Retrieval Augmented Generation (RAG): A Comprehensive Visual Walkthrough 🧠📖🔗🤖

A Simple Illustrated Guide!

JAIGANESAN · Follow
Published in Towards AI · 15 min read · 6 days ago

Photo by Andrea De Santis on Unsplash

You might have heard of Retrieval Augmented Generation, or RAG, a method that's been making waves in the world of AI. But what exactly is RAG, and

why is it gaining so much attention? In this article, we'll take a closer look at this innovative approach and explore what makes it so **special.** 👻

We'll start by examining What problems does it solve, and how does it address the limitations of traditional Models Like LLM? Next, we'll dive deeper into the inner workings of the RAG architecture, exploring its mechanism and how it Generate content.

By the end of this article, you'll have a clear understanding of RAG and its potential to transform the way we Generate content.

> ***Note:*** *This article is not about building RAG with code. It's about understanding RAG. This will be a Simple illustrated Article to understand RAG.*

how does RAG work? It's quite simple. The process involves three key steps: Retrieval, Augment, and Generation.

Here's a breakdown of each step:

RAG Pipeline.

Retrieval → Retrieve Context (Chunks) from vector database relevant to the query.

Augment Generation → Generate content based on Query, Prompt, and retrieved Context.

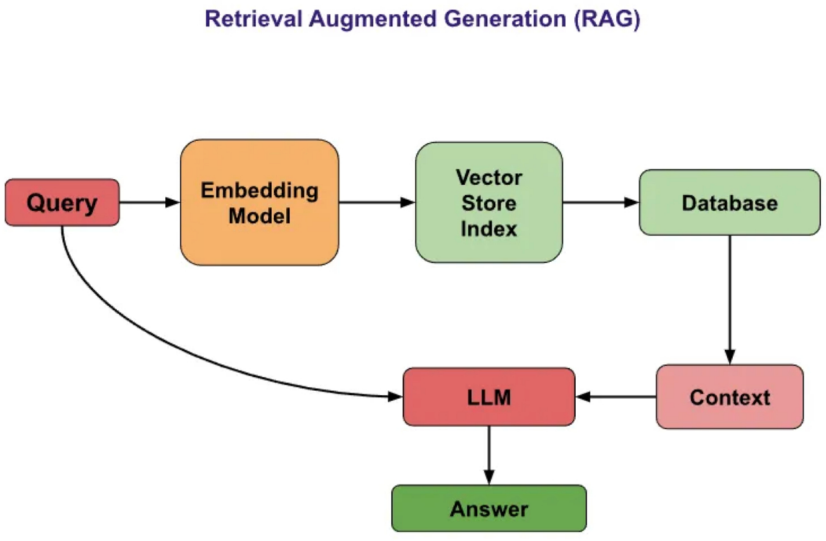


Image 1: Simple RAG Architecture. Created by author

Let's take a closer look at the Simple RAG architecture shown in **Image 1.** This innovative design enables queries to navigate through an embedding model, ultimately retrieving content (chunks) from a vector database that's highly relevant to the query.

Specifically, we'll be exploring the following topics:

👉🏽 What problem does RAG solve?

👉🏽 Role of the embedding model in RAG

👉🏽 How context retrieval works in RAG

👉🏽 Response generation in RAG

1. So, What problem does RAG solve? 🥳🤤

LLMs are trained on Large amounts of text data. It can answer queries based on the training data. Here, I would like to share a quote that I have read from one restaurant that serves organic and hygienic food, which is relevant to this context.

> Your body is what you eat, and your mind is what you listen, read, and what you tell to yourself.

The same applies to AI Models also (LLMs). LLM can **only answer** questions that **have been trained on**. The Problem with current existing LLMs are not able to learn new data continuously. Once the LLM is trained, it does not update or learn from new data in real-time. Its learning process is time discrete because they are retrained or fine-tuned at specific points in time to get new knowledge.

When it comes to Large Language Models (LLMs), there are ways to learn new knowledge: fine-tuning or retraining. While these methods do allow LLMs to learn from new data, they come with a significant drawback 😠. Both fine-tuning and retraining are computationally expensive — require a lot of **processing power and resources**.

To make matters worse, if new data becomes available, we have to go through the entire process again — retraining or fine-tuning the model. This is **inefficient** and **time-consuming**, which can hinder the model's ability to stay up-to-date with the latest information and affect deployed applications in real-time 😫.

For Medium and Small-level organizations, Retraining and fine-tuning for continuous learning can be challenging. To address the limitations of LLMs, a new approach, **RAG** was proposed. RAG offers a promising solution to the problem of continuous learning. By leveraging RAG, we **can avoid the need to fine-tune or retrain** our Large Language Models (LLMs). 🥳

When new information becomes available, it's converted into embeddings,

which are then stored in a vector database or local storage. This allows the model to quickly retrieve relevant information when a query matches the embeddings in the vector storage. We'll dive deeper into the details of RAG in the following sections, exploring how it works.

