Retrieval-Augmented Generation

July 6, 2024

0.1 Topic

Extending the capabilities of LLMs using Retrieval-Augmented Generation techniques. Comparing the accuracy of answering questions between standard LLM vs. RAG LLM for different LLMs.

0.2 Abstract

This research aims to enhance the capabilities of Large Language Models (LLMs) using Retrieval-Augmented Generation (RAG) techniques. By comparing the accuracy of question-answering between standard LLMs and those integrated with RAG systems, this study explores the potential improvements in performance and robustness that RAG systems can offer.

0.3 Introduction

LLMs are quite useful and have taken the world of AI by storm. They have extensive usages, and we are still rediscovering what they can be used for. However, there are some aspects that are not ideal, for example:

- **Limitation in Memory**: LLMs are trained once on a set of data. Once trained, they cannot answer questions or generate text on data they have not been trained on (problem of under-fitting and high bias).
- **Hardware Requirements and Cost**: Running useful general-purpose LLM requires serious hardware capabilities, which comes with additional costs.
- **Privacy**: Large and useful LLMs are typically run by companies, requiring users to send data to externally managed servers. This may pose problems for individual users who need to submit private data and for companies, especially with GDPR requirements that might not be satisfied.

0.4 What is a Retrieval augmented generation

Retrieval-augmented generation (RAG) is an advanced artificial intelligence (AI) technique that combines information retrieval with text generation, allowing AI models to retrieve relevant information from a knowledge source and incorporate it into generated text.[6]

The basic idea around a RAG system is extending the capabilities of an already trained machine learning model (MLM) in particular an LLM.

One common problem in LLMs is that in order get a correct output/answer the LLM needs to be trained on a dataset that is relevant to the input/question. For example, asking an LLM who is the winner of 2024 euro championship will not result in a correct answer if the model is trained before 2024. Regardless of the capabilities of the chosen LLM, it just does not contain memory related to

this information. In case like this the LLM will need to be retrained. Different problem that occurs in LLMs and machine models in general is that perhaps the LLM is not powerful enough for the given task. Example for the second case is that if a model is asked extremely difficult question, even though the model has been trained on all the relevant data it might still not produce correct output in case of a difficult task.

Rag systems do not suffer as much as other models from this problem, their memory can be extended without the need for training or adding additional parameters. The system has two types of memory:

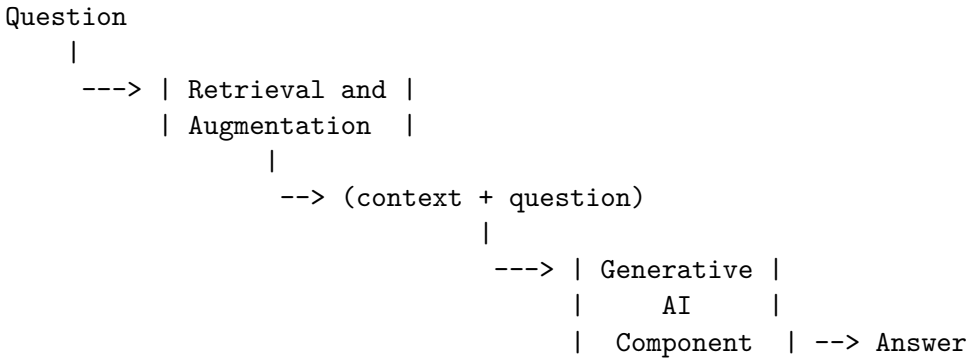
- **Parametric memory** - refers to the knowledge in the LLM generative model
- **Non-parametric memory** - stored in special types of databases. This non-parametric memory can be extended with additional knowledge easily and it still can be used by the LLM. **This is the core idea in RAG. Without any additional training new knowledge can be made available to the LLM.**

The RAG technique integrates a retrieval model and generative model. RAG systems usually work around extending the capabilities of NLP(Natural Language processing models) giving them additional memory or information on various topics. Therefor a production RAG system in the context of LLMs is build from 2 sub-systems:

- **Retrieval and Augmentation**
- **Generative AI component** or parametric memory, it is usually language generation tool like LLMs

From the perspective of usage of the RAG system, generating an answer is done in three phases:

- Phase 1 Retrieval, Knowledge retrieval, When asked a question the RAG retrieval system will retrieve all known information by submitting an appropriate query to the non-parametric memory.
- Phase 2 Augmentation, with the retrieved information, a context around the question is created, the context should contain all information for answering the question.
- Phase 3 Generation, The context and question will be passed to an LLM that will generate an answer.



0.5 Generative AI component or a LLM transformer

The Generative AI component is an LLM transformer. Typically, the transformer has two main components, the encoder and the decoder. The input text is initially tokenized, meaning it is split into small continuous lists of characters. In the next step, the model transforms the tokens into fixed-size vectors called embeddings. In the final step, in a so-called attention layer, an additional linear transformation on the embeddings is performed, making them dependent on the context they appear in. These context-dependent embeddings, also called contextual embeddings, are the output of the encoder and the input to the decoder. The encoder is of extreme importance in this text as it is a crucial part of a RAG system.

