Deep Learning & Gen AI Interview Questions and Answers

1) What is RAG, and how does it enhance LLMs?

RAG (Retrieval-Augmented Generation) enhances LLMs by **retrieving relevant documents from an external knowledge base** before generating responses. This mitigates **hallucinations**, improves factual accuracy, and allows **domain-specific fine-tuning** without modifying the LLM's internal weights.

2) How do you optimize RAG models for production?

- Efficient Indexing: Use FAISS, Weaviate, or Pinecone for optimized vector search.
- Query Expansion: Use semantic search, BM25, or hybrid search for improved retrieval.
- Latency Reduction: Use pre-filtering, caching, or asynchronous retrieval.
- Fine-tuning LLMs: Adapt the retrieval pipeline by fine-tuning models on domain-specific queries.

3) What are the key components of a RAG pipeline?

- Retriever – Fetches relevant documents (e.g., Dense Passage Retrieval, BM25).
- Generator – Generates responses using an LLM.
- Fusion Techniques – Rank and merge retrieved documents.
- Memory & Indexing – Uses Vector Databases (FAISS, Pinecone, ChromaDB).
- Evaluation Metrics – BLEU, ROUGE, Recall@K.

4) How does fine-tuning improve an LLM's performance for specific tasks?

Fine-tuning is a process where a pre-trained LLM is further trained on a **domain-specific dataset** to improve its accuracy for a particular task. LLMs, like GPT, are first trained on a vast dataset using **unsupervised learning** to understand general language patterns. However, for tasks like **legal text analysis, medical diagnosis, or customer support**, fine-tuning helps the model specialize in that area by exposing it to labeled data relevant to the domain. This process adjusts the model's weights while retaining the knowledge from the original training. Fine-tuning also helps mitigate biases and improves reliability in specialized applications. **Few-shot and zero-shot learning techniques** allow fine-tuned models to generalize better with minimal additional training data, making them efficient for real-world deployment.

5) What makes Large Language Models (LLMs) like GPT different from traditional NLP models?

Large Language Models (LLMs) like GPT differ from traditional NLP models due to their ability to process massive amounts of text data and generate human-like responses using deep learning architectures. Traditional NLP models relied on rule-based systems or statistical approaches, whereas LLMs use Transformers with self-attention mechanisms to understand context across long sequences. Unlike older methods like RNNs or LSTMs, which struggle with long-range dependencies, LLMs can handle complex queries, multi-turn conversations, and creative text generation. Another key difference is that LLMs are pre-trained on diverse datasets, allowing them to generalize well across various tasks like summarization, code generation, and reasoning, while traditional models needed to be trained separately for each specific task.

6) Why was RAG considered instead of a traditional chatbot for this project?

A traditional chatbot relies only on a pre-trained LLM, meaning it generates responses based on patterns it learned rather than retrieving factual information. This leads to hallucinations and inaccurate answers. A RAG-based system was planned because it allows the chatbot to fetch relevant FAQs from a knowledge base before generating responses, ensuring accuracy and reducing misinformation.

7) How did you ensure the chatbot retrieved the most relevant FAQs?

To ensure the chatbot retrieved the most relevant FAQs, we built a hybrid retrieval system combining vector-based search (FAISS) and keyword-based search (BM25).

First, we pre-processed and structured FAQs by removing duplicates, tagging them with metadata, and chunking long answers into smaller parts. Then, we used SBERT embeddings to store these FAQs in a vector database (FAISS), allowing us to retrieve the closest matches when a user asked a question.

For search, we combined semantic retrieval (dense vector search) with keyword retrieval (BM25) to ensure both meaning-based and exact matches were retrieved. After retrieval, we applied reranking models (Cross-Encoders) to reorder the FAQs based on relevance.

To further improve accuracy, we used confidence-based filtering to avoid low-confidence answers and implemented memory-based retrieval to maintain context in multi-turn conversations.

These optimizations significantly improved retrieval accuracy and reduced incorrect chatbot responses, ensuring customers received the most relevant information.

8) Can you explain FAISS and why it's used in RAG-powered AI?

FAISS (Facebook AI Similarity Search) is a library designed for fast, large-scale similarity search. It helps retrieve the most relevant FAQs by converting text into vector embeddings and storing them in an indexed format.

When a user asks a query, FAISS finds the most similar vectors (FAQs) efficiently, even in millions of records. We used hybrid retrieval by combining FAISS (semantic search) with BM25 (keyword search) to ensure both meaning-based and exact keyword matches were retrieved.

FAISS improves retrieval accuracy and makes chatbot responses more contextually relevant, reducing hallucinations and incorrect answers.

For example, if a customer asks *'Can I cancel my booking?'*, traditional search only finds FAQs that **contain those exact words**. FAISS, on the other hand, **converts text into numerical vectors and searches for conceptually similar results**.

In my chatbot project at Sonata, we used FAISS to ensure that users **got the most relevant answers**, even if their question was phrased differently. This improved retrieval accuracy, reduced irrelevant responses, and enhanced customer experience.

9) What are the key challenges in deploying LLMs in real-world applications?

Deploying Large Language Models (LLMs) comes with multiple challenges, including **computational cost, latency, bias, hallucination, and security risks**. LLMs require **high GPU/TPU resources** for inference, making them expensive to run in production. Latency is another concern, as generating long responses can slow down real-time applications like chatbots. **Bias and ethical concerns** arise because LLMs learn from vast datasets that may contain **prejudices, misinformation, or harmful content**, requiring careful filtering and moderation. **Hallucination** is a critical issue where LLMs generate factually incorrect or made-up responses, which can be dangerous in sensitive applications like medical or legal fields. Security vulnerabilities like **prompt injection attacks** and

data leakage risks further complicate safe deployment. To address these, companies implement **model compression, retrieval-augmented generation (RAG), reinforcement learning from human feedback (RLHF), and robust monitoring systems** to improve LLM reliability.

10) You mentioned A/B testing—can you give an example of an experiment you ran?

One A/B test we conducted was to analyze how **policy pricing changes impacted customer renewals**. We divided customers into:

- **Group A:** Offered a **5% discount on policy renewals**.
- **Group B:** Standard renewal pricing.

After **4 weeks**, we analyzed:

✓ **Conversion Rate** – % of customers who renewed.

✓ **Revenue Impact** – Did the discount result in more profit or loss?

✓ **Customer Retention Trends** – Did they continue renewing in future cycles?

The results showed that while **Group A had higher renewals**, overall **profitability was lower** due to discounts, leading to a decision to modify the offer strategy.

11) What is a vector database and how did you use it in your chatbot project?

A vector database is a system that stores and retrieves high-dimensional vector representations of data. Unlike traditional keyword-based databases, vector databases allow semantic similarity search, meaning they can find conceptually related items even if exact words don't match.

In my chatbot project at Sonata, we used a vector database (FAISS) to store FAQ embeddings. When a user asked a question, the chatbot converted it into a vector and searched for the closest matching FAQ vectors. This improved retrieval accuracy, ensuring the chatbot provided fact-based answers rather than generating random responses.

✓ **Each word is converted into a vector** → Captures meaning & relationships.

✓ **Sentence-level vectors** are used for **contextual understanding**.

✓ **Vector similarity search** retrieves **the most relevant FAQs**.

12) Can you explain what a sentence-level vector is and how it is used in NLP?

A sentence-level vector is a single numerical representation of an entire sentence, capturing its meaning and context rather than just individual words.

For example, if a user asks:

📌 *"Can I get a refund for a canceled flight?"*

A model like SBERT (Sentence-BERT) converts this into a high-dimensional vector, such as:

📌 [0.245, -0.788, 0.412, ..., 0.923]

This vector encodes the sentence's meaning in a way that allows us to compare it with other stored vectors in a vector database (like FAISS or Pinecone).

So, when the chatbot searches for an answer, it doesn't just match keywords—it finds the most semantically similar FAQ, even if the wording is different.

For example, the question *'What is the refund policy for canceled flights?'* has a high cosine similarity (0.98) with the original query, so it gets retrieved as the best match.

In my project at Sonata, we used sentence-level vectors to ensure our chatbot retrieved factually correct, context-aware responses instead of relying on simple keyword matching, reducing hallucinations and improving accuracy.

13) What are embeddings in NLP?

Embeddings are **numerical representations** of words or sentences that capture **semantic meaning**. Instead of relying on exact keyword matching, embeddings allow AI models to find **contextually similar words or sentences** in a high-dimensional space.

For example, in my chatbot project at Sonata, we used **sentence embeddings** to retrieve the most relevant FAQs. When a user asked *'Can I get a refund for a canceled flight?'*, we generated an embedding for that sentence and searched a **vector database (FAISS/Pinecone)** for the most similar stored embeddings. This allowed the chatbot to return **fact-based, contextually accurate answers**, improving retrieval accuracy and reducing hallucinations.

14) What are LLM hallucinations, and how do you reduce them?

LLM hallucinations occur when a model generates false or misleading information that is not based on real knowledge. To reduce hallucinations, I used:

✓ Retrieval-Augmented Generation (RAG) – Fetched real FAQs before generating responses.

✓ Confidence-based filtering – Only used responses if retrieval confidence exceeded a threshold.

✓ Prompt refinement – Clarified model instructions to only generate responses based on retrieved documents.

✓ Post-processing validation – Compared LLM responses with ground-truth FAQs for accuracy testing.

15) What is NLP, and how is it different from traditional text processing?

Natural Language Processing (NLP) is a subfield of AI that enables machines to understand, interpret, and generate human language. It combines linguistics, rule-based models, statistical methods, and deep learning to process text data.

- Traditional text processing: *Regex-based search* finds exact matches (e.g., “cancellation” but not “cancel”).
- NLP-based processing: Word embeddings (Word2Vec, BERT) recognize that *“cancel”* and *“cancellation”* are related.

16) What are stop words, and why do we remove them?

Stop words are common words like **'the', 'is', 'and'** that do not add much value in NLP tasks. Removing them improves efficiency by reducing dimensionality and focusing on important words.

17) What’s the difference between stemming and lemmatization?

Both are text normalization techniques used to reduce words to their base form:

✓ **Stemming** – Removes suffixes without understanding meaning.

✓ **Lemmatization** – Uses a dictionary to convert words to their root form.

Word	***Stemming***	***Lemmatization***
Running	*Run*	*Run*
Better	*Bett*	*Good*
Studies	*Studi*	*Study*

Lemmatization is more accurate but computationally expensive, whereas stemming is faster but can be inaccurate.

18) What is TF-IDF, and how does it work?

TF-IDF (Term Frequency-Inverse Document Frequency) is a technique to rank words by importance in a document. It helps remove frequently occurring but unimportant words.

TF-IDF = (Term Frequency) × (Inverse Document Frequency)

- Term Frequency (TF) = (Number of times a word appears) / (Total words in the document)
- Inverse Document Frequency (IDF) = log(Total documents / Documents with the word)

Example Calculation:

If the word *"flight cancellation"* appears 10 times in one document but is common across 1000 documents, its IDF score is low.

💡 Use Case: TF-IDF is used in document ranking, keyword extraction, and search engines.

19) How does Word2Vec work?

Word2Vec is a technique to learn word embeddings using neural networks. It uses two approaches:

✓ CBOW (Continuous Bag of Words) – Predicts a word from its surrounding words.

✓ Skip-gram – Predicts surrounding words given a central word.

Example:

- Sentence: *"The dog chased the cat."*
- CBOW: Predicts *"dog"* given (*"The"*, *"chased"*, *"the"*, *"cat"*).
- Skip-gram: Predicts *"The"*, *"chased"*, *"the"*, *"cat"* from *"dog"*.

💡 Use Case: Used in search engines, chatbots, and recommendation systems.

20) What are transformers in NLP, and why are they important?

Transformers are deep learning architectures that use self-attention mechanisms to process words in parallel instead of sequentially (like RNNs). They power models like BERT, GPT, T5, and LLaMA.