2. Role of the embedding model in RAG

Embedding models are an important component of Retrieval Augmented Generation. As humans, we can easily understand words, sentences, and paragraphs, as well as the relationships between them.

But have you ever wondered how models understand words and sentences? The answer lies in numerical representations, also known as vectors. Models understand the words and sentences by representing them as vectors in a multidimensional space.

How are chunks Converted into Sentence Vectors?

To do this, **sentences** are first broken down into **individual tokens**, which are then represented as **indices in a vocabulary** (using a one-hot representation). These index representations are then converted into vectors (Numerical Representations of Words and Sentences) as shown in Image 2.

Input words (Ex., Chunks, Queries)								
A Large Language Model is an autoregressive model								
Token								
A	Large	Language	Model	is	an	Auto	regressive	Model
One hot representation (Vocabulary Index)								
5	1001	4563	7684	9	12	43	501	7684
Word Embedding (9, 768)								
0.63167	-1.157963	-0.852893	-0.136668	-0.702993	0.252234	-0.302899	-0.518876	-0.1366677
-0.1907	1.2057905	0.970417	-0.133376	-0.89344	-1.456634	-1.435337	0.171586	-0.1333761
-0.18304	-0.178745	1.181625	-0.019606	1.074921	-1.079207	-0.638025	-0.148254	-0.0196062
1.22445	-1.385803	-1.295329	0.002428	0.158683	0.4562164	0.960311	-0.34365	0.00242828
0.42426	-0.322022	0.619989	-0.236081	0.215261	-0.819276	1.006216	-0.380615	-0.2360814
0.45144	0.0855985	0.706255	0.565646	0.565146	0.2780123	0.091926	0.210502	0.56564584
0.73696	1.1573582	-0.54676	1.356768	0.444684	-1.422231	-0.868668	-0.545618	1.3567677
1.3232	-0.194124	-1.49879	-0.436102	1.215918	0.9220019	-0.865242	-1.177459	-0.4361024
...	...	...	...	...	...	...	...	...
...	...	...	...	...	...	...	...	...
...	...	...	...	...	...	...	...	...
-1.36716	-1.279107	-0.88722	0.969476	-0.221586	-1.464506	-0.913651	-0.054009	0.96947605
-0.25897	0.6665937	0.180833	-1.002644	1.0813	-0.136742	-0.710683	1.272237	-1.0026444

Image 2: Word Vectors. Created by author

```
# Begining of training --> Vector initialization

word_embeddings = torch.nn.Embedding(num_embeddings, embedding_dim, padding_idx=

# num_embeddings = Vocab size (Ex., 32000)
# embedding_dim = Model dimension or Vector Dimension (Ex., 768)
```

> ***Note:*** *During the training or inferencing of Large Language Models (LLMs), positional information is incorporated into the input vectors. This is done using a sinusoidal function, as seen in the Vanilla Transformer, or the RoPE method.*

Initializing Word Embeddings :

So, how are these embeddings initialized in the first place? Initially, embeddings are randomly initialized. As the model (Both LLM and Embedding model) continues to train, something interesting happens. Tokens that are similar in meaning start to **cluster together** in the multidimensional space. As a result, the similarity between the vectors of similar words becomes very high, close to 1. This is why the vectors of words with similar meanings end up being close to each other in multi-dimensional space.

We have seen how words are represented in multi-dimensional space. But how are sentences or chunks represented as vectors?

By Changing the Final Linear layer in the Encoder transformer model like BERT, we can get the **sentence Vector or sentence embeddings.**

```
from sentence_transformers import SentenceTransformer

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

#Example chunks
chunks = ['Retrieve Context from vector database relevant to the query',
             'Augment the retrieved context based on prompts and other methods',
            'Generate content based on Query, Prompt, and retrieved Context']

#encode chunks
embeddings = model.encode(chunks)
# Embeddings or vectors
for chunk, embedding in zip(chunks, embeddings):
    print("Chunk:", chunk)
    print("Embedding size:", embedding.shape)
    print("Embedding:", embedding[0:10])

# Output: **

Chunk: Retrieve Context from vector database relevant to the query
Embedding size: (768,)
Embedding: [ 0.04679576 -0.41188022  1.1416095   0.79346526  0.5990003   0.12312
 -0.7845492   0.24790691  0.56256    -0.00114676]
Chunk: Augment the retrieved context based on prompts and other methods
Embedding size: (768,)
Embedding: [ 0.10020391 -0.31519264  2.2251918   0.53598183  0.67712927 -0.08380
 -0.6464507   0.08609668  0.558082   -0.703896  ]
Chunk: Generate content based on Query, Prompt, and retrieved Context
Embedding size: (768,)
Embedding: [-0.17665297 -0.21692613  1.7279574   0.76316905  0.4093552  -0.08587
 -0.47385958  0.52885044  0.58931875 -0.08809798]
```

Encoder :
① Self Attention
② Feed Forward NN.

