# Retrieval Augmented LMs

CSE 5525: Foundations of Speech and Natural Language Processing

https://shocheen.github.io/courses/cse-5525-spring-2026

THE OHIO STATE UNIVERSITY

Slide Credits: Graham Neubig, Maarten Sap, Ana Marasović

# Logistics

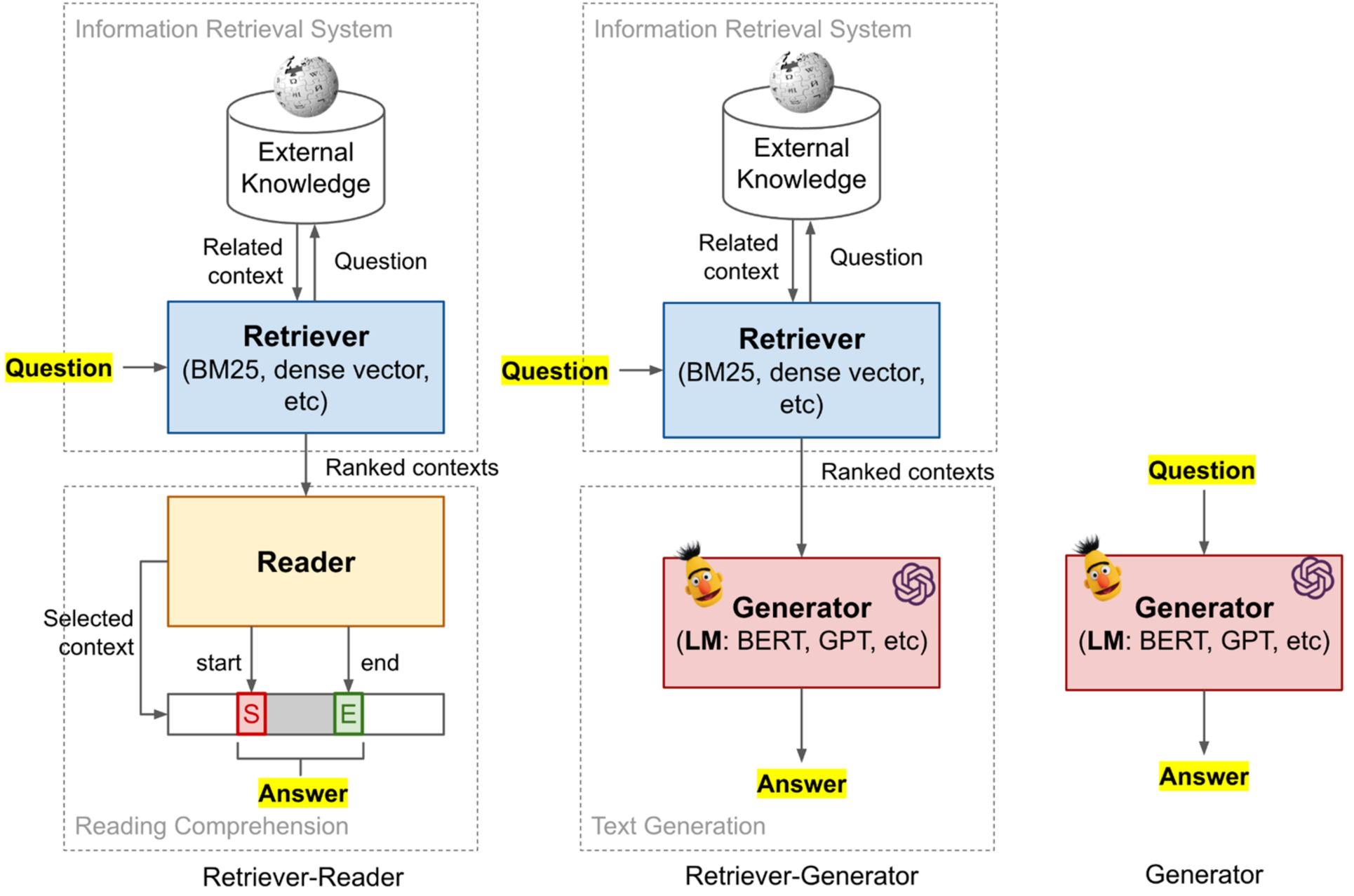
- Project presentations schedule is posted on teams/canvas.

- Final report gradescope will be posted soon. Deadline is April 30. Up to 8 pages. Other details will be posted soon.

- Guest lecture on Friday. Topic: LM Agents.

# Recap: Question Answering / Retrieval

- QA Landscape. Two distinct QA use-cases
  - Evaluate models in their ability to retrain / recall knowledge.
    - Typically evaluated using multi-choice datasets or Cloze style tasks (fill in the blanks). Very common during pretraining to observe how the models are improving.
  - Information seeking – actual users of the language models want to find out information.
    - Certain information should be / can be memorized by models – especially knowledge that does not change with time.
    - If the knowledge changes with time, cannot rely on memory. That's where we want to rely on finding (aka **RETRIEVING**) that information on the internet before answering the user's question.




Source: https://lilianweng.github.io/posts/2020-10-29-odqa/

# **Goal:** Dive into one NLP application: Question Answering

- Retrieval
- Answer Extractor
- Retrieval Augmented Generation (RAG)
- RAG: Overview of Retriever-Generator training options

# Retrieval

- Input: Query Q, a set of documents {D1, D2, …., D_N }
- Goal: Given a query Q, find relevant documents that could answer the query.
  - Ideally: return a ranked list of documents by relevance.
- How would you design a retriever?
  - Need to define a relevance function: r(Q, D), which tell us how relevant is document D to query Q.
  - Compute r(Q, D) for all documents D1, D2, ….
    - Reverse sort and return top-K documents.

# How to design this relevance function

- A simple (but a very effective) set of methods:
  - String overlap / matching

- More complex, but usually better: learn the relevance function.
  - Can you think of other examples in this course where we built something similar?
  - Keywords: similarity, scoring

# String matching based retrieval

- Query Q: [t1, t2, … tL]
- Document D:
- Term frequency or TF(ti, D): how many times does the token ti appears in document D.
- A very simple relevance function, r(Q, D) = sum(TF(ti, D) for ti in Q).
  - Issues?

# String matching based retrieval

- Term frequency alone can inflate the relevance scores if too many “common” words are present in the query – like the, of etc.
  - Because they appear in nearly every document.

- Solution: Inverse document frequency (IDF)
  - Document Frequency(qi, D): How many documents does the token “qi” appear in.
  - IDF: Inverse of document frequency: N/DF(qi, D). N is the total number of documents in the collection.
    - Typically used as log (N / DF (qi, D)).

# String matching: TF-IDF

```
t = token
d = document (movie review)
term_frequency(t,d) = number of times t occurs in d
document_frequency(t) = # documents t occurs in
N = number of documents
inverse_document_frequency(t) = N / document_frequency(t)
tf-idf(t,d) = tf(t,d) x inverse_document_frequency(t)
score(query,d) = sum([tf-idf(t,d) for t in query])
```

Return documents that are the most similar to the query

# TF-IDF ⇒ BM25 [Robertson et al., 1995]

**BM25** is really just a more refined version of TF–IDF with **two additional hyperparameters** [Kamphuis et al. (2020)]

k, a knob that adjust the balance between term frequency and IDF,

- k→0, term frequency has almost no effect, IDF-driven
- E.g., keywords in short texts like tweets/titles, repetition stylistic, special domains

b, which controls the importance of document length normalization

$$\overbrace{\sum_{t \in q} \log\left(\frac{N}{df_t}\right)}^{\text{IDF}} \overbrace{\frac{tf_{t,d}}{k\left(1 - b + b\left(\frac{|d|}{|d_{\text{avg}}|}\right)\right) + tf_{t,d}}}^{\text{weighted tf}}$$

[Equation from: Jurafsky & Martin]

# Inverted Index

We need to efficiently find documents that contain tokens in the question/query

The basic search problem in IR is thus to find all documents that contain a term

The data structure for this task is the **inverted index**, which we use for making this search efficient, and also conveniently storing useful information like the document frequency and the count of each term in each document

[Jurafsky & Martin]

# Inverted Index (cont.)

An **inverted index**, given a query term, gives a list of documents that contain the term

It consists of two parts:

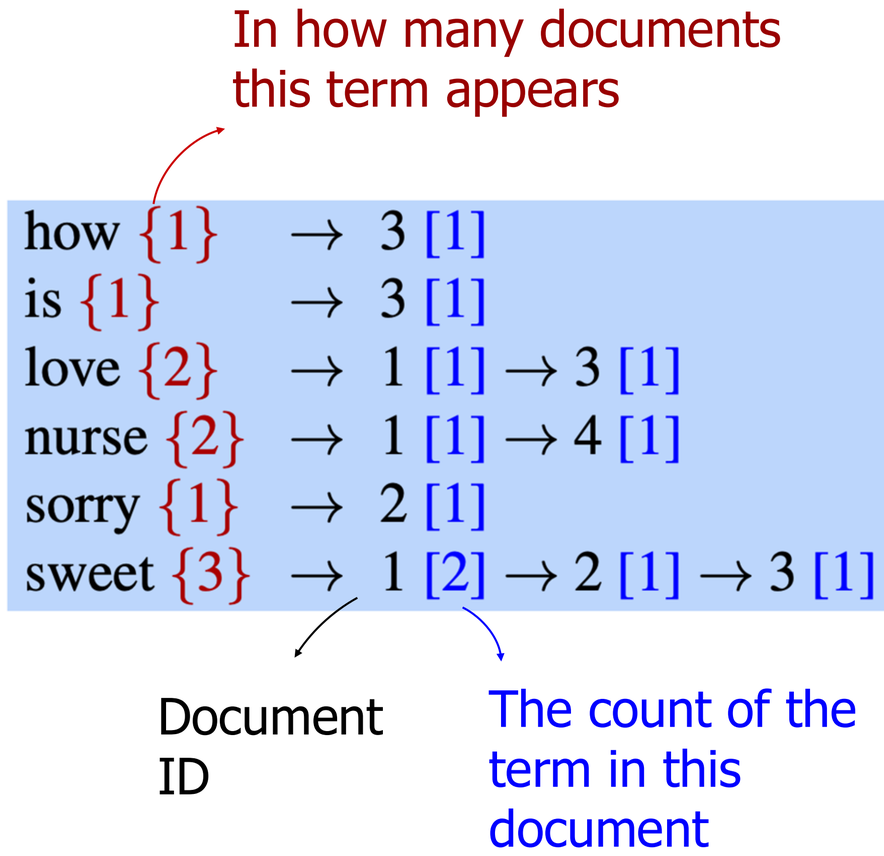
1. Dictionary
2. Postings

**Dictionary** is a list of terms (designed to be efficiently accessed), each pointing to a postings list for the term

- The dictionary can also store the document frequency for each term

A **postings list** is the list of document IDs associated with each term

- It can also contain information like the term frequency or even the exact positions of terms in the document




[Jurafsky & Martin]

# Why could this classic IR approach to retrieval be limited?

tf-idf/BM25 algorithms work only if there is exact overlap of tokens between the question/query and document

What if a question contains a lot of synonyms of tokens in a relevant document?

**Vocabulary mismatch problem:** The user posing a query (or asking a question) needs

to guess exactly what words the writer of the answer might have used

- Instead of (sparse) word-count vectors, using (dense) embeddings

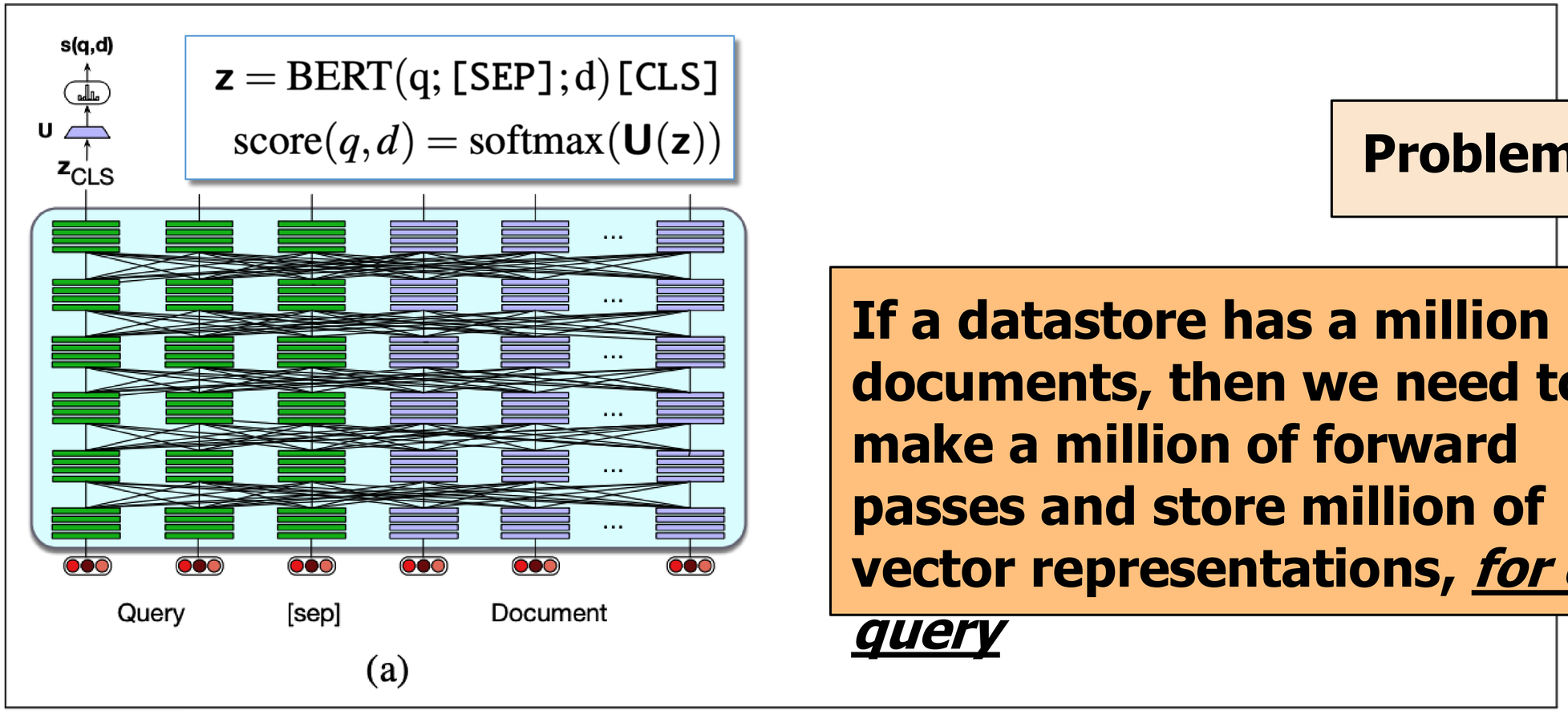
# Dense Retrieval

- Learn a similarity function between query and document S(q, D) – parameterized by a neural network

- Given a new query, compute similarity which document in the collection and return a ranked list (or rather top-k documents in the ranked list)

Figure: [Jurafsky and Martin]

# Two ways to dense retrieval




**Problem?**

**If a datastore has a million documents, then we need to make a million of forward passes and store million of vector representations, *for each query***

**Figure 14.7** Two ways to do dense retrieval, illustrated by using lines between layers to schematically represent self-attention: (a) Use a single encoder to jointly encode query and document and finetune to produce a relevance score with a linear layer over the CLS token. This is too compute-expensive to use except in rescoring (b) Use separate encoders for query and document, and use the dot product between CLS token outputs for the query and document as the score. This is less compute-expensive, but not as accurate.

