Development of a Conversational Assistant (Chatbot) using an LLM for Data Extraction

H.Yahya M.Youssef

September 22, 2024

Abstract

The development of conversational assistants, leveraging Large Language Models (LLMs), represents a significant advance in human-computer interaction, enabling more natural and context-aware communication. LLMs not only enhance the capabilities of chatbots but also facilitate efficient data extraction by comprehending and processing complex textual information across diverse domains. However, the deployment of LLMs, particularly during initial training and fine-tuning phases, incurs substantial costs. These include energy consumption, as well as hardware demands such as storage, processors, and network resources.

This paper explores the intersection of LLM-powered chatbot development and data extraction using different methods, while critically evaluating the energy that demands. To provide practical insights, we will thus present a straightforward approach to estimate the energy consumption of an NVIDIA GPU during both the training and inference phases of a relatively simple neural network model.

Contents

1 Introduction

In recent years, conversational assistants powered by Large Language Models (LLMs) have transformed the way humans interact with computers, offering more seamless and contextually rich communication. By harnessing the advanced capabilities of LLMs, chatbots have evolved to handle complex textual data and extract valuable information across various domains. Despite these advancements, the widespread deployment of LLMs introduces significant challenges, particularly in terms of the costs associated with their initial training and fine-tuning processes. These costs include not only the computational power required but also the substantial energy consumption and the demands placed on hardware resources such as storage, processors, and networking infrastructure. For further information about how Large Language Models Are Transforming Conversational AI, click here.

2 How to Build a Chatbot

Let us talk about LlamaIndex[1].
LlamaIndex acts as a link between your data and Large Language Models (LLMs), offering a comprehensive toolkit to create a query interface for tasks like question-answering and summarization.

In this paper, we will demonstrate how to build a context-enhanced chatbot using a Data Agent. This agent, driven by LLMs, can perform tasks intelligently across your data. The final product is a chatbot equipped with powerful data interface tools from LlamaIndex, designed to handle queries related to your data.

2.1 Frameworks installation:

We will be using OpenAI's APIs and models as well as LlamaIndex's service (We will discuss later on the insertion of other LLM models from HuggingFace as an example) :

```
!pip install -q llama-index
!pip install -q openai
!pip install -q transformers
!pip install -q accelerate
!pip install llama-index-llms-openai
!pip install llama-index-readers-file
!pip install llama-index-embeddings-openai
```

For Frameworks explanation, please refer to section I.

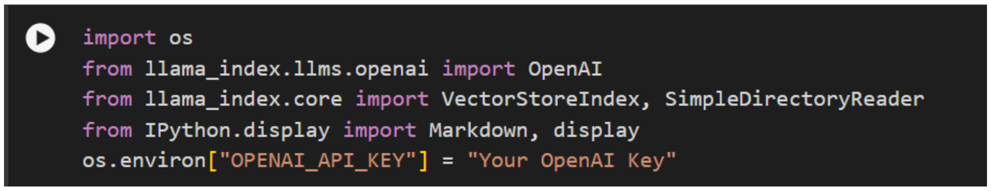
2.2 The core structure of the chatbot:

This tutorial[2] provides a step-by-step explanation. We recommend following along by creating a document in Google Colab to facilitate the process.

[1]LlamaIndex (formerly known as GPT Index) is a framework designed to bridge the gap between data and Large Language Models

[2]An OpenAI API key is required. For guidance on obtaining one, refer to this video tutorial.