BERT Model :

The above code shows the Sentence vector generation from the BERT-based model. This BERT has been trained to generate similar embeddings for similar chunks. The **BERT** stands for **Bidirectional Encoder Representations from Transformers**. The Encoder layer is composed of two main components: **self attention** and **feed forward network** layers. These layers

components: **self-attention** and **feed-forward network** layers. These layers work together to help the model understand the entire sentence or chunk of text.

Decoder :

Auto Regressive Model

LLM (Decoder architecture) is an autoregressive model, which means the next token is predicted based on the current context. By applying a causal mask in the attention layer, LLM obtains the Autoregressive property. When Causal masks are applied, the current token can only attend to previous tokens, not the next tokens in the sequence, which helps LLM to predict the next token based on the current context.

But BERT is different, In BERT the **causal mask** is **not applied**, so the current token attends all the tokens in the sequence(chunks, query). It attends previous tokens and the next tokens. Tokens attend other tokens in **both directions** in the sequence. That's why this transformer method got its name, **Bidirectional Encoder Representation.** The Sentence embedding vectors have contextual information, positional information, and the relationship between tokens in the sequence.

```
# By Changing the Linear Layer we can get the sentence vector
token_embeddings = torch.nn.Linear(model_dim, vector_size, bias=False)
sentence_embedding = torch.mean(token_embeddings, dim=0)
```

The Output vectors of the BERT have rich information about the sequence. We use the mean pool method to combine all sentence vectors into a single vector. This sentence vector comprehensively represents the sequence/chunks/queries.

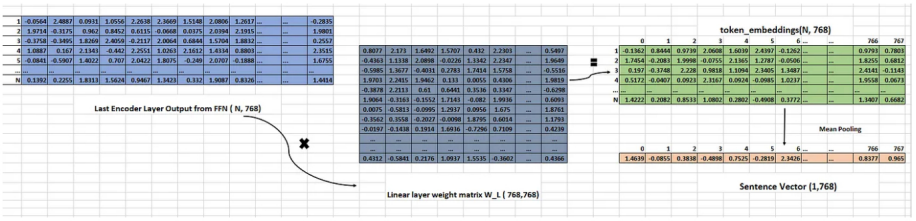


Image 3: Sentence Vector. Created by author

Linear

CLASS torch.nn.Linear(*in_features*, *out_features*, *bias=True*, *device=None*, *dtype=None*) [SOURCE]

Applies a linear transformation to the incoming data: $y = xA^T + b$.

Image 4: Linear layer function. source: pytorch.org

As illustrated in Image 3, the encoder layer output matrix is the size of (N, 768), where N is chunks or query length (No.of Tokens) and 768 is the model

dimension. This Matrix is then multiplied by a linear layer weight matrix and averaged, resulting in a sentence vector of size 768 that effectively captures the information of the entire input.

> ***Note:*** *Both the model dimension and Sentence vector dimension are 768. The higher the dimension and enough training data, the representation of word will be good.*

3. How context retrieval works in RAG

To understand context retrieval, we need to understand Chunk, Vector database and index, Cosine Similarity, and Vector search in RAG.

3.1 Chunks

When we talk about chunks, we're simply referring to **small amounts of information**. In the real world, organizations often have their data stored in various formats like PDFs, Word documents, text files, or databases. These documents usually contain a vast amount of diverse knowledge. To make sense of this data and retrieve specific answers to our questions, we need to break it down into smaller, manageable pieces. This process is called chunking. For example, if we have a document with 10,000 words and we decide to divide it into chunks of 500 words each, we would end up with 20 smaller chunks of information.

```
# Chunking from LLama Index
from llama_index.core.node_parser import SentenceSplitter

splitter = SentenceSplitter( chunk_size=500, chunk_overlap=20,)
nodes = splitter.get_nodes_from_documents(documents) # documents --> PDF, word d
nodes[0]
```

There are various ways to chunk data, and the approach you choose depends on the type of data, accuracy, latency, and the application you're building. For example, you might use a semantic splitter, code splitter, Sentence Splitter, or token splitter, each with its strengths and weaknesses.

