

HOMework 4^{*}

RETRIEVAL AUGMENTED GENERATION (RAG) AND MEMORY[†]

24-880 AI AGENTS FOR ENGINEERS
<https://cmuagents.github.io>

OUT: February 15th, 2026
DUE: February 22nd, 2026
TAs: Peter Pak
CAs: Hardik Choudhary

Instructions

- **Collaboration Policy:** Please read the collaboration policy in the syllabus.
- **Late Submission Policy:** See the late submission policy in the syllabus.
- **Submitting your work:** You will use Gradescope to submit answers to all questions and code.
 - **Written:** You will submit your completed homework as a PDF to Gradescope. Please use the provided template for the submissions which can be written in $\text{\LaTeX}$ (via [Overleaf](#)) or directly on the PDF document. Each answer should be within the box provided. If you do not follow the template, your assignment may not be graded correctly and there will be a **2% penalty** (e.g., if the homework is out of 100 points, 2 points will be deducted from your final score).
 - **Programming:** Gradescope's Autograder will be used to evaluate the coding portions of this assignment. In order to best obtain the points for this assignment, please follow the code upload instructions. If you do not follow the instructions correctly the autograder may not work properly.

^{*}Adapted from CMU 10-623 Generative AI Homework Template

[†]Compiled on Wednesday 11th February, 2026 at 19:34

Question	Points
Dense Passage Retrieval (Written)	25
Dense Passage Retrieval (Programming)	20
RAG-Sequence and RAG-Token (Written)	20
RAG-Sequence and RAG-Token (Programming)	20
Memory and Long-Term Context	10
Programming Scripts Upload	0
Collaboration and Generative AI Questions	5
Total:	100

Introduction

In this homework assignment we'll take a brief look into the Retrieval Augmented Generation (RAG) and some concepts around memory for Large Language Models.

At a high-level, this homework will cover the following aspects.

1. Understand how Retrieval Augmented Generation (RAG) works at a high level and look into in more detail
2. Explore Dense Passage Retrieval (DPR) and how it works.
3. Compare RAG-Sequence and RAG-Token Models.
4. Explore methods for long term context and memory.

Local Environment

For this homework we'll be using `uv` as the project and package manager. If you haven't install `uv`, you can follow the installation instructions here <https://docs.astral.sh/uv/>. You can explore all of the available commands using `uv --help` but in short, here are a couple of useful commands that you'll often use:

<code>source .venv/bin/activate</code>	(Linux and Mac) Activates python environment.
<code>source .venv/Scripts/Activate</code>	(Windows) Activates python environment.
<code>uv sync</code>	Installs / synchronizes projects dependencies.
<code>uv add <package></code>	Installs <code>pip</code> package and adds it to project dependencies.
<code>uv remove <package></code>	Removes <code>pip</code> package dependency from project.
<code>uv run <script.py></code>	Runs python script inside of <code>uv</code> environment.

1. Navigate to folder project directory and synchronize dependencies with:

```
uv sync
```

2. Activate the environment

Mac and Linux:

```
source .venv/bin/activate
```

Windows:

```
source .venv/Scripts/activate
```

3. Test `uv` works and run `main.py` file.

```
uv run main.py
```

Retrieval-Augmented Generation

Retrieval-Augmented Generation (RAG) is a hybrid approach for language generation as it utilizes non-parametric memory (samples not in training dataset) along with parametric memory (pretrained data) to generate responses utilizing the retrieved data without additional pretraining [Lewis et al. \[2020\]](#). The approach is composed in two parts; the retriever and the generator. The retriever is composed of Dense Passage Retriever (DPR) [Karpukhin et al. \[2020\]](#) which, in broad terms, indexes a set of passages into a low dimensional and continual space to retrieve the most relevant passages from a provided input query [Lewis et al. \[2020\]](#). The generator is a pretrained transformer which concatenates the input query x with the retrieved content y in order to generate a response using the augmented context [Lewis et al. \[2020\]](#).

