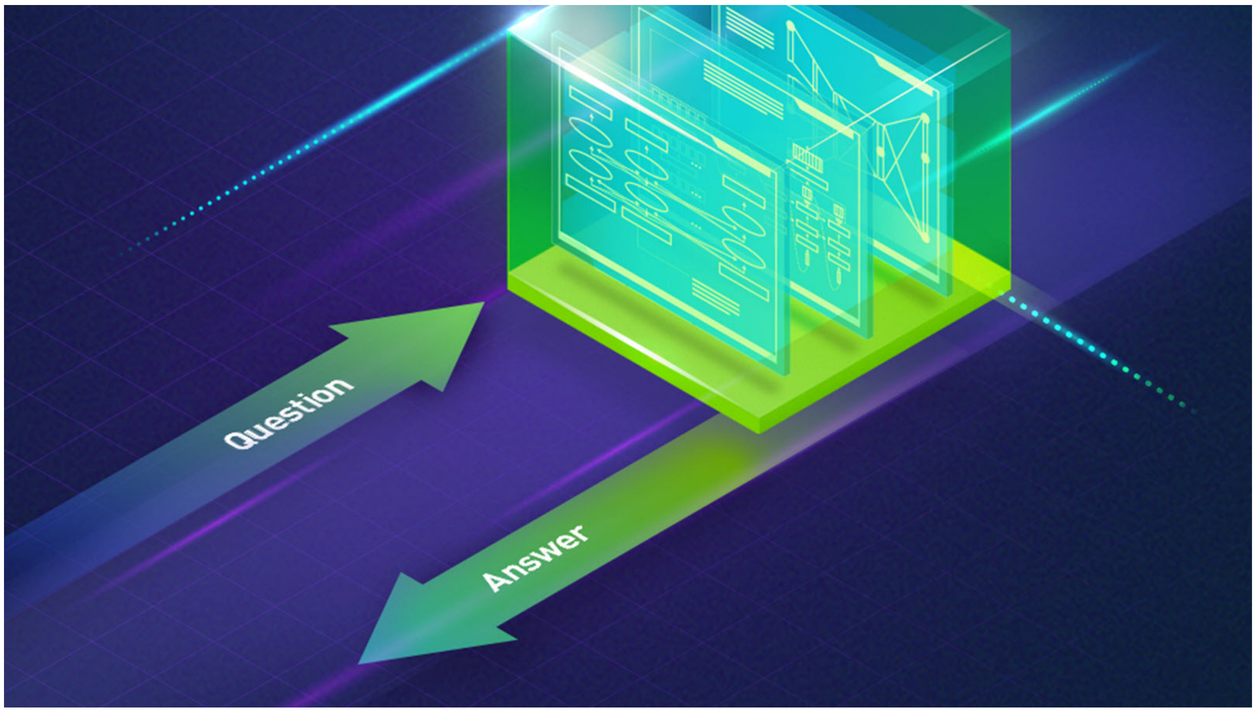


NVIDIA Q&A LLM

01.02.2024

Bhaskar Boruah
boruah.bhaskar@gmail.com

Problem Statement

Problem-01: AI-Assisted Learning for NVIDIA SDKs and Toolkits

The objective is to develop an AI-powered Language Model (LLM) that assists users in understanding and effectively using NVIDIA SDKs and toolkits. The envisioned platform will serve as an interactive and user-friendly hub, providing comprehensive information, examples, and guidance on NVIDIA's SDKs and toolkits. By leveraging the power of language models and NVIDIA's cutting-edge technologies, the aim is to simplify the learning curve for developers and empower them to utilize NVIDIA's technologies more efficiently.

Goals

- Develop an AI-powered Language Model (LLM) that assists users in understanding and effectively using NVIDIA SDKs and toolkits

Overview

A highly effective use case for LLMs is the development of intelligent question-and-answer (Q&A) chatbots. These are programs that provide answers to inquiries concerning particular source data. The method utilized in these applications is called **Retrieval Augmented Generation**, or **RAG**.RAG is a method for adding new data to enhance LLM knowledge.Retrieval Augmented Generation (RAG) is the process of bringing the relevant data and inserting it into the model prompt.