📌 Advantages:

✓ Process entire sentences at once, improving efficiency.

✓ Capture long-range dependencies between words.

✓ Used in machine translation, text generation, and chatbots.

21) How do you evaluate an NLP model's performance?

✓ **Perplexity** – Measures how well a language model predicts text (lower is better).

✓ **BLEU Score** – Compares generated text to reference translations (used in machine translation).

✓ **ROUGE Score** – Evaluates summarization quality based on word overlap.

✓ **WER (Word Error Rate)** – Used for speech-to-text accuracy."

Metric	***What It Measures***	***Best For***
BLEU (Bilingual Evaluation Understudy)	Overlap of generated vs. reference text (word-based)	Machine Translation
ROUGE (Recall-Oriented Understudy for Gisting Evaluation)	Measures recall by checking common words/phrases	Summarization
Perplexity	How well a model predicts the next word (lower is better)	Language Models (GPT, LLMs)
METEOR (Metric for Evaluation of Translation with Explicit ORdering)	Improves BLEU by considering synonyms	Translation

✓ Use classification metrics (Accuracy, F1-score) for sentiment analysis, spam detection, etc.

✓ Use BLEU, ROUGE, and Perplexity for text generation models.

✓ Use Cosine Similarity, Recall@K for retrieval-based models (chatbots, RAG).

✓ Use WER (Word Error Rate) for speech-to-text models.

22) If a word appears twice in a sentence, will it have the same embedding?

It depends on the type of embeddings used:

✓ **Word2Vec/GloVe (Static Embeddings)** – Each word has a fixed vector regardless of its position or context. So, in a sentence like *'My cat is my pet'*, both instances of *"my"* will have the exact same vector.

✓ **BERT/GPT (Contextual Embeddings)** – Words get different vectors depending on surrounding words. In this case, *"my"* will have slightly different embeddings each time because the model understands its context.

✓ Static embeddings (Word2Vec, GloVe) assign the same vector to every occurrence of a word.

✓ Contextual embeddings (BERT, GPT) generate different vectors depending on the word's context.

✓ This helps NLP models differentiate between meanings, improving accuracy in chatbots, search, and text generation.

This ability to assign different vectors based on meaning is what makes BERT and GPT superior for NLP tasks like chatbots, translation, and text retrieval.

23) How is tokenization different from embedding?

Tokenization is the process of breaking text into smaller parts (tokens) to make it easier for NLP models to process. It can be word-based, subword-based (BPE, WordPiece), or character-based.

On the other hand, embedding converts words or sentences into numerical vectors that capture semantic meaning. Unlike tokenization, embeddings allow NLP models to understand relationships between words and perform semantic search, similarity matching, and text generation.

For example, the sentence "ChatGPT is amazing!" after tokenization becomes ["ChatGPT", "is", "amazing", "!"]. Each of these tokens can then be converted into an embedding like [0.45, -0.78, 0.12, ...] for deeper language understanding.

24) Define Tokenization?

Tokenization is the process of breaking text into smaller units, known as tokens, to make it easier for NLP models to process. These tokens can be words, subwords, or individual characters, depending on the type of tokenization applied.

There are three main types of tokenization. Word tokenization splits text into individual words, such as converting "I love NLP" into ["I", "love", "NLP"]. Subword tokenization, which includes methods like Byte Pair Encoding (BPE) and WordPiece, breaks words into smaller meaningful units, especially useful for handling rare or unknown words. For example, "unhappiness" would be tokenized as ["un", "happiness"]. Character tokenization further breaks down text into individual characters, such as "AI" becoming ["A", "I"].

For instance, if we take the sentence "ChatGPT is amazing!", word tokenization would produce ["ChatGPT", "is", "amazing", "!"]. Subword tokenization might break "ChatGPT" into ["Chat", "##GPT"], while character tokenization would split it into ["C", "h", "a", "t", "G", "P", "T", "i", "s", "a", "m", "a", "z", "i", "n", "g", "!"].

25) How This RAG Model Works for TUI's Chatbot? (Personalized question for my project)

The RAG-powered AI chatbot we were building followed a **Hybrid RAG model architecture** integrated with an **LLM** to ensure accurate FAQ retrieval and response generation.
When a user submits a query, such as *"How long does it take to get the refund amount?"*, the system first **converts the query into vector embeddings** and stores them in the **FAISS vector database**. During retrieval, the model performs a **hybrid search**—first using **BM25** for **keyword-based retrieval**, then leveraging **FAISS for semantic search**, allowing it to retrieve both exact phrase matches and contextually similar documents.
Once the relevant documents are retrieved, they are **ranked using cosine similarity** and **filtered based on confidence thresholds** to ensure accuracy. In the generation phase, an **LLM (such as GPT-4 or BART)** takes the top-ranked FAQ as input and formulates a **coherent, policy-compliant response** that aligns with TUI's official guidelines. Finally, the system delivers the **final response**, ensuring that the chatbot provides **factually accurate, retrieval-grounded answers**, minimizing the risk of hallucinations and improving customer support reliability.

26) Explain the transformer architecture?

The Transformer architecture is a deep learning model introduced in the paper *"Attention Is All You Need"*. It replaces sequential processing (RNNs) with a fully attention-based mechanism, enabling parallelization and better long-range dependencies. The core components include:
1 **Tokenization & Embeddings:** Convert words to numerical vectors.
2 **Positional Encoding:** Injects word position information.
3 **Self-Attention:** Each word attends to other words in the sentence.
4 **Multi-Head Attention:** Different heads focus on different aspects of relationships.
5 **Feed-Forward Layers:** Transform attention outputs for better understanding.
6 **Final Output:** Passes to classification or text generation layers.

27) Can you explain the difference between Multi-Head Attention and Cross Attention in Transformers?

Both Multi-Head Attention (MHA) and Cross Attention are attention mechanisms in Transformers, but they serve different purposes.
Multi-Head Attention is used in both the encoder and decoder to help words in a sentence relate to each other. Instead of computing a single attention score, it splits the input into multiple attention heads, allowing the model to capture different types of relationships in parallel. For example, in the sentence 'The cat sat on the mat,' one attention head might focus on the subject-verb relationship ('cat' → 'sat'), while another might focus on location ('sat' → 'on the mat').
Cross Attention, on the other hand, is used only in the decoder. It allows the decoder to focus on the encoder's output while generating the translation. For instance, when translating 'The cat sat on the mat' into Tamil ('பூனை மெத்தையில் அமர்ந்தது'), Cross Attention helps the decoder align 'cat' with 'பூனை' and 'sat' with 'அமர்ந்தது'. The attention weights ensure that each generated word is linked to the correct source word.
The key difference is that Multi-Head Attention helps words within the same sequence interact, while Cross Attention helps the decoder learn from the encoder's context when generating output**. Cross Attention works by taking the Key (K) and Value (V) from the encoder output, while the Query (Q) comes from the decoder.**
This is different from Self-Attention, where Q, K, and V all come from the same input sequence.

28) Zero-Shot Prompting?

Zero-shot prompting is when you ask an LLM to perform a task without providing any examples. The model relies on its pre-trained knowledge to generate a response. **When to Use:**

- When the task is **simple and well-known** to the model.
- When you **don't have labeled examples** to guide the model.
- When experimenting with how well the model understands a task.

29) Few-Shot Prompting?

Few-shot prompting provides **a few labeled examples** before asking the model to complete a task. This improves accuracy by helping the model understand the pattern. **When to Use:**

- When **zero-shot prompting is inaccurate** or inconsistent.
- When the task is **specific** and requires examples for clarity.
- When you need **better generalization** across different inputs.

30) Chain-of-Thought (CoT) Prompting?

Chain-of-Thought prompting forces the model to think step by step, improving logical reasoning and problem-solving accuracy. When to Use:

- When the task requires reasoning, such as math problems or logical inference.
- When the model jumps to conclusions and gives incorrect answers.

Why is CoT Useful?

- Helps the model avoid mistakes in complex reasoning tasks.
- Works well for math problems, logic puzzles, and decision-making tasks.
- Makes the AI more interpretable because you can see how it reached its answer.

Example:

A store sells apples for $3 each. If you buy 4 apples and pay with a $20 bill, how much change do you get? Think step by step.

LLM Output:

1. Each apple costs $3.
2. Buying 4 apples costs 4 × $3 = $12.
3. Paying with $20 means the change is $20 - $12 = $8.
   Final Answer: $8

31) Role-Based Prompting?

Role-Based Prompting is a prompt engineering technique where you assign a specific role to the AI before asking it to generate a response. This helps the model adopt a particular style, tone, or expertise level when answering. **When to Use:**

- When the task requires **domain-specific knowledge** (e.g., legal, medical).
- When you need a **structured response** based on a professional role.

Example: AI as a Data Scientist (Providing Technical Details):

You are an experienced data scientist. Explain how a decision tree algorithm works in machine learning.

32) What are Delimiters in Prompt Engineering?

Delimiters are **symbols or text markers** used in prompts to **clearly define the input boundaries**, helping the LLM understand which parts of the text belong to instructions, examples, or user input.

Example:

Extract the job title from the following:

<job>

Title: Data Scientist
Company: SecureKloud
</job>
Output:
Title: Data Scientist
Using <job> as a delimiter helps the AI focus only on extracting the job title.

33) What is the role of the temperature parameter in LLMs?

The temperature parameter controls the randomness of an LLM's responses. A lower value makes outputs more deterministic, while a higher value makes them more creative and diverse. A higher temperature (0.8 - 1.5) encourages more creative and diverse responses. A low temperature makes the model's responses more focused and predictable, which is useful for fact-based answers, coding assistance, and structured summarization.
Imagine you ask an AI model:
Prompt:
"Give me a short description of a cat."
🔹 **Low Temperature (0.2) - More Predictable, Conservative Response**
"A cat is a small, furry animal that is often kept as a pet. It is known for its agility and independence."
✓ **Why?** The model picks the **most likely** words, resulting in a **safe, factual** response.

🔹 **High Temperature (1.2) - More Creative, Random Response**
"A cat is a mischievous fluffball, a tiny tiger that rules your house with elegance and occasional chaos."
✓ **Why?** The model picks **more diverse** words, leading to a **more imaginative** response.

34) Explain Top-K sampling parameter?

Top-k sampling is a decoding technique used in Large Language Models (LLMs) to control randomness by limiting word choices to the k most probable words at each step. Instead of selecting from all possible words, the model only considers the top k most likely words, ensuring more controlled and meaningful outputs.

Example:
Let's say you ask an LLM:
Prompt: "The sky is..."
The model predicts the next word probabilities:

- "blue" → **0.35**
- "clear" → **0.25**
- "dark" → **0.15**
- "red" → **0.05**
- "raining" → **0.02**
- "full" → **0.01**
- ... many other words (very low probability)

Scenario 1: Without Top-k (Full Sampling)

- The model could choose any word, even those with very low probability, leading to random or irrelevant results.

Scenario 2: With Top-k (k = 3)

- The model keeps only the top 3 words:
  ["blue" (0.35), "clear" (0.25), "dark" (0.15)]
- The next word is chosen randomly from these 3, making the response more predictable and relevant.

35) Explain Top-P Sampling (Nucleus Sampling) parameter?

Top-p sampling, also known as nucleus sampling, is a decoding technique used in Large Language Models (LLMs) to generate more diverse yet coherent responses. Unlike Top-k sampling, which picks from a fixed number (k) of highest probability words, Top-p selects a dynamic set of words whose cumulative probability adds up to p (e.g., p = 0.9 means the model selects words until their combined probability reaches 90%**).

Example:

Imagine prompting an LLM with:

Prompt: "The sky is..."