# Two ways to dense retrieva

$$\mathbf{z}_q = \text{BERT}_Q(\text{q})[\text{CLS}]$$
$$\mathbf{z}_d = \text{BERT}_D(\text{d})[\text{CLS}]$$
$$\text{score}(q,d) = \mathbf{z}_q \cdot \mathbf{z}_d$$

Figure: [Jurafsky and Martin]

[**Dense Passage Retrieval** or **DPR**; Karpukhin et al., 2020]

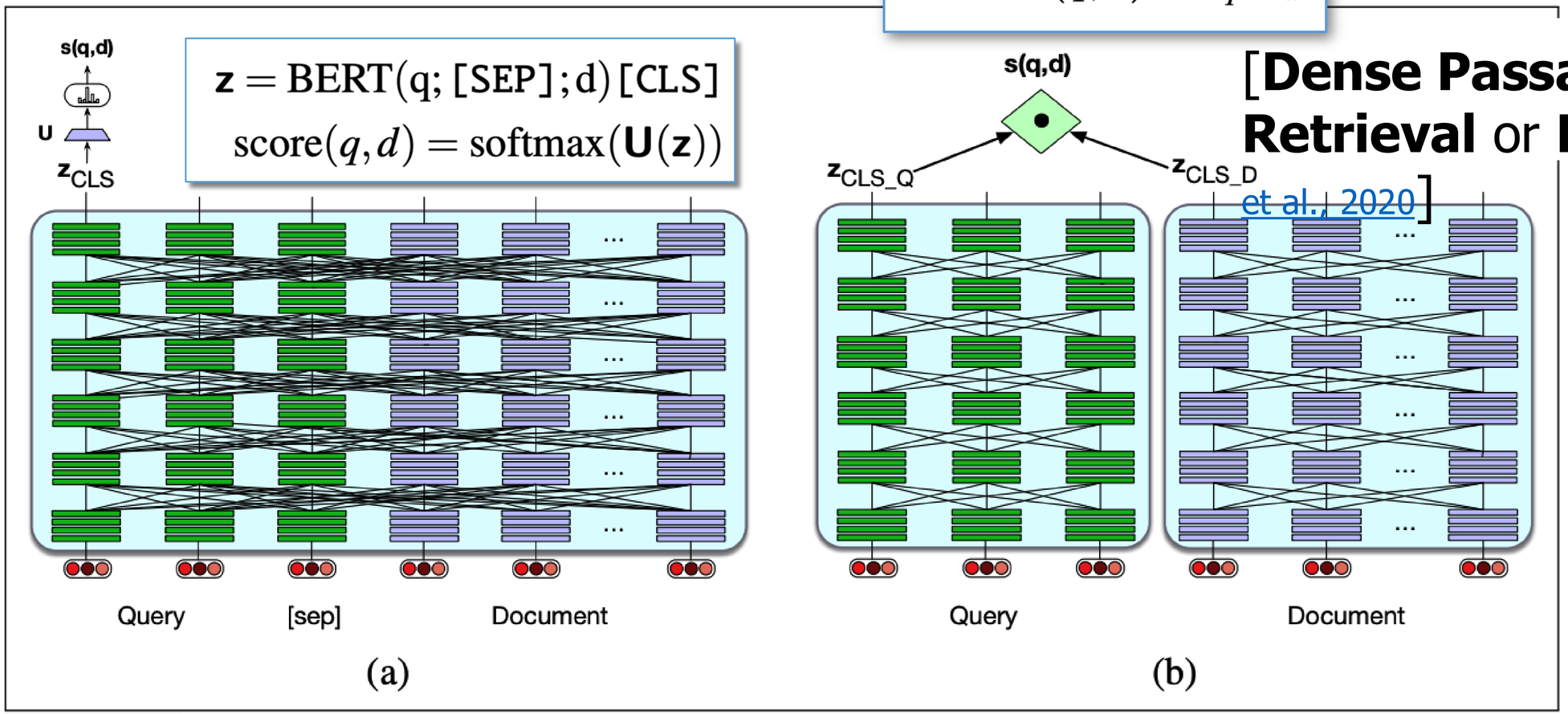



**Figure 14.7** Two ways to do dense retrieval, illustrated by using lines between layers to schematically represent self-attention: (a) Use a single encoder to jointly encode query and document and finetune to produce a relevance score with a linear layer over the CLS token. This is too compute-expensive to use except in rescoring (b) Use separate encoders for query and document, and use the dot product between CLS token outputs for the query and document as the score. This is less compute-expensive, but not as accurate.

# Another way to the second approach – ColBERT

[Khattab et al., 2021]

**Problem:** DPR representations are relatively coarse *They encode each passage into a single high-dimensional vector & estimate relevance via one dot-product*

$$\text{score}(q,d) = \sum_{i=1}^{N} \max_{j=1}^{m} \mathbf{E}_{q_i} \cdot \mathbf{E}_{d_j}$$

BERT output vectors rescaled to unit length

Essentially, for each token in q, ColBERT finds the most contextually similar token in d, & then sums up these similarities

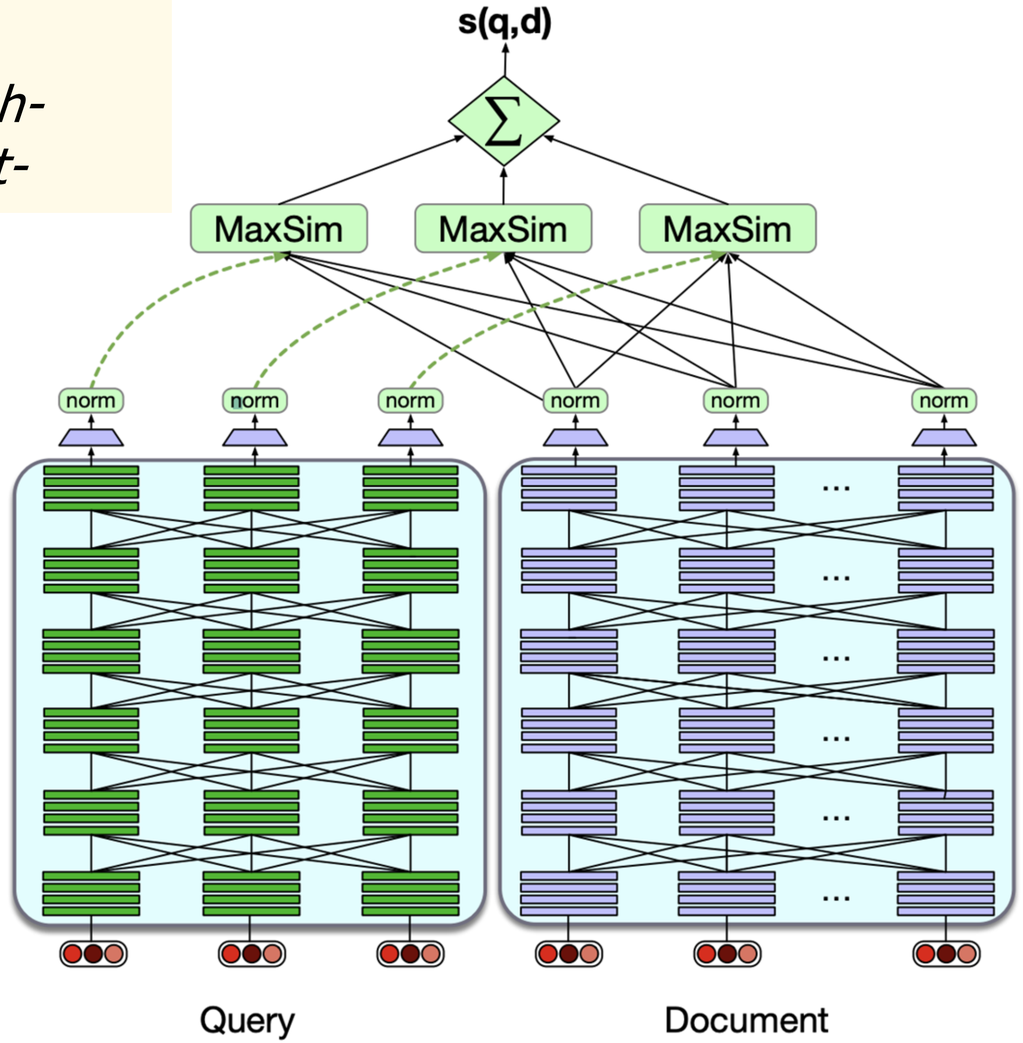



Figure: [Jurafsky and Martin]

