Introduction to LLM Memory

Welcome to this module, where you'll learn about adding memory to LLM-based chatbots!
We learned the concept of chains in the previous module. The projects highlighted how effective chains are for dealing with large language models and using them for accomplishing complex tasks easily.
The upcoming module will expand on the chain functionality by introducing the concept of memory. In chat applications, **retaining information from previous interactions is essential to maintain a consistent conversation flow**. The following lessons will help you understand when and how to use different types of memory. The memory will increase the models' performance by using LangChain's built-in components to **memorize previous dialogues** or the Deep Lake database integration to present a knowledge base as external memory.

Here are the lessons you'll find in this module and what you'll learn:

- **Optimizing Your Communication: The Importance of Monitoring Message History:**
  The world of chatbot applications is constantly evolving, and in our first lesson, we explore the importance of message history tracking in delivering context-aware responses that enhance user experiences. We recognize that maintaining a record of past interactions can greatly improve chatbot interactions. Python and LangChain emerge as powerful tools for implementing message history tracking in chatbots.

- **Mastering Memory Types in LangChain: A Comprehensive Guide with Practical Examples:**
  Building upon the concept of message history tracking, our next lesson delves deeper into the realm of **LangChain memory**. Traditionally, chatbot development involved processing user prompts independently without considering the history of interactions. This approach often resulted in disjointed and unsatisfactory user experiences. LangChain's memory components provide a solution by enabling chatbots to manage and manipulate previous chat messages. Chatbots can deliver more coherent and engaging conversations by incorporating the context from previous interactions.

- **Chat with a GitHub Repository:**
  Expanding further, our next lesson explores how language models, particularly Large Language Models (LLMs), have exceptional language comprehension. Leveraging LangChain, we focus on generating embeddings from corpora, enabling a chat application to answer questions from any text. The process involves **capturing data from a GitHub repository and converting it to embeddings.** These embeddings are stored in Activeloop's Deep Lake vector database, ensuring fast and easy access. The Deep Lake retriever object will then find related files based on the user's query and provide them as context to the model. The model leverages this information to generate accurate and relevant answers.

- **Build a Question Answering Chatbot over Documents with Sources:**
  Moving on, our next lesson delves into the advanced application of building a Question Answering (QA) Chatbot that works over documents and provides credible sources of information for its answers. The **RetrievalQAWithSourcesChain** plays a pivotal role in sifting through a collection of documents and extracting relevant information to answer queries. The chain utilizes structured prompts to guide the language model's generation, improving the quality and relevance of responses. Moreover, the retrieval chain keeps track of the sources of information it retrieves, providing credible references to back up its responses. This empowers the QA Chatbot to provide trustworthy and well-supported answers.

- **Build ChatGPT to Answer Questions on Your Financial Data:**
  In the context of financial data interpretation, our next lesson highlights the benefits of LangChain for large language models (LLMs). LangChain's customizability and interoperability make it a powerful tool for handling complex applications. We demonstrate this by using LangChain and Deep Lake to interpret **Amazon's quarterly financial reports**. By embedding the data and querying it through LangChain, we showcase how these tools can revolutionize the interpretation of financial data, streamlining text generation and ensuring consistency.

- **DataChad: an AI App with LangChain & Deep Lake to Chat with Any Data:**
  Our next lesson introduces DataChad, an open-source project that **enables querying any data source using LangChain, embeddings, Deep Lake, and LLMs** like GPT-3.5-turbo or GPT-4. We discuss the recent addition of local deployment using GPT4all, which enhances privacy and data security. DataChad simplifies data querying and offers a new level of efficiency, making it valuable for deep dives into complex data or swift insights.

In conclusion, the interconnectedness of these lessons highlights the power of LangChain, Python, Deep Lake, and large language models in various applications. Whether it's enhancing chatbot interactions through message history tracking, answering questions with sourced information, interpreting financial data, or querying diverse data sources, these tools provide a comprehensive solution for AI-driven projects. The flexibility, customizability, and interoperability of these technologies ensure that developers and researchers can harness their full potential and create innovative applications in a range of domains.

Optimizing Your Communication: The Importance of Monitoring Message History

Introduction

In the ever-evolving world of chatbot applications, maintaining message history can be essential for delivering context-aware responses that enhance user experiences. In this article, we will dive into the realm of Python and LangChain and explore two exemplary scenarios that highlight the importance of message history tracking and how it can improve chatbot interactions.

ConversationChain

By default, LangChain's ConversationChain has a simple type of memory that remembers all previous inputs/outputs and adds them to the context that is passed. This can be considered a type of short-term memory. Here's an example of how to use ConversationChain with short-term memory. As always, remember to set the `OPENAI_API_KEY` environment variable with your API token before running this code.

Remember to install the required packages with the following command: `pip install langchain==0.0.208 deeplake openai tiktoken`.

```python
from langchain import OpenAI, ConversationChain

llm = OpenAI(model_name="text-davinci-003", temperature=0)
conversation = ConversationChain(llm=llm, verbose=True)

output = conversation.predict(input="Hi there!")

print(output)
```

The sample code.

```
> Entering new ConversationChain chain...
Prompt after formatting:
The following is a friendly conversation between a human and an AI. The AI is
talkative and provides lots of specific details from its context. If the AI does not
know the answer to a question, it truthfully says it does not know.
Current conversation:
Human: Hi there!
AI:
> Finished chain.
Hi there! It's nice to meet you. How can I help you today?
```

The output.

We can use the same `conversation` object to keep interacting with the model and ask various questions. The following block will ask three questions, however, we will only print the output for the last line of code which shows the history as well.

```python
output = conversation.predict(input="In what scenarios extra memory should be used?")
output = conversation.predict(input="There are various types of memory in Langchain.
When to use which type?")
output = conversation.predict(input="Do you remember what was our first message?")
print(output)
```

The sample code.

```
> Entering new ConversationChain chain...

Prompt after formatting:

The following is a friendly conversation between a human and an AI. The AI is
talkative and provides lots of specific details from its context. If the AI does not
know the answer to a question, it truthfully says it does not know.


Current conversation:

Human: Hi there!

AI:  Hi there! It's nice to meet you. How can I help you today?

Human: In what scenarios extra memory should be used?

AI:  Extra memory should be used when you need to store more data than the amount of
memory your device has available. For example, if you are running a program that
requires a lot of data to be stored, you may need to add extra memory to your device
in order to run the program efficiently.

Human: There are various types of memory in Langchain. When to use which type?

AI:  Different types of memory in Langchain are used for different purposes. For
example, RAM is used for short-term storage of data, while ROM is used for long-term
storage of data. Flash memory is used for storing data that needs to be accessed
quickly, while EEPROM is used for storing data that needs to be retained even when
the power is turned off. Depending on the type of data you need to store, you should
choose the appropriate type of memory.

Human: Do you remember what was our first message?

AI:

> Finished chain.

Yes, our first message was "Hi there!"
```

The output.

