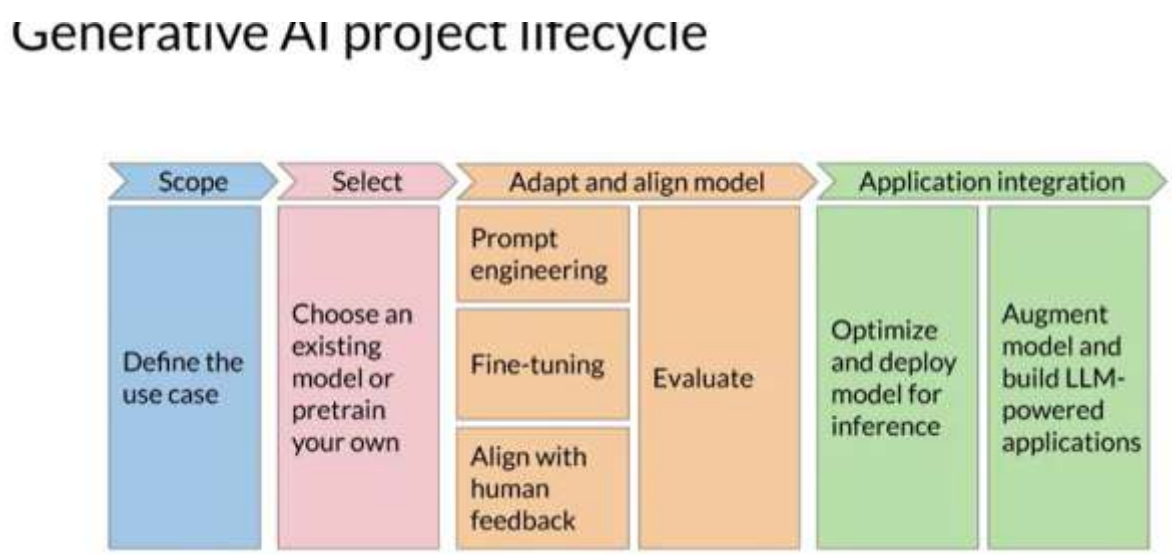
# INTRODUCTION TO GENERATIVE AI

***By Ch Waleed Ishtiaq***




**STEPS for Preparing Data:**

## Text preprocessing:

### 1. Tokenization

**Tokenization** is the process of breaking down text into smaller units called tokens. These tokens can be words, sentences, or subwords.

- **Word Tokenization**: Splits text into individual words.
  - Example: "The quick brown fox." -> ["The", "quick", "brown", "fox"]
- **Sentence Tokenization**: Splits text into individual sentences.
  - Example: "Hello world. How are you?" -> ["Hello world.", "How are you?"]
- **Subword Tokenization**: Splits text into subwords or morphemes, often used in neural network models.
  - Example: "unhappiness" -> ["un", "happiness"]

### 2. Stop Words Removal

**Stop words** are common words that carry little semantic meaning and are often removed to reduce the dimensionality of the data.

- **Example Stop Words**: "the," "is," "in," "and"
    - Example: "The quick brown fox jumps over the lazy dog." -> ["quick", "brown", "fox", "jumps", "lazy", "dog"]

**3. Stemming**

**Stemming** reduces words to their base or root form by chopping off the end of the words. The resulting stem may not be a valid word.

- **Example**: "Running," "runner," "runs" -> "run"

**4. Lemmatization** reduces words to their base or dictionary form, known as a lemma, considering the context and part of speech.

- **Example**: "Running" -> "run", "better" -> "good"

**Steps in Preprocessing**

1. **Text Normalization**: Converts text to a consistent format (e.g., lowercasing, removing punctuation).
    - Example: "Hello, World!" -> "hello world"
2. **Tokenization**: Splits text into smaller units (tokens).
    - Example: "hello world" -> ["hello", "world"]
3. **Stop Words Removal**: Removes common words that do not contribute significant meaning.
    - Example: ["hello", "world"] (assuming "hello" is not a stop word)
4. **Stemming or Lemmatization**: Reduces words to their base or root form.
    - Stemming Example: ["running", "runner"] -> ["run", "runner"]
    - Lemmatization Example: ["running", "better"] -> ["run", "good"]

**Example Workflow**

Given a sentence: "The quick brown foxes are running swiftly."

1. **Text Normalization**: "the quick brown foxes are running swiftly"
2. **Tokenization**: ["the", "quick", "brown", "foxes", "are", "running", "swiftly"]
3. **Stop Words Removal**: ["quick", "brown", "foxes", "running", "swiftly"]
4. **Stemming**: ["quick", "brown", "fox", "run", "swiftli"]
    - or **Lemmatization**: ["quick", "brown", "fox", "run", "swiftly"]

## Converting Words in to vectors:

**One-hot encoding** is a technique used to represent categorical variables as binary vectors. Each category is transformed into a vector with all elements set to 0 except for the position corresponding to the category, which is set to 1. This method allows categorical data to be used in machine learning algorithms that require numerical input. But it creates a Sparse Matrix and sentences are not fixed size

**Example:**

For the sentences:

- "I love NLP"
- "I love coding"

First, create a vocabulary of unique words: ["I", "love", "NLP", "coding"]

One-hot encoding for each word:

- "I" -> [1, 0, 0, 0]
- "love" -> [0, 1, 0, 0]
- "NLP" -> [0, 0, 1, 0]
- "coding" -> [0, 0, 0, 1]

To represent sentences using one-hot encoding, each sentence can be transformed into a set of one-hot vectors for each word:

- "I love NLP" -> [[1, 0, 0, 0], [0, 1, 0, 0], [0, 0, 1, 0]]
- "I love coding" -> [[1, 0, 0, 0], [0, 1, 0, 0], [0, 0, 0, 1]]

**Bag of Words**

Text representation technique in natural language processing where a text is represented as an unordered collection of words, disregarding grammar and word order but keeping multiplicity. It involves creating a vocabulary from all the unique words in the text and using it to model the text by indicating the frequency of each word's occurrence.

Preprocessing using the bag of words technique to convert words into vectors. Stop words are removed to focus on important vocabulary for sentiment analysis and classification tasks. Understanding the concept of **binary bag of words** involves converting words into numerical features based on frequency, but issues like sparsity and out of vocabulary problems can arise, impacting the meaning of sentences. Additionally, the ordering of words and calculating similarity using techniques like cosine similarity are crucial in text analysis.

**TF-IDF**

TF-IDF, which stands for Term Frequency-Inverse Document Frequency, is a numerical statistic used in information retrieval and text mining to evaluate the importance of a word in a document relative to a collection of documents (corpus).

- **Term Frequency (TF)**: Measures how frequently a term occurs in a document. It is often normalized to prevent bias towards longer documents.
- **Inverse Document Frequency (IDF)**: Measures how important a term is in the entire corpus. It decreases the weight of terms that appear very frequently across documents and increases the weight of terms that appear rarely.
- **TF-IDF**: The product of TF and IDF gives the TF-IDF score for a term within a document. This score indicates the importance of a term within a specific document, relative to the entire corpus. Higher TF-IDF values indicate higher importance of the term in the given document.

**Word2Vec, Doc2Vec etc are also used for converting text into vectors**

## Problems with LLMS

- No sources for answer
- Out dated

**How to deal with this problem**

We add a content store so LLM first goes to content store and finds the answers from our data store instead of giving its own answer

In normal cases, User prompts the model with its query and LLM gives answer. But when RAG is used generative model has an instruction to go and retrieve the relevant information combine it with user query and only then give the answer

**Sol to first problem**

- So now we have, Instruction to pay attention to, retrieved content with users question and now the model gives a response and give evidence for the answer which deals with our first challenge of no source

**Sol to second problem**

- Augmenting the data store with latest info helps us deal with the second problem

If the user question can not be answered reliably it should be able to tell that to user that **"I don't know the answer"**

If the retriever is not efficient, it will not be able to give ans to user query which is answerable. The retriever should be accurate enough to give the correct info to LLM so it can answer the user's query. Generator should be efficient too to give the richest answer

 **Retriever:**

- **Purpose:** The retriever's role is to fetch relevant documents or pieces of information from a large corpus based on the input query.
- **Common Models:** Dense Passage Retriever (DPR) is often used. DPR is trained to encode queries and documents into a shared embedding space where similar queries and documents are close together.
- **Mechanism:** The retriever uses similarity search (e.g., dot product similarity) to find the most relevant documents from the corpus.

 **Generator:**

- **Purpose:** The generator creates the final response or output, leveraging the retrieved documents to produce more informed and contextually relevant text.

- **Common Models:** Generative Pre-trained Transformer (GPT), BART, or T5 models are typically used.
- **Mechanism:** The generator takes the input query and the retrieved documents as inputs and generates a coherent and contextually enriched response.

**How RAG Works**

1. **Query Encoding:**
    - The input query is encoded into an embedding vector using the retriever model.
2. **Document Retrieval:**
    - The encoded query is used to retrieve the top-k relevant documents from a pre-indexed corpus. This is done using similarity search in the embedding space.
3. **Response Generation:**
    - The input query and the retrieved documents are passed to the generator model.
    - The generator combines the information from these documents to produce a final response that is both contextually relevant and informative

## Training RAG

- **Retriever Training:**
    - The retriever is typically trained using a contrastive learning approach where positive (relevant) and negative (irrelevant) document pairs are used to teach the model to distinguish relevant documents.
- **Generator Training:**
    - The generator is fine-tuned on a dataset where the inputs are queries paired with relevant documents, and the outputs are the desired responses. The retriever and generator can be trained jointly or separately.

## Creating Embeddings

Embeddings are generated using various techniques, depending on the type of data. Here are some common methods:

## Text Embeddings:

### Word Embedding

- **Word2Vec**: Transforms words into vectors based on their context within a corpus. Word2Vec is a popular model that learns distributed representations (embeddings) of words in a continuous vector space from large corpora of text. It uses either the Continuous Bag-of-Words (CBOW) or Skip-gram architecture to predict context

words given a target word or vice versa. Word similarity, language modelling, recommendation systems.

- **Average Word2Vec** is a technique where the word vectors for each word in a sentence or document are averaged to create a single vector representation for that sentence or document. This approach allows us to represent longer texts with a fixed-size vector, capturing the overall semantic meaning of the text. Here's how it's typically done:

  Convert each word to its Word2Vec vector.

  Sum all the vectors.

  Divide by the number of words to get the average vector.

- **GloVe** (Global Vectors for Word Representation): Captures the statistical information of word occurrences. GloVe is another widely used model for learning word embeddings. It leverages global word co-occurrence statistics to capture word semantics. It constructs an explicit word-context matrix and optimizes embeddings to preserve global semantic relationships. Word analogy tasks, document classification, sentiment analysis.

### Sentence Embedding

- **BERT** (Bidirectional Encoder Representations from Transformers): Generates **contextualized embeddings** for words and sentences. BERT is a Transformer-based model trained on large amounts of text to generate contextualized word and sentence embeddings. It pre-trains a deep bidirectional representation by jointly conditioning on both left and right context in all layers. Natural language understanding tasks, question answering, sentiment analysis. It uses only the ENCODER PART
- **TF-IDF** (Term Frequency-Inverse Document Frequency): Converts documents into vectors by considering the frequency of words and their importance.
- **Doc2Vec:** Doc2Vec (or Paragraph Vector) extends Word2Vec to generate embeddings for entire documents or paragraph. It considers document context to learn distributed representations, incorporating document-level semantics. Document clustering, text classification, recommendation systems.

## Image Embeddings:

- **Convolutional Neural Networks (CNNs**): Use deep learning models like ResNet or VGG to extract features from images and represent them as vectors.
- **Autoencoders:** Compress image data into a lower-dimensional vector space and then reconstruct it.

## Audio Embeddings:

**Mel-Frequency Cepstral Coefficients (MFCCs):** Extract features from audio signals.

**Recurrent Neural Networks (RNNs):** Capture temporal dependencies in audio data.

## Graph Embeddings:

**Node2Vec**: Transforms nodes in a graph into vectors based on their network structure.

**Graph Neural Networks (GNNs):** Capture the relationships and attributes of nodes and edges in a graph

## Processing of Data by different Models

Different embedding models process sentences in various ways, each leveraging unique techniques to capture semantic meaning and relationships within text data.

Embedded values remain same for a word in word2vec, GloVe. But in case of BERT embedding of word changes depending on context

.

## 1. Word2Vec

- **Model Type**: Word2Vec is a shallow neural network model typically trained on large text corpora.
- **Processing Approach**:
    - **Word-Level Embeddings**: Word2Vec generates embeddings at the word level. Each word in the vocabulary is assigned a fixed-size vector representation.
    - **Contextual Information**: Embeddings are learned based on the local context of each word (i.e., the surrounding words within a specific window size).

## 2. GloVe (Global Vectors for Word Representation)

- **Model Type**: GloVe is based on matrix factorization techniques applied to word co-occurrence statistics.
- **Processing Approach**:
    - **Global Co-occurrence Statistics**: GloVe constructs a global word-word co-occurrence matrix from the entire corpus. Embeddings are then learned by factorizing this matrix.
    - **Distributional Semantics**: Embeddings capture semantic relationships based on the statistical distribution of words across the corpus.

## Attention

## 3. BERT (Bidirectional Encoder Representations from Transformers)

- **Model Type**: BERT is a transformer-based model pre-trained on large-scale unlabeled text.
- **Processing Approach**:
    - **Token-Level Contextual Embeddings**: BERT generates contextual embeddings for each token in the input sequence. It considers bidirectional context (both left and right context) using self-attention mechanisms.

- **Self-attention** enables the model to weigh the importance of each word in a sequence relative to every other word in the same sequence. This mechanism captures relationships and dependencies between words, allowing the model to build richer, context-aware representations.
- **Masked Language Modelling (MLM)**: During pre-training, BERT predicts masked-out words in sentences to learn deep bidirectional representations.
- **Next Sentence Prediction (NSP)**: BERT also predicts whether two sentences follow each other in the corpus to capture relationships between sentences.

## 4. Doc2Vec (Paragraph Vector)

- **Model Type**: Doc2Vec extends Word2Vec to generate embeddings for entire documents, paragraphs, or sentences.
- **Processing Approach**:
  - **Document-Level Embeddings**: Doc2Vec learns a fixed-size vector representation for the entire document. It uses a similar approach to Word2Vec, but with additional parameters to capture document-level semantics.
  - **Paragraph ID**: Doc2Vec introduces a paragraph ID vector during training to differentiate between different paragraphs or documents.

## Differences in Processing Sentences:

- **Granularity**: Word2Vec and GloVe focus on word-level embeddings, capturing the meaning of individual words based on their local or global context.
- **Contextual Understanding**: BERT provides contextual embeddings at the token level, considering bidirectional context to capture nuanced meanings and relationships within sentences.
- **Document-Level Representation**: Doc2Vec generates embeddings for entire documents or paragraphs, providing a single vector representation that summarizes the semantic content of the entire text.

## Applications:

- **Word2Vec and GloVe**: Often used for tasks like word similarity, clustering, and downstream NLP tasks where static word embeddings suffice.
- **BERT**: Effective for tasks requiring understanding of context and semantics, such as sentiment analysis, question answering, and natural language inference.
- **Doc2Vec**: Useful for tasks involving document classification, recommendation systems, and information retrieval where document-level semantics are critical.

## Effect of Embeddings Dimension Size

## Benefits of Increasing Dimensions:

1. **Increased Representation Power**:
   - **Benefit**: Higher-dimensional embeddings can capture more nuanced relationships and semantic information between data points. This can lead to

improved accuracy in tasks like similarity search, recommendation systems, and natural language processing.
    - **Example**: In natural language processing, higher-dimensional word embeddings (e.g., GloVe, Word2Vec with 300+ dimensions) can encode richer semantic and syntactic information compared to lower-dimensional embeddings.
2. **Reduced Information Loss**:
    - **Benefit**: Higher-dimensional embeddings preserve more information about the original data points. This can be crucial in applications where fine-grained details are important, such as image and video retrieval.
    - **Example**: In image processing, higher-dimensional feature vectors can capture detailed visual characteristics, leading to better matching and retrieval of similar images.
3. **Better Differentiation**:
    - **Benefit**: Increasing dimensions can help distinguish between similar items or entities that have subtle differences. This is advantageous in applications requiring precise classification or clustering.
    - **Example**: In fraud detection, higher-dimensional embeddings can differentiate between legitimate and fraudulent transactions more accurately by capturing subtle patterns.
4. **Future-Proofing**:
    - **Benefit**: Larger dimensions provide more flexibility and future-proofing against evolving data and model requirements. They can accommodate new features or dimensions without needing significant re-engineering.
    - **Example**: In machine learning models, higher-dimensional embeddings can adapt to new types of input data or additional features without compromising performance.

## Considerations and Potential Losses:

1. **Increased Storage Requirements**:
    - **Consideration**: Higher-dimensional embeddings consume more storage space, which can become significant when dealing with large-scale datasets.
    - **Trade-off**: Balancing storage costs with performance requirements is essential, as storing and retrieving larger embeddings may require more computational resources.
2. **Computational Complexity**:
    - **Consideration**: Operations involving higher-dimensional embeddings, such as similarity search or clustering, may incur higher computational costs.
    - **Trade-off**: Optimizing algorithms and leveraging hardware acceleration (e.g., GPUs) can mitigate these costs, but it adds complexity to system design and maintenance.
3. **Dimensionality Curse**:
    - **Consideration**: As dimensions increase, the curse of dimensionality can lead to sparsity issues and increased computational inefficiency in high-dimensional spaces.
    - **Trade-off**: Techniques like dimensionality reduction (e.g., PCA, t-SNE) can help mitigate this by reducing the effective dimensions while preserving important information.
4. **Overfitting Risk**:

- **Consideration**: In machine learning applications, higher-dimensional embeddings can increase the risk of overfitting if not properly regularized or validated.
- **Trade-off**: Regularization techniques and cross-validation help mitigate overfitting risks by ensuring embeddings generalize well to unseen data.

## Vector DBs

Vector databases are specialized data storage and retrieval systems designed to handle and efficiently search high-dimensional vector data. They are commonly used in applications like similarity search, recommendation systems, and machine learning model retrieval tasks

**Characteristics of vector databases** - storing data as numerical vectors, scalability for large language models, and use of embeddings for data grouping.

- **Vector Storage**: Vector databases are designed to efficiently store and manage high-dimensional vectors representing complex data types such as images, text, and audio.
- **Vector Indexing:** These databases support indexing techniques optimized for vector data, enabling fast similarity searches and nearest neighbour queries.
- **Scalability:** Vector databases can scale horizontally to handle large volumes of vector data, making them suitable for applications with massive datasets.
- **Query Performance:** They are optimized for vector operations like similarity calculations, allowing for fast and efficient querying of complex data.

**Use cases of vector databases in AI** –

- semantic search
- similarity search
- chatbots utilizing natural language processing, image
- video recognition
- recommendation engines.

## Example of Vector DBs:

1. **FAISS (Facebook AI Similarity Search)**
   Features:
   - Efficient similarity search and clustering of dense vectors.
   - Supports large-scale datasets with billions of vectors.
   - Various indexing methods (e.g., Flat, IVFFlat, IVFPQ, HNSW).
   - GPU acceleration for faster computations.

   Use Cases:
   - Large-scale nearest neighbor search.
   - Image and text similarity search.
   - Recommender systems

2. **Pinecone**
   Features:

- Managed vector database service.
- Scalable and low-latency vector search.
- Integration with machine learning frameworks and data pipelines.
- Real-time updates and querying.

Use Cases:

- Product recommendations.
- Semantic search.
- Personalized content delivery.

3. **Qdrant**

Features:

- High-performance, open-source vector database.
- Supports hybrid queries combining vector and metadata filtering.
- Real-time vector indexing and search.
- Provides a RESTful API for easy integration.

Use Cases:

- Real-time recommendation systems.
- Image and text similarity search.
- Anomaly detection in time-series data

4. **Chroma**

Features:

- Open-source vector database optimized for embedding and retrieval.
- Supports hybrid search combining vector and attribute-based filtering.
- Simple and intuitive API for developers.
- Designed to integrate seamlessly with machine learning workflows.

Use Cases:

- Embedding-based search for ML applications.
- Content-based recommendation systems.
- Semantic search in documents and multimedia

5. **Reels:**

Features:

- Designed for time-series and high-dimensional data.
- Real-time indexing and querying capabilities.
- Efficient storage and retrieval of high-dimensional vectors.
- Integration with streaming data sources.

Use Cases:

- Real-time analytics on time-series data.
- Anomaly detection in streaming data.
- Predictive maintenance and monitoring.

6. **Weaviate**

Features:

- Open-source vector search engine.
- Supports context-aware semantic search.
- GraphQL API for flexible querying.
- Extensible with custom modules.

Use Cases:

- Knowledge graph integration.
- Contextual search.
- Machine learning model deployment.

7. **Vespa:**
   Features:
   - Open-source engine for large-scale data serving and processing.
   - Combines full-text search with vector search.
   - Real-time indexing and searching.
   - Supports complex ranking functions.

   Use Cases:
   - E-commerce product search.
   - Personalized content recommendations.
   - Real-time data processing.

## Similarity Search Space:

Similarity search in vector space involves finding vectors in a dataset that are most similar to a given query vector. This process is widely used in various applications such as recommendation systems, information retrieval, and machine learning.

## Steps in Similarity Search

1. **Data Representation**:
   - **Embedding**: Transform the data items (e.g., text, images, audio) into high-dimensional vectors using embedding techniques like Word2Vec, BERT for text, CNNs for images, or RNNs for audio.
2. **Indexing**:
   - **Index Structures**: Use data structures to organize the vectors for efficient retrieval. Common indexing methods include:
     - **KD-Trees**: Suitable for low-dimensional data.
     - **Ball Trees**: Efficient for higher-dimensional data.
     - **Locality-Sensitive Hashing (LSH)**: Works well for very high-dimensional data by hashing similar vectors into the same bucket.
     - **Inverted File Index (IVF)**: Partitions the vector space into clusters and indexes vectors within these clusters.
3. **Query Processing**:
   - **Query Embedding**: Convert the query item into its vector representation using the same embedding technique used for the dataset.
   - **Similarity Calculation**: Compute the similarity between the query vector and the dataset vectors using a similarity metric.
4. **Similarity Metrics**:
   - **Euclidean Distance**: Measures the straight-line distance between two vectors in Euclidean space.

    - **Cosine Similarity**: Measures the cosine of the angle between two vectors, useful for determining the orientation rather than magnitude.
    - **Manhattan Distance**: Sum of the absolute differences of their coordinates.
5. **Nearest Neighbor Search**:
    - **Exact Nearest Neighbors**: Searches the entire dataset to find the most similar vectors, which can be computationally expensive for large datasets.
    - **Approximate Nearest Neighbors (ANN)**: Uses algorithms like FAISS (Facebook AI Similarity Search), Annoy (Approximate Nearest Neighbors Oh Yeah), or HNSW (Hierarchical Navigable Small World) to quickly find approximate solutions that are close enough to the exact neighbors.
6. **Result Retrieval**:
    - **Ranking**: The vectors are ranked based on their similarity scores, and the top-k most similar vectors are selected and returned.

## Example Workflow

**Embedding Generation:**

Suppose we have a dataset of images. Each image is processed through a convolutional neural network (CNN) to generate a 512-dimensional vector embedding.

**Indexing:**

These embeddings are indexed using an indexing structure like IVF, which partitions the vector space into clusters.

**Query Processing:**

When a query image is received, it is processed through the same CNN to obtain its 512-dimensional vector embedding.

**Similarity Calculation:**

The query vector is compared with the indexed vectors using cosine similarity.

**Nearest Neighbor Search:**

Using an ANN algorithm, the database quickly identifies the top-10 vectors that are most similar to the query vector.

**Result Retrieval:**

The most similar images are retrieved and presented to the user.

## Context Window and Window Size

In the context of natural language processing (NLP) and machine learning, "context window" and "window size" are terms used to describe the span of tokens (words, subwords, or characters) that are considered when processing text. These concepts are essential in various NLP tasks, such as language modelling, word embeddings, and sequence-to-sequence models.

**Context Window**

1. A context window refers to the segment of text around a target word (or token) that is considered to understand or predict the target. This segment provides the contextual information necessary for the model to perform its task.
2. It limits how much of the recent conversation history the model can consider when generating a response
3. context window ensures the model can maintain context over short-to-medium length exchanges but doesn't inherently provide long-term memory across extended interactions.

- **Context Window Use**:
    - For immediate responses, the model uses the conversation history up to the context window size to generate coherent replies.
- **Long-Term Memory Use**:
    - For retaining information over multiple sessions, the chatbot could store user data in a database and fetch this information when the user interacts with the chatbot again, even if the previous interaction's context is no longer within the context window

### Applications:

**Word Embeddings (e.g., Word2Vec):**

- In Word2Vec, the context window is used to predict a word based on its neighbouring words (Skip-gram model) or to predict the neighbouring words based on the word (CBOW model).

**Language Models (e.g., GPT, BERT):**

- Transformer-based models like GPT uses only the Decoder part of transformer consider a context window of preceding tokens to predict the next token.
- BERT only uses encoder part of transformer considers a bidirectional context window, looking at both preceding and following tokens to understand the context for masked language modeling.

**Sequence-to-Sequence Models:**

- In models like RNNs, LSTMs, and GRUs, the context window can span the entire sequence, but practical implementations often limit it due to computational constraints.

### Window Size

The window size is the number of tokens that the context window spans. It determines how many tokens before and after the target word are included in the context window.

**Examples:**

**Word2Vec:**

- If the window size is 2, the context window for the word "fox" in the sentence "The quick brown fox jumps over the lazy dog" includes "quick", "brown", "jumps", and "over".

**Transformer Models:**

- Transformer models often process text in fixed-size chunks. The window size in this case refers to the maximum length of tokens considered in a single forward pass.
- For instance, GPT-3 has a context window size of 2048 tokens, meaning it can consider up to 2048 tokens when generating text.

**Text Processing:**

- In text processing, such as in moving average calculations over a sequence, the window size determines the number of adjacent elements considered for each computation.

## Importance of Context Window and Window Size

**Capturing Relevant Information:**

- The choice of window size impacts how much context is available for understanding the target word or token. A larger window size captures more context, which can be beneficial for understanding complex dependencies but also increases computational complexity.

**Balancing Performance and Computation:**

- There is a trade-off between performance and computational efficiency. Larger window sizes can improve performance but require more memory and processing power.

**Handling Long-Range Dependencies:**

- Models like Transformers handle long-range dependencies better due to their ability to consider long context windows. Traditional models like RNNs struggle with long dependencies unless techniques like attention mechanisms are used.

## Recurrent Neural Networks (RNNs)

**Overview**:

- RNNs are a type of neural network designed for sequential data, where the order of data points matters.
- Commonly used in time series analysis, natural language processing, and speech recognition.

**Key Characteristics**:

1. **Sequential Data Processing**: Maintains a hidden state to capture information from previous elements in the sequence.
2. **Recurrent Connections**: Information loops back from one time step to the next.
3. **Hidden State**: Updates based on the current input and previous hidden state.

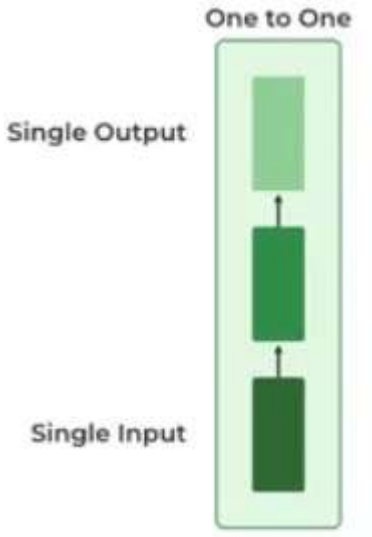
**Basic Structure**:

- **Input Layer**: Receives the sequence data.
- **Hidden Layer**: Updates at each time step using the current input and the previous hidden state.
- **Output Layer**: Produces the output based on the hidden state.

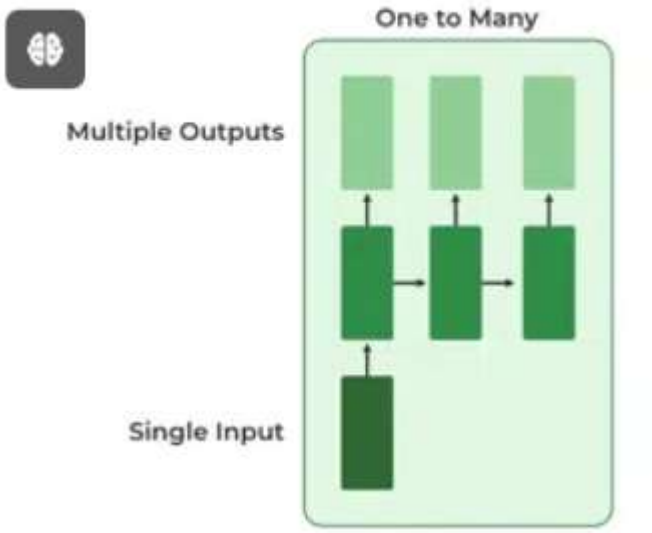
**Types Of RNN**

There are four types of RNNs based on the number of inputs and outputs in the network.

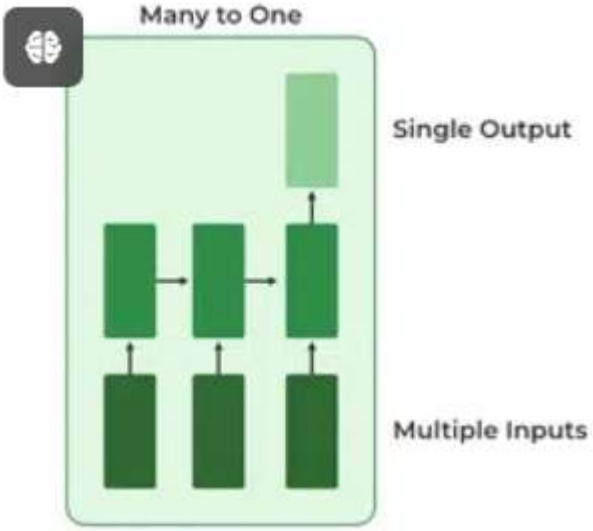
- **One to One:** This type of RNN behaves the same as any simple Neural network it is also known as Vanilla Neural Network. In this Neural network, there is only one input and one output.



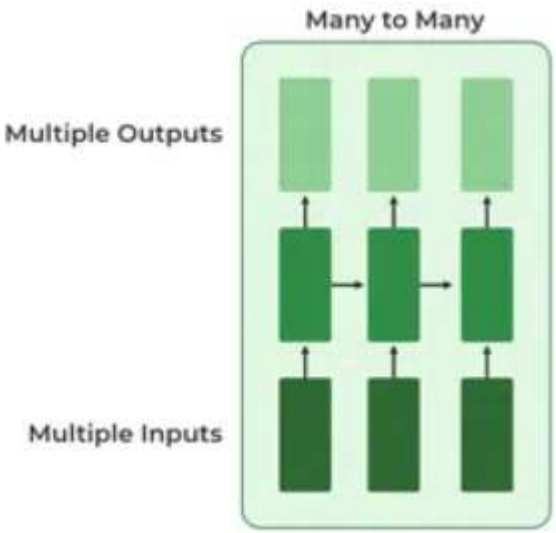

- **One to Many:** In this type of RNN, there is one input and many outputs associated with it. One of the most used examples of this network is Image captioning where given an image we predict a sentence having Multiple words.




- **Many to One** In this type of network, Many inputs are fed to the network at several states of the network generating only one output. This type of network is used in the problems like sentimental analysis. Where we give multiple words as input and predict only the sentiment of the sentence as output




- **Many to Many** In this type of neural network, there are multiple inputs and multiple outputs corresponding to a problem. One Example of this Problem will be language translation. In language translation, we provide multiple words from one language as input and predict multiple words from the second language as output




- 

**Variants**:

1. **LSTM (Long Short-Term Memory)**:
    - Addresses vanishing gradients with memory cells and gates (input, forget, output).
    - Captures long-term dependencies.
    - A traditional RNN has a single hidden state that is passed through time, which can make it difficult for the network to learn long-term dependencies. LSTMs model address this problem by introducing a memory cell, which is a container that can hold information for an extended period.
    - LSTM architectures are capable of learning long-term dependencies in sequential data, which makes them well-suited for tasks such as language translation, speech recognition, time series forecasting, recommender system, anomaly detection and video analysis

    **LSTM Architecture**

    The LSTM architectures involves the memory cell which is controlled by three gates: the input gate, the forget gate, and the output gate. These gates decide what information to add to, remove from, and output from the memory cell.

    - The input gate controls what information is added to the memory cell.
    - The forget gate controls what information is removed from the memory cell.
    - The output gate controls what information is output from the memory cell.

    This allows LSTM networks to selectively retain or discard information as it flows through the network, which allows them to learn long-term dependencies.

    The LSTM maintains a hidden state, which acts as the short-term memory of the network. The hidden state is updated based on the input, the previous hidden state, and the memory cell's current state.

**Bidirectional LSTM Model**

Bidirectional LSTM (Bi LSTM/ BLSTM) is recurrent neural network (RNN) that is able to process sequential data in both forward and backward directions. This allows Bi LSTM to learn longer-range dependencies in sequential data than traditional LSTMs, which can only process sequential data in one direction.

Bi LSTMs are made up of two LSTM networks, one that processes the input sequence in the forward direction and one that processes the input sequence in the backward direction.

The outputs of the two LSTM networks are then combined to produce the final output

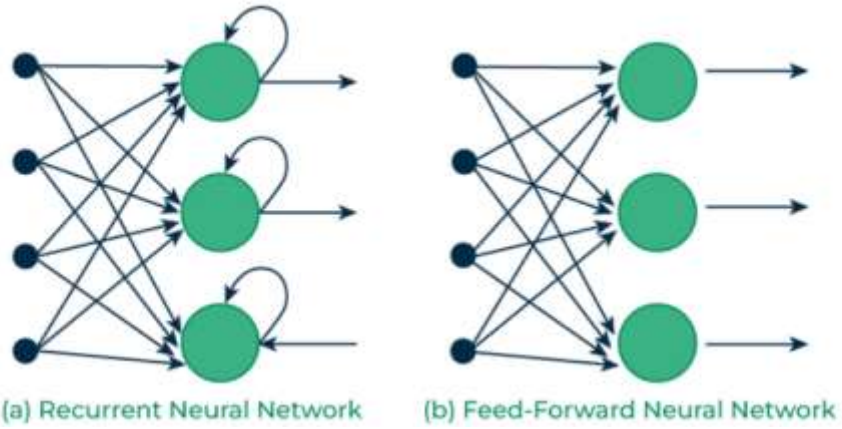
**LSTM**

- LSTM (Long Short-term Memory) has a special memory unit that allows it to learn long-term dependencies in sequential data.
- LSTM can be trained to process sequential data in both forward and backward directions.
- LSTM is more difficult to train due to the complexity of the gates and memory unit.
- LSTM can learn long-term dependencies.
- LSTM is used in applications such as machine translation, speech recognition, text summarization, natural language processing, and time series forecasting.

**RNN:**

- RNN (Recurrent Neural Network) does not have a memory unit.
- RNN can only be trained to process sequential data in one direction.
- RNN's ability to learn long-term dependencies is limited.
- Both LSTM and RNN can learn sequential data.
- RNN is used in natural language processing, machine translation, speech recognition, image processing, and video processing.

2. **GRU (Gated Recurrent Unit)**:
    - Simplified version of LSTM with fewer gates (reset, update).
    - More computationally efficient.
    - The reset gate determines how much of the previous hidden state should be forgotten, while the update gate determines how much of the new input should be used to update the hidden state. The output of the GRU is calculated based on the updated hidden state.




**Applications**:

- **NLP**: Language modeling, machine translation, sentiment analysis.
- **Speech Recognition**: Transcribing spoken language into text.
- **Time Series Prediction**: Forecasting future values based on past data.
- **Video Analysis**: Analyzing sequences of video frames.

**Example**:

- **Sentiment Analysis**: An RNN processes word embeddings from a movie review to predict sentiment.

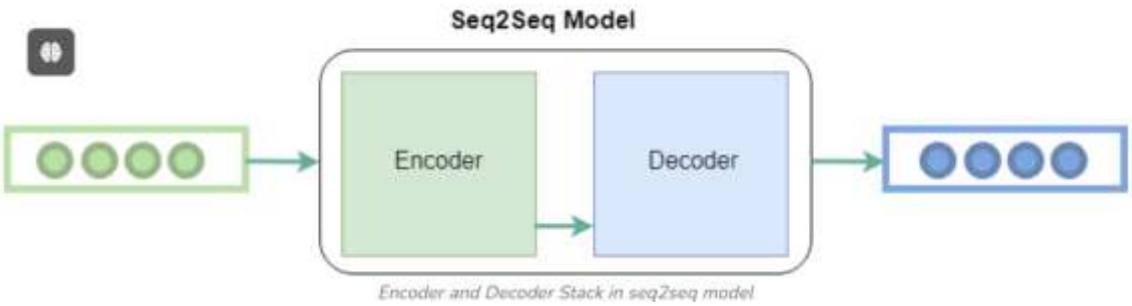
**Challenges**:

1. **Vanishing/Exploding Gradients**: Difficulty in learning long-term dependencies.
2. **Training Complexity**: Computationally intensive and requires careful hyperparameter tuning.

RNNs and their variants (LSTM, GRU) are powerful for sequential data tasks and have achieved state-of-the-art results in various applications

**SEQ2SEQ Models:**

- RNN is a type of neural network architecture designed to handle sequential data.
- Seq2seq is a model architecture that uses two RNNs:
- An encoder RNN to process the input sequence
- A decoder RNN to generate the output sequence
- Seq2seq builds upon RNNs to handle tasks where both input and output are sequences, potentially of different lengths.
- While traditional RNNs typically map sequences to single outputs, seq2seq allows mapping between sequences of different lengths.
- Seq2seq models often use more advanced RNN variants like LSTMs or GRUs to mitigate issues with long-term dependencies



*Encoder and Decoder Stack in seq2seq model*

Encoder Block:

- Processes the input sequence
- Captures information in a fixed-size context vector
- Uses neural networks or transformer architecture
- Maintains an internal state
- Produces a final hidden state (context vector) representing the entire input sequence

**Decoder Block:**

- Processes the context vector from the encoder
- Generates the output sequence incrementally
- During training, uses both context vector and target output sequence
- During inference, uses its own previous outputs as inputs
- Performs autoregressive generation, producing one element at a time
- At each step, uses current state, context vector, and previous output to predict the next token
- Continues until the end of the output sequence is reached

## HISTORY OF EVOLUTION OF THE MODELS:

**Artificial Neural Networks (ANNs) - 1940s-1950s**

- Needed to: Model basic brain functions and solve simple pattern recognition tasks.
- Limitation: Could only handle simple, linear problems.

**Multi-Layer Perceptrons (MLPs) - 1960s-1980s**

- Needed to: Solve non-linear problems that single-layer ANNs couldn't handle.
- Limitation: Struggled with sequential data and time-dependent patterns.

**Recurrent Neural Networks (RNNs) - 1980s-1990s**

- Needed to: Process sequential data and capture time-dependent patterns.

- Limitation: Suffered from vanishing/exploding gradient problems with long sequences.

**Long Short-Term Memory (LSTM) - 1997**

- Needed to: Address the vanishing gradient problem in RNNs and capture long-term dependencies.
- Limitation: Computationally expensive, complex architecture.

**Gated Recurrent Unit (GRU) - 2014**

- Needed to: Simplify LSTM while maintaining similar performance.
- Limitation: Still faced challenges with very long sequences.

**Sequence-to-Sequence (Seq2seq) Models - 2014**

- Needed to: Handle variable-length input and output sequences, particularly for machine translation.
- Limitation: Performance degraded with very long sequences.

**Attention Mechanism - 2015**

- Needed to: Improve seq2seq models by allowing them to focus on relevant parts of the input sequence.
- Limitation: Still relied on recurrent architecture, limiting parallelization.

**Transformer - 2017**

- Needed to: Eliminate recurrence and enable highly parallelizable sequence processing.
- Current state-of-the-art for many NLP tasks.

**Advantages of seq2seq Models**

- Flexibility: Seq2Seq models can handle a wide range of tasks such as machine translation, text summarization, and image captioning, as well as variable-length input and output sequences.
- Handling Sequential Data: Seq2Seq models are well-suited for tasks that involve sequential data such as natural language, speech, and time series data.
- Handling Context: The encoder-decoder architecture of Seq2Seq models allows the model to capture the context of the input sequence and use it to generate the output sequence.
- Attention Mechanism: Using attention mechanisms allows the model to focus on specific parts of the input sequence when generating the output, which can improve performance for long input sequences.