3.2 Vector database and index

A vector store index is a type of data structure or database that is specifically designed to store, search, and manage vectors. It allows for efficient searching and retrieval of data based on their vector representations. This is important for tasks such as nearest neighbor search and similarity search, which involve finding vectors that are close to a given query in the vector space.

Chroma, Pinecone, and Qdrant are some of the Famous Vector databases. These databases index and store vector embeddings for fast retrieval and similarity search.

The **chunks** are converted **into high-dimensional vectors** using an embedding model. We have seen, how the chunks are converted into sentence vectors. The vector embedding is inserted into the vector databases, with some reference to the original content the embedding was created from.

When the query is given, the same embedding model converts the query into embedding, and those embeddings are used to query the database for similar embeddings as I mentioned before, those similar embeddings are associated (Database) with the original content used to create vector embeddings.

3.3 Cosine Similarity

We have seen How the chunks are converted into vectors and where the vectors are stored. Before we explore search methods, we should be familiar with **how the search happens**. There 2 methods (Cosine similarity and Euclidean or Manhattan distance) often used in finding chunks similar to the query.

Cosine similarity helps us to calculate the similarity between two vectors. This cosine similarity ranges from -1 to 1 ([-1,1]). Where 1 means similar or identical vectors, 0 means they are orthogonal, and -1 means they are complete opposite vectors.

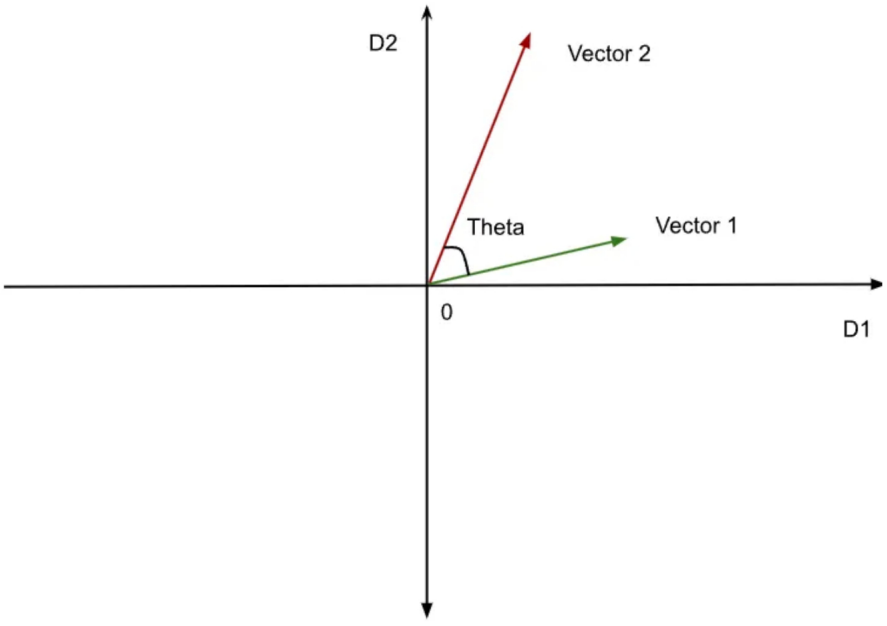


Image 5: Cosine distance between 2 vectors in 2D space. Created by author

For two vectors to be considered similar, the angle between them (θ) needs to be close to 0 degrees. The closer the angle gets to 0, the more similar the vectors are, and the cosine of that angle (cos(θ)) will approach 1.

To illustrate this, let's take a look at Image 5, which shows the angle between two vectors in the 2-dimensional space. As the angle θ gets closer to 0, the cosine of that angle (cos(θ)) gets closer to 1. Opposite to this, if the angle is close to 180 degrees, the cosine of that angle will be close to -1.

How is the Cosine similarity calculated for Multi-dimensional vectors?

To calculate the similarity between two vectors in a high-dimensional space, we use a specific formula. This formula helps us determine how alike or dissimilar the vectors are.

$$Similarity\ (A, B) = \frac{A.B}{(||A||.||B||)}$$

Image 6: Cosine similarity formula. Image by author

Cosine similarity (Query_vector, chunk_vector) = (Query_vector.chunk_vector)/ (|| Query_vector || . ||Chunk_vector||)

Query_vector.chunk_vector is the dot product of two vectors (two vectors must be in the same dimension).

|| Query_vector || is the magnitude (norm) of Query_vector.