We start by importing necessary packages and loading our API key :

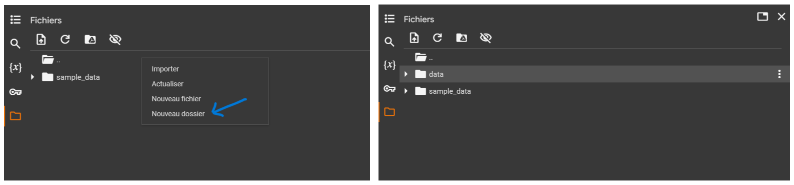


```python
import os
from llama_index.llms.openai import OpenAI
from llama_index.core import VectorStoreIndex, SimpleDirectoryReader
from IPython.display import Markdown, display
os.environ["OPENAI_API_KEY"] = "Your OpenAI Key"
```

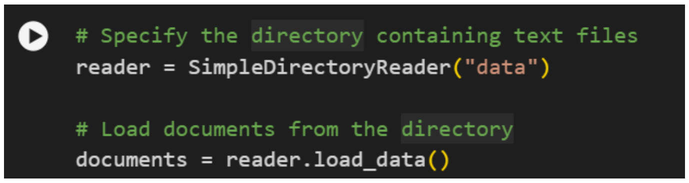
Figure 1: Useful packages

We will then use the powerful SimpleDirectoryReader[3] reader offered by LlamaIndex. This class is used for reading and processing multiple text files from a directory, which can then be indexed or analyzed by language models or other algorithms.

We begin by creating a folder named **"data"**, which will serve as the repository for the files used to communicate with the chatbot :



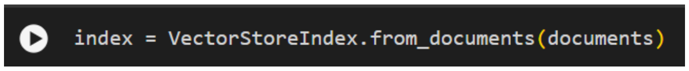
Once created, we can directly load data using our function :



```python
# Specify the directory containing text files
reader = SimpleDirectoryReader("data")

# Load documents from the directory
documents = reader.load_data()
```

Figure 2: Useful packages

Now, we move on to the most crucial part: **indexation**. Now that we loaded data in the variable `documents` we want to create an index to perform efficient keyword searches or similarity searches. The `VectorStoreIndex.from_documents(documents)` method helps by creating an index that organizes and optimizes these documents for quick retrieval. It is used as follows :

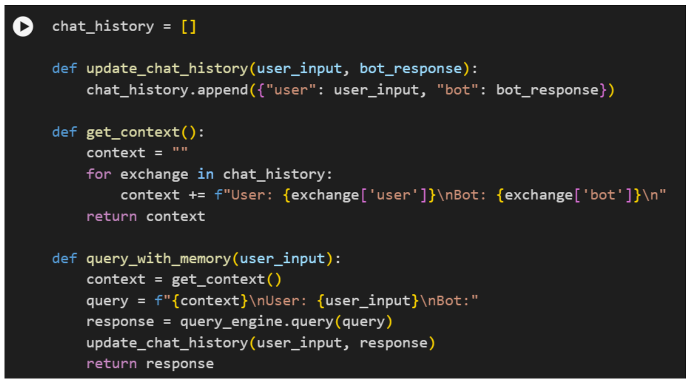


```python
index = VectorStoreIndex.from_documents(documents)
```

Figure 3: Indexation:

[3]We do note use predefined function in python for reading files because we will index loaded data.

We are now prepared to build the chatbot. To do so, we will implement the following functions:



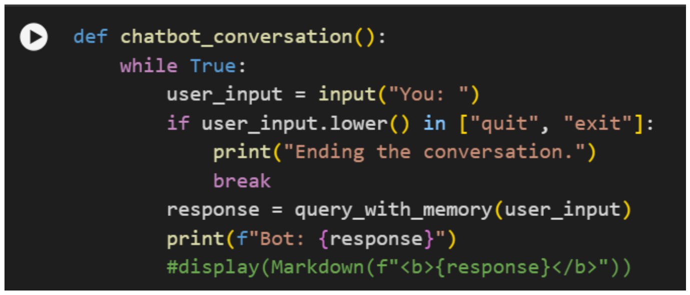
```
chat_history = []

def update_chat_history(user_input, bot_response):
    chat_history.append({"user": user_input, "bot": bot_response})

def get_context():
    context = ""
    for exchange in chat_history:
        context += f"User: {exchange['user']}\nBot: {exchange['bot']}\n"
    return context

def query_with_memory(user_input):
    context = get_context()
    query = f"{context}\nUser: {user_input}\nBot:"
    response = query_engine.query(query)
    update_chat_history(user_input, response)
    return response
```

Figure 4: Useful functions:

For detailed information about each function, go to section II.