Embeddings are vector representations, typically produced by a neural network, whose main objective is to map (embed) the input media item into a vector space, where the locality encodes the semantics of the input [2].

The decoder takes the input and generates new embeddings and again combines the output with the embedded tokens to produce new tokens. The vector transformation where token embeddings are transformed to different ones to capture the context of the input text is called attention. It is described in the paper Attention Is All You Need: As a result of the initial research paper on transformers: The LLM transformers do not suffer from maintaining context and dependencies as much as recurrent or convolutional networks.

Transformers can be trained significantly faster than architectures based on recurrent or convolutional layers. Transformers outperform the best previously reported models in translation. [1]

The quality of the generated text generally depends on the number of the parameters of the model. Regardless of the quality of the output, every LLM for a given input is expected to generate grammatically correct text and semantically related to the input. For example if I ask even the smallest LLM "What is the capital of France?" it might return "Berlin is the capital of France". The answers will perhaps not be correct, but they will contain the context of the question and grammatically correct.

The generative component of the RAG system needs to be able for a given input and context to generate grammatically correct sentences related to the given context.

This is exactly what is achieved with the LLM transformers and that is why they are ideal to be used in the generation phase of the RAG system.

0.6 Retrieval System

The retrieval system is used to store and retrieve knowledge outside the knowledge maintained in the LLM. The knowledge here is in the form of paragraphs, sentences, or other forms of continuous text, referred to here as a text block. The text blocks are stored as embeddings.

This part of the system is composed of two components or modules:

- Sentence Embedding Model
- Non-parametric memory

0.6.1 Storing Phase:

The idea of the retrieval phase is for a given text block to create embedding and store this embedding in the non-parametric memory.

The process is as follows:

1. Take a text block, sentence, or paragraph.
2. Passes this text to the Sentence Embedding model to get an embedding(dense vector).
3. Stores this embedding in a non-parametric memory.

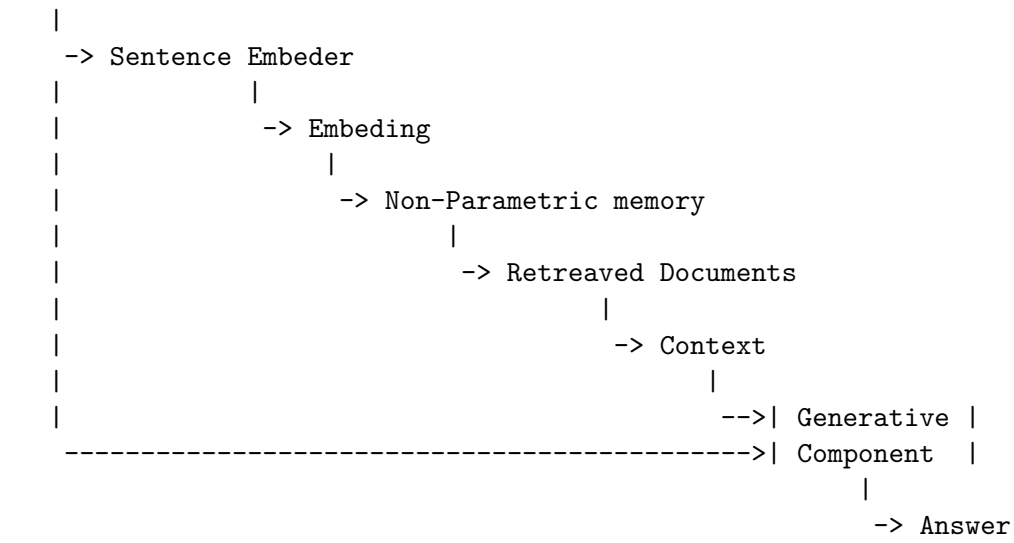
```
Sentence
|
 -> Tokenization
     |
      -> Embedded Vectors
          |
           -> Contextual Vectors
```

0.6.2 Retrieval Phase:

The idea of the retrieval phase is to return all relevant information from the non-parametric memory for a given text, query, or question.

The process is as follows:

1. Takes input text, referred to as a query.
2. Passes the query text to the Sentence Embedding model to get a query embedding or a query dense vector.
3. Using the query embedding, fetches a list of related knowledge from the non-parametric memory.



0.6.3 Non-parametric memory - vector databases

The non-parametric memory is usually a dense vector database. Besides being able to store vectors, a vector database must also have the capability to query for the k closest vectors to a given input vector. Like any database, fast storage and querying are important. Ideally, these databases should be able to store data on disk to enable horizontal scaling and data distribution, which are essential for a RAG system to achieve scalability compared to a traditional LLM model. This project will use the Faiss database as it offers all of the above requirements.

The Faiss library is dedicated to vector similarity search, a core functionality of vector databases. Faiss is a toolkit of indexing methods and related primitives used to search, cluster, compress, and transform vectors [2].