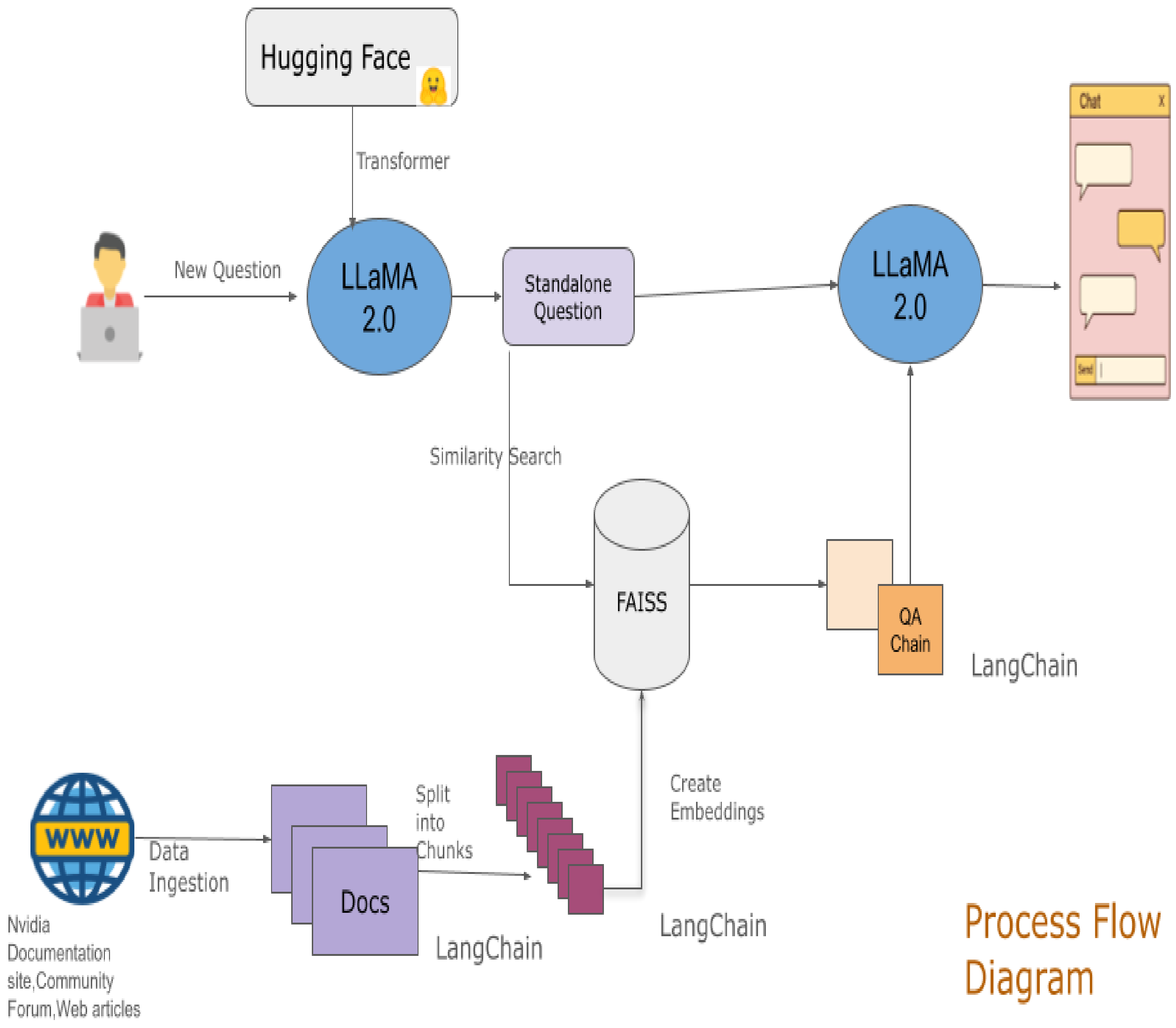
We will explore how we can use the open source **Llama-2-7b-chat** model in both **Hugging Face transformers** and **LangChain** framework with **FAISS library** vector store over the documents that are fetched online from the Nvidia documentation website and other pertinent articles and tutorials from the internet and Community Forums and Q&A Platforms like NVIDIA Developer Forums in order to develop an AI-powered Language Model (LLM) that helps users understand and use NVIDIA SDKs and toolkits effectively.