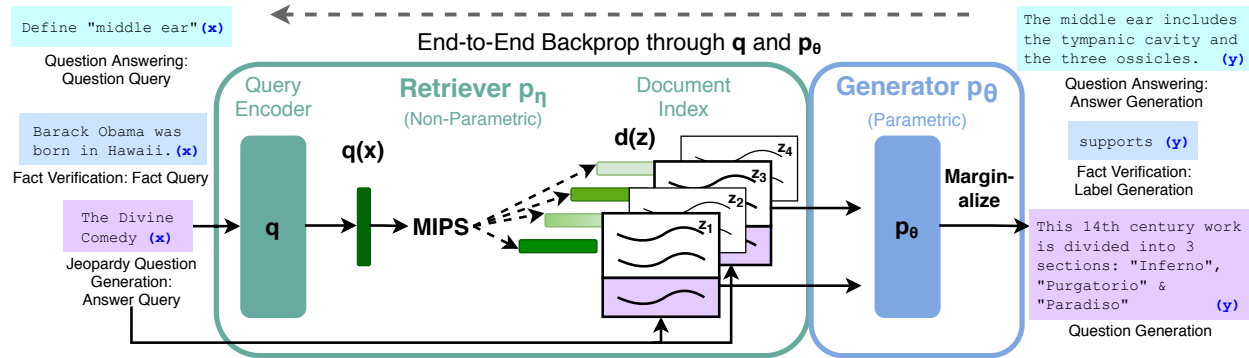


Figure 1: Illustrations from [Lewis et al. \[2020\]](#). Overview of the RAG approach, A pre-trained retriever (Query Encoder + Document Index) with a pre-trained seq2seq model (Generator) is fine-tuned end-to-end. For query x , Maximum Inner Product Search (MIPS) is used to find the top-K documents z_i . For final prediction y , z is treated as a latent variable and the seq2seq predictions are marginalized to give different documents.

1 Dense Passage Retrieval (Written) (25 points)

The Dense Passage Retriever (DPR) is the retrieval portion of RAG and utilizes two separate transformers to create the embeddings for the queries (q) and the embeddings for the passages (p) [Karpukhin et al. \[2020\]](#). The embeddings for both the query and passage texts are obtained from the class token of the tokenized inputs. Query text is not chunked and pre-processed by appending the character sequence with a separation token (`[SEP]`), for example a preprocessed query would look something like `What is machine learning? [SEP]`. Passage text is obtained by chunking source text, such as Wikipedia articles, into blocks consisting of 100 words where the article title is prepended to the block and separated using a SEP token (i.e. `Machine Learning [SEP] Machine learning (ML) is a field of study in artificial intelligence...`) [Karpukhin et al. \[2020\]](#). The tokenized output is obtained by first passing the input text through an encoder (i.e. BERT [Devlin et al. \[2019\]](#)) network which generates the hidden states and appends a class token (`[CLS]`). The class token, which attends to all of the tokens via self-attention [Vaswani et al. \[2023\]](#), provides the dense embedding used to determine similarity between the query and passage [Karpukhin et al. \[2020\]](#). The similarity is defined in Equation 1 as dot products of the encoders for the query and the passage vectors [Karpukhin et al. \[2020\]](#).

$$\text{sim}(q, p) = E_Q(q)^\top \cdot E_P(p) \quad (1)$$

Before we implement Dense Passage Retrieval, lets go over some of it's conceptual aspects.

1.1. From which token is the embedding for a query or passage obtained?

1.1.a. (2 points) Multiple Choice

- ☐ [SEP] token
- ☐ [CLS] token
- ☐ [EOS] token
- ☐ [MASK] token

1.1.b. (3 points) Explanation

1.2. How are queries handled with respect to chunking?

1.2.a. (2 points) Multiple Choice

- ☐ Input is split into multiple 100-word chunks
- ☐ Input is split into character-level tokens and averaged
- ☐ Input is treated as a single sequence

1.2.b. (3 points) Explanation

1.3. How are passages handled with respect to chunking?

1.3.a. (2 points) Multiple Choice

- ☐ Input is split into multiple 100-word chunks
- ☐ Input is split into character-level tokens and averaged
- ☐ Input is treated as a single sequence

1.3.b. (3 points) Explanation

1.4. What is a key advantage of dense retrieval (DPR) over sparse retrieval methods like BM25 or TF-IDF?

1.4.a. (2 points) Multiple Choice

- ☐ Dense retrieval requires less computational resources at query time
- ☐ Dense retrieval does not require any training data
- ☐ Dense retrieval can match semantically similar text even without lexical overlap
- ☐ Dense retrieval produces more interpretable relevance scores

1.4.b. (3 points) Explanation

1.5. Why does DPR use two separate encoders (E_Q and E_P) instead of a single shared encoder for both queries and passages?