According to the referenced research, the Faiss database was tested with 768-dimensional ContrieverDb (name of the database from where vectors are imported) dense vector embeddings with up to 1M vectors with 64 bytes per dimension and also with 100M vectors from DeepDb that have a dimension of 96 and 8 bytes per dimension.

0.6.4 Sentence embedding models

Sentence embedding models are derived from LLM transformer models. LLMs transformer contains two general components, encoder and decoder. As described in the section "Generative AI component or a LLM transformer", the input of an encoder is a text, the output is a contextual embedding.

From particular interest for the Sentence embedding models and the RAG system is the encoder layer of the LLMs transformer.

Here is a description on how sentences are converted to embedding by an encoder.

LLM Transformer's encoder

```
Sentence
|
 -> Tokenization
     |
      -> Embedded Vectors
          |
           -> Contextual Vectors
```

The Sentence embedding models incorporates the LLM transformers encoder layer and adds additional layer of *pooling*. This layer operates on the token embedding and groups them in one embedding. This outputs embedding are dependent on the context of the input text, and are called sentence contextual embedding or sentence embedding. All the sentence contextual embedding have the same size, and it is equal to the size of the token embedding.

Here is a more visual description on the process of creating sentence embedding by a Sentence embedding model

Sentence embedding model

```
Sentence
 |
  -> Tokenization
      |
       -> Embedded Vectors
           |
            -> Contextual Vectors
                |
                 -> | Mean Pooling            |
                    | Max Pooling             |
                    | CLS Token               |
                    | Attention-based Pooling |
                       |
                        -> Sentence embedding
```

Pooling The pooling phase involves the task of combining multiple contextual token embeddings into one sentence embedding. The dimension of the token embeddings is the same as the created sentence embedding. Here are some of the popular pooling methods used in sentence embedding models:

- Mean Pooling
- Max Pooling
- CLS Token
- Attention-based Pooling

The most representative method is Mean Pooling, which calculates an embedding using the mean:

$$\text{meanpool}(e_1, e_2, \dots, e_n) = \frac{1}{n} \sum_{i=1}^{n} e_i$$

According to research paper [5], Mean Pooling, Max Pooling, and CLS Token are commonly used techniques.

The goal of sentence embedding models is to produce an embedding or vector from text. As mentioned before, they are built by adding a layer to the LLM's transformer encoder component. LLMs produce token embeddings, while sentence embedding models produce embeddings for entire sentences.

The requirement of the sentence embedding model is as follows:

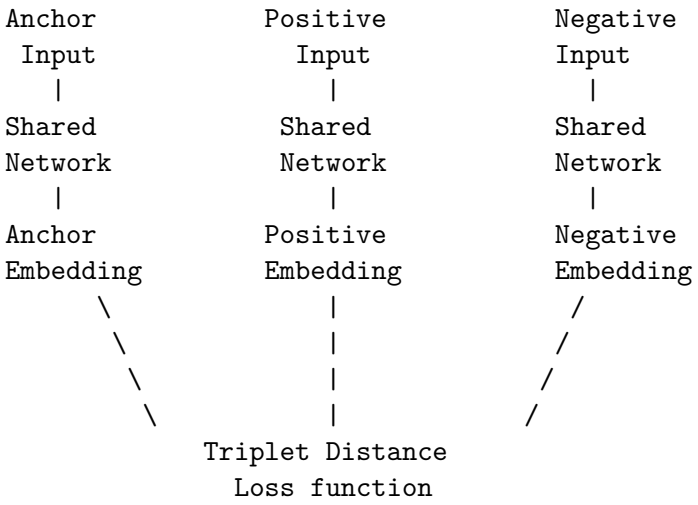
For every three blocks of text A, P, N, the model SE is to provide three embeddings $SE(A)$, $SE(P)$, $SE(N)$,
such that: if A and P are semantically more similar A and N, then the norm $|SE(A) - SE(P)| < |SE(A) - SE(N)|$.

The simple interpretation is that the sentence embedding model takes a block of text and captures its meaning or semantics by describing it as a vector. This is an extremely useful feature because it allows for representing the meaning and information of sentences in a way that enables the measurement of semantic information.

Siamese training network and loss function Up to this point, I have explained how a sentence embedding model works, including its input and output. Even though the input and output are in the required format, sentence embedding models are additionally trained to meet the semantic requirements defined above.

The main issue during training is that we don't really know the exact output for a block of text, it can be any embedding. It is the relative distance between three embeddings that is important. Semantically close embeddings should be closer than embeddings that have different meanings. To solve this problem, the model is trained using a Siamese network or a variation of the Siamese network, where conceptually three copies of the network model are used, and one single test case consists of three text blocks: A, P, and N. * The text block A is used for comparison and is called an anchor. * The text block P is semantically similar to A and is called the positive case. * The text block N is semantically different from A and P and is called the negative case.