As demonstrated, we have initialized a variable `chat_history = []` to maintain an ongoing conversation with the chatbot, thereby enabling a coherent and continuous interaction.

Now, let's define the `chatbot_conversation` function :



```
def chatbot_conversation():
    while True:
        user_input = input("You: ")
        if user_input.lower() in ["quit", "exit"]:
            print("Ending the conversation.")
            break
        response = query_with_memory(user_input)
        print(f"Bot: {response}")
        #display(Markdown(f"<b>{response}</b>"))
```

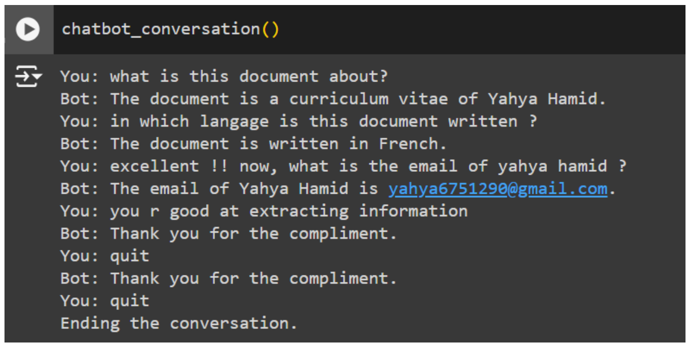
Figure 5: chatbot conversation function :

To start conversation we simply run the code discussed previously, and execute the following line : `chatbot_conversation()`

Note : Please ensure that the files you intend to use are placed in the folder named data before executing the code[4].

[4]In this tutorial, we have chosen this method for simplicity. However, it is also possible to download a file from a specified URL, save it to a designated directory, and rename it. Refer to section III for for detailed instructions.

Here is an example of such a conversation :



```
chatbot_conversation()

You: what is this document about?
Bot: The document is a curriculum vitae of Yahya Hamid.
You: in which langage is this document written ?
Bot: The document is written in French.
You: excellent !! now, what is the email of yahya hamid ?
Bot: The email of Yahya Hamid is yahya6751290@gmail.com.
You: you r good at extracting information
Bot: Thank you for the compliment.
You: quit
Bot: Thank you for the compliment.
You: quit
Ending the conversation.
```

Figure 6: chatbot_conversation function :

In this tutorial we built a simple chatbot that can answer questions about files uploaded. As we previously mentioned, the main component is the index method. However, we used the default one, how can we customize it ? Are there other methods for indexing ? What if we don't want to use OpenAI service?

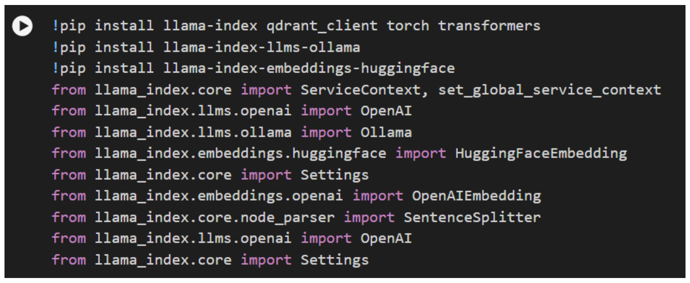
The following discussion will answer these questions.

3 Improvement :

In this section we will be presenting additional methods to extract data using our chatbot.