# Training dense retrievals - Overview

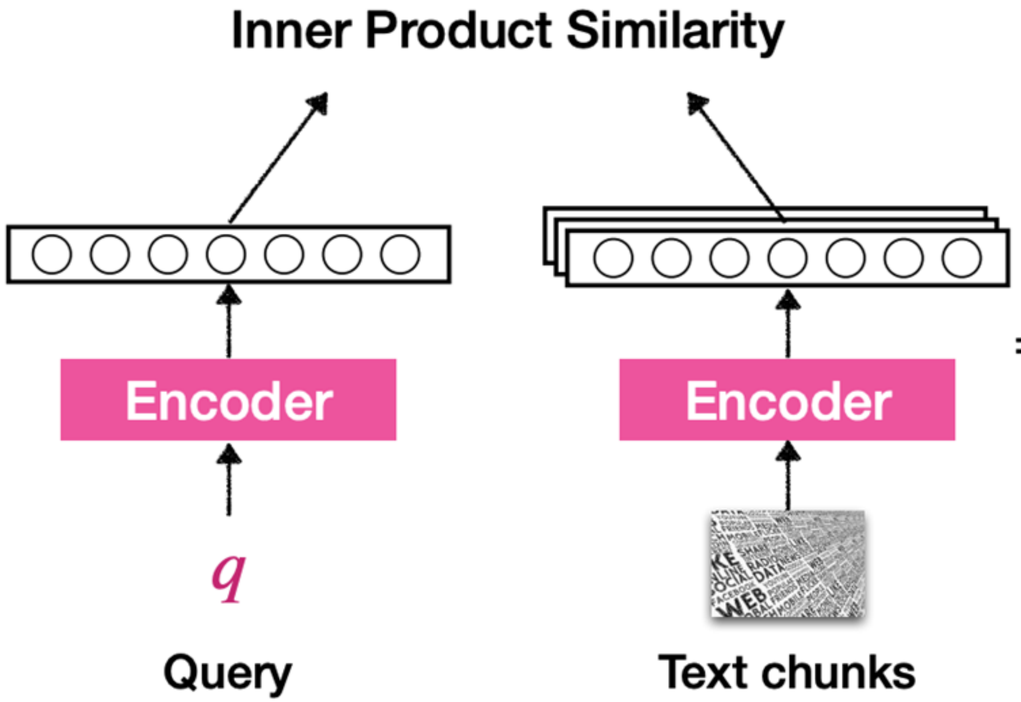
**Objective:** Train a model to predict the relevance of a document to a given query by optimizing a similarity score (e.g., dot product)

**Training Data:**

- Requires queries paired with relevant (positive) and irrelevant (negative) documents
- **Positive examples:** From datasets containing human-annotated relevant documents
- **Negative examples:**
  - Typically sampled from the top-1000 results retrieved by an existing retriever (e.g., BM25)
  - This ensures that the negatives are challenging ⇒ Discourage the model from learning only trivial distinctions between relevant and irrelevant documents
- If there are no labeled positive examples, use **relevance-guided supervision** [Khattab et al., 2021]:
  - Extract the top-ranked documents from an existing IR system that contain relevant answer strings (treat these as positive examples)
  - Use documents from the same top-ranked results that lack answer strings as negative examples
  - Train a new retriever and iterate

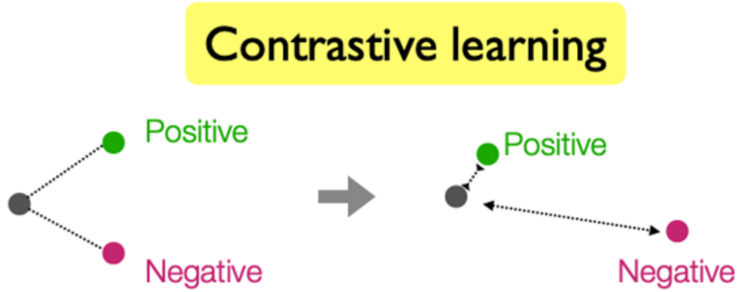
# Training dense retrievals (cont.)

Train the model to maximize the score for positive documents and minimize the score for negative ones:




$$L(q, p^+, p_1^-, p_2^-, \ldots, p_n^-)$$

$$= -\log \frac{\exp(\text{sim}(q, p^+))}{\exp(\text{sim}(q, p^+)) + \sum_{j=1}^{n} \exp(\text{sim}(q, p_j^-))}$$






Source: ACL 2023 Tutorial: Retrieval-based Language Models and Applications; Section 3: Retrieval-based LMs: Architecture

# Maximum Inner Product Search (MIPS)

**MIPS:** The problem of determining a small subset of items in a datastore whose vector representations have the highest dot product with a given query vector

Although there is an **obvious linear-time implementation O(#datastore items ·ↀ· vector size)**, it is generally **too slow** to be used on practical problems because the datastores are huge (millions or billions of vectors, each hundreds or thousands of dimensions)

To optimize the retrieval speed, the common choice is the approximate nearest neighbors (ANN) algorithm to return approximately top-k nearest neighbors to **trade off a little accuracy lost for a huge speedup**; https://ann-benchmarks.com/

# Index

During the **indexing phase**, each document/passage is encoded into a vector

All these document vectors are stored in the **index**, typically in a structure that allows for fast similarity search

**FAISS (Facebook AI Similarity Search)** https://github.com/facebookresearch/faiss

- A library developed by Meta (Facebook) to perform fast similarity search on large-scale dense vectors
- FAISS builds an index by dividing the dense space into clusters.
- At search time, the query vector is compared only with the most relevant clusters, significantly reducing the number of comparisons

# Retrieval Evaluation

Each document returned by the IR system is either relevant to our purposes or not relevant

How to evaluate the retrieval then?

- **Precision:** # relevant documents among all documents retrieved (to be relevant)
- **Recall:** # truly relevant documents that are retrieved
- **F1:** as always, a weighted harmonic mean of the precision and recall

*Issue*: It doesn't really measure the performance of a system that ranks the documents

| | What do retrieve? | How to use retrieval? | When to retrieve? |
|---|---|---|---|
| REALM (Guu et al 2020) | Text chunks | Input layer | Once |
| Retrieve-in-context LM (Shi et al 2023, Ram et al 2023) | Text chunks | Input layer | Every n tokens |
| RETRO (Borgeaud et al. 2021) | Text chunks | Intermediate layers | Every n tokens |
| kNN-LM (Khandelwal et al. 2020) | Tokens | Output layer | Every token |
| FLARE (Jiang et al. 2023) | Text chunks | Input layer | Every n tokens *(adaptive)* |
| Adaptive kNN-LM (He et al 2021, Alon et al 2022, etc) | Tokens | Output layer | Every n tokens *(adaptive)* |
| Entities as Experts (Fevry et al. 2020), Mention Memory (de Jong et al. 2022) | Entities or entity mentions | Intermediate layers | Every entity mentions |
| Wu et al. 2022, Bertsch et al. 2023, Rubin & Berant. 2023 | Text chunks ***from the input*** | Intermediate layers | Once or every n tokens |

Source: [ACL 2023 Tutorial: Retrieval-based Language Models and Applications]; Section 3

**Goal:** Dive into one NLP application: Question Answering

- QA Landscape
- An Overview of Retriever-Reader & Retriever-Generator Architectures for Open-Ended QA
- Dense Retrieval
- Answer Extractor
- Retrieval Augmented Generation (RAG)
- RAG: Overview of Retriever-Generator training options

Figure: [Jurafsky and Martin]

# Reader algorithms: Answer span extractor

**Goal:** Given retrieved text (e.g., passage) and the question, predict which token is the start of the answer span and which token is its end

S, E … vectors of the size of the final token representations, randomly initialized & trained

Concatenate question & passage, and delineate them:

- With BERT-like models, use [SEP]
- With today's LM use text like "Passage:"

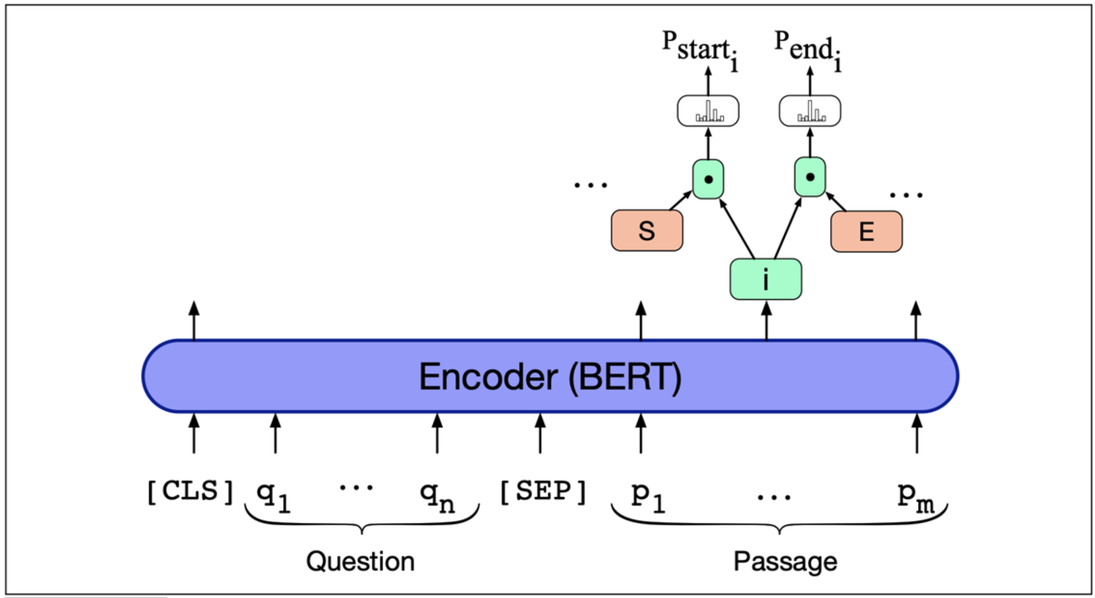



**Figure 14.12** An encoder model (using BERT) for span-based question answering from reading-comprehension-based question answering tasks.