Siamese network with triplet loss function



This Siamese network ends with a triplet distance activation function. The function calculates the distance $D(A, P)$ $D(A, N)$ and is activated if $D(A, P) > D(A, N) + \text{margin}$. The triplet function from the perspective of the Sentence embedder model is a Loss function and its output is used for training.

$$Loss = \max(0, D(A, P) - D(A, N) + \text{margin})$$

[5]

If $D(A, P) < D(A, N)$, the function returns 0 and nothing should be adjusted for the current test case. If $D(A, P) > D(A, N)$, then the function returns a value greater than 0, so the model's parameters should be adjusted to minimize this distance. The margin is there to give a bias to the loss function and make sure that only for really small $D(A, P)$ compared to $D(A, N)$ the model should not be adjusted, but for most cases, it will be adjusted. For no adjustments to happen, the following condition should be true: $D(A, P) + \text{margin} < D(A, N)$.

At this point, all of the components of the sentence embedding model are explained. A representative example of a sentence embedding model is SBERT, which is based on the popular LLM BERT. This resource will use SBERT as a Sentence Embedder Model.

0.7 Related work

0.7.1 Augmentation

Interesting research has been done using the so-called *RAG-Token Model*. The RAG technique described, initially retrieves n documents related to the question then augments the context of the question and sends the context and the question as an input to an LLM or the generator. This type of RAG system is called the RAG-Sequence Model. The RAG-Token Model retrieves new documents on every generated token. Once a token is generated, the token is appended to all previously generated tokens and this are appended to the initial question. This results in a string of the following form "Question + GenerateTockenList". This new string is then used again as an input in the RAG, or as an input for the RAG retrieved. The process retrieves new documents for every new generated token and uses this documents to generate the next token. This makes

sure that every new generated token is equally dependent on the question and the non-parametric memory.

The RAG-Sequence model uses the same retrieved document to generate the complete sequence. Technically, it treats the retrieved document as a single latent variable that is marginalized to get the seq2seq probability p(y|x) via a top-K approximation. Concretely, the top K documents are retrieved using the retriever, and the generator produces the output sequence probability for each document, which are then marginalized

RAG-Token Model, In the RAG-Token model we can draw a different latent document for each target token and marginalize accordingly. This allows the generator to choose content from several documents when producing an answer. Concretely, the top K documents are retrieved using the retriever, and then the generator produces a distribution for the next output token for each document, before marginalizing, and repeating the process with the following output token [3]

0.7.2 Architecture

Naive RAG is the type of rag described in this document.

Advanced RAG Focuses on improving the retrieval of embeddings or data. It adds additional pre-retrieval and post-retrieval phase. In the pre-retrieve several optimization methods are employed like: * query optimization * storage optimization and retrieval * extending the stored data with metadata In the post-retrieval phase idea is to create more usable context from the retrieved data. The data now is re-ranked and compressed, cleaned up [4]

Modular RAG This type of architecture tries to split the system in to more modules that can be independently scaled and optimized. Example: Query Processing Module, Retriever, Re-ranking, Context Management, Generation Module.

Also research is done in the field of dense vector databases.

0.8 Research methodology

In this research, a simple RAG system will be coded. The system will be flexible and modular enough so can use different LLMS as generators. The system will also have a Retriever that will be able to create embeddings based on sentences and also on paragraphs. Depending on the retriever policy for every LLM three RAG systems will be created: 1. The retriever will retrieve one sentence 2. The retriever will retrieve two sentences 3. The retriever will retrieve one paragraph

All of the chosen LLMs will and their appropriate RAG will be asked the same Open-domain question. This is a type of question that can't be answered with simple yes or no and requires accessing a broad range of information to provide an answer and.

Hardware For the particular reasons an open sourced LLMs will be used that can be run locally on the pc. The chosen hardware is a m3 macbook with 36GB of RAM.

Retrieval and augmentation There will be 3 types of rag augmentation, in this text called retrieval:

- Retrieval of one sentence, will create a context for the question from one sentence using a facts sentence based database.
- Retrieval of two sentences, will create a context for the question from two sentences using a facts sentence based database.
- Retrieval of one paragraph, will create a context for the question from one paragraph using a facts paragraph based database.

The database for the sentences and paragraphs are created from two files containing semantically correct but not necessarily related sentences and paragraphs.

LLMs Depending on the LLMs training, two types of LLM models will be used:

- model for causal LLM, or casual models
- model question answering or QA models

Regarding the model size, the experiments will be done on small LLMs ranging from 66M up to 1.3B.

Model	Description	Number of Parameters	Type
GPT-2	OpenAI's Generative Model	124M	Causal Model
GPT-Neo 125M	EleutherAI's Generative Model	125M	Causal Model
GPT-Neo 1.3B	EleutherAI's Generative Model	1.3B	Causal Model
DistilBERT	Hugging Face's Optimized BERT for QA	66M	Question Answering Model
deepset/roberta-base-squad2	Facebook AI's Optimized BERT for QA	125M	Question Answering Model
bert-large-uncased-whole-word-masking-finetuned-squad	Google's NLU Model for QA	340M	Question Answering Model

For a comparison, the currently used in production chatGpt 3.5 according to chatGPT has a size of around of 175B. ChatGpt 4.o has not exposed any information about the parameters or architecture. Most probably two models employ RAG technique to improve their performance.