The model predicts the next word probabilities:

- "blue" → **0.40**
- "clear" → **0.25**
- "dark" → **0.15**
- "cloudy" → **0.10**
- "stormy" → **0.05**
- "full" → **0.02**
- "empty" → **0.01**
- … many others (low probability)

Scenario 1: With Top-k (k = 3)

- The model picks from only the top 3 words:
  ["blue", "clear", "dark"]
- But what if the 4th word ("cloudy") is still a reasonable choice? Top-k wouldn't include it.

Scenario 2: With Top-p (p = 0.9)

- The model keeps words until their cumulative probability reaches 90%:
  ["blue" (0.40), "clear" (0.25), "dark" (0.15), "cloudy" (0.10)]
- The next word is randomly selected only from these four words.
- The word "stormy" (0.05) is excluded because adding it would exceed 90%, ensuring coherence.

36) Explain Max Tokens parameter?

Max Tokens is a parameter that controls the maximum number of tokens an LLM can generate in response to a prompt. It helps limit output length, manage computational costs, and maintain response relevance.

✅ Max Tokens controls response length, preventing overly long or short answers.

✅ Setting it too low may cut off important information.

✅ Setting it too high may cause unnecessary verbosity or increased costs (in API-based models).

37) Similarly explain the different quantization techniques with clear examples?

Quantization is a technique used to **reduce the memory and computational requirements** of Large Language Models (LLMs) by converting high-precision numerical values into lower-precision representations. This improves efficiency without significantly affecting accuracy. Quantization reduces the number of bits used to represent model parameters (like weights and activations) from 32-

bit floating-point (FP32) to lower-precision formats such as 16-bit (FP16), 8-bit (INT8), or even 4-bit (INT4).

🔷 Why is Quantization Important?

✅ Reduces memory usage (helps deploy models on edge devices).

✅ Speeds up inference time (faster responses).

✅ Reduces power consumption (useful for mobile & embedded systems).

🔷 1. Post-Training Quantization (PTQ)

📌 Definition:

PTQ applies quantization after the model has been fully trained. It reduces the model size but may slightly affect accuracy.

📌 Example:

- A GPT model trained in FP32 (32-bit) is converted to INT8 (8-bit).
- This reduces the memory usage by 4× (from 32-bit to 8-bit).

✅ Best for: When you have a pre-trained model and want to optimize it for deployment.

🚀 Real-World Example:

Google's TFLite (TensorFlow Lite) applies Post-Training Quantization to reduce model size for mobile devices.

🔷 2. Quantization-Aware Training (QAT)

📌 Definition:

QAT simulates quantization during training so the model learns to adapt to lower precision. This results in better accuracy compared to PTQ.

📌 Example:

- Instead of training in FP32 and then converting, QAT trains using simulated INT8 weights.
- The model adjusts its parameters to minimize accuracy loss from quantization.

✅ Best for: When high accuracy is needed after quantization.

🚀 Real-World Example:

NVIDIA’s TensorRT QAT helps models maintain high accuracy in autonomous driving applications.

🔷 3. Dynamic Quantization

📌 Definition:

Dynamic quantization only quantizes weights, while activations stay in high precision (FP32). It’s applied only during inference (runtime).

📌 Example:

- A model keeps activations in FP32 but stores weights in INT8.
- During inference, activations are converted to INT8 when needed, reducing computation cost.

✅ Best for: Reducing latency in CPU-based inference while keeping accuracy high.

🚀 Real-World Example:

Microsoft’s ONNX Runtime uses Dynamic Quantization for deploying transformer models on CPUs.

🔷 4. Static Quantization

📌 Definition:

Static quantization pre-computes weight and activation quantization before inference. It’s more optimized than dynamic quantization.

📌 Example:

- A BERT model quantized to INT8 for both weights & activations before deployment.
- This reduces both memory and latency significantly.

✅ Best for: Running models efficiently on edge devices (e.g., mobile & IoT).

🚀 Real-World Example:
Facebook's QNNPACK framework uses static quantization for mobile AI models.

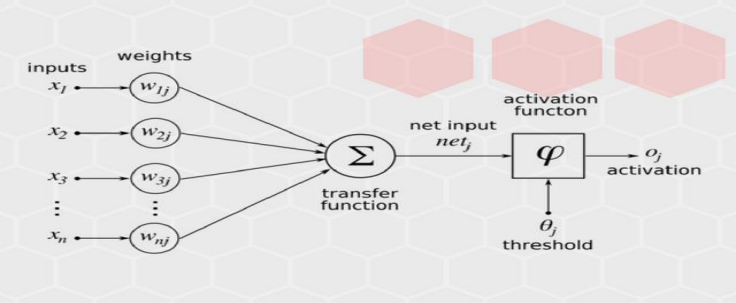
- Use PTQ for quick post-training optimizations.
- Use QAT when you need higher accuracy after quantization.
- Use Dynamic Quantization when running models on CPUs.
- Use Static Quantization for mobile and edge devices.
- Use Weight-Only Quantization for large-scale LLMs.

38) Explain the Neural Network Architecture: A Complete Breakdown

Neural networks are the backbone of deep learning, and their architecture consists of different layers and components. Basic Structure of a Neural Network:

A neural network consists of neurons arranged in layers, where each neuron processes and transmits information. The structure is as follows:

1. **Input Layer**
    - The first layer of the network that takes in the raw data (e.g., images, text, numbers).
    - Each neuron in this layer corresponds to a feature in the input data.
2. **Hidden Layers**
    - Layers between the input and output layers.
    - Perform computations using **weights, biases, and activation functions**.
    - The depth of a network depends on the number of hidden layers.
3. **Output Layer**
    - Produces the final prediction.
    - The number of neurons in this layer depends on the problem (e.g., 1 neuron for binary classification, multiple neurons for multi-class classification).



Activation Functions:

Activation functions introduce non-linearity into the neural network, allowing it to learn complex patterns. Some commonly used activation functions are:

- **Sigmoid**: Maps inputs to values between 0 and 1, commonly used for binary classification.
- **ReLU (Rectified Linear Unit)**: Outputs zero for negative values and retains positive values, making it efficient for deep networks.
- **Tanh**: Similar to the sigmoid but maps values between -1 and 1.
- **Softmax**: Converts outputs into probabilities, making it useful for multi-class classification problems.

Each activation function has its own advantages and disadvantages, and the choice depends on the problem being solved.

Loss Function:

The loss function quantifies the error between the predicted output and the actual label. It guides the learning process by providing a measure that needs to be minimized. Some common loss functions include:

- **Mean Squared Error (MSE)**: Used for regression problems by measuring the average squared difference between actual and predicted values.
- **Binary Cross-Entropy**: Used for binary classification tasks where the output is a probability.
- **Categorical Cross-Entropy**: Used for multi-class classification tasks.

Gradient and Backpropagation:

The gradient is the rate of change of the loss function with respect to the weights. It indicates how much a small change in the weight affects the loss. Backpropagation is the process of computing these gradients and using them to adjust the weights to minimize the loss.

Backpropagation follows these steps:

1. Compute the forward pass and calculate the output.
2. Compute the loss function.
3. Calculate the gradient of the loss with respect to each weight using partial derivatives.
4. Update the weights using gradient descent.

Backpropagation uses the chain rule of differentiation to calculate gradients layer by layer.

Gradient Descent:

Gradient descent is an optimization algorithm used to update the weights of the neural network in a way that reduces the loss function. There are different types of gradient descent:

- **Batch Gradient Descent**: Computes gradients using the entire dataset, leading to stable updates but slow computation.
- **Stochastic Gradient Descent (SGD)**: Updates weights after each individual data point, making learning faster but noisier.
- **Mini-Batch Gradient Descent**: Uses small batches of data for updates, balancing speed and stability.

Vanishing and Exploding Gradient Problem:

The vanishing gradient problem occurs when gradients become too small, leading to very slow learning in deep networks. This is common when using the sigmoid or tanh activation functions. The exploding gradient problem happens when gradients become excessively large, causing unstable training.
Solutions to these problems include:

- Using **ReLU** activation instead of sigmoid/tanh.
- **Batch normalization**, which normalizes activations layer-wise.
- **Gradient clipping**, which restricts gradient values within a certain range.

Regularization Techniques:
Regularization helps prevent overfitting by reducing model complexity. Some techniques include:

- **L1 Regularization (Lasso)**: Adds an absolute weight penalty, encouraging sparsity.
- **L2 Regularization (Ridge)**: Adds a squared weight penalty, preventing large weight values.
- **Dropout**: Randomly drops neurons during training, forcing the network to generalize better.
- **Batch Normalization**: Normalizes layer inputs to stabilize and accelerate training.

Learning Rate and Hyperparameter Tuning:
The learning rate controls how big the weight updates are. A high learning rate may cause divergence, while a low learning rate slows convergence. Hyperparameter tuning involves optimizing values such as the learning rate, number of layers, number of neurons, and batch size to improve model performance.
Techniques like grid search and random search are used to find the best hyperparameters. Adaptive learning rate methods like Adam and RMSprop adjust the learning rate dynamically.
Optimizers:
Optimizers improve the efficiency of training by adjusting weights effectively. Some common optimizers include:

- **SGD (Stochastic Gradient Descent)**: A basic optimizer with a constant learning rate.
- **Adam (Adaptive Moment Estimation)**: Dynamically adjusts learning rates based on past gradients.
- **RMSprop**: Designed for non-stationary problems, preventing rapid fluctuations in gradients.

Choosing the right optimizer depends on the problem, dataset, and network architecture.

These operations take place in each and every neuron in the network, including neurons in the input, hidden, and output layers. Here's how:

1. **Weights and Biases:**
    - Every connection between neurons has a weight.
    - Each neuron (except input neurons) has a bias.
    - The weighted sum of inputs and the bias are computed at each neuron in the network.
2. **Activation Function:**
    - Applied at each neuron (except input neurons) after computing the weighted sum.
    - It introduces non-linearity and determines whether the neuron should be activated.
3. **Gradient Computation (Backpropagation):**
    - Gradients are calculated for each neuron to update its weights and biases.
    - The chain rule is applied layer by layer, passing gradients from the output layer back to each neuron in previous layers.

In short, these operations do not happen only at the hidden layers but at every neuron in the hidden and output layers during forward propagation and backpropagation.

39) Explain the steps in CNN in short?

Order of Operations in CNN:

1 Input Image →

2 Padding (if required) →

3 Convolution + Activation Function (e.g., ReLU) →

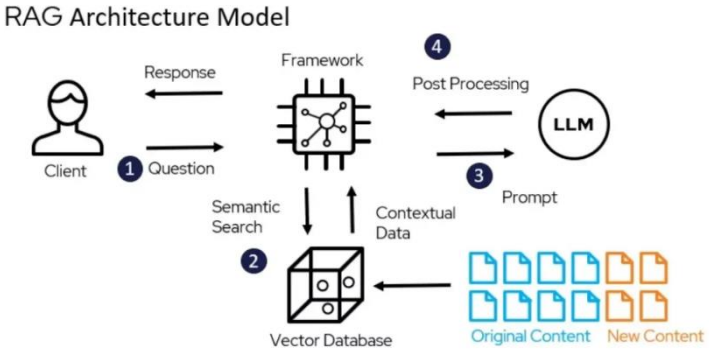
4 Pooling (to reduce dimensions and retain important features) →

5 Repeat for multiple layers →

6 Flatten the feature maps and pass them to a Fully Connected Layer →

7 Final classification output (Softmax or Sigmoid for probabilities)

40) Explain how RAG Works in Simple Steps?



1 User Query → The user asks a question.

2 Query Embedding → The input is converted into a vector representation using an embedding model.

3 Retrieval from KB (Vector DB) → The retriever searches for similar embeddings in the knowledge base (vector database).

4 Relevant Documents Fetched → The most relevant documents are retrieved.

5 Augment LLM Input → The retrieved documents are added as context to the LLM prompt.

6 LLM Generates Response → The generator (LLM like GPT) creates the best possible response based on the retrieved information.