RAG Architecture - A typical RAG application has two main components:

Indexing: a pipeline for ingesting data from a source and indexing it. This usually happens offline.

Retrieval and generation: the actual RAG chain, which takes the user query at run time and retrieves the relevant data from the index, then passes that to the model.

Process flow diagram :



Prerequisite :

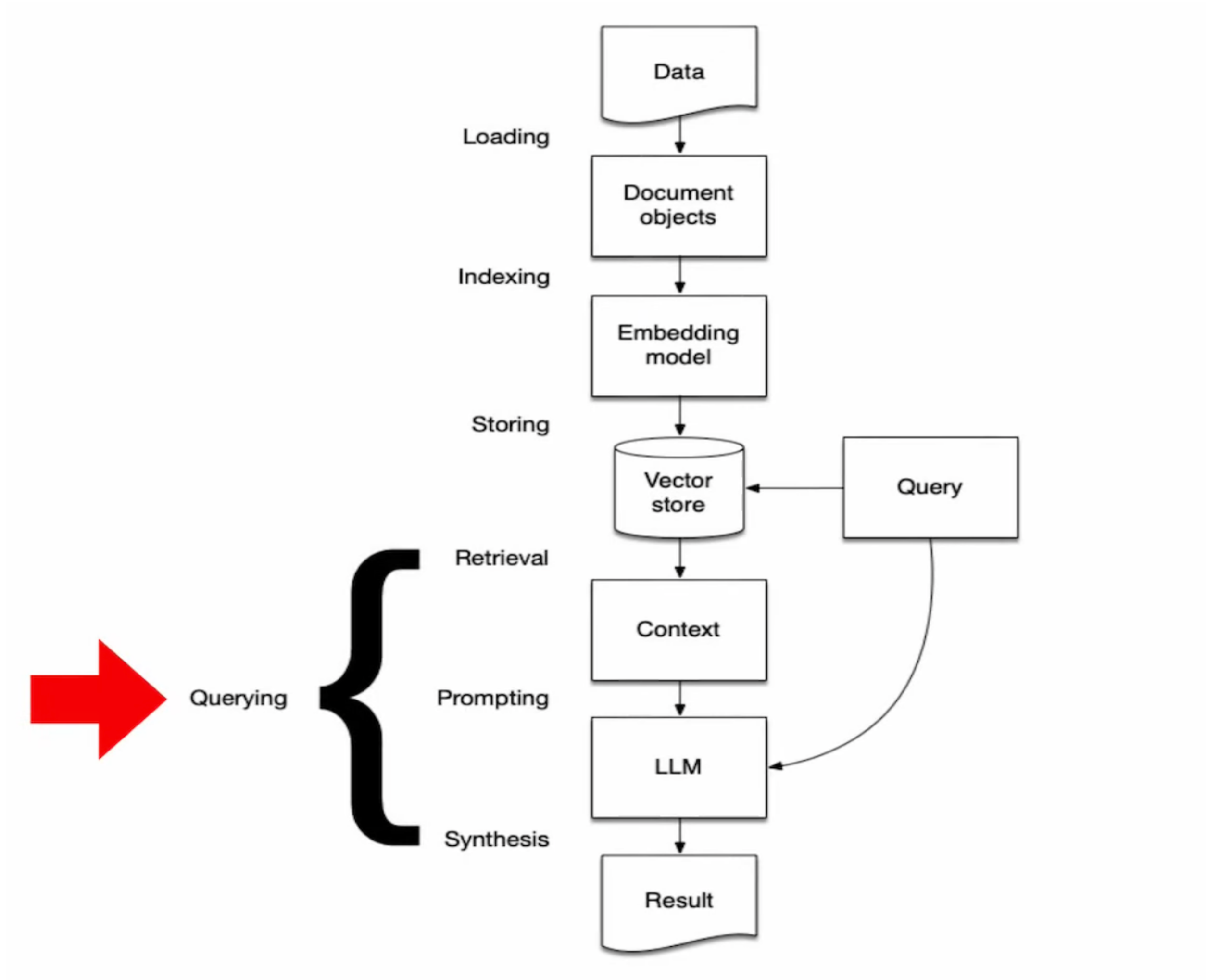
1. To avoid slowness, Change the runtime type in Google Colab as below :

```
Runtime > Change runtime type > Hardware accelerator > T4 GPU
```