For the experiments the LLMs will be grouped in two sets, casual and QA models. It is important to point out the casual models are not trained for answering questions but for generating text based on input where the question answering models require a question and some context related to the question. The second are more suitable for usage in a rag system but running a rag around casual LLMs will give an idea of how powerful a rag system can be.

RAG solution The RAG system will be a manual solution, not an existing RAG system. The idea is to show that even a simple RAG system can extend the capabilities of the LLM.

Sentence embedder for sentence embedder a SBART based embedder will be used `sentence_transformers import SentenceTransformer`

Non-Parametric library For non-parametric library `faiss` will be used, it is more than enough to handle this experiment

Environment Jupyter notebook

Experiments definition

Experiment	Generators Type	Retrieval Type	Retrievals
1	Causal language models	Sentence	1
2	Causal language models	Sentence	2
3	Causal language models	Paragraph	1
4	QA language models	Sentence	1
5	QA language models	Sentence	2
6	QA language models	Paragraph	1

0.9 Results and discussion

The question 'What is the biggest city in Europe?' question is asked on list off llms and rags using different rag retrieval and augmentation techniques.

Experiment 1

- Generators type: Causal language models
- Retrieval type: Sentence
- Retreavels : 1

Model	LLM Answer	RAG Answer
gpt2	The biggest city in Europe is Berlin	The largest city in Europe is Moscow
EleutherAI/gpt-neo-125M	The city of Berlin is the most populous city in the world, and it is the most populous city in the world	The biggest city in Europe is Moscow
EleutherAI/gpt-neo-1.3B	The biggest city in Europe is London	The answer is: Moscow

In this experiment all 3 RAGs answered correctly vs all 3 LLMs answered wrongly

Experiment 2

- Generators type: Causal language models
- Retrieval type: Sentence
- Retrievals : 2

Model	LLM Answer	RAG Answer
gpt2	The biggest city in Europe is Berlin	The largest city in Europe is Berlin
EleutherAI/gpt-neo-125M	The city of Berlin is the most populous city in the world, and it is the most populous city in the world	The city of Moscow is the largest city in Europe
EleutherAI/gpt-neo-1.3B	The biggest city in Europe is London	The biggest city in Europe is Berlin, which is the capital of Germany

Only EleutherAI/gpt-neo-125M answered correctly

Experiment 3

- Generators type: Causal language models
- Retrieval type: Paragraph
- Retrievals : 1

Model	LLM Answer	RAG Answer
EleutherAI/gpt-neo-1.3B	The biggest city in Europe is London	The biggest city in Europe is London
gpt2	The biggest city in Europe is Berlin	The largest city in Europe is the capital city of Russia

Only gpt2 answered correctly

Experiment 4

- Generators type: QA language models
- Retrieval type: Sentence
- Retreavels : 1

Model	RAG Answer
distilbert-base-uncased-distilled-squad	moscow
deepset/roberta-base-squad2	What is the biggest city in Europe? Moscow
bert-large-uncased-whole-word-masking-finetuned-squad	moscow

All models answered correctly

Experiment 5

- Generators type: QA language models
- Retrieval type: Sentece
- Retrievals : 2

Model	RAG Answer
distilbert-base-uncased-distilled-squad	Moscow is the largest city in Europe by population. Moscow is the capital city of Russia
deepset/roberta-base-squad2	Moscow
bert-large-uncased-whole-word-masking-finetuned-squad	Moscow

all models answered correctly

Experiment 6

- Generators type: QA language models
- Retrieval type: Paragraph
- Retrievals : 1

Model	RAG Answer
distilbert-base-uncased-distilled-squad	Moscow
deepset/roberta-base-squad2	
bert-large-uncased-whole-word-masking-finetuned-squad	Moscow

all models answered correctly except deepset/roberta-base-squad2 did not answer

Casual LLMs Results Experiments 1, 2, and 3 used so-called casual LLMs. Despite low initial expectations, these models performed quite well. Enriched context often influenced their ability to answer questions, and they generally performed similarly. GPT-2 was able to answer questions when provided with an entire paragraph of context but struggled with a two-sentence context. EleutherAI/gpt-neo-125M answered well but has a limited input capacity of 100 characters, making it unsuitable for handling paragraphs or larger documents. EleutherAI/gpt-neo-1.3B did not perform as well, answering correctly in only one experiment. Since the test was limited to one question, it is difficult to determine the most performant model definitively. Nonetheless, it is crucial that the models can answer questions correctly most of the time. These three experiments aim to evaluate whether RAG can produce context capable of influencing the generated text. Given that the models are not specifically trained for question answering and are relatively small, the results are impressive.

QA LLMs Results Experiments 1, 2, and 3 utilized question answering LLMs. With the exception of deepset/roberta-base-squad2, which failed to produce an answer when the context was a paragraph, all models performed excellently. These models can be considered viable candidates for a production RAG implementation.

As an additional point, the question "What is the biggest city in Europe?" perhaps can be changed. The reason is that several models answered London, which might be correct depending on what they consider to be the biggest. The embeddings or the facts stored in the non-parametric model indicate that Moscow has the largest population. Repeating the experiment with a different question that has a more universally agreed-upon answer could help achieve more consistent results across models.