So, in a RAG model, when a user asks a query, instead of training, the model is connected with the external knowledge base which is a vector DB with embeddings. The input is also converted into embeddings and the system matches it with the data in KB. relevant documents are retrieved by the retriever component and are handed to the LLM. The generator then generates the most optimal, compliance effective output for the user using the LLM like GPT.

The **framework** acts as the **orchestrator** that integrates:

1. **Query processing** – Converts user input into embeddings
2. **Retrieval** – Fetches relevant documents from the vector DB
3. **Augmentation** – Passes retrieved knowledge to the LLM
4. **Generation & Post-processing** – Refines the final response before sending it to the user

💡 Some popular RAG frameworks include:

✅ **LangChain** – Popular for building RAG with OpenAI, Hugging Face, etc.

✅ **LlamaIndex** – Specializes in knowledge retrieval for LLMs.

✅ **Haystack** – Open-source NLP framework with RAG capabilities.

41) Fine-Tuning vs. RAG – Key Differences?

- Fine-tuning = Learning new knowledge by modifying LLM weights.
- RAG = Enhancing responses by retrieving external knowledge dynamically.
- Fine-tuning is "learning," while RAG is "looking up information on demand.

Feature	Fine-Tuning (Training the Model)	RAG (Retrieval-Augmented Generation)
Method	Modifies LLM weights by training on custom data.	Keeps LLM unchanged, retrieves relevant data dynamically.
Data Storage	The model stores learned knowledge in its weights.	Uses an external knowledge base (vector DB) for real-time retrieval.
Computational Cost	Expensive (GPU-intensive), requires labeled datasets.	Cheaper (no model retraining, only retrieval).
Flexibility	Knowledge is static after fine-tuning, needs retraining for updates.	Knowledge is dynamic—just update the vector database.
Response Quality	Works well for structured knowledge, domain-specific tasks.	Best for answering factual questions with up-to-date information.
Customization	Allows deep customization of model behavior and language style.	Limited to retrieved data but maintains the LLM's original behavior.
Implementation Complexity	Requires significant data, tuning, and computing resources.	Easier to implement, mostly requires vector DB and retriever setup.
Use Cases	Specialized chatbots, industry-specific applications, sentiment analysis.	Real-time search, FAQ bots, knowledge-based applications.
Updates & Maintenance	Requires full retraining for knowledge updates.	Simply update the knowledge base, no model retraining needed.

42) How is a Sentence Converted into a Vector?

A sentence is first broken into tokens, then each token is mapped to a high-dimensional vector using a pre-trained model. The transformer processes these embeddings to capture semantic meaning, and a single vector representation is generated through pooling and normalization. This final vector helps in similarity searches in FAISS. We use a sentence embedding model like SBERT or OpenAI embeddings. The model takes the text and maps it to a high-dimensional vector space where semantically similar sentences have similar embeddings. Then, FAISS is used to retrieve the closest matching vectors.

43) Complete RAG-Based Chatbot Workflow – Step-by-Step Explanation?

1 Data Collection

To build a high-quality chatbot, the first step is gathering relevant data. In this case, travel-related FAQs, support documents, and user queries from chatbot logs were collected. The data sources included internal company documents, scraped web data (using BeautifulSoup or Scrapy), and third-party APIs like TripAdvisor and Expedia. The challenge here is ensuring that the dataset is comprehensive and structured, as raw text can often be inconsistent. Tools like Pandas and regex were used to clean and standardize the collected data, ensuring that duplicate or irrelevant entries were removed.

---

2 Data Preprocessing & Vectorization

Once the raw text was collected, it was processed to make it usable for retrieval. This involved text cleaning (removing stopwords, lemmatization using NLTK or SpaCy), tokenization (splitting text into words or phrases), and embedding generation (converting text into vector representations). The embeddings were created using OpenAI's text-embedding-ada-002 model or Sentence-BERT (SBERT), which maps sentences into high-dimensional vectors. The embeddings were then stored in a vector database like FAISS or Pinecone, which allows for fast similarity searches during retrieval.

---

3 Building the RAG Pipeline

Retrieval-Augmented Generation (RAG) combines information retrieval with an LLM. When a user asks a question, the system converts the query into an embedding and searches for similar documents in the vector database. The most relevant documents are retrieved using FAISS (Facebook AI Similarity Search) or Pinecone, ranked using cosine similarity, and passed to the LLM (GPT-4, Llama-2, etc.) as context. This ensures that the chatbot responds with fact-based answers instead of generating responses purely from its internal knowledge.

---

4 Model Development (Retriever + Generator)

The retriever model (FAISS/Pinecone) searches for similar embeddings based on user queries, ensuring that only relevant information is sent to the generator model (LLM). The LLM then generates the final response based on both the query and retrieved context. This step uses OpenAI's GPT API, Hugging Face Transformers, or LlamaIndex to generate accurate and context-aware responses. The quality of retrieval significantly impacts the accuracy of responses, making it crucial to fine-tune embedding models for better results.

---

5 Optimizing Prompt Engineering

To improve response quality, prompt engineering techniques were used. This involved few-shot learning (providing examples before the query), instruction tuning (guiding the LLM with structured instructions), and guardrails (adding rules to prevent hallucinations). LangChain was used to structure prompts dynamically, ensuring that retrieved documents were inserted into the prompt in a logical manner before being sent to the LLM. Techniques like temperature tuning (controlling randomness in responses) and response re-ranking (ordering responses based on relevance) helped improve chatbot accuracy.

---

6 Model Evaluation & Testing

After developing the RAG system, the next step was evaluating how well it performed. Metrics like Recall@K (measuring how many relevant documents were retrieved), BLEU/ROUGE scores (comparing generated responses with actual answers), and hallucination rate (measuring incorrect information generation) were used. The evaluation involved running test queries and comparing chatbot responses with human-labeled ground truth. Tools like Power BI, Pandas, and Hugging Face's evaluate library were used to analyze and visualize performance metrics.

---

7 Deployment & Scaling

Finally, the chatbot was deployed in a production environment. The LLM was hosted on OpenAI/Azure OpenAI API, while FAISS/Pinecone stored and served the embeddings. A FastAPI backend handled user queries, integrating with the vector database and the LLM API for real-time responses. To optimize speed, asynchronous API calls and batch processing were implemented, ensuring that multiple user queries could be handled simultaneously. For cost efficiency, a hybrid

approach was explored, where simpler queries were answered using a local embedding model, and complex ones were sent to the LLM.

44) How do you handle multiple data sources and convert them into a format for RAG?

Since data can come from PDFs, CSVs, text files, and JSON, I preprocess them using different tools like PyMuPDF for PDFs, Pandas for structured data, and basic file operations for text files. The extracted text is cleaned, formatted, and stored in a JSON format, which is then used to generate embeddings. JSON is often preferred as it allows storing metadata like document source and timestamps, making retrieval more effective in the RAG system.

45) Does text cleaning happen before or after text extraction?

Text extraction happens first to pull raw text from different sources like PDFs, CSVs, or JSON. After that, we perform text cleaning to remove noise, standardize formatting, handle special characters, and tokenize text. This ensures that the text is in a structured and consistent format before we generate embeddings and store it in the vector database.

46) How is text converted into embeddings in a RAG pipeline?

We use transformer-based models like SBERT to convert text into dense vector representations. The model tokenizes the text, processes it through a neural network, and applies pooling to generate a fixed-size embedding. These embeddings are then stored in a vector database for efficient similarity search during retrieval.

47) Why use SBERT instead of normal BERT?

SBERT is optimized for sentence embeddings, meaning it produces meaningful vector representations that work well for semantic search and retrieval in RAG. Unlike standard BERT, which requires pairwise comparison for similarity, SBERT directly outputs sentence-level embeddings, making it faster and more efficient.

48) How does SBERT convert sentences into vectors?

SBERT tokenizes the input sentence, passes it through transformer layers to generate contextual embeddings, applies pooling to get a fixed-length sentence vector, and outputs a dense numerical representation. This vector can be used for retrieval tasks like RAG.

49) Explain the architecture of SBERT completely?

SBERT (Sentence-BERT) is an improved version of BERT that is designed to understand the meaning of entire sentences, not just words. It takes two sentences at a time and processes them using the same neural network (like twins, or "Siamese" networks). Instead of analyzing individual words separately, SBERT converts entire sentences into fixed-length numerical representations (called embeddings) that capture their meaning. It uses techniques like averaging all the word embeddings or taking the special [CLS] token to get a single sentence representation. During training, it learns to make similar sentences have similar embeddings and different sentences have distinct embeddings. This helps in tasks like finding similar sentences, searching for relevant information, and clustering related texts efficiently.

Mean Pooling (Averaging Word Embeddings):

- BERT processes a sentence and produces an embedding for each token (word or subword).
- These token embeddings are stacked together, forming a matrix.

- **Mean pooling** takes the average of all token embeddings to get a single vector for the entire sentence.

Let's take a simple example to understand why mean pooling usually works better than using just the [CLS] token.

Example Sentence:

"The cat sat on the mat."

How BERT Processes This Sentence:

BERT breaks this sentence into tokens and assigns a numerical vector (embedding) to each token.

Let's assume each token gets a **3-dimensional** (Max. either 384 or 768-d) embedding (for simplicity).

Token	Embedding (Example)
[CLS]	[0.1, 0.2, 0.3]
The	[0.5, 0.4, 0.3]
cat	[0.8, 0.7, 0.6]
sat	[0.6, 0.5, 0.7]
on	[0.2, 0.3, 0.4]
the	[0.5, 0.6, 0.5]
mat	[0.7, 0.8, 0.9]

In mean pooling, we average the embeddings of all the words (except [CLS] and [SEP]).

Token Embeddings: [0.5, 0.4, 0.3]
[0.8, 0.7, 0.6]
[0.6, 0.5, 0.7]
[0.2, 0.3, 0.4]
[0.5, 0.6, 0.5]
[0.7, 0.8, 0.9]

Summing All Embeddings: [0.5 + 0.8 + 0.6 + 0.2 + 0.5 + 0.7,
0.4 + 0.7 + 0.5 + 0.3 + 0.6 + 0.8,
0.3 + 0.6 + 0.7 + 0.4 + 0.5 + 0.9]

[3.3, 3.3, 3.4]

Dividing by Number of Tokens (6): [3.3 / 6, 3.3 / 6, 3.4 / 6]

Final Sentence Embedding: [0.55, 0.55, 0.57]

This embedding considers all words in the sentence, making it a more complete representation of its meaning.

50) How do you manage the computational workload (For RAG)? (Personalized question for my project)

For training, we leveraged AWS p4d instances with NVIDIA A100 GPUs, which provided high-speed parallel processing for embedding generation and retrieval tasks. For inference, we optimized performance to run efficiently on CPUs, reducing costs while maintaining real-time response capabilities.

51) How do you manage the computational workload (For XGBoost)? (Personalized question for my project)

💻 **CPU for:**

✅ SQL-based claims data analysis (AWS Redshift, Pandas on high-RAM instances)

✅ Data preprocessing & feature engineering (handling missing values, transformations)
✅ Model inference (serving predictions in real-time)
🚀 **GPU for:**
✅ XGBoost training (only if dataset is very large, e.g., 10M+ records)
✅ SMOTE (only if high-dimensional dataset slows down CPU processing)
For claims data processing and feature engineering, we used CPU-based instances since SQL and Pandas work efficiently on high-RAM machines. For XGBoost training, we initially used CPUs, but for very large datasets, we leveraged AWS GPUs to speed up model training. Model inference was done on CPU to optimize costs.

52) What is Amazon Redshift?

Amazon Redshift is a fully managed, cloud-based data warehouse service designed for large-scale data analytics. It enables businesses to run complex queries on petabyte-scale data quickly using columnar storage and MPP (Massively Parallel Processing) architecture. For AI workloads, Redshift is used as a data warehouse for structured data, and models access it via SQL or Python APIs like SQLAlchemy.

