**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.

**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

1. **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.

1. **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

1. **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

1. **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.

1. **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.

1. **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.

**Effect of 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.

**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.
* **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.
* **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.

**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 Modeling (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.

**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**

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.

**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 consider a context window of preceding tokens to predict the next token.
* BERT 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.

**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 BERT7

 **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]

1. **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 vector​L
     + 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.

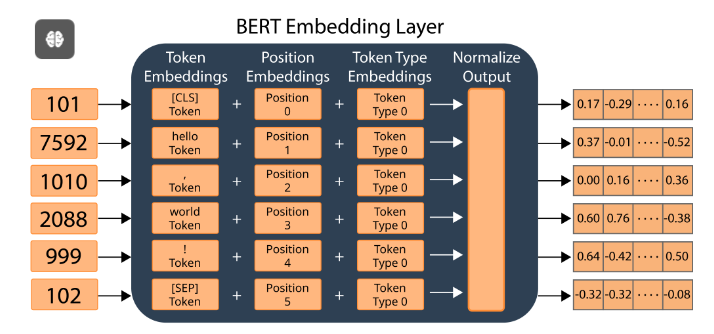
 **Forward Pass**: The embeddings are processed through the network to generate context-aware representations.

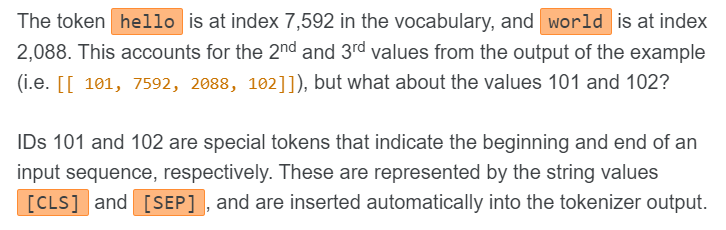
 **Prediction**: The model predicts the next token in the sequence.

 **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.

****

****

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**]**