0.10 Conclusion

This text demonstrates through experimentation that Retrieval-Augmented Generation (RAG) systems effectively address some of the most significant challenges faced by Large Language Models (LLMs) and other models with parametric memory. Compared to models solely relying on parametric memory, RAG systems offer several advantages:

- **Horizontally Scalable**: RAG systems can efficiently scale by distributing the retrieval process across multiple nodes, allowing for handling larger datasets and more complex queries.
- **Distributive**: The modular nature of RAG systems enables distribution of tasks across different components, enhancing robustness and flexibility.
- **Mitigates High Bias**: By integrating external knowledge retrieval, RAG systems reduce the need for extremely large models with numerous parameters, thus avoiding issues related to under-fitting and high bias in therms of the real world data set, or the cross validation set The model is just not curved enough to handle real world data.
- **Performance Enhancement**: RAG systems can surpass the performance of their base generative models by leveraging external knowledge, resulting in more accurate and contextually relevant responses.

As an example, a decent quality question answering system can be created using only a 66M parameter pertained question answering model like DistilBERT when using an index like FAISS.

In summary, RAG systems provide a scalable, distributive, and efficient solution to enhance the capabilities of LLMs, addressing key limitations and improving overall performance without the necessity for excessively large and complex models.

0.11 References

1. Attention Is All You Need, Ashish Vaswani, Noam ShazeerNiki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, Illia Polosukhin, https://arxiv.org/pdf/1706.03762
2. The Faiss Library, Matthijs Douze, Alexandr Guzhva, Chengqi Deng, Jeff Johnson, Gergely Szilvasy, Pierre-Emmanuel Mazaré, Maria Lomeli, Lucas Hosseini, Hervé Jégou, https://arxiv.org/pdf/2401.08281
3. Lewis, Wen-tau Yih, Tim Rocktäschel, Sebastian Riedel, Douwe Kiela, https://arxiv.org/pdf/2005.11401v4
4. Retrieval-Augmented Generation for Large Language Models: A Survey, Yunfan Gao, Yun Xiong, Xinyu Gao, Kangxiang Jia, Jinliu Pan, Yuxi Bi, Yi Dai, Jiawei Sun, Qianyu Guo, Meng Wang, Haofen Wang, https://arxiv.org/pdf/2312.10997
5. Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks, Nils Reimers, Iryna Gurevych, https://arxiv.org/pdf/1908.10084
6. https://www.datastax.com/guides/what-is-retrieval-augmented-generation
7. https://nexocode.com/blog/posts/retrieval-augmented-generation-rag-llms
8. https://implementconsultinggroup.com/article/building-high-quality-rag-systems

0.12 Rag Implementation

[ ]:
```
# Prepare requered libraries for
%conda install -y numpy
# Retrieval and storage requered libraries
%conda install -y sentence-transformers
%conda install -y faiss-cpu
# LLMs related libraryes
%conda install -y transformers
%conda install -y pytorch
%conda install -y ipywidgets
# %conda install texlive-core
%jupyter nbextension uninstall --all
%jupyter nbextension install --all
%jupyter nbextension enable --all
```

[ ]:
```
```

[ ]:
```
# smoke test
import torch
print(torch.__version__)
print(torch.backends.mps.is_available())
```

[ ]:
```
# LLM related libs
import torch
from transformers import AutoModelForCausalLM, AutoTokenizer,␣
 ↪PreTrainedTokenizer, AutoModelForQuestionAnswering

# Retrieval and Storage Related libs
```

```python
from sentence_transformers import SentenceTransformer
import numpy as np
import faiss

import re
import json


def read_and_split_sentences(file_path)->list[str]:
    with open(file_path, 'r') as file:
        text = file.read()
    sentences = text.strip().split('.')
    return sentences

def read_and_split_paragraphs(file_path) -> list[str]:
    with open(file_path, 'r') as file:
        text = file.read()
    paragraphs = text.strip().split('\n\n')
    return paragraphs

def save_dict_to_json_file(data, file_name):
    with open(file_name, 'w') as json_file:
        json.dump(data, json_file, indent=4)

def get_answers(sentence_retreaver, device, question, llms):
    """
    Generates answers using different LLMs and returns the results.

    :param sentence_retreaver: Instance of Retreaver with added documents
    :param device: Torch device (e.g., 'cpu', 'cuda', 'mps')
    :param question: The question to be answered
    :param llms: List of LLMs to generate answers
    :return: List of results with answers from each LLM
    """
    results = []

    for llm in llms:
        generator = Generator(model_name_in=llm, device_in=device)
        single_result = {}
        single_result["llm"] = llm
        single_result["question"] = question
        single_result["LLM Answer"] = generator.generate_answer(question)
        single_result["RAG Answer"] = generator.generate_answer(question,␣
↪context=sentence_retreaver.retrieve_documents(question, 2))
        results.append(single_result)

    return results
```