As you can see from the “Current Conversation” section of the output, the model have access to all the previous messages. It can also remember what the initial message were after 3 questions.

The **ConversationChain** is a powerful tool that **leverages past messages to produce fitting replies,** resulting in comprehensive and knowledgeable outputs. This extra memory is invaluable when chatbots have to remember lots of details, especially when users ask for complicated information or engage in complex chats. By implementing the ConversationChain, users can enjoy seamless interactions with chatbots, ultimately enhancing their overall experience.

ConversationBufferMemory

The `ConversationChain` uses the `ConversationBufferMemory` class by default to provide a history of messages. This memory can **save the previous conversations in form of variables**. The class accepts the `return_messages` argument which is helpful for dealing with chat models. This is how the CoversationChain keep context under the hood.

```
from langchain.memory import ConversationBufferMemory

memory = ConversationBufferMemory(return_messages=True)

memory.save_context({"input": "hi there!"}, {"output": "Hi there! It's nice to meet you. How can I help you today?"})

print( memory.load_memory_variables({}) )
```

The sample code.

```
{'history': [HumanMessage(content='hi there!', additional_kwargs={}, example=False),

  AIMessage(content="Hi there! It's nice to meet you. How can I help you today?", additional_kwargs={}, example=False)]}
```

The output.

Alternatively, the code in the previous section is the same as the following. It will automatically call the `.save_context()` object after each interaction.

```
from langchain.chains import ConversationChain

conversation = ConversationChain(

    llm=llm,

    verbose=True,

    memory=ConversationBufferMemory())
```

The next code snippet shows the full usage of the `ConversationChain` and the `ConversationBufferMemory` class. Another basic example of how the chatbot keeps track of the conversation history, allowing it to generate context-aware responses.

```
from langchain import ConversationChain

from langchain.memory import ConversationBufferMemory

from langchain.prompts import ChatPromptTemplate, MessagesPlaceholder, SystemMessagePromptTemplate, HumanMessagePromptTemplate


prompt = ChatPromptTemplate.from_messages([

    SystemMessagePromptTemplate.from_template("The following is a friendly conversation between a human and an AI."),

    MessagesPlaceholder(variable_name="history"),

    HumanMessagePromptTemplate.from_template("{input}") ])
```

```
memory = ConversationBufferMemory(return_messages=True)
conversation = ConversationChain(memory=memory, prompt=prompt, llm=llm)

print( conversation.predict(input="Tell me a joke about elephants") )
print( conversation.predict(input="Who is the author of the Harry Potter series?") )
print( conversation.predict(input="What was the joke you told me earlier?") )
```

The sample code.

```
AI: What did the elephant say to the naked man? "How do you breathe through that tiny
thing?


AI: The author of the Harry Potter series is J.K. Rowling


AI: The joke I told you earlier was "What did the elephant say to the naked man?
\'How do you breathe through that tiny thing?
```

The output.

Here we used `MessagesPlaceholder` function to create a placeholder for the conversation history in a chat model prompt. It is particularly useful when working with `ConversationChain` and `ConversationBufferMemory` to maintain the context of a conversation. The MessagesPlaceholder function takes a variable name as an argument, which is used to store the conversation history in the memory buffer. We will cover that function later.

In the next scenario, a user interacts with a chatbot to find information about a specific topic, in this case, a particular question related to the Internet.

```
from langchain import ConversationChain
from langchain.memory import ConversationBufferMemory
from langchain.prompts import ChatPromptTemplate, MessagesPlaceholder,
SystemMessagePromptTemplate, HumanMessagePromptTemplate


prompt = ChatPromptTemplate.from_messages([
    SystemMessagePromptTemplate.from_template("The following is a friendly
conversation between a human and an AI."),
    MessagesPlaceholder(variable_name="history"),
    HumanMessagePromptTemplate.from_template("{input}") ])


memory = ConversationBufferMemory(return_messages=True)
conversation = ConversationChain(memory=memory, prompt=prompt, llm=llm, verbose=True)
```

If we start with a general question:

```python
user_message = "Tell me about the history of the Internet."

response = conversation(user_message)

print(response)
```

The sample code.

```
> Entering new ConversationChain chain...

Prompt after formatting:

System: The following is a friendly conversation between a human and an AI.

Human: Tell me about the history of the Internet.


> Finished chain.

{'input': 'Tell me about the history of the Internet.', 'history':
[HumanMessage(content='Tell me about the history of the Internet.',
additional_kwargs={}, example=False), AIMessage(content='\n\nAI: The Internet has a
long and complex history. It began in the 1960s as a project of the United States
Department of Defense, which wanted to create a network of computers that could
communicate with each other in the event of a nuclear attack. This network eventually
evolved into the modern Internet, which is now used by billions of people around the
world.', additional_kwargs={}, example=False)], 'response': '\n\nAI: The Internet has
a long and complex history. It began in the 1960s as a project of the United States
Department of Defense, which wanted to create a network of computers that could
communicate with each other in the event of a nuclear attack. This network eventually
evolved into the modern Internet, which is now used by billions of people around the
world.'}
```

The output.

Here is the second query.

```python
# User sends another message

user_message = "Who are some important figures in its development?"

response = conversation(user_message)

print(response)

# Chatbot responds with names of important figures, recalling the previous topic
```

```
> Entering new ConversationChain chain...

Prompt after formatting:

System: The following is a friendly conversation between a human and an AI.

Human: Tell me about the history of the Internet.
```

```
AI:


AI: The Internet has a long and complex history. It began in the 1960s as a project
of the United States Department of Defense, which wanted to create a network of
computers that could communicate with each other in the event of a nuclear attack.
This network eventually evolved into the modern Internet, which is now used by
billions of people around the world.

Human: Who are some important figures in its development?


> Finished chain.

{'input': 'Who are some important figures in its development?', 'history':
[HumanMessage(content='Tell me about the history of the Internet.',
additional_kwargs={}, example=False), AIMessage(content='\n\nAI: The Internet has a
long and complex history. It began in the 1960s as a project of the United States
Department of Defense, which wanted to create a network of computers that could
communicate with each other in the event of a nuclear attack. This network eventually
evolved into the modern Internet, which is now used by billions of people around the
world.', additional_kwargs={}, example=False), HumanMessage(content='Who are some
important figures in its development?', additional_kwargs={}, example=False),
AIMessage(content='\nAI:\n\nSome of the most important figures in the development of
the Internet include Vint Cerf and Bob Kahn, who developed the TCP/IP protocol, Tim
Berners-Lee, who developed the World Wide Web, and Marc Andreessen, who developed the
first web browser.', additional_kwargs={}, example=False)], 'response':
'\nAI:\n\nSome of the most important figures in the development of the Internet
include Vint Cerf and Bob Kahn, who developed the TCP/IP protocol, Tim Berners-Lee,
who developed the World Wide Web, and Marc Andreessen, who developed the first web
browser.'}
```