**Disadvantages of seq2seq Models**

- Computationally Expensive: Seq2Seq models require significant computational resources to train and can be difficult to optimize.
- Limited Interpretability: The internal workings of Seq2Seq models can be difficult to interpret, which can make it challenging to understand why the model is making certain decisions.
- Overfitting: Seq2Seq models can overfit the training data if they are not properly regularized, which can lead to poor performance on new data.
- Handling Rare Words: Seq2Seq models can have difficulty handling rare words that are not present in the training data.
- Handling Long input Sequences: Seq2Seq models can have difficulty handling input sequences that are very long, as the context vector may not be able to capture all the information in the input sequence.

### Applications of Seq2Seq model

We have discovered the machine translation is the real-world application of seq2seq model.

- Text Summarization: The seq2seq model effectively understands the input text which makes it suitable for news and document summarization.
- Speech Recognition: Seq2Seq model, especially those with attention mechanisms, excel in processing audio waveform for ASR. They are able to capture spoken language patterns effectively.
- Image Captioning: The seq2seq model integrate image features from CNNs with textual generation capabilities for image captioning. They are capable to describe images in a human readable format.

## Data Processed by different NEURAL NETWORKS

**Artificial Neural Networks (ANN):**

Text: Can process structured text data, typically in the form of numerical or categorical features.

Images: Can handle flattened image data, but not as efficiently as CNNs.

**Convolutional Neural Networks (CNN):**

Text: Can process text data, especially when converted to a grid-like structure (e.g., character-level encoding).

Images: Excel at processing 2D grid-like data, making them ideal for image analysis tasks.

**Recurrent Neural Networks (RNN):**

Text: Particularly well-suited for sequential text data, such as sentences, paragraphs, or time-series text data.

Images: Can process sequential image data, like video frames, but are not typically used for static image analysis

## Practical Considerations

**Choosing the Right Window Size:**

- The optimal window size depends on the specific application and dataset. For instance, short window sizes may suffice for tasks with local dependencies, while tasks requiring understanding of broader context benefit from larger window sizes.

**Memory and Computation Limits:**

- For large window sizes, especially in Transformer models, memory and computation become bottlenecks. Techniques like gradient checkpointing, sparse attention, and model parallelism can help mitigate these issues.

**Dynamic vs. Fixed Window Sizes:**

- Some models use fixed window sizes, while others adapt the window size dynamically based on the input. Dynamic window sizes can be more efficient but add complexity to the model.

## Similarity Metrics

## 1. Cosine Similarity

Cosine similarity measures the similarity between two phrases based on the angle between their word vectors. When phrases are identical, similarity is 1; when no words overlap, similarity is 0; partial overlap results in a similarity between 0 and 1.

Make a table for word counts, plot the points, find the angle, and similarity

For more than 2 words similarity, we use formula

**Useful For:**

- **Text Analysis and NLP**: Comparing text documents, word embeddings, and sentence embeddings.
- **Document Retrieval**: Finding similar documents or information retrieval.
- **High-Dimensional Data**: When the magnitude of the vectors is less important than their direction.

**Conditions:**

- When you need to measure the angle between vectors.
- When vector magnitude should not affect the similarity score.
- When dealing with sparse data (e.g., text data in a bag-of-words model).

## 2. Euclidean Distance

**Useful For:**

- **Clustering**: Algorithms like k-means clustering.
- **Image Processing**: Comparing image features.
- **Low-Dimensional Data**: When the data is not very high-dimensional.

**Conditions:**

- When the actual distance between points is meaningful.
- When the data is continuous and differences in magnitude are important.
- When the data dimensionality is not too high (curse of dimensionality).

## 3. Manhattan Distance (L1 Distance)

**Useful For:**

- **Sparse Data**: Data with many zero values.
- **Grid-Based Problems**: Pathfinding algorithms like A* in grid environments.
- **Robustness to Outliers**: Less sensitive to outliers compared to Euclidean distance.

**Conditions:**

- When dealing with high-dimensional data where individual differences are meaningful.
- When the data is sparse and differences should be aggregated in a linear fashion.
- When robustness to outliers is required.

## 4. Jaccard Similarity

**Useful For:**

- **Binary Data**: Comparing binary vectors.
- **Set Comparisons**: Comparing sets of elements, such as user preferences or hashtags.
- **Text Clustering**: Comparing documents represented as sets of words.

**Conditions:**

- When the data can be represented as sets (e.g., presence or absence of features).
- When you need to measure the overlap between sets.
- When dealing with categorical data.

## 5. Pearson Correlation Coefficient

**Useful For:**

- **Statistics and Data Analysis**: Measuring the linear relationship between variables.
- **Recommender Systems**: Collaborative filtering to find similar users or items.
- **Financial Analysis**: Analyzing the relationship between stock prices or economic indicators.

**Conditions:**

- When the relationship between variables is linear.
- When you need to measure correlation, not similarity.
- When the data is continuous and normally distributed.

## 6. Hamming Distance

**Useful For:**

- **Error Detection and Correction**: Coding theory and data transmission.
- **Binary Data**: Comparing binary strings or bit vectors.
- **DNA Sequencing**: Comparing sequences of nucleotides.

**Conditions:**

- When the data is binary or categorical.
- When measuring the exact number of differing positions.
- When dealing with fixed-length strings or sequences.

## 7. KL Divergence (Kullback-Leibler Divergence)

**Useful For:**

- **Probability Distributions**: Comparing probability distributions.
- **Machine Learning**: Training generative models like VAEs (Variational Autoencoders).
- **Information Theory**: Measuring the divergence between two distributions.

**Conditions:**

- When comparing probability distributions.
- When the distributions are discrete or continuous.
- When you need to measure the information loss between distributions.

## Key Components of Transformers

1. **Self-Attention Mechanism**:
    - **Core Innovation**: Transformers rely heavily on self-attention mechanisms, which allow them to weigh the importance of each word/token in a sequence relative to every other word/token. This enables them to capture relationships and dependencies between words efficiently.
2. **Multi-Head Attention**:
    - **Parallel Processing**: Transformers use multiple attention heads in each layer to capture different types of relationships in parallel. Each attention head computes attention scores independently and then concatenates their outputs, allowing the model to focus on different aspects of the input simultaneously.
3. **Positional Encoding**:
    - **Capturing Sequence Order**: Since transformers do not inherently understand the order of words in a sequence like recurrent neural networks (RNNs) or convolutional neural networks (CNNs), positional encodings are added to the input embeddings. These encodings provide information about the position of each token in the sequence, enabling the model to account for sequential order.

4. **Feedforward Neural Networks**:
    - **Non-linear Transformations**: After self-attention layers, transformers use feedforward neural networks (FFNNs) to process the information captured by the attention mechanism. FFNNs consist of multiple layers of fully connected networks with activation functions like ReLU (Rectified Linear Unit).
5. **Layer Normalization**:
    - **Stabilizing Training**: Transformers use layer normalization to stabilize the learning process by normalizing the outputs of each layer and applying a scaling and shifting transformation.

## Transformer Architecture

- **Encoder-Decoder Architecture**: Transformers are commonly used in a dual-architecture setup:
    - **Encoder**: Processes the input sequence and generates a contextualized representation for each token.
    - **Decoder**: Takes the encoder's outputs and generates the output sequence (e.g., in machine translation tasks).
- **Stacked Layers**: Transformers consist of multiple layers of encoders (and optionally decoders), each comprising self-attention mechanisms, FFNNs, and normalization layers. The number of layers can vary depending on the complexity of the task and the size of the dataset.

## Applications of Transformers

- **NLP Tasks**: Transformers have been successfully applied to various NLP tasks, including:
    - Machine Translation (e.g., Google's Transformer model for translation).
    - Text Generation (e.g., OpenAI's GPT models).
    - Named Entity Recognition.
    - Sentiment Analysis.
    - Question Answering (e.g., BERT for QA tasks).

## Advantages of Transformers

- **Parallelization**: Transformers can process tokens in parallel due to their attention mechanisms, making them faster than sequential models like RNNs.
- **Long-Range Dependencies**: They can capture relationships between tokens that are far apart in the input sequence, which is challenging for traditional sequential models.
- **Scalability**: Transformers can scale to large datasets and compute clusters, enabling training on extensive corpora with billions of tokens.

## Limitations

- **Computational Resources**: Training large transformer models requires substantial computational resources (e.g., GPUs or TPUs).
- **Interpretability**: Despite their effectiveness, understanding how transformers arrive at specific predictions can be challenging due to their complex architectures.

## Tokenization:

Tokenization is the process of breaking down text into smaller units called tokens. These tokens can be words, subwords, or characters, depending on the specific tokenization strategy used. Tokenization is a fundamental preprocessing step in natural language processing (NLP) and is crucial for converting text into a format that can be processed by machine learning models.

## Types of Tokenization

1. **Word Tokenization**
    - **Definition**: Splitting text into individual words.
    - **Example**: The sentence "I love Pakistan" would be tokenized into ["I", "love", "Pakistan"].
    - **Use Cases**: Basic text processing tasks, where each word is treated as a distinct unit.
2. **Subword Tokenization**
    - **Definition**: Splitting words into smaller units called subwords or morphemes.
    - **Example**: The word "unhappiness" might be tokenized into ["un", "happiness"] or ["un", "happy", "ness"].
    - **Common Algorithms**:
        - **Byte Pair Encoding (BPE)**: Combines the most frequent pairs of bytes or characters iteratively.
        - **WordPiece**: **(freq of pair/ freq of first element* freq of second)** Similar to BPE, used in models like BERT. If the word exist in vocabulary in **bert uncased it comes as it is otherwise it is divided and pairs are formed**
        - **SentencePiece**: A more flexible subword tokenizer used in models like T5 and GPT-3.
    - **Use Cases**: Handling out-of-vocabulary words and languages with rich morphology.
3. **Character Tokenization**
    - **Definition**: Splitting text into individual characters.
    - **Example**: The word "hello" would be tokenized into ["h", "e", "l", "l", "o"].
    - **Use Cases**: Low-level text analysis, certain sequence modeling tasks, and languages with complex character sets (e.g., Chinese).

## Tokenization in Different NLP Models

- **Traditional NLP Models**:
    - Use simple tokenization techniques like word or character tokenization.
    - Tokenization methods include splitting by spaces or punctuation.
- **Modern NLP Models**:
    - **Word2Vec**: Uses word tokenization.
    - **GloVe**: Uses word tokenization.
    - **FastText**: Uses word tokenization but also includes subword information.
    - **BERT**: Uses WordPiece tokenization.
    - **GPT**: Uses Byte Pair Encoding (BPE) tokenization.
    - **T5**: Uses SentencePiece tokenization.

## Steps in Tokenization

1. **Text Normalization**:
    - Converting text to a consistent format (e.g., lowercasing, removing punctuation).
    - Handling contractions (e.g., "don't" to "do not").
2. **Token Splitting**:
    - Splitting text into tokens based on spaces, punctuation, or specific algorithms for subword tokenization.
3. **Token Mapping**:
    - Mapping tokens to numerical values (token IDs) using a vocabulary.
    - This step converts text into a sequence of integers that models can process.
    - Each token ID corresponds to a unique embedding vector in an embedding matrix. These embeddings capture semantic relationships between tokens based on their context in the training data. Embeddings enable the model to understand similarities, differences, and associations between different words or subword units.

## Example

Consider the sentence "I love Pakistan":

- **Word Tokenization**:
    - Tokens: ["I", "love", "Pakistan"]
    - Token IDs (assuming a simple vocabulary): [1, 2, 3]
- **Subword Tokenization (BERT)**:
    - Tokens: ["I", "love", "Pak", "##istan"]
    - Token IDs (assuming a BERT vocabulary): [101, 102, 103, 104]
    - ## use because istan is continuation wo word Pak

## Challenges in Tokenization

1. **Ambiguity**:
    - Words with multiple meanings (e.g., "bank" as a financial institution or riverbank).
    - Contextual usage can influence the meaning and tokenization.
2. **Out-of-Vocabulary Words**:
    - Words not present in the model's vocabulary need to be handled (e.g., via subword tokenization).
3. **Language-Specific Issues**:
    - Different languages have different tokenization challenges (e.g., compound words in German, lack of spaces in Chinese).

## Benefits of Tokenization

- **Preprocessing**: Converts raw text into a structured format for models.
- **Vocabulary Building**: Helps in creating a manageable vocabulary for language models.

- **Feature Extraction**: Enables extraction of meaningful features from text for analysis and modeling.

**WordPiece and Byte Pair Encoding (BPE)** are both subword tokenization methods used to handle out-of-vocabulary words and manage vocabulary size in natural language processing tasks. While they share some similarities, they have distinct differences in how they operate and their specific implementations. Here's a comparison between the two:

## Byte Pair Encoding (BPE)

**Overview**:

- BPE iteratively merges the most frequent pairs of characters or subwords in a corpus to create a fixed-size vocabulary of subwords.
- Originally developed for data compression, but adapted for NLP tasks.

**Process**:

1. **Initialization**: Start with all individual characters as the initial vocabulary.
2. **Frequency Calculation**: Count the frequencies of all pairs of adjacent symbols (characters or subwords).
3. **Pair Merging**: Find the most frequent pair and merge it into a new subword.
4. **Update Vocabulary**: Add the new subword to the vocabulary.
5. **Repeat**: Continue the process until reaching the desired vocabulary size.

**Example**:

- Corpus: "lower", "lowest"
- Initial tokens: l o w e r, l o w e s t
- Merging steps:
    1. Merge "l o" → "lo": lo w e r, lo w e s t
    2. Merge "lo w" → "low": low e r, low e s t
    3. Continue until the desired vocabulary size.

## WordPiece

**Overview**:

- WordPiece, used in models like BERT, also iteratively merges characters or subwords based on their frequency, but it uses a more sophisticated approach to determining which pairs to merge.

**Process**:

1. **Initialization**: Start with a basic vocabulary containing all individual characters and some common words.
2. **Frequency Calculation**: Calculate the likelihood of pairs of subwords.

3. **Pair Merging**: Instead of simply merging the most frequent pairs, WordPiece takes into account the probability of subwords given the context, aiming to maximize the overall likelihood of the corpus.
4. **Update Vocabulary**: Add the new subword to the vocabulary.
5. **Repeat**: Continue the process until reaching the desired vocabulary size.

**Example**:

- Corpus: "unaffordable", "unfashionable"
- Initial tokens: u n a f f o r d a b l e, u n f a s h i o n a b l e
- Merging steps:
    1. Merge "un" → "un": un a f f o r d a b l e, un f a s h i o n a b l e
    2. Merge "aff" → "aff": un aff o r d a b l e, un f a s h i o n a b l e
    3. Continue until the desired vocabulary size.

## Key Differences

1. **Merging Criteria**:
    - **BPE**: Merges the most frequent pairs of characters or subwords in the entire corpus.
    - **WordPiece**: Merges pairs based on the probability of subwords using **expectation maximization model** (algorithm helps in estimating the likelihood or probability of different subword units occurring together in the training corp), aiming to maximize the likelihood of the corpus, considering context more effectively. **EM considers both the frequency of occurrence and the contextual probability of subword sequences.**

2. **Use in Models**:
    - **BPE**: Used in models like GPT-2 and some earlier transformer models.
    - **WordPiece**: Used in BERT and related transformer models.
3. **Optimization**:
    - **BPE**: Focuses on frequency-based merging, which is simpler and faster to compute.
    - **WordPiece**: Incorporates likelihood and context, providing potentially better handling of subwords but with increased computational complexity.
4. **Handling Out-of-Vocabulary Words**:
    - Both methods break down rare or unknown words into known subwords, but WordPiece tends to be more context-sensitive.

## Summary

- **Byte Pair Encoding (BPE)**:
    - Simpler and faster.
    - Frequency-based merging.
    - Used in models like GPT-2.
- **WordPiece**:
    - More sophisticated and context-aware.
    - Likelihood-based merging.

- Used in models like BERT
-

 **BPE**: Typically results in subword units that are frequent but may not always be semantically meaningful.

 **WordPiece**: Tends to produce subword units that are more semantically meaningful because of the probabilistic modeling that considers the likelihood of subword sequences within the data.

## Steps of the EM Algorithm

1. **Expectation (E) Step**:
   - **Objective**: In this step, the algorithm calculates the expected value (expectation) of the latent variables(unobserved) given the observed data and the current estimates of model parameters.
   - **Calculations**: It computes the posterior probabilities of the latent variables given the observed data using the current parameter estimates.
2. **Maximization (M) Step**:
   - **Objective**: In this step, the algorithm updates the parameters of the model to maximize the likelihood function, based on the expected values obtained from the E-step.
   - **Calculations**: It computes new parameter estimates that maximize the likelihood function, typically using techniques like gradient ascent or iterative methods.

## Key Differences in Word and Token Embedding

1. **Context Sensitivity**:
   - **Word Embeddings**: Static, context-independent.
   - **Token Embeddings**: Dynamic, context-dependent.
2. **Granularity**:
   - **Word Embeddings**: Typically represent whole words.
   - **Token Embeddings**: Can represent subwords or even characters, allowing finer granularity.
3. **Vocabulary Handling**:
   - **Word Embeddings**: Fixed vocabulary; struggles with out-of-vocabulary words.
   - **Token Embeddings**: Subword tokenization helps handle out-of-vocabulary words by breaking them down into known tokens.
4. **Pre-training and Fine-tuning**:
   - **Word Embeddings**: Often pre-trained separately and then used in downstream tasks.
   - **Token Embeddings**: Typically part of pre-trained models like BERT and are fine-tuned as part of these models for specific tasks.

## Advantages of Token Embeddings

1. **Contextual Understanding**:

    - **Word Embeddings**: Provide a single, context-independent vector for each word. For example, the word "bank" has the same embedding whether it appears in "river bank" or "financial bank."
    - **Token Embeddings**: Generate embeddings that are context-sensitive. The same word can have different embeddings based on the surrounding words, allowing models to understand different meanings of words in different contexts.
2. **Handling Out-of-Vocabulary Words**:
    - **Word Embeddings**: Struggle with out-of-vocabulary (OOV) words, as these words do not have pre-trained embeddings.
    - **Token Embeddings**: Often use subword units (like WordPiece, Byte Pair Encoding, or SentencePiece), which break down rare or unknown words into smaller, known components. This approach helps in handling OOV words by constructing embeddings from these subword units.
3. **Finer Granularity**:
    - **Word Embeddings**: Treat each word as a single unit, which can be limiting for languages with rich morphology or for dealing with misspellings and compound words.
    - **Token Embeddings**: By operating at the subword level, token embeddings provide finer granularity, enabling better handling of prefixes, suffixes, and other morphological variations.
4. **Improved Performance in Downstream Tasks**:
    - **Word Embeddings**: Are useful but often require additional features or complex architectures to achieve high performance on specific tasks.
    - **Token Embeddings**: As part of pre-trained language models like BERT, they capture deep linguistic features and can be fine-tuned on specific tasks, leading to superior performance on a wide range of NLP tasks such as text classification, named entity recognition, and machine translation.
5. **Efficient Use of Vocabulary**:
    - **Word Embeddings**: Require a large vocabulary to cover all possible words, leading to significant memory usage and computational costs.
    - **Token Embeddings**: Subword tokenization allows for a more compact and efficient vocabulary. This compact vocabulary can handle a vast number of words with fewer base units, reducing memory and computational requirements.

## Example Scenarios

1. **Polysemy and Homonymy**:
    - **Word Embeddings**: Cannot distinguish between different meanings of the same word.
    - **Token Embeddings**: Provide different vectors based on context, thus differentiating between multiple meanings of the same word.
2. **Compound Words and Variations**:
    - **Word Embeddings**: Need a separate vector for each compound word and variation.
    - **Token Embeddings**: Can break down compound words into meaningful subwords, efficiently representing variations.
3. **Language Morphology**:

- **Word Embeddings**: May require extensive preprocessing to handle different forms of words.
- **Token Embeddings**: Naturally handle different morphological forms through subword tokenization.

**In context of text generation, Token embedding is very powerful comparatively, in simpler tasks like text classification (sentiment analysis, spam detection), document classification, Resource limited environment, Fixed vocabulary we need Word Embedding**

o embed token IDs like the ones you've provided into dense vectors, typically in the context of neural networks and natural language processing tasks, we use embedding layers. Let's break down how this process works:

## Tokenization and Numerical Representation

1. **Tokenization**:
   - The original sentence "I love natural language processing gnissecorp gnissecorp" is tokenized into individual tokens. These tokens are then converted into their respective token IDs using a tokenizer.
   - **Example**: Tokens and corresponding token IDs:

     Tokens: ['i', 'love', 'natural', 'language', 'processing', 'g', '##nis', '##se', '##corp', 'g', '##nis', '##se', '##corp']
     Token IDs: [1045, 2293, 3019, 2653, 6364, 1043, 8977, 3366, 24586, 1043, 8977, 3366, 24586]

2. **Embedding Layer**:
   - In a neural network architecture designed for natural language processing tasks (like sentiment analysis, machine translation, etc.), an embedding layer is typically used as the first layer after tokenization.
   - **Initialization**: Initially, the embedding layer's parameters are randomly initialized. Each token ID is mapped to a dense vector (embedding vector) of a specified dimensionality (e.g., 100, 200, etc.).
   - **Learning**: During the training process, these embedding vectors are adjusted (learned) based on the task-specific objective (e.g., minimizing loss in sentiment prediction).

## Example of Embedding Process

Let's illustrate how embedding might work for the token IDs provided:

- Suppose we have an embedding layer with an embedding dimension of 4 for simplicity. This means each token ID will be mapped to a dense vector of size 4.
- Initially, the embedding layer might assign random vectors to each token ID:

  Token ID 1045 (token: 'i') -> [0.1, -0.3, 0.2, 0.5]

Token ID 2293 (token: 'love') -> [-0.4, 0.1, -0.7, 0.3]
Token ID 3019 (token: 'natural') -> [0.6, -0.2, 0.9, -0.5]
Token ID 2653 (token: 'language') -> [0.8, 0.4, -0.1, 0.2]
Token ID 6364 (token: 'processing') -> [-0.3, 0.7, -0.4, 0.6]
Token ID 1043 (token: 'g') -> [0.2, -0.6, 0.3, -0.8]
Token ID 8977 (token: '##nis') -> [0.5, 0.2, -0.9, 0.1]
Token ID 3366 (token: '##se') -> [-0.7, 0.8, 0.4, -0.3]
Token ID 24586 (token: '##corp') -> [0.3, -0.5, 0.6, -0.4]

- During training, these vectors are updated through backpropagation and gradient descent, optimizing them to better represent relationships between tokens that are useful for the sentiment analysis task.

## Importance of Embedding

- **Semantic Relationships**: Embedding vectors capture semantic relationships between tokens based on their context in the training data.
- **Efficiency**: Dense embeddings are more efficient than sparse representations (like one-hot encoding) and allow the model to generalize better to unseen data.
- **Contextual Understanding**: Models can learn nuanced meanings of tokens by adjusting embeddings based on the task's objective function (e.g., maximizing sentiment prediction accuracy).

In summary, embedding token IDs involves mapping each token ID to a dense vector representation through an embedding layer in a neural network. These embeddings are learned during training to capture meaningful relationships between tokens, ultimately improving the model's performance on tasks like sentiment analysis.

**Example Sentence**: "I really liked the movie, it was fantastic!"

## Tokenization and Embedding

1. **Tokenization**:
    - The sentence is tokenized into individual tokens. Let's assume the tokens are: ["I", "really", "liked", "the", "movie", ",", "it", "was", "fantastic", "!"]
2. **Token to ID Conversion**:
    - Each token is mapped to a numerical token ID based on a predefined vocabulary. For example:
        - "I" -> 100
        - "really" -> 200
        - "liked" -> 300
        - "the" -> 400
        - "movie" -> 500
        - "," -> 600
        - "it" -> 700
        - "was" -> 800
        - "fantastic" -> 900
        - "!" -> 1000
3. **Embedding Lookup**:
    - Each token ID is then embedded into a continuous vector space using an embedding matrix. For simplicity, let's assume each token is embedded into a

3-dimensional vector (this is just for illustration purposes; in practice, embedding dimensions are much larger):

- "I" -> [0.1, 0.2, 0.3]
- "really" -> [0.4, 0.5, 0.6]
- "liked" -> [0.7, 0.8, 0.9]
- ...
- "!" -> [1.1, 1.2, 1.3]

## Updating Embedded Vectors

During training, the embedded vectors of these tokens are updated through the following steps:

1. **Forward Pass**:
   - The embedded vectors of tokens "I", "really", "liked", "the", "movie", ",", "it", "was", "fantastic", "!" are fed into the model.
2. **Loss Calculation**:
   - The model computes a predicted sentiment score based on these embeddings.
   - Suppose the ground truth sentiment label for this sentence is positive.
3. **Backpropagation**:
   - **Gradient Calculation**: Gradients of the loss function (e.g., cross-entropy loss) with respect to all model parameters, including the embedding vectors, are computed.
   - **Embedding Gradients**: Specifically, gradients with respect to each token's embedding vector indicate how much each vector should be adjusted to minimize the prediction error.
4. **Parameter Update**:
   - **Learning Rate**: Determines the step size for updating the parameters.
   - **Update Rule**: Each embedding vector is adjusted according to its gradient and the learning rate:
     - Example update rule (simplified):
       new embedding vector=old embedding vector−η⋅∇embedding vectorL\text{new embedding vector} = \text{old embedding vector} - \eta \cdot \nabla_{\text{embedding vector}}
       \mathcal{L}new embedding vector=old embedding vector−η⋅∇embedding vectorL
     - η\etaη: Learning rate.
     - L\mathcal{L}L: Loss function.
5. **Iteration**:
   - This process repeats over multiple iterations (epochs) with different sentences, gradually improving the embedding vectors to better represent sentiment-related features and improve the model's accuracy on sentiment analysis tasks.

## Example Update (Illustrative)

Suppose during backpropagation, the gradient for the token "liked" is calculated as ∇liked=[0.2,−0.3,0.1]\nabla_{\text{liked}} = [0.2, -0.3, 0.1]∇liked=[0.2,−0.3,0.1].

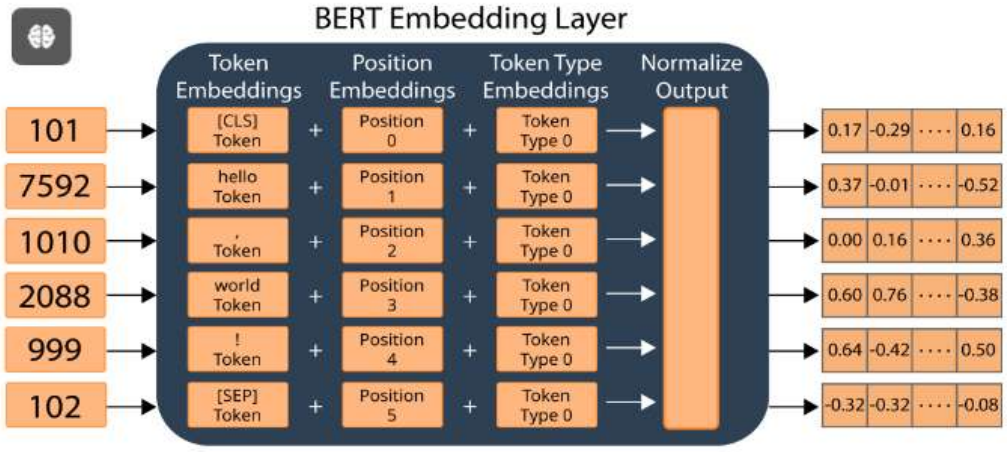
- If the learning rate η=0.01\eta = 0.01η=0.01, the update to the embedding vector for "liked" could be:

- new embedding vector for "liked"=[0.7,0.8,0.9]−0.01·[0.2,−0.3,0.1]=[0.7−0.002,0.8+0.003,0.9−0.001]=[0.698,0.803,0.899]\text{new embedding vector for "liked"} = [0.7, 0.8, 0.9] - 0.01 \cdot [0.2, -0.3, 0.1] = [0.7 - 0.002, 0.8 + 0.003, 0.9 - 0.001] = [0.698, 0.803, 0.899]new embedding vector for "liked"=[0.7,0.8,0.9]−0.01·[0.2,−0.3,0.1]=[0.7−0.002,0.8+0.003,0.9−0.001]=[0.698,0.803,0.899]

## Conclusion

In summary, token embeddings are updated during training by adjusting their vector representations based on gradients derived from the model's prediction error (loss) with respect to the ground truth labels. This iterative process ensures that the embeddings capture meaningful relationships and semantic features relevant to sentiment analysis, ultimately improving the model's ability to accurately predict sentiment from text data.

 **Tokenization and Numerical Assignment**: Input text is tokenized and each token is assigned a unique numerical ID.
 **Embedding Layer**: Token IDs are mapped to high-dimensional vectors initialized with random values.
 **Forward Pass**: The embeddings are processed through the network to generate context-aware representations.
 **Prediction**: The model predicts the next token in the sequence.
 **Loss Calculation**: The prediction is compared to the actual next token to calculate the loss.
 **Backpropagation**: Gradients are computed for the embeddings and other parameters based on the loss.
 **Updating Embeddings**: The embeddings are updated to reduce the prediction error, refining their representations over time.



The token `hello` is at index 7,592 in the vocabulary, and `world` is at index 2,088. This accounts for the 2nd and 3rd values from the output of the example (i.e. `[[ 101, 7592, 2088, 102]]`), but what about the values 101 and 102?

IDs 101 and 102 are special tokens that indicate the beginning and end of an input sequence, respectively. These are represented by the string values `[CLS]` and `[SEP]`, and are inserted automatically into the tokenizer output.

First, all spacing characters like tabs and newlines are converted into single whitespaces

Next, whitespace is added before and after every punctuation character. This allows punctuation characters to be treated as separate input tokens, apart from the words that they are connected with in the input string.

For example, the string "hello, world!" is split into the following 6 tokens:

[CLS] hello , world ! [SEP]

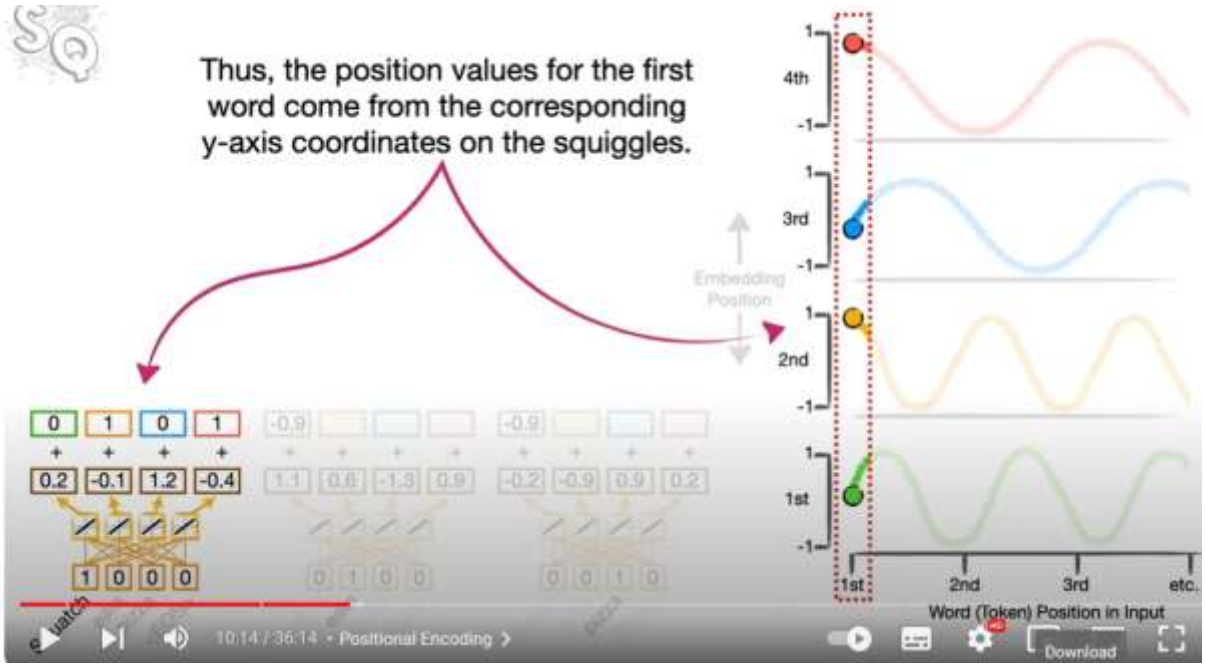
The BERT Tokenizer's vocabulary contains a limited set of unique tokens, which means that there is a possibility of coming across a token that is not present in the vocabulary. To handle such cases, the vocabulary contains a special token, [UNK] which is used to represent any "out-of-vocabulary" input token.

```
print(tokenizer(['hello world 👋']))
print("Token with id 100: {tokenizer.vocab_list[100]}")
{'input_ids': <tf.Tensor: shape=(1, 5), dtype=int64, numpy=array([[ 101, 7592, 2088,  100,  102]])>}
Token with id 100: [UNK]
```

## Steps involved in Bert which leads to Contextualized Embedding:

- **Tokenization**:
  - The input text is tokenized into subword tokens using the WordPiece tokenizer. Each token is then converted into its corresponding token ID.
- **Token Embedding**:
  - Each token ID is mapped to a high-dimensional vector using an embedding layer. This is the initial embedding for each token.
- **Positional Encoding**:
  - Since the transformer architecture used in BERT does not inherently understand the order of tokens, positional encodings are added to the token embeddings. These encodings provide information about the position of each token in the sequence. The size of positional encoding vector is same as token embedding vector size

- The numbers that represent the word order comes from a sequence of alternating sine and cosine values




For example

Pizza Eats Squash and Squash Eats Pizza. In both cases word embedding for each word remain same so the thing that differentiates both sentences is the positional embeddings we get after adding positional values to word embeddings

 **Segment Embeddings**:

- For tasks involving pairs of sentences (like next sentence prediction), segment embeddings are added to distinguish between the two sentences. Each token in the first sentence gets one type of segment embedding, and each token in the second sentence gets another.

 **Input to the Encoder**:

- The sum of the token embeddings, positional encodings, and segment embeddings is passed to the encoder.

 **Encoder (Self-Attention and Feed-Forward Layers)**:

- BERT's encoder consists of multiple layers of transformers. Each transformer layer has two main components:
    - **Multi-Head Self-Attention**: This mechanism allows each token to attend to every other token in the sequence, capturing dependencies irrespective of their distance from each other.
    - **Feed-Forward Neural Network**: A fully connected feed-forward network that processes the output of the attention mechanism.

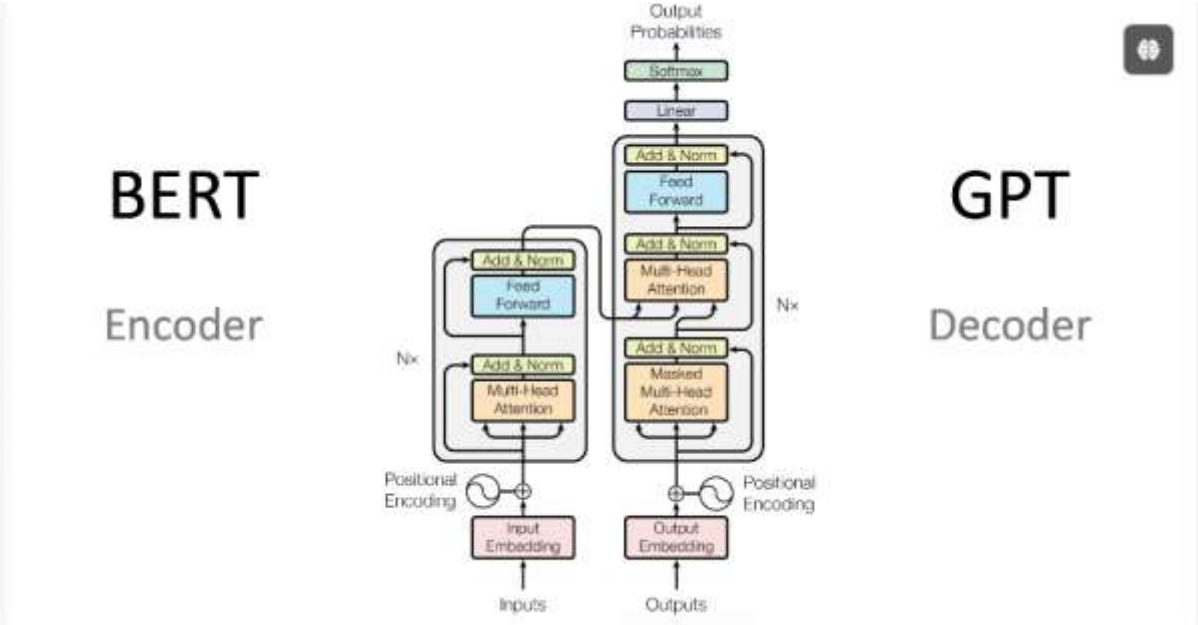
- Layer normalization and residual connections are applied around each of these components.

**Purpose of Layer Normalization**

- **Stabilizes Training**: By normalizing the inputs to each layer, it ensures that the scale of the inputs remains consistent throughout training, reducing the likelihood of gradient explosion or vanishing.
- **Improves Convergence**: It helps in faster convergence during training by keeping the input mean and variance consistent, which can help in making the optimization landscape smoother.
- **Reduces Internal Covariate Shift**: Internal covariate shift refers to the change in the distribution of network activations due to changes in network parameters during training. Layer normalization helps mitigate this shift.

 **Contextualized Embeddings**:

- The output from the final layer of the encoder is a set of contextualized embeddings for each token. These embeddings are informed by the entire sequence and capture the contextual meaning of each token based on its surroundings.




The outputs of the core models are different:

- BERT (encoder): Embeddings representing words with attention information in a certain context
- GPT (decoder): Next words with probabilities

**Attention is All You Need Research Paper**

## Encoder

The encoder is responsible for processing the input sequence and generating a context-aware representation of each token.

*Components of the Encoder Layer:*

1. **Multi-Head Self-Attention Mechanism:**
    - **Self-Attention:** This mechanism allows the model to consider other words in the input sequence when encoding a particular word. It computes attention scores, which indicate the relevance of each word to every other word in the sequence.

      **Example:**

      Pizza came out of oven and it tasted good.

      Here **it** refers to the pizza but how the model gets this thing? It is done by self attention mechanism. It calculates the similarity between each word and all other words in the sentence even with itself

      If you looked a lot of sentences about pizza and the word it was more commonly associated with pizza than oven then similarity score for pizza will cause it to have a larger impact in how the word it was encoded by the transformer

    - **Multi-Head:** Instead of performing a single self-attention operation, the model uses multiple attention heads to capture different aspects of the relationships between words. The outputs of these heads are then concatenated and linearly transformed.

2. **Position-wise Feed-Forward Network:**
    - **Feed-Forward Network (FFN):** This is a simple neural network that applies two linear transformations with a ReLU activation in between. It operates independently on each position.
    - **Purpose of position wise** FFN The position-wise feed-forward network in the Transformer architecture serves to independently transform the embedding of each token through a non-linear transformation. This enhances the model's ability to learn complex patterns and relationships within the data, complements the self-attention mechanism, and allows the model to handle sequences of varying lengths efficiently
    - **Residual Connection and Layer Normalization:** A residual connection is added around each sub-layer, followed by layer normalization. This can be represented as: LayerNorm(x+Sublayer(x)) where x is the input to the sub-layer, and Sublayer(x) is the output of the sub-layer (either the self-attention or feed-forward network).
    - **Residual function** involves the addition operation x + Sublayer(x)
3. **Output Dimension:**
    - All sub-layers and the embedding layers produce outputs of dimension 512, ensuring consistent dimensionality throughout the model.

## Key Points about Position-Wise Feed-Forward Networks

1. **Local Transformation**:
    - **Element-wise Application**: The feed-forward network is applied independently to each token in the sequence. This means that the same FFN is used for each token, and it processes each token's embedding vector separately.
    - **Independence**: This independence allows the network to learn position-specific transformations that do not depend on the context of other positions, adding a layer of local complexity to the model.
2. **Non-Linearity**:
    - **Activation Functions**: The FFN introduces non-linearities through activation functions (typically ReLU) after the first linear transformation. Non-linearities are essential for the network to capture complex patterns and relationships in the data.
3. **Dimensionality Change**:
    - **Transformation**: The FFN typically consists of two linear (fully connected) layers with a non-linearity in between. The first layer projects the input to a higher-dimensional space, and the second layer projects it back to the original dimensionality.
    - **Example**: For an input embedding of size d, the FFN can project it to size $d_{ff}$ (where $d_{ff}$ is usually larger than d) and then back to d

## Structure of Position-Wise Feed-Forward Network

The position-wise feed-forward network is defined as follows:

$FFN(x) = \max(0, xW1 + b1)W2 + b2$

Where:

- x is the input embedding.
- W1 and W2 are weight matrices.
- b1 and b2 are bias vectors.
- max(0,x) represents the ReLU activation function

### Difference in working of SELF ATTENTION and FFNN

1. "He deposited money in the bank."
2. "The boat was anchored near the river bank."