[ ]:
```
class Generator:

    # Causal language models
    # gpt_j_6B = "EleutherAI/gpt-j-6B"
    gpt_2 = "gpt2"
    gpt_neo_125M = "EleutherAI/gpt-neo-125M"
    gpt_neo_1_3B = "EleutherAI/gpt-neo-1.3B"
    # Question answering models
    distilbert = "distilbert-base-uncased-distilled-squad"
    roberta = "deepset/roberta-base-squad2"
    bert = "bert-large-uncased-whole-word-masking-finetuned-squad"

    casual_llms = [gpt_2, gpt_neo_125M, gpt_neo_1_3B]
    qa_llms = [distilbert, roberta, bert]


    def __init__(self, model_name_in: str, device_in):
        self.device = device_in
        self.model_name = model_name_in
        self.model, self.tokenizer = self.get_model(self.model_name, self.
↪device)

    def get_model(self, model_name_in: str, device_in):
        if model_name_in in self.casual_llms:
            _model = AutoModelForCausalLM.from_pretrained(model_name_in).
↪to(device_in)
        else:
            _model = AutoModelForQuestionAnswering.
↪from_pretrained(model_name_in).to(device_in)
        _tokenizer = AutoTokenizer.from_pretrained(model_name_in)
        return _model, _tokenizer

    def generate_answer(self, question: str, context=None):
        if self.model_name in self.casual_llms:
            return self.generate_answer_causal(question, context)
        else:
            return self.generate_answer_qa(question, context)

    def generate_answer_causal(self, question: str, context_in=None,␣
↪max_length=100):
        _context = None
        if isinstance(context_in, list):
            _context = " and ".join(context_in)
            _context = _context.lstrip()
            _context = _context.rstrip()
            if(not _context.endswith((',','.','?','!'))):
                _context = _context + '.'
```

```
        _input_text = question if not context_in else "Given, " + _context + "␣
↪" + question +""
        _inputs = self.tokenizer.encode_plus(_input_text, return_tensors="pt")
        _inputs = _inputs.to(self.device)
        # Disable gradient calculation for faster inference
        with torch.no_grad():
            _outputs = self.model.generate(
                input_ids=_inputs['input_ids'],
                attention_mask=_inputs.get('attention_mask'),
                max_length=max_length,
                pad_token_id=self.tokenizer.eos_token_id
            )
        _response = self.tokenizer.decode(_outputs[0][_inputs['input_ids'].
↪size(1):], skip_special_tokens=False)
        _response = _response.lstrip('\n')
        _response = _response.lstrip('A:')
        _response_array = re.split(r' *[\.\?!][\'"\)\]]* *',_response.
↪replace('\n',' ').strip())
        _one_sentece_response =_response_array[0]
        return _one_sentece_response

    def generate_answer_qa(self, question_in: str, context_in=None):
        if context_in is None:
            context_in = ""
        if isinstance(context_in, list):
            context_in = " ".join(context_in)
        inputs = self.tokenizer.encode_plus(question_in, context_in,␣
↪add_special_tokens=True, return_tensors="pt").to(self.device)
        with torch.no_grad():
            outputs = self.model(**inputs)
        answer_start = torch.argmax(outputs.start_logits)  # Get the most␣
↪likely beginning of answer
        answer_end = torch.argmax(outputs.end_logits) + 1  # Get the most␣
↪likely end of answer

        answer = self.tokenizer.convert_tokens_to_string(self.tokenizer.
↪convert_ids_to_tokens(inputs["input_ids"][0][answer_start:answer_end]))
        return answer
```

[ ]:
```python
class Retriever:
    def __init__(self, model_name_in='all-MiniLM-L6-v2'):
        self.documents = []
        self.model = SentenceTransformer(model_name_in)
        self.document_embeddings = None
        # dimension is 384 for the 'all-MiniLM-L6-v2'
        d = self.model.get_sentence_embedding_dimension()
        self.index = faiss.IndexFlatL2(d)

    def add_documents(self, new_documents_in: list[str]):
        # MaximumCharacters  512tokens×4characters/token=2048characters
        new_document_embeddings = self.model.encode(new_documents_in).astype(np.
 ↪float32)
        self.index.add(new_document_embeddings)
        self.documents.extend(new_documents_in)

    def retrieve_documents(self, query_in, results_size_in=1):
        query_embedding = self.model.encode([query_in]).astype(np.float32)
        distances, indices = self.index.search(query_embedding, results_size_in)
        retrieved_documents = [self.documents[idx] for idx in indices[0]]
        return retrieved_documents
```

[ ]:
```python
#### Experiment 1
# Simple Question experiment
# Generators type: Causal language models
# Retrieval type: Sentence retrieval
# Retrievals : 1

sentence_retriever = Retriever()
sentence_retriever.add_documents(read_and_split_sentences("facts_sentences"))

device = torch.device("mps" if torch.backends.mps.is_available() else "cpu")
question = "What is the biggest city in Europe?"
results = []

for llm in Generator.casual_llms:
    generator = Generator(model_name_in=llm, device_in=device)
    single_result = {}
    single_result["llm"] = llm
    single_result["question"] = question
    single_result["LLM Answer"] = generator.generate_answer(question)
    single_result["RAG Answer"] = generator.generate_answer(question, context =␣
 ↪sentence_retriever.retrieve_documents(question,1))
    results.append(single_result)

save_dict_to_json_file(results, "experiments/
 ↪CasualModelsSimpleQuestion1SentenceRetriever.json")
```