2. Generate an access token to allow downloading the model from Hugging Face in your code. **For that, go to your Hugging Face Profile > Settings > Access Token > New Token > Generate a Token.** Just copy the token and add it in the below code to Implement the Hugging Face pipeline in LangChain.

```python
from torch import cuda, bfloat16
import transformers

model_id = 'meta-llama/Llama-2-7b-chat-hf'

device = f'cuda:{cuda.current_device()}' if cuda.is_available() else 'cpu'

# set quantization configuration to load large model with less GPU memory
# this requires the `bitsandbytes` library
bnb_config = transformers.BitsAndBytesConfig(
   load_in_4bit=True,
   bnb_4bit_quant_type='nf4',
   bnb_4bit_use_double_quant=True,
   bnb_4bit_compute_dtype=bfloat16
)

# begin initializing HF items, you need an access token
hf_auth = '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'
model_config = transformers.AutoConfig.from_pretrained(
   model_id,
   use_auth_token=hf_auth
)

model = transformers.AutoModelForCausalLM.from_pretrained(
   model_id,
   trust_remote_code=True,
   config=model_config,
   quantization_config=bnb_config,
   device_map='auto',
    use_auth_token=hf_auth
)


# enable evaluation mode to allow model inference
model.eval()


print(f"Model loaded on {device}")
```

Detailed Steps:



Indexing

1. Load: First we need to load our data. This is done with DocumentLoaders.

- Data Collection : We will prepare a function get_all_links which sends a request to the specified URL, parses the HTML content using BeautifulSoup, and then extracts and returns all the URLs found in the anchor () tags on the page.

- Data Preprocessing : By using the links , we will ingest data using WebBaseLoader document loader which collects data by scraping web pages.

2. Split: Text splitters break large Documents into smaller chunks. This is useful both for indexing data and for passing it into a model, since large chunks are harder to search over and won't fit in a model's finite context window.

- Split the text into small pieces. We will need to initialize RecursiveCharacterTextSplitter and call it by passing the documents.

```
from langchain.text_splitter import RecursiveCharacterTextSplitter

text_splitter=RecursiveCharacterTextSplitter(chunk_size=500,
chunk_overlap=20)

all_splits = text_splitter.split_documents(documents)
```

3. Create embeddings: converting the chunks of text into numerical values, also known as embeddings. These embeddings are used to search and retrieve similar or relevant documents quickly in large databases, as they represent the semantic meaning of the text.We have to create embeddings for each small chunk of text and store them in the vector store FAISS (i.e. Facebook AI Similarity Search). We will be using all-mpnet-base-v2 Sentence Transformer to convert all pieces of text in vectors while storing them in the vector store.

4. Load embeddings into vector stores: We need somewhere to store and index our splits, so that they can later be searched over. This is often done using a VectorStore and Embeddings model.loading the embeddings into a vector store i.e. "FAISS" in this case. Vector stores perform extremely well in similarity search using text embeddings compared to the traditional databases.