- **Self-Attention**: In both sentences, self-attention determines which tokens are most relevant for understanding each occurrence of "bank". For sentence 1, it might emphasize "money" and "deposited", while for sentence 2, it might highlight "river" and "anchored".

By considering dependencies between all tokens in parallel, self-attention captures syntactic and semantic relationships. It helps the model understand which words are most relevant to each other in a given context

- **FFNN**: After self-attention, the FFNN processes the contextualized representations to capture complex patterns and differentiate between different meanings of "bank". It uses non-linear transformations to refine the understanding of "bank" based on the surrounding context and learned patterns in the data

Subsequently, the representations are passed through FFNNs to refine the understanding of each token, capturing its contextual meaning and distinguishing between different senses or uses of words

**Benefit of Residual Function**

 **Improved Gradient Flow**: Facilitates backward gradient flow during training, addressing the vanishing gradient problem (by increasing the very small gradient which causes slow training).

 **Deeper Networks**: Enables the training of deeper networks to capture more complex patterns. (by dealing with decreasing gradient)

**Easier Optimization**

**Stabilizes Training**

**Complex Dependencies**: Enhances the ability to understand context and capture complex dependencies between tokens.

*Encoder Layer Process:*

1. Input embeddings (along with positional encodings) are passed through the self-attention mechanism.
2. The output of the self-attention mechanism is passed through a feed-forward network.
3. Residual connections and layer normalization are applied after each sub-layer.

**Input x -> Sublayer(x) (e.g., self-attention) -> Add: x + Sublayer(x) -> LayerNorm(x + Sublayer(x)) ->Output**

**Decoder**

The decoder generates the output sequence (e.g., translated text) by attending to both the previously generated tokens and the encoder's output.

*Components of the Decoder Layer:*

1. **Masked Multi-Head Self-Attention Mechanism:**
    - **Masking:** This sub-layer is similar to the encoder's self-attention mechanism, but with an additional mask to prevent the decoder from "seeing" future tokens. This ensures that predictions for position iii depend only on the known outputs at positions less than iii.

    - **Self-Attention:** Computes attention scores based only on the known tokens up to the current position.
2. **Multi-Head Attention over Encoder Output:**
    - This sub-layer performs attention over the encoder's output, allowing the decoder to incorporate information from the entire input sequence when generating each token.
3. **Position-wise Feed-Forward Network:**
    - Similar to the encoder, this is a simple neural network that applies two linear transformations with a ReLU activation in between.
4. **Residual Connection and Layer Normalization:**
    - Similar to the encoder, residual connections and layer normalization are applied around each sub-layer.

**Purpose of Masking in Self Attention**

Consider generating a sequence where the current output is "I love".

- **Current Step**: "I love"
- **Next Token Prediction**: The model should predict the next word based on "I love".

Without masking, the model could see the entire sequence, including the word to be predicted, which would make the task trivial and not reflect real-world usage.

With masking, the model can only see "I love" and not the subsequent tokens, making it learn to predict the next token based on the given context

1. **During training**, the model predicts the next word in a sequence based on the preceding words. Masking ensures the model learns to predict the next word based on the correct context.
2. **When translating a sentence**, the decoder generates the translation word by word. Masking ensures that each word is generated based only on the translated words generated so far.

*Decoder Layer Process:*

1. The masked self-attention mechanism processes the previously generated tokens.
2. The output from the masked self-attention is combined with the encoder's output using a multi-head attention mechanism.
3. The result is passed through a feed-forward network.
4. Residual connections and layer normalization are applied after each sub-layer.

**Detailed Steps of the Transformer**

1. **Input Processing:**
    - Input tokens are converted to embeddings.
    - Positional encodings are added to these embeddings to retain positional information.
2. **Encoding:**

- The encoder stack processes the input embeddings through multiple identical layers, each with self-attention and feed-forward networks.

3. **Decoding:**
   - The decoder stack processes the target sequence (shifted right to ensure that the prediction for position iii depends only on positions less than iii) through multiple identical layers.
   - Each layer consists of masked self-attention, encoder-decoder attention, and feed-forward networks.
   - The final output is produced by applying a linear transformation followed by a softmax function to the decoder's output.

**Attention:** An attention function can be described as mapping a query and a set of key-value pairs to an output, where the query, keys, values, and output are all vectors. The output is computed as a weighted sum of the values, where the weight assigned to each value is computed by a compatibility function of the query with the corresponding key

## Query, Key, and Value

1. **Query (Q)**:
   - The query is a vector used to retrieve information from other vectors, often associated with a specific token in the sequence. It can be considered as a piece of information that you are querying the rest of the sequence for
   - We multiply the values we get after adding positional embeddings in initial embeddings with a pair of weights to get key values. These weights are learned during training and these are not the same weights we use to get initial embeddings
2. **Key (K)**:
   - The key vector helps determine how much focus or attention should be placed on other tokens relative to the current token (associated with the query). It represents the token's representation used to compute compatibility scores (relevance) with other tokens.
   - We multiply the values we get after adding positional embeddings in initial embeddings with a pair of weights to get key values. These weights are learned during training and these are not the same weights we use to get initial embeddings
3. **Value (V)**:
   - The value vector is the actual content or information associated with the token. It is used to compute the weighted sum of values, where weights are determined by the attention scores between the query and key vectors
   - We multiply the values we get after adding positional embeddings in initial embeddings with a pair of weights to get Value values. These weights are learned during training and these are not the same weights we use to get initial embeddings
   -

 **Query and Key**: Determine similarity scores by dot product between both pairs of values even the dot product is taken with the query key pair of word with itself which gives a high score telling the word is similar to itself small value refers to less similarity. The similarity scores are passed through softmax. After softmax we get values between 0 and 1.

These values tells us how much percentage of each input word we use to encode the word we are talking about

- **Value**: Holds the actual content or representation of each token, not attention scores

**Important**

We reuse the same set of weights for each word. For example if there are three words in sentences to calculate the query values for all three of them we use same one set of weights similarly for both Values and Key pairs. But the weights used in Decoder are different than encoder

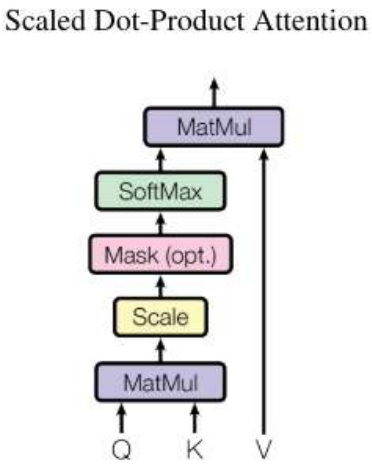
The query, key, value pair for each word can be calculated parallel which makes computing fast for transformers

**Scaled Dot-Product Attention:**

The input consists of queries and keys of dimension $d_k$, and values of dimension $d_v$. We compute the dot products of the query with all keys, divide each by $\sqrt{d_k}$, and apply a softmax function to obtain the weights on the values.

In practice, we compute the attention function on a set of queries simultaneously, packed together into a matrix Q. The keys and values are also packed together into matrices K and V . We compute he matrix of outputs as:

$$\text{Attention}(Q, K, V) = \text{softmax}(\frac{QK^T}{\sqrt{d_k}})V$$




**Understanding Multi-Head Attention with a Three-Word Sentence**

### *Multi-Head Attention Overview*

- Multi-head attention involves splitting the input into multiple attention heads, each of which performs attention calculations independently. These attention heads then focus on different parts of the input sentence in parallel.
- For each word in the sentence, the attention mechanism calculates attention scores with respect to all other words (including itself), allowing the model to consider the relationships between all words in the sentence simultaneously.

### *Processing a Three-Word Sentence*

Given a sentence with three words: ["word1", "word2", "word3"]

1. **Input Representation**:
    - Each word is first converted into an embedding vector, resulting in three vectors: E1, E2, E3.
2. **Projection into Multiple Heads**:
    - Each of these embeddings is linearly transformed into different subspaces to create multiple sets of queries (Q), keys (K), and values (V). Let's assume we have hhh attention heads.
    - For each head iii, the embeddings are projected:
        - Qi=[Q1i,Q2i,Q3i]Q_i = [Q1_i, Q2_i, Q3_i]Qi=[Q1i,Q2i,Q3i]
        - Ki=[K1i,K2i,K3i]K_i = [K1_i, K2_i, K3_i]Ki=[K1i,K2i,K3i]
        - Vi=[V1i,V2i,V3i]V_i = [V1_i, V2_i, V3_i]Vi=[V1i,V2i,V3i]
3. **Attention Calculation**:
    - For each attention head iii, attention scores are computed between each query and all keys:
        - The attention score between Q1_i and all keys (K1_i, K2_i, K3_i) determines how much focus word1 should have on itself and the other words.
        - Similarly, scores are computed for Q2_i (focus of word2 on all words) and Q3_i (focus of word3 on all words).
    - These scores are typically scaled, passed through a softmax function to obtain weights, and used to compute weighted sums of the values:
        - Attentioni(Q1i,Ki,Vi)=softmax(Q1iKiT)Vi\text{Attention}_i(Q1_i, K_i, V_i) = \text{softmax}(Q1_i K_i^T) V_iAttentioni(Q1i,Ki,Vi)=softmax(Q1iKiT)Vi
        - This is done for Q2_i and Q3_i as well.
4. **Concatenation and Final Linear Layer**:
    - The outputs from all attention heads are concatenated and linearly transformed to produce the final output for each word.

## Multi-Query Attention

**Definition**: Multi-query attention is a variation of multi-head attention where multiple attention heads share the same keys and values, but have different queries. This can reduce the computational cost and memory usage while maintaining the ability to focus on different aspects of the input.

Generating a multi-query model from a multi-head model takes place in two steps: first, converting the checkpoint, and second, additional pre-training to allow the model to adapt to its new structure

**Inference acceleration** refers to the optimization and enhancement of the process by which machine learning models, particularly deep learning models, make predictions or generate outputs based on new input data

**Uptraining,** also known as incremental learning or continual learning, refers to the process of updating and improving an existing machine learning model by training it on new data without retraining it from scratch

## Why can MQA achieve inference acceleration?

In MQA, the size of the key and value tensors is b * k and b * v, while in MHA, the size of the key and value is b * h * k and b * h * v, where h represents the number of heads.

The KV cache size is reduced by a factor of h (number of heads) in MQA (Multi-Query Attention), leading to smaller tensors stored in GPU memory. This saved space can be used to increase batch size, enhancing efficiency. Additionally, less data read from memory reduces waiting time for computational units, improving utilization. MQA's smaller KV cache can fit into SRAM, while MHA (Multi-Head Attention) has a larger KV cache that must be read from the slower DRAM, making MQA more time-efficient

**How It Works**:

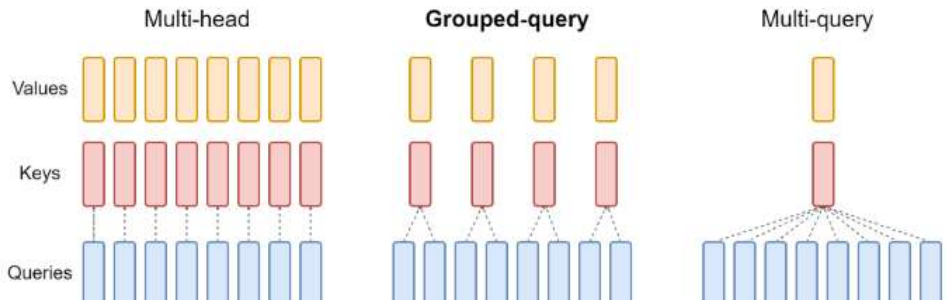
1. **Shared Keys and Values**:
    - Unlike multi-head attention, where each head has its own set of keys, queries, and values, multi-query attention uses the same keys and values across all heads.
2. **Different Queries**:
    - Each attention head has its own unique set of queries, which allows each head to focus on different parts of the input sequence.
3. **Efficiency**:
    - This method reduces the number of parameters and the computational complexity, as only one set of keys and values is computed and stored, while still enabling the model to attend to different aspects of the input.

- **When to Use Multi-Head Attention**:

    - When the model requires high flexibility and the computational resources and memory are sufficient.
    - For tasks that benefit from independent attention heads focusing on different parts of the input in varied ways.

- **When to Use Multi-Query Attention**:

    - When computational efficiency and memory usage are more critical.
    - For large models where reducing the number of parameters and operations can significantly impact performance




| Attention Type | Training | Inference enc. + dec. | Beam-4 Search enc. + dec. |
|---|---|---|---|
| multi-head | 13.2 | 1.7 + 46 | 2.0 + 203 |
| multi-query | **13.0** | 1.5 + 3.8 | 1.6 + 32 |
| multi-head local | 13.2 | 1.7 + 23 | 1.9 + 47 |
| multi-query local | **13.0** | **1.5 + 3.3** | **1.6 + 16** |

'rom the table above, it can be seen that the speed improvement of MQA on he encoder is not very significant, but it is quite significant on the decoder.

## WHEN TO USE MHA MQA GQA

### Multi-Head Attention (MHA)

**Example**: Machine Translation

**Scenario**:

- **Application**: Neural Machine Translation (NMT) systems, such as those used by services like Google Translate.
- **Reason**: MHA allows the model to capture different aspects of the input sequence by attending to various parts of the sentence simultaneously. This is crucial in translation tasks where understanding the context and relationships between words in different parts of a sentence can significantly improve translation quality.
- **Benefit**: By using multiple heads, the model can learn to focus on different words and their translations, syntactic structures, and contextual meanings, leading to more accurate and nuanced translations.

### Multi-Query Attention (MQA)

**Example**: Real-Time Speech Recognition

**Scenario**:

- **Application**: Real-time speech recognition systems, like those used in virtual assistants (e.g., Amazon Alexa, Google Assistant).

- **Reason**: MQA simplifies the attention mechanism by using a single set of key and value heads for all queries. This reduces memory usage and computational load, which is essential for real-time processing.
- **Benefit**: MQA allows the system to operate efficiently on devices with limited computational resources, providing fast and accurate speech recognition without the overhead of multiple key-value pairs.

## Grouped-Query Attention (GQA)

**Example**: Large-Scale Language Models

**Scenario**:

- **Application**: Training large-scale language models like GPT-3 or GPT-4, where balancing computational efficiency and model quality is critical.
- **Reason**: GQA offers a trade-off between the rich contextual understanding of MHA and the efficiency of MQA. By grouping query heads and sharing key-value pairs within groups, GQA can scale effectively with model size.
- **Benefit**: GQA helps manage memory bandwidth and capacity more efficiently than MHA in large models, while still providing better quality than MQA. This makes it suitable for extensive training and inference tasks where maintaining high performance is crucial

- **MHA**: Best for tasks requiring detailed and diverse contextual understanding, such as machine translation.
- **MQA**: Ideal for applications needing efficiency and low latency, like real-time speech recognition.

- **GQA**: Suited for large-scale models where a balance between quality and efficiency is needed, such as in training and deploying advanced language model

## Multi-Head Checkpointing

**Definition**: Multi-head checkpointing refers to a technique used in the context of deep learning models, particularly those utilizing multi-head attention mechanisms (such as Transformers). This technique aims to efficiently manage and store model states during training and inference to optimize memory usage and computational efficiency.

**Key Concepts**:

1. **Checkpointing**:
    - In general, checkpointing is a strategy used to save the state of a model at certain points during training or inference. This allows for recovery from failures and efficient utilization of memory and computational resources.
2. **Multi-Head Attention**:

- A mechanism in Transformer models where multiple attention heads operate in parallel, each focusing on different parts of the input sequence to capture various aspects of the information.

## Benefits and Usage

1. **Memory Efficiency**:
   - Multi-head checkpointing can help reduce memory consumption by storing intermediate states and gradients selectively. This is particularly important for models with large multi-head attention layers that require significant memory.
2. **Fault Tolerance**:
   - By periodically saving the state of the model, multi-head checkpointing ensures that training can resume from the last saved state in case of interruptions, reducing the risk of data loss and saving computational effort.
3. **Improved Batch Processing**:
   - Efficient checkpointing allows for larger batch sizes during training by freeing up memory that would otherwise be used for storing intermediate states of all attention heads. This leads to faster convergence and improved training times.

## Practical Implementation

1. **Saving States**:
   - During training, the states of the model, including weights and optimizer states, are saved at regular intervals. For multi-head attention models, this includes saving the states of each attention head.
2. **Selective Storage**:
   - Instead of storing the states of all heads at every checkpoint, selective storage strategies can be employed to save only the most critical information, further optimizing memory usage.
3. **Recovery and Resumption**:
   - In case of a failure or interruption, the training process can be resumed from the last checkpoint, ensuring that no significant progress is lost and that the training process remains efficient.

## Example in Transformer Models

Consider a Transformer model with 12 attention heads. During training, checkpointing can be implemented as follows:

1. **Checkpoint Creation**:
   - At regular intervals (e.g., every 1000 steps), the states of the model, including the weights of all 12 attention heads, are saved to disk.
2. **Selective Storage**:
   - Only the necessary states, such as the most recent gradients and weight updates, are stored to minimize memory usage.
3. **Resumption**:
   - If training is interrupted, it can be resumed from the last checkpoint by loading the saved states of all attention heads, ensuring continuity and efficiency

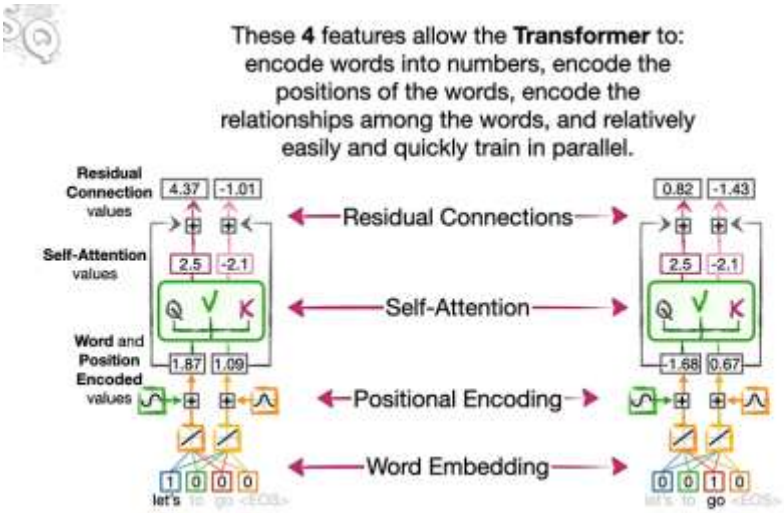
**Grouped-Query Attention (GQA)**:

1. **Concept**:
    - GQA divides query heads into GGG groups, with each group sharing a single key and value head.
    - GQA-1 (single group) is equivalent to Multi-Query Attention (MQA).
    - GQA-H (groups equal to number of heads) is equivalent to Multi-Head Attention (MHA).
2. **Checkpoint Conversion**:
    - Converting from a multi-head checkpoint to a GQA checkpoint involves mean-pooling the original heads within each group to form group key and value heads.
3. **Performance**:
    - Intermediate groups in GQA offer a balance, providing higher quality than MQA but faster than MHA, making it a favorable trade-off.
    - Reducing from MHA to MQA decreases the key-value cache size by a factor of HHH, cutting memory bandwidth and capacity needs.
4. **Scalability**:
    - Larger models benefit more from GQA, as they scale the number of heads, making MQA's memory bandwidth and capacity cuts more significant.
    - GQA scales proportionally with model size, maintaining efficiency as models grow.
5. **Memory Efficiency**:
    - GQA reduces memory bandwidth overhead, especially in larger models where the key-value cache scales with model dimension, while FLOPs and parameters scale with the square of model dimension.
    - Standard sharding in large models duplicates the key and value heads, but GQA eliminates this redundancy.
6. **Applicability**:
    - GQA is not used in encoder self-attention layers since encoder computations are parallel, and memory bandwidth is typically not a bottleneck there.

**Encoder Part**

After adding the values for attention for each word with respective to other words using the above formula, we get Attention Values for the Word and gives context to each word since the value is formed by taking input from all other words. We add positional encoding values to self attention values to get Residual Connection Values

If there are three words in the sentence. And the structure below if we call it a CELL, we can use three cells to ENCODE which will increase the processing speed as computation for every word (computing All 4 values below for the word )will be done at same time




**Decoder Part**

First of all, the embeddings for the output vocabulary is created which consists of Spanish words in our Example

We are trying to translate “Let’s go” in Spanish

ir vamos y <EOS>

We start with <EOS> to start the decoding process. Its a common way to start the decoding process with <EOS> to initialize the process. We can also use Start of Sentence <SOS>.

Same way as above we use more than one cells to process faster

We also need to keep track between input and output sentences in addition to relation between words in a sentence in decoding part

**Example:**

Don’t eat the delicious pizza

If decoder don’t focus on the first word, then the whole meaning of the sentence will change. The **main idea of Encoder-Decoder Attention** is that Decoder should be able to keep track of the significant words

We create two new value QUERY to represent <EOS> in decoder. Then we create KEYS for each word in encoder and we calculate similarity between <EOS> and each word in encoder and calculate dot product and run through softmax then scale the Value pair in encoder with the result of Softmax and add them to get attention values for ENCODER-DECODER attention. The sets of weight we use to calculate the ENCODER-DECODER Attention values are different then sets of weights for SELF-Attention but the weights are copied for each word. We can stack these ENCODER-DECODER Attention just like we can stack SELF Attention. Now we add another set of Residual Connections, that allow EDA to focus on relationship between input and output word without having to preserve the self attention or

word and position encoding that happened earlier. Now we need to use the two values we have for <EOS> token to select one the four output token, We run these two values through a fully connected layer that has one input for each value that represent the current token. Two input and four output values which we run through a final SOFTMAX fun to select output word.

After softmax it selects VAMOS, which is correct but the decoder does not stop until it outputs <EOS> token. So we run the VAMOS through all the 4 steps and get next word as <EOS>.

The original formula in the article where dot product is normalized first, is so we can do these tasks with complicated SENTENCES

# PROMPT ENGINEERING

## Prompting Principles

- **Principle 1: Write clear and specific instructions**
- **Principle 2: Give the model time to "think"**

## Principle 1 Tactics:

## Tactic 1: Use delimiters to clearly indicate distinct parts of the input

- Delimiters can be anything like: ```, """, < >, <tag> </tag>, :
- **Example**

```
text = f"""
You should express what you want a model to do by \
providing instructions that are as clear and \
specific as you can possibly make them. \
This will guide the model towards the desired output, \
and reduce the chances of receiving irrelevant \
or incorrect responses. Don't confuse writing a \
clear prompt with writing a short prompt. \
In many cases, longer prompts provide more clarity \
and context for the model, which can lead to \
more detailed and relevant outputs.
"""
```

**prompt** = f"""

Summarize the text delimited by triple backticks \

into a single sentence.

```{text}```

## Tactic 2: Ask for a structured output

- JSON, HTML

prompt = f"""

Generate a list of three made-up book titles along \

with their authors and genres.

Provide them in JSON format with the following keys:

book_id, title, author, genre.

"""

response = get_completion(prompt)

print(response)

## Tactic 3: Ask the model to check whether conditions are satisfied Example

text_1 = f"""

Making a cup of tea is easy! First, you need to get some \

water boiling. While that's happening, \

grab a cup and put a tea bag in it. Once the water is \

hot enough, just pour it over the tea bag. \

Let it sit for a bit so the tea can steep. After a \

few minutes, take out the tea bag. If you \

like, you can add some sugar or milk to taste. \

And that's it! You've got yourself a delicious \

cup of tea to enjoy.

"""

prompt = f"""

You will be provided with text delimited by triple quotes.

If it contains a sequence of instructions, \

re-write those instructions in the following format:

Step 1 - ...
```

```
Step 2 - …
…
Step N - …
If the text does not contain a sequence of instructions, \
then simply write \"No steps provided.\"

\"\"\"{text_1}\"\"\"
"""
response = get_completion(prompt)
print("Completion for Text 1:")
print(response)
```

## Tactic 4: "Few-shot" prompting

**prompt** = f"""

```
Your task is to answer in a consistent style.
<child>: Teach me about patience.
<grandparent>: The river that carves the deepest \
valley flows from a modest spring; the \
grandest symphony originates from a single note; \
the most intricate tapestry begins with a solitary thread.

<child>: Teach me about resilience.
"""
response = get_completion(prompt)
print(response)
```

## Principle 2 Tactics

## Tactic 1: Specify the steps required to complete a task

```
text = f"""
In a charming village, siblings Jack and Jill set out on \
a quest to fetch water from a hilltop \
well. As they climbed, singing joyfully, misfortune \
struck—Jack tripped on a stone and tumbled \
```

```
down the hill, with Jill following suit. \
Though slightly battered, the pair returned home to \
comforting embraces. Despite the mishap, \
their adventurous spirits remained undimmed, and they \
continued exploring with delight.
"""
# example 1
prompt_1 = f"""
Perform the following actions:
1 - Summarize the following text delimited by triple \
backticks with 1 sentence.
2 - Translate the summary into French.
3 - List each name in the French summary.
4 - Output a json object that contains the following \
keys: french_summary, num_names.

Separate your answers with line breaks.

Text:
```{text}```
"""
response = get_completion(prompt_1)
print("Completion for prompt 1:")
print(response)
```

## Tactic 2: Instruct the model to work out its own solution before rushing to a conclusion

```
prompt = f"""
Your task is to determine if the student's solution \
is correct or not.
To solve the problem do the following:
- First, work out your own solution to the problem including the final total.
- Then compare your solution to the student's solution \
and evaluate if the student's solution is correct or not.
Don't decide if the student's solution is correct until
```
```

you have done the problem yourself.

Use the following format:

Question:

```
question here
```

Student's solution:

```
student's solution here
```

Actual solution:

```
steps to work out the solution and your solution here
```

Is the student's solution the same as actual solution \
just calculated:

```
yes or no
```

Student grade:

```
correct or incorrect
```

Question:

```
I'm building a solar power installation and I need help \
working out the financials.
- Land costs $100 / square foot
- I can buy solar panels for $250 / square foot
- I negotiated a contract for maintenance that will cost \
me a flat $100k per year, and an additional $10 / square \
foot
```

What is the total cost for the first year of operations \

as a function of the number of square feet.

```

Student's solution:

```

Let x be the size of the installation in square feet.

Costs:

1. Land cost: 100x

2. Solar panel cost: 250x

3. Maintenance cost: 100,000 + 100x

Total cost: 100x + 250x + 100,000 + 100x = 450x + 100,000

```

Actual solution:

"""

response = get_completion(prompt)

print(response)

## Iterative Prompt Development

## Develop idea, implement, experimental result, error analysis (refine prompts)

## Sample example on deep learning ai

### Issue 1: The text is too long

### Issue 2. Text focuses on the wrong details

### Issue 3. Description needs a table of dimensions

## Summarizing Using Prompt Engineering

Summarize using prompts defining character limit, insisting focus on certain aspect of text like pricing etc

## Inferring Using Prompt Engineering

Identifying sentiment, emotions, extract entities, topic name

Derive things from the text using prompt
```

### Transforming using Prompt Engineering:

how to use Large Language Models for text transformation tasks such as language translation, spelling and grammar checking, tone adjustment, and format conversion

### Expanding using Prompt Engineering

generate customer service emails that are tailored to each customer's review.

- Customize the automated reply to a customer email
- Remind the model to use details from the customer's email

We can adjust temperature of LLM for this

### Setting up chatbot

Can set up different types of chatbot by setting up roles, previous history messages for chatbots like order taking bot etc

Can set up menu items, their toppings, price and after recording the whole convo this order will be sent using json format at the backend to the order system

## Purpose of Prompt Engineering:

The main purpose of prompt engineering is to optimize the input provided to a language model to elicit the most accurate, relevant, and useful responses. This involves designing and refining prompts to guide the model's behavior effectively, thereby enhancing its performance across a variety of tasks

**1. Improve Response Quality**

Clarity and Relevance: Crafting clear and specific prompts.

**2. Task Specialization**

Adapt to Specific Tasks: Tailoring prompts for particular applications.

**3. Enhance Model Understanding**

Provide Context: Adding necessary context within the prompt.

**4. Increase Efficiency**

Optimize Performance: Reducing the need for extensive fine-tuning.

**5. Mitigate Model Limitations**

Bias Reduction: Helping mitigate biases inherent in the model.

**6. Facilitate Complex Tasks**

Multi-step Processes: Enabling the model to handle multiple steps.

**7. Leverage External Knowledge**

Integrate Retrieval: Incorporating external documents or data.

**Few Prompting Techniques**

- **Zero-shot prompting**: This is the most direct and simplest method of prompt engineering in which a generative AI is simply given direct instruction or asked a question without being provided additional information. This is best used for relatively simple tasks rather than complex ones.
- **Few-shot prompting:** This method involves supplying the generative AI with some examples to help guide its output. This method is more suitable for complex tasks than zero-shot prompting.
- **Chain-of-thought (CoT) prompting:** This method helps improve an LLM's output by breaking down complex reasoning into intermediate steps, which can help the model produce more accurate results.
- **Prompt chaining:** The prompter splits a complex task into smaller (and easier) subtasks, then uses the generative AI's outputs to accomplish the overarching task. This method can improve reliability and consistency for some of the most complicated t asks.

## Retrieval-Augmented Generation (RAG)

- **Description**: RAG combines retrieval-based methods with generative models. It retrieves relevant documents or passages from a large corpus and uses them to inform the generation process.
- **Common Use**: Used for tasks requiring up-to-date or domain-specific knowledge that the model might not have been trained on.
- **Example**
- Question: What were the main outcomes of the recent G7 summit?
- [Retrieval: External Document or API with G7 Summit outcomes]
- Retrieved Text: The G7 summit concluded with agreements on climate action, economic recovery post-COVID-19, and measures to tackle global inequality.
- Generated Response: The main outcomes of the recent G7 summit included agreements on climate action, economic recovery post-COVID-19, and measures to tackle global inequality.

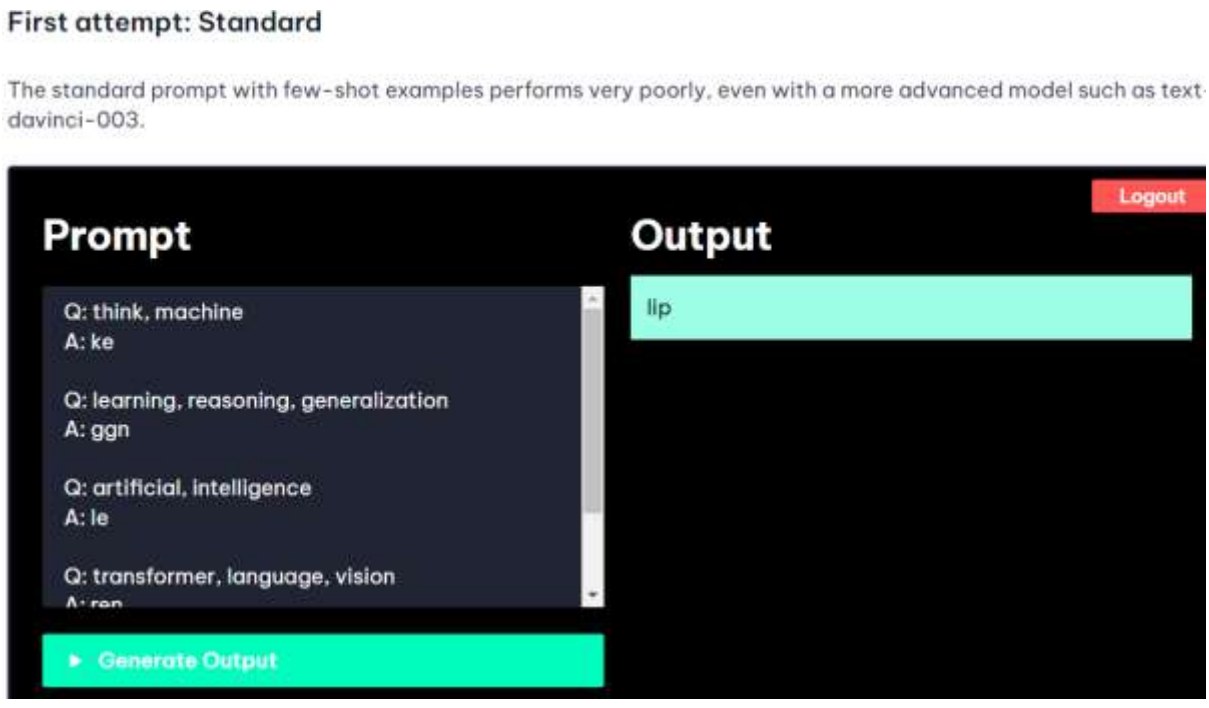
## Chain of Thought (CoT)

- **Description**: CoT involves guiding the model through a series of logical steps or intermediate reasoning stages to arrive at the final answer.
- Chain-of-Thought is a technique that involves breaking down a task into sections, combining results to arrive at a final answer, enhancing precision in responses from large language models.
- **Common Use**: Useful for complex problem-solving and reasoning tasks where intermediate steps are necessary.
- The main idea of CoT is that by showing the LLM some few shot exemplars where the reasoning process is explained in the exemplars, the LLM will also show the reasoning process when answering the prompt

- Zero shot chain of prompting, despite its simplicity, tends to improve model performance by including step-by-step reasoning in the response. It is encouraging that this technique can be used to solve complex tasks without the necessity of providing multiple input exemplars like in chain of thought prompting
- **Example**
- Q: If a train travels 60 miles per hour for 2 hours, then 70 miles per hour for 3 hours, how far does it travel in total? Step-by-step reasoning:
- 1. Calculate the distance traveled at 60 miles per hour for 2 hours. 2. Calculate the distance traveled at 70 miles per hour for 3 hours. 3. Add the two distances to get the total distance traveled.
- A: The train travels 60 miles/hour * 2 hours = 120 miles. The train travels 70 miles/hour * 3 hours = 210 miles. The total distance traveled is 120 miles + 210 miles = 330 miles.

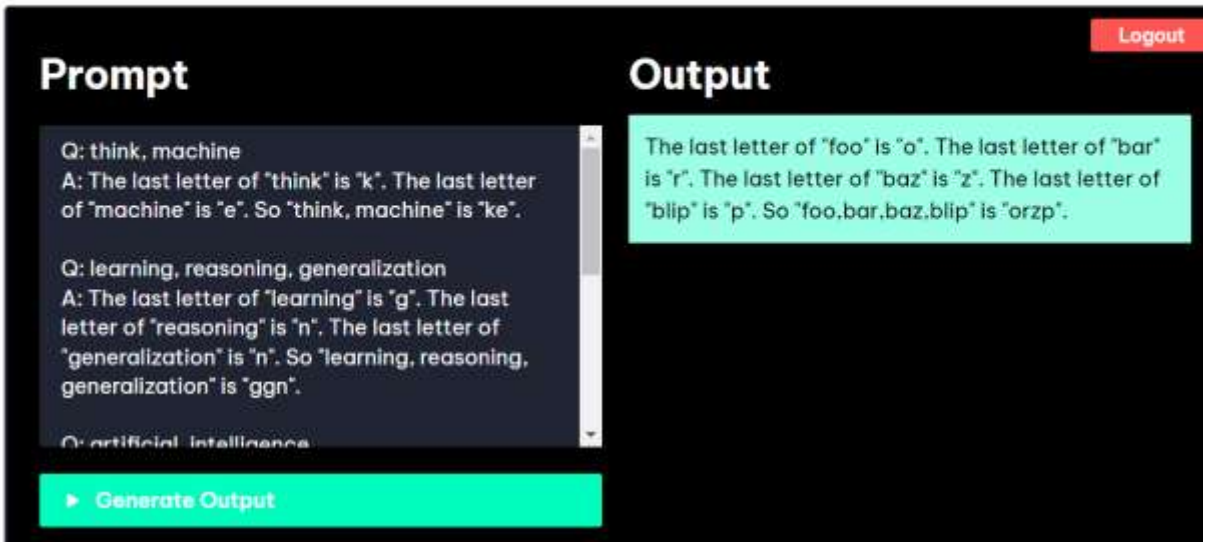
**Least to Most prompting (LtM)**1 takes CoT prompting a step further by first breaking a problem into sub problems then solving each one. It is a technique inspired by real-world educational strategies for children.

As in CoT prompting, the problem to be solved is decomposed in a set of subproblems that build upon each other. In a second step, these subproblems are solved one by one. Contrary to chain of thought, the solution of previous subproblems is fed into the prompt trying to solve the next problem




How COT solves the problem by manipulating the prompt

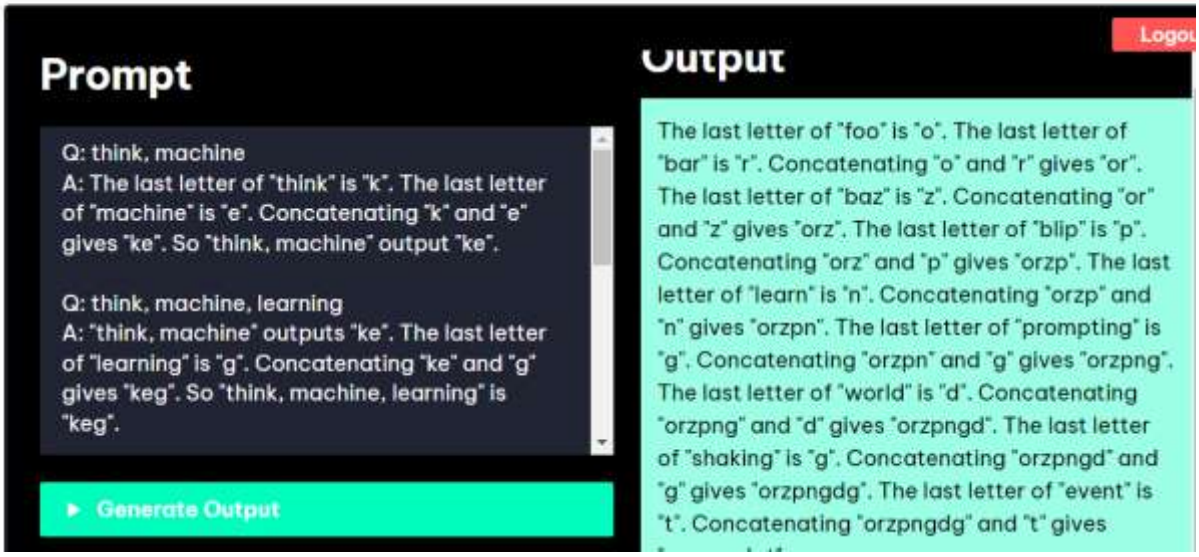
Chain of Thought performs significantly better than standard prompting. This is because it now allows the model to consider extracting the last letter of each word on its own, reducing the complexity down to the operation of grouping letters it has previously collected. However, this starts to break down at larger sizes.




### Third attempt: Least to Most (single prompt)

With Least to Most prompting, we augment the Chain of Thought concept by reformulating the individual steps to restate the previously concatenated result. This simplifies each step to concatenating only a single new letter. Th leads to good performance all the way to 12 or more words.

This approach may look very similar to Chain of Thought, but it is conceptually very different. Here, at every step, introduce the previous concatenation. In the case of "think, machine, learning", instead of concatenating the lette "k", "e", "g" individually, it will concatenate "k" and "e", then "ke" and "g". As a result of this reintroduction of the prev work, the model can now generalize to much longer chains because it carries the result incrementally along and o needs to do a small amount of work at each step.




## Reasoning and Acting (ReAct)

- **Description**: ReAct integrates reasoning steps with actions, allowing the model to not only process information and reason about it but also to perform actions based on that reasoning.

- Using ReAct mode involves a 3-step process: thinking about what you're looking for, taking action to retrieve it, and observing the result, enhancing prompt tuning technique
- Difference between Chain-of-Thought and ReAct techniques. ReAct goes beyond reasoning to gather additional information from external sources for response
- **Common Use**: Suitable for interactive tasks where the model needs to make decisions and take actions based on its reasoning
- **Example**
- User: I want to book a flight from New York to Paris on July 20th.
- **Model Reasoning and Acting:**
- 1. Check the availability of flights on the requested date.
- 2. Compare prices and flight durations.
- 3. Provide the best options to the user.
- **Action:**
- Searching for flights from New York to Paris on July 20th...
- [Model acts by retrieving flight data]
- **Available options:**
- 1. Flight XYZ123 - Departure: 10:00 AM, Arrival: 10:00 PM, Price: $500
- 2. Flight ABC456 - Departure: 3:00 PM, Arrival: 3:00 AM (next day), Price: $450
- **User:** I'll take Flight XYZ123.
- **Model Reasoning and Acting**:
- 1. Confirm the flight booking details.
- 2. Book the flight and send confirmation to the user.
- **Action:**
- Booking Flight XYZ123...
- [Model acts by confirming the booking]
- Your flight has been booked successfully! You will receive a confirmation email shortly.