52) How can we identify if the model overfits?

Overfitting can be detected by checking if there's a large gap between training and validation/test performance. A useful technique is plotting the loss curves—if training loss keeps decreasing while validation loss starts increasing, it indicates overfitting. We can also use k-fold cross-validation to check how well the model generalizes across different subsets of data.
We can determine overfitting by comparing training and validation/test metrics. If the training accuracy is very high but validation/test accuracy is significantly lower, it indicates overfitting. Some techniques to check this include:"
✅ **Train-Test Performance Gap** → If training accuracy is much higher than test accuracy, it's a red flag.
✅ **Validation Loss Curve** → If validation loss starts increasing while training loss keeps decreasing, it means the model is memorizing training data instead of generalizing.
✅ **Cross-Validation** → If the model performs well on some validation folds but poorly on others, it's likely overfitting.
✅ **Regularization Effect** → If adding L1/L2 regularization improves test performance but reduces training accuracy, the original model was likely overfitting.

53) Complete Explanation of XGBoost from Start to End?

XGBoost (Extreme Gradient Boosting) is an ensemble learning algorithm that builds multiple decision trees sequentially, improving upon the errors of previous trees. It is highly efficient, scalable, and one of the most widely used algorithms for structured/tabular data in machine learning. XGBoost works by sequentially building decision trees that correct the mistakes of previous trees using gradient boosting. When we upload a dataset, it is first pre-processed and converted into an optimized format (DMatrix). The model then iteratively learns by reducing residual errors and optimizing a loss function using first and second derivatives.
Unlike standard decision trees, XGBoost **does not use Gini Impurity or Information Gain** for splits. Instead, it uses **Gradients and Hessians** from the loss function.

- **Gradient (First Derivative):** Measures how much the loss function changes with respect to the predicted value.

- **Hessian (Second Derivative):** Measures the rate of change of the gradient.

Regularization techniques like L1/L2 prevent overfitting, and tree pruning ensures efficiency. Finally, multiple weak tree predictions are aggregated for final output, which is evaluated using appropriate metrics before deployment.

XGBoost starts by initializing all predictions to the log-odds of the target variable (usually 0 for binary classification, corresponding to 0.5 probability). It then computes gradients (errors) and hessians (second derivatives) of the loss function to measure how much each sample contributes to the error. The model builds a decision tree by finding the best feature split that minimizes loss, using a gain formula based on gradient and hessian values. After splitting, leaf values are calculated to update predictions, which are adjusted using a learning rate (η). These new predictions are converted back to probabilities using the sigmoid function. XGBoost continues adding trees, where each new tree corrects the residuals (errors) of the previous ones, while regularization (L1/L2) and tree pruning prevent overfitting. The final model is an ensemble of boosted decision trees that optimally predicts the target variable.

54) Explain the RAG-powered Chatbot implementation? (Personalized question for my project – General implementation)

To enhance customer support automation, we start by **collecting data from multiple sources**, including PDFs, CSV files, and scraped content from the web. This raw data is then **converted into a structured Q&A format (e.g., JSON or CSV)** using tools like **pandas, spaCy, and regex-based text processing**. The preprocessing phase involves **tokenization, stopword removal, lemmatization, and standardization** to ensure the data is clean, consistent, and ready for further processing.

Next, we build a **vector database** to implement the **RAG (Retrieval-Augmented Generation) framework**. The cleaned Q&A dataset is **transformed into numerical vector representations** using the embedding model **Sentence-BERT (SBERT**. These **vectorized representations are stored in FAISS (Facebook AI Similarity Search)**, an optimized database designed for fast and efficient similarity searches.

When a user submits a query, the input text is **embedded into a vector representation** using the same model used for document embeddings. The vectorized query is then **compared with the stored document vectors** in FAISS using **cosine similarity, L2 distance, or inner product** to **retrieve the most relevant documents**. FAISS performs a **top-k nearest neighbors (KNN) search** to quickly identify the closest matching documents based on the query.

The **retrieved documents, along with the user's query, are then passed as a prompt** to an **LLM (e.g., GPT-4, Llama 2, Claude, or Mistral)**. **Prompt engineering techniques** ensure that the response is structured, compliant, and factually accurate. For instance, we can include system instructions like **"Answer using only the provided document; do not generate additional information."** This prevents **hallucinations and ensures that the response aligns with the retrieved knowledge base.**

The **main advantage of the RAG framework** is that it eliminates the need to **fine-tune or retrain the LLM** whenever new data is added. Instead, we simply **update the vector database with new embeddings**, allowing the chatbot to **stay up-to-date dynamically** without expensive model retraining. This approach significantly **reduces computational costs, improves response accuracy, and ensures real-time updates** to the knowledge base.

By combining **document embeddings, FAISS for retrieval, and LLM-based response generation**, we create an efficient, cost-effective, and scalable AI-driven customer support automation system that **delivers fast, accurate, and contextually relevant responses.**

55) Explain KNN Imputation?

KNN (K-Nearest Neighbors) imputation fills missing values by finding the K most similar rows (neighbors) based on other available features. It computes similarity using distance metrics like Euclidean or Manhattan distance and imputes the missing value by averaging (or taking a weighted mean) of the corresponding feature values from the nearest neighbors. Unlike mean imputation, which uses a fixed average, KNN ensures that the imputed value is contextually similar to the given data, preserving underlying patterns.

One limitation of KNN imputation is its computational cost, especially for large datasets, since it requires calculating distances for each missing value. Additionally, if data is sparse or if missing values are clustered, KNN might not find truly representative neighbors, leading to biased imputations. However, when used correctly, it is highly effective in maintaining data distribution and relationships between features, making it superior to simple imputation techniques.

56) Explain Iterative Imputation?

Iterative imputation treats missing value estimation as a machine learning problem, where each feature with missing values is predicted using regression models based on other available features. The process iteratively selects one feature at a time, treating it as the target variable and using the remaining features as predictors. A model (e.g., linear regression, decision tree, or Bayesian ridge regression) is trained to predict the missing values, which are then updated, and the process repeats until convergence or a stopping criterion is met.

This method is particularly useful when features are highly correlated, as it can infer missing values using patterns from the dataset. However, it assumes that relationships between features are linear or predictable, which may not always hold. Additionally, iterative imputation can introduce bias if the underlying assumptions about feature dependencies are incorrect. Nevertheless, it is a powerful approach for datasets with complex missing data patterns, making it more advanced than KNN for structured, correlated datasets.

57) Explain the use of DenseNet neural network?

✅ DenseNet-121, 169, etc., refer to the total number of layers, chosen based on empirical results to balance accuracy and efficiency.

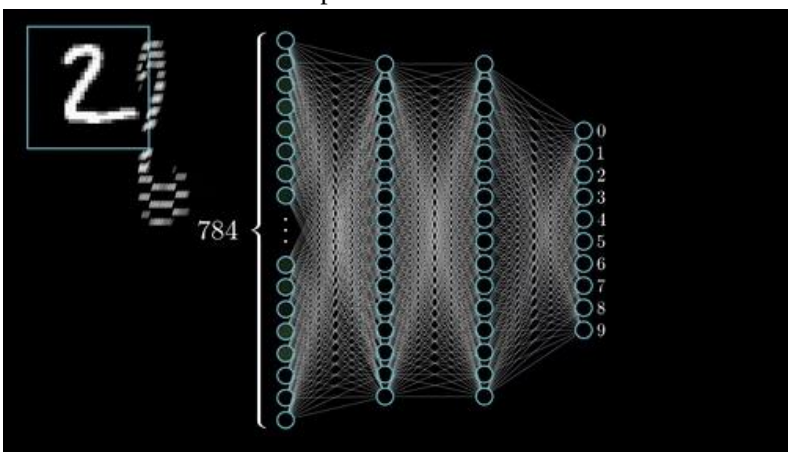
✅ Increasing the number of layers can improve accuracy to an extent, but beyond a point, it leads to overfitting and computational inefficiency.

✅ DenseNet differs from traditional CNNs by using dense connections, reducing redundancy, improves feature reuse, and improving gradient flow, making it more efficient than deep traditional CNNs.

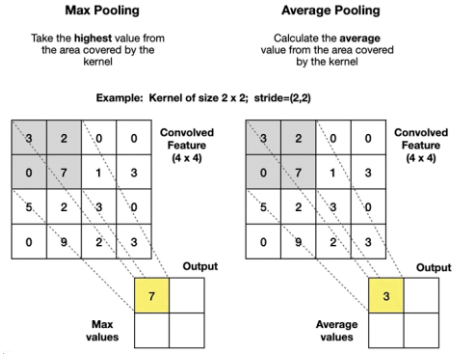
58) Explain the working of CNN in detail from end-to-end?

A Convolutional Neural Network (CNN) processes images through a series of layers designed to detect patterns and features at different levels. It begins with the **convolutional layer**, where small filters slide over the image, detecting edges, textures, and shapes. The result of this operation is a **feature map**, highlighting areas where specific patterns are present. To reduce the computational complexity and retain important features, a **pooling layer** (like Max Pooling) downsamples the feature maps by selecting the most prominent values. Multiple convolution and pooling layers extract deeper, more abstract patterns, allowing the network to recognize complex structures such as eyes, faces, or objects. The extracted features are then passed to **fully connected layers**, which act as a classifier by combining all the learned patterns and mapping them to specific output categories. The

final layer uses an activation function like **softmax** (for multi-class classification) or **sigmoid** (for binary classification) to generate the final probabilities for each class.
A key technique in CNNs is **Global Average Pooling (GAP)**, which replaces dense layers by averaging the feature maps instead of flattening them. This reduces parameters and prevents overfitting while maintaining spatial information. CNNs leverage techniques like **dropout** (to prevent overfitting) and **batch normalization** (to speed up training) for better generalization. The entire training process involves **backpropagation** and **gradient descent**, adjusting filter weights to minimize error. Unlike traditional models, CNNs learn hierarchical features: early layers detect simple patterns, while deeper layers understand high-level structures. This makes CNNs highly effective for image classification, object detection, and computer vision applications.
1 **Convolutional Layer:** This is the core of CNNs, where small filters slide over the input image, computing dot products to create feature maps. These maps highlight patterns like edges, textures, and shapes, allowing the network to understand spatial hierarchies.

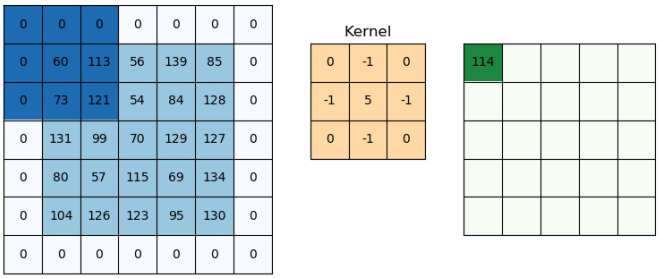


2 **Feature Maps:** These are the output of convolutional layers, representing detected patterns in an image. Each feature map corresponds to a specific filter and encodes unique visual features that contribute to higher-level understanding.
3 **Pooling Layer:** A downsampling operation (e.g., Max Pooling) that reduces the spatial size of feature maps while retaining important information. This helps decrease computational cost, control overfitting, and make the model more robust to small shifts in input data.



4 **Fully Connected Layer (FC):** After extracting hierarchical features, the FC layer flattens and processes them to form the final classification. It connects all neurons, learns complex feature relationships, and outputs predictions using activation functions like softmax or sigmoid.
5 **Global Average Pooling (GAP):** Instead of flattening feature maps into long vectors, GAP averages the values within each feature map, reducing parameters and improving generalization. This helps make CNNs more efficient and less prone to overfitting.