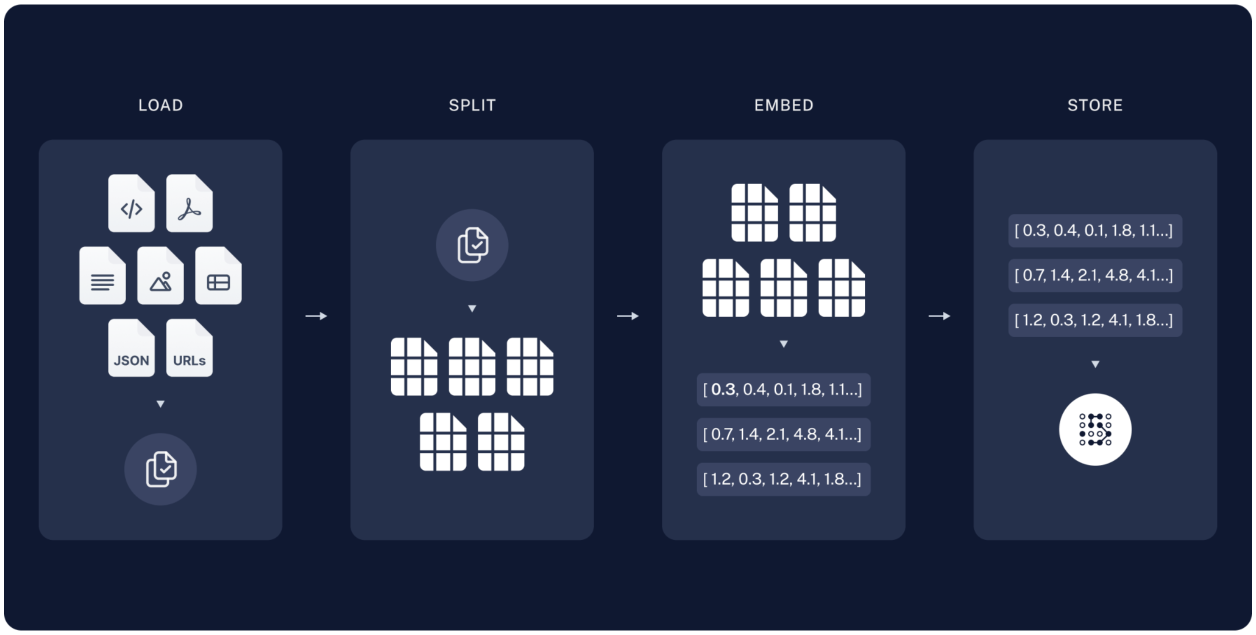
```
from langchain.embeddings import HuggingFaceEmbeddings

from langchain.vectorstores import FAISS

model_name = "sentence-transformers/all-mpnet-base-v2"

model_kwargs = {"device": "cuda"}

embeddings=HuggingFaceEmbeddings(model_name=model_name,
model_kwargs=model_kwargs)

# storing embeddings in the vector store

vectorstore = FAISS.from_documents(all_splits, embeddings)
```

- Save the embedding data to a local location where the vector store is saved.

```
vs=vectorstore.save_local(folder_path='/content/vs_data')
```

- Load the vector store data from the local file -

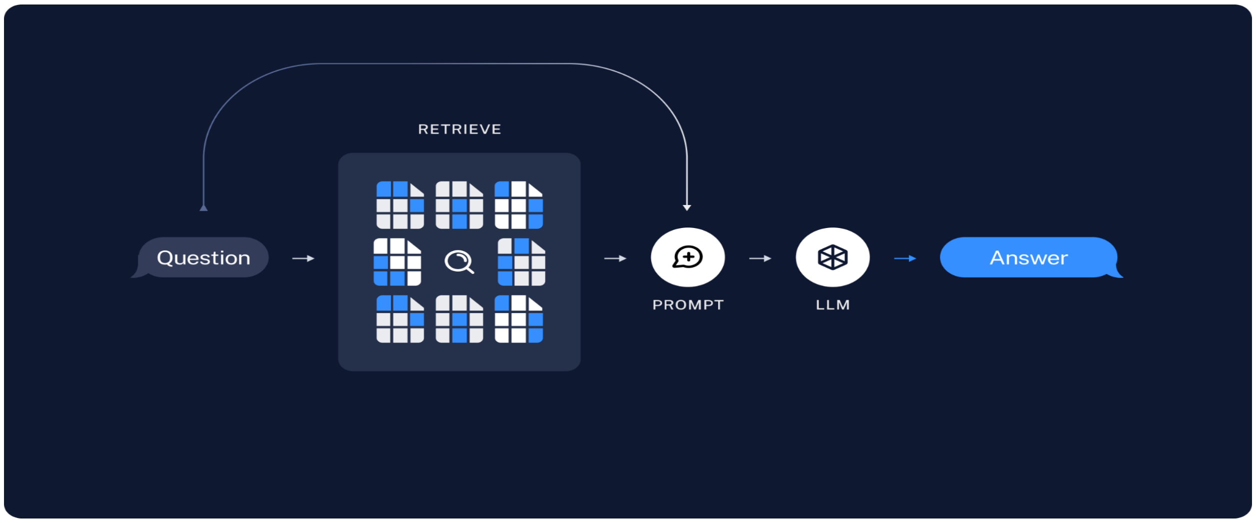
```
vectorstore_data = vectorstore.load_local('/content/vs_data/',embeddings)
```