||Chunk_vector|| is the magnitude (norm) of Chunk_vector.

```
import numpy as np

# 10 dimensional vectors

vector_1 = np.array([1.23,1.89,-0.23,-0.12,1.75,0.543,0.671,0.11,2.3,-1.6])
vector_2 = np.array([2.33,1.76,-0.12,0.01,1.35,0.211,0.96,-0.12,1.7,-1.5])

# Magnitude or Length of vectors
length_v1 = np.linalg.norm(vector_1)
length_v2 = np.linalg.norm(vector_2)

print("Magnitude of vector_1:", length_v1)
print("Magnitude of vector_1:", length_v2)

# Cosine Similarity
cosine_similarity = np.dot(vector_1, vector_2) / (np.linalg.norm(vector_1) * np.

print("Cosine similarity:", cosine_similarity)

# Output like this :-

Magnitude of vector_1: 4.101461934481411
Magnitude of vector_1: 4.060051846959593
Cosine similarity: 0.9390219750577464
```

For example, let's take a look at the code snippet that shows how to calculate

the Cosine similarity between two 10-dimensional vectors. This code gives us a practical demonstration of how the formula works in real-world scenarios.

To give you a better understanding I demonstrate cosine similarity using sentence Chunks.

```
from sklearn.metrics.pairwise import cosine_similarity
from sentence_transformers import SentenceTransformer

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

#chunks
chunks = ['Retrieve Context from vector database relevant to the query',
          'Augment the retrieved context based on prompts and other methods',
          'Generate content based on Query, Prompt, and retrieved Context']

#Chunks are converted into vectors
embedding_vectors = embedding_model.encode(chunks)

cs = cosine_similarity([embedding_vectors[0]], [embedding_vectors[1]])[0][0]
print (f"Cosine similarity: {cs:.4f}")

# Output :

Cosine similarity: 0.7222
```

The first two chunks are 72 % similar. This is how the similarity between two vectors is calculated in a vector database.

Note: Euclidean distance or Manhattan distance helps us calculate the distance between two vectors in the Multidimensional space (Similar to KNN). A smaller distance means the two vectors are close in multi-dimensional space.

3.3 Vector Search Methods

When it comes to searching for vectors (Searching matching chunks for queries), there are several methods that are widely used today. In this part, we'll delve into two of the methods: Naive search, and HNSW. They differ in terms of how efficient and effective they are.

Naive Search (FLAT)

In this method, the query is searched against all the chunks in the vector index. There will be N (number of chunks) number of similarity computations in this method. Even though this method yields better results, this method is inefficient, Computationally expensive, and has high latency. This method is a kind of brute force to find all the query's nearest neighbors in the multi-dimensional space. At the End, Top k high similarity chunks are retrieved and given to LLM as Input with Prompt.

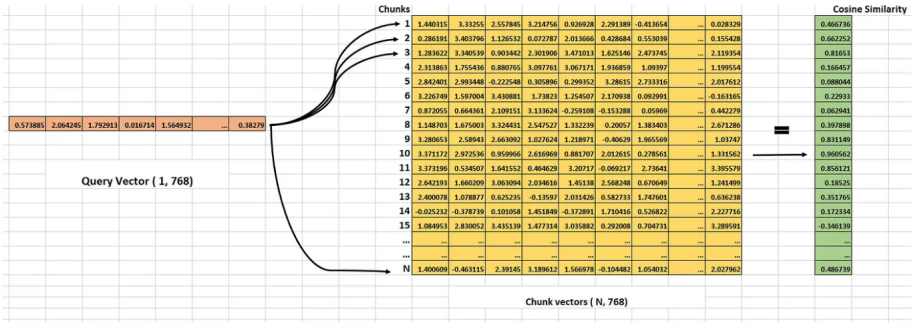


Image 7: Naive search. Image by author

HNSW (Hierarchical Navigable small worlds)

The Metric we usually care about in similarity search is Recall. Recall score means among all the true values, how many values are predicted as True. NSW and HNSW, **approximate Nearest neighbors**. It may not give all the nearest neighbors (Chunks), But it is computationally efficient and yields good results. HNSW powers Qdrant, open source vector database. And Qdrant only uses HNSW as a **vector index algorithm.**

HNSW is an Evolution of the **Navigable small worlds (NSW)** algorithm for the approximate nearest neighbors (Finding most similar chunks), which works based on the concept of **SIX Degrees of separation**.

What SIX Degrees of separation means is that people all over the world are all connected by six degrees of separation. This means that I and the Prime Minister of India are Only 6 connections (Maximum connections) apart. This concept is Proven.

Navigable Small World

The NSW algorithm builds a graph that (Similar to social media connections) connects close vectors with each other but keeps the total number of connections small (To mimic the Six degrees of separation concept).