6 **Backpropagation & Optimization:** The CNN is trained using backpropagation and gradient descent, adjusting filter weights to minimize loss. Techniques like dropout and batch normalization further optimize performance and improve model accuracy.
7 **Padding Layer:** Padding in CNNs is the process of adding extra pixels (usually zeros) around an input image to prevent size reduction after convolution. It helps preserve spatial dimensions, ensuring edge features are not lost and allowing the filter to cover all regions equally. Padding also maintains the output size when needed, improving feature extraction and preventing excessive shrinking of feature maps. There are two main types: valid padding (no padding, reduces size) and same padding (adds enough padding to maintain the same size as input). This technique ensures better performance in deep networks by retaining essential spatial information.



Method	Retains
Max Pooling	**Strongest edges and features** (e.g., thick strokes of "8")
Average Pooling	**Overall shape and structure** (e.g., blurry but intact "8")
Global Average Pooling (GAP)	**Feature presence summary** (e.g., "Is there a loop? Yes/No")
Padding	**Preserves full image shape** (e.g., keeps the outer strokes of 8)

59) Explain the working of GAP in detail?

Global Average Pooling (GAP) reduces the spatial dimensions of feature maps by computing the average value of each channel, resulting in a single value per channel. If the input to GAP is an **8×8×256** feature map, it outputs a **1×1×256** vector by averaging each of the 256 feature maps. Unlike fully connected (FC) layers that flatten all spatial features into a large vector, GAP retains **spatially summarized feature strengths**, reducing overfitting and improving efficiency. These 256 values are not probabilities but **high-level feature representations** learned by the CNN. A final dense layer with **sigmoid or softmax activation** converts these values into class probabilities, enabling classification. For example, in a **dog vs. cat classifier**, GAP might produce values such as **Feature 1: 3.2 (Fur texture detected), Feature 2: -1.1 (Ear shape weakly detected), Feature 3: 5.4 (Round face strongly detected), …, Feature 256: 2.8**. These values are then passed to a dense layer, which assigns weights to each feature and computes a final output. If a **sigmoid activation** is used, it may yield a probability of **0.92 (Dog) and 0.08 (Cat)**, enabling classification.

60) Shortened version of CNN explanation for interview?

CNN consists of three major components: convolution, pooling, and classification layers. The convolution layer applies filters that detect features like edges, textures, and shapes. If needed, padding is used to maintain the image size. Pooling layers (Max/Average/GAP) help reduce spatial dimensions while preserving important features. The extracted feature map is then passed to either a Fully Connected (FC) layer or a GAP layer. GAP is more efficient as it averages feature maps into a single vector, reducing overfitting compared to FC layers that use many parameters. Finally, a dense

layer assigns weights, and a softmax or sigmoid activation function classifies the image based on probability values.

61) What is Web Scraping?

Web scraping is the process of automatically extracting data from websites by fetching the HTML content and parsing it to retrieve useful information. This is commonly done using Python libraries like BeautifulSoup and Scrapy.

In NLP and Data Science, high-quality labeled data is crucial. However, not all data is available in structured formats like CSVs or databases. A lot of valuable information exists on websites in unstructured text. Web scraping helps collect this data for various use cases:

✅ **Chatbots & RAG-based LLMs** → Scraping FAQ pages, product details, and customer reviews for knowledge bases.

✅ **Sentiment Analysis** → Extracting tweets, news articles, or customer reviews to analyze public opinion.

✅ **Price Monitoring & Competitor Analysis** → Scraping e-commerce sites for dynamic pricing strategies.

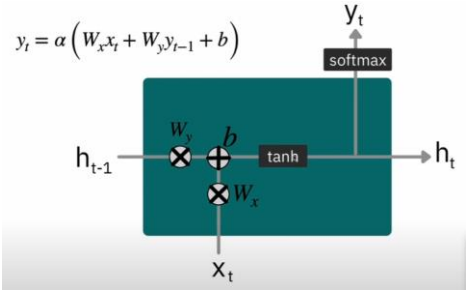
✅ **Stock Market Prediction** → Gathering news, financial reports, and stock trends from financial websites.

✅ **Resume Parsing & Job Market Analysis** → Scraping job postings to analyze hiring trends and salary insights.

Selenium is used for web automation and dynamic web scraping. Unlike BeautifulSoup, it can interact with JavaScript-rendered pages, click buttons, and fill forms. It's commonly used in web scraping, automated testing, and AI/ML data collection tasks. For static websites, BeautifulSoup is more efficient, but for modern web apps, Selenium is essential.

62) What is an RNN? Detailed Explanation?

A Recurrent Neural Network (RNN) is a type of artificial neural network that is designed for sequential data processing. Unlike traditional feedforward neural networks, which process inputs independently, RNNs have a "memory" that retains information from previous inputs, making them effective for tasks where order and context matter (e.g., time series, speech recognition, NLP).



RNNs are useful for sequential tasks like time series forecasting, speech recognition, and sentiment analysis. They are widely used in fraud detection, handwriting recognition, and chatbots. However, for long-sequence tasks like translation and text summarization, Transformers (like BERT, GPT) are preferred due to their better memory handling and parallel processing capabilities.

Recurrent Neural Networks (RNNs) are designed for sequential data processing, making them ideal for tasks like text generation, speech recognition, and time-series forecasting. Unlike traditional neural networks, which process data independently, RNNs remember past inputs using a hidden state that carries information from previous time steps. At each step, the RNN takes an input (e.g., a word),

updates its hidden state based on past information, and passes this state to the next step. This allows it to maintain context and make predictions based on prior knowledge.
During training, RNNs use Backpropagation Through Time (BPTT) to update weights, minimizing errors by adjusting connections. However, standard RNNs suffer from vanishing gradient problems, which makes learning long-term dependencies difficult. To solve this, advanced architectures like LSTM (Long Short-Term Memory) and GRU (Gated Recurrent Unit) introduce gates to regulate information flow, helping the network retain long-term dependencies. RNNs finalize predictions using a Softmax layer, assigning probabilities to possible outputs. The highest probability word is chosen, making the model context-aware.

63) Explain the RAG-powered Chatbot implementation – specific to a real-world project?

Step – 1: Data preprocessing & Cloud storage

First, we extracted raw customer support queries from multiple sources—PDFs, CSVs, and APIs. We structured the data into Q&A pairs using Pandas and NLP preprocessing (tokenization, lemmatization). Once cleaned, we stored it as a JSON file and uploaded it to AWS S3. From there, the chatbot pipeline retrieved the data using a scalable vector database (Pinecone) for AI-powered search and retrieval.

Step – 2: Embedding Generation & Vector Storage

After retrieving the cleaned Q&A data from AWS S3, we convert each question into numerical vector embeddings using OpenAI's text-embedding-ada-002 model. These embeddings capture the semantic meaning of queries, enabling efficient similarity search. The vectorized data is then stored in Pinecone, a cloud-based vector database, allowing fast top-k nearest-neighbor retrieval when users submit queries. This setup significantly improves search speed and accuracy, making chatbot responses more relevant and context-aware.

Step – 3: Retrieval & Response Generation

When a user submits a query, it is converted into an embedding using OpenAI's model and searched against stored embeddings in Pinecone to retrieve the most relevant answers. The top-k matching responses are then formatted into a structured prompt and passed to GPT-4, ensuring that the chatbot generates accurate, context-aware responses based on retrieved knowledge. This approach improves response relevance, reduces hallucination, and enhances user experience.

64) What are the tools and techs that are used in this RAG implementation?

For preprocessing, I used Pandas, spaCy, and PyPDF2 to structure raw customer queries into a Q&A format before storing them in AWS S3. For retrieval, I generated embeddings using OpenAI's text-embedding-ada-002 and stored them in Pinecone, which allowed fast similarity-based searches. When a user submits a query, it's converted into an embedding, matched with stored data in Pinecone, and passed to GPT-4 for generating an accurate, context-aware response. The system was integrated with a FastAPI backend, making it accessible as an API.

65) What is Pinecone?

Pinecone is a fully managed vector database designed for high-speed similarity search. It is commonly used in applications like recommendation systems, AI search, and large-scale retrieval-augmented generation (RAG) for LLMs.
Pinecone is a serverless, real-time vector database that allows you to store, index, and search high-dimensional vector embeddings efficiently. It is used in AI applications where you need to find similar items based on vector representations rather than exact matches. Pinecone is a **fully managed,**

cloud-based vector database, meaning it **does not run on-premise** but is hosted on cloud infrastructure. Pinecone runs on major cloud providers like AWS and GCP, and its databases (indexes) are stored in specific cloud regions based on your selection when creating an index.

66) Did you use GPT-4 for response generation? Why not Llama?

For initial development, we used OpenAI's GPT-4 because it provided out-of-the-box accuracy, reducing the need for immediate fine-tuning. However, for long-term cost savings and customization, we explored self-hosted models like Llama 2. The tradeoff is that Llama 2 requires fine-tuning and GPU infrastructure, whereas GPT-4 delivers instant results via API.

67) Final Workflow Summary (End-to-End)?

1 User submits a query → (Sent to Backend API)

2 Query converted into an embedding → (OpenAI text-embedding-ada-002)

3 Find similar knowledge from database → (Pinecone / FAISS retrieves top matches)

4 LLM (GPT-4/Llama 2) generates response → (Uses retrieved knowledge)

5 Response sent back to user → (Chatbot UI displays response)

When a user submits a query, the backend first converts it into an embedding using OpenAI's text-embedding-ada-002 model. This embedding is then searched against stored knowledge in Pinecone to retrieve the most relevant documents. The retrieved documents, along with the original user query, are passed to GPT-4 (or Llama 2), ensuring that the response is grounded in factual knowledge. Finally, the response is returned to the chatbot UI for the user.

Initially, we collect data from multiple sources like PDFs, CSVs, APIs, JSON files, and databases. Once we have sufficient data, we preprocess it using Pandas, remove stopwords using NLTK, apply lemmatization, and structure it into a Q&A format stored as a JSON file. This cleaned dataset is then uploaded to an AWS S3 bucket to ensure easy accessibility across teams.

"Next, we retrieve the structured data from S3 and generate numerical vector embeddings using OpenAI's text-embedding-ada-002 model. These embeddings capture the semantic meaning of each document. We then store these vectors in Pinecone, a vector database optimized for semantic search. Unlike traditional keyword-based searches, Pinecone allows for similarity-based retrieval, ensuring that contextually relevant documents are retrieved."

"When a user submits a query, it is first converted into an embedding using the same OpenAI model. The backend API then sends this vectorized query to Pinecone, which performs a nearest-neighbor search based on cosine similarity to retrieve the most relevant documents. The top-k matched documents, along with the original user query, are formatted into a structured prompt and sent to GPT-4 (or Llama 2) for response generation. This ensures that the LLM provides a well-structured, compliance-based answer using factual information rather than hallucinated data. The final response is then sent back to the user via the chatbot interface."

"This entire workflow follows the RAG (Retrieval-Augmented Generation) framework, which eliminates the need for costly fine-tuning of the LLM. Instead of retraining the model every time new information is added, we simply update the knowledge base in Pinecone, ensuring that the chatbot always retrieves the latest and most relevant information.

68) What Is Cosine Similarity Threshold?

✅ Cosine Similarity is a measure of how similar two vectors (numerical representations of text) are, based on the angle between them rather than their magnitude.

✅ Threshold is the minimum similarity score required for two documents to be considered a "match."

🔹 **If the threshold is too low** → Irrelevant documents may be retrieved.

🔹 **If the threshold is too high** → The model may return too few or no documents.

✅ Cosine similarity threshold controls retrieval quality.

✅ It is fine-tuned by analyzing user feedback & experimenting with values (e.g., 0.5 vs. 0.7).

✅ In production, we filter out low-relevance documents using this threshold.

69) Explain the different classification metrics in detail?

Scenario: Spam Email Detection

Imagine you work for an email provider like Gmail, and you've built a classifier to detect spam emails (positive class) vs. non-spam (ham) emails (negative class).