A span-start probability for each position i:

$$P_{\text{start}_i} = \frac{\exp(S \cdot p'_i)}{\sum_j \exp(S \cdot p'_j)}$$

The final representation of the j-th token

A span-end probability for each position i:

$$P_{\text{end}_i} = \frac{\exp(E \cdot p'_i)}{\sum_j \exp(E \cdot p'_j)}$$

Figure: [Jurafsky and Martin]

# Reader algorithms: Answer span extractor

Minimizing a negative log-likelihood loss encourages the model to maximize the probabilities of the true start (i*) and end (j*) positions

The probability of correctly predicting the answer span involves:

1. Predicting the correct start position
2. Predicting the correct end position

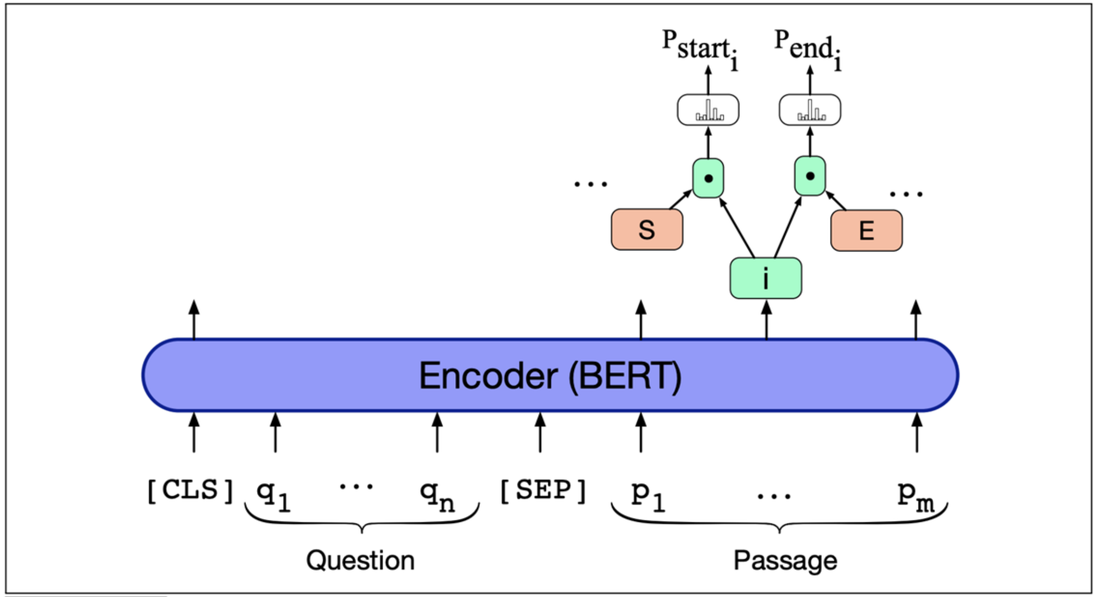



**Figure 14.12** An encoder model (using BERT) for span-based question answering from reading-comprehension-based question answering tasks.

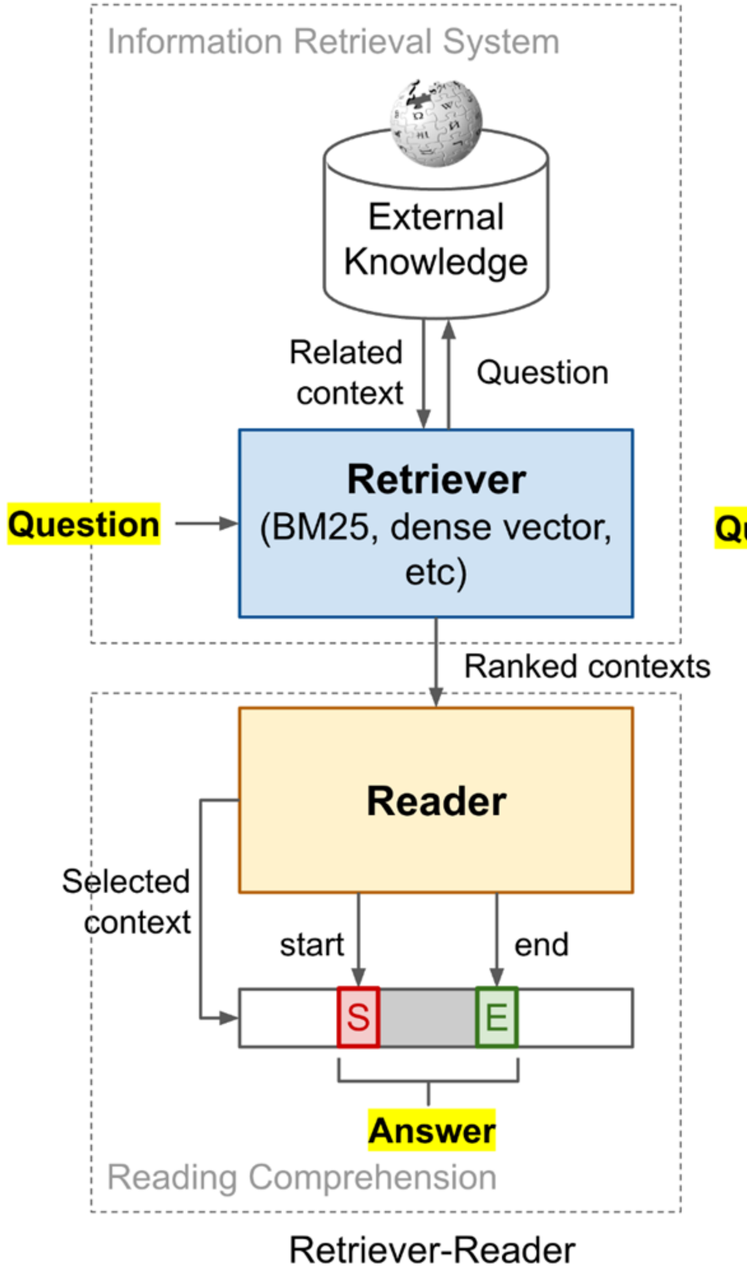
These two events are treated as independent events, the joint probability of both predictions being correct is the product of the two probabilities

Therefore the loss is:

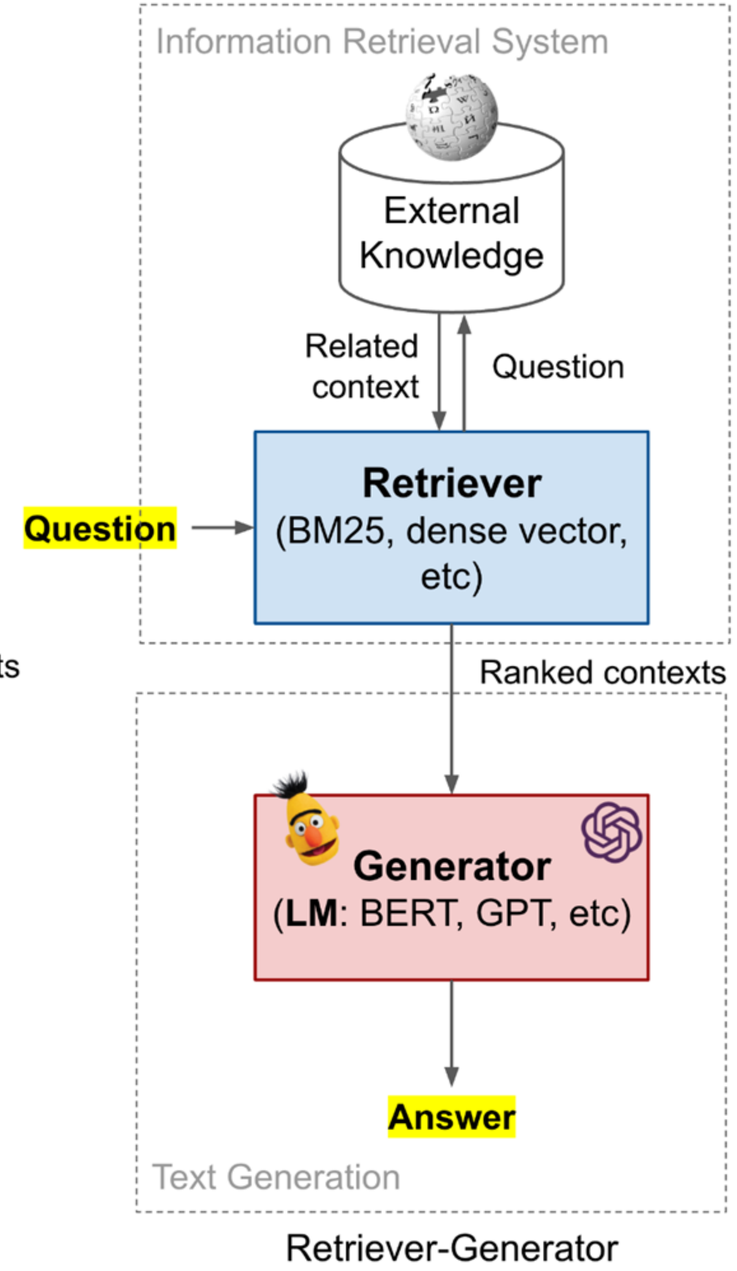
$$L = -\log(P_{\text{start}_{i*}} \cdot P_{\text{end}_{j*}}) = -\log P_{\text{start}_{i*}} - \log P_{\text{end}_{j*}}$$

**Goal:** Dive into one NLP application: Question Answering

- QA Landscape
- An Overview of Retriever-Reader & Retriever-Generator Architectures for Open-Ended QA
- Dense Retrieval
- Answer Extractor
- Retrieval Augmented Generation (RAG)
- RAG: Overview of Retriever-Generator training options



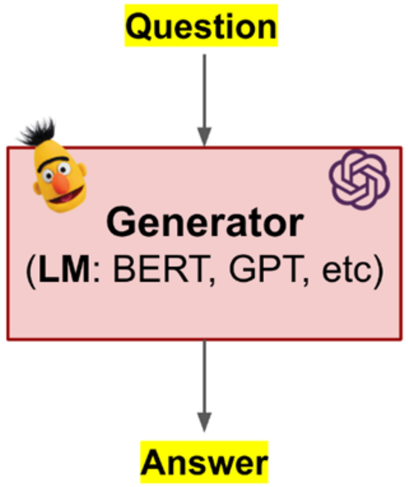

Retriever-Reader




Retriever-Generator

A LM's ***parametric knowledge***: The information that the model has encoded within its parameters/weights during training that it can then use to do tasks for which that knowledge is required

*So, why LLMs need retrieval?*




Generator

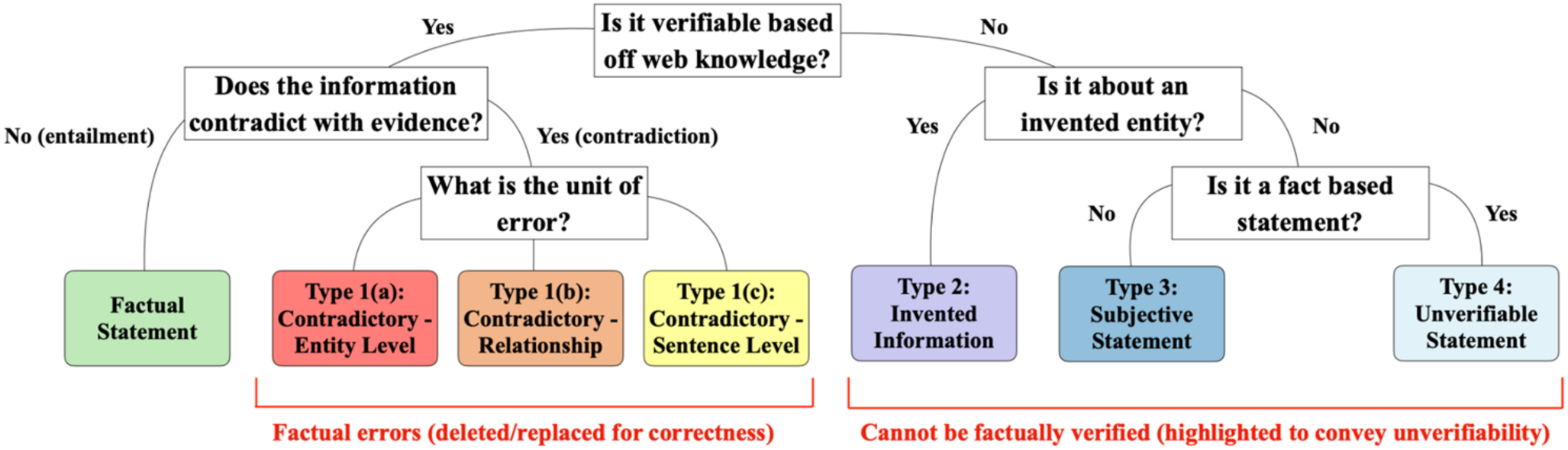


Source: https://lilianweng.github.io/posts/2020-10-29-odqa/

# Hallucination & Factuality

A hallucination is a response that is not faithful to the facts of the world

Fine-grained hallucination taxonomy [Mishra et al., 2024]:



# Retrieval Augmented Generation (**RAG**)

1. Use a retriever to identify top-k most relevant documents for the user's prompt
2. Add the retrieved documents to the input
3. Ask the LLM to generate the answer based on this rich context
4. Finetune the answer-generator LLM and/or retrieval

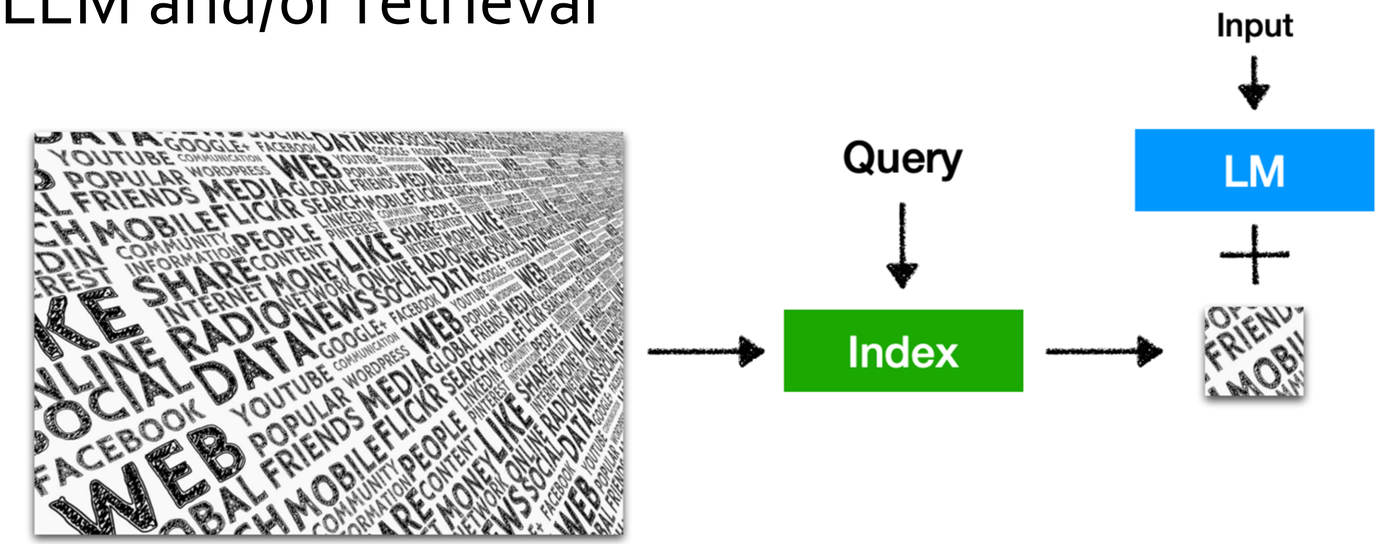



Figure Source: ACL 2023 Tutorial: Retrieval-based Language Models and Applications; Section 2