[ ]:

[ ]:
```
#### Experiment 2
# Simple Question experiment
# Generators type: Causal language models
# Retrieval type: Sentence
# Retrievals : 2

sentence_retriever = Retriever()
sentence_retriever.add_documents(read_and_split_sentences("facts_sentences"))

device = torch.device("mps" if torch.backends.mps.is_available() else "cpu")
question = "What is the biggest city in Europe?"
results = []

for llm in Generator.casual_llms:
    generator = Generator(model_name_in=llm, device_in=device)
    single_result = {}
    single_result["llm"] = llm
    single_result["question"] = question
    single_result["LLM Answer"] = generator.generate_answer(question)
    single_result["RAG Answer"] = generator.generate_answer(question, context =␣
 ↪sentence_retriever.retrieve_documents(question,2))
    results.append(single_result)

save_dict_to_json_file(results, "experiments/
 ↪CasualModelsSimpleQuestion2SentenceRetrieval.json")
```

[ ]:
```
#### Experiment 3
# Simple Question experiment
# Generators type: Causal language models
# Retrieval type: Paragraph
# Retrievals : 1

sentence_retriever = Retriever()
sentence_retriever.add_documents(read_and_split_paragraphs("facts_paragraphs"))

device = torch.device("mps" if torch.backends.mps.is_available() else "cpu")
question = "What is the biggest city in Europe?"

results = []

for llm in [Generator.gpt_neo_1_3B, Generator.gpt_2]:
    print(f"Processing {llm}")
    generator = Generator(model_name_in=llm, device_in=device)
    single_result = {}
    single_result["llm"] = llm
```

```
    single_result["question"] = question
    single_result["LLM Answer"] = generator.generate_answer(question)
    single_result["RAG Answer"] = generator.generate_answer(question, context =␣
 ↪sentence_retriever.retrieve_documents(question,1))
    results.append(single_result)


save_dict_to_json_file(results, "experiments/
 ↪CasualModelsSimpleQuestionParagraphRetrieval.json")
```

[ ]:
```
#### Experiment 4
# Simple Question experiment
# Generators type: QA language models
# Retrieval type: Sentence
# Retrievals : 1

sentence_retriever = Retriever()
sentence_retriever.add_documents(read_and_split_sentences("facts_sentences"))

device = torch.device("mps" if torch.backends.mps.is_available() else "cpu")

question = "What is the biggest city in Europe?"

results = []
for llm in Generator.qa_llms:
    generator = Generator(model_name_in=llm, device_in=device)
    single_result = {}
    single_result["llm"] = llm
    single_result["question"] = question
    single_result["RAG Answer"] = generator.generate_answer(question, context =␣
 ↪sentence_retriever.retrieve_documents(question,1))
    results.append(single_result)

save_dict_to_json_file(results, "experiments/
 ↪QAModelsSimpleQuestionOneSentenceRetrieval.json")
```

[ ]:
```
#### Experiment 5
# Simple Question experiment
# Generators type: QA language models
# Retrieval type: Sentence
# Retrievals : 2

sentence_retreaver = Retriever()
sentence_retreaver.add_documents(read_and_split_sentences("facts_sentences"))

device = torch.device("mps" if torch.backends.mps.is_available() else "cpu")

question = "What is the biggest city in Europe?"

results = []
for llm in Generator.qa_llms:
    generator = Generator(model_name_in=llm, device_in=device)
    single_result = {}
    single_result["llm"] = llm
    single_result["question"] = question
    single_result["RAG Answer"] = generator.generate_answer(question, context =␣
 ↪sentence_retreaver.retrieve_documents(question,2))
    results.append(single_result)

save_dict_to_json_file(results, "experiments/
 ↪QAModelsSimpleQuestionTwoSentenceRetrieval.json")
```

[ ]:
```
#### Experiment 6
# Simple Question experiment
# Generators type: QA language models
# Retrieval type: Paragraph
# Retrievals : 1

sentence_retriever = Retriever()
sentence_retriever.add_documents(read_and_split_paragraphs("facts_paragraphs"))

device = torch.device("mps" if torch.backends.mps.is_available() else "cpu")

question = "What is the biggest city in Europe?"

results = []
for llm in Generator.qa_llms:
    generator = Generator(model_name_in=llm, device_in=device)
    single_result = {}
    single_result["llm"] = llm
    single_result["question"] = question
    single_result["RAG Answer"] = generator.generate_answer(question, context =␣
 ↪sentence_retriever.retrieve_documents(question,1))
```

```
    results.append(single_result)

save_dict_to_json_file(results, "experiments/
  ↪QAModelsSimpleQuestionParagraphRetrieval.json")
```