Your classifier gives four possible outcomes:

1. **True Positive (TP):**
   - The classifier correctly identifies a spam email as spam.
   - ✅ Example: You receive a "Win a free iPhone!" email, and it gets marked as spam.
2. **False Positive (FP):** (Type I Error)
   - The classifier incorrectly marks a ham email as spam.
   - ❌ Example: Your friend sends you an email about weekend plans, but the classifier mistakenly puts it in the spam folder.
3. **True Negative (TN):**
   - The classifier correctly identifies a ham email as ham.
   - ✅ Example: You get a work email, and it stays in your inbox.
4. **False Negative (FN):** (Type II Error)
   - The classifier fails to identify spam, marking it as ham.
   - ❌ Example: A phishing email from a fake bank slip into your inbox instead of going to spam.

Now, using these, let's understand the key classification metrics.

1. Accuracy:

👉 Measures how many predictions were correct overall.

$$\text{Accuracy} = TP+TN/TP+TN+FP+FN$$

Issue: If spam emails are rare (say 95% of emails are ham), a dumb model that predicts everything as ham will still have 95% accuracy, but it's useless!

---

2. Precision (Positive Predictive Value):

👉 Out of all the emails predicted as spam, how many were actually spam?

$$\text{Precision} = TP/TP+FP$$

Example:

- If precision = **80%**, it means 80% of emails classified as spam were truly spam, but 20% were mistakenly marked spam (false positives).
- High precision means fewer important emails wrongly flagged as spam.

When to prioritize precision?

- If False Positives are costly.
- Example: A fraud detection system that wrongly blocks legit transactions.

---

3. Recall (Sensitivity or True Positive Rate):

👉 Out of all actual spam emails, how many were correctly classified as spam?

$$\text{Recall} = TP/TP+FN$$

Example:

- If recall = **90%**, it means 90% of spam emails were correctly marked as spam, but 10% were missed (false negatives).
- High recall means almost all spam emails are caught.

When to prioritize recall?

- If False Negatives are dangerous.
- Example: A cancer detection system must have high recall, as missing a cancer case is life-threatening.

---

4. F1-Score: (Harmonic mean of Precision & Recall)

👉 A balance between precision and recall.

$$F1 = 2 \times Precision \times Recall / Precision + Recall$$

Why do we need F1-score?

- If precision is high but recall is low, or vice versa, F1-score balances both.
- Example: If precision is 90% but recall is 50%, the model is rejecting too many spam emails as non-spam.

NLP TECHNIQUES

70) What is Sentiment Analysis?

Sentiment Analysis (also known as opinion mining) is a Natural Language Processing (NLP) technique used to determine the emotional tone behind a piece of text. It helps in understanding whether a given text expresses a positive, negative, or neutral sentiment.

Sentiment Analysis Pipeline:

- **Preprocessing** (cleaning text)
- **Feature Extraction** (numerical representation)
- **Sentiment Classification** (lexicon-based, ML, deep learning)
- **Prediction Output** (positive, negative, neutral)

1.1 Text Preprocessing

Before analyzing sentiment, we need to clean and process the text. This includes:

- **Tokenization**: Splitting text into words or phrases.
- **Lowercasing**: Converting text to lowercase for uniformity.
- **Stopword Removal**: Removing words like "is," "the," "and," etc., which do not contribute to sentiment.
- **Lemmatization/Stemming**: Converting words to their root forms (e.g., "running" → "run").
- **Punctuation and Special Character Removal**: Cleaning unnecessary symbols.

📌 *Example before & after preprocessing:*

💬 Original: "The movie was AMAZING!!! But the ending was terrible 😠"

✅ Cleaned: ["movie", "amazing", "ending", "terrible"]

1.2 Feature Extraction (Vectorization)

After preprocessing, the text needs to be converted into a numerical format that a machine can understand. Common Feature Extraction Methods:

1. **Bag of Words (BoW)** – Represents text as a set of word occurrences.

2. **TF-IDF (Term Frequency - Inverse Document Frequency)** – Weighs words based on importance in the document.
3. **Word Embeddings** (like Word2Vec, GloVe, BERT) – Capture word relationships in a meaningful way.

1.3 Sentiment Classification Approaches

Once we have features, we use algorithms to classify sentiment:

Lexicon-Based Approach

- Uses a predefined dictionary of words with sentiment scores.
- Example: "good" has a +1 score, "bad" has a -1 score.
- Summing up scores gives overall sentiment.

Machine Learning Approach

- Uses models like:
  - ❖ **Naive Bayes**
  - ❖ **Support Vector Machines (SVM)**
  - ❖ **Random Forest**
  - ❖ **Logistic Regression**

Deep Learning Approach

- Uses advanced models like:
  - ❖ **Recurrent Neural Networks (RNN)**
  - ❖ **Long Short-Term Memory (LSTM)**
  - ❖ **Transformers (BERT, GPT, etc.)**

71) What are the different types of Sentiment Analysis?

1. **Binary Sentiment Analysis** – Classifies text as positive or negative.
2. **Multiclass Sentiment Analysis** – Includes positive, neutral, and negative.
3. **Fine-Grained Sentiment Analysis** – Uses detailed ratings like very positive, positive, neutral, negative, very negative.
4. **Aspect-Based Sentiment Analysis** – Determines sentiment towards specific aspects of a product (e.g., “Camera quality is great, but battery life is poor”).
5. **Emotion Detection** – Identifies emotions like happiness, sadness, anger, surprise.

72) What are some popular sentiment lexicons used?

1. **VADER (Valence Aware Dictionary and sEntiment Reasoner)** – Used for social media sentiment analysis.
2. **SentiWordNet** – Assigns sentiment scores to words.
3. **AFINN** – Assigns words integer sentiment scores (-5 to +5).

73) Explain Rule-Based Sentiment Analysis?

Rule-Based Sentiment Analysis is an approach that does not require training a machine learning model but instead relies on predefined rules, lexicons, and heuristics to determine sentiment.

📌 What is Rule-Based Sentiment Analysis?

It assigns sentiment scores based on word matching, predefined dictionaries, and grammatical rules.

Example:

- "This movie is amazing!" → Positive (because "amazing" is a positive word)
- "This is a terrible experience!" → Negative (because "terrible" is a negative word)

VADER:

Vader (Valence Aware Dictionary and sEntiment Reasoner) is a **rule-based sentiment analysis tool** that assigns sentiment scores to text using a **predefined lexicon** of words with positive, negative, and neutral sentiment values. It enhances accuracy by handling **intensifiers** (e.g., "very" increases positivity), **negations** (e.g., "not good" reduces positivity), **punctuation & capitalization** (e.g., "AWESOME!!!" amplifies sentiment), **emojis & special characters**, and **context-based adjustments** (e.g., "but" shifts sentiment emphasis). VADER computes four sentiment scores: **positive, negative, neutral, and compound**, where the **compound score** (ranging from -1 to +1) determines overall sentiment polarity. It is highly effective for **short, informal text** like tweets and reviews without requiring training data. VADER is a hybrid approach that combines both lexicon-based and rule-based techniques for better accuracy.

74) Explain Naïve Bayes Algorithm in detail and how it is used in sentiment analysis?

Naïve Bayes is a probabilistic machine learning algorithm based on **Bayes' Theorem**, which calculates the probability of a class given a set of input features. It assumes that all features (words in text classification) are **independent** given the class, making it a **naïve** assumption. The algorithm computes the **posterior probability** for each class using prior probabilities and likelihoods, and assigns the most probable class. Despite its simplicity, it is highly effective for text classification tasks, especially when dealing with high-dimensional data. To handle unseen words, **Laplace smoothing** is applied, ensuring no probability is ever zero.

Naïve Bayes for Sentiment Analysis

In sentiment analysis, Naïve Bayes classifies text as **positive, negative, or neutral** by analyzing word frequencies in labeled training data. First, it calculates the **prior probability** of each sentiment class based on historical data. Then, it computes the **likelihood probability** of words appearing in each sentiment class using word counts. When a new sentence is given, the model multiplies the likelihoods of its words with the prior probability to compute the **posterior probability** for each sentiment. The class with the highest probability is chosen as the sentiment. This makes Naïve Bayes highly efficient for classifying opinions, reviews, and social media comments.

1. Compute Prior Probability for each class.
2. Compute Likelihood Probability for each word in the given sentence.
3. Multiply all likelihoods together with the prior probability.
4. Compare the computed probabilities for each class.
5. Assign the class with the highest posterior probability.

Naïve Bayes uses prior probabilities to determine posterior probabilities while applying **Laplace smoothing** to handle unseen words.

75) What is Named Entity Recognition (NER)?

NER is a Natural Language Processing (NLP) technique that identifies and classifies key information (entities) in text into predefined categories, such as:

- Person → "Elon Musk"
- Organization → "Google"
- Location → "New York"
- Date & Time → "March 14, 2025"
- Monetary Values → "$100 million"

NER is typically performed using Machine Learning (ML) or Deep Learning (DL) models, which process text and classify words into entity categories.

🔹 How Does NER Work?

1. **Tokenization** → Split text into words.

2. **Entity Recognition** → Detect words/phrases that represent an entity.
3. **Entity Classification** → Assign each entity to a category.

spaCy is an industrial-strength NLP library designed for fast, efficient, and accurate natural language processing. It provides pre-trained models that perform tasks like tokenization, part-of-speech tagging, dependency parsing, and named entity recognition (NER). Unlike traditional rule-based NLP methods, spaCy uses statistical machine learning models trained on large datasets, making it robust for real-world applications. Its pipeline-based approach processes text through different components sequentially, extracting meaningful insights efficiently. Additionally, spaCy supports deep learning-based NER by integrating transformer models (like BERT), making it suitable for modern AI applications. Its speed, scalability, and ease of use make it a preferred choice in AI-driven applications, chatbots, and text analytics.

76) Explain the types of regularization techniques?

Regularization is a technique used in machine learning to reduce overfitting and improve a model's generalization to new data. It does this by adding a penalty to the model's loss function, discouraging it from learning overly complex patterns that may not generalize well to unseen data.

In simple terms, regularization prevents the model from memorizing training data and ensures it learns meaningful patterns instead. A machine learning model can face two main problems:

🔷 Underfitting → The model is too simple and cannot capture the data's underlying pattern.

🔷 Overfitting → The model becomes too complex and captures even noise in the training data, performing poorly on unseen data.

Regularization helps prevent overfitting by penalizing overly complex models, forcing them to focus on important features rather than noise.

🔷 L1 Regularization (Lasso Regression)

✓ Adds a penalty proportional to the absolute values of the weights to the loss function.

✓ Encourages sparsity, meaning some feature weights become zero, effectively removing irrelevant features.

✓ Useful for feature selection, as it helps identify the most important predictors.

✓ Used in Lasso Regression, where the loss function is modified as:

$$\textbf{Loss} = \textbf{MSE} + \lambda \sum |w_i|$$

📌 Impact: Leads to simpler models by eliminating unnecessary features.

🔷 L2 Regularization (Ridge Regression)

✓ Adds a penalty proportional to the square of the weights to the loss function.

✓ Prevents large weight values, distributing importance across all features.

✓ Used in Ridge Regression, where the loss function is modified as:

$$\textbf{Loss} = \textbf{MSE} + \lambda \sum w_i^2$$

📌 Impact: Reduces model complexity but retains all features.

🔷 Elastic Net Regularization

✓ Combination of L1 (Lasso) and L2 (Ridge) regularization.

✓ Helps when L1 alone removes too many features, while L2 alone keeps too many.

✓ Works well when features are correlated.

📌 Impact**:** Balances feature selection and weight reduction.

77) Explain the Vanishing and the Exploding Gradient problems in detail?

To understand these problems, we first need to understand concepts like Gradient, Slope, and how back propagation works because those are the deciding factors that largely contributes to these problems.