3.1 Customizing index method

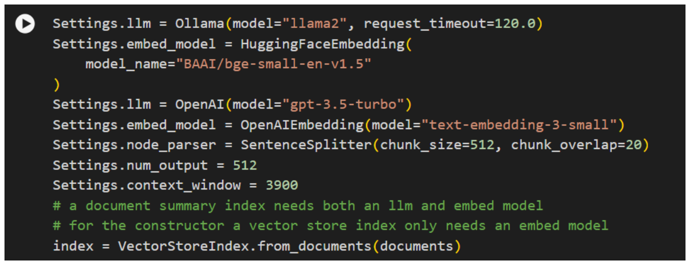
To customize index method we will need to import some packages :



```
!pip install llama-index qdrant_client torch transformers
!pip install llama-index-llms-ollama
!pip install llama-index-embeddings-huggingface
from llama_index.core import ServiceContext, set_global_service_context
from llama_index.llms.openai import OpenAI
from llama_index.llms.ollama import Ollama
from llama_index.embeddings.huggingface import HuggingFaceEmbedding
from llama_index.core import Settings
from llama_index.embeddings.openai import OpenAIEmbedding
from llama_index.core.node_parser import SentenceSplitter
from llama_index.llms.openai import OpenAI
from llama_index.core import Settings
```

Figure 7: Useful packages :

We can then customize our index method the way we want.
Here is an example of such a customization :



```
Settings.llm = Ollama(model="llama2", request_timeout=120.0)
Settings.embed_model = HuggingFaceEmbedding(
    model_name="BAAI/bge-small-en-v1.5"
)
Settings.llm = OpenAI(model="gpt-3.5-turbo")
Settings.embed_model = OpenAIEmbedding(model="text-embedding-3-small")
Settings.node_parser = SentenceSplitter(chunk_size=512, chunk_overlap=20)
Settings.num_output = 512
Settings.context_window = 3900
# a document summary index needs both an llm and embed model
# for the constructor a vector store index only needs an embed model
index = VectorStoreIndex.from_documents(documents)
```

Figure 8: Useful packages :

3.2 Pinecone as VectorStore method:

We can go ahead and load the SQUAD dataset from HuggingFace [5], which contains questions and answer pairs from Wikipedia articles.

Datasets github repo.

We'll then convert the dataset into a pandas DataFrame and keep only the unique 'context' fields, which are the text passages that the questions are based on.

The code below converts our DataFrame into a list of Document objects, preparing them for indexing using `llama_index`.

Each document includes the following elements:

- A text passage
- A unique ID
- An additional field for the article title

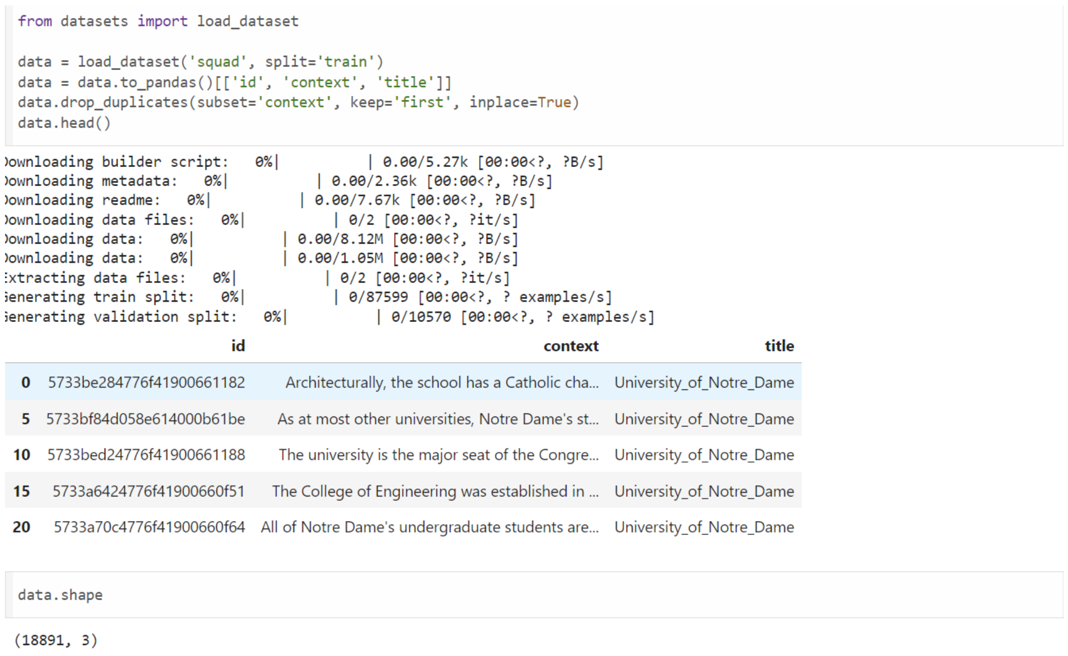
3.3 Indexing in Pinecorne

Pinecone is a vector database designed for fast and scalable indexing and searching of high-dimensional vectors, often used in machine learning and AI applications. In Pinecone, **indexing** refers to the process of organizing and storing these vectors in a way that enables efficient similarity search and retrieval. By converting text, images, or other data into vector embeddings, Pinecone allows you to search for the most similar items within your dataset. This is especially useful in tasks like recommendation systems, natural language processing, or image recognition, where the goal is to find items with similar patterns or features.

3.3.1 1. Initializing PineconeVectorStore

We start by initializing a `PineconeVectorStore` with the previously created Pinecone index. This object acts as the interface for storing and retrieving document embeddings in Pinecone's vector database.

[5] HuggingFace is a technology company that provides tools for natural language processing (NLP) and machine learning (ML)



```
from datasets import load_dataset

data = load_dataset('squad', split='train')
data = data.to_pandas()[['id', 'context', 'title']]
data.drop_duplicates(subset='context', keep='first', inplace=True)
data.head()
```

```
)ownloading builder script:   0%|          | 0.00/5.27k [00:00<?, ?B/s]
)ownloading metadata:   0%|          | 0.00/2.36k [00:00<?, ?B/s]
)ownloading readme:   0%|          | 0.00/7.67k [00:00<?, ?B/s]
)ownloading data files:   0%|          | 0/2 [00:00<?, ?it/s]
)ownloading data:   0%|          | 0.00/8.12M [00:00<?, ?B/s]
)ownloading data:   0%|          | 0.00/1.05M [00:00<?, ?B/s]
:xtracting data files:   0%|          | 0/2 [00:00<?, ?it/s]
;enerating train split:   0%|          | 0/87599 [00:00<?, ? examples/s]
;enerating validation split:   0%|          | 0/10570 [00:00<?, ? examples/s]
```

	id	context	title
0	5733be284776f41900661182	Architecturally, the school has a Catholic cha...	University_of_Notre_Dame
5	5733bf84d058e614000b61be	As at most other universities, Notre Dame's st...	University_of_Notre_Dame
10	5733bed24776f41900661188	The university is the major seat of the Congre...	University_of_Notre_Dame
15	5733a6424776f41900660f51	The College of Engineering was established in ...	University_of_Notre_Dame
20	5733a70c4776f41900660f64	All of Notre Dame's undergraduate students are...	University_of_Notre_Dame

```
data.shape
```

```
(18891, 3)
```

Figure 9: Transformation of DataFrame into a list o Document objects :

3.3.2 2. Initializing GPTVectorStoreIndex

Next, we initialize the `GPTVectorStoreIndex` with our list of `Document` objects, using the`PineconeVectorStore` for storage and the `OpenAIEmbedding` model for generating embeddings. The `StorageContext` is used to configure the storage setup, while the `ServiceContext` sets up the embedding model. The `GPTVectorStoreIndex` handles the indexing and querying process using the provided storage and service contexts.

3.3.3 3. Building the Query Engine

Finally, we can build a query engine from the index we've created and use this engine to perform queries.