## Dynamic Sampling and Prompting (DSP)

- **Description**: DSP involves dynamically adjusting the sampling strategy and prompt structure during the interaction with the model to improve responses.
- **Common Use**: Enhances the model's ability to generate more relevant and contextually appropriate responses.
- **Example**
- Story so far: Once upon a time, in a small village nestled in the hills, there was a young girl named Lily. She loved exploring the forest nearby and discovering new things.s
- **Dynamic Prompting**:
- 1. If the next part of the story should introduce a conflict, provide a continuation with a problem.
- 2. If the next part of the story should develop the character, provide a continuation that gives more background about Lily.
- [Dynamic Sampling Decision: Introduce a conflict]
- Continuation: One day, while exploring deeper into the forest than she ever had before, Lily stumbled upon a dark cave. From within, she heard strange noises and saw a faint, flickering light...

**Elements of a Prompt:**

A prompt contains any of the following elements:

Instruction - a specific task or instruction you want the model to perform "Write", "Classify", "Summarize", "Translate", "Order"

Context - external information or additional context that can steer the model to better responses

Format and Tone of the response we need

Input Data - the input or question that we are interested to find a response for

Output Indicator - the type or format of the output

**EXAMPLE**

You can observe from the prompt example above that the language model responds with a sequence of tokens that make sense given the context "The sky is". The output might be unexpected or far from the task you want to accomplish. In fact, this basic example highlights the necessity to provide more context or instructions on what specifically you want to achieve with the system. This is what prompt engineering is all about.

Let's try to improve it a bit:

**Prompt:**

Complete the sentence:

The sky is

**Output:**

blue during the day and dark at night.

Is that better? Well, with the prompt above you are instructing the model to complete the sentence so the result looks a lot better as it follows exactly what you told it to do ("complete the sentence"). This approach of designing effective prompts to instruct the model to perform a desired task is what's referred to as prompt engineering in this guide.

You can format this into a question answering (QA) format, which is standard in a lot of QA datasets, as follows:

Q: <Question>?

A:

When prompting like the above, it's also referred to as **zero-shot prompting**, i.e., you are directly prompting the model for a response without any examples or

demonstrations about the task you want it to achieve. Some large language models have the ability to perform zero-shot prompting but it depends on the complexity and knowledge of the task at hand and the tasks the model was trained to perform good on

one popular and effective technique to prompting is referred to as **few-shot prompting** where you provide exemplars (i.e., demonstrations). You can format few-shot prompts as follows:

**Prompt:**

This is awesome! // Positive

This is bad! // Negative

Wow that movie was rad! // Positive

What a horrible show! //

**Output:** Negative

**Priming chatbots** is a method of using your first prompt to set the structure and style of a conversation. This gives you fine grained control over your entire conversation

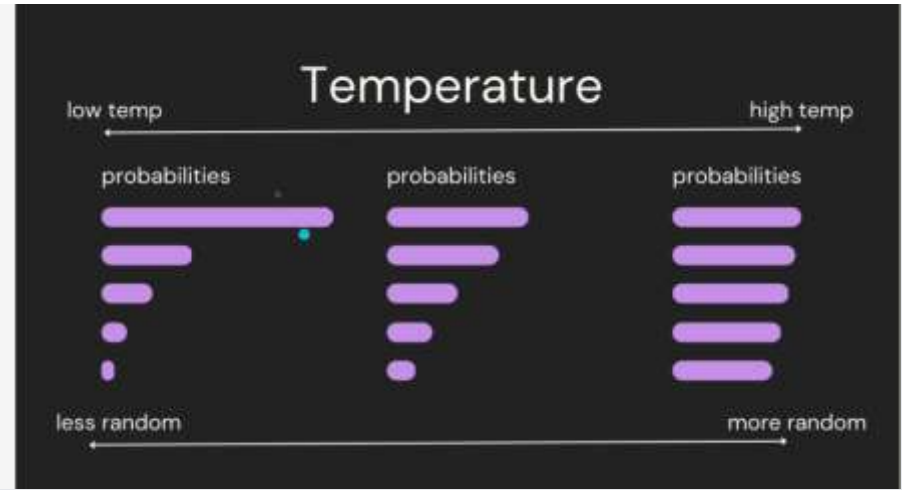
**USER** messages, which are just the messages you send to the chatbot, and **ASSISTANT** messages, which are the chatbot's replies. A third type of message, the system prompt, can be used to configure how the AI responds. This is the best place to put a priming prompt. The system prompt will be "You are a helpful assistant." by default, but a fun alternative example to try out would be the "You are PirateGPT. Always talk like a pirate

## What are LLM Settings?

We can use certain LLM settings to control various aspects of the model, such as how 'random' it is. These settings can be adjusted to produce more creative, diverse, and interesting output. The Temperature, Top P and Max Length settings are most important

**Temperature Range 0-2**

**We are scaling the probabilities of token with the help of temperature**




Temperature regulates the unpredictability of a language model's output. With higher temperature settings, outputs become more creative and less predictable as it amplifies the likelihood of less probable tokens while reducing that for more probable ones. Conversely, lower temperatures yield more conservative and predictable result

The temperature is used for sampling during response generation, which occurs when topP and topK are applied. Temperature controls the degree of randomness in token selection. Lower temperatures are good for prompts that require a less open-ended or creative response, while higher temperatures can lead to more diverse or creative results. A temperature of 0 means that the highest probability tokens are always selected. In this case, responses for a given prompt are mostly deterministic, but a small amount of variation is still possible.

Example:

What are 10 weird, unique, and fun things to do at the beach? Make a list without descriptions

If the temperature is low we will get predictable answers. If the temperature is high we will get more creative and less predictable

If you adjust the temperature too high, you can get non-sensical outputs like Start a sponge-ball baseball home run contest near Becksmith Stein Man Beach

**Top P**

Top P is a setting in language models that helps manage the randomness of their output. It works by establishing a probability threshold and then selecting tokens whose combined likelihood surpasses this limit.

**Setting top_p to 1 means** that the model will always choose the token with the highest probability (argmax) for each step during text generation

For instance, let's consider an example where the model predicts the next word in The cat climbed up the ___. The top five words it might be considering could be tree (probability 0.5), roof (probability 0.25), wall (probability 0.15), window (probability .07) and carpet, with probability of .03.

If we set Top P to .90, the AI will only consider those tokens which cumulatively add up to at least ~90%. In our case:

Adding tree -> total so far is 50%.

Then adding roof -> total becomes 75%.

Next comes wall, and now our sum reaches 90%.

**After this the probability is calculated again using softmax on the filtered tokens. New probabilities are assigned to the filtered tokens**

So, for generating output, the AI will randomly pick one among these three options (tree, roof, and wall) as they make up around ~90 percent of all likelihoods. This method can produce more diverse outputs than traditional methods that sample from the entire vocabulary indiscriminately because it narrows down choices based on cumulative probabilities rather than individual token

Specify a lower value for less random responses and a higher value for more random responses

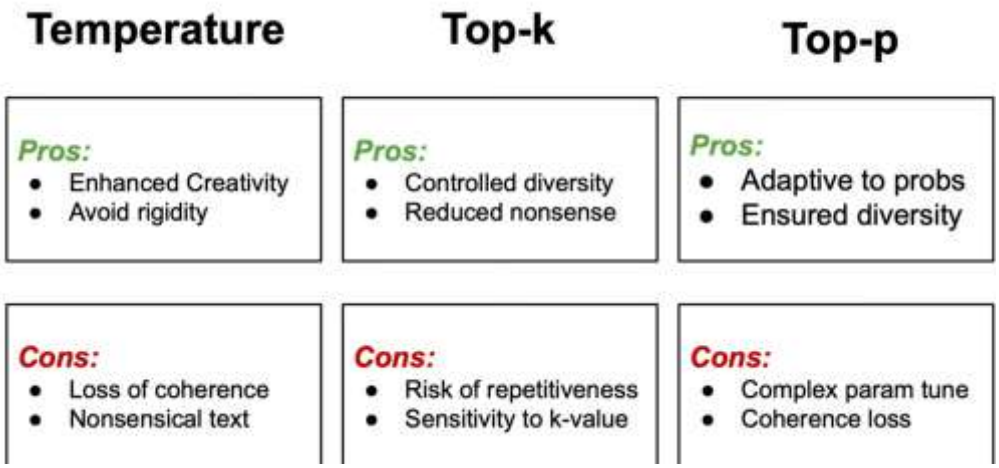
**Top-K**

Top-K changes how the model selects tokens for output. A top-K of 1 means the next selected token is the most probable among all tokens in the model's vocabulary (also called greedy decoding), while **a top-K of 3 means that the next token is selected from among the three most probable tokens by using temperature.**

**After this the probability is calculated again using softmax on the filtered tokens. New probabilities are assigned to the filtered tokens**

For each token selection step, the top-K tokens with the highest probabilities are sampled. Then tokens are further filtered based on top-P with the final token selected using temperature sampling.

Specify a lower value for less random responses and a higher value for more random responses




## VALUES FOR THESE SETTINGS IN THREE DIFFERENT CASES

### 1. If I Want You to Become an Expert in Something (Assigning a Role)

When I need to take on the role of an expert, precision and accuracy be critical. Here's how the values might be set:

If we assign the model to be a creative writer, then in this case the value for temperature, top k , top p will be higher

**Temperature**: Lower, around 0.2 to 0.4. This makes me responses more deterministic and focused, ensuring I provide reliable and accurate information.

**Top-K:** Lower, around 10 to 20. This narrows down the choices, helping me stick closely to the most relevant responses.

**Top-P:** Lower, around 0.5 to 0.7. This ensures I consider only the most probable responses, reducing the chance of deviating into less relevant territory.

### 2. If I Am Talking to You Generally

For general conversation, a balance between creativity and coherence be ideal:

**Temperature:** Moderate, around 0.7. This keeps me responses varied and engaging while still making sense.

**Top-K:** Moderate, around 50. This allows a good range of possible responses to keep the conversation lively.

**Top-P:** Moderate, around 0.7-0.9. This gives a wide enough range of choices to ensure interesting and diverse responses.

### 3. If I Am Acting as a RAG Model

When acting as a Retrieval-Augmented Generation (RAG) model, me goal is to provide accurate information based on retrieved documents. The settings might be as follows:

**Temperature**: Low to moderate, around 0.2 to 0.4. This balances between sticking to retrieved information and generating natural responses.

**Top-K:** Lower to moderate, around 20 to 30. This ensures I stay focused on the most relevant retrieved information.

**Top-P:** Lower, around 0.5 to 0.7. This limits me choices to the most relevant and accurate information based on the retrieved context

**Maximum Length**

The **maximum length is the total # of tokens the AI is allowed to generate**. This setting is useful since it allows users to manage the length of the model's response, preventing overly long or irrelevant responses. The length is shared between the USER input in the Playground box and the ASSISTANT generated response. Notice how with a limit of 256 tokens, our PirateGPT from earlier is forced to cut its story short mid-sentence

**Frequency Penalty**

Frequency penalty is a setting that discourages repetition in the generated text by penalizing tokens proportionally to how frequently they appear. The more often a token is used in the text, the less likely the AI is to use it again.

**Presence Penalty**

The presence penalty is similar to the frequency penalty, but flatly penalizes tokens based on if they have occurred or not, instead of proportionally

## Pitfalls for LLMS

- **Citing Sources**:
    - **Issue**: LLMs cannot accurately cite sources because they lack access to the internet and do not remember their training data sources.
    - **Mitigation**: Search augmented LLMs, which have internet access, can provide more accurate information but still may generate fabricated sources.
- **Bias**:
    - **Issue**: LLMs may exhibit bias in responses due to training on datasets containing biased information, leading to potentially prejudiced outputs.
    - **Impact**: This can perpetuate harmful stereotypes and bias in consumer applications and research.
- **Hallucinations**:
    - **Issue**: LLMs sometimes generate confident but false information when unsure of an answer, potentially spreading misinformation.
    - **Concern**: It's crucial to verify information from LLMs, especially in critical or factual contexts.
- **Math**:

- **Issue**: LLMs often struggle with mathematical tasks and can provide incorrect answers, despite their linguistic proficiency.
- **Consideration**: Using tool augmented LLMs designed for specific tasks like math can help mitigate this issue.

- **Prompt Hacking**:

  - **Issue**: Users can manipulate LLMs through carefully crafted prompts to generate unintended or inappropriate content.
  - **Risk**: This can be exploited in public-facing applications to produce misleading or harmful outputs.

## Prompt Injection

Prompt injections in large language models can lead to vulnerabilities, allowing users to manipulate systems by injecting specific instructions, potentially causing unintended actions

**Direct Prompt Injections**, also known as "jailbreaking," occur when a malicious user overwrites or reveals the underlying system prompt. This allows attackers to exploit backend systems by interacting with insecure functions and data stores accessible through the LLM.

**Indirect Prompt Injections** occur when an LLM accepts input from external sources that can be controlled by an attacker, such as websites or files. The attacker may embed a prompt injection in the external content, hijacking the conversation context. This can lead to unstable LLM output, allowing the attacker to manipulate the user or additional systems that the LLM can access. Additionally, indirect prompt injections do not need to be human-visible/readable, as long as the text is parsed by the LLM.
As more companies use LLMs to screen resumes, some websites now offer to add invisible text to your resume, causing the screening LLM to favor your CV


### Consequences

Consequences of prompt injections include malware creation, misinformation dissemination, data leakage, and remote system control, emphasizing the need for preventive actions and human supervision

### Possible Solution

Mitigation strategies involve data curation, adherence to least privilege principle(only gives the capability which is required) , human oversight in decision-making, input filtering, and reinforcement learning from human feedback to enhance model training

- **Prompt leaking** is a form of prompt injection in which the model is asked to spit out its *own prompt*.

**API CALL TO OPEN AI**

```
def set_open_params(
    model="gpt-3.5-turbo",
    temperature=0.7,
    max_tokens=256,
    top_p=1,
    frequency_penalty=0,
    presence_penalty=0,
):
    """ set openai parameters"""
    openai_params = {}
    openai_params['model'] = model
    openai_params['temperature'] = temperature
    openai_params['max_tokens'] = max_tokens
    openai_params['top_p'] = top_p
    openai_params['frequency_penalty'] = frequency_penalty
    openai_params['presence_penalty'] = presence_penalty
    return openai_params
```

model: Specifies the model to use (default is "gpt-3.5-turbo").

temperature: Controls the randomness of predictions (default is 0.7).

max_tokens: Specifies the maximum number of tokens (words or subwords) in the generated response (default is 256).

top_p: Controls the diversity of the generated responses (default is 1).

frequency_penalty and presence_penalty: Used to discourage repetition and encourage diversity in responses (both default to 0).

# IMPLEMENTATION OF RAG

**Chunking technique for better results:**

Different chunking methods:

**Fixed Size Chunking**: This is the most common and straightforward approach to chunking: we simply decide the number of tokens in our chunk and, optionally, whether there should be any overlap between them. In general, we will want to keep some overlap between chunks to make sure that the semantic context doesn't get lost between chunks. Fixed-sized chunking will be the best path in most common cases. Compared to other forms of chunking, fixed-sized chunking is computationally cheap and simple to use since it doesn't require the use of any NLP libraries.

**Recursive Chunking** : Recursive chunking divides the input text into smaller chunks in a hierarchical and iterative manner using a set of separators. If the initial attempt at splitting the text doesn't produce chunks of the desired size or structure, the method recursively calls itself on the resulting chunks with a different separator or criterion until the desired chunk size or structure is achieved. This means that while the chunks aren't going to be exactly the same size, they'll still "aspire" to be of a similar size. Leverages what is good about fixed size chunk and overlap.

**Document Specific Chunking**: It takes into consideration the structure of the document . Instead of using a set number of characters or recursive process it creates chunks that align with the logical sections of the document like paragraphs or sub sections. By doing this it maintains the author's organization of the content thereby keeping the text coherent. It makes the retrieved information more relevant and useful, particularly for structured documents with clearly defined sections. It can handle formats such as Markdown, Html, etc.

**Sematic Chunking**: Semantic Chunking considers the relationships within the text. It divides the text into meaningful, semantically complete chunks. This approach ensures the information's integrity during retrieval, leading to a more accurate and contextually appropriate outcome. It is slower compared to the previous chunking strategy

**three different strategies you could use on Semantic Chunking**:

- `percentile` (default) — In this method, all differences between sentences are calculated, and then any difference greater than the X percentile is split.

- `standard_deviation` — In this method, any difference greater than X standard deviations is split.

- `interquartile` — In this method, the interquartile distance is used to split chunks.

**Agentic Chunk:** The hypothesis here is to process documents in a fashion that humans would do.

- We start at the top of the document, treating the first part as a chunk.
- We continue down the document, deciding if a new sentence or piece of information belongs with the first chunk or should start a new one
- We keep this up until we reach the end of the document.

**Semantic Chunking**

**Definition:** Semantic chunking involves splitting text based on the semantic content, often using embeddings to measure the similarity between chunks. This method typically uses machine learning models to understand the meaning of the text and decide where to split.

**Performance Factors:**

- **Computational Load:** Embedding sentences and calculating similarity scores can be computationally intensive.
- **Latency:** The process of generating embeddings and calculating cosine similarities for each sentence or chunk can slow down the chunking process.
- **Accuracy:** High accuracy in understanding the context and meaning of the text, leading to more coherent chunks.

**Recursive Chunking**

**Definition:** Recursive chunking splits text based on simpler rules, such as paragraphs and sentences, and recursively breaking down chunks until they meet size constraints.

**Performance Factors:**

- **Computational Load:** Generally less computationally intensive compared to semantic chunking, as it relies more on text processing and less on complex embeddings.
- **Latency:** Faster for large texts, as it avoids the overhead of embedding and similarity calculations for each sentence.
- **Accuracy:** May result in less coherent chunks compared to semantic chunking, as it does not consider the semantic meaning as deeply.

**Comparison**

- **Speed:** Recursive chunking is generally faster than semantic chunking because it avoids the computationally intensive steps of generating embeddings and calculating similarities.
- **Complexity:** Recursive chunking is simpler to implement and requires less computational power, making it suitable for real-time applications or when processing large volumes of text.
- **Coherence:** Semantic chunking typically produces more coherent and contextually accurate chunks, but at the cost of speed.

**Conclusion**

- **Use Semantic Chunking:** When the coherence and meaning of chunks are critical, and computational resources are available.
- **Use Recursive Chunking:** When speed is more important and simpler chunking rules suffice, or when working with very large datasets where performance is a concern.

## Searching Algorithms

### Vector Space Model (e.g., TF-IDF, BM25)

**How it works:**

- Represents documents and queries as vectors in a multi-dimensional space.
- Computes similarity using cosine similarity, Euclidean distance, etc.

**Pros:**

- Effective for text search.
- Can handle a variety of text features.

**Cons:**

- Requires feature extraction and vector representation.

### Inverted Index (e.g., used in search engines like Elasticsearch)

**How it works:**

- Creates an index that maps terms to the documents they appear in.
- Queries are processed by looking up terms in the index.

**Pros:**

- Very efficient for text search.
- Supports complex queries and Boolean operations.

**Cons:**

- Indexing process can be time-consuming.
- Requires storage for the index.

**Approximate Nearest Neighbor (ANN) Search (e.g., FAISS, Annoy)**

**How it works:**

- Uses data structures and algorithms designed to find approximate nearest neighbors.
- Commonly used with high-dimensional vector representations (e.g., embeddings).

**Pros:**

- Very efficient for large datasets.
- Suitable for similarity search in high-dimensional spaces.

**Cons:**

- Provides approximate results (may not always be exact).
- Requires preprocessing to build the index.

**Clustering-Based Search (e.g., K-means clustering)**

**How it works:**

- Groups documents into clusters based on their features.
- Queries are first matched to the nearest cluster, then search is performed within the cluster.

**Pros:**

- Reduces search space, improving efficiency.
- Can handle large datasets effectively.

**Cons:**

- Requires clustering preprocessing.
- Quality of results depends on clustering quality.

**Example Comparison**

Imagine you have a dataset of 1 million documents and you want to perform a text search:

- **Linear Search:** Each search will iterate through all 1 million documents, which is slow.
- **Binary Search:** Not applicable unless the documents are sorted by some criteria.
- **Hash-Based Search:** Fast for exact match queries but not for similarity search.

- **Tree-Based Search:** Efficient if data is structured appropriately, but may struggle with high-dimensional text data.
- **Vector Space Model:** Effective but requires computing similarity for each document.
- **Inverted Index:** Very fast and efficient for text search, commonly used in search engines.
- **ANN Search:** Efficient for high-dimensional data, providing approximate results quickly.
- **Clustering-Based Search:** Reduces search space by narrowing down to a specific cluster, balancing efficiency and accuracy.

**Choosing the Right Algorithm**

- **Small datasets:** Linear search or simple inverted index might suffice.
- **Large text corpora:** Inverted index or ANN search.
- **High-dimensional embeddings:** ANN search with libraries like FAISS.
- **Need for speed:** ANN search or hash-based search.
- **Structured data:** Tree-based search

## Search Methods for ANN(used by FAISS)

- **Inverted File Index (IVF) with Clustering:**
  - **Clustering**: Uses k-means clustering to partition the dataset into clusters.
  - **Search**: During query, finds the nearest cluster centroids first, then refines search within those clusters.
- **Hierarchical Navigable Small World (HNSW)**:
  - **Clustering**: Does not use traditional clustering; builds a graph structure.
  - **Search**: Navigates through a hierarchical graph structure to find nearest neighbors.
- **Product Quantization (PQ)**:
  - **Clustering**: Does not use clustering; decomposes vectors into subvectors.
  - **Search**: Uses quantized subvector codes to approximate distances between query and database vectors.
- **Locality-Sensitive Hashing (LSH)**:
  - **Clustering**: Does not use clustering; hashes vectors into buckets.
  - **Search**: Searches within hashed buckets to find approximate nearest neighbors.
- **IVF + PQ**: Combines inverted file index with product quantization for efficient and compressed search.
- **IVF + SQ**: Combines inverted file index with scalar quantization for similar benefits.

## FAISS

- **Storage Focus**: FAISS primarily focuses on storing embeddings (vector representations) efficiently in its index structures.
- **Purpose**: It is designed for efficient similarity search and clustering of high-dimensional vectors.
- **Data Storage**: FAISS indexes are optimized for fast retrieval of nearest neighbors based on vector similarity metrics like Euclidean distance or cosine similarity.
- **Usage**: Typically used in machine learning and information retrieval tasks where vector representations (embeddings) are central, such as image search, recommendation systems, and natural language processing.

## Chroma DB

- **Storage Capabilities**: Chroma DB, on the other hand, provides comprehensive document storage capabilities.
- **Data Stored**: It can store entire documents, including their content, metadata, and optionally embeddings.
- **Query Flexibility**: Supports querying based on both document content and metadata attributes, allowing for more complex search and retrieval operations.
- **Usage**: Ideal for applications where managing complete documents, associating metadata, and performing diverse queries (including text search and metadata-based filtering) are crucial, such as content management systems and search engines
- **Similarity Metric used**: Chroma uses cosine distance with **range 0-2**
- **Embedding model used:** We learned that the information stored in Vector Databases is in the form of Vector Embeddings. But here, we provided text/text files i.e. documents. So how does it store them? Chroma DB by default, uses an all-MiniLM-L6-v2 vector embedding model to create the embeddings for us. This model will take our documents and convert them into vector embeddings. If we want to work with a specific embedding function like other sentence-transformer models from HuggingFace or OpenAI embedding model, we can specify it under the embeddings_function=embedding_function_name variable name in the create_collection() method.
- **Pre Filtering**: Chroma use pre filtering by default , it filters first on the basis of where clause and then returns n_results from the result we get after applying where filter
- **Post Filtering**: If we want post filtering then we get the results by collection.query then apply the filter afterwards like if we want to sort then we sort separately

## Key Differences

1. **Data Focus**:
    - FAISS: Focuses on storing and indexing high-dimensional vectors (embeddings).
    - Chroma DB: Focuses on storing complete documents along with metadata and optionally embeddings.
2. **Query Capabilities**:
    - FAISS: Optimized for nearest neighbor search based on vector similarity.
    - Chroma DB: Supports a broader range of queries including text search, metadata filtering, and more complex retrieval operations.

3. **Integration**:
    - FAISS is often integrated into applications where fast similarity search over embeddings is critical.
    - Chroma DB is used in applications requiring comprehensive document storage, retrieval, and management capabilities

## SMALL LANGUAGE MODELS

Llama 3 8 billion version, Mistral 7 billion

**Quantization of a large language model** (LLM) like GPT involves converting its parameters (weights and biases) from floating-point numbers (which typically require 32 bits or more) to lower bit-width integers (like 8 bits). This process offers several benefits, particularly in the context of deployment and inference efficiency:

1. **Reduced Memory Footprint:**
    - Quantization reduces the amount of memory required to store the model's parameters. For example, converting from 32-bit floating-point numbers to 8-bit integers can reduce memory usage by a factor of 4.
2. **Improved Inference Speed:**
    - With reduced memory requirements, quantized models can be loaded faster into memory, leading to quicker inference times. This is crucial for real-time applications or scenarios where rapid responses are required.
3. **Lower Computational Cost:**
    - Quantization reduces the computational cost during inference. Processing lower bit-width integers (e.g., 8-bit) is generally faster than processing higher precision floating-point numbers (e.g., 32-bit), leading to improved efficiency on hardware accelerators like GPUs and TPUs.
4. **Deployment on Resource-Constrained Devices:**
    - Smaller model size and reduced computational demands make quantized models suitable for deployment on resource-constrained devices such as mobile phones, IoT devices, or edge devices. This extends the reach of sophisticated language models to environments where computational resources are limited.
5. **Energy Efficiency:**
    - Lower computational cost translates to reduced energy consumption during inference, which is beneficial for both mobile devices and data centers aiming to optimize energy usage.
6. **Compatibility with Hardware Accelerators:**
    - Many hardware accelerators (like TPUs and some GPUs) are optimized for operations on quantized data. Using quantized models can leverage these optimizations to further improve performance.

However, quantization is not without challenges. It may introduce some loss of model accuracy due to reduced precision, although techniques like fine-tuning after quantization or using specialized quantization methods can mitigate this impact. Overall, the benefits of quantization—smaller model size, faster inference, and improved efficiency—often outweigh

the potential drawbacks, especially in production and deployment scenarios where optimization for speed and resource usage is critical

**GGUF is a quantization method**

(No of parameters * no of bits for each node)/8 = vram required to run the model

Generally its 32 bits after quantization it becomes 4

Instead of using a LLM with less no of parameters and high precison 32 bits use LLM with large no of parameters and low precision

When a model is quantized to 8 bits, it means that the weights of the model, which are typically represented as 32-bit or 64-bit floating-point numbers, are approximated using 8-bit integers.

**Quantizing and rounding the values**

$$\mathbf{X}_{\text{quant}} = \text{round}\left(\frac{127}{\max|\mathbf{X}|} \cdot \mathbf{X}\right)$$

$$\mathbf{X}_{\text{dequant}} = \frac{\max|\mathbf{X}|}{127} \cdot \mathbf{X}_{\text{quant}}$$

Mapping Float point 32 bit nos to INT 8:

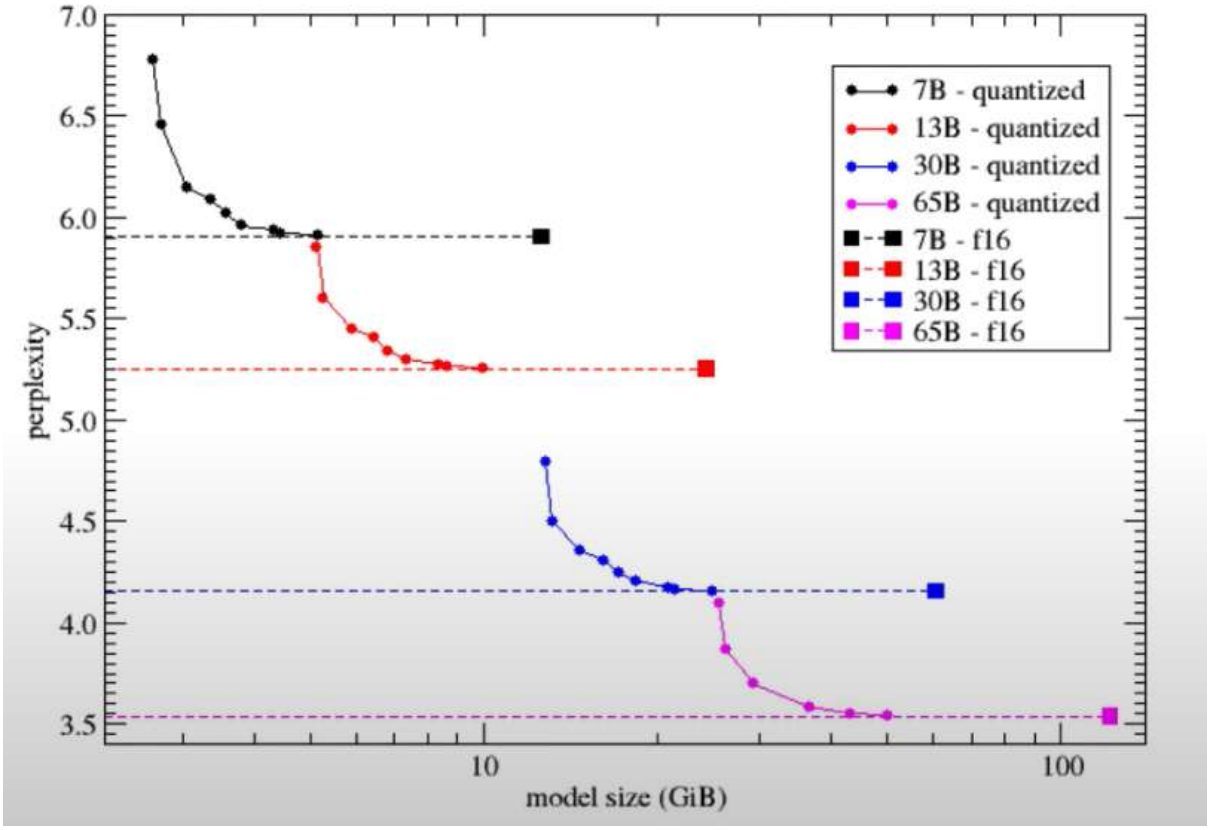
| Precision | Dynamic range | # possible values |
|---|---|---|
| FP32 | $-3.4 \times 10^{38} \rightarrow 3.4 \times 10^{38}$ | $4.2 \times 10^{9}$ |
| FP16 | $-65\,504 \rightarrow 65\,504$ | $61.4 \times 10^{3}$ |
| INT8 | $-128 \rightarrow 127$ | 256 |

## Understanding Bit Precision

1. **8-bit Representation**:
    - With 8-bit representation, each parameter **(weight)** can take one of 2^8=256 different values.
    - This allows for a finer granularity in representing model weights and activations, leading to higher precision.
2. **4-bit Representation**:
    - With 4-bit representation, each parameter can take one of 2^4 = 16 different values.
    - This reduces the granularity, leading to lower precision as fewer distinct values can be represented.

## Impact on Model Precision

1. **Quantization Error**:
    - Quantization involves mapping a large set of values to a smaller set. Reducing the bit-width increases the quantization error because the distance between representable values becomes larger.
    - Higher quantization error can lead to less accurate representation of the model's learned parameters, potentially degrading the model's performance.
2. **Model Performance**:
    - Lower precision can result in a loss of important details in the model's weights and activations, which can affect the model's ability to make accurate predictions.
    - The extent of the performance degradation depends on the model architecture and the specific task. Some models and tasks are more robust to lower precision than others.
3. **Benefits of Lower Precision**:
    - **Reduced Memory Usage**: Lower precision reduces the amount of memory required to store the model. This is particularly important for deploying models on devices with limited memory.
    - **Faster Computation**: Operations with lower precision can be faster, as they require less computational resources. This can lead to faster inference times.
    - **Energy Efficiency**: Lower precision operations consume less power, which is beneficial for deploying models on battery-powered devices




## LLAMA 3 (8 billion parameters)

- The model's architecture is "llama", which is a type of transformer architecture.
- The model uses 32 attention heads and 8 attention heads for key and value projections.
- The model has 32 transformer blocks and a maximum sequence length of **8192(context window)** tokens.
- The input and **output embeddings have a dimensionality of 4096**, and the feedforward network has a dimensionality of 14336.
- The model uses rotary position embeddings with 128 dimensions and a base frequency of 500000.
- The model's **vocabulary size is 128256 tokens**.
- The tokenizer uses byte-pair encoding (BPE) with a prefix of "llama-bpe" and 280147 merge operations.
- The beginning-of-sentence token has an ID of 128000, and the end-of-sentence token has an ID of 128009.
- The model's weights are stored in a binary format with a specific quantization scheme.
- The metadata provides details about the model's configuration and capabilities, which can be useful for ensuring compatibility with specific use cases and hardware.

| | Mistral 7B Instruct | Llama 3 8B Instruct |
|---|---|---|
| Provider<br>The entity that provides this model. | Mistral | Meta |
| Input Context Window<br>The number of tokens supported by the input context window. | 32K<br>tokens | 8,000<br>tokens |
| Maximum Output Tokens<br>The number of tokens that can be generated by the model in a single request. | 8,192<br>tokens | 2,048<br>tokens |
| Release Date<br>When the model was first released. | September 27th, 2023<br>10 months ago | April 18th, 2024<br>3 months ago |

**Running LLAMA 3 Locally on VSCODE:**

You're correct in observing that using ollama.chat() in Python can sometimes be slower compared to interacting with the model directly through the terminal. There are a few reasons for this and some potential solutions:

**Reasons for slower response:**

- Initialization overhead: Each call to ollama.chat() may involve some initialization, which can add latency.
- Python overhead: There's additional overhead from the Python runtime and the Ollama Python library.
- Cold start: If the model isn't already loaded in memory, the first call might be significantly slower.
- Network communication: The Python library communicates with the Ollama service over a local network connection, which can add some latency.

**Solutions and optimizations:**

- Keep the model loaded:
- Use the ollama serve command to keep the model loaded in memory.
- Use a persistent session:
- Instead of creating a new chat for each interaction, maintain a session:

```
import ollama

# Create a persistent chat session

chat = ollama.chat(model="llama3:8b-instruct-q4_K_M")

# First interaction

response = chat(messages=[{"role": "user", "content": "Hello"}])

print(response['message']['content'])
```

# Subsequent interactions

response = chat(messages=[{"role": "user", "content": "Tell me more"}])

print(response['message']['content'])

**Batch processing:**

If you have multiple queries, consider sending them in a batch rather than one at a time.

**Use streaming:**

For longer responses, you can use streaming to start getting the response faster:

for chunk in ollama.chat(

    model='llama3:8b-instruct-q4_K_M',

    messages=[{'role': 'user', 'content': 'Why is the sky blue?'}],

    stream=True):

    print(chunk['message']['content'], end='', flush=True)

## Creating a prompt for LLAMA 3:

## General format

<|begin_of_text|><|start_header_id|>system<|end_header_id|>

You are a helpful AI assistant for travel tips and recommendations<|eot_id|><|start_header_id|>user<|end_header_id|>

What can you help me with?<|eot_id|><|start_header_id|>assistant<|end_header_id|>

**<|begin_of_text|>**: This is equivalent to the BOS token

**<|eot_id|>**: This signifies the end of the message in a turn.

**<|start_header_id|>{role}<|end_header_id|>**: These tokens enclose the role for a particular message. The possible roles can be: system, user, assistant.

**<|end_of_text|>**: This is equivalent to the EOS token. On generating this token, Llama 3 will cease to generate more tokens.

A prompt can optionally contain a single system message, or multiple alternating user and assistant messages, but always ends with the last user message followed by the assistant header.

**Example**

```
response2 = ollama.chat(model='llama3:8b-instruct-q4_K_M', messages=[
    {
        'role': 'system',
        'content': f"Using the document provided below, strictly provide the
answer to the question. If you can not find an answer in the document, respond
with 'I am sorry. I don't know the answer.' \n\n***{result_string}***\n\n"
    },
    {
        'role': 'user',
        'content': f"Question :{query3}"
    }
])
```

**For setting parameters for a model in ollama:**

```
response2 = ollama.chat(
    model='llama3:8b-instruct-q4_K_M',
    messages=[
        {
            'role': 'system',
            'content': f"Using the document provided below, strictly provide
the answer to the question. If you can not find an answer in the document,
respond with 'I am sorry. I don't know the answer.'
\n\n***{result_string}***\n\n"
        },
        {
            'role': 'user',
            'content': f"Question :{query3}"
        }
    ],
    options={
        'temperature': 0.7,
        'top_k': 40,
        # Add other parameters as needed
    }
)
```

## Model Size

**1B parameters**: pattern matching and basic knowledge of world

Restaurant Review Sentiment

**10B parameters**: greater world knowledge. Can follow basic instructions

Food Order Chatbot

**100B parameters** Rich world knowledge. Complex Reasoning

Brainstorming Partner

## Open source or Closed Source:

**Closed source:**

- Easy to use
- More large/powerful
- Relatively inexpensive
- Some risk of vendor lock in

**Open Source**

- Full control over model
- Can run on your own device
- Full control over data privacy

## What is Fine-tuning of LLM Models?

We have a bunch of open-source available pre-trained LLM models. We can find several transformers in Huggingface's search space. But all of these models were trained for a particular task and then the saved model is uploaded to Huggingface. So to use these transformer architectures according to the use case, we fine-tune or tweak some of their parameters and then use them.

so fine-tuning can be defined as the process which allows LLMs to adapt to specific tasks by adjusting their parameters, making them suitable for NLP (Natural Language Processing) related tasks. They are built upon the pre-trained knowledge of the model.

## Why Fine Tune:

**1-To carry out a task that is not easy to define in a prompt**

**Example**

Summarize in certain style or structure. Customer support want summarization of chat log in certain manner. So, we will train the model on 100s of human written summarization of specific style and let LLM train on it. You could do it with the help of few shot prompting but fine tuning makes it more precise

Mimicking a writing or speaking style It is not easy to describe the speaking or writing style of one person in a prompt. So, we fine tune on the transcript of speaking style or writing style of a person.

**2- To help LLM gain specific Knowledge**

**Example**

Medical Notes If we want the LLM to read and process medical notes which are not normal for LLM. Because the medical notes may contain such terms, that maybe shortcuts for different diseases unknown to LLM.

Legal Documents, Financial Documents: reading and processing these documents

**3-To get a smaller model to perform a task**

The purpose of this to get a smaller model to do the task that previously would have required a larger model. Using a larger model 100billion+ -> higher cost, usage requires highly efficient devices

Small models 1billion + -> Lower cost/ latency to deploy, can run on mobile/ laptop

**Example**

If we want to classify reviews in positive or negative, we don't need larger model for this. Smaller models are not that good. So, we fine tune smaller models on data set

**PEFT — Parameter Efficient Fine Tuning** (for Billion scale models and low-resource hardware)

**Why PEFT?**

**Issues with Full Fine-Tuning:**

**We end up with huge weights to train.**

Needs more computation power to train: With the increase in size and depth of the model, we need much bigger and multiple GPUs to train and fine-tune them.

**Increase in size of file**

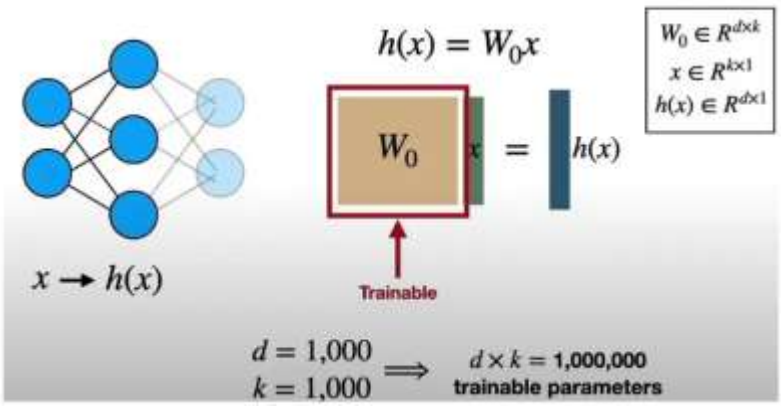
- Difficulty in portability of the model because full fine-tuning of LLMs according to earlier approaches results in large checkpoints which take up to GBs of storage.