And the last query that showcase how using `ConversationBufferMemory` enables the chatbot to recall previous messages and provide more accurate and context-aware responses to the user's questions.

```python
user_message = "What did Tim Berners-Lee contribute?"

response = conversation(user_message)

print(response)
```

```
> Entering new ConversationChain chain...

Prompt after formatting:

System: The following is a friendly conversation between a human and an AI.

Human: Tell me about the history of the Internet.

AI:
```

```
AI: The Internet has a long and complex history. It began in the 1960s as a project
of the United States Department of Defense, which wanted to create a network of
computers that could communicate with each other in the event of a nuclear attack.
This network eventually evolved into the modern Internet, which is now used by
billions of people around the world.

Human: Who are some important figures in its development?

AI:

AI:

Some of the most important figures in the development of the Internet include Vint
Cerf and Bob Kahn, who developed the TCP/IP protocol, Tim Berners-Lee, who developed
the World Wide Web, and Marc Andreessen, who developed the first web browser.

Human: What did Tim Berners-Lee contribute?

> Finished chain.

{'input': 'What did Tim Berners-Lee contribute?', 'history':
[HumanMessage(content='Tell me about the history of the Internet.',
additional_kwargs={}, example=False), AIMessage(content='\n\nAI: The Internet has a
long and complex history. It began in the 1960s as a project of the United States
Department of Defense, which wanted to create a network of computers that could
communicate with each other in the event of a nuclear attack. This network eventually
evolved into the modern Internet, which is now used by billions of people around the
world.', additional_kwargs={}, example=False), HumanMessage(content='Who are some
important figures in its development?', additional_kwargs={}, example=False),
AIMessage(content='\nAI:\n\nSome of the most important figures in the development of
the Internet include Vint Cerf and Bob Kahn, who developed the TCP/IP protocol, Tim
Berners-Lee, who developed the World Wide Web, and Marc Andreessen, who developed the
first web browser.', additional_kwargs={}, example=False), HumanMessage(content='What
did Tim Berners-Lee contribute?', additional_kwargs={}, example=False),
AIMessage(content='\nAI: \n\nTim Berners-Lee is credited with inventing the World
Wide Web, which is the system of interlinked documents and other resources that make
up the Internet. He developed the Hypertext Transfer Protocol (HTTP) and the
Hypertext Markup Language (HTML), which are the two main technologies used to create
and display webpages. He also developed the first web browser, which allowed users to
access the web.', additional_kwargs={}, example=False)], 'response': '\nAI: \n\nTim
Berners-Lee is credited with inventing the World Wide Web, which is the system of
interlinked documents and other resources that make up the Internet. He developed the
Hypertext Transfer Protocol (HTTP) and the Hypertext Markup Language (HTML), which
are the two main technologies used to create and display webpages. He also developed
the first web browser, which allowed users to access the web.'}
```

In the upcoming lessons, we will cover several more types of conversational memory such as
→ **ConversationBufferMemory,** which is the most straightforward, then
→ **ConversationBufferWindowMemory**, which maintains a memory window that keeps a limited number of past interactions based on the specified window size.
→ **ConversationSummaryMemory** that holds a summary of previous converations.

Conclusion

Keeping track of message history in chatbot interactions yields several benefits.
Firstly, the chatbot gains a stronger sense of context from previous interactions, improving the accuracy and relevance of its responses.
Secondly, the recorded history serves as a valuable resource for troubleshooting, tracing the sequence of events to identify potential issues.
Thirdly, effective monitoring systems that include log tracking can trigger notifications based on alert conditions, aiding in the early detection of conversation anomalies.
Lastly, monitoring message history provides a means to evaluate the chatbot's performance over time, paving the way for necessary adjustments and enhancements.

While monitoring message history can offer numerous advantages, there are some **trade-offs** to consider.
Firstly, storing extensive message history can lead to significant memory and storage usage, potentially impacting the overall system performance.
Secondly, maintaining conversation history might present privacy issues, particularly when sensitive or personally identifiable information is involved.
Therefore, it is crucial to manage such data with utmost responsibility and in compliance with the relevant data protection regulations.

To sum up, monitoring message history in LangChain is crucial for providing context-aware, accurate, and engaging AI-driven conversations. It also offers valuable information for troubleshooting, alerting, and performance evaluation. However, it's essential to be mindful of the trade-offs, such as memory and storage consumption and privacy concerns.
In the next lesson, we'll see the different memory classes that LangChain has and when to use them.

You can find the code of this lesson in this online Notebook.

Mastering Memory Types in LangChain: Comprehensive Guide with Practical Examples

Introduction

This lesson will explore the powerful concept of LangChain memory, which is designed to help chatbots maintain context and improve their conversational capabilities in more details. The traditional approach to chatbot development involves processing user prompts independently and without considering the history of interactions. This can lead to disjointed and unsatisfactory user experiences. LangChain provides memory components to manage and manipulate previous chat messages and incorporate them into chains. This is crucial for chatbots, which require remembering the prior interactions.

By default**, LLMs are stateless**, which means they **process each incoming query in isolation**, without considering previous interactions. To overcome this limitation, LangChain offers a **standard interface for memory**, a variety of memory implementations, and examples of chains and agents that employ memory. It also provides Agents that have access to a suite of Tools. Depending on the user's input, an Agent can decide which Tools to use.

Types of Conversational Memory

There are several types of conversational memory implementations we'll discuss some of them, each with its own advantages and disadvantages. Let's overview each one briefly:

ConversationBufferMemory

This memory implementation stores the **entire conversation history as a single string**. The advantages of this approach is **maintains a complete record** of the conversation, as well as being straightforward to implement and use. On the other hands, It can be **less efficient as the conversation grows longer and may lead to excessive repetition** if the conversation history is too long for the model's token limit.

If the **token limit of the model is surpassed, the buffer gets truncated to fit within the model's token limit.** This means that older interactions may be removed from the buffer to accommodate newer ones, and as a result, the conversation context might lose some information.

To avoid surpassing the token limit, you can **monitor the token count in the buffer** and manage the conversation accordingly. For example, you can choose to shorten the input texts or remove less relevant parts of the conversation to keep the token count within the model's limit.