1.5.a. (2 points) Multiple Choice

- ☐ To reduce the total number of model parameters
- ☐ To ensure queries and passages have different embedding dimensions
- ☐ To allow passage embeddings to be pre-computed and cached offline
- ☐ To prevent overfitting during training

1.5.b. (3 points) Explanation

2 Dense Passage Retrieval (Programming) (20 points)

- 2.1. (20 points) For this part of the assignment we'll implement a simplified version of Dense Passage Retrieval outlined in the starter code: `./programming/dpr.py`

This file consists of preprocessing functions and a `SimpleDPR` class:

1. `preprocess_query()` - Format queries with `[SEP]` token
2. `preprocess_passage()` - Combine title and text with `[SEP]` token
3. `chunk_document()` - Split documents into word chunks
4. `SimpleDPR.encode()` - Encode texts into dense embeddings
5. `SimpleDPR.index_passages()` - Build passage embedding index
6. `SimpleDPR.retrieve()` - Retrieve top-k passages using dot product similarity

Please fill out the remaining code indicated by the `# Fill in` comments within the code.

You can run and test the code with the following commands:

<code>uv run programming/dpr.py</code>	Runs script with example case (useful for development)
<code>uv run test.py dpr</code>	Runs test script that will be used in autograder

The provided tests are the exact ones that will run on the autograder (i.e. if your local tests pass, they should also pass on the autograder).

Note: The autograder runs on CPU only. The starter code is configured to use CPU to ensure consistent results between your local environment and the autograder. You may modify the `DEVICE` variable to use GPU for faster local development, but ensure you set it back to `"cpu"` before running tests.

3 RAG-Sequence and RAG-Token (Written) (20 points)

The generation component $p_\theta(y_i \mid x, z, y_{1:i-1})$ of RAG utilizes an encoder-decoder model (originally BART-large [Lewis et al. \[2019\]](#)) in order to generate responses from the query passage inputs. Within their work, Lewis et al. [Lewis et al. \[2020\]](#) offers a sequence model approach along with a token model approach for generating the output. The RAG-Sequence model forms k different generator model inputs for a given query. These inputs are sourced from individually top-K retrieved passages z using the retriever $p_\eta(z \mid x)$ and couple together with the given query to produce k different model output sequences where in which the highest scoring sequence is selected (Equation 2) [Lewis et al. \[2020\]](#).

$$p_{\text{RAG-Sequence}}(y \mid x) \approx \sum_{z \in \text{top-k}(p(\cdot \mid x))} p_\eta(z \mid x) \prod_i^N p_\theta(y_i \mid x, z, y_{1:i-1}) \quad (2)$$

The RAG-Token model applies the same concept but ranks individually generated tokens $y_{1:i-1}$ rather than the entire sequence, allowing for a mixture of different passages to influence the generation of the output sequence (Equation 3) [Lewis et al. \[2020\]](#).

$$p_{\text{RAG-Token}}(y \mid x) \approx \prod_i^N \sum_{z \in \text{top-k}(p(\cdot \mid x))} p_\eta(z \mid x) p_\theta(y_i \mid x, z, y_{1:i-1}) \quad (3)$$

Of the two approaches, the authors found that the RAG-Token model outperformed the RAG-Sequence model in question answer tasks specifically those in open-domain [Lewis et al. \[2020\]](#). This is attributed to the nature of open-domain questions are required to combine facts from multiple passages and the token level approach allows for generation to switch passages mid-generation [Lewis et al. \[2020\]](#). In the broader scope, RAG was found to generate more specific and accurate results than the existing parameteric only seq2seq models available at the time [Lewis et al. \[2020\]](#).