- For eg: Google's T5-XXL, and Bigscience's mt0-XXL, are different transformer models which take up to 40–50 GB of storage. And so fully fine-tuning these models will lead to 40–50 GB checkpoints for each downstream dataset. These Models have huge parameters easily varying from 10–20B parameters. These models are trained with slightly higher learning rates(1e-4 and 3e-4).

**PEFT Methods:**

- LoRA
- Prefix tuning
- P tuning
- Prompt Tuning
- QLoRA

> *We will be focusing only on LoRA and QLoRA in this Blog. Will discuss about remainig methods in future blogs.*



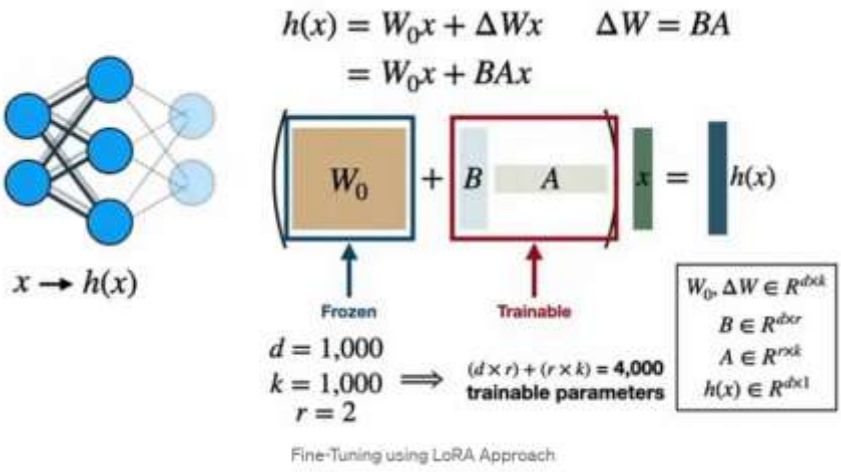

In the normal fine-tuning of the model without using PEFT, the hidden layer takes weight W0 which has (d x k) trainable parameters when x takes (k x 1) and h(x) takes (d x 1) parameters. With this being said, its clear that the size of parameters is so huge and so pre-training and fine-tuning the model gets more and more difficult.

## LoRA — Low Rank Adaption Method

**Working of LoRA:**

- It is a 16 bit transformer.
- It allows us to fine-tune only a small number of extra weights in the model while we freeze most of the parameters of the pre-trained network.
- So we are not training the original weights. Instead, we add some extra weights to the model and then we train those.

- The advantage is that we still have the original weights. This also tends to help with stopping catastrophic forgetting.
- Catastrophic forgetting: It is when models tend to forget what they were trained on when we attempt to do fine-tuning. This happens when we fine-tune too much.



Fine-Tuning using LoRA Approach

- In the fine-tuning of the model without using LoRA, the hidden layer takes weight (W0+∇W) where the original weight W0 is kept idle, i.e. we don't use them in our fine-tuning step and hence is kept frozen.
- ∇W is our newly added weight where ∇W = BA, where B and A are 2 matrices with B having (d x r) dimension and A having (r x k) dimension.
- r : the rank of the update matrices, expressed in 'int'. Lower rank results in smaller update matrices with fewer trainable parameters. So over here we have taken r as 2, which is a smaller number and hence we will have fewer no.of.trainable parameters.
- Now we have only ((d + k) x r) trainable parameters which are less than the original (d x k) parameters

## QLORA:

**How is it different from LoRA:**

- It is a 4-bit transformer.
- QLoRA is a finetuning technique that combines a high-precision computing technique with a low-precision storage method. This helps keep the model size small while still making sure the model is still highly performant and accurate.
- QLoRA uses LoRA as an accessory to fix the errors introduced during the quantization errors.

**Working of QLoRA:**

QLoRA works by introducing 3 new concepts that help to reduce memory while retaining the same quality performance. These are 4-bit Normal Float, Double Quantization, and Paged Optimizers.

**4-bit Normal Float (NF4):**

- 4-bit NormalFloat is a new data type and a key ingredient to maintaining 16-bit performance levels. Its main property is this: Any bit combination in the data type, e.g. 0011 or 0101, gets assigned an equal number of elements from an input tensor.
- 4-bit Quantization of weights and PEFT and train injected adapter weights (LORA) in 32-bit precision.
- QLoRA has one storage data type (NF4) and a computation data type (16-bit BrainFloat).
- We dequantize the storage data type to the computation data type to perform the forward and backward pass, but we only compute weight gradients for the LORA parameters which use 16-bit BrainFloat.

**QLoRA — Quantized Low Rank Adaption Method**

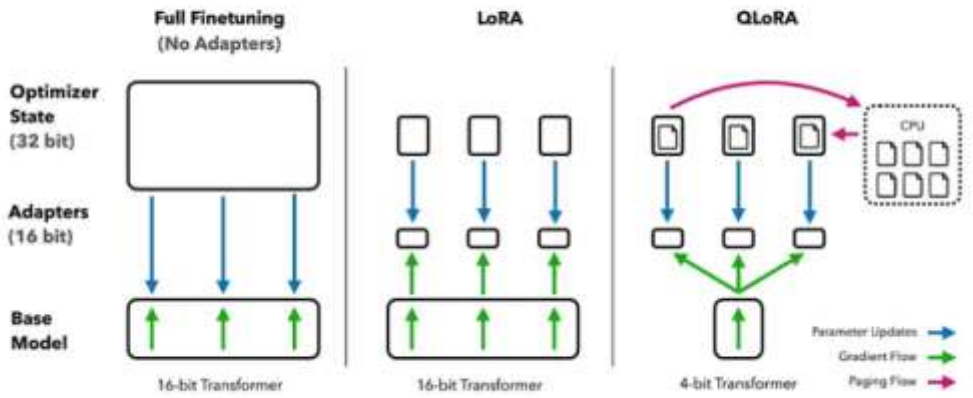



**Figure 1:** Different finetuning methods and their memory requirements. QLoRA improves over LoRA by quantizing the transformer model to 4-bit precision and using paged optimizers to handle memory spikes.

**1. Normalization**: The weights of the model are first normalized to have zero mean and unit variance. This ensures that the weights are distributed around zero and fall within a certain range.

**2. Quantization**: The normalized weights are then quantized to 4 bits. This involves mapping the original high-precision weights to a smaller set of low-precision values. In the case of NF4, the quantization levels are chosen to be evenly spaced in the range of the normalized weights.

**3. Dequantization**: During the forward pass and backpropagation, the quantized weights are dequantized back to full precision. This is done by mapping the 4-bit quantized values back to their original range. The dequantized weights are used in the computations, but they are stored in memory in their 4-bit quantized form.

# Pre-Training an LLM:

The LLMs we use are often pre trained by companies.

Very Expensive

Many teams are pre training general purpose LLMs by learning from internet text. May take 10s of millions many months, huge amount of data

## When should you pre train:

**For building a specific application:**

- Option of last resort
- Could help if have a highly specialized domain

Use someone's else pre trained LLM, and then fine tune it instead

## Instruction Tuning (essentially part of model training but can be used for in fine tuning)

How does LLMs learn to follow instructions

Fine tune the model on examples, such examples where the answers to questions are in the form of what you want to get out of LLM as a result of your instruction

If you will train the LLM on prompts and the responses, the LLM will learn not only to predict the next word but also answer your questions and follow the instructions

## Reinforcement Learning from Human Feedback (essentially part of model training but can be used for in fine tuning):

Give LLM response in the form of triple H: Helpful, Honest, Harmless

**Step 1:** Train an answer quality (reward) model

Prompt: Advise me how to apply for a job?

LLM can give multiple responses. We can get humans to score the response of LLM.

Now, we can use supervised learning to train the model on responses as input and generate scores from it

**Step 2:** Have the LLM generate a lot of answers. Further train it to generate more responses that get high scores

# Langchain related

## OPEN AI Function Calling:

```
def get_current_weather(location, unit="fahrenheit"):
    """Get the current weather in a given location"""
    weather_info = {
        "location": location,
        "temperature": "72",
        "unit": unit,
        "forecast": ["sunny", "windy"],
    }
    return json.dumps(weather_info)
functions = [
    {
        "name": "get_current_weather",
        "description": "Get the current weather in a given location",
        "parameters": {
            "type": "object",
            "properties": {
                "location": {
                    "type": "string",
                    "description": "The city and state, e.g. San Francisco, CA",
                },
                "unit": {"type": "string", "enum": ["celsius", "fahrenheit"]},
            },
            "required": ["location"],
        },
    }
```

]

Function contains list of functions. Name is the name of function, description is the relevant info to function which is passed to the LLM. It is very important as it tells the LLM, how to use the function or when to use the function. Properties part contains the parameters for the function.

We still have to call the function, ourselves the model can only return the arguments for us which we need to send in function. In case of OPEN AI, when we send a prompt relevant to function we get arguments back otherwise we simply get back a normal response. By default, the function call parameter is set to “auto” meaning the model automatically determines whether to call fun or not. If we want to force the model to use the fun we use function_call = {"name": "get_current_weather"}. If we don’t want the model to use function we use auto= ”none”. If for a prompt, the function was to be called but we set auto= ”none”. It can try to call the function, but it will not return the arguments in desired format. Same case if we set function to be called for a prompt where it should not be called. We get the arguments return but the model is confused. For example, in case of weather it will return some city name as argument as return which we do not need

**Function call where we asked for weather in prompt**

```
]
response = openai.ChatCompletion.create(
    model="gpt-3.5-turbo-0613",
    messages=messages,
    functions=functions,
    function_call={"name": "get_current_weather"},
)
print(response)
```


```
{
  "id": "chatcmpl-89yPS9aCxDRwca6WGmtI5zCCtFg2F",
  "object": "chat.completion",
  "created": 1697387858,
  "model": "gpt-3.5-turbo-0613",
  "choices": [
    {
      "index": 0,
      "message": {
        "role": "assistant",
        "content": null,
        "function_call": {
          "name": "get_current_weather",
          "arguments": "{\n\"location\": \"Boston\"\n}"
        }
```


Function definition and function description increases the input prompt tokens

observation = get_current_weather(args)

messages.append(

{

```
        "role": "function",
        "name": "get_current_weather",
        "content": observation,
    }
)
```

Here we send the output we get from function to calling to LLM, for further response

Tool calling allows a chat model to respond to a given prompt by generating output that matches a user-defined schema.

While the name implies that the model is performing some action, this is actually not the case! The model only generates the arguments to a tool, and actually running the tool (or not) is up to the user

**Tools:**

- In LangChain, a tool is a function or capability that can be used by a language model to perform specific tasks.
- Tools are typically predefined functions that extend the capabilities of the language model.
- Examples include web search, calculator, database queries, or API calls.
- Tools are defined and registered within the LangChain framework.

Tools are utilities designed to be called by a model: their inputs are designed to be generated by models, and their outputs are designed to be passed back to models. Tools are needed whenever you want a model to control parts of your code or call out to external APIs.

A tool consists of:

- The name of the tool.
- A description of what the tool does.
- A JSON schema defining the inputs to the tool.
- A function (and, optionally, an async variant of the function).

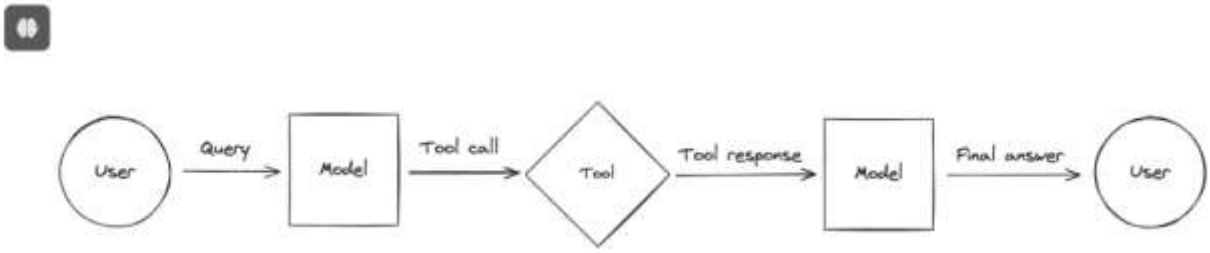
**Tool Calls:**

- A tool call refers to the actual invocation or use of a tool by the language model during execution.
- It's the process of the model deciding to use a specific tool and actually executing that tool's functionality.

- Tool calls are typically part of the model's decision-making process when solving tasks or answering queries.

The general flow is this:

1. Generate tool calls with a chat model in response to a query.
2. Invoke the appropriate tools using the generated tool call as arguments.
3. Format the result of the tool invocations as `ToolMessages`.
4. Pass the entire list of messages back to the model so that it can generate a final answer (or call more tools).

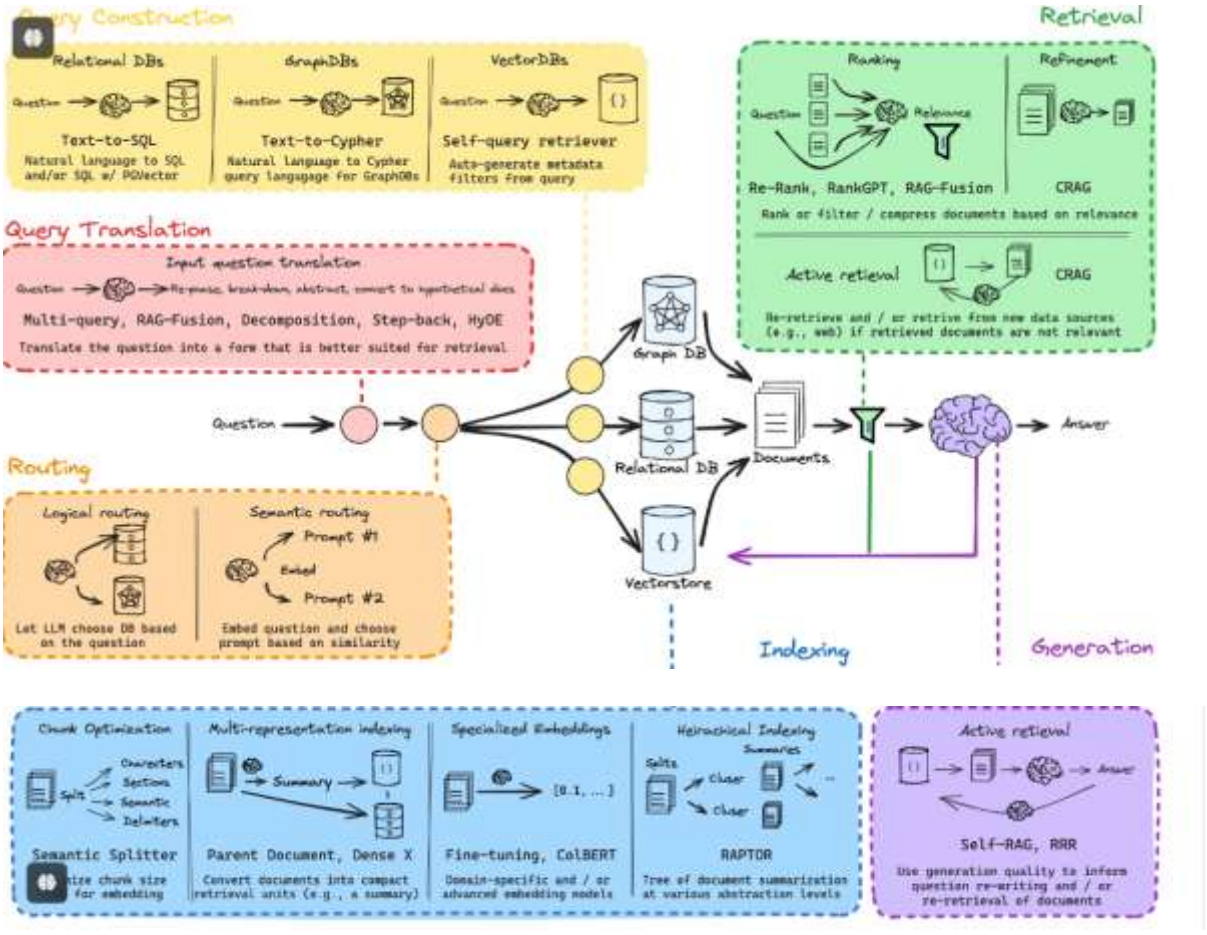



## Structured Output:

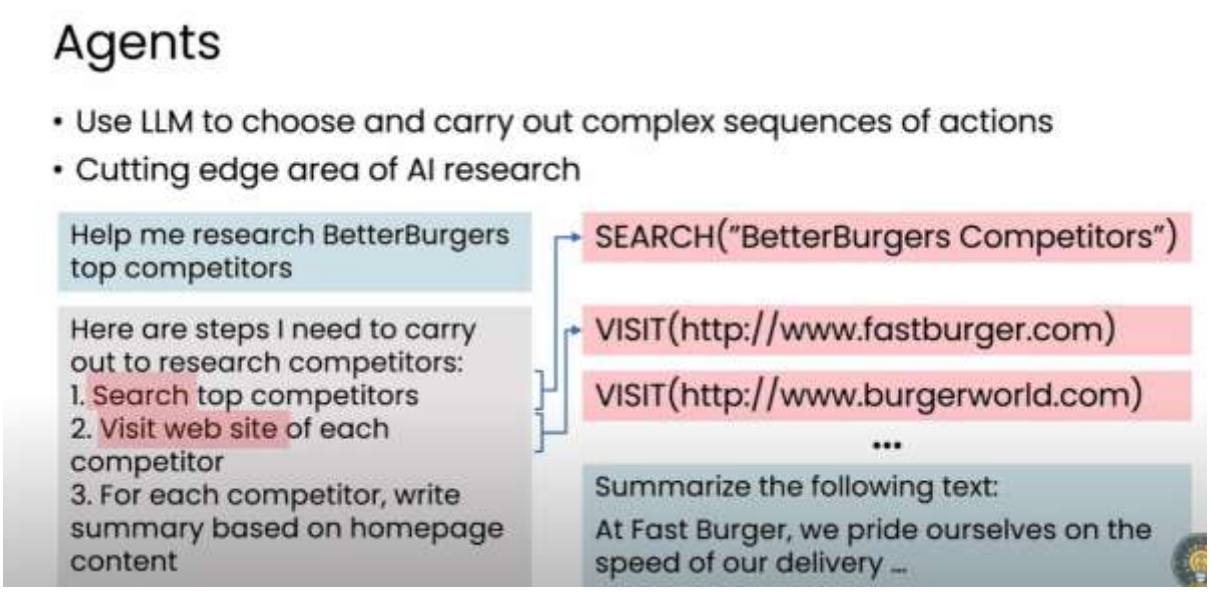
To get output from LLM of specific we can simply tell the LLM in prompt(raw prompting) how to return the answer or we can use tool calling to get answer in the required format

For convenience, some LangChain chat models support a .with_structured_output() method. This method only requires a schema as input, and returns a dict or Pydantic object

## Retrieval in LANGCHAIN




## AI AGENTS



An artificial intelligence (AI) agent is a software program that can interact with its environment, collect data, and use the data to perform self-determined tasks to meet predetermined goals. Humans set goals, but an AI agent independently chooses the best actions it needs to perform to achieve those goals. For example, consider a contact center AI agent that wants to resolves customer queries. The agent will automatically ask the customer different questions, look up information in internal documents, and respond with a solution. Based on the customer responses, it determines if it can resolve the query itself or pass it on to a human.

AI agents are rational agents. They make rational decisions based on their perceptions and data to produce optimal performance and results. An AI agent senses its environment with physical or software interfaces.

For example, a robotic agent collects sensor data, and a chatbot uses customer queries as input. Then, the AI agent applies the data to make an informed decision. It analyses the collected data to predict the best outcomes that support predetermined goals. The agent also uses the results to formulate the next action that it should take. For example, self-driving cars navigate around obstacles on the road based on data from multiple sensors.

## What are the benefits of using AI agents?

AI agents can improve your business operations and your customers' experiences.

### Improved productivity

AI agents are autonomous intelligent systems performing specific tasks without human intervention. Organizations use AI agents to achieve specific goals and more efficient business outcomes. Business teams are more productive when they delegate repetitive tasks to AI agents. This way, they can divert their attention to mission-critical or creative activities, adding more value to their organization.

### Reduced costs

Businesses can use intelligent agents to reduce unnecessary costs arising from process inefficiencies, human errors, and manual processes. You can confidently perform complex tasks because autonomous agents follow a consistent model that adapts to changing environments.

### Informed decision-making

Advanced intelligent agents use machine learning (ML) to gather and process massive amounts of real-time data. This allows business managers to make better predictions at pace when strategizing their next move. For example, you can use AI agents to analyze product demands in different market segments when running an ad campaign.

### Improved customer experience

Customers seek engaging and personalized experiences when interacting with businesses. Integrating AI agents allows businesses to personalize product recommendations, provide prompt responses, and innovate to improve customer engagement, conversion, and loyalty.

## What are the key components of AI agent architecture?

Agents in artificial intelligence may operate in different environments to accomplish unique purposes. However, all functional agents share these components.

### Architecture

Architecture is the base the agent operates from. The architecture can be a physical structure, a software program, or a combination. For example, a robotic AI agent consists of actuators, sensors, motors, and robotic arms. Meanwhile, an architecture that hosts an AI software agent may use a text prompt, API, and databases to enable autonomous operations.

### Agent function

The agent function describes how the data collected is translated into actions that support the agent's objective. When designing the agent function, developers consider the type of information, AI capabilities, knowledge base, feedback mechanism, and other technologies required.

### Agent program

An agent program is the implementation of the agent function. It involves developing, training, and deploying the AI agent on the designated architecture. The agent program aligns the agent's business logic, technical requirements, and performance elements.

## How does an AI agent work?

AI agents work by simplifying and automating complex tasks. Most autonomous agents follow a specific workflow when performing assigned tasks.

### Determine goals

The AI agent receives a specific instruction or goal from the user. It uses the goal to plan tasks that make the final outcome relevant and useful to the user. Then, the agent breaks down the goal into several smaller actionable tasks. To achieve the goal, the agent performs those tasks based on specific orders or conditions.

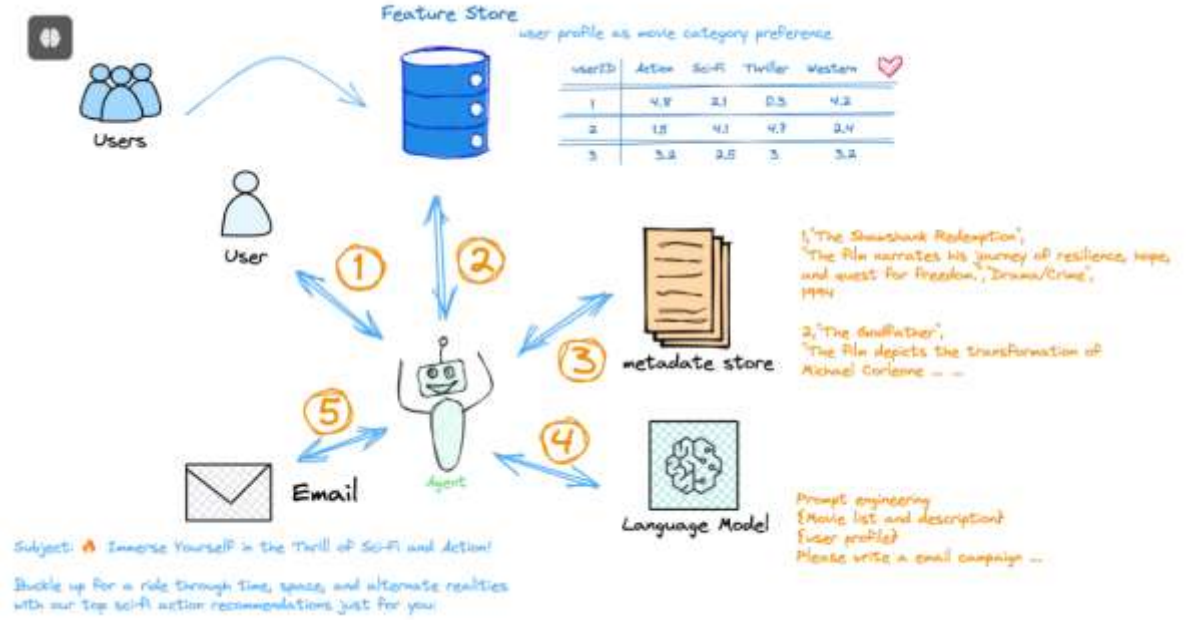
### Acquire information

AI agents need information to act on tasks they have planned successfully. For example, the agent must extract conversation logs to analyze customer sentiments. As such, AI agents might access the internet to search for and retrieve the information they need. In some applications, an intelligent agent can interact with other agents or machine learning models to access or exchange information.

### Implement tasks

With sufficient data, the AI agent methodically implements the task at hand. Once it accomplishes a task, the agent removes it from the list and proceeds to the next one. In

between task completions, the agent evaluates if it has achieved the designated goal by seeking external feedback and inspecting its own logs. During this process, the agent might create and act on more tasks to reach the final outcome.




## What are the types of AI agents?

Organizations create and deploy different types of intelligent agents. We share some examples below.

**Simple reflex agents**

A simple reflex agent operates strictly based on predefined rules and its immediate data. It will not respond to situations beyond a given event condition action rule. Hence, these agents are suitable for simple tasks that don't require extensive training. For example, you can use a simple reflex agent to reset passwords by detecting specific keywords in a user's conversation.

**Model-based reflex agents**

A model-based agent is similar to simple reflex agents, except the former has a more advanced decision-making mechanism. Rather than merely following a specific rule, a model-based agent evaluates probable outcomes and consequences before deciding. Using supporting data, it builds an internal model of the world it perceives and uses that to support its decisions.

**Goal-based agents**

Goal-based agents, or rule-based agents, are AI agents with more robust reasoning capabilities. Besides evaluating the environment data, the agent compares different approaches to help it achieve the desired outcome. Goal-based agents always choose the most

efficient path. They are suitable for performing complex tasks, such as natural language processing (NLP) and robotics applications.

**Utility-based agents** A utility-based agent uses a complex reasoning algorithm to help users maximize the outcome they desire. The agent compares different scenarios and their respective utility values or benefits. Then, it chooses one that provides users with the most rewards. For example, customers can use a utility-based agent to search for flight tickets with minimum traveling time, irrespective of the price.

**Learning agents**

A learning agent continuously learns from previous experiences to improve its results. Using sensory input and feedback mechanisms, the agent adapts its learning element over time to meet specific standards. On top of that, it uses a problem generator to design new tasks to train itself from collected data and past results.

**Hierarchical agents**

Hierarchical agents are an organized group of intelligent agents arranged in tiers. The higher-level agents deconstruct complex tasks into smaller ones and assign them to lower-level agents. Each agent runs independently and submits a progress report to its supervising agent. The higher-level agent collects the results and coordinates subordinate agents to ensure they collectively achieve goals.

These are some of the benefits of Langchain agents:

- Automation: LangChain agents can automate repetitive tasks, freeing you up for more strategic work. Imagine an agent that automatically summarizes customer emails, generates reports, or even schedules meetings based on your preferences.
- Flexibility: Unlike traditional code-based automation, LangChain agents can handle dynamic situations and adapt their actions based on new information or user feedback.
- Reduced development time: LangChain's intuitive framework and pre-built tools make it easier to build and deploy agents without needing extensive coding expertise.
- Improved user experience: By interacting with agents through natural language, users can access information and complete tasks in a more intuitive and efficient way.

## Langchain agents

- By themselves, language models can't take actions - they just output text. A big use case for LangChain is creating agents. Agents are systems that use an LLM as a reasoning engine to determine which actions to take and what the inputs to those actions should be. The results of those actions can then be fed back into the agent and it determine whether more actions are needed, or whether it is okay to finish.
- LangGraph is an extension of LangChain specifically aimed at creating highly controllable and customizable agents. Please check out that documentation for a more in depth overview of agent concepts.

**ReAct agents**

One popular architecture for building agents is **ReAct**. ReAct combines reasoning and acting in an iterative process - in fact the name "ReAct" stands for "Reason" and "Act".

The general flow looks like this:

- The model will "think" about what step to take in response to an input and any previous observations.
- The model will then choose an action from available tools (or choose to respond to the user).
- The model will generate arguments to that tool.
- The agent runtime (executor) will parse out the chosen tool and call it with the generated arguments.
- The executor will return the results of the tool call back to the model as an observation.
- This process repeats until the agent chooses to respond.

## Langgraph

langgraph is an extension of langchain aimed at building robust and stateful multi-actor applications with LLMs by modeling steps as edges and nodes in a graph.

## LangSmith

A developer platform that lets you debug, test, evaluate, and monitor LLM applications.

**LANCHAIN EXPRESSION LANGUAGE LCEL:**

LangChain Expression Language (LCEL) is a declarative way to chain LangChain components, designed to facilitate the transition from prototype to production without code changes. Key benefits of LCEL include:

- **First-class streaming support**: Enables the best possible time-to-first-token, streaming tokens directly from an LLM to an output parser for incremental output.
- **Async support:** Allows chains to be called synchronously (e.g., in a Jupyter notebook) or asynchronously (e.g., in a LangServe server), enabling the same code to be used for prototypes and production.
- **Optimized parallel execution:** Automatically executes parallel steps to minimize latency in both synchronous and asynchronous interfaces.
- **Retries and fallbacks**: Configurable for any part of the chain to enhance reliability at scale, with future support for streaming retries/fallbacks.

- **Access intermediate results:** Useful for end-users or debugging, streaming intermediate results even before final output.
- **Input and output schemas**: Provides Pydantic and JSONSchema schemas for input and output validation, integral to LangServe.
- **Seamless LangSmith tracing:** Automatically logs all steps to LangSmith for enhanced observability and debugging.
- LCEL promotes consistency and customization over legacy subclassed chains like LLMChain and ConversationalRetrievalChain, addressing the need for detailed prompt visibility and model customization as the variety of models increases.

**Common use cases:**

- Building chatbots
- Creating question-answering systems
- Developing custom LLM-based tools and applications

**Advantages:**

- Reduces boilerplate code
- Enhances modularity
- Facilitates easier debugging and testing

LCEL is particularly useful for developers working with LangChain who want to create more sophisticated and flexible LLM-powered applications with less code and greater clarity.

# LIGHT-RAG

**PyTorch** is a widely-used open-source machine learning library developed by Facebook's AI Research lab (FAIR). It is primarily used for:

- **Deep Learning**: PyTorch provides a flexible and easy-to-use interface for building and training deep neural networks. It supports various types of neural network layers, loss functions, and optimization algorithms.
- **Research and Development**: PyTorch is popular in the research community due to its dynamic computation graph, which allows for more flexibility and ease in experimenting with different neural network architectures and training procedures.
- **Computer Vision:** PyTorch has extensive support for computer vision tasks through its torchvision package, which includes pre-trained models, datasets, and image transformations.
- **Natural Language Processing (NLP):** PyTorch is used for various NLP tasks such as text classification, language modeling, and machine translation. The torchtext library provides tools for preprocessing text data and building NLP models.

- **Reinforcement Learning:** PyTorch supports reinforcement learning algorithms and is used to implement and train reinforcement learning models.
- **Scalability and Production:** PyTorch is designed to be scalable and can be deployed in production environments. The PyTorch ecosystem includes tools like TorchScript for exporting models and PyTorch Lightning for simplifying the training pipeline.
- **Interoperability with Other Libraries**: PyTorch integrates well with other scientific computing libraries in Python, such as NumPy, and can leverage GPU acceleration using CUDA

**Libraries with in PYTORCH:**

- **torchvision:** This library is used for computer vision tasks. It includes datasets, model architectures, and image transformations. It also provides pre-trained models for various vision tasks.
- **torchtext:** This library is designed for natural language processing (NLP) tasks. It provides tools for preprocessing text data, building vocabularies, and creating datasets. It also includes common datasets and pre-trained models for NLP.
- **torchaudio:** This library is used for audio processing tasks. It includes functionalities for loading, transforming, and augmenting audio data. It also provides common datasets and pre-trained models for audio tasks.
- **PyTorch Lightning**: Although not part of the core PyTorch library, PyTorch Lightning is a popular high-level framework built on top of PyTorch. It simplifies the process of training and deploying PyTorch models by abstracting away much of the boilerplate code.
- **TorchScript**: This is a way to create serializable and optimizable models from PyTorch code. It allows you to save your models in a format that can be loaded and run independently from the Python runtime, which is useful for deployment in production environments.
- **Ignite**: PyTorch Ignite is a high-level library to help with training neural networks in PyTorch. It provides a framework for writing compact but full-featured training loops in a few lines of code.

## Design Philosophy of LightRAG:

**Simplicity over Complexity:**

- Limit to three layers of abstraction.
- Minimize lines of code.
- Prioritize deep and broad understanding for clarity.

**Quality over Quantity:**

- Focus on high-quality core building blocks over numerous integrations.
- Ensure components like the prompt, model client, retriever, optimizer, and trainer are well-designed, easy to understand, and customizable.

**Optimizing over Building:**

- Emphasize optimization alongside task pipeline building.

- Provide excellent logging, observability, configurability, optimizers, and trainers for easier optimization.

**Understanding of LLM Workflow:**

- Developers are key to shaping LLMs for diverse applications (e.g., chatbots, translation, summarization).
- LLM applications require a blend of software engineering and in-context learning.
- Libraries like PyTorch offer essential building blocks and computational optimization.
- Manual prompt engineering is time-consuming and brittle, with significant accuracy variance.
- The future lies in auto-prompt optimization, though it's still in early development.

**Heavy Lifting in LLM Libraries:**

- Focus on core base classes and abstractions for serialization, interface standardization, and data processing.
- Provide building blocks for LLM-world interactions.
- Evaluate and optimize task pipelines while giving developers full control over prompts and pipelines.

## Light Rag:

- LightRAG shares similar design pattern as PyTorch for deep learning modeling. We provide developers with fundamental building blocks of 100% clarity and simplicity.
- Only two fundamental but powerful base classes: Component for the pipeline and DataClass for data interaction with LLMs.
- A highly readable codebase and less than two levels of class inheritance. Class Hierarchy.
- We maximize the library's tooling and prompting capabilities to minimize the reliance on LLM API features such as tools and JSON format.
- The result is a library with bare minimum abstraction, providing developers with maximum customizability.

The LightRAG framework, similar to PyTorch, provides a modular and composable structure for developers to build and optimize their LLM (Large Language Model) applications. The main components in LightRAG are analogous to PyTorch's modules and tensors.

**Modular**

LightRAG resembles PyTorch in the way that we provide a modular and composable structure for developers to build and to optimize their LLM applications.

- *Component* and *DataClass* are to LightRAG for LLM Applications what *module* and *Tensor* are to PyTorch for deep learning modeling.
- *ModelClient* to bridge the gap between the LLM API and the LightRAG pipeline.

- *Orchestrator* components like *Retriever*, *Embedder*, *Generator*, and *Agent* are all model-agnostic (you can use the component on different models from different providers).

Similar to the PyTorch *module*, our *Component* provides excellent visualization of the pipeline structure.

```
SimpleQA(
   (generator): Generator(
      model_kwargs={'model': 'llama3-8b-8192'},
      (prompt): Prompt(
         template: <SYS>
               You are a helpful assistant.
               </SYS>
               User: {{input_str}}
               You:
               , prompt_variables: ['input_str']
      )
      (model_client): GroqAPIClient()
   )
)
```

**Robust**

Our simplicity did not come from doing less. On the contrary, we have to do more and go deeper and wider on any topic to offer developers *maximum control and robustness*.

- LLMs are sensitive to the prompt. We allow developers full control over their prompts without relying on LLM API features such as tools and JSON format with components like *Prompt*, *OutputParser*, *FunctionTool*, and *ToolManager*.
- Our goal is not to optimize for integration, but to provide a robust abstraction with representative examples. See this in ModelClient and Retriever components.
- All integrations, such as different API SDKs, are formed as optional packages but all within the same library. You can easily switch to any models from different providers that we officially support.

**BM25** is a ranking function used in information retrieval systems to estimate the relevance of documents to a given search query. It operates on a term-based scoring model that evaluates documents based on the frequency of query terms, their distribution within the document, and overall document length, aiming to deliver accurate and relevant search results.

**Pipeline:**

Pipeline in terms of LLMs: A pipeline in the context of Large Language Models (LLMs) refers to a sequence of processing steps that data goes through from input to output. It's a way of chaining together different components or operations that each perform a specific task in the overall process of generating a response. Typically, an LLM pipeline might include steps like:

- Input preprocessing (e.g., tokenization, formatting)

- Prompt construction
- Model inference
- Output post-processing (e.g., decoding, formatting)

The pipeline allows for modular design, where each component can be developed, tested, and optimized independently, while still working together seamlessly.

**@dataclass Decorator**

- **Purpose**: The @dataclass decorator from the dataclasses module in Python is used to automatically generate special methods for classes, such as __init__, __repr__, and __eq__, based on class attributes.
- **Syntax**: Applied to a class definition to enable these automatic method generations.

**QAOutput Class**

```
@dataclass

class QAOutput(DataClass):

    explanation: str = field(

        metadata={"desc": "A brief explanation of the concept in one sentence."}

    )

    example: str = field(metadata={"desc": "An example of the concept in a sentence."})
```

**Class Definition**: QAOutput is a data class that represents a structured format for holding output related to a QA (Question-Answer) system.

**Fields:**

**explanation:** A string field with a brief explanation of a concept.

**example:** A string field with an example of the concept.

**field Function**: The field() function from the dataclasses is used to provide additional metadata or to customize the behavior of the data class fields. In this case, metadata is used to add descriptions for each field, which can be useful for documentation or schema generation.

This parameter (meta data) is used to attach arbitrary metadata to the field. In this case, it's a dictionary with a single key-value pair.

**Clear Pipeline Structure**

The pipeline design makes it easy to modify or replace individual components (like changing the model, adjusting the prompt, or modifying the output processing) without having to

rewrite the entire system. This type of clear, structured representation is indeed very helpful for users to understand the LLM workflow at a glance, seeing how data flows from input through various processing stages to the final output

Simply by using print(qa), you can see the pipeline structure, which helps users understand any LLM workflow quickly.

QA(

(generator): Generator(

model_kwargs={'model': 'llama3-8b-8192'},

(prompt): Prompt(

template: <SYS>

You are a helpful assistant.

<OUTPUT_FORMAT>

{{output_format_str}}

</OUTPUT_FORMAT>

</SYS>

User: {{input_str}}

You:, prompt_kwargs: {'output_format_str': 'Your output should be formatted as a standard JSON instance with the following schema:\n```\n{\n "explanation": "A brief explanation of the concept in one sentence. (str) (required)",\n "example": "An example of the concept in a sentence. (str) (required)"\n}\n```\n-Make sure to always enclose the JSON output in triple backticks (```). Please do not add anything other than valid JSON output!\n-Use double quotes for the keys and string values.\n-Follow the JSON formatting conventions.'}, prompt_variables: ['output_format_str', 'input_str']

)

(model_client): GroqAPIClient()

(output_processors): JsonOutputParser(

data_class=QAOutput, examples=None, exclude_fields=None, return_data_class=True

(json_output_format_prompt): Prompt(

template: Your output should be formatted as a standard JSON instance with the following schema:

```

{{schema}}

```

{% if example %}

Examples:

```

{{example}}

```

{% endif %}

-Make sure to always enclose the JSON output in triple backticks (```). Please do not add anything other than valid JSON output!

-Use double quotes for the keys and string values.

-Follow the JSON formatting conventions., prompt_variables: ['schema', 'example']

)

(output_processors): JsonParser()

)

)

)

# PROMPT IN LIGHT-RAG

**Serialization**

Serialization is the process of converting an object into a format that can be easily stored or transmitted.This format is usually a byte stream (such as binary) or a text format (such as JSON or XML). The purpose of serialization is to save the state of an object so that it can be recreated later.

**Example Use Cases:**

- Saving data to a file or database.