4 LlamaIndex OpenAI assistant agent

In this report, several key libraries are employed to interact with OpenAI's API and manage data indexing. The following section provides an explanation of the setup and describes the main components involved in the process.

4.1 Libraries and Setup

The following libraries are used:

- **llama-index**: A library for building indices and querying data. It provides efficient search and retrieval of information.
- **watermark**: This tool displays metadata about the environment, such as the Python version and the versions of installed packages.
- **openai**: The official Python package for interacting with OpenAI's models.

The first step is to install the required packages:

Listing 1: Package Installation

```
%%capture
!pip install llama-index watermark openai
```

Once installed, the `watermark` extension is loaded, allowing the author to display version information for reproducibility:

Listing 2: Using Watermark Extension

```
%load_ext watermark
%watermark -a "Sudarshan Koirala" -vmp llama_index, openai
```

4.2 API Key Setup

To interact with OpenAI's models, an API key is required. The following code prompts the user to securely input their OpenAI API key and stores it in an environment variable:

Listing 3: OpenAI API Key Setup

```
import openai
import os
from getpass import getpass

OPENAI_API_KEY = getpass()  # Prompt for API key
os.environ["OPENAI_API_KEY"] = OPENAI_API_KEY
```

This allows the program to access OpenAI's API in subsequent operations.

4.3 OpenAI Assistant Agent

The `OpenAIAssistantAgent` is imported from the `llama_index` library to interface with OpenAI's assistant functionalities. It simplifies the communication with the models, abstracting away many lower-level details:

Listing 4: Importing OpenAI Assistant Agent

```
from llama_index.agent.openai import OpenAIAssistantAgent
```

This agent serves as the core component for querying and retrieving information from OpenAI models in an efficient manner.

5 Assistant with Query Engine Tools

To enhance the capabilities of the OpenAI Assistant, query engines are integrated to allow seamless information retrieval from large datasets or documents. This enables the assistant to answer complex questions that require searching through indexed data.

5.1 Query Engine Integration

The `llama_index` library provides tools to integrate various query engines with the OpenAI Assistant. These engines can index documents, retrieve relevant information, and return it to the user in a conversational manner. Below is an example of how to set up a query engine and integrate it with the assistant:

Listing 5: Setting up Query Engine

```
from llama_index import GPTVectorStoreIndex
from llama_index.query_engine import SimpleQueryEngine

# Create a vector store index for document retrieval
documents = ["Document␣1␣text", "Document␣2␣text"]  # Example document data
index = GPTVectorStoreIndex.from_documents(documents)

# Initialize a simple query engine using the index
query_engine = SimpleQueryEngine(index)

# Integrate the query engine with the assistant
agent = OpenAIAssistantAgent(query_engine=query_engine)

# Query the assistant with a question
response = agent.query("What␣is␣the␣summary␣of␣Document␣1?")
print(response)
```

In this example, the assistant can now use the query engine to search through indexed documents and retrieve relevant information based on the user's query.

5.2 Using the Assistant

Once the query engine is integrated, the assistant can handle more sophisticated queries that require searching through large datasets. The following code shows how to issue queries to the assistant:

Listing 6: Querying the Assistant

```
question = "What␣are␣the␣key␣points␣of␣Document␣2?"
response = agent.query(question)

# Display the assistant's response
print("Assistant's␣response:", response)
```

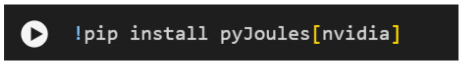
By combining the assistant with the power of query engines, the system becomes highly effective at answering complex questions by accessing indexed data.