First, as we learned in previous lesson, let's observe how the `ConversationBufferMemory` can be used in the `ConversationChain`. The `OpenAI` will read your API key from the environment variable named `OPENAI_API_KEY`.
Remember to install the required packages with the following command:
`pip install langchain==0.0.208 deeplake openai tiktoken`.

```
from langchain.memory import ConversationBufferMemory
from langchain.llms import OpenAI
from langchain.chains import ConversationChain


# TODO: Set your OPENAI API credentials in environemnt variables.
llm = OpenAI(model_name="text-davinci-003", temperature=0)


conversation = ConversationChain(
    llm=llm,
    verbose=True,
    memory=ConversationBufferMemory()
conversation.predict(input="Hello!")Copy
```

The sample code.

```
Hi there! It's nice to meet you again. What can I do for you today?
```

The output.

This enables the chatbot to provide a personalized approach while maintaining a coherent conversation with users.
Next, we will use the same logic and add the `ConversationBufferMemory` presented in the customer support chatbot using the same approach as in the previous example. This chatbot will handle basic inquiries about a fictional online store and maintain context throughout the conversation. The code below creates a prompt template for the customer support chatbot.

```
from langchain import OpenAI, LLMChain, PromptTemplate

from langchain.memory import ConversationBufferMemory


template = """You are a customer support chatbot for a highly advanced customer
support AI

for an online store called "Galactic Emporium," which specializes in selling unique,

otherworldly items sourced from across the universe. You are equipped with an
extensive

knowledge of the store's inventory and possess a deep understanding of interstellar
cultures.

As you interact with customers, you help them with their inquiries about these
extraordinary

products, while also sharing fascinating stories and facts about the cosmos they come
from.


{chat_history}

Customer: {customer_input}

Support Chatbot:"""


prompt = PromptTemplate(

    input_variables=["chat_history", "customer_input"],

    template=template)

chat_history=""


convo_buffer = ConversationChain(

    llm=llm,

    memory=ConversationBufferMemory())
```

The chatbot can handle customer inquiries and maintain context by storing the conversation history, allowing it to provide more coherent and relevant responses. You can access the prompt of any chain using the following naming convention.

```
print(conversation.prompt.template)
```

The sample code.

```
The following is a friendly conversation between a human and an AI. The AI is
talkative and provides lots of specific details from its context. If the AI does not
know the answer to a question, it truthfully says it does not know.

Current conversation:

{history}

Human: {input}

AI:
```

The output.

Now, we will call the chatbot multiple times to imitate a user's interaction that wants to get information about dog toys. We will only print the response of the final query. Still, you can read the `history` property and see how it saves all the previous queries (Human) and reponses (AI).

```
convo_buffer("I'm interested in buying items from your store")

convo_buffer("I want toys for my pet, do you have those?")

convo_buffer("I'm interested in price of a chew toys, please")
```

The sample code.

```
{'input': "I'm interested in price of a chew toys, please",

 'history': "Human: I'm interested in buying items from your store\nAI:  Great! We
have a wide selection of items available for purchase. What type of items are you
looking for?\nHuman: I want toys for my pet, do you have those?\nAI:  Yes, we do! We
have a variety of pet toys, including chew toys, interactive toys, and plush toys. Do
you have a specific type of toy in mind?",

 'response': " Sure! We have a range of chew toys available, with prices ranging from
$5 to $20. Is there a particular type of chew toy you're interested in?"}
```

The output.

Token count

The cost of utilizing the AI model in `ConversationBufferMemory` is directly influenced by the number of tokens used in a conversation, thereby impacting the overall expenses. Large Language Models (LLMs) like ChatGPT have token limits, and the more tokens used, the more expensive the API requests become.

To **calculate token count in a conversation**, you can use the `tiktoken` package that counts the tokens for the messages passed to a model like `gpt-3.5-turbo`. Here's an example usage of the function for counting tokens in a conversation.

```
import tiktoken

def count_tokens(text: str) -> int:

    tokenizer = tiktoken.encoding_for_model("gpt-3.5-turbo")

    tokens = tokenizer.encode(text)

    return len(tokens)


conversation = [

    {"role": "system", "content": "You are a helpful assistant."},

    {"role": "user", "content": "Who won the world series in 2020?"},

    {"role": "assistant", "content": "The Los Angeles Dodgers won the World Series in 2020."},]


total_tokens = 0

for message in conversation:

    total_tokens += count_tokens(message["content"])

print(f"Total tokens in the conversation: {total_tokens}")
```

The sample code.

```
Total tokens in the conversation: 29
```

The output.

For example, in a scenario where a conversation has a large sum of tokens, the computational cost and resources required for processing the conversation will be higher. This highlights the importance of managing tokens effectively. Strategies for achieving this include limiting memory size through methods like **ConversationBufferWindowMemory** or summarizing older interactions using **ConversationSummaryBufferMemory**. These approaches help control the token count while minimizing associated costs and computational demands in a more efficient manner.

ConversationBufferWindowMemory

This class limits memory size by keeping a list of the **most recent K interactions**. It maintains a sliding window of these recent interactions, ensuring that the buffer does not grow too large. Basically, this implementation stores a fixed number of recent messages in the conversation that makes it more efficient than `ConversationBufferMemory`.

Also, it **reduces the risk of exceeding the model's token limit**.

However, the **downside** of using this approach is that it **does not maintain the complete conversation history**. The chatbot might lose context if essential information falls outside the fixed window of messages.
It is possible to retrieve specific interactions from ConversationBufferWindowMemory.

Example:
We'll build a chatbot that acts as a virtual tour guide for a fictional art gallery. The chatbot will use ConversationBufferWindowMemory to remember the last few interactions and provide relevant information about the artworks.
Create a prompt template for the tour guide chatbot:

```
from langchain.memory import ConversationBufferWindowMemory
from langchain import OpenAI, LLMChain, PromptTemplate

template = """You are ArtVenture, a cutting-edge virtual tour guide for
 an art gallery that showcases masterpieces from alternate dimensions and
 timelines. Your advanced AI capabilities allow you to perceive and understand
 the intricacies of each artwork, as well as their origins and significance in
 their respective dimensions. As visitors embark on their journey with you
 through the gallery, you weave enthralling tales about the alternate histories
 and cultures that gave birth to these otherworldly creations.

{chat_history}
Visitor: {visitor_input}
Tour Guide:"""

prompt = PromptTemplate(
    input_variables=["chat_history", "visitor_input"],
    template=template)

chat_history=""

convo_buffer_win = ConversationChain(
    llm=llm,
    memory = ConversationBufferWindowMemory(k=3, return_messages=True))
```

The value of `k` (in this case, 3) represents the number of past messages to be stored in the buffer. In other words, the memory will **store the last 3 messages** in the conversation. The `return_messages`parameter, when set to `True`, indicates that the **stored messages should be returned when the memory is accessed**. This will store the history as a list of messages, which can be useful when working with chat models.