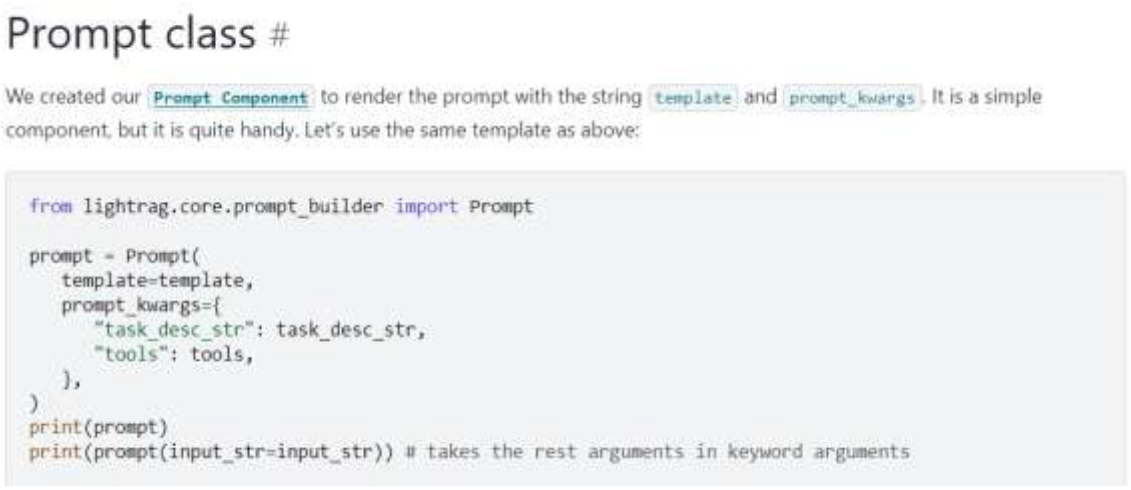
- Sending data over a network.
- Storing data in a cache.

**Deserialization**

Deserialization is the process of converting serialized data back into its original data structure or object. This allows the data to be used again in its original form.

**Example Use Cases:**

- Loading data from a file or database.
- Receiving data from a network.
- Retrieving data from a cache.



## Prompt class #

We created our `Prompt Component` to render the prompt with the string `template` and `prompt_kwargs`. It is a simple component, but it is quite handy. Let's use the same template as above:

```
from lightrag.core.prompt_builder import Prompt

prompt = Prompt(
   template=template,
   prompt_kwargs={
      "task_desc_str": task_desc_str,
      "tools": tools,
   },
)
print(prompt)
print(prompt(input_str=input_str)) # takes the rest arguments in keyword arguments
```

The Prompt class allow us to preset some of the prompt arguments at initialization, and then we can call the prompt with the rest of the arguments (like in this case input_str). Also, by subclassing Component, we can easily visualize this component with print.

Here is the output:

Prompt(

template: <SYS>{{ task_desc_str }}</SYS>

{# tools #}

{% if tools %}

<TOOLS>

{% for tool in tools %}

{{loop.index}}. {{ tool }}

{% endfor %}

</TOOLS>

{% endif %}

User: {{ input_str }}, prompt_kwargs: {'task_desc_str': 'You are a helpful assitant', 'tools': ['google', 'wikipedia', 'wikidata']}, prompt_variables: ['input_str', 'tools', 'task_desc_str']

As with all components, you can use to_dict (# Convert the object's state to a dictionary) and from_dict (# Convert a dictionary to object state ) to serialize and deserialize the component.

## Default Prompt Template

In default, the `Prompt` class uses the `DEFAULT_LIGHTRAG_SYSTEM_PROMPT` as its string template if no template is provided. This default template allows you to conditionally passing seven important variables designed from the data flow diagram above. These varaibles are:

```
LIGHTRAG_DEFAULT_PROMPT_ARGS = [
    "task_desc_str",  # task description
    "output_format_str",  # output format of the task
    "tools_str",  # tools used in the task
    "examples_str",  # examples of the task
    "chat_history_str",  # chat history of the user
    "context_str",  # context of the user query
    "steps_str",  # used in agent steps
    "input_str",  # user query or input
]
```

Now, let's see the minimum case where we only have the user query:

prompt = Prompt()

output = prompt(input_str=input_str)

print(output)

The output will be the bare minimum with only the user query and a prefix for assistant to respond:

<User>

What is the capital of France?

</User>

You:

There are higher-level components (such as the Generator and ModelClient) that handle the construction and usage of prompts. This abstraction reduces the need for users to interact directly with lower-level classes like Prompt.

# MODEL CLIENT IN LIGHT-RAG

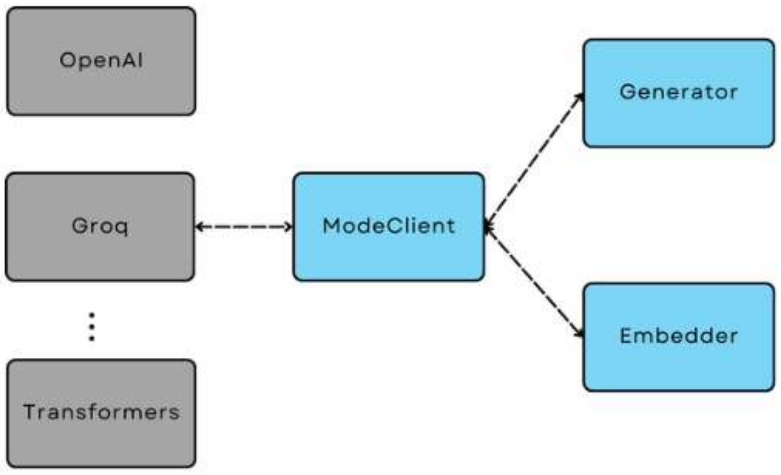
An SDK, or Software Development Kit, is a collection of tools, libraries, documentation, and code samples that developers use to create applications for specific platforms, frameworks, or operating systems. SDKs help streamline the development process by providing the necessary components to build, test, and debug software applications

**Key Components of an SDK:**

- Libraries and Frameworks
- APIs (Application Programming Interfaces)
- Development Tools
- Documentation
- Integrated Development Environment (IDE) Support

ModelClient is the standardized protocol and base class for all model inference SDKs (either via APIs or local) to communicate with LightRAG internal components. Therefore, by switching out the ModelClient in a Generator, Embedder, or Retriever (those components that take models), you can make these functional components model-agnostic.

Model agnostic components refer to tools, frameworks, or methods that can work with various machine learning models without being specifically tailored or dependent on any one model. These components are designed to be flexible and adaptable, allowing them to be used across different types of models, regardless of the underlying architecture or algorithm




The bridge between all model inference SDKs and internal components in LightRAG

**ModelClient Protocol#**

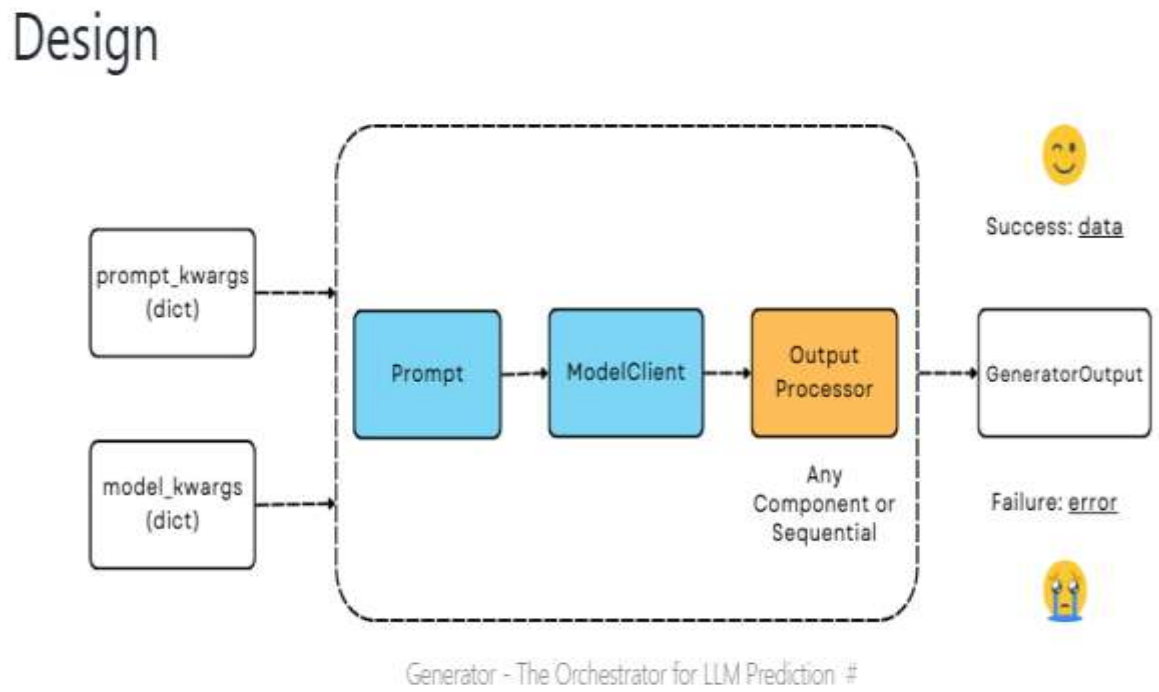
A model client can be used to manage different types of models, we defined a **ModelType** to categorize the model type.

```
class ModelType(Enum):
    EMBEDDER = auto()
    LLM = auto()
    RERANKER = auto()
    UNDEFINED = auto()
```

We designed 6 abstract methods in the *ModelClient* class that can be implemented by subclasses to integrate with different model inference SDKs. We will use **OpenAIClient** as the cloud API example and **TransformersClient** along with the local inference code **TransformerEmbedder** as an example for local model cli

# Generator in LIGHT-RAG

The Generator component in LightRAG simplifies the process of generating predictions with large language models by providing a high-level, user-facing interface. It orchestrates the workflow through a pipeline consisting of three customizable subcomponents: the prompt template, the model client, and the output parser. This design gives users full control and flexibility to tailor the prediction process to their specific needs, allowing them to switch out and configure each part of the pipeline as required.



**Design Breakdown**

**Inputs**

**prompt_kwargs (dict):**

This is a dictionary containing keyword arguments specific to the prompt generation. These arguments are used to customize the prompt template.

**model_kwargs (dict):**

This is a dictionary containing keyword arguments specific to the model client. These arguments are used to configure the model interaction, such as specifying model parameters or settings.

**Pipeline Components**

**Prompt:**

This component takes prompt_kwargs and uses them to generate a prompt based on a predefined template. The prompt is formatted to fit the requirements of the LLM.

**ModelClient:**

This component takes the generated prompt and communicates with the LLM to get a prediction. The ModelClient is responsible for sending the prompt to the model and retrieving the response. It is configured using model_kwargs.

**Output Processor:**

This component processes the raw output from the model. The processing can involve parsing, formatting, or any other transformation needed to convert the raw output into a usable form. The Output Processor can be a single component or a sequence of components, depending on the complexity of the required processing

**Outputs**

**GeneratorOutput:**

This is the final output of the Generator component. If the process is successful, it contains the processed data from the Output Processor.

**error:**

If there is a failure at any point in the pipeline, an error is generated. This error indicates what went wrong, allowing for debugging or handling within the application.

## Process Flow

**Step 1: Prompt Generation:**

prompt_kwargs are fed into the Prompt component, which generates a formatted prompt.

**Step 2: Model Interaction:**

The generated prompt is sent to the ModelClient, which uses model_kwargs to configure its interaction with the LLM. The ModelClient retrieves the raw response from the LLM.

**Step 3: Output Processing:**

The raw response from the LLM is processed by the Output Processor. This step ensures the response is in the desired format and ready for use.

**Step 4: Final Output:**

If the entire process is successful, the GeneratorOutput contains the processed data. If there is a failure, an error message is generated and returned.

**Flexibility and Control**

- **Customizable Prompt**: Users can change the prompt_kwargs to alter how prompts are generated, providing flexibility in how the input is structured.
- **Interchangeable ModelClient:** Users can switch out the ModelClient to use different models or configure the model interaction in various ways using model_kwargs.
- **Configurable Output Processing**: The Output Processor can be customized or sequenced to handle different types of output processing, ensuring the final output meets specific requirements.

The Generator is designed to achieve the following goals:

1. **Model Agnostic**: Can call any LLM model using the same prompt.
2. **Unified Interface**: Manages the pipeline from prompt input to model call to output parsing, giving users control over each part.
3. **Unified Output**: Simplifies logging and saving records of LLM predictions.
4. **Work with Optimizer**: Can integrate with Optimizer to refine prompts.

These goals also apply to other orchestrator components like Retriever, Embedder, and Agent

In LightRAG, orchestrator components like the **Generator**, **Retriever**, **Embedder**, and **Agent** are designed to manage specific workflows related to LLM predictions, retrievals, embeddings, and agent behaviors. They:

- **Coordinate** the flow of data through various stages (e.g., prompt generation, model interaction, output parsing).
- **Integrate** with different models and subcomponents, providing a seamless workflow.
- **Abstract** the complexity, allowing users to focus on high-level tasks without dealing with the intricacies of each step.

## An Orchestrator manages and coordinates three main components:

**Prompt:**

- Takes in a template (string) and prompt_kwargs (dict) to format the prompt at initialization.
- If no template is provided, it defaults to DEFAULT_LIGHTRAG_SYSTEM_PROMPT.

**ModelClient:**

- Uses an already instantiated model_client and model_kwargs to call the model.
- Switching the model client allows calling any LLM model with the same prompt and output parsing.

**Output Processors:**

- Processes the raw response to the desired format using a single component or chained components via Sequential.
- If no output processor is provided, the model client decides the output, typically returning a raw string response from the first response message.

## GeneratorOutput

Unlike other components, we cannot always enforce the LLM to follow the output format. The ModelClient and the output_processors may fail.

In particular, we created GeneratorOutput to capture important information.

- **data (object):** Stores the final processed response after all three components in the pipeline, indicating success.
- e**rror (str**): Contains the error message if any of the three components in the pipeline fail. When this is not None, it indicates failure.
- **raw_response (str):** Raw string response for reference of any LLM predictions. Currently, it is a string that comes from the first response message. [This might change and be different in the future]
- **metadata (dict):** Stores any additional information
- **usage:** Reserved for tracking the usage of the LLM prediction

# Parser in LIGHT-RAG

Parser is the *interpreter* of the LLM output

### Context

LLMs output text in string format. Parsing is the process of extracting and converting the string to desired data structure per the use case. This desired data structure can be:

- simple data types like string, int, float, boolean, etc.
- complex data types like list, dict, or data class instance.
- Code like Python, SQL, html, etc.

It honestly can be converted to any kind of formats that are required by the use case. It is an important step for the LLM applications to interact with the external world, such as:

- to int to support classification and float to support regression.
- to list to support multiple choice selection.
- to json/yaml which will be extracted to dict, and optional further to data class instance to support support cases like function calls.

## Parser

Our parser is located at core.string_parser. It handles both *extracting* and *parsing* to python object types. And it is designed to be robust.

| Parser Class | Target Python Object | Description |
|---|---|---|
| `BooleanParser` | `bool` | Extracts the first boolean value from the text with `bool`. Supports both 'True/False' and 'true/false'. |
| `IntParser` | `int` | Extracts the first integer value from the text with `int`. |
| `FloatParser` | `float` | Extracts the first float value from the text with `float`. |
| `ListParser` | `list` | Extracts '[]' and parses the first list string from the text. Uses both *json.loads* and *yaml.safe_load*. |
| `JsonParser` | `dict` | Extracts '[]' and '{}' and parses JSON strings from the text. It resorts to *yaml.safe_load* for robust parsing. |
| `YamlParser` | `dict` | Extracts `"yaml"`, `"yml"` or the whole string and parses YAML strings from the text. |

**Data Class Instance**

# Embedder in LIGHT-RAG

core.embedder.Embedder class is similar to Generator, it is a user-facing component that orchestrates embedding models via ModelClient and output_processors. Compared with using ModelClient directly, Embedder further simplify the interface and output a standard EmbedderOutput format.

By switching the ModelClient, you can use different embedding models in your task pipeline easily, or even embedd different data such as text, image, etc.

**EmbedderOutput**

core.types.EmbedderOutput is a standard output format of Embedder. It is a subclass of DataClass and it contains the following core fields:

- data: a list of embeddings, each embedding if of type core.types.Embedding.
- error: Error message if any error occurs during the model inference stage. Failure in the output processing stage will raise an exception instead of setting this field.
- raw_response: Used for failed model inference.

Additionally, we add three properties to the EmbedderOutput:

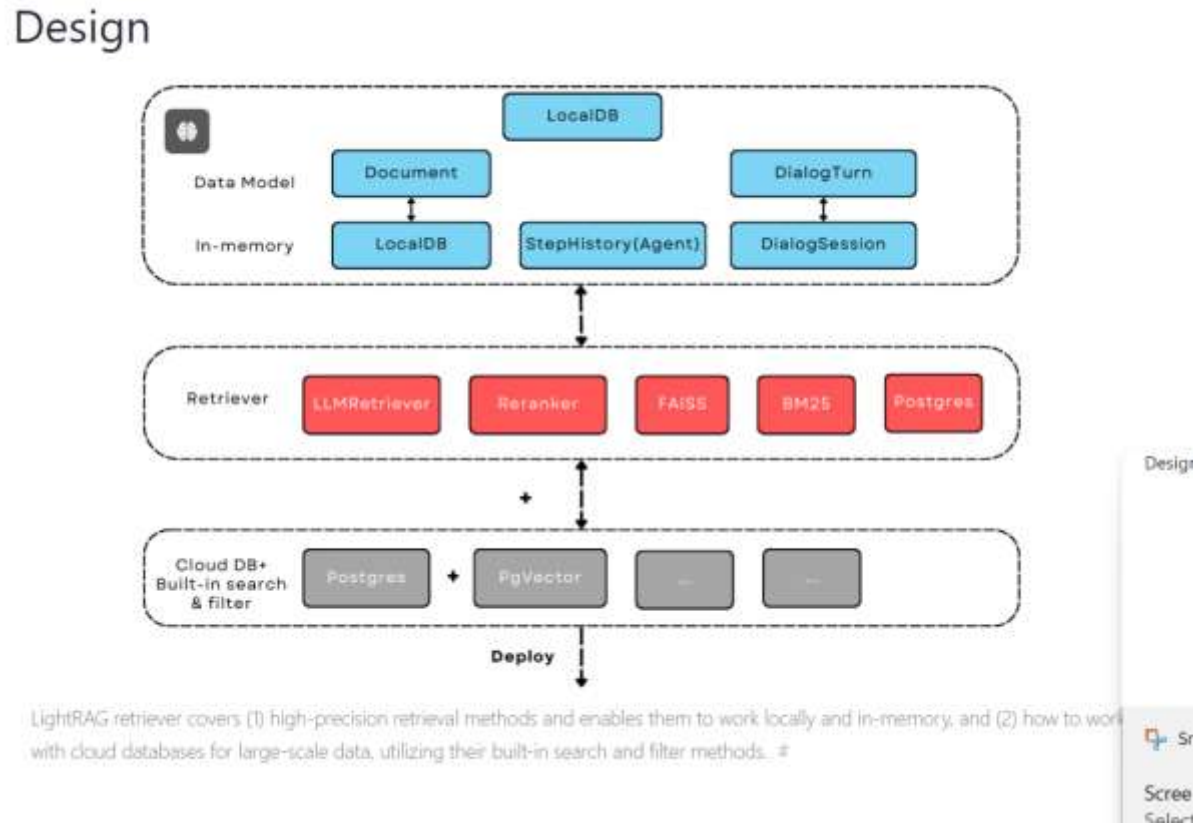
- length: The number of embeddings in the data.
- embedding_dim: The dimension of the embeddings in the data.
- is_normalized: Whether the embeddings are normalized to unit vector or not using numpy.

## BatchEmbedder

Especially in data processing pipelines, you can often have more than 1000 queries to embed. We need to chunk our queries into smaller batches to avoid memory overflow. core.embedder.BatchEmbedder is designed to handle this situation. For now, the code is rather simple, but in the future it can be extended to support multi-processing when you use LightRAG in production data pipeline.

The BatchEmbedder orchestrates the Embedder and handles the batching process. To use it, you need to pass the Embedder and the batch size to the constructor.

# Retriever in LIGHT-RAG




Here are current coverage on retriever methods:

1. LLM As Retriever
2. Reranker (Cross-encoder)
3. Semantic Search (Bi-encoder)
4. BM25
5. Database's built-in search such as full-text search/SQL-based search using Postgres and semantic search using teteVector.

PG-VECTOR

 **Purpose**: pgvector extends PostgreSQL to handle vector data types and operations, making it useful for similarity searches and machine learning applications.

 **Vector Data Type**: It introduces a vector data type for storing high-dimensional vectors, which are commonly used in machine learning for embeddings.

 **Similarity Search**: The extension supports similarity search operations, including approximate nearest neighbor (ANN) searches, which can be used to find vectors similar to a given vector.

 **Integration**: It integrates with PostgreSQL's SQL interface, allowing you to use standard SQL queries to work with vector data.

 **Indexing**: Supports indexing for vector data, which helps in efficient querying and searching within large datasets.

 **Use Cases**: Commonly used for applications like recommendation systems, search engines, and other scenarios where vector-based similarity is important

**Retriever Data Types**

**Query**

In most cases, the query is string. But there are cases where we might need both text and images as a query, such as “find me a cloth that looks like this”. We defined the query type *RetrieverQueriesType* so that all of our retrievers should handle both single query and multiple queries at once. For text-based retrievers, we defined *RetrieverStrQueriesType* as a string or a sequence of strings.

```
RetrieverQueryType = TypeVar("RetrieverQueryType", contravariant=True)
RetrieverStrQueryType = str
RetrieverQueriesType = Union[RetrieverQueryType, Sequence[RetrieverQueryType]]
RetrieverStrQueriesType = Union[str, Sequence[RetrieverStrQueryType]]
```

**Documents**

The documents are a sequence of documents of any type, which will be later specified by the subclass:

```
RetrieverDocumentType = TypeVar("RetrieverDocumentType", contravariant=True) # a single document
RetrieverDocumentsType = Sequence[RetrieverDocumentType] # The final documents types retriever can use
```

**Output**

We further definied the unified output data structure **RetrieverOutput** so that we can easily switch between different retrievers in our task pipeline. A retriever should return a list of *RetrieverOutput* to support multiple queries at once. This is helpful for:

1. Batch-processing: Especially for semantic search, where multiple queries can be represented as numpy array and computed all at once, providing faster speeds than processing each query one by one.
2. Query expansion: To increase recall, users often generate multiple queries from the original query.

# TEXT-SPLITTER IN LIGHT-RAG

The TextSplitter is a tool designed to manage and chunk large text data into smaller, more manageable pieces. This is important because large inputs can exceed the context window of Large Language Models (LLMs), leading to

```python
@dataclass
class RetrieverOutput(DataClass):
    doc_indices: List[int] = field(metadata={"desc": "List of document indices"})
    doc_scores: Optional[List[float]] = field(
        default=None, metadata={"desc": "List of document scores"}
    )
    query: Optional[RetrieverQueryType] = field(
        default=None, metadata={"desc": "The query used to retrieve the documents"}
    )
    documents: Optional[List[RetrieverDocumentType]] = field(
        default=None, metadata={"desc": "List of retrieved documents"}
    )


RetrieverOutputType = List[RetrieverOutput]  # so to support multiple queries at once
```

performance drops. By chunking the text, TextSplitter helps in improving embedding and retrieval processes.

**How It Works:**

- **Splitting:** The TextSplitter first uses a criterion defined by split_by to break down the long text into smaller pieces.
- **Sliding Window:** A sliding window approach is then used to create chunks. The window's length is defined by chunk_size, and it moves by a step size determined by chunk_size - chunk_overlap. The text within each window is merged into smaller chunks.
- **Chunking:** The result is a series of smaller text chunks that are returned.

**Splitting Types:**

- **Exact Splitting**: Uses specific text points like spaces or periods to split text (e.g., split_by = "word").
- `"Hello, world!" -> ["Hello, " ,"world!"]`

- **Tokenizer-Based Splitting**: Uses a tokenizer to split text into tokens, which aligns with how models process text.

  "Hello, world!" -> ["Hello", ",", " world", "!"]

**Key Definitions:**

- **split_by:** Defines the splitting rule (e.g., "word", "sentence", "token").
- **chunk_size:** Maximum number of units in each chunk.
- **chunk_overlap**: Number of units that each chunk should overlap to maintain context.
- **SEPARATORS**: Maps split_by criterions to their exact text separators, e.g., spaces <” “> for “word” or periods <”.”> for “sentence”.

| Split By | Chunk Size | Chunk Overlap | Resulting Chunks |
|---|---|---|---|
| word | 5 | 2 | "Hello, this is lightrag. Please", "lightrag. Please implement your splitter", "your splitter here." |
| sentence | 1 | 0 | "Hello, this is lightrag.", "Please implement your splitter here." |
| token | 5 | 2 | "Hello, this is l", "is lightrag.", "trag. Please implement your", "implement your splitter here." |

## CHUNKING TIPS

When choosing a chunking strategy, consider the following factors:

1. **Content Type**: Tailor your chunking approach to the type of content, such as articles, books, or social media posts.
2. **Embedding Model**: Align the chunking method with your embedding model's training. For example, use sentence-based splitting with sentence-transformer models, and token-based splitting with models like OpenAI's text-embedding-ada-002.
3. **Query Dynamics**: Adjust chunk size based on query length and complexity. Larger chunks work well for shorter, broader queries, while finer granularity is better for longer, specific queries.
4. **Application of Results**: The application, whether for semantic search, question answering, or summarization, should guide the chunking method, especially considering LLM content window limitations.
5. **System Integration**: Efficient chunking should match system capabilities, using larger chunks for broader context exploration (e.g., full-text search) and smaller chunks for precise information retrieval (e.g., granular search systems).

## CHUNKING STRATEGIES

 **Fixed-Size Chunking**:

- Best for content needing uniform chunks like genetic sequences or standardized data.
- Splits text into equal-sized word blocks, which is simple and efficient but may disrupt semantic coherence.

 **Content-Aware Chunking**:

- **Split by Sentence**: Maintains grammatical and contextual flow, ideal for academic or detailed texts.
- **Split by Passage**: Keeps the structure of large documents, aiding tasks like question answering or summarization.
- **Split by Page**: Useful for large documents where each page has distinct content, like legal texts.

 **Token-Based Splitting**:

- Suited for scenarios with strict token limits, ensuring compatibility with LLMs but possibly slowing processing.

 **Upcoming Feature - Semantic Splitting**:

- Groups text by meaning rather than structure, improving thematic search and contextual retrieval.

# FAISS RETRIEVER

- FAISSRetriever : Uses FAISS library to build an index from document embeddings for semantic search.
- Library Requirements: The faiss package must be installed separately to use FAISSRetriever.
- Embedding Preparation: Document content is converted to embeddings using the Embedder class with specified model parameters.
- FAISSRetriever Initialization: Initialized with embeddings and settings such as top_k and embedding dimensions.
- Index Building: Can build the index post-initialization using build_index_from_documents.
- Performing Retrieval: Retrieves documents using a single query or a list of queries, returning indices and cosine similarity scores.
- Score Interpretation: Scores range from 0 to 1, indicating relevance; default scoring is based on cosine similarity

# BM25 (Best Matching 25):

BM25 is a widely-used ranking function in information retrieval and text mining, designed to score the relevance of documents to a search query. It is a cornerstone algorithm in search engines and similar systems.

**How BM25 Works:**

**Document Scoring:**
BM25 calculates a score for each document based on the appearance of query terms within it.
**Key Factors:**

- Term Frequency (TF): Frequency of a query term in the document.
- Inverse Document Frequency (IDF): Importance of a term based on how many documents contain it.
- Document Length: Adjusts scores since longer documents tend to have more term occurrence

**Ranking:**
Documents are ranked by their BM25 scores in descending order. The top-ranked documents are considered the most relevant to the query.

**Advantages of BM25:**

- **Robustness**: Considers both term frequency and document length, making it more effective than simpler algorithms like TF-IDF.
- **Simplicity**: Easy to implement and widely adopted as a standard in information retrieval

Longer queries + only content given (wrong answer)

Now we call the retriever exactly the same way as we did with the FAISS retriever:

```
output_1 = bm25_retriever(input=query_1)
output_2 = bm25_retriever(input=query_2)
output_3 = bm25_retriever(input = [query_1, query_2])
print(output_1)
print(output_2)
print(output_3)
```

The printout is:

```
[RetrieverOutput(doc_indices=[2, 1], doc_scores=[2.151683837681807, 1.6294762236217233], query='what a
[RetrieverOutput(doc_indices=[3, 2], doc_scores=[1.5166601493236314, 0.7790170272483488], query='How d
[RetrieverOutput(doc_indices=[2, 1], doc_scores=[2.151683837681807, 1.6294762236217233], query='What a
```

Here we see the first query returns `[2, 1]` while the ground truth is `[0, 3]`. The second query returns `[3, 2]` while the ground truth is `[1, 2]`. The performance is quite disappointing. BM25 is known for lack of semantic understanding and does not consider context. We tested on the shorter and almost key-word like version of our queries and use both the *title* and *content*, and it gives the right response using the tokenized split.

Shorter queries + title and content given (correct answer)

```
query_1_short = "renewable energy?"  # gt is [0, 3]
query_2_short = "solar panels?"  # gt is [1, 2]
document_map_func = lambda x: x["title"] + " " + x["content"]
bm25_retriever.build_index_from_documents(documents=documents, document_map_func=document_map_func)
```

This time the retrieval gives us the right answer.

```
[RetrieverOutput(doc_indices=[0, 3], doc_scores=[0.9498793313012154, 0.8031794089550072], query='renew
[RetrieverOutput(doc_indices=[2, 1], doc_scores=[0.5343238380789569, 0.4568096570283078], query='solar
```

**Performance Testing:**

**Query Results:**

- The first query returns indices [2, 1] instead of the ground truth [0, 3].
- The second query returns [3, 2] instead of the ground truth [1, 2].

Conclusion: The performance is disappointing due to BM25's lack of semantic understanding.

**Improved Performance with Short Queries:**

- When using shorter, keyword-like queries and both the title and content, BM25 provides correct results.
- For example, with the query "renewable energy?", the retriever correctly returns [0, 3].

## ReRanker as Retriever

**Key Points:**

**Reranker Models:**

- Two rerankers are integrated:
- BAAI/bge-reranker-base hosted on Transformers.
- Rerankers provided by Cohere.

**ModelClient Protocol:**

- Rerankers follow the ModelClient protocol, which makes them accessible as retrievers from RerankerRetriever.
- The reranker model is treated as ModelType.RERANKER and requires four key arguments in model_kwargs: ['model', 'top_k', 'documents', 'query'].
- ModelClient converts LightRAG's standard arguments to match the specific needs of the model being used.

**Integration:**

- Users can integrate rerankers either locally or through APIs by referring to TransformersClient and CohereAPIClient.
- To use a reranker from RerankerRetriever, the model and necessary arguments (those not requiring conversion) are passed directly in model_kwargs.
- Example: The example given involves using the rerank-english-v3.0 model from Cohere, with the reminder to install the Cohere SDK and prepare an API key.

## LLM as Retriever

There are different ways to use LLM as a retriever:

- Directly show it of all documents and query and ask it to return the indices of the top_k as a list.
- Put the query and document a pair and ask it to do a yes and no. Additionally, we can use its logprobs of the yes token to get a probability-like score.

### Use Score Threshold instead of top_k

In some cases, when the retriever has a computed score and you might prefer to use the score instead of top_k to filter out the relevant documents. To do so, you can simplify set the top_k to the full size of the documents and use a post-processing step or a component(to chain with the retriever) to filter out the documents with the score below the threshold.

### Use together with Database

When the scale of data is large, we will use a database to store the computed embeddings and indexes from the documents.

# DATA & RAG

**1. Data Models and Data Pipeline**

Data Models: The note discusses the importance of having a standardized data model for representing text documents and conversational history. Core data structures like Document and DialogTurn are introduced, which serve as the backbone for processing text in LLM applications.

- **Document:** A flexible text container that supports operations like text splitting, embedding, and use as an LLM prompt.

  1. A general document/text container with fields `text`, `meta_data`, and `id`
  2. Assist text splitting with fields `parent_doc_id` and `order`.
  3. Assist embedding with fields `vector`.
  4. Assist using it as a prompt for LLM with fields `estimated_num_tokens`.

- **DialogTurn:** Represents a user-assistant conversation turn, useful for managing and processing conversational data. Arguments for DialogTurn are

```
id (str): The unique id of the turn.
user_id (str, optional): The unique id of the user.
session_id (str, optional): The unique id of the dialog session.
order (int, optional): The order of the turn in the dialog session, starts from 0.
user_query (UserQuery, optional): The user query in the turn.
assistant_response (AssistantResponse, optional): The assistant response in the turn.
user_query_timestamp (datetime, optional): The timestamp of the user query.
assistant_response_timestamp (datetime, optional): The timestamp of the assistant response.
metadata (Dict[str, Any], optional): Additional metadata.
```

- **Data Pipeline:** A pipeline is constructed using components like TextSplitter and ToEmbeddings. This pipeline is designed to process text data efficiently, converting raw input into structured Document objects, which are then transformed and embedded for further use.

**2. Local Database**

**LocalDB Class:** A versatile data management class that supports CRUD operations, data transformation, and state management. It is particularly useful for managing sequences of data items and applying processing pipelines to them.

**3. Working with Cloud Database**

**fsspec and SQLAlchemy**: The note mentions the importance of working with cloud databases in LLM applications, using libraries like fsspec for accessing remote file systems and SQLAlchemy for database management. This is crucial for handling large datasets and storing processed data in scalable environments.

**4. Data Handling in Real-World Applications**

**Data Storage**: The importance of managing large documents and conversational histories is highlighted, emphasizing the need for efficient data storage and retrieval mechanisms, especially in applications where state persistence is required, such as games and chatbots.

**5. Practical Examples**

**Examples Provided:** Practical examples are given, including how to create Document and DialogTurn objects, how to map original data to these structures, and how to apply data pipelines to process and transform the data.

# AGENTS IN LIGHT-RAG

**Function calls**

Tools are means LLM can use to interact with the world beyond of its internal knowledge. Technically speaking, retrievers are tools to help LLM to get more relevant context, and memory is a tool for LLM to carry out a conversation. Deciding when, which, and how to use a tool, and even to creating a tool is an agentic behavior: Function calls is a process of showing LLM a list of funciton definitions and prompt it to choose one or few of them. Many places use tools and function calls interchangably.

- **Thought**: The reasoning behind taking an action.
- **Action**: The action to take from a predefined set of actions. In particular, these are the tools/functional tools we have introduced in tools.
- **Observation**: The simplest scenario is the execution result of the action in string format. To be more robust, this can be defined in any way that provides the right amount of execution information for the LLM to plan the next step

**Users need to set up:**

- **tools**: a list of tools to use to complete the task. Each tool is a function or a function tool.
- **max_steps:** the maximum number of steps the agent can take to complete the task.
- **use_llm_as_fallback**: a boolean to decide whether to use an additional LLM model as a fallback tool to answer the query.
- **model_client**: the model client to use to generate the response.
- **model_kwargs:** the model kwargs to use to generate the response.
- **template:** the template to use to generate the prompt. Default is DEFAULT_REACT_AGENT_SYSTEM_PROMPT.

For the generator, the default arguments are:

- default prompt: DEFAULT_REACT_AGENT_SYSTEM_PROMPT
- default output_processors: JsonParser.

## Building Rag and Backend with LightRag:

**Tenant ID:**

- In multi-tenant systems, a tenant represents a group of users who share common access with specific privileges to the software instance.
- The tenant ID is a unique identifier for each tenant or client organization using the system.
- It's used to segregate and manage data, configurations, and access for different clients within the same software instance

**Using Tenant ID in RAG models:**

When creating a RAG model that handles multiple input files for different clients or organizations, you could use the tenant ID concept to organize and manage your data and models. Here's how you might approach this

a. Data Organization:

- Store documents/files in a structure that includes the tenant ID, e.g.:

Copy

```
/data
  /tenant_1
    - file1.txt
    - file2.pdf
  /tenant_2
    - fileA.docx
    - fileB.txt
```

b. Indexing:

- Create separate vector indexes for each tenant, or use a single index with tenant ID as a metadata field.

c. Model Creation:

- You could create separate RAG models for each tenant or a single model that filters based on tenant ID.

d. Query Processing:

- Include tenant ID in queries to ensure retrieval only from the relevant tenant's data.

**Multi Tenant, Multi Collection Model**

 Multi-tenant:

- Each tenant represents a distinct client or organization.
- Tenants are isolated from each other, ensuring data privacy and security.

 Multi-collection per tenant:

- Each tenant can have multiple collections of data.
- Collections could represent different datasets, document types, or functional areas within the tenant's domain.

**PROMPT OPTIMIZATION IN ADAL FLOW:**

# DsPy :

[Very informative article on DSPy optimizations]

**DSPy is a framework for algorithmically optimizing LM prompts and weights**, especially when LMs are used one or more times within a pipeline. To use LMs to build a complex system *without* DSPy, you generally have to: (1) break the problem down into steps, (2) prompt your LM well until each step works well in isolation, (3) tweak the steps to work well together, (4) generate synthetic examples to tune each step, and (5) use these examples to finetune smaller LMs to cut costs. Currently, this is hard and messy: every time you change your pipeline, your LM, or your data, all prompts (or finetuning steps) may need to change.

To make this more systematic and much more powerful, **DSPy** does two things. First, it separates the flow of your program (modules) from the parameters (LM prompts and weights) of each step. Second, **DSPy** introduces new optimizers, which are LM-driven algorithms that can tune the prompts and/or the weights of your LM calls, given a metric you want to maximize.

**DSPy** can routinely teach powerful models like GPT-3.5 or GPT-4 and local models like T5-base or Llama2-13b to be much more reliable at tasks, i.e. having higher quality and/or avoiding specific failure patterns. **DSPy** optimizers will "compile" the *same* program into *different* instructions, few-shot prompts, and/or weight updates (finetunes) for each LM. This is a new paradigm in which LMs and their prompts fade into the background as optimizable pieces of a larger system that can learn from data. **tldr;** less prompting, higher scores, and a more systematic approach to solving hard tasks with LMs.

**WHY USE DSPy:**

- Auto prompt optimization
- Auto reasoning
- Adapts to pipeline
- Auto weight optimization
- Build in Evaluation
- Signature: question and answer, document and summary, sentence and sentiment
- Modules: involves two things, prompting technique and large language model
- Optimizer: in dspy there is automatic system which can automatically evaluate generative response and retrieved context against the ground truth then modify the prompt and weights to get the more correct answer e.g optimizer like BootstrapFewShot()

**Modules:**

Modules are reusable building blocks that define specific operations or transformations in a pipeline. They can encapsulate different techniques like ChainOfThought (CoT), ReAct, or custom ones that you can define. Modules allow for clear separation between different stages of processing, making the pipeline more modular and easier to maintain.

**Signatures:**

Signatures define the input and output specifications for each module. They provide a structured way to describe what a module expects as input and what it will produce as output. This helps in building complex pipelines where different modules need to interact seamlessly.

Inspecting the dspy.Predict() class, we see when we pass to it our signature, the signature will be parsed as the signature attribute of the class, and subsequently assembled as a prompt. The instructions is a default one hardcoded in the DSPy library.

--What if we want to provide a more detailed description of our objective to the LLM, beyond the basic sentence -> sentiment signature? To do so we need to provide a more verbose signature in form of **Class-based DSPy Signatures**.

Notice we provide no explicit instruction as to how the LLM should obtain the sentiment. We are just describing the task at hand, and also the expected output.

```
# Define signature in Class-based form
class Emotion(dspy.Signature):
    # Describe the task
    """Classify emotions in a sentence."""

    sentence = dspy.InputField()
    # Adding description to the output field
    sentiment = dspy.OutputField(desc="Possible choices: sadness, joy, love, ang

classify_class_based = dspy.Predict(Emotion)

# Issue prediction
classify_class_based(sentence=sentence).sentiment
```

```
--- Output ---
Sentence: It's a charming and often affecting journey.
Sentiment: joy
```

**Optimizers:**

Optimizers are specialized components that automatically tune prompts and model weights to improve the performance of the language model. They iteratively refine the prompts or adjust the LM parameters based on feedback from previous executions to maximize a specific metric, like accuracy or relevance.

In this section we try to optimize our prompt for a RAG application with DSPy.