1 What is a Slope?

A slope tells us how steep a line is. Mathematically, the slope is:

slope=change in y/ change in x

This means how much y (output) changes when we change x (input).

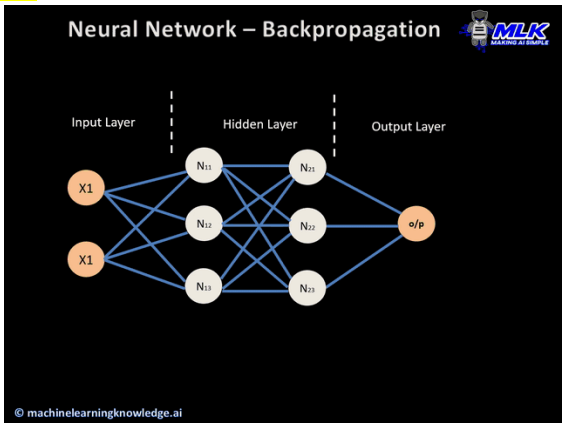
2 What is a Gradient?

Now, in machine learning, we don't deal with just one input variable xxx, but multiple weights w1,w2,... The gradient tells us how much the loss function changes when we slightly adjust each weight. The gradient is just a multi-variable version of the slope:

Gradient = $\partial L/\partial w$

This means how much the loss function L changes when we change weight w. The gradient is the multi-variable version of a slope. It tells us how much the loss function changes when we adjust model weights. Just like a slope tells you how steep a hill is, the gradient tells us the steepness of the loss function. In gradient descent, we use this information to update weights and minimize loss.

3 What is Back Propagation?



Backpropagation is the algorithm used to train neural networks. It works by calculating how much each weight contributes to the loss and updating weights using gradient descent. It has two main steps: (1) the forward pass, where predictions are made, and (2) the backward pass, where errors are propagated backward using the chain rule. This helps adjust weights efficiently to minimize loss and improve the model's accuracy.

4 What is the Chain Rule?

The chain rule is a fundamental rule in calculus that helps us find the derivative of a function that is composed of other functions. **In simple words**: The chain rule **breaks down** a complex derivative into smaller, easier derivatives that are multiplied together.

In deep learning, we don't just have a single function to differentiate. Instead, we have layers upon layers of functions, each transforming inputs into outputs.

- We need to compute the gradient from the output layer back to the input layer (backpropagation).
- Since each layer's output is the input to the next layer, we use the chain rule to systematically compute derivatives layer by layer.

Explain how it works in backpropagation:

Since the output depends on multiple intermediate computations, we use the chain rule to compute gradients layer by layer. This allows us to determine how much each weight contributes to the final

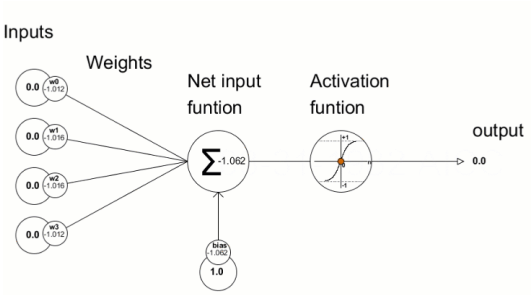
error, helping us update them effectively using gradient descent. For example, if we have a neural network with a loss function LLL, the chain rule helps us compute:

$$\mathbf{dL/dW = dL/da \cdot da/dz \cdot dz/dW}$$

This allows us to adjust weights efficiently to minimize the error.

- ✓ **L (Loss Function):** This is the final error of the neural network. It tells us how far our prediction is from the actual value.
- ✓ **z (Linear Transformation):** This is the weighted sum before applying an activation function.
  - z = Wx + b (for one layer)
- ✓ **a (Activation Output):** This is the result after applying the activation function.
  - a = σ(z), where σ is an activation function (like ReLU, sigmoid, etc.).
- ✓ **W (Weights):** These are the learnable parameters that the network adjusts to make better predictions.

5 What does an Activation Function actually do?



At a practical level, an activation function transforms the output of a neuron before passing it to the next layer.

✅ Without an activation function → Each layer would just perform linear transformations, meaning the entire network would be a big linear model (like a simple regression).

✅ With an activation function → It adds non-linearity, meaning the network can learn complex patterns like curves, edges, and decision boundaries.

Think of it like this:

👉 Without non-linearity = A straight line trying to separate classes in a dataset (very limited).

👉 With non-linearity = A curved decision boundary that can separate complex data (powerful).

Let's go beyond theory and see what "introducing non-linearity" actually means in numbers.

Consider this simple neural network layer:

$$\mathbf{Output = W \cdot X + B}$$

If we don't use an activation function, this is just a linear equation (like a line in 2D, or a plane in higher dimensions). No matter how many layers you add, it's still a linear model!

Example 1: Without Activation Function (Linear Model)

If you input two numbers (2, 3) into a model with weight = 2 and bias = 1:

$$\mathbf{Output = (2\times2) + (2\times3) + 1 = 11}$$

If you add another layer:

$$\mathbf{New\ Output = (2\times11) + 1 = 23}$$

👉 This is still a straight-line transformation—nothing complex is happening! The model is stuck in linearity.

Example 2: With Activation Function (Introducing Non-Linearity)

Now, let's introduce ReLU (Rectified Linear Unit):

$$\mathbf{ReLU(x)=max(0,x)}$$

Applying ReLU after the first layer:

ReLU(11)=max(0,11)=11

Now, in the second layer:

New Output=(2×11)+1=23

Now, if the input was negative:

ReLU(−5)=max(0,−5)=0

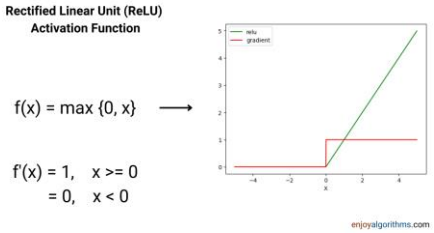
That's non-linearity in action—the model doesn't behave like a straight line anymore. It can introduce complex decision boundaries and help the network learn more powerful patterns.

If you use activation functions, your network can learn:

- Edges in images (important for CNNs)
- Sequential dependencies (important for RNNs)
- Complex decision boundaries (important for classification problems)

This is why activation functions are essential in deep learning. They enable the network to learn complex patterns instead of just drawing straight lines!

6 Explain ReLU Activation Function in detail?



ReLU is a widely used activation function that helps deep networks learn complex patterns efficiently. It keeps positive values unchanged and sets negative values to zero, introducing non-linearity. Compared to Sigmoid and Tanh, ReLU avoids the vanishing gradient problem, speeds up training, and is computationally efficient. However, it has the 'Dying ReLU' problem, which can be fixed using Leaky ReLU.

Consider this simple neural network layer:

Output = W · X + B

If we don't use an activation function, this is just a linear equation (like a line in 2D, or a plane in higher dimensions). No matter how many layers you add, it's still a linear model!

Now, if we apply ReLU to the outputs of this layer, something interesting happens:

- Positive values remain unchanged.
- Negative values become zero.

This breaks the linearity because the model now treats positive and negative values differently, allowing it to learn more complex patterns.

💡 Think of ReLU like a "gate" that selectively allows values to pass while blocking others. This selective behavior makes deep neural networks capable of modeling complex, non-linear relationships. Before ReLU, activation functions like **Sigmoid and Tanh** were widely used, but they had **big problems**:

🔘 Problem 1: Vanishing Gradient Issue (Sigmoid & Tanh)

- In deep networks, small gradients stop weight updates, making lower layers learn very slowly.
- Sigmoid and Tanh squish values between -1 and 1, leading to tiny gradients.
- ReLU doesn't squash values, so the gradients remain larger, allowing deep networks to learn faster.

🔘 Problem 2: Computational Inefficiency

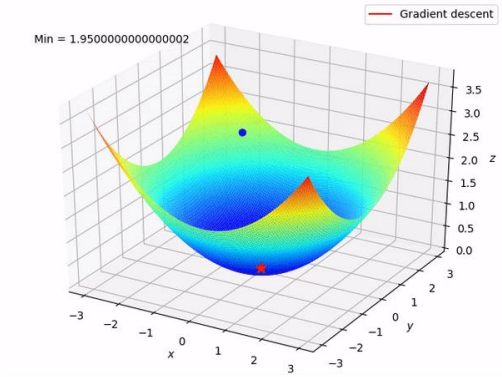
- Sigmoid and Tanh involve exponentials (slow to compute).
- ReLU is just a simple "if-else" check, making it much faster.

✅ Faster Training → Since ReLU keeps positive gradients large, networks converge faster compared to Sigmoid/Tanh.

✅ Better Performance → ReLU avoids the saturation problem of Sigmoid/Tanh, making deep networks work better.

✅ Efficient Computation → Simple mathematical operation means it runs much faster on GPUs.

7 The problems in Gradient Descent?



Gradient descent faces three main challenges: (1) Vanishing gradients, where gradients become too small in deep networks, making training slow. We fix this using ReLU and Batch Normalization. (2) Exploding gradients, where gradients become too large, causing instability. We solve this using gradient clipping and better weight initialization. (3) Slow convergence and local minima, which can trap the model in poor solutions. Using optimizers like Adam and momentum-based methods helps in faster and better convergence.

Update each weight using gradient descent:

$$w_{new} = w_{old} - \alpha \cdot \partial L / \partial w_{old}$$

where α (learning rate) controls step size. This efficiently distributes error across all layers, ensures each neuron learns properly, allows deep networks to train faster.

Vanishing Gradient Problem

The vanishing gradient problem occurs when gradients become too small as they are backpropagated through the layers of a deep neural network. This causes earlier layers (closer to the input) to learn very slowly or stop learning altogether. The main reason behind this issue is the Sigmoid and Tanh activation functions, which squash values into a small range. Their derivatives are very small when inputs are at the extremes, making gradients shrink exponentially during backpropagation. As a result, weight updates become almost negligible, and the network fails to learn effectively. To solve this, we use activation functions like ReLU (Rectified Linear Unit), which does not squash values and allows gradients to flow better. Batch Normalization also helps by normalizing activations and maintaining stable gradients. Additionally, Residual Networks (ResNets) introduce skip connections that allow gradients to bypass multiple layers, preventing excessive shrinkage.

Exploding Gradient Problem

The exploding gradient problem occurs when gradients become too large, leading to unstable weight updates. Instead of slowly adjusting weights to reach an optimal solution, the weight values grow exponentially, causing erratic training behavior and making the model unable to converge. This happens when the weight matrices have large values, and multiplication during backpropagation keeps increasing the gradient magnitude. It is common in deep networks with many layers or when using a very high learning rate. To prevent this, we use Gradient Clipping, which limits the maximum

value of gradients during backpropagation. Using smaller learning rates and better weight initialization techniques like Xavier or He initialization can also help maintain stable gradients.

Slow Convergence & Getting Stuck in Local Minima

Sometimes, gradient descent takes too long to converge, or it gets stuck in a bad solution called a local minimum instead of finding the best possible loss (global minimum). This happens when the loss surface has many bumps (local minima and saddle points), especially in complex models. If the learning rate is too small, gradient descent makes very slow progress toward the minimum. If it's too high, the model might overshoot and never settle into an optimal solution. To solve this, we use momentum-based optimizers like Adam, RMSProp, or Momentum SGD, which help avoid bad local minima by smoothing out updates. Adaptive Learning Rate techniques dynamically adjust the step size for faster and more efficient convergence. Running the training multiple times with different initializations can also help avoid bad local minima.

8 Ways to mitigate these problems?

Vanishing/exploding gradients happen during backpropagation in gradient-based learning algorithms. They occur mainly in deep networks when gradients become too small (vanish) or too large (explode). Adam optimizer helps, but it does not completely eliminate these issues.

Additional techniques like **better weight initialization, ReLU activations, batch normalization, and gradient clipping** are used to mitigate these problems.