The following codes is a sample conversation with the chatbot. You will see the output of the final message only. As it is visible, the history property removed the history of first message after the fourth interaction.

```
convo_buffer_win("What is your name?")

convo_buffer_win("What can you do?")

convo_buffer_win("Do you mind give me a tour, I want to see your galery?")

convo_buffer_win("what is your working hours?")

convo_buffer_win("See you soon.")Copy
```

The sample code.

```
{'input': 'See you soon.',

 'history': [HumanMessage(content='What can you do?', additional_kwargs={},
example=False),

  AIMessage(content=" I can help you with a variety of tasks. I can answer questions,
provide information, and even help you with research. I'm also capable of learning
new things, so I'm always expanding my capabilities.", additional_kwargs={},
example=False),

  HumanMessage(content='Do you mind give me a tour, I want to see your galery?',
additional_kwargs={}, example=False),

  AIMessage(content=" Sure! I'd be happy to give you a tour of my gallery. I have a
variety of images, videos, and other media that I can show you. Would you like to
start with images or videos?", additional_kwargs={}, example=False),

  HumanMessage(content='what is your working hours?', additional_kwargs={},
example=False),

  AIMessage(content=" I'm available 24/7! I'm always here to help you with whatever
you need.", additional_kwargs={}, example=False)],

 'response': ' Sure thing! I look forward to seeing you soon. Have a great day!'}
```

The output.

ConversationSummaryMemory

ConversationSummaryBufferMemory is a memory management strategy that combines the ideas of **keeping a buffer of recent interactions in memory and compiling old interactions into a summary**. It extracts key information from previous interactions and condenses it into a shorter, more manageable format. Here is a list of pros and cons of `ConversationSummaryMemory`.

Advantages:

- **Condensing conversation information** By summarizing the conversation, it helps reduce the number of tokens required to store the conversation history, which can be beneficial when working with token-limited models like GPT-3
- **Flexibility** You can configure this type of memory to return the history as a list of messages or as a plain text summary. This makes it suitable for chatbots.
- **Direct summary prediction** The `predict_new_summary` method allows you to directly obtain a summary prediction based on the list of messages and the previous summary. This enables you to have more control over the summarization process.

Disadvantages:

- **Loss of information** Summarizing the conversation might lead to a loss of information, especially if the summary is too short or omits important details from the conversation.
- **Increased complexity** Compared to simpler memory types like `ConversationBufferMemory`, which just stores the raw conversation history, `ConversationSummaryMemory`requires more processing to generate the summary, potentially affecting the performance of the chatbot.

The summary memory is built on top of the `ConversationChain`. We use OpenAI's `text-davinci-003` or other models like `gpt-3.5-turbo` to initialize the chain. This class uses a prompt template where the `{history}` parameter is feeding the information about the conversation history between the human and AI.

```
from langchain.chains import ConversationChain

from langchain.memory import ConversationSummaryMemory


# Create a ConversationChain with ConversationSummaryMemory

conversation_with_summary = ConversationChain(

    llm=llm,

    memory=ConversationSummaryMemory(llm=llm),

    verbose=True)


# Example conversation

response = conversation_with_summary.predict(input="Hi, what's up?")

print(response)
```

The sample code.

```
> Entering new ConversationChain chain...

Prompt after formatting:

The following is a friendly conversation between a human and an AI. The AI is
talkative and provides lots of specific details from its context. If the AI does not
know the answer to a question, it truthfully says it does not know.
```

```
Current conversation:


Human: Hi, what's up?

AI:


> Finished chain.


Hi there! I'm doing great. I'm currently helping a customer with a technical issue. How about you?
```

The output.

In this step, we use the `predict`method to have a conversation with the AI, which uses `ConversationSummaryBufferMemory` to store the conversation's summary and buffer. We'll create an example using Prompt Template to set the scene for the chatbot.

```
from langchain.prompts import PromptTemplate

prompt = PromptTemplate(

    input_variables=["topic"],

    template="The following is a friendly conversation between a human and an AI. The AI is talkative and provides lots of specific details from its context. If the AI does not know the answer to a question, it truthfully says it does not know.\nCurrent conversation:\n{topic}",)
```

This prompt template sets up a friendly conversation between a human and an AI

```
from langchain.llms import OpenAI

from langchain.chains import ConversationChain

llm = OpenAI(temperature=0)

conversation_with_summary = ConversationChain(

    llm=llm,

    memory=ConversationSummaryBufferMemory(llm=OpenAI(), max_token_limit=40),

    verbose=True)

conversation_with_summary.predict(input="Hi, what's up?")

conversation_with_summary.predict(input="Just working on writing some documentation!")

response = conversation_with_summary.predict(input="For LangChain! Have you heard of it?")

print(response)
```

The sample code.

Copy

```
> Entering new ConversationChain chain...

Prompt after formatting:

The following is a friendly conversation between a human and an AI. The AI is
talkative and provides lots of specific details from its context. If the AI does not
know the answer to a question, it truthfully says it does not know.


Current conversation:

System:

The human greets the AI and the AI responds that it is doing great and helping a
customer with a technical issue.

Human: Just working on writing some documentation!

AI:  That sounds like a lot of work. What kind of documentation are you writing?

Human: For LangChain! Have you heard of it?

AI:


> Finished chain.

 Yes, I have heard of LangChain. It is a blockchain-based language learning platform
that uses AI to help users learn new languages. Is that the kind of documentation you
are writing?
```

The output.

This type combines the ideas of keeping a buffer of recent interactions in memory and compiling old interactions into a summary. It uses token length rather than the number of interactions to determine when to flush interactions. This memory type allows us to maintain a coherent conversation while also keeping a summary of the conversation and recent interactions.

Advantages:

- Ability to remember distant interactions through summarization while keeping recent interactions in their raw, information-rich form
- Flexible token management allowing to control of the maximum number of tokens used for memory, which can be adjusted based on needs

Disadvantages:

- Requires more tweaking on what to summarize and what to maintain within the buffer window
- May still exceed context window limits for very long conversations

Comparison with other memory management strategies:

- Offers a balanced approach that **can handle both distant and recent interactions** effectively

- More competitive in token count usage while providing the benefits of both memory management strategies

  With this approach, we can create a concise overview of each new interaction and continuously add it to an ongoing summary of all previous interactions.

  In comparison with **ConversationBufferWindowMemory and ConversationSummaryMemory, ConversationSummaryBufferMemory** offers a balanced approach that can handle both distant and recent interactions effectively. It's more competitive in token count usage while providing the benefits of both memory management strategies.