# *Reminder:* Input Context Length

**The attention matrix is quadratic in the maximum sequence length**

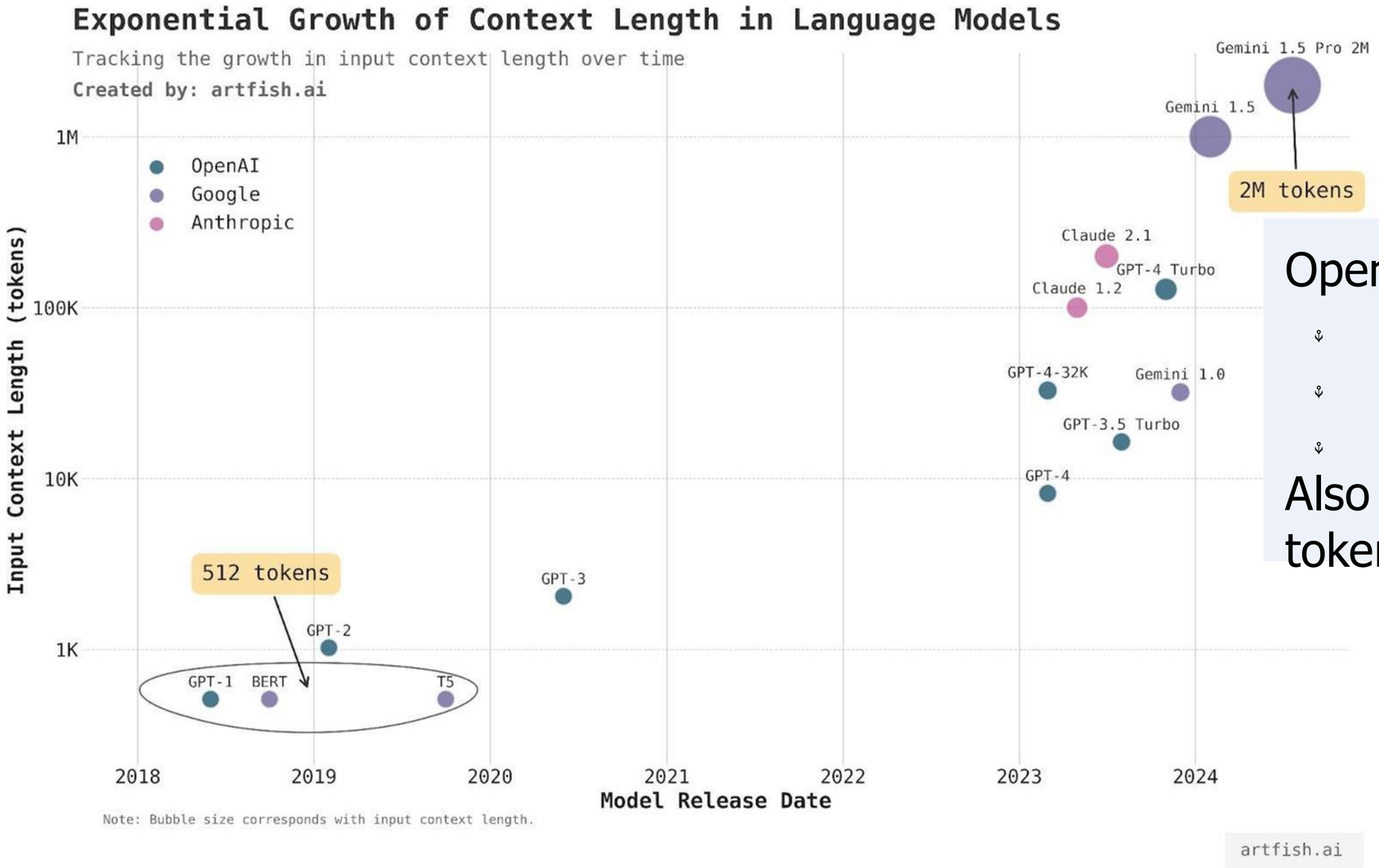
- If the length of an input sequence doubles, the amount of memory required quadruples
- Training an LLM on sequence lengths of 128k will require ~1024x the memory compared to training on sequence lengths of 4k
- GPU memory

**Poor generalization due to position encoding:**

- RoPE is the current choice (aims to preserve the relative distance between tokens)
- Performance quickly breaks down for sequence lengths significantly longer than the model has seen before [Press et al., 2022]

You can process sequences of arbitrary lengths, but you shouldn't expect a good performance for sequences longer than what's used for pretraining/post-training because of RoPE, & creators of LLMs are prevented from increasing the sequence length drastically due to the GPU memory limits

# Input Context Length




Open-weight LLMs:

- LLaMA 3.1
- Qwen 2.5
- Mistral-Large

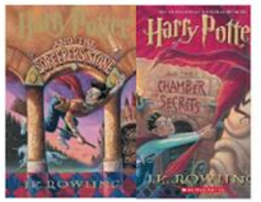
Also fit 128K tokens!

Source: https://www.artfish.ai/p/long-context-llms

# Google's Gemini 1.5 can (almost) fit the entire Harry Potter + Lord of the Ring series in its 2 million context window

2M — Gemini 1.5 2M (June 2024)


200K — Claude 2.1 (July 2023)



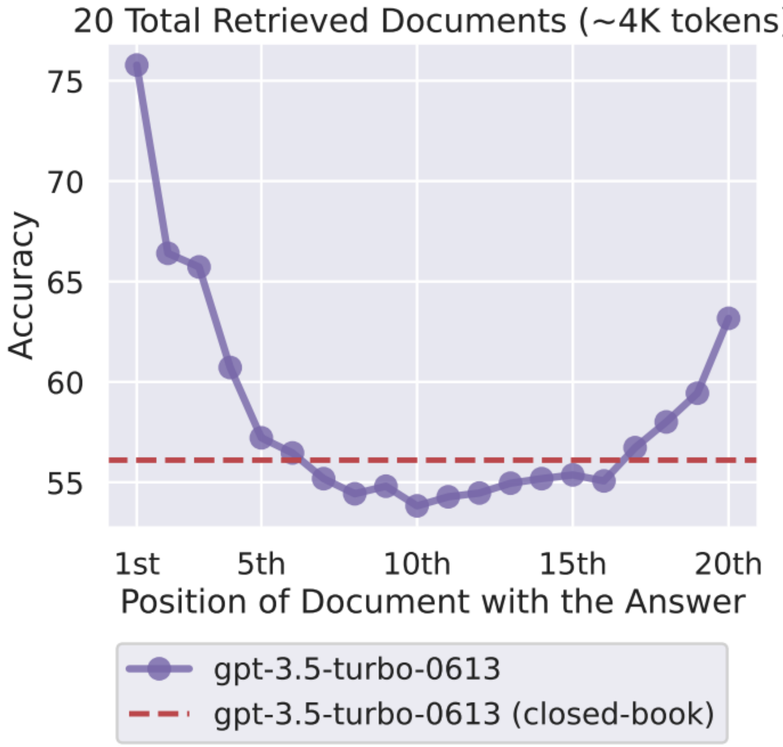

128K — GPT-4 Turbo (March 2023)

GPT-3.5 Turbo (March 2022)

Source: https://www.artfish.ai/p/long-context-llms

# Practical Considerations

1. Longer the input context length, the more potentially relevant documents we can squeeze, but:
   - [Liu et al., 2023]: "*models are **better at using relevant information** that occurs **at the very beginning** (primacy bias) **or end of its input context** (recency bias), and performance degrades significantly when models must access and use information located in the middle of its input context*"
1. [Ying et al., 2024]: Conflicts between internal/parametric knowledge and knowledge in a given context
   - Curation of the context is thus important