3.1. In RAG-Sequence, how are the retrieved passages used during generation?

3.1.a. (2 points) Multiple Choice

- ☐ All passages are concatenated into a single input for the generator
- ☐ Each passage generates a complete sequence independently, then scores are marginalized
- ☐ Passages are used only to initialize the decoder hidden state
- ☐ A single randomly selected passage is used for generation

3.1.b. (3 points) Explanation

3.2. When might RAG-Sequence be preferred over RAG-Token?

3.2.a. (2 points) Multiple Choice

- ☐ When the task requires combining facts from multiple documents
- ☐ When computational efficiency is the primary concern
- ☐ When generating coherent text that should be grounded in a single source
- ☐ When the retriever has low accuracy

3.2.b. (3 points) Explanation

3.3. How does RAG-Token differ from RAG-Sequence in terms of when marginalization over passages occurs?

3.3.a. (2 points) Multiple Choice

- ☐ RAG-Token marginalizes only at the end of generation
- ☐ RAG-Token marginalizes before the retrieval step
- ☐ RAG-Token does not perform marginalization
- ☐ RAG-Token marginalizes over passages at each token generation step

3.3.b. (3 points) Explanation

3.4. Why does RAG-Token outperform RAG-Sequence on open-domain question answering tasks according to [Lewis et al. \[2020\]](#)?

3.4.a. (2 points) Multiple Choice

- ☐ RAG-Token uses a larger generator model
- ☐ RAG-Token retrieves more passages than RAG-Sequence
- ☐ RAG-Token can combine facts from multiple passages by switching sources mid-generation
- ☐ RAG-Token has lower latency during inference

3.4.b. (3 points) Explanation

4 RAG-Sequence and RAG-Token (Programming) (20 points)

In this section, you will implement the probability computations for both RAG-Sequence and RAG-Token models. These implementations will help you understand the key difference between the two approaches: **when** marginalization over passages occurs.

4.1. (10 points) RAG-Sequence Implementation

Implement the RAG-Sequence probability computation in: `./programming/rag_sequence.py`

This file contains three functions to implement:

1. `compute_sequence_log_prob()` - Compute $\log \prod_i p(y_i|x, z, y_{1:i-1})$
2. `marginalize_over_passages()` - Compute $\log \sum_z p(z|x) \cdot p(y|x, z)$
3. `rag_sequence_probability()` - Full RAG-Sequence computation (Equation 2)

Key insight: In RAG-Sequence, the sum over passages is *outside* the product over tokens. Each passage generates a complete sequence independently.

<code>uv run programming/rag_sequence.py</code>	Runs script with example case
<code>uv run test.py rag_sequence</code>	Runs test script for autograder

4.2. (10 points) RAG-Token Implementation

Implement the RAG-Token probability computation in: `./programming/rag_token.py`

This file contains two functions to implement:

1. `marginalize_token_over_passages()` - Compute $\log \sum_z p(z|x) \cdot p(y_i|x, z, y_{1:i-1})$ for a single token
2. `rag_token_probability()` - Full RAG-Token computation (Equation 3)

Key insight: In RAG-Token, the sum over passages is *inside* the product over tokens. This allows different passages to influence different tokens in the output sequence.

<code>uv run programming/rag_token.py</code>	Runs script with example case
<code>uv run test.py rag_token</code>	Runs test script for autograder

Note: All computations are done in log-space for numerical stability. The `log_sum_exp()` function is provided for you.

The provided tests are the exact ones that will run on the autograder (i.e. if your local tests pass, they should also pass on the autograder).

5 Memory and Long-Term Context (10 points)

Large language models are fundamentally constrained by their context window (maximum number of tokens they can process in a single forward pass). While recent models have expanded context windows significantly (from 4K to over 1M tokens), this constraint remains a fundamental bottleneck for tasks requiring persistent memory across long interactions or access to large knowledge bases.