6 Energy consumption :

In this section we will be presenting a practical way to measure energy consumption of an NVIDIA GPU during both the training and inference phases of a simple neural network model.
First of all let us present the method which we are going to use :

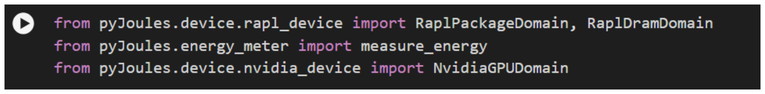
6.1 PyJoules :

PyJoules is a Python library designed to measure the energy[6] consumption of a Python program or a specific code segment. It is particularly useful in contexts where energy efficiency is critical, such as in embedded systems, edge computing, or AI/machine learning applications.
To install the package pyJoules we type the following line of code :



```
!pip install pyJoules[nvidia]
```

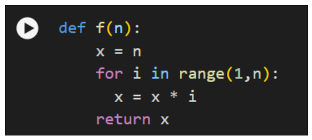
We then import necessary packages:



```
from pyJoules.device.rapl_device import RaplPackageDomain, RaplDramDomain
from pyJoules.energy_meter import measure_energy
from pyJoules.device.nvidia_device import NvidiaGPUDomain
```

Figure 10: Useful packages

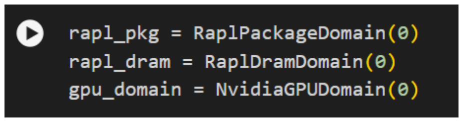
Now consider the following function which calculates the factorial of a positive integer :



```
def f(n):
    x = n
    for i in range(1,n):
      x = x * i
    return x
```

Figure 11: Simple function

To calculate the enrgy consumption of the NVIDIA GPU used to evaluate $f(100000)$ [7] we first create instances of specific domains used to measure energy consumption in different components of our machine as follows :

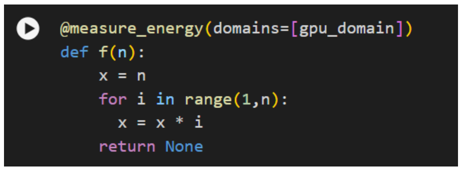


```
rapl_pkg = RaplPackageDomain(0)
rapl_dram = RaplDramDomain(0)
gpu_domain = NvidiaGPUDomain(0)
```

Figure 12: Creation of the environment

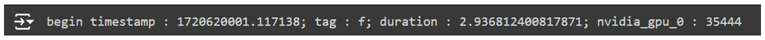
[6]The unit used is μJ
[7]We have to evaluate the factorial of a big number in order to have a significant amount of energy.

We then add a decorator to our function (with some changes)[8] :



```
@measure_energy(domains=[gpu_domain])
def f(n):
    x = n
    for i in range(1,n):
      x = x * i
    return None
```

Here is the result of this execution :



```
begin timestamp : 1720620001.117138; tag : f; duration : 2.936812400817871; nvidia_gpu_0 : 35444
```

As indicated, the energy consumption of GPU is $35444 \mu J$

6.2 Example of a neural network model :

In this section we will work with the following Google Colab notebook.

Execute the code in the notebook and fell free to try different parameters of the model to compare the energy consumption.

Make sure that you are using the correct execution type[9]. For explanation on how to do it, please refer to section IV.

[8]We changed the return of our function to match the new type of return after using the decorator.

[9]It is important to understand that we are mesuring energy consumption in a hardware, so if we are trying to use google colab, we have to make sure that we are mesuring the energy consumption of a GPU

Appendix

I Frameworks Explained

The following commands install necessary Python libraries to work with natural language processing models and data indexing:

1. **`!pip install -q llama-index`**: Installs the `llama-index` library for managing and querying indexed data.
2. **`!pip install -q openai`**: Installs the OpenAI client library, enabling access to AI models provided by OpenAI.
3. **`!pip install -q transformers`**: Installs the `transformers` library from Hugging Face for using pre-trained language models like GPT and BERT.
4. **`!pip install -q accelerate`**: Installs the `accelerate` package for optimizing model performance, especially in large-scale machine learning tasks.
5. **`!pip install llama-index-llms-openai`**: Installs an extension for `llama-index` to integrate OpenAI's language models, enabling advanced natural language processing functionalities.
6. **`!pip install llama-index-readers-file`**: Adds file-reading capabilities to the `llama-index` library, allowing it to index data from various file formats.
7. **`!pip install llama-index-embeddings-openai`**: Installs a module to use OpenAI's embedding models with `llama-index` for semantic representation and querying of textual data.