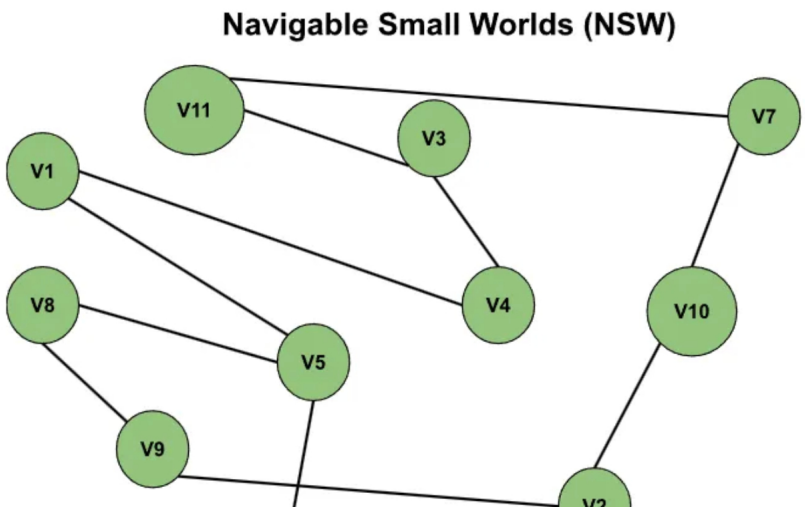


Image 8: Navigable small worlds. Image by author

For example, Let's take an example of 11 chunks of vectors in the Vector database. Each vector is connected with vectors that are very similar as shown in image 8. Chunk 5 (V5) is connected with chunk 1,6, and 8.

When a **query** is given, the process begins by randomly selecting one chunk vector, also known as a node. For example, let's say the V6 node is chosen. The next step is to calculate the similarity score for this node.

From there, the process moves on to the nodes connected to V6, which are V5 and V2. Again, the similarity scores are calculated for these nodes. The node with the better similarity score is then selected, and its connected nodes are evaluated in the same way.

This process continues until no better similarity score can be found. In other words, the algorithm keeps exploring connected nodes until it reaches a point where the similarity scores no longer improve. The highest similarity score in this example is chunk 8 (V8).

Finally, the top k nodes from all the traveled nodes are selected, as illustrated in Image 9.

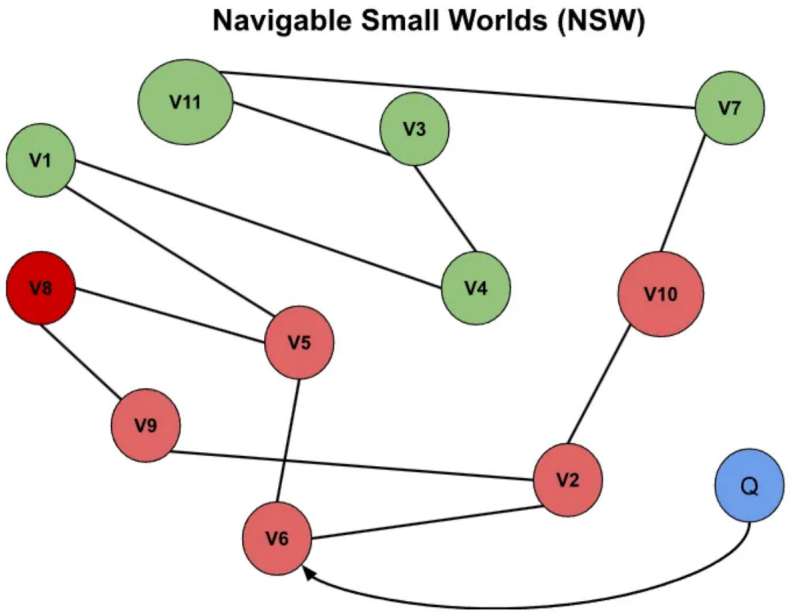


Image 9: NSW search. Image by author

This process repeated. We repeat the search with randomly chosen starting points and keep the top k among all the visited nodes. In the end, the Top K selected chunks are given to LLM to generate the augmented Response

selected chunks are given to LLM to generate the augmented response.

HNSW (Hierarchical Navigable small worlds) is the combination of Navigable small worlds and Skip list data structure.