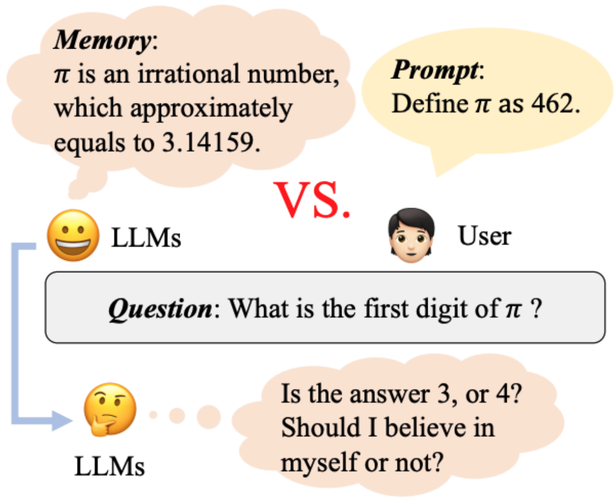




Figure 1: In conflict situation , LLMs may depend on the prompt or intuitively answer based on memory

**Goal:** Dive into one NLP application: Question Answering

- QA Landscape
- An Overview of Retriever-Reader & Retriever-Generator Architectures for Open-Ended QA
- Dense Retrieval
- Answer Extractor
- Retrieval Augmented Generation (RAG)
- RAG: Overview of Retriever-Generator training options

# Retrieval-based LMs: Training

Training challenges:

- External datastore is huge
  ⇒ Expensive to update the index = recompute dense vectors for all documents
- LLMs are large
  ⇒ Expensive to finetune an LLM to generate answers

**Option 1 – Independent Training:**
Retrieval models and language models are trained independently

Source: ACL 2023 Tutorial: Retrieval-based Language Models and Applications; Section 4

# Problem with this training option

**We ignore that a retriever and a generator work together**

We want to improve the RAG system such that:

- The retriever learns to select passages that make the generator's job easier
- The generator learns to adapt to whatever the retriever gives

$$L = -\log \sum_{d \in \text{Docs}} p_{\text{retriever}}(d|q;\theta_r) p_{\text{generator}}(y|q,d;\theta_g)$$

Millions or billions ⇒ Too slow

# Retrieval-based LMs: Training

**Option 2 – Sequential Training:**
One component is first trained independently & then fixed, the other component is trained with an objective that depends on the first one

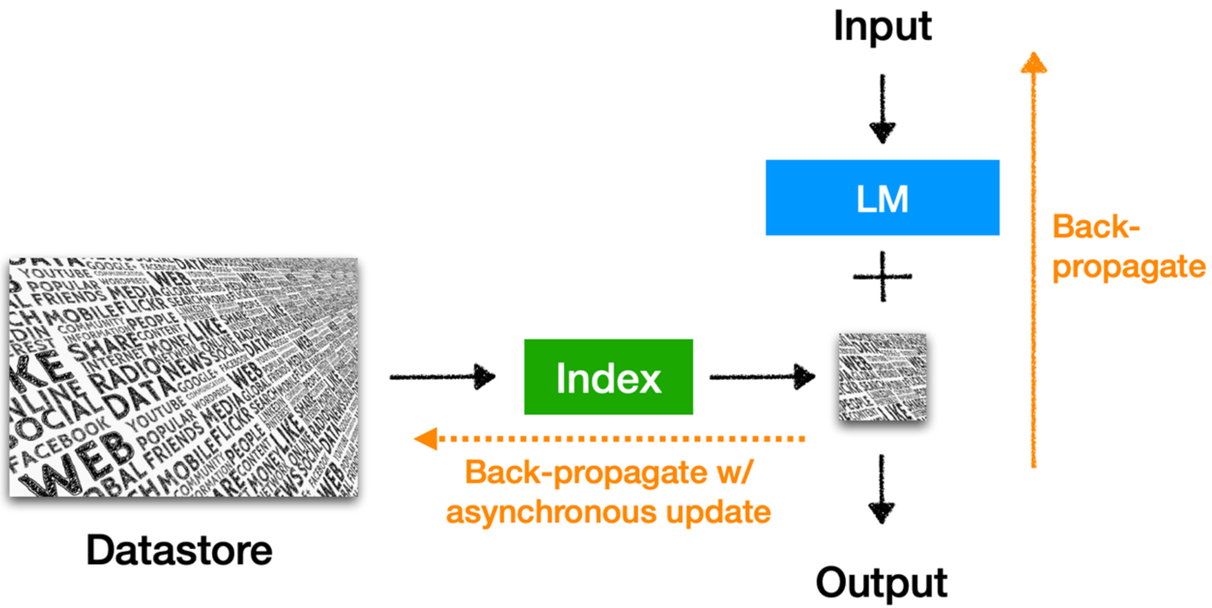
- Retrievers are trained to provide text that helps LMs the most


Source: ACL 2023 Tutorial: Retrieval-based Language Models and Applications; Section 4

# Retrieval-based LMs: Training

**Option 3 – Joint training w/ asynchronous index update:**

- Retrieval models and language models are trained jointly
- Allow the index to be "stale"; rebuild the retrieval index every T steps




Source: ACL 2023 Tutorial: Retrieval-based Language Models and Applications; Section 4

# Retrieval-based LMs: Training

## Option 4 – Joint training w/ in-batch approximation:

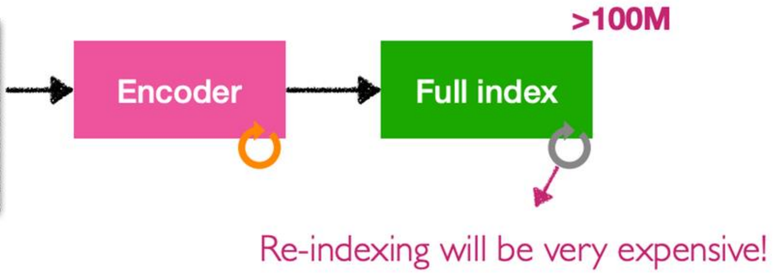
- Retrieval models and language models are trained jointly
- Use "in-batch index" instead of full index
- Treat each other document in the batch as a negative example for a given query

$$L_{\text{retriever}} = -\log p_{\text{retriever}}(d^+|q;\theta_r) = -\log \frac{e^{\text{sim}(q,d^+)}}{\sum_{d' \in \text{Batch}} e^{\text{sim}(q,d')}}$$

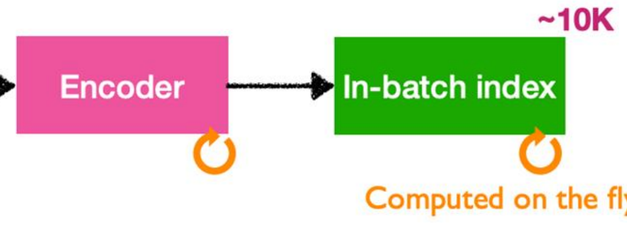
$$L_{\text{generator}} = -\log p_{\text{generator}}(y_{\text{correct}}|q,d^*), \quad \text{where } d^* \sim p_{\text{retriever}}(\cdot|q)$$

$$L = L_{\text{retriever}} + L_{\text{generator}}$$

Full corpus




Training batch




Source: [ACL 2023 Tutorial: Retrieval-based Language Models and Applications](); Section 4

| Training method | 👍 | 👎 |
| --- | --- | --- |
| Independent training (Ram et al 2023; Khandelwal et al 2020)<br>Sequential training (Borgeaud et al 2021; Shi et al 2023) | ✳ Easy to implement: off-the-shelf models<br>✳ Easy to improve: sub-module can be separately improved | ✳ Models are not end-to-end trained — suboptimal performance |
| Joint training: async update (Guu et al 2020; Izacard et al 2022)<br>Joint training: in-batch approx (Zhong et al 2022; Min et al 2023; Rubin and Berant 2023) | ✳ End-to-end trained — very good performance! | ✳ Training may be complicated (overhead, batching methods, etc)<br>✳ Train-test discrepancy still remains |

How do retrieval-based language models perform on downstream tasks? → **Section 5**!

Source: [ACL 2023 Tutorial: Retrieval-based Language Models and Applications]; Section 4

# ODQA “Developer Guide”

https://drive.google.com/file/d/1vh8S13V-LvgdhTlBrcvoXUiPYuzLCSTA/view?usp=sharing