II Detailed Function Descriptions

II.1 Function: `update_chat_history`

Purpose: The `update_chat_history` function is used to maintain a record of interactions between the user and the chatbot. It updates the chat history by appending a new entry to a list that tracks all user inputs and corresponding bot responses.
Functionality:

- Creates a dictionary containing the user's input and the chatbot's response.
- Appends this dictionary to the list of chat history.

Usage: This function is called whenever a new interaction occurs, ensuring that each user input and corresponding bot response are logged and available for future reference.

II.2 Function: `get_context`

Purpose: The `get_context` function retrieves the entire chat history and formats it into a string. This string is used to provide context for generating relevant responses in the chatbot.
Functionality:

- Initializes an empty string to accumulate the chat history.
- Iterates through each entry in the chat history list, appending user inputs and bot responses.
- Returns the formatted string representing the complete conversation history.

Usage: This function is useful for generating a context string that includes all past interactions, which can be used to maintain coherence in ongoing conversations.

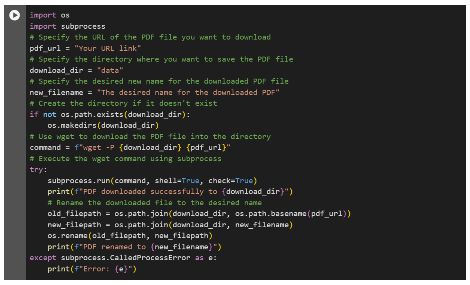
II.3 Function: `query_with_memory`

Purpose: The `query_with_memory` function processes a new user input by incorporating the entire chat history as context, sending this context along with the new input to the chatbot engine, and updating the chat history with the response.
Functionality:

- Retrieves the current chat history using the `get_context` function.
- Constructs a query string that includes the entire history followed by the new user input.
- Sends the query to the chatbot engine and obtains a response.
- Updates the chat history with the new user input and the chatbot's response.
- Returns the response from the chatbot.

Usage: This function integrates previous conversation context with new user input to generate a response, ensuring that the chatbot's replies are contextually relevant and coherent. It also maintains the conversation history for future interactions.

III Upload data from a specified URL :



```python
import os
import subprocess
# Specify the URL of the PDF file you want to download
pdf_url = "Your URL link"
# Specify the directory where you want to save the PDF file
download_dir = "data"
# Specify the desired new name for the downloaded PDF file
new_filename = "The desired name for the downloaded PDF"
# Create the directory if it doesn't exist
if not os.path.exists(download_dir):
    os.makedirs(download_dir)
# Use wget to download the PDF file into the directory
command = f"wget -P {download_dir} {pdf_url}"
# Execute the wget command using subprocess
try:
    subprocess.run(command, shell=True, check=True)
    print(f"PDF downloaded successfully to {download_dir}")
    # Rename the downloaded file to the desired name
    old_filepath = os.path.join(download_dir, os.path.basename(pdf_url))
    new_filepath = os.path.join(download_dir, new_filename)
    os.rename(old_filepath, new_filepath)
    print(f"PDF renamed to {new_filename}")
except subprocess.CalledProcessError as e:
    print(f"Error: {e}")
```

Figure 13: Code to download data :

IV How to change the excecution mode to T4 GPU

1. Open your Google Colab notebook.
2. Go to the menu bar and click on **Execution**.
3. Select **Change execution type**.
4. In the pop-up window, find the **Hardware accelerator** dropdown.
5. Choose **T4 GPU** from the dropdown.
6. Click **Save**.

To verify which GPU you are using, run the following code snippet in a code cell:

```
import tensorflow as tf
print(tf.config.list_physical_devices('GPU'))
```

This will display the details of the GPU being used.