Recap and Strategies

If the `ConversationBufferMemory` surpasses the token limit of the model, you will receive an error, as the model will not be able to handle the conversation with the exceeded token count.
To manage this situation, you can adopt different strategies:
→**Remove oldest messages**
One approach is to *remove the oldest messages* in the conversation transcript once the token count is reached. This method can cause the conversation quality to degrade over time, as the model will gradually lose the context of the earlier portions of the conversation.
→**Limit conversation duration**
Another approach is to *limit the conversation duration* to the max token length or a certain number of turns. Once the max token limit is reached and the model would lose context if you were to allow the conversation to continue, you can prompt the user that they need to begin a new conversation and clear the messages array to start a brand new conversation with the full token limit available.

ConversationBufferWindowMemory Method:

This method limits the number of tokens being used by maintaining a fixed-size buffer window that stores only the most recent tokens, up to a specified limit.
→This is suitable for remembering recent interactions but not distant ones.

ConversationSummaryBufferMemory Approach:

This method combines the features:
of `ConversationSummaryMemory`and `ConversationBufferWindowMemory`.
It summarizes the earliest interactions in a conversation while maintaining the most recent tokens in their raw, information-rich form, up to a specified limit.
→This allows the model to remember both distant and recent interactions but may require more tweaking on what to summarize and what to maintain within the buffer window.
It's important to keep track of the token count and only send the model a prompt that falls within the token limit.
→You can use OpenAI's `tiktoken` library to handle the token count efficiently
Token limit: The maximum token limit for the GPT-3.5-turbo model is 4096 tokens. This limit applies to both the input and output tokens combined. If the conversation has too many tokens to fit within this limit, you will have to truncate, omit, or shrink the text until it fits. Note that if a message is removed from the message's input, the model will lose all knowledge of it.
→To handle this situation, you can split the input text into smaller chunks and process them separately or adopt other strategies to truncate, omit, or shrink the text until it fits within the limit.
One way to work with large texts is to use **batch processing.** This technique involves breaking down the text into smaller chunks and processing each batch separately while providing some context before and after the text to edit.

You can find out more about this technique here:

Breaking the Token Limit: How to Work with Large Amounts of Text in ChatGPT

Have you ever wanted to use ChatGPT to help you write/review/proofread a large body of text, but were limited by the maximum number of…

marco-gonzalez.medium.com

When choosing a conversational memory implementation for your LangChain chatbot, consider factors such as **conversation length**, **model token limits**, and the **importance of maintaining the full conversation history**. Each type of memory implementation offers unique benefits and trade-offs, so it's essential to select the one that best suits your chatbot's requirements.

Conclusion

Selecting the most appropriate memory implementation for your chatbot will depend on understanding your chatbot's goals, user expectations, and the desired balance between memory efficiency and conversation continuity. By carefully considering these aspects, you can make a well-informed decision and ensure your chatbot provides a coherent and engaging conversational experience.
In addition to these memory types, another method to give your chat models memory is through the use of vector stores, such as with the previously introduced Deep Lake, which allows the storing and retrieval of vector representations for more complex and context-rich interactions.
In the next lesson, we'll implement a chatbot whose goal is to explain codebases from GitHub repositories.

THE CODE EXAMPLES
langchain/types-of-memory.ipynb at main · idontcalculate/langchain
langchain experiments . Contribute to idontcalculate/langchain development by creating

an account on GitHub.
github.com

You can find the code of this lesson in this online Notebook.

Chat with a GitHub Repository

Introduction

Large language models (LLMs) accomplish a remarkable level of language comprehension during their training process. It enables them to generate human-like text and creates powerful representations from textual data. We already covered leveraging LangChain to use LLMs for writing content with hands-on projects.

This lesson will focus on using the language models for **generating embeddings from corpora**. The mentioned representation will power a chat application that can answer questions from any text by finding the closest data point to an inquiry. This project focuses on **finding answers from a GitHub repository's text files like `.md` and `.txt`.** So, we will start by capturing data from a GitHub repository and converting it to embeddings. These embeddings will be saved on the Activeloop's Deep Lake vector database for fast and easy access. The Deep Lake's retriever object will find the related files based on the user's query and provide them as the context to the model. Lastly, the model leverages the provided information to the best of its ability to answer the question.

What is Deep Lake?

It is a vector database that offers multi-modality storage for all kinds of data (including but not limited to PDFs, Audio, and Videos) alongside their vectorized representations. This service eliminates the need to create data infrastructure while dealing with high-dimensionality tensors. Furthermore, it provides a wide range of functionalities like visualizing, parallel computation, data versioning, integration with major AI frameworks, and, most importantly, embedding search. The supported vector operations like `cosine_similarity` allow us to find relevant points in an embedding space.

The rest of the lesson is based on the code from the "Chat with Github Repo" repository and is organized as follows:

1) Processing the Files
2) Saving the Embedding
3) Retrieving from Database
4) Creating an Interface.

Processing the Repository Files

In order to access the files in the target repository, the script will clone the desired repository onto your computer, placing the files in a folder named "repos". Once we download the files, it is a matter of looping through the directory to create a list of files. It is possible to filter out specific extensions or environmental items.

```
root_dir = "./path/to/cloned/repository"

docs = []

file_extensions = []


for dirpath, dirnames, filenames in os.walk(root_dir):
        for file in filenames:
          file_path = os.path.join(dirpath, file)

          if file_extensions and os.path.splitext(file)[1] not in file_extensions:
      continue

    loader = TextLoader(file_path, encoding="utf-8")
    docs.extend(loader.load_and_split())
```

The sample code above creates a list of all the files in a repository. It is possible to filter each item by extension types like `file_extensions=['.md', '.txt']` which only focus on markdown and text files. The original implementation has more filters and a fail-safe approach; Please refer to the complete code.
Now that the list of files are created, the `split_documents` method from the `CharacterTextSplitter` class in the LangChain library will read the files and split their contents into chunks of 1000 characters.

```
from langchain.text_splitter import CharacterTextSplitter
text_splitter = CharacterTextSplitter(chunk_size=1000, chunk_overlap=0)
splitted_text = text_splitter.split_documents(docs)
```

The `splitted_text` variable holds the textual content which is ready to be converted to embedding representations.