Two primary strategies have emerged to address this limitation:

1. **Long Context (LC):** Extending the model's context window to accommodate more tokens directly
2. **Retrieval-Augmented Generation (RAG):** Selectively retrieving relevant information from external storage when needed

Recent work like MemGPT [Packer et al. \[2024\]](#) draws inspiration from operating systems, treating the context window as “main memory” and external storage as “disk,” with the LLM managing data movement between tiers through function calls.

5.1. Why can't we simply make LLM context windows infinitely large, and how do hierarchical memory systems (like MemGPT) address this limitation?

5.1.a. (2 points) Multiple Choice

- ☐ Context windows are limited only by available GPU memory, so larger GPUs solve the problem
- ☐ The limitation is purely a software constraint that will be removed in future model versions
- ☐ Attention complexity scales quadratically with sequence length, making very long contexts computationally prohibitive
- ☐ Longer contexts always lead to worse model performance due to overfitting

5.1.b. (3 points) Explanation

5.2. When comparing Long Context (LC) approaches versus Retrieval-Augmented Generation (RAG) for handling information beyond a model's native context window, what is a key advantage of RAG over simply extending context length?

5.2.a. (2 points) Multiple Choice

- ☐ RAG always produces more accurate answers than long context models
- ☐ RAG requires less training data than long context models
- ☐ RAG can access arbitrarily large knowledge bases without proportional increases in inference cost
- ☐ RAG eliminates the need for a retriever model

5.2.b. (3 points) Explanation

6 Programming Scripts Upload (0 points)

To upload your code to Gradescope, run the following command to generate a zip file:

```
uv run programming/zip_programming.py
```

Not using the command above may cause Autograder to not work properly!

Upload the generated zip file to the appropriate programming slot on Gradescope.

6.1. Make sure the following files are included when uploading to Gradescope

- ☐ `dpr.py`
- ☐ `rag_sequence.py`
- ☐ `rag_token.py`

7 Collaboration and Generative AI Questions (5 points)

After you have completed all other components of this assignment, report your answers to these questions regarding the collaboration policy. Details of the policy can be found in the syllabus.

- 7.1. (2 points) Did you collaborate with anyone on this assignment? If so, list their name or Andrew ID and which problems you worked together on.

- 7.2. (3 points) Did you use generative AI for parts of this assignment? If so, include the input prompts along with links to the conversation.

References

- Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding, May 2019. URL <http://arxiv.org/abs/1810.04805>. arXiv:1810.04805.
- Vladimir Karpukhin, Barlas Oguz, Sewon Min, Patrick Lewis, Ledell Wu, Sergey Edunov, Danqi Chen, and Wen-tau Yih. Dense Passage Retrieval for Open-Domain Question Answering. In Bonnie Webber, Trevor Cohn, Yulan He, and Yang Liu, editors, *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 6769–6781, Online, November 2020. Association for Computational Linguistics. doi: 10.18653/v1/2020.emnlp-main.550. URL <https://aclanthology.org/2020.emnlp-main.550/>.
- Mike Lewis, Yinhan Liu, Naman Goyal, Marjan Ghazvininejad, Abdelrahman Mohamed, Omer Levy, Ves Stoyanov, and Luke Zettlemoyer. BART: Denoising Sequence-to-Sequence Pre-training for Natural Language Generation, Translation, and Comprehension, October 2019. URL <http://arxiv.org/abs/1910.13461>. arXiv:1910.13461 [cs].
- Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. Retrieval-augmented generation for knowledge-intensive NLP tasks. In *Proceedings of the 34th International Conference on Neural Information Processing Systems, NIPS '20*, pages 9459–9474, Red Hook, NY, USA, December 2020. Curran Associates Inc. ISBN 978-1-7138-2954-6. URL <https://dl.acm.org/doi/10.5555/3495724.3496517>.
- Charles Packer, Sarah Wooders, Kevin Lin, Vivian Fang, Shishir G. Patil, Ion Stoica, and Joseph E. Gonzalez. MemGPT: Towards LLMs as Operating Systems, February 2024. URL <http://arxiv.org/abs/2310.08560>. arXiv:2310.08560 [cs].
- Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention Is All You Need, August 2023. URL <http://arxiv.org/abs/1706.03762>. arXiv:1706.03762 [cs].