "List", and many more. These variations often have an embedding that is very similar to the original token. The following is a graph of the token "list" and its close neighbors, including its variations.

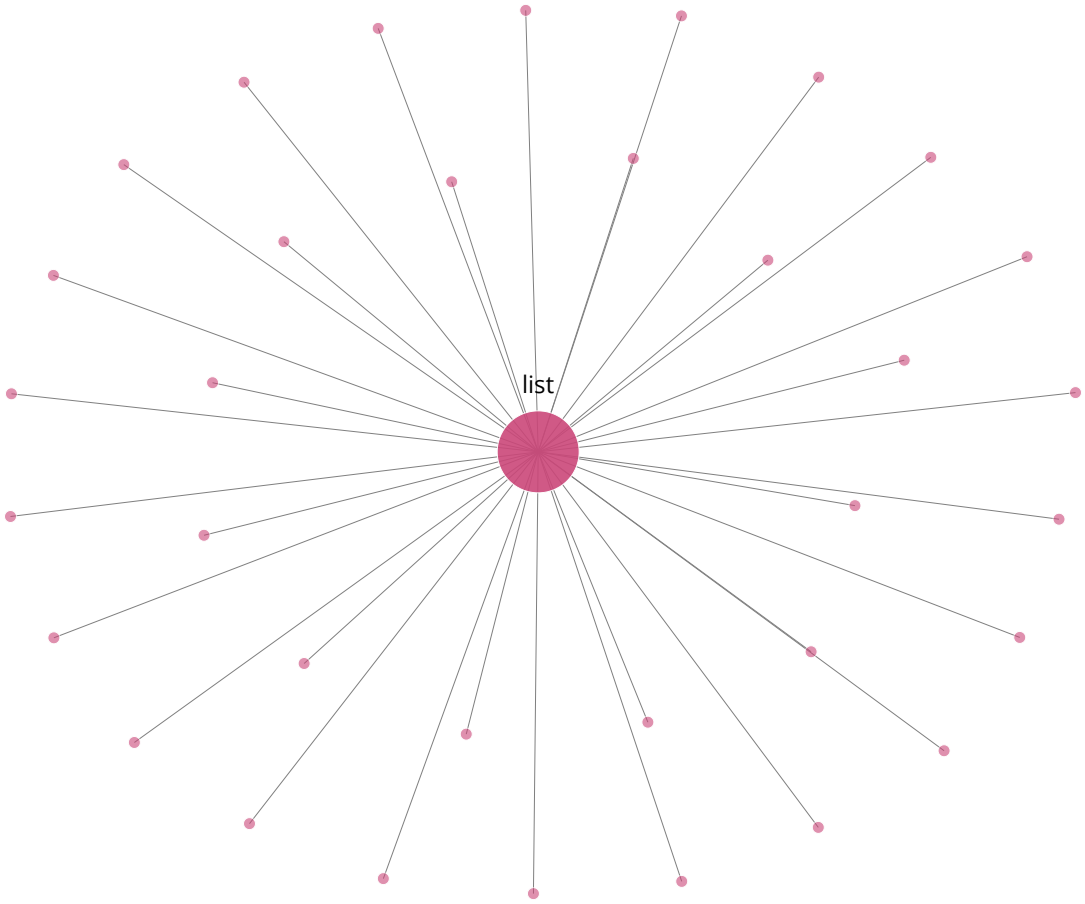


Figure 3: The word “list” and some of the closest tokens to it including its variants.

Let's Wrap Up

This blog post's template is inspired by the distill.pub, if you want to use it for your own blog post, you can find the template here.

Embeddings remain as one of the fundamental parts in natural language processing and modern large language models. While the research in machine learning and LLMs discover new methods and techniques every day, embeddings haven't seen much change in large language models (and that has to mean something). They are essential, easy to understand, and easy to work with.

In this blog post we went through the basics of what you need to know about embeddings, and their evolution from traditional statistical methods into their use case in today's LLMs. My hope is that this has been a comprehensive jump-start to help you gain an intuitive understanding of the word embeddings and what they represent.

Thank you for reading through this article. If you found this useful, consider following me on X (Twitter) and Hugging Face to be notified about my next projects.

If you have any questions or feedback, please feel free to write in the community.

Citation

For attribution in academic contexts, please cite this work as

```
"LLM Embeddings Explained: A Visual and Intuitive Guide", 2025.
```

BibTeX citation

```
@misc{llm_embeddings_explained,
        title={LLM Embeddings Explained: A Visual and Intuitive Guide},
        author={Hesam Sheikh Hessani},
        year={2025},
        }
```

References

1. **A Comprehensive Guide to Word Embeddings in NLP** [link]
   Vardhan, H., 2024. Medium.
2. **A Guide on Word embeddings in NLP** [link]
   Turing,, 2022.
3. **Efficient estimation of word representations in vector space** [PDF]
   Mikolov, T., Chen, K., Corrado, G. and Dean, J., 2013. arXiv.org.
4. **Implementing deep learning methods and feature engineering for text data: The Continuous Bag of Words (CBOW)** [HTML]
   Sarkar, D.. KDnuggets.
5. **word2vec** [link]
   . Google Code Archive.
6. **BERT Neural Network - EXPLAINED!** [link]
   CodeEmporium,, 2020. YouTube.
7. **BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding** [PDF]
   Devlin, J., Chang, M., Lee, K. and Toutanova, K., 2018. arXiv.

8. **Build a large language model (From scratch)** [link]
. Manning Publications.

9. **LLMs-from-scratch** [link]
Rasbt,. GitHub.

10. **embeddings** [link]
Chrishayuk,. GitHub.