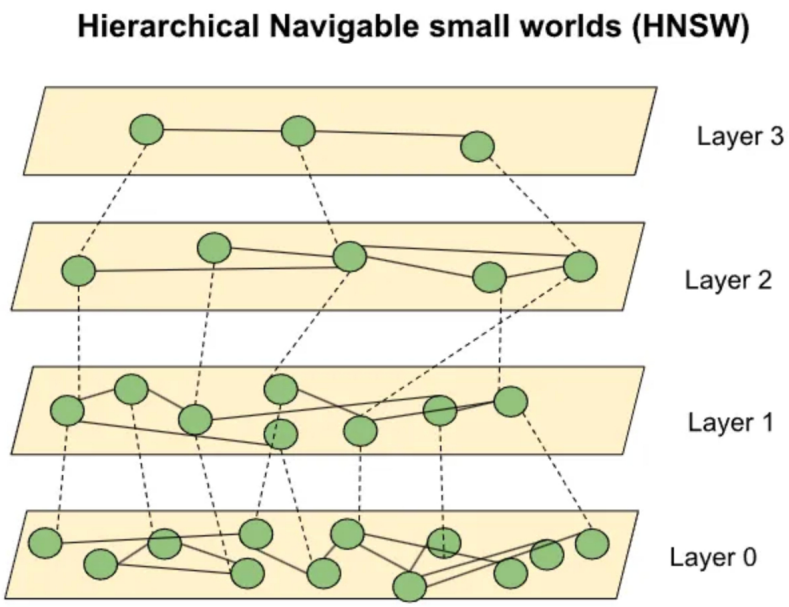


Image 10: HNSW. Image by author

In HNSW, there are multiple layers, The Top layer is sparse — a very minimum number of nodes will be there. The bottom layer is Dense.

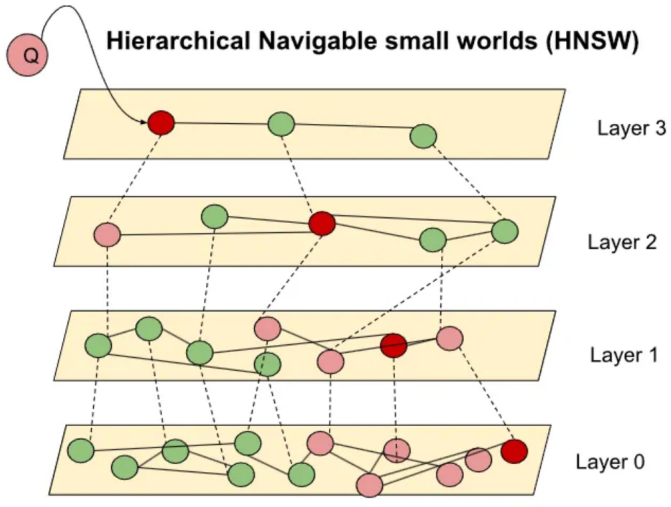


Image 11: HNSW search. Image by author

When the Query is given, It searches against a randomly chosen node in the Top Layer (Layer 3 — Sparse), calculates cosine similarity, and moves to its connected nodes. when it finds the Local best--high similarity scores compared to visited nodes (Each layer same NSW process), It Moves to the next layer (Layer 2).

In Layer 2 the cosine similarity is calculated to the node that is connected to the previous layer. Then the similarity scores are calculated for nodes that are connected, and when it finds the Local best, It moves to the Next Layer. This will happen for all layers. Then Top k nodes are selected from visited nodes.

We repeat the search with randomly chosen starting points in the top layer and keep the top k among all the visited nodes. This is how the Nodes (chunks) are efficiently retrieved.

4. Response generation in RAG

The Retrieved Top K Chunks are given to LLM with Prompts. Then, Using Retrieved chunks, Query, and system Prompt, the LLM augments the Input and Generate the response to the query.

```
# Code from one of my project based on RAG
    base_retriever = vector_index.as_retriever(similarity_top_k=similarity_top_n
    response = base_retriever.query(query)
```

From this code, you can understand that the query is searched against nodes in the vector index and retrieves the top k similarity nodes. The retrieved Nodes are given to LLM with Prompt and Query to generate the response.

```
# Example simple Prompt template
template = """<|system|>
              1. you are a Question answering system based AI, Machine Learning
                 Deep Learning , Generative AI, Data science, Data Analytics and
                 Mathematics.
              2. Mention Clearly Before response " RAG Output :\n".
              3. Please check if the following pieces of context has any mention
                 of the keywords provided in the question.Generate response as
                 much as you could with context you get.
              4. If the following pieces of Context does not relate to Question,
                 You must not answer on your own, you don't know the answer.
              </s>

              <|user|>
              Question:{query_str}</s>

              <|assistant|> """

Settings.llm = OpenAI(model="gpt-3.5-turbo-0125", temperature=0.1, model_kwargs=
                      max_tokens=512, system_prompt=template)
```

RAG has many practical applications, such as answering questions, serving as a personal assistant, generating content, and providing customer support. What makes RAG particularly effective is its ability to tap into **external knowledge**.

In summary, RAG is a powerful approach that combines the best of both worlds — retrieval-based methods and generative models. By pulling relevant information from a vast library of documents and using it to generate more accurate and informed responses, RAG outperforms traditional models that rely solely on generation without retrieval. I hope this article has helped clarify how RAG works and its benefits

. . .