Retrieval and generation

5. Retrieve: Given a user input or question, relevant splits are retrieved from storage using the embeddings and similarity search technique such as cosine similarity or Euclidean distance . Since we are using Langchain and FAISS , these will be taken care of by the model itself.

6. Generate: A ChatModel / LLM produces an answer using a prompt that includes the question and the retrieved data. We will use **Llama-2-7b-chat** model for generating response



Initializing Chain

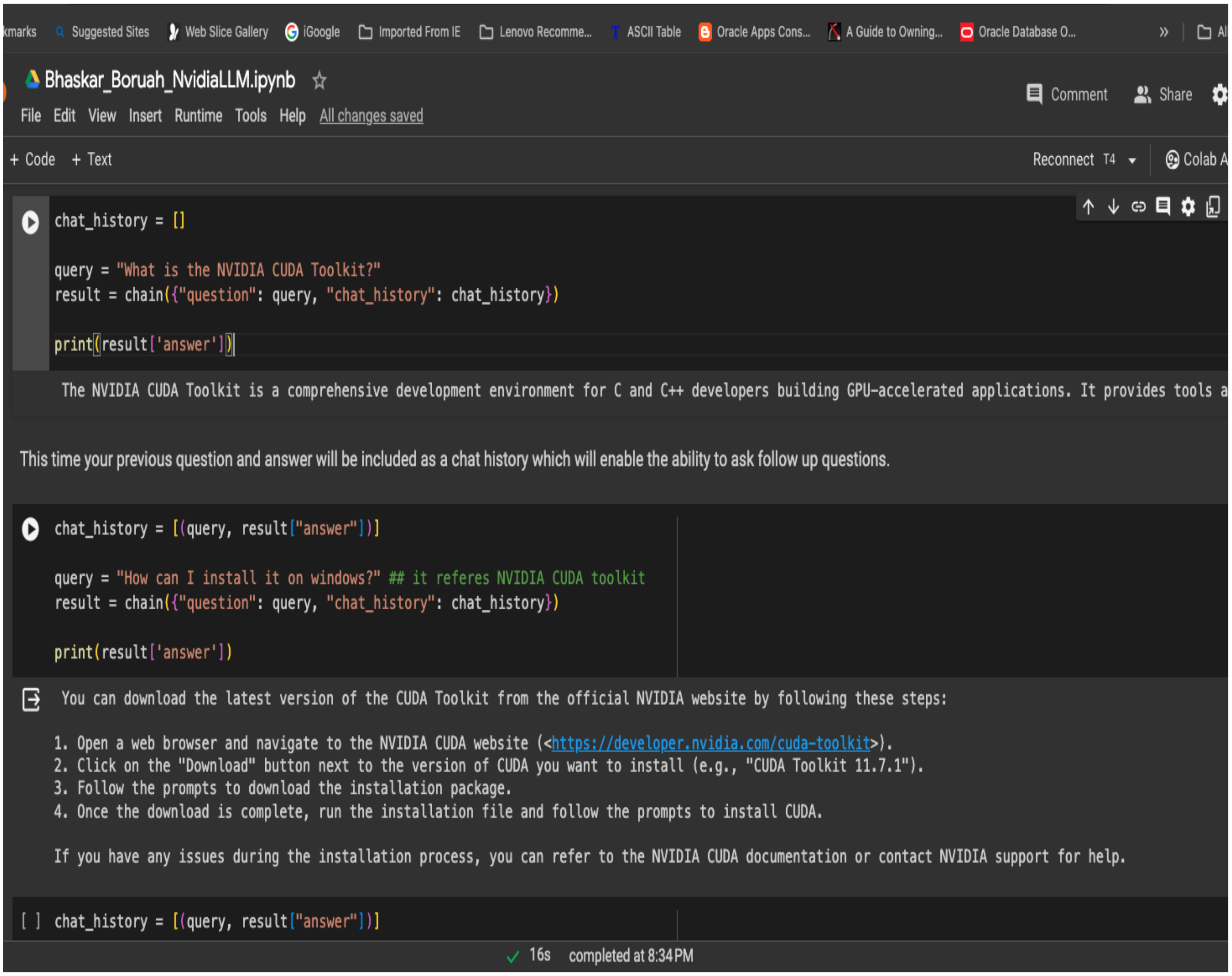
We have to initialize the ConversationalRetrievalChain. This chain allows us to have a chatbot with memory while relying on a vector store to find relevant information from your document.Additionally, we can return the source documents used to answer the question by specifying an optional parameter i.e. return_source_documents=True when constructing the chain.

```
from langchain.chains import ConversationalRetrievalChain

chain = ConversationalRetrievalChain.from_llm(llm,
vectorstore_data.as_retriever(), return_source_documents=True)
```