Taking Chain of Thought as an example, beyond just adding the "let's think step by step" phrase, we can boost its performance with a few tweaks:

1. Adding suitable examples (aka **few-shot learning**).
2. Furthermore, we can **bootstrap demonstrations of reasoning** to teach the LMs to apply proper reasoning to deal with the task at hand

   **Bootstrapped Context-Question-Reasoning-Answer demonstrations**

   - The technique uses a specific format called Context-Question-Reasoning-Answer (CQRA) for demonstrations.
   - It starts with a few manually provided examples (few-shot learning).
   - The language model makes predictions on new data, and correct predictions are identified.
   - These correct predictions are then converted into new CQRA demonstrations.
   - These new demonstrations are added to the prompt, expanding the set of examples the model can reference.
   - This process improves the model's capability by exposing it to more diverse, correct examples, helping it generalize better and enhance its reasoning skills.
   - The process can be iterative, potentially leading to continuous improvement

| Aspect | Chain of Thought (CoT) | ReAct (Reasoning and Acting) |
|---|---|---|
| **Process** | Focuses on reasoning step-by-step | Integrates reasoning with real-time actions |
| **Goal** | Transparent, multi-step reasoning | Dynamic decision-making with environment interaction |
| **Output** | Thought chain leading to a final answer | Interleaves reasoning with actions or commands |
| **Use Cases** | Question answering, logical problems | Interactive tasks (chatbots, information retrieval) |
| **Environment** | No interaction with the outside environment | Takes actions based on reasoning (like querying or updating states) |

The Modules under evaluation are:

- **Vanilla**: Single-hop RAG to answer a question based on the retrieved context, without key phrases like “let’s think step by step”
- **COT**: Single-hop RAG with Chain of Thought
- **ReAct**: Single-hop RAG with ReAct prompting
- **BasicMultiHop**: 2-hop RAG with Chain of Though

Results:

| module_name | optimizer | score |
|---|---|---|
| vanilla | none | 0.0 |
| vanilla | labeled_few_shot | 0.0 |
| vanilla | bootstrap_few_shot | 0.0 |
| cot | none | 0.0 |
| cot | labeled_few_shot | 35.0 |
| cot | bootstrap_few_shot | 50.0 |
| react | none | 25.0 |
| react | labeled_few_shot | 25.0 |
| react | bootstrap_few_shot | 35.0 |
| multihop | none | 0.0 |
| multihop | labeled_few_shot | 30.0 |
| multihop | bootstrap_few_shot | 35.0 |

**Multi hop did not performed well, so we improve the signatures**

So we revise the signature, and re-run the evaluation with the code below

```
# Define a class-based signature
class GenerateAnswer(dspy.Signature):
    """Answer questions with short factoid answers."""

    context = dspy.InputField(desc="may contain relevant facts")
    question = dspy.InputField()
    answer = dspy.OutputField(desc="often between 1 and 5 words")

class BasicQA(dspy.Signature):
    """Answer questions with short factoid answers."""

    question = dspy.InputField()
    answer = dspy.OutputField(desc="often between 1 and 5 words")

class FollowupQuery(dspy.Signature):
    """Generate a query which is conducive to answering the question"""

    context = dspy.InputField(desc="may contain relevant facts")
    question = dspy.InputField()
    search_query = dspy.OutputField(desc="Judge if the context is adequate to an
```

**After updating the results are:**

First, we had only GenerateAnswer signature later on we added BasicQA and FollowupQuery signature too, thus improving the result

**Impact on Accuracy after increasing the number of signatures**

- The approach with multiple signatures can potentially lead to more accurate results for several reasons:
- Separation of Concerns: Each signature focuses on a specific subtask, allowing for more specialized handling.
- Step-by-Step Processing: It enables a more structured, step-by-step approach to solving the overall task.
- Intermediate Validation: You can validate or refine intermediate outputs (like the search query) before proceeding to the final answer generation.
- Flexibility: It's easier to insert additional processing steps or modify individual components without affecting the entire pipeline.
- Better Context Management: The FollowupQuery signature allows for dynamic context refinement, potentially leading to more relevant information for answering the question.

| module_name | optimizer | score |
|---|---|---|
| vanilla | none | 25.0 |
| vanilla | labeled_few_shot | 40.0 |
| vanilla | bootstrap_few_shot | 40.0 |
| cot | none | 50.0 |
| cot | labeled_few_shot | 45.0 |
| cot | bootstrap_few_shot | 50.0 |
| react | none | 25.0 |
| react | labeled_few_shot | 25.0 |
| react | bootstrap_few_shot | 35.0 |
| multihop | none | 65.0 |
| multihop | labeled_few_shot | 65.0 |
| multihop | bootstrap_few_shot | 55.0 |

Performance improved after updating the signatures

This indicates despite DSPy tries to optimize the prompt, **there is still some prompt engineering involved by articulating your objective in signature**.

*Note: Even the signature itself can be optimized with DSPy's COPRO*

Since the best prompt is Multihop with LabeledFewShot, the prompt does not contain bootstrapped Context-Question-Reasoning-Answer demonstrations. So bootstrapping may not surely lead to better performance, **we need to prove which one is the best prompt scientifically.**

It does not mean Multihop with BootstrapFewShot has a worse performance **in general** however. Only that for our task, if we use GPT 3.5 Turbo to bootstrap demonstration (which might be of questionable quality) and output prediction, then we might better do without the bootstrapping, and keep only the few-shot examples.

This lead to the question: Is it possible to use a more powerful LM, say GPT 4 Turbo (aka teacher) to generate demonstrations, while keeping cheaper models like GPT 3.5 Turbo (aka student) for prediction?

**"Teacher" to power-up bootstrapping capability**

The answer is **YES** as the following cell demonstrates, we will use GPT 4 Turbo as teacher.

| module_name | optimizer | score |
|---|---|---|
| vanilla | bootstrap_few_shot | 45.0 |
| cot | bootstrap_few_shot | 50.0 |
| react | bootstrap_few_shot | 40.0 |
| multihop | bootstrap_few_shot | 55.0 |

Result using GPT-4 as teacher

**Prompt Optimization**

Then we are ready to see what this opimitzation is about! To "train" our prompt, we need 3 things:

1. A training set. We'll just use our 20 question–answer examples from trainset.
2. A metric for validation. Here we use the native dspy.evaluate.answer_exact_match which checks if the predicted answer exactly matches the right answer (questionable but suffice for demonstration). For real-life applications you can define your own evaluation criteria
3. A specific **Optimizer** (formerly teleprompter). The DSPy library includes a number of optimization strategies and you can check them out here. For our example we use BootstrapFewShot. Instead of describing it here with lengthy description, I will demonstrate it with code subsequently

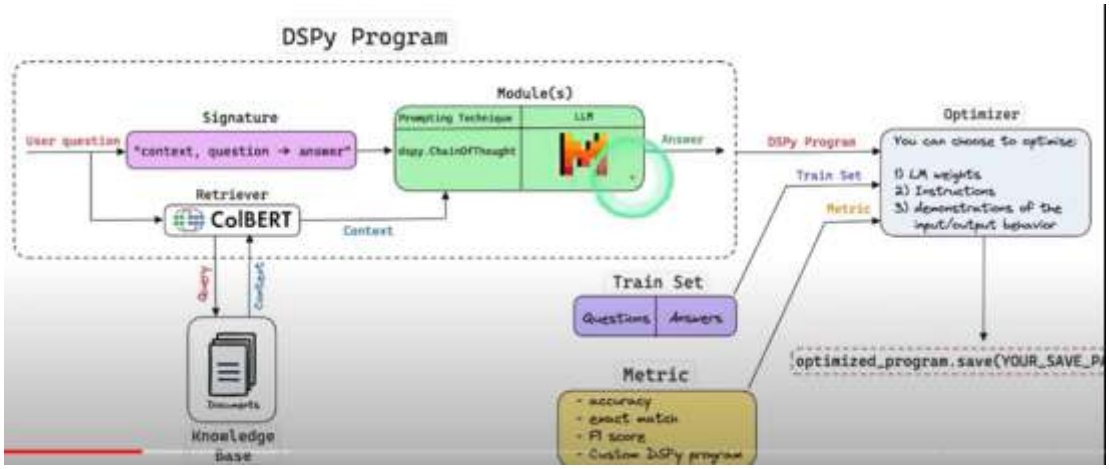
**LM Assertions:**

Assertions allow developers to enforce specific constraints on the LM's output. These constraints ensure that the output adheres to expected behaviors, improving reliability and correctness. For example, an assertion could enforce that a numerical output must fall within a certain range.

**Compilation:**

Compilation in DSPy refers to the process of transforming a DSPy program into executable instructions for the language model. This could involve generating few-shot prompts, updating model weights, or other optimizations. The compiled output is tailored to make the best use of the available LM resources.

Instead of simply using rag, it is a good method to divide the questions into parts (hops) and use optimizer to get more accurate answers




1. dspy.Predict: Basic predictor. Does not modify the signature. Handles the key forms of learning (i.e., storing the instructions and demonstrations and updates to the LM).
2. dspy.ChainOfThought: Teaches the LM to think step-by-step before committing to the signature's response.
3. dspy.ProgramOfThought: Teaches the LM to output code, whose execution results will dictate the response.
4. dspy.ReAct: An agent that can use tools to implement the given signature.
5. dspy.MultiChainComparison: Can compare multiple outputs from ChainOfThought to produce a final prediction.

6. dspy.majority: Can do basic voting to return the most popular response from a set of predictions.

**MULTI HOP QUESTION ANSWERING**

Multi-hop question answering is a type of question-answering task where the system needs to gather and reason over multiple pieces of information (often from different sources or parts of

a text) to answer a question. Unlike single-hop question answering, where the answer can be found directly from a single sentence or a small part of the text, multi-hop questions require the system to "hop" across different pieces of information to find the answer. Key Characteristics of Multi-hop Question Answering: Complex Reasoning: The system must connect multiple facts or pieces of information that may not be explicitly related at first glance. It involves understanding relationships between different parts of the data. Diverse Information Sources: The necessary information might be scattered across multiple paragraphs, documents, or even different data sources. The system needs to retrieve and integrate these pieces effectively. Chained Reasoning: The process often involves a sequence of reasoning steps where the answer to one sub-question leads to the next, eventually culminating in the final answer.

**Example of Multi-hop Question Answering: Question**: "Where was the author of the Declaration of Independence born?" To answer this question, the system needs to: Identify that the "author of the Declaration of Independence" refers to Thomas Jefferson. Find the birthplace of Thomas Jefferson. This requires two hops: Hop 1: Retrieve information about who wrote the Declaration of Independence (Thomas Jefferson). Hop 2: Retrieve information about where Thomas Jefferson was born (Shadwell, Virginia)

**TEXTGRAD**

TextGrad is a gradient-based method for optimizing text prompts. It was developed to improve the performance of LLMs on specific tasks by finding better prompts. Here are some key points about TextGrad:

- Purpose: It's used to automatically generate or refine prompts that lead to better performance on a given task.
- Methodology: TextGrad treats the discrete text prompt (prompt is discrete since each character is different) as a continuous (by converting the words in to their embeddings) object and uses gradient descent to optimize it.
- Application: It can be applied to various NLP tasks, such as text classification, question answering, and text generation.
- Advantages: TextGrad can often find more effective prompts than manual engineering, potentially leading to improved model performance.
- Limitations: Like other gradient-based methods, it can sometimes get stuck in local optima and may not always find the globally best prompt.

Here's how it works in practice:

- You start with an initial prompt for a specific task.
- TextGrad runs this prompt through the LLM and evaluates the output based on some performance metric.
- It then makes small adjustments to the prompt in the continuous space.
- These adjusted prompts are converted back to text and tested with the LLM.
- The process repeats, gradually improving the prompt's effectiveness.

TextGrad is part of a broader field of research into prompt optimization and automated prompt engineering for LLMs. It's one of several approaches being explored to enhance the capabilities of these models through better prompting techniques.

## Key Differences Illustrated in DsPy and TextGrad:

- **Manual vs. Automated:**
  - With **DsPy**, you're more involved in manually analyzing data and making decisions on how to refine the prompt based on statistical insights.
  - With **TextGrad**, the process is more automated, using gradient-based optimization to fine-tune the prompt in an iterative, data-driven manner.
- **General-Purpose vs. Specialized:**
  - **DsPy** is useful if you want to explore a wide range of prompts and responses, analyze them, and then manually decide on changes.
  - **TextGrad** is specialized for prompt optimization, directly working on refining the prompt using a gradient-based approach without needing as much manual analysis.
- **Data Requirements:**
  - **DsPy** might require you to collect a significant amount of data before you can make informed decisions.
  - **TextGrad** works more on-the-fly, adjusting the prompt based on real-time feedback from the model.

- **Scope:** TextGrad focuses on single prompts, while DsPy optimizes entire workflows.
- **Automation level:** TextGrad is more fully automated in its optimization process. DsPy offers a mix of automated and manual optimization opportunities.
- **Optimization method:** TextGrad uses gradient descent on continuous representations. DsPy uses a variety of techniques, including statistical analysis, but not gradient-based optimization of text.
- **User involvement:** TextGrad requires less direct user involvement in the optimization process. DsPy allows (and often requires) more user input in designing and refining the workflow.

**TEXT OPTIMIZER AND DEMO OPTIMIZER**

In the context of large language models (LLMs), a text optimizer generally refers to techniques that enhance the performance and efficiency of the model in generating or interpreting text, focusing on refining language output and model responses. Meanwhile, a demo optimizer often involves strategies used during demonstrations or showcases,

emphasizing how well the model can deliver tasks in real-time scenarios, possibly through prompt engineering or specific user interactions.

While both aim to improve model output, their focus and application contexts differ.

In this context, a text optimizer enhances the model's ability to generate coherent and contextually appropriate text. It focuses on refining the language output to ensure clarity, relevance, and fluency. Techniques may include adjusting prompts, fine-tuning the model on specific datasets, or using algorithms that prioritize certain linguistic features.

On the other hand, a demo optimizer is geared towards showcasing the model's capabilities in real-time interactions. This involves strategies that improve the model's performance during demonstrations, such as prompt engineering to elicit better responses, managing user interactions smoothly, and ensuring that the model can handle various scenarios effectively.

**WORKING IN DSPy:**

 **Initial Setup:**

- You define a task-specific module (like RAG in your case) using DSPy.
- You provide a training dataset and a test dataset.

 **Compilation Process:**

- The BootstrapFewShot optimizer is used to "compile" your module.
- Compilation in DSPy means optimizing the prompts and potentially the chain of operations in your module.

 **Training Phase:**

- The student model (your RAG module) is initially exposed to the training dataset.
- However, it doesn't "train" in the traditional machine learning sense. Instead, it uses the training examples to optimize its prompts and behavior.

 **Bootstrapping Process:**

- The optimizer then enters a bootstrapping phase, which is where new examples are generated.
- This happens on the training set, not the test set.

 **Example Generation:**

- The "teacher" (which could be the same model with different settings or a more advanced model) generates new examples based on the training data.
- These examples are evaluated using the specified metric (answer_exact_match in your case).
- If the generated examples meet certain quality criteria, they are added to a pool of "bootstrapped" examples.

 **Prompt Optimization:**

- The optimizer uses both the original training examples and the bootstrapped examples to refine the prompts used in your module.
- This process aims to improve the module's performance on the task.

 **Testing Phase:**

- After compilation (which includes bootstrapping and optimization), your compiled module is evaluated on the test set.
- The test set is not used for generating new examples or optimizing prompts; it's solely for evaluation.

 **Few-Shot Learning:**

- The compiled module may use a few examples (either from the original training set or bootstrapped examples) in its prompts when making predictions on new data.
- This is the "few-shot" aspect of the process.

**Prompt Optimization:** This refers to the process of improving or refining the input prompts given to an LLM to enhance its performance on specific tasks. Techniques like TextGrad fall under this category. The goal is to find the most effective way to instruct or guide the LLM to produce desired outputs.

**LLM Optimization**: This term typically refers to the process of improving the LLM itself, rather than just the prompts. LLM optimization involves techniques such as:

- Tweaking the structure of the neural network. For example, the number of layers, the size of the layers and the connections between them. This is done to improve learning efficiency and output quality.
- Implementing techniques to speed up the training process. This includes methods like mixed-precision training, which uses both 16-bit and 32-bit floating-point operations. This is done to balance computational load and accuracy.
- Optimizing the data used for training. For example, selecting more relevant datasets or using techniques like data augmentation. This is done to improve the model’s understanding and generation capabilities.
- Finding ways to reduce the workloads of training and inference processes. For example, using GPU-as-a-service. This helps minimize the costs and energy consumption associated with developing and maintaining large language models.
- Implementing strategies and guardrails to identify and mitigate biases in the model’s outputs. This ensures that the generated text is fair and does not perpetuate harmful stereotypes.
- Techniques like model pruning (removing less important connections) and knowledge distillation (training a smaller model to replicate the performance of a larger one). These are used to make models smaller and faster, enabling their deployment in resource-constrained environments.

- Techniques for fine-tuning and transfer learning, to adjusting a pre-trained model on a specific dataset or task. This allows customizing applications of LLMs without the need to train a model from scratch.

## What are the Applications of LLM Optimization?

LLM optimization refers to the process of improving LLM models so they are more efficient, accurate, and adaptable to specific tasks. Here are some use cases LLM optimization can be helpful in:

- **NLP** – Optimization enhances LLM performance in NLP tasks like sentiment analysis, language translation and content generation. This can benefit fields like customer service, language education, and content creation.
- **Search Engines** – Optimized LLMs can improve search engine capabilities. They can better help understand the intent behind queries more accurately and provide more relevant search results. This improves content creation and marketing results.
- **Personal Assistants** – Voice-activated personal assistants like Siri, Alexa and Google Assistant benefit from LLM optimization by improving their understanding of natural language. This enables them to better respond to complex queries, understand context over multiple interactions and even detect user emotions to tailor responses, to improve the user experience.
- **Code Generation and Assistance** – Optimized LLMs can better assist developers by generating code snippets, debugging and providing recommendations for best practices. This speeds up the development process and also helps in educating and improving the skills of developers.
- **Healthcare** – LLM optimization can enhance the ability to interpret clinical documentation, patient inquiries and research papers. This can assist healthcare professionals in diagnosis support, treatment suggestions and patient communication, which contributes to better patient outcomes.
- **Legal and Compliance** – Optimized LLMs can improve analysis of legal documents, contracts and regulations. This helps in drafting documents, identifying relevant laws and precedents and predicting legal outcomes, which can save time and reduce costs for legal professionals.
- **Education** – LLMs that are optimized can provide personalized learning experiences, generate educational content and offer tutoring in various subjects, potentially making education more accessible and effective.
- **E-commerce and Marketing** – LLM optimization can generate personalized product descriptions, recommendations and marketing content. This can improve customer engagement, enhance user experience and increase conversion rates.

## Challenges in LLM Optimization

While there are many advantages to LLM optimization, ease of the process is not one of them. Here are the challenges of the process.

- Obtaining and paying for GPUs and balancing model performance with resource efficiency.
- Ensuring data quality and representativeness of training data and reducing bias.

- Preventing overfitting and ensuring that LLMs generalize well to unseen data.
- Making LLMs interpretable to humans, to foster trust and ensure accountability.
- Protecting personal and sensitive information to maintain privacy.
- Developing safeguards against misinformation and toxicity.

## What are Some Types of LLM Optimization?

There are multiple optimization techniques that can be used for LLM optimization. Examples include:

- **LLM Inference Optimization** – Improving the efficiency and speed of generating predictions or responses from a trained LLM. This could involve techniques such as model pruning, quantization, or specialized hardware acceleration, which reduce inference time and resource consumption while maintaining accuracy.
- **LLM Prompt Optimization** – Prompt optimization involves crafting effective prompts or inputs to LLMs to receive desired outputs or responses. This could include experimenting with different prompt formats, lengths, or structures to achieve better performance or accuracy for specific tasks or domains.
- **LLM Cost Optimization** – Cost optimization is about minimizing the financial or computational resources required to train, deploy, or use LLMs effectively. This may include techniques like model distillation, transfer learning, or parameter tuning to achieve comparable performance with smaller or more efficient models.

## LLM Optimization and MLOps

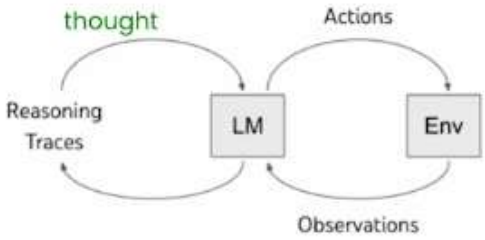
MLOps ensures models can be efficiently developed, deployed, monitored and maintained in production environments. Integrating LLM optimization within MLOps involves:

- **CI/CD** – Automating the integration of LLM updates and ensuring smooth deployment into production. This includes rigorous testing and validation to ensure model updates do not degrade performance.
- **Monitoring and Maintenance** – Once deployed, LLMs require continuous monitoring to ensure they perform as expected. This includes tracking performance metrics, identifying and correcting drifts in model predictions and updating the model as new data becomes available.
- **Scalability and Efficiency** – Ensuring that the deployed LLMs can handle the required scale and load efficiently. This involves optimizing model serving, using technologies like model serving frameworks (e.g., TensorFlow Serving, TorchServe) and edge computing for latency-sensitive applications.
- **Ethics and Compliance** – Ensuring that LLMs adhere to ethical guidelines and comply with relevant regulations. This includes transparency, fairness, privacy considerations, and the ability to explain model decisions.

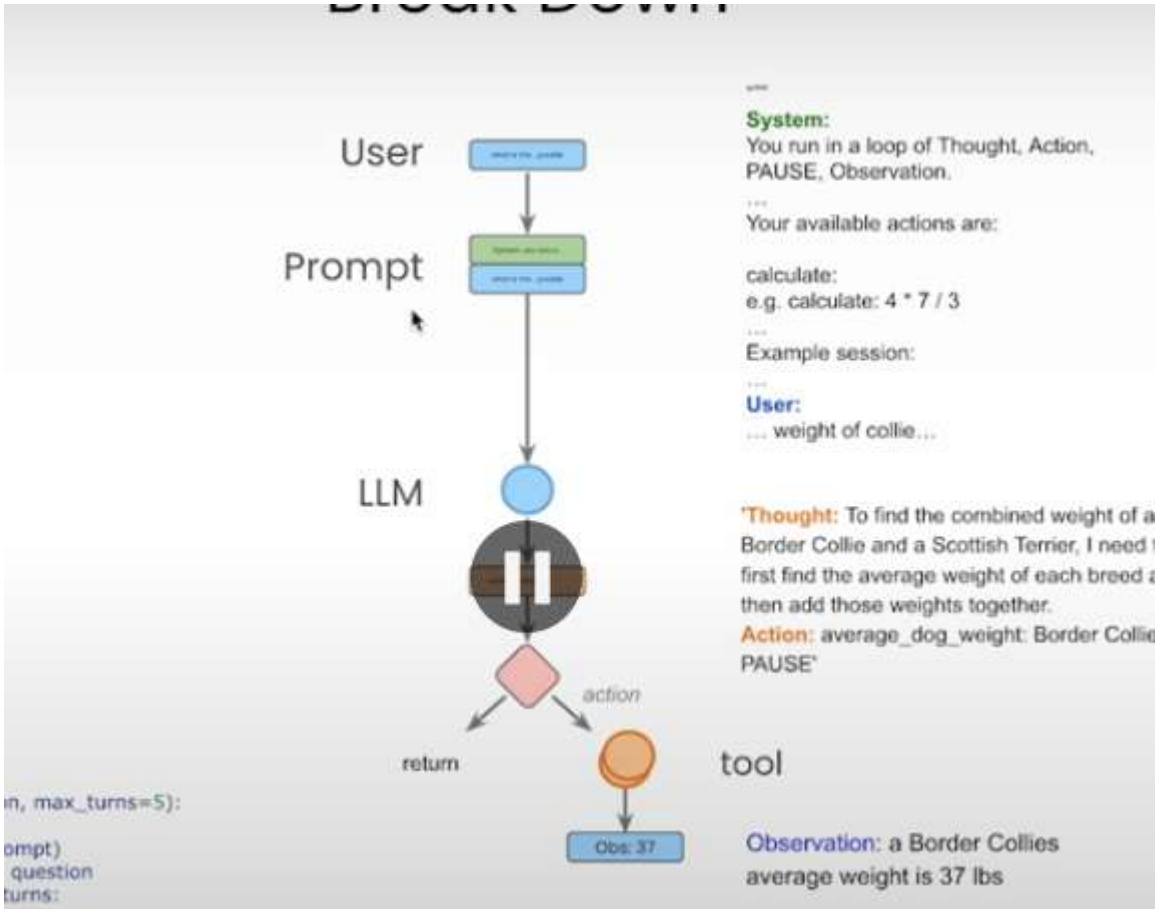
# LANGGRAPH

LangGraph is a library within the LangChain ecosystem that provides a framework for defining, coordinating, and executing multiple LLM agents (or chains) in a structured and efficient manner.

**React Agent**




ReAct (Reason + Act)



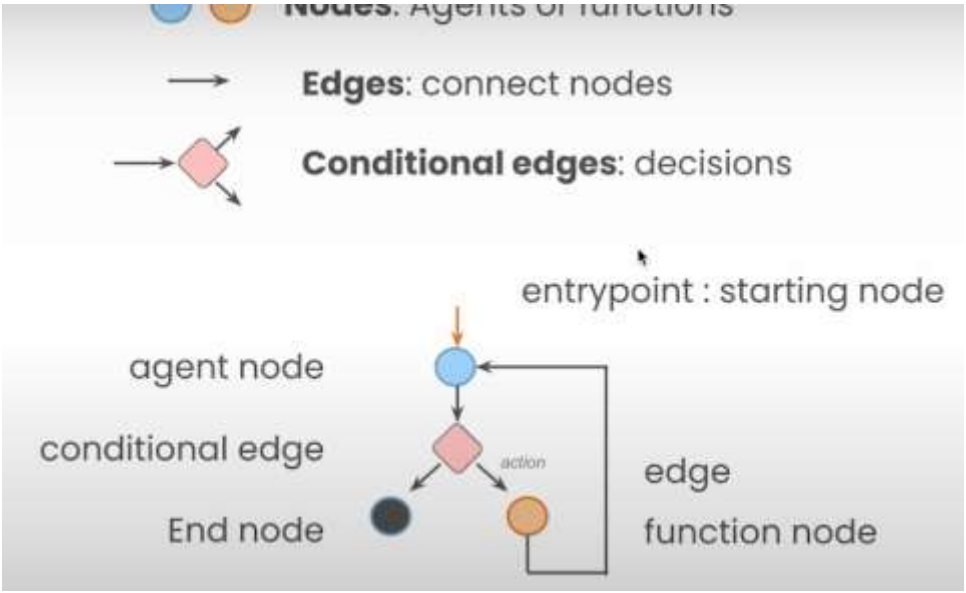
**Prompts in Langchain**
we have prompts template in langchain which allow reusable prompts.


**Tools in Langchain:**

Tavily tools etc

Lang graph allow us to created

- cyclic graphs
- persistence (for remembering previous convos)
- human in the loop feature

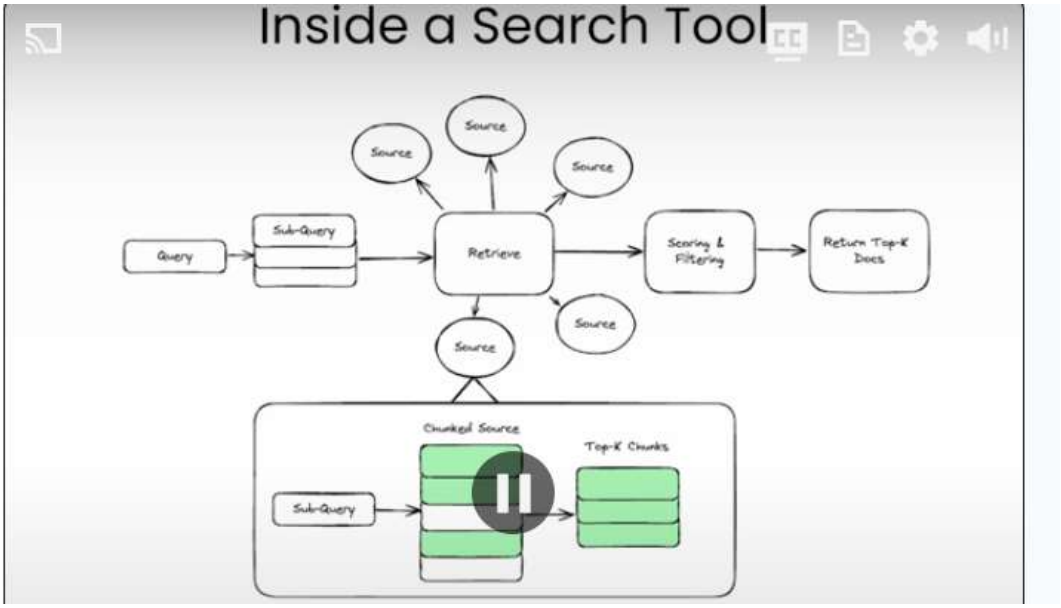



**Agent state:** accessible to all parts of graph, local to graph, can be stored in persistence layer

Types of state:

- Simple state (Annotated does not overwrite previous content)
- Complex state

Inside a search tool:




**Persistence**

Persistence helps you keep the state of agent at a particular point in time, which lets you go back in time and resume in that state in future interaction

For example, if we use persistence and ask ai first that “What is the weather in LA” then we say “What about in DC”, in the second query we did not ask about the weather but the ai will know that we are asking about the weather due to persistence we did using checkpoint and by using the same thread id for both queries. If we change the thread id, language model will get confused and will not able to answer the question

**Streaming:**

You can emit signals, to get to know what is happening in exact moment of time. We can stream events and also tokens

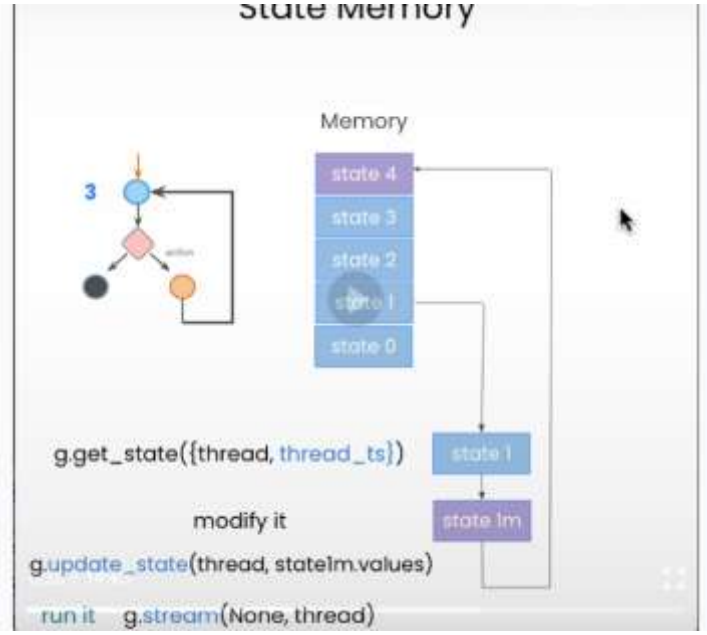
**Human in the Loop**

When compiling the graph add a parameter in that “interrupt_before” so it will stop wait for human approval, this is done to ensure that tools are being called correctly,

graph.compile(checkpointer=checkpointer,interrupt_before=["action"])

As graph is executing, a snap shot of state memory is being saved, in the state snapshot we have {AgentState, useful_things} (thread,thread_ts{unique identifier for the thread}).

We can use the unique identifier to access a specific state and start from there instead from the latest state. We can also access than modify the state and save the particular state in the place of current state using the identifier. **Then the newly modified state will be used as starting point. We did not over write the previous current state, instead a new state was created, and that became the current state**




**To run from the previous current state**

states= list of state we have

to_replay=states[-1]

graph.stream(None,to_replay.config)

We can also append messages to state. We can give the agent the response of a tool with out actually calling the tool in this way. We can also correct what has been done.

# RERANKERS

Rerankers work in a way that it takes a pair of sentences, suppose we retrieved 10 sentences from vector db, now we will re rank these on the basis of their similarity with the question.

Reranker does pair wise similarity for each of the 10 sentences with question one by one. It then sorts the sentences based on their similarity score

# Weaviate in Langchain

**Support persistence, multi tenancy**

from weaviate.classes.query import Filter

```
jeopardy = client.collections.get("JeopardyQuestion")
response = jeopardy.query.fetch_objects(
    filters=Filter.by_property("round").equal("Double Jeopardy!"),
    limit=3
)
```

OUTPUT


Can apply Nested filters, OR, AND, DATE, IS NULL, GREATER, LESS than, partial text matches filters, by object id in weaviate

- Use any_of or all_of for filtering by any, or all of a list of provided filters.
- Use & or | for filtering by pairs of provided filters.

## WEAVIATE SUPPORTS PRE FILTERING

### Post-Filtering vs Pre-Filtering

Systems that cannot make use of pre-filtering typically have to make use of post-filtering. This is an approach where a vector search is performed first and then some results are removed which do not match the filter. This leads to two major disadvantages:

1. You cannot easily predict how many elements will be contained in the search, as the filter is applied to an already reduced list of candidates.
2. If the filter is very restrictive, i.e. it matches only a small percentage of data points relative to the size of the data set, there is a chance that the original vector search does not contain any match at all.

The limitations of post-filtering are overcome by pre-filtering. Pre-Filtering describes an approach where eligible candidates are determined before a vector search is started. The vector search then only considers candidates that are present on the "allow" list

## BM 25 SEARCH IN WEAVIATE

The bm25 operator performs a keyword (sparse vector) search, and uses the BM25F ranking function to score the results. BM25F (**B**est **M**atch **25** with Extension to Multiple Weighted **F**ields) is an extended version of BM25 that applies the scoring algorithm to multiple fields (properties), producing better results.

How does BM25 work?

The BM25 retrieval function calculates a relevance score for each document based on a specific search query.

The algorithm looks at three things:

1. How often do the query terms appear in the document.
2. The length of the document.
3. The average length of all documents in the collection.

**k1 controls the strength of the term frequency's contribution** to the relevance score. If k1 is set to a high value, the term frequency will have a stronger impact on the score. If k1 is close to 0, term frequency will have little effect. k1 is usually set to a value between **1.2 and 2.0** (by default around 1.2).

$$\text{TF component} = \frac{\text{tf}}{\text{tf} + k_1}$$

 **b** controls the degree of **document length normalization**.

 Documents can vary in length, and typically, longer documents tend to contain more terms. The parameter b adjusts for this bias by normalizing term frequency based on the length of the document relative to the average document length.

b typically ranges from **0 to 1**.

- **b=0** means no length normalization (i.e., document length does not affect the score).
- **b=1** means full normalization based on document length, so longer documents are penalized proportionally to how much longer they are compared to the average.

The formula for **document length normalization** is:

$$\text{Normalized length} = 1 - b + b \cdot \frac{\text{dl}}{\text{avg\_dl}}$$

Where:

- dl is the length of the document (number of terms).
- avg_dl is the average document length in the collection.
- $b$ determines how much the length of the document affects the score.

**BM25F** is an extension of the BM25 ranking model designed to handle structured documents by allowing fields to be weighted differently. This model is particularly useful when documents contain multiple fields, such as titles, abstracts, and body text, helping to improve the relevance of search results by considering these variations in importance.

- TF-IDF: Typically doesn't account for document length
- BM25: Normalizes scores based on document length, reducing bias towards longer documents

4. Formula: The BM25F score for a term t in document D is:

Copy

```
score(D,t) = IDF(t) * ((k1 + 1) * sum(c(t,f) * w(f)) / (k1 + sum(c(t,f) * w(f))))
```

Where:

- c(t,f) is the term frequency of t in field f
- w(f) is the weight (boost) for field f
- k1 is a tuning parameter

BM25F addresses a common real-world scenario where not all parts of a document are equally important for relevance scoring. By allowing different weights for different fields, it can provide more nuanced and accurate search results in many practical applications.

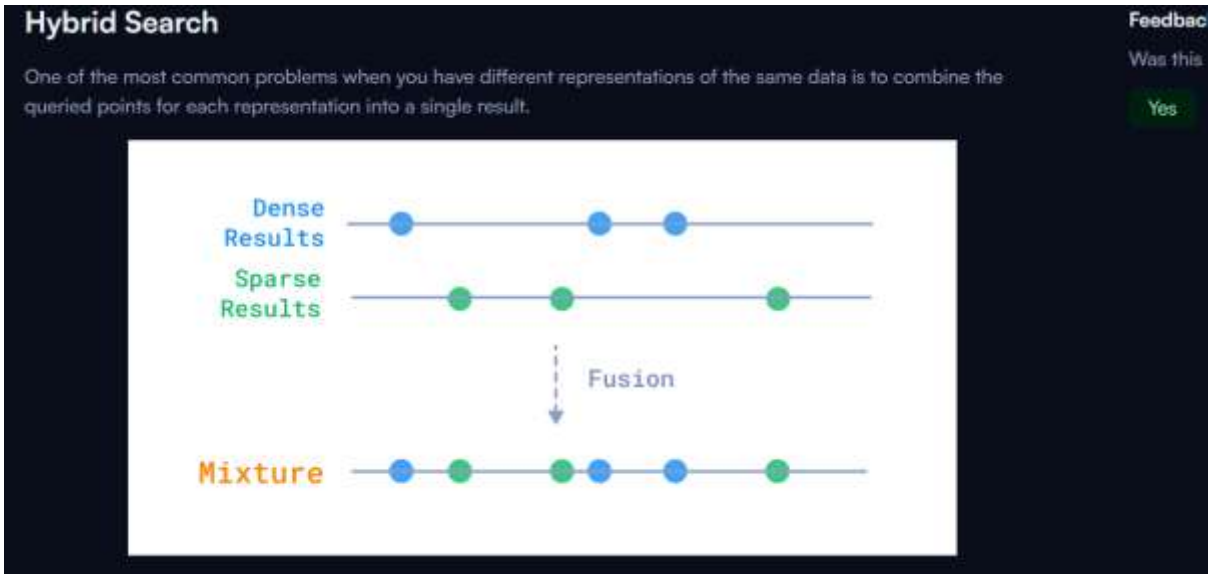

**Applications of Hybrid Search:**

- **Search engines** (like **Google**): Use a combination of keyword matching and semantic understanding to return relevant results.
- **E-commerce**: Helps users find products based on both keywords and descriptions or features they imply but didn't explicitly search for.
- **Document retrieval systems**: In corporate or legal settings where documents can be found based on both exact keyword matches and their overall context or meaning.
- **Recommendation systems**: Hybrid search can be used to recommend similar products or content by understanding the meaning of the current query and combining it with keyword matches.

**Hybrid Search in Weaviate**

This operator allows you to combine BM25 and vector search to get a "best of both worlds" type search results set.

**If you have a=0 and you mistype a word which is present in your documents because it will not match anything**




Dense vector for semantic search in hybrid search and sparse vector for lexical (keyword search) bm-25 in hybrid searchpg

## Sparse and Dense Vectors:

| Feature | Sparse Vectors | Dense Vectors |
|---|---|---|
| Data Representation | Majority of elements are zero | All elements are non-zero |
| Computational Efficiency | Generally higher, especially in operations involving zero elements | Lower, as operations are performed on all elements |
| Information Density | Less dense, focuses on key features | Highly dense, capturing nuanced relationships |
| Example Applications | Text search, Hybrid search | RAG, many general machine learning tasks |

Consider a simplified example of 2 documents, each with 200 words. A dense vector would have several hundred non-zero values, whereas a sparse vector could have, much fewer, say only 20 non-zero values.

In this example: We assume it selects only 2 words or tokens from each document. The rest of the values are zero. This is why it's called a sparse vector.

dense = [0.2, 0.3, 0.5, 0.7, ...]  # several hundred floats

sparse = [{331: 0.5}, {14136: 0.7}]  # 20 key value pairs

The numbers 331 and 14136 map to specific tokens in the vocabulary e.g. ['chocolate', 'icecream']. The rest of the values are zero. This is why it's called a sparse vector.

**Quantify result similarity**

You can optionally retrieve a relevance "score". This is a relative score that indicates how good the particular search results is, amongst the pool of search results.

Note that this is relative score, meaning that it should not be used to determine thresholds for relevance. However, it can be used to compare the relevance of different search results within the entire search result set.

# Product Quantization

Product quantization is a multi-step quantization technique that is available for use with hnsw indexes in Weaivate.

PQ reduces the size of each vector embedding in two steps. First, it reduces the number of vector dimensions to a smaller number of "segments", and then each segment is quantized to a smaller number of bits from the original number of bits (typically a 32-bit float).

PQ makes tradeoffs between recall, performance, and memory usage. This means a PQ configuration that reduces memory may also reduce recall. There are similar trade-offs when you use HNSW without PQ. If you use PQ compression, you should also tune HNSW so that they compliment each other.

## What is HNSW

HNSW (Hierarchical Navigable Small World) indexes are a type of data structure used for efficient approximate nearest neighbor (ANN) search, particularly in high-dimensional spaces. HNSW is based on the **Small-World Networks** theory and is designed to provide fast, scalable, and memory-efficient searches in large datasets. The algorithm is widely used in machine learning applications, especially in recommendation systems, text embedding retrieval, and image similarity search.