Thanks for reading this article 🤩. If you found my article **useful** 👍, give it a **clap**👏😉! Feel free to follow for more insights.

Let's keep the conversation going! Feel free to connect with me on LinkedIn **www.linkedin.com/in/jaiganesan-n/** 🌏❤️

and join me on this exciting journey of exploring AI innovations and their potential to shape our world.

References:

[1] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Attention is all you Need (2017). Research Paper (Arxiv).

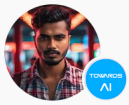
[2] The Ultimate Guide to Understanding Advanced Retrieval Augmented Generation Methodologies (2024), article.

[3] Nils Reimers, Iryna Gurevych, Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks(2019). Research Paper (Arxiv).

[4] Hierarchical Navigable Small World (HNSW) Pinecone, Blog

Retrieval Augmented Gen | Artificial Intelligence | Machine Learning | Deep Learning | Llm

63 2



Written by JAIGANESAN

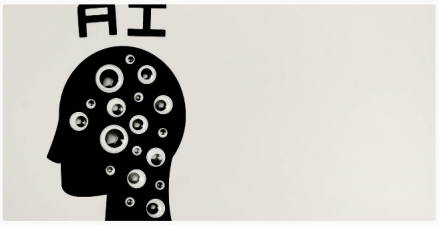
Written by JAIGANESAN

Follow

26 Followers · Writer for Towards AI

Loves sharing my thoughts and views on the world, AI, Books, Movies and geopolitics | 🤝
LinkedIn : https://www.linkedin.com/in/jaiganesan-n/

More from JAIGANESAN and Towards AI

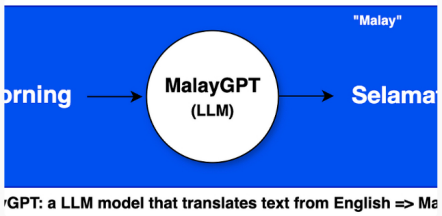


JAIGANESAN in Towards AI

Large Language Model (LLM)🤖: In and Out

Delving into the Architecture of LLM: Unraveling the Mechanics Behind Large...

May 25 28



Milan Tamang in Towards AI

Build your own Large Language Model (LLM) From Scratch Using...

A Step-by-Step guide to build and train an LLM named MalayGPT. This model's task is t...

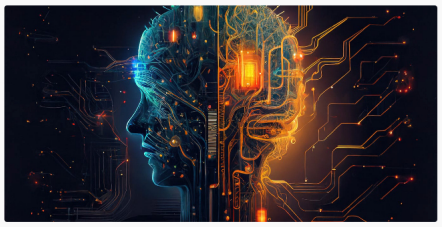
Jun 5 759 4

Mandar Karhade, MD. PhD. in Towards AI

Why RAG Applications Fail in Production

It worked as a prototype; then all went down!

Mar 19 1.7K 21



JAIGANESAN in Towards AI

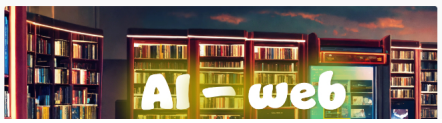
Breaking down Mistral 7B ⚡🍚

Exploring Mistral's Rotary positional Embedding, Sliding Window Attention, KV...

May 29 1

See all from JAIGANESAN | See all from Towards AI

Recommended from Medium



Carlos Jose Garces Rodrigues in AI Advances

4 Strategies to Optimize Retrieval-Augmented Generation

Using Private Data and Private Infrastructure for Enhanced AI Solutions

4d ago 305

Fabio Matricardi in Generative AI

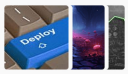
Make the Web your best friend data provider

Build your own up-to-date Document store and give your LLM new wings

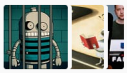
6d ago 406

Lists

Natural Language Processing
1541 stories · 1074 saves



Predictive Modeling w/ Python
20 stories · 1322 saves



AI Regulation
6 stories · 490 saves

Practical Guides to Machine Learning
10 stories · 1589 saves

Morgan Senechal in Neo4j Developer Blog

LLM Knowledge Graph Builder: From Zero to GraphRAG in Five...

Extract and Use Knowledge Graphs in Your GenAI App

6d ago 122

Justin Laughlin in Towards Data Science

Analyzing Unstructured PDF Data w/ Embedding Models and LLMs

How to turn PDFs into actionable insights

Jun 15 126 1

Ignacio de Gregorio

Mixture-of-Agents Beats ChatGPT-4o

Collaboration is Intelligence

4d ago 468 10

Walid Amamou in UBIAI NLP

LLM reinforcement learning: What is Essential in 2024 ?

In the swiftly evolving realm of artificial intelligence, Reinforcement Learning (RL)...

May 7 4 2

See more recommendations