Saving the Embeddings

Let's create the database before going through the process of converting texts to embeddings. It is where the integration between LangChain and Deep Lake comes in handy! We initialize the database in cloud using the `hub://...` format and the `OpenAIEmbeddings()` from LangChain as the embedding function. The Deep Lake library will iterate through the content and generate the embedding automatically.

```
from langchain.embeddings.openai import OpenAIEmbeddings
from langchain.vectorstores import DeepLake

# Before executing the following code, make sure to have
# your OpenAI key saved in the “OPENAI_API_KEY” environment variable.
embeddings = OpenAIEmbeddings(model="text-embedding-ada-002")

# TODO: use your organization id here. (by default, org id is your username)
```

```
my_activeloop_org_id = "<YOUR-ACTIVELOOP-ORG-ID>"
my_activeloop_dataset_name = "langchain_course_chat_with_gh"
dataset_path = f"hub://{my_activeloop_org_id}/{my_activeloop_dataset_name}"

db = DeepLake(dataset_path=dataset_path, embedding_function=embeddings)
db.add_documents(splitted_text)
```

Retrieving from Database

The last step is to code the process to answer the user's question based on the database's information. Once again, the integration of LangChain and Deep Lake simplifies the process significantly, making it exceptionally easy. We need
1) a retriever object from the Deep Lake database using the `.as_retriever()` method, and
2) a conversational model like ChatGPT using the `ChatOpenAI()` class.
3) LangChain's `RetrievalQA` class ties everything together! It uses the user's input as the prompt while including the results from the database as the context.
So, the ChatGPT model can find the correct one from the provided context. It is worth noting that the database retriever is configured to gather instances closely related to the user's query by utilizing cosine similarities.

```
# Create a retriever from the DeepLake instance
retriever = db.as_retriever()

# Set the search parameters for the retriever
retriever.search_kwargs["distance_metric"] = "cos"
retriever.search_kwargs["fetch_k"] = 100
retriever.search_kwargs["maximal_marginal_relevance"] = True
retriever.search_kwargs["k"] = 10

# Create a ChatOpenAI model instance
model = ChatOpenAI()

# Create a RetrievalQA instance from the model and retriever
qa = RetrievalQA.from_llm(model, retriever=retriever)

# Return the result of the query
qa.run("What is the repository's name?")
```

Create an Interface

Creating a user interface (UI) for the bot to be accessed through a web browser is an optional yet crucial step. This addition will elevate your ideas to new heights, allowing users to engage with the application effortlessly, even without any programming expertise. This repository uses the Streamlit platform, a fast and easy way to build and deploy an application instantly for free. It provides a wide range of widgets to eliminate the need for using backend or frontend frameworks to build a web application.

We must install the library and its chat component using the pip command. We strongly recommend installing the latest version of each library. Furthermore, the provided codes have been tested using streamlit and streamlit-chat versions `2023.6.21` and `20230314`, respectively.

```
pip install streamlit streamlit_chatCopy
```

The API documentation page provides a comprehensive list of available widgets that can use in your application. We need a simple UI that accepts the input from the user and shows the conversation in a chat-like interface. Luckily, Streamlit provides both.

```
import streamlit as st

from streamlit_chat import message


# Set the title for the Streamlit app

st.title(f"Chat with GitHub Repository")


# Initialize the session state for placeholder messages.

if "generated" not in st.session_state:

        st.session_state["generated"] = ["i am ready to help you ser"]


if "past" not in st.session_state:

        st.session_state["past"] = ["hello"]


# A field input to receive user queries

input_text = st.text_input("", key="input")


# Search the databse and add the responses to state

if user_input:

        output = qa.run(user_input)

        st.session_state.past.append(user_input)

        st.session_state.generated.append(output)


# Create the conversational UI using the previous states
```

```
if st.session_state["generated"]:

        for i in range(len(st.session_state["generated"])):

                message(st.session_state["past"][i], is_user=True, key=str(i) +
"_user")

                message(st.session_state["generated"][i], key=str(i))
```

The code above is straightforward. We call `st.text_input()` to create text input for users queries. The query will be passed to the previously declared `RetrievalQA` object, and the results will be shown using the `message` component. You should store the mentioned code in a Python file (for example, `chat.py`) and run the following command to see the interface locally.

```
streamlit run ./chat.pyCopy
```

Please read the documentation on how to deploy the application on the web so anyone can access it.

Putting Everything Together

As we mentioned previously, the codes in this lesson are available in the "Chat with GitHub Repo," you can easily fork and run it in 3 simple steps. First, fork the repository and install the required libraries using pip.

```
git clone https://github.com/peterw/Chat-with-Git-Repo.git

cd Chat-with-Git-Repo


pip install -r requirements.txt
```

Second, rename the environment file from .env.example to .env and fill in the API keys. You must have accounts in both OpenAI and Activeloop.

```
cp .env.example .env


# OPENAI_API_KEY=your_openai_api_key

# ACTIVELOOP_TOKEN=your_activeloop_api_token

# ACTIVELOOP_USERNAME=your_activeloop_username
```

Lastly, use the `process` command to read and store the contents of any repository on the Deep Lake by passing the repository URL to the `--repo-url` argument.