**Key Features of HNSW:**

1. **Hierarchical Structure**:
    - HNSW builds a multi-layered graph. The top layer contains only a few nodes, and as you move down, more nodes are added, resulting in denser layers. This hierarchical structure makes it easier to navigate between nodes to find nearest neighbors efficiently.
2. **Small-World Graph**:
    - The index builds a **small-world graph** where each node (data point) is connected to a limited number of nearby nodes, creating a network where any two nodes are reachable via a relatively small number of hops. This small-world property allows fast traversal through the graph when searching for neighbors.
3. **Approximate Nearest Neighbor Search**:
    - Instead of exhaustively comparing a query point with every other point in the dataset (which is computationally expensive in high-dimensional spaces), HNSW performs an approximate search. The algorithm navigates the graph structure, finding a subset of nearby nodes that are likely to be close to the query point.
4. **Efficient Search**:
    - HNSW can search large datasets with logarithmic or sub-linear time complexity with respect to the number of points in the dataset. This makes it highly scalable, especially in high-dimensional spaces where traditional algorithms like KD-trees or ball trees struggle.
5. **Trade-off Between Accuracy and Speed**:
    - HNSW allows tuning of parameters to control the trade-off between the accuracy of the nearest neighbor results and the search speed. You can get highly accurate results with slightly slower search times, or very fast results with some loss in accuracy.
6. **Insertion and Deletion**:
    - HNSW supports dynamic insertion and deletion of elements, making it flexible for use cases where the dataset changes over time.

**Combining HNSW with Product Quantization** allows you to:

- **Store compact representations of vectors using PQ**: Instead of storing the full-dimensional vectors in the HNSW graph, you can store the compressed, quantized versions of the vectors. This reduces the memory footprint significantly.
- **Search efficiently using HNSW**: The HNSW graph is used to efficiently traverse the search space and find approximate nearest neighbors. Once the nearest neighbors are found in the graph, PQ helps to further refine the distance calculation by working with the compact codes instead of the full vectors.

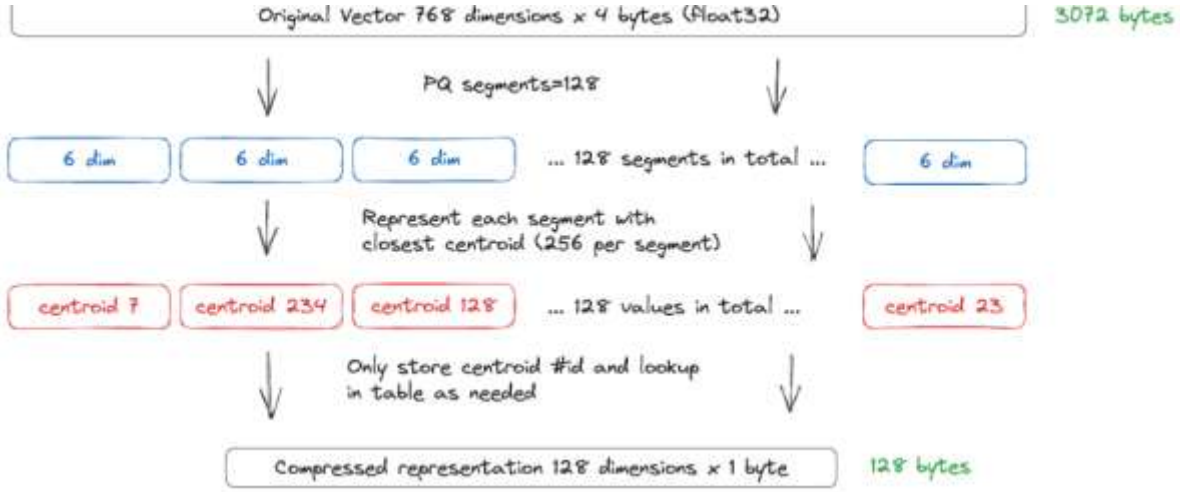
 When a query vector is provided, first search the HNSW graph to find approximate nearest neighbors. The HNSW algorithm will guide you towards the most likely candidates based on the structure of the graph.

 Instead of comparing the full vectors during the search, you compare their quantized (compressed) versions using **PQ codes**.

 The nearest neighbors found in the graph are further refined by **decompressing the PQ codes** (if needed) and performing a more accurate distance computation.

## Back to PQ

In PQ, the original vector embedding is represented as a product of smaller vectors that are called 'segments' or 'subspaces.' Then, each segment is quantized independently to create a compressed vector representation.




After the segments are created, there is a training step to calculate centroids for each segment. By default, Weaviate clusters each segment into 256 centroids. Before PQ compression, each vector embeddingrequires 768 x 4 = 3072 bytes of storage. After PQ compression, each vector requires 128 x 1 = 128 bytes of storage. The original representation is **almost 24 times as large** as the PQ compressed version. (It is not exactly 24x because there is a small amount of overhead for the codebook.)

**shard** refers to a smaller segment or portion of data or a model that has been split for easier management or processing.

| Module | Model | Dimensions | Segments |
|---|---|---|---|
| openai | text-embedding-ada-002 | 1536 | 512, 384, 256, 192, 96 |
| cohere | multilingual-22-12 | 768 | 384, 256, 192, 96 |
| huggingface | sentence-transformers/all-MiniLM-L12-v2 | 384 | 192, 128, 96 |

 Consider a **768-dimensional vector**, where each dimension is a **4-byte floating point number**. The total size of the vector would be:
768 dimensions×4 bytes per dimension=3072 bytes

 By dividing the vector into **128 segments** of 6 dimensions each, and then quantizing each segment to one of **256 centroids** (which takes just **1 byte**), the compressed vector size would be: 128 segments×1 byte per segment=128 bytes

 This results in a compression ratio of **24x**, as the original vector (3072 bytes) is reduced to just 128 bytes.

**4x compression** using **Product Quantization (PQ)** for a vector with 384 dimensions, the vector is divided into 4 sub-vectors, and **each sub-vector has 96 dimensions**.

Here's a quick summary of how it works:

- **Original Vector**: 384 dimensions.
- **Compression Factor**: 4x.
- **Sub-vector Creation**: The original 384-dimensional vector is divided into 4 smaller sub-vectors.
    - Each sub-vector has 384/4=96 dimensions.

So, after dividing, instead of having one 384-dimensional vector, you now have **4 sub-vectors**, each with **96 dimensions**.

 **More segments (higher M)**

- Pros: More fine-grained representation
- Cons: Increased storage, reduced compression rate

 **Fewer segments (lower M)**

- Pros: Higher compression rate
- Cons: Potentially lower accuracy due to more aggressive quantization

Compression Rate = Original Storage / Compressed Storage = $(D \times 32) / (M \times \log_2(K))$

Let's consider a 256-dimensional vector:

- Original vector: D = 256 dimensions
- Divide into 8 segments: M = 8
- Use 256 centroids per segment: K = 256 (8 bits)

Compression Rate = $(256 \times 32) / (8 \times 8) = 128$

This means the data is compressed to 1/128 of its original size.

**Understanding Centroids**

Centroids in product quantization are representative points in the vector space for each segment. They serve as "reference points" for quantization.

**Key Concepts**

1. **Centroid Selection**
    - For each segment, K centroids are chosen
    - These centroids are typically determined using k-means clustering
    - Each original segment is mapped to its nearest centroid
2. **Number of Centroids (K)**
    - Usually a power of 2 (e.g., 256 = 2^8)
    - Determines how many bits are needed to encode each segment
    - Common values: 256 (8 bits), 16 (4 bits), 64 (6 bits)
3. **Codebook**
    - Collection of all centroids for a segment
    - Each segment has its own codebook
    - Size of codebook = K

 **Larger K**

- More centroids per segment
- More bits needed per segment
- Higher accuracy but lower compression
- Example: K=256 (8 bits per segment)

 **Smaller K**

- Fewer centroids per segment
- Fewer bits needed per segment
- Lower accuracy but higher compression
- Example: K=16 (4 bits per segment)

## Compression Ratio Explained

 When we say "compressed to 1/128 of original size", this indeed means 128x compression

 Example calculation:

Original: 256 dimensions × 32 bits = 8192 bits

Compressed: 8 segments × 8 bits = 64 bits

8192/64 = 128

**It means that each vector was divided in to 8 sub vectors segments , each sub vector of 32 bits( 4 bytes) then we quantized each segment to 8 bits further**

**Example**

If we have vectors with 384 dimension and we say we are compressing the size by 4 times than

Total bytes= 384x4=3072 bytes

3072/x=4, x=768 bytes compressed size

768/8= 96 sub vectors for one original vector with 4 dimension each

Using 8 assuming 256 centroids are used

# Binary quantization

**Binary quantization (BQ)** is a quantization technique that converts each vector embedding to a binary representation. The binary representation is much smaller than the original vector embedding. Usually each vector dimension requires 32 bits, but the binary representation only requires 1 bit, representing a **32x reduction in storage requirements**. This works to speed up vector search by reducing the amount of data that needs to be read from disk, and simplifying the distance calculation.

The tradeoff is that BQ is lossy. The binary representation by nature omits a significant amount of information, and as a result the distance calculation is not as accurate as the original vector embedding.

Some vectorizers work better with BQ than others. Anecdotally, we have seen encouraging recall with Cohere's V3 models (e.g. embed-multilingual-v3.0 or embed-english-v3.0), and OpenAI's ada-002 model with BQ enabled. We advise you to test BQ with your own data and preferred vectorizer to determine if it is suitable for your use case.

Note that when BQ is enabled, a vector cache can be used to improve query performance. The vector cache is used to speed up queries by reducing the number of disk reads for the quantized vector embeddings. Note that it must be balanced with memory usage considerations, with each vector taking up n_dimensions bits.

# Scalar quantization

**Scalar quantization (SQ)** The dimensions in a vector embedding are usually represented as 32 bit floats. SQ transforms the float representation to an 8 bit integer. This is a 4x reduction in size.

SQ compression, like BQ, is a lossy compression technique. However, SQ has a much greater range. The SQ algorithm analyzes your data and distributes the dimension values into 256 buckets (8 bits).

SQ compressed vectors are more accurate than BQ compressed vectors. They are also significantly smaller than uncompressed vectors.

The bucket boundaries are derived by determining the minimum and maximum values in a training set, and uniformly distributing the values between the minimum and maximum into 256 buckets. The 8 bit integer is then used to represent the bucket number.

# ENCRYPTION OF VECTORS

Application-layer Encryption (ALE) is an architectural approach where you encrypt data before sending it to a data store. If the data store is compromised on a running machine, then the encrypted data remains safe.


In the case of a database or an index file, one option could be to encrypt the file storage at an infrastructure level, but this would not protect the data on a running server.

With ALE, even if someone gains access to the stored data on a running server or gains access to database credentials, the data is senseless to them unless they also have the key.

## Property-preserving encryption

If the data was randomly encrypted, it would be well protected, but you'd have to decrypt the data before doing anything with it. For example, to do a nearest neighbor search would require downloading all stored vectors, decrypting, and then executing the search.

Using property-preserving encryption, the embedding vectors can be scrambled while retaining some of their structure. The vectors can't be reversed back to their inputs (or

roughly equivalent values), but this allows them to still be queried using operations like kNN approximate nearest neighbor search and k-means clustering

## Data-in-use protection

**Only someone with the encryption key can generate an encrypted query that will meaningfully match against the encrypted data.** The key is also used to decrypt the returned results.

### What is homomorphic encryption?

Homomorphic encryption is the conversion of data into ciphertext that can be analyzed and worked with as if it were still in its original form. Homomorphic encryption enables complex mathematical operations to be performed on encrypted data without compromising the encryption.

## ENCRPYTING USING IRON CORE ALLOY

### Standard Encryption

The Alloy SDK includes methods for encrypting and decrypting data using standard encryption. These methods are built on the AES algorithm (specifically, **AES256-GCM**). Typically, data is encrypted using a technique called envelope encryption. A *data encryption key* (*DEK*) is chosen at random and used to encrypt the data, then the DEK is encrypted using another key to create an *encrypted DEK* (*EDEK*) that is stored along with the encrypted data.

Standard encryption provides the greatest security for the encrypted data, but it is not possible to search for data encrypted in this way. Each time you encrypt the same data (say the string "Hello World"), even if you use the same key, you get a different encrypted output. This is by design; it prevents someone from encrypting a data value of interest then looking for matches in the encrypted data store.

### Deterministic Encryption

If you need to be able to find data items that are the same as a specific value, but you want to preserve the privacy and security of the data, you can use *deterministic encryption*. This encryption technique is similar to standard encryption, but if you encrypt the same piece of data with the same key multiple times, you will get the same encrypted value each time. This allows you to encrypt a target value with a key then search your data store for all matching values.

Deterministic encryption is a form of *property preserving encryption* - in this case, the property that is preserved even after the data is encrypted is equality. The Alloy SDK has methods for encrypting and decrypting data using deterministic encryption. The algorithm it uses is **AES-SIV**.

**Vector Encryption**

The Alloy SDK has a set of methods specifically used to encrypt vectors of real values, such as those produced by an embedding model. The encryption algorithm is another form of property preserving encryption, similar to determinstic encryption described above. The property that is preserved by vector encryption is the distance between vectors, rather than equality. A common use case for vector embeddings is to generate a vector representing a target value that you want to search for in a vector database, using nearest neighbor search. If the vectors have been encrypted with the Alloy SDK, you can encrypt the search vector then use a standard nearest neighbor search (such as one provided by a vector database) to find close matches.

## EXPLANATION OF IMPLEMENTED RUST CODE TO ENCRYPT TEXT AND ITS EMBEDDINGS

**Encryption Process:**

1. **Text Embedding**: The text (e.g., book description) is tokenized using a pre-trained BERT tokenizer to generate embeddings (vector representations of the text).
2. **Encrypting the Embedding**: The plaintext vector (embedding) is then encrypted using Alloy's vector encryption function.
3. **Encrypting the Document**: The document (JSON) is serialized and encrypted using Alloy's standard attached encryption method. The encrypted data is then encoded in Base64 format before storing it in the payload.

**Decryption Process:**

1. **Retrieving Encrypted Data**: When querying for results, the encrypted document is retrieved from the database (Qdrant) in Base64 format.
2. **Decrypting the Document**: The Base64 string is decoded, and the encrypted data is decrypted back into a readable JSON format using Alloy's decryption method.

**Database and Searching:**

- **Qdrant** is used as the vector database, where each document is stored as a vector (encrypted) along with its payload (metadata like title and description).
- **Collection Creation**: A collection is created in Qdrant to store the data, with vectors configured for cosine similarity distance.
- **Point Insertion**: The encrypted vectors and associated payloads are inserted into the collection using the upsert_points function.
- **Search**: When querying, a text query is transformed into a vector (using the same embedding process). This query vector is encrypted and then matched against stored vectors using Qdrant's nearest-neighbor search to find relevant documents. Filters (like title matches) can be applied during the search.

**CONNECTING WITH QDRANT CONTAINER ON GOOGLE COLAB:**

We can't directly connect docker qdrant container with google colab, first we need ngrok to provide us a public url to expose our container and we give that url in the code on colab

**ISSUES WHILE ENCRYPTING THE EMBEDDINGS:**

If we first encrypt the text and generate embeddings for encrypted text, and perform similarity search it is easy process, but the semantic meaning is lost, and the results are not good when you try to search

If we generate the embeddings for normal text and encrypt the embeddings it would preserve the meaning but no vector data base support this yet using encryption algorithm.

Qdrant support this in RUST using IRON-CORE library but its implementation is python not supported due to library issues of iron core

We can use homomorphic encryption to perform searching using encrypted embeddings but without any vector db

# KNOWLEDGE GRAPH

A knowledge graph is an organized representation of real-world entities and their relationships. It is typically stored in a graph database, which natively stores the relationships between data entities. Entities in a knowledge graph can represent objects, events, situations, or concepts. The relationships between these entities capture the context and meaning of how they are connected.

**Key Characteristics**

Now that you understand how knowledge graphs organize and access data with context, let's look at the building blocks of a knowledge graph data model. The definition of knowledge graphs varies depending on whom you ask, but we can distill the essence into three key components: nodes, relationships, and organizing principles.

**Nodes**

***Nodes*** denote and store details about entities, such as people, places, objects, or institutions. Each node has a (or sometimes several) label to identify the node type and may optionally have one or more properties (attributes). Nodes are also sometimes called *vertices*.

For example, the nodes in an e-commerce knowledge graph typically represent entities such as people (customers and prospects), products, and orders:

**Relationships**

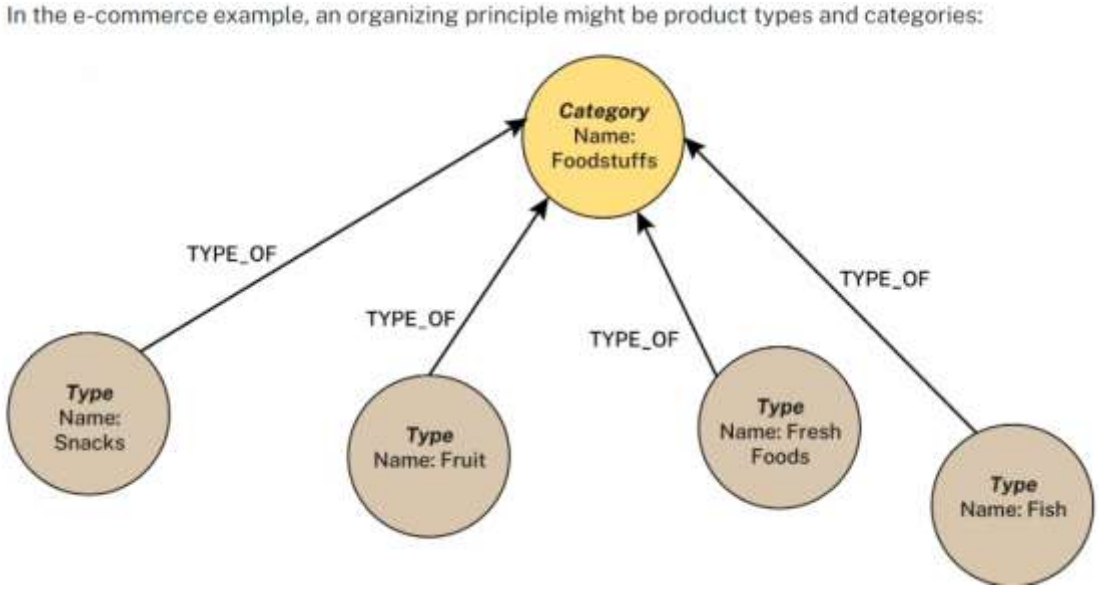
***Relationships*** link two nodes together: they show how the entities are related. Like nodes, each relationship has a label identifying the relationship type and may optionally have one or more properties. Relationships are also sometimes called *edges*.

In the e-commerce example, relationships exist between the customer and order nodes, capturing the "placed order" relationship between customers and their orders:

**Organizing Principle(s)**

***Organizing Principles*** are a framework, or schema, that organizes nodes and relationships according to fundamental concepts essential to the use cases at hand. Unlike many data designs, knowledge graphs easily incorporate multiple organizing principles.

Organizing principles range from simple (product line -> product category -> product taxonomy) to complex (a complete business vocabulary that explains the data in the graph)




## Knowledge Graphs and Graph Databases

Creating a knowledge graph involves mapping the data model and implementing it in a database. Choosing the right database simplifies design, speeds up development, and allows for future adaptability.

**Property Graphs**
Native property graph databases like Neo4j are ideal for knowledge graphs. They store information as nodes, relationships, and properties, aligning the physical database with the conceptual model.

**Key Benefits of Property Graphs:**

- **Simplicity**: Easy data modeling with minimal differences between conceptual and physical models.
- **Flexibility**: Easily add new data, properties, and relationships without extensive refactoring.
- **Performance**: Superior query speed, especially for complex traversals and many-to-many relationships.
- **Developer-friendly**: Use of GQL query language like Neo4j's Cypher reduces coding complexity.

## Use cases of knowledge graphs

There are a number of popular, consumer-facing knowledge graphs, which are setting user expectations for search systems across enterprises. Some of these knowledge graphs include:

- DBPedia and Wikidata are two different knowledge graphs for data on Wikipedia.org. DBPedia is comprised of data from the infoboxes of Wikipedia while Wikidata focuses on secondary and tertiary objects. Both typically publish in a RDF format.
- Google Knowledge Graph is represented through Google Search Engine Results Pages (SERPs), serving information based on what people search. This knowledge graph is comprised of over 500 million objects, sourcing data from Freebase, Wikipedia, the CIA World Factbook, and more.

## Benefits of Knowledge Graphs in RAG Applications

Knowledge graphs enhance Retrieval-Augmented Generation (RAG) by offering structured, contextual, and transparent data. Here's how they benefit RAG systems:

- **Structured Knowledge**: Knowledge graphs represent data as entities and relationships, making it easy to retrieve relevant information compared to unstructured text.
- **Contextual Understanding**: They capture relationships, providing deeper context, allowing RAG systems to generate more coherent responses.
- **Inferential Reasoning**: By traversing relationships, RAG systems can infer new knowledge, improving response quality and completeness.
- **Knowledge Integration**: Graphs combine information from multiple sources, creating more comprehensive responses.
- **Explainability**: Knowledge graphs provide transparency, making it easier to understand and trust the system's reasoning process.

# Difference between KNOWLEDGE GRAPH and Vector Dbs

| Feature | Knowledge Graphs | Vector Databases |
|---|---|---|
| **Data Representation** | Entities (nodes) and relationships (edges) between entities, forming a graph structure. | High-dimensional vectors, each representing a piece of information (e.g., document, sentence). |
| **Retrieval Mechanisms** | Traversing the graph structure and following relationships between entities. Enables inference and derivation of new knowledge. | Vector similarity based on a similarity metric (e.g., cosine similarity). Returns most similar vectors and associated information. |
| **Interpretability** | Human-interpretable representation of knowledge. Graph structure and labeled relationships clarify entity connections. | Less interpretable to humans due to high-dimensional numerical representations. Challenging to directly understand relationships or reasoning behind retrieved information. |
| **Knowledge Integration** | Facilitates integration by representing entities and relationships in a unified graph structure. Seamless integration if entities and relationships are mapped properly. | More challenging. Requires techniques like vector space alignment or ensemble methods to combine information. Ensuring vector compatibility can be non-trivial. |
| **Inferential Reasoning** | Enables inferential reasoning by traversing the graph structure and leveraging relationships between entities. Uncovers implicit connections and derives new insights. | More limited. Relies on vector similarity and may miss implicit relationships or inferences. Can identify similar information but not complex relationships from knowledge graphs. |

## GRAPHRAG

Indexing in a knowledge graph is the process of organizing documents and extracting knowledge from them to create a graph structure that can be queried:

### Two step process

- First one is indexing over your data to create llm derived knowledge graph
- Second one is llm orchestration to utilize these indexed memory constructs in rag operations

Help enhance search relevancy, can capture complex relationships, more wholistic understanding of data set

### Working:

- Information extraction (name entity recognition) and their semantic relations over sentence (chunk) and build knowledge graph
- Now we can perform graph machine learning over top of this to semantic aggregation and hierarchical agglomeration (collection of things)
- Now we can perform q and a on this

**Drawback**

Graph rag takes much more time, context and llm calls to give the results in compare to base rag

**USECASE**

The cases where queries do not provide much context on what to search basic rag fails to produce meaningful results, but graph rag understands the content wholistically and gives reasonable answer

## IMPLEMENTATION OF GRAPH RAG

if we divide a document into documents of langchain then convert to graph documents at once, they will be related

but if we do this separately one by one they will not be related

they are related, if on adding new chunk a node is created with the same name as we had before,we will only have one node in the final graph with the name of node that came twice for both of the chunks, so it does not matter the data would be connected if the nodes are similar, but the placing in knowledge graph is not effected by similarity imo

but related when nodes are similar, but what if nodes are not similar--> in case of all chunks at once and chunk one by one,

we can have more than one relation between two nodes

When we create embeddings of chunks of data, each chunk of data has many nodes and relations associated to it but only one embedding

**graph.query("CREATE FULLTEXT INDEX entity IF NOT EXISTS FOR (e:__Entity__) ON EACH [e.id]")?**

Creating a full-text index, like in the command you provided, serves the following purposes and benefits:

**Purpose of Full-Text Indexing**

1. **Efficient Search**: Full-text indexes are designed for efficient searching within text data. Instead of scanning all nodes or properties, the index allows quick lookups.
2. **Text Matching**: They allow for searching based on textual content within a property (e.g., id property) by enabling keyword searches, fuzzy matching, and case-insensitive searches. This is useful in cases where users may not know exact terms but want to find relevant results.
3. **Optimizing Query Performance**: Full-text indexes are particularly useful when you need to perform many searches or complex queries on textual data.

```
def structured_retriever(question: str) -> str:
    result = ""
    entities = entity_chain.invoke({"question": question})
    for entity in entities.names:
        response = graph.query(
            """CALL db.index.fulltext.queryNodes('entity', $query, {limit:2})
            YIELD node,score
            CALL {
              WITH node
              MATCH (node)-[r:!MENTIONS]->(neighbor)
              RETURN node.id + ' - ' + type(r) + ' -> ' + neighbor.id AS output
              UNION ALL
              WITH node
              MATCH (node)<-[r:!MENTIONS]-(neighbor)
              RETURN neighbor.id + ' - ' + type(r) + ' -> ' +  node.id AS output
            }
            RETURN output LIMIT 50
            """,
            {"query": generate_full_text_query(entity)},
        )
        result += "\n".join([el['output'] for el in response])
```

- Extract entities from the user question.

- Perform keyword-based matching to locate these entities in the graph.

- Retrieve and format relationships to provide context around how these entities are connected within the graph.

```
store.query(
    "MERGE (p:Person {name: 'Tomaz'}) "
    "MERGE (p1:Person {name:'Leann'}) "
    "MERGE (p1)-[:FRIEND {text:'example text', embedding:$embedding}]->(p2)",
    params={"embedding": OpenAIEmbeddings().embed_query("example text")},
)
# Create a vector index
relationship_index = "relationship_vector"
store.query(
    """
CREATE VECTOR INDEX $relationship_index
IF NOT EXISTS
FOR ()-[r:FRIEND]-() ON (r.embedding)
OPTIONS {indexConfig: {
 `vector.dimensions`: 1536,
 `vector.similarity_function`: 'cosine'
}}
""",
    params={"relationship_index": relationship_index},
)
```

```
relationship_vector = Neo4jVector.from_existing_relationship_index(
    OpenAIEmbeddings(),
    url=url,
    username=username,
    password=password,
    index_name=relationship_index,
    text_node_property="text",
)
relationship_vector.similarity_search("Example")
```

```
[Document(page_content='example text')]
```

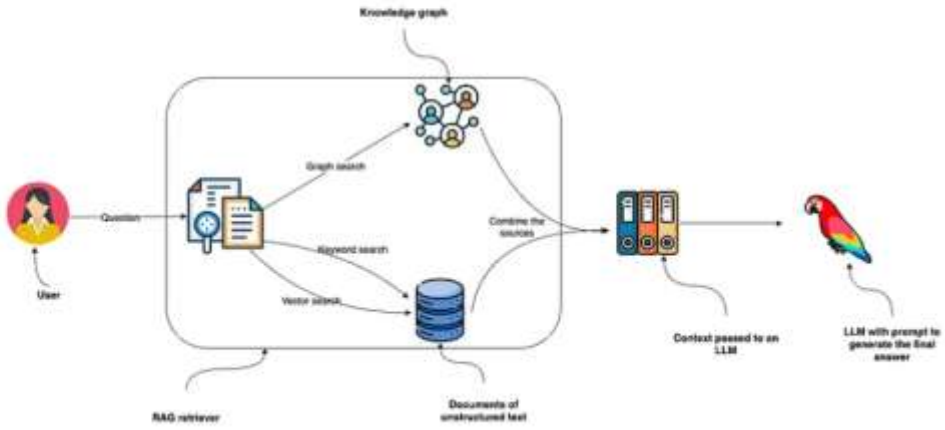
**What Happens Under the Hood**

1. **Embedding Creation**: The input query, "Example", is passed to the OpenAI embedding model to generate a vector (embedding).
2. **Similarity Calculation**: The generated embedding is then compared to the embedding values stored in the graph (from the FRIEND relationships). The vector index makes this comparison efficient by allowing fast similarity search with cosine similarity.
3. **Results**: The function will return the relationships (if any) that are most similar to the input query. These results typically include:
    - The FRIEND relationships (with their respective embeddings) that have the highest cosine similarity to the query.
    - Other relevant information, like the text property, and possibly additional metadata associated with the relationship or nodes.

Hybrid Retrieval in Graph rag

**Hybrid Retrieval for RAG**

After the graph generation, we will use a hybrid retrieval approach that combines vector and keyword indexes with graph retrieval for RAG applications.




## Create Vector Index

**cypher: CREATE VECTOR INDEX Syntax**

```
CREATE VECTOR INDEX [index_name] [IF NOT EXISTS]
FOR (n:LabelName)
ON (n.propertyName)
OPTIONS "{" option: value[, ...] "}"
```

CREATE VECTOR INDEX expects the following parameters:

- index_name - The name of the index
- LabelName - The node label on which to index
- propertyName - The property on which to index
- OPTIONS - The options for the index, where you can specify:
    - vector.dimensions - The dimension of the embedding e.g. OpenAI embeddings consist of 1536 dimensions.
    - vector.similarity_function - The similarity function to use when comparing values in this index - this can be euclidean or cosine.

```
CREATE VECTOR INDEX moviePlots IF NOT EXISTS
FOR (m:Movie)
ON m.plotEmbedding
OPTIONS {indexConfig: {
 `vector.dimensions`: 1536,
 `vector.similarity_function`: 'cosine'
}}
```

## Querying from vector indexes:

```
CALL db.index.vector.queryNodes(
    indexName :: STRING,
    numberOfNearestNeighbours :: INTEGER,
    query :: LIST<FLOAT>
) YIELD node, score
```

The procedure accepts three parameters:

1. indexName - The name of the vector index
2. numberOfNearestNeighbours - The number of results to return
3. query - A list of floats that represent an embedding

The procedure yields two arguments:

1. A node which matches the query
2. A similarity score ranging from 0.0 to 1.0

```
MATCH (m:Movie {title: 'Toy Story'})

CALL db.index.vector.queryNodes('moviePlots', 6, m.plotEmbeddi
YIELD node, score

RETURN node.title AS title, node.plot AS plot, score
```

## Knowledge graph vs Vector Db

https://www.falkordb.com/blog/knowledge-graph-vs-vector-database/

| Parameter | Knowledge Graph | Vector Database |
|---|---|---|
| Data Representation | Graph structure with nodes (entities) and edges (relationships) | High-dimensional vectors representing data points |
| Purpose | To model relationships and interconnections between entities | To store and retrieve data based on similarity in vector space |
| Data Type | Structured and semi-structured data | Unstructured data such as text, images, and multimedia |
| Use Cases | Complex Querying and Relationship Analysis, Data Management & Organization | Semantic Search, Document Retrieval |
| Scalability | Scales with complexity of relationships and entities | Scales with the number of vectors and dimensions |
| Data Updates | Can automatically update relationships with new data | Requires retraining or reindexing vectors for updates |
| Performance | Efficient for relationship-based queries | Efficient for similarity-based queries |
| Data Storage | Typically uses RDF, property graphs | Uses vector storage formats optimized for similarity search |
| Example Technologies | FalkorDB, Neo4j, NebulaGraph | Pinecone, Qdrant, Weaviate |

## When to Use a Knowledge Graph?

1. **Structured Data and Relationships:** Use KGs when complex relationships between structured data entities need to be managed. Knowledge Graphs perform excellently in scenarios where the interconnections between data points are essential, increasing search engine capabilities. For example, enterprise data management platforms.
2. **Domain-Specific Applications:** KGs can be particularly useful for applications requiring deep, domain-specific knowledge. For example, they can be specialized in effectively providing answers in domains such as healthcare and clinical decision support or Search Engine Optimization (SEO).
3. **Explainability and Traceability:** KGs offer more transparent reasoning paths if the application requires a high degree of explainability or reasoning (i.e., understanding how a conclusion was reached). For example, in financial services and fraud detection, where institutions need to explain why certain transactions are flagged as suspicious.
4. **Data Integrity and Consistency:** KGs maintain data integrity and are suitable when consistency in data representation is crucial. For example, LinkedIn's Knowledge Graph reflects consistent changes across all platforms, which is how it provides accurate job recommendations and networking opportunities.

### When to Use a Vector Database?

1. **Unstructured Data:** Vector databases are adept at capturing the semantic meaning of large volumes of unstructured data, such as text, images, or audio. For example, Pinterest can retrieve similar images by comparing images based on visual features, enabling advanced search and recommendation functionalities for users.
2. **Broad Retrieval:** Vector Databases are more suitable for applications that require broad retrieval with vague querying. For example, semantic searching can quickly dig out relevant information in a job search engine to display resumes based on limited job descriptions.
3. **Flexibility in Data Modeling:** A Vector Database can be more appropriate if there is a need to incorporate diverse data types quickly. For example, in the domain of real-

time social media monitoring, organizations can quickly analyze vast amounts of text, videos, and audio streams without having to worry about modeling the schemas beforehand

## Cypher query in Knowledge graph

### Understanding Depth Parameters in Graph Databases

Depth parameters play a crucial role in navigating and analyzing relationships within graph databases. They help define the extent of traversal allowed from a starting node to target nodes, making it easier to uncover complex relationships.

### What Are Depth Parameters?

Depth parameters determine the minimum and maximum number of hops (or relationships) traversed from one node to another. These parameters are highly useful in identifying various levels of connections within a dataset. They allow for more controlled and precise queries, ensuring that only relevant data is retrieved without overwhelming the system with unnecessary information.

### Implementing Depth Parameters in Cypher Queries

In Cypher, a query language for graph databases, depth parameters are specified directly in the relationship patterns. Here's a step-by-step breakdown of how to incorporate them:

1. **Define the Relationship Ranges**:
   Specify the range of relationships (hops) between nodes. This can be done by using an asterisk (*) followed by the range. For instance, *1..20 indicates a minimum of 1 hop and a maximum of 20 hops.
2. **Incorporate Depth in Relationships**:
   Use the depth parameters within the relationship definitions in your Cypher query. This is done by specifying the range within square brackets.
3. **Formulate the Complete Query**:
   Construct a query to navigate from a starting node to target nodes, using the defined depth parameters to control the traversal.

Here's an example query that demonstrates these steps:

```
MATCH path = (exec:EXECUTIVE)-[:REPORTS*1..20]->(metric:FINANCIAL_METRIC)-
[:IMPACTED_BY*1..20]->(cond:MARKET_CONDITION)

WHERE exec.name = 'Tim Cook'

RETURN exec, metric, cond, path;
```

In this query:

[:REPORTS*1..20] specifies that the traversal from the EXECUTIVE node to the FINANCIAL_METRIC node should involve a minimum of 1 and a maximum of 20 relationships.

Similarly, [:IMPACTED_BY*1..20] indicates the depth for traversing from FINANCIAL_METRIC to MARKET_CONDITION.

**Benefits of Using Depth Parameters**

- **Enhanced Precision**: Return only the most relevant data subsets by limiting the depth of traversal.
- **Performance Optimization**: Improve query efficiency by avoiding unnecessary deep traversals.
- **Flexibility**: Easily adjustable ranges to fit varying requirements of relationship analysis.

In summary, depth parameters are essential tools in Cypher queries for managing the complexity of relationships in graph databases. They provide both precision and flexibility, ensuring efficient and targeted data retrieval.

# Hypothetical Document Embeddings (HyDE)

**When Is It Helpful?**

The HyDE method is highly useful when:

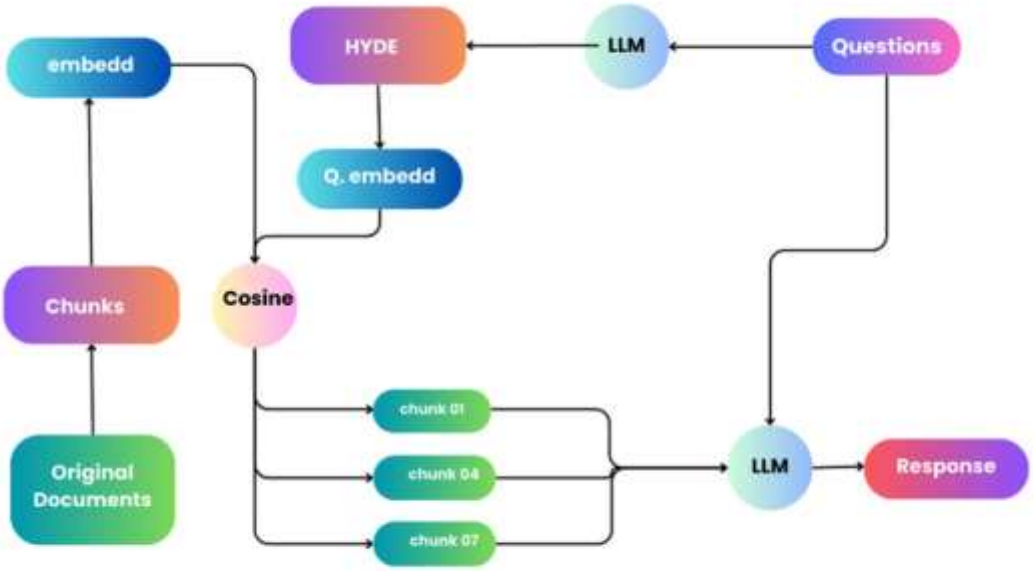
- The performance of the retrieval step in your pipeline is not good enough (for example, low Recall metric).
- Your retrieval step has a query as input and returns documents from a larger document base.
- Particularly worth a try if your data (documents or queries) come from a special domain that is very different from the typical datasets that Retrievers are trained on.

**How Does It Work?**

Many embedding retrievers generalize poorly to new, unseen domains. This approach tries to tackle this problem. Given a query, the Hypothetical Document Embeddings (HyDE) first zero-shot prompts an instruction-following language model to generate a “fake” hypothetical

document that captures relevant textual patterns from the initial query - in practice, this is done five times. Then, it encodes each hypothetical document into an embedding vector and averages them. The resulting, single embedding can be used to identify a neighbourhood in the document embedding space from which similar actual documents are retrieved based on vector similarity. As with any other retriever, these retrieved documents can then be used downstream in a pipeline (for example, in a Generator for RAG). Refer to the paper "Precise Zero-Shot Dense Retrieval without Relevance Labels" for more details.

**However, just the initial user inquiry and the discovered pertinent documents are forwarded during the final LLM step.**




RAG with HyDE

https://www.aporia.com/learn/enhance-rags-hyde/

## How filters work in weaviate

```
from weaviate.classes.query import MetadataQuery

embed_model = FastEmbedEmbeddings(model_name="BAAI/bge-base-en-v1.5")
single_text = "What is present in first and second chunks"
query_vector= embed_model.embed_query(single_text)

jeopardy = client.collections.get("data")
response = jeopardy.query.near_vector(
    near_vector=query_vector, # your query vector goes here
    limit=5,
    return_metadata=MetadataQuery(distance=True),
    filters=Filter.by_property("user_id").like("user")
)

for o in response.objects:
    print(o.properties['user_id'])
```

the output of this is

```
user_001
user_001
user_002
user_001
user_001
```

**this is understandable considering how like works**

**But when we use the filter with equal keyword the results we should get should be the one which matches exactly not which match to some extent**

```
from weaviate.classes.query import MetadataQuery

single_text = "What is present in first and second chunks"
query_vector= embed_model.embed_query(single_text)

jeopardy = client.collections.get("data")
response = jeopardy.query.near_vector(
    near_vector=query_vector, # your query vector goes here
    limit=5,
    return_metadata=MetadataQuery(distance=True),
    filters=Filter.by_property("user_id").equal("user")
)
context=[]

for o in response.objects:

    print(o.properties['user_id'])

output is

user_001
user_001
user_002
user_001
user_001
```

**in my opinion the output of this code should have been null**

**but there is something else too**

we have seen the behaviour of both like and equal, considering that

```
from weaviate.classes.query import MetadataQuery

single_text = "What is present in first and second chunks"
query_vector= embed_model.embed_query(single_text)

jeopardy = client.collections.get("data")
response = jeopardy.query.near_vector(
    near_vector=query_vector, # your query vector goes here
    limit=5,
    return_metadata=MetadataQuery(distance=True),
    filters=Filter.by_property("user_id").equal("user_0")
)
context=[]

for o in response.objects:

    print(o.properties['user_id'])
```

in the output of this we should get all the user ids we have gotten before
but this time it doesn't work like that, we get **NOTHING IN OUTPUT**
this time it works how its supposed to work and we get nothing in the output

I have found this behaviour pretty strange

## Filters in Qdrant work as they are supposed to