Question-Answering

Searching for the relevant information stored in the vector store using the embeddings.passing the standalone question and the relevant information to the question-answering chain where the language model is used to generate an answer.

Example of some NVIDIA question and answer generated by the model is given below -


Bhaskar_Boruah_NvidiaLLM.ipynb
File Edit View Insert Runtime Tools Help All changes saved
+ Code + Text
Reconnect T4
Comment Share

```
chat_history = []

query = "What is the NVIDIA CUDA Toolkit?"
result = chain({"question": query, "chat_history": chat_history})

print(result['answer'])
```

```
The NVIDIA CUDA Toolkit is a comprehensive development environment for C and C++ developers building GPU-accelerated applications. It provides tools a
```

This time your previous question and answer will be included as a chat history which will enable the ability to ask follow up questions.

```
chat_history = [(query, result["answer"])]

query = "How can I install it on windows?" ## it referes NVIDIA CUDA toolkit
result = chain({"question": query, "chat_history": chat_history})

print(result['answer'])
```

```
You can download the latest version of the CUDA Toolkit from the official NVIDIA website by following these steps:

1. Open a web browser and navigate to the NVIDIA CUDA website (<https://developer.nvidia.com/cuda-toolkit>).
2. Click on the "Download" button next to the version of CUDA you want to install (e.g., "CUDA Toolkit 11.7.1").
3. Follow the prompts to download the installation package.
4. Once the download is complete, run the installation file and follow the prompts to install CUDA.

If you have any issues during the installation process, you can refer to the NVIDIA CUDA documentation or contact NVIDIA support for help.
```

```
chat_history = [(query, result["answer"])]
```

✓ 16s completed at 8:34 PM

Conclusion : The model has the capability to answer questions based on NVIDIA data. We may need to further work on ingesting more Nvidia data as well as we can develop a chatbot application.

Reference :

[1] https://huggingface.co/blog/llama2

[2] https://venturebeat.com/ai/llama-2-how-to-access-and-use-metas-versatile-open-source-chatbot-right-now/

[3] https://www.pinecone.io/learn/series/langchain/langchain-intro/

[4] https://engineering.fb.com/2017/03/29/data-infrastructure/faiss-a-library-for-efficient-similarity-search/

[5] https://ai.meta.com/tools/faiss/

[6] https://blog.bytebytego.com/p/how-to-build-a-smart-chatbot-in-10

[7] https://newsletter.theaiedge.io/p/deep-dive-building-a-smart-chatbot

[8] https://www.youtube.com/watch?v=6iHVJyX2e50

[9]https://github.com/pinecone-io/examples/blob/master/learn/generation/llm-field-guide/llama-2/llama-2-70b-chat-agent.ipynb