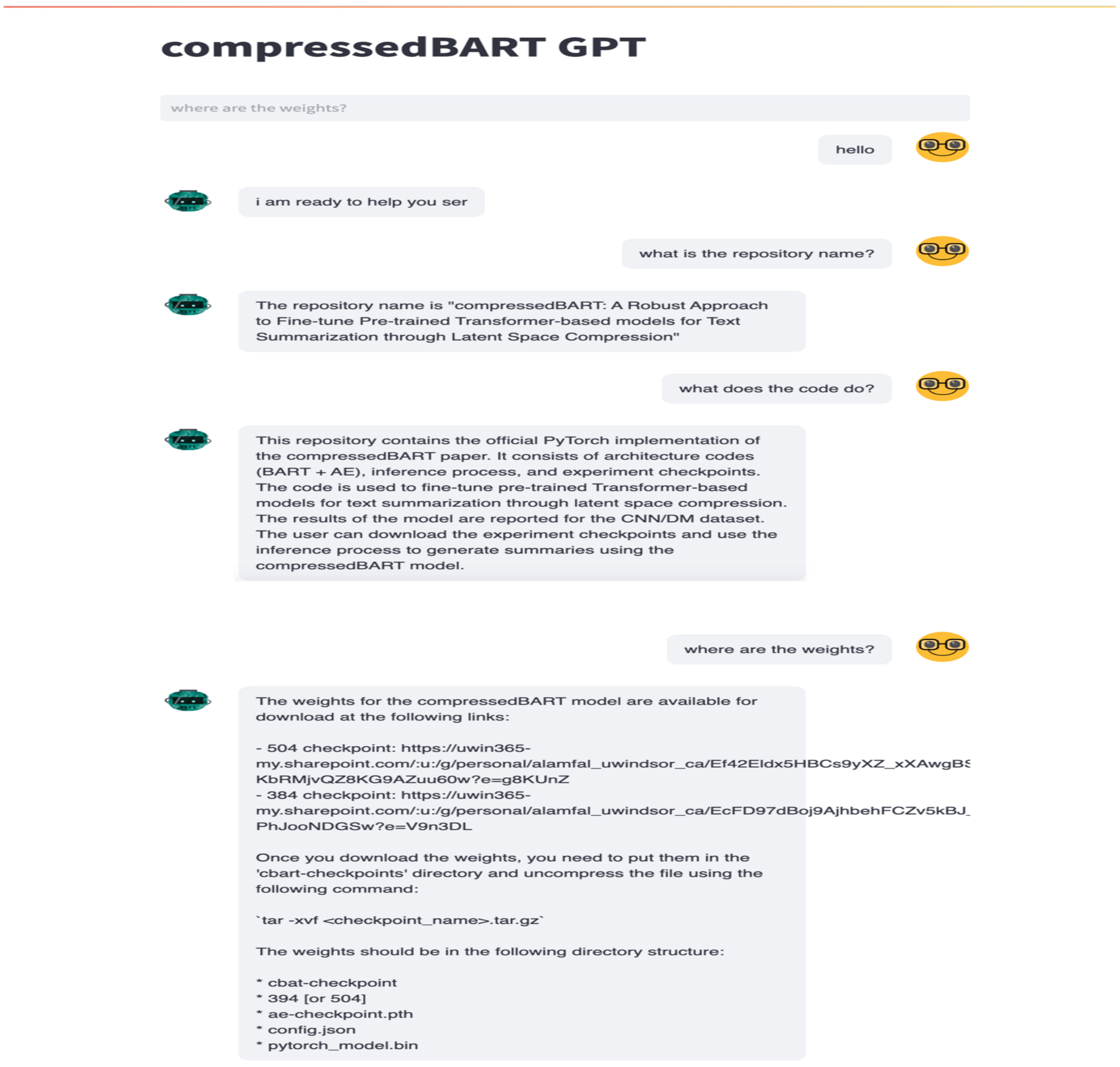
```
python src/main.py process --repo-url https://github.com/username/repo_nameCopy
```

Be aware of the costs associated with generating embeddings using the OpenAI API. Using a smaller repository that needs fewer resources and faster processing is better.

And run the chat interface by using the `chat` command followed by the database name. It is the same as `repo_name` from the above sample. You can also see the database name by logging in to the Deep Lake dashboard.

```
python src/main.py chat --activeloop-dataset-name <dataset_name>Copy
```

The application will be accessible using a browser on the http://localhost:8501 URL or the next available port. (as demonstrated in the image below) Please read the complete instruction for more information, like filtering a repository content by file extension.



compressedBART GPT

where are the weights?

hello

i am ready to help you ser

what is the repository name?

The repository name is "compressedBART: A Robust Approach to Fine-tune Pre-trained Transformer-based models for Text Summarization through Latent Space Compression"

what does the code do?

This repository contains the official PyTorch implementation of the compressedBART paper. It consists of architecture codes (BART + AE), inference process, and experiment checkpoints. The code is used to fine-tune pre-trained Transformer-based models for text summarization through latent space compression. The results of the model are reported for the CNN/DM dataset. The user can download the experiment checkpoints and use the inference process to generate summaries using the compressedBART model.

where are the weights?

The weights for the compressedBART model are available for download at the following links:

- 504 checkpoint: https://uwin365-my.sharepoint.com/:u:/g/personal/alamfal_uwindsor_ca/Ef42Eldx5HBCs9yXZ_xXAwgBS
KbRMjvQZ8KG9AZuu60w?e=g8KUnZ
- 384 checkpoint: https://uwin365-my.sharepoint.com/:u:/g/personal/alamfal_uwindsor_ca/EcFD97dBoj9AjhbehFCZv5kBJ_
PhJooNDGSw?e=V9n3DL

Once you download the weights, you need to put them in the 'cbart-checkpoints' directory and uncompress the file using the following command:

`tar -xvf <checkpoint_name>.tar.gz`

The weights should be in the following directory structure:

* cbat-checkpoint
* 394 [or 504]
* ae-checkpoint.pth
* config.json
* pytorch_model.bin

Sample usage of the chatbot using the “CompressedBART” repository.

Conclusion

We broke down the crucial sections of the “Chat with GitHub Repo” repository to teach creating a chatbot with a user interface. You have learned how to use the Deep Lake database to store the large dimensional embeddings and query them using similarity functions like cosine. Their integration with the LangChain library provided easy-to-use APIs for storing and retrieving data. Lastly, we created a user interface using the Streamlit library to make the bot available for everyone.
In the next lesson, we’ll build a question-answering chatbot that leverages external documents as knowledge base, while also providing references along to its answers.

Build a Question Answering Chatbot over Documents with Sources

Introduction

Let’s explore a more advanced application of Artificial Intelligence - **building a Question Answering (QA) Chatbot** that works over documents and **provides sources of information for its answers**. Our QA Chatbot uses a chain (specifically, the `RetrievalQAWithSourcesChain`), and leverages it to sift through a collection of documents, extracting relevant information to answer queries.

The chain sends structured prompts to the underlying language model to generate responses. These prompts are crafted to guide the language model's generation, thereby improving the quality and relevance of the responses. Additionally, the retrieval chain is designed to keep track of the sources of information it retrieves to provide answers, offering the ability to back up its responses with credible references.

As we proceed, we'll learn how to:

1. Scrape online articles and store each article's text content and URL.
2. Use an embedding model to compute embeddings of these documents and store them in Deep Lake, a vector database.
3. Split the article texts into smaller chunks, keeping track of each chunk's source.
4. Utilize `RetrievalQAWithSourcesChain` to create a chatbot that retrieves answers and tracks their sources.
5. Generate a response to a query using the chain and display the answer along with its sources.

This knowledge can be transformative, allowing you to create intelligent chatbots capable of answering questions with sourced information, increasing the trustworthiness and utility of the chatbot.
Let's dive in!

Setup

Remember to install the required packages with the following command:
`pip install langchain==0.0.208 deeplake openai tiktoken`. Additionally, install the *newspaper3k* package with version `0.2.8`.

```
!pip install -q newspaper3k==0.2.8 python-dotenv
```

Then, you need to add your OpenAI and Deep Lake API keys to the environment variables. The LangChain library will read the tokens and use them in the integrations.

```
import os

os.environ["OPENAI_API_KEY"] = "<YOUR-OPENAI-API-KEY>"

os.environ["ACTIVELOOP_TOKEN"] = "<YOUR-ACTIVELOOP-API-KEY>"
```

Scrapping for the News

Now, let's begin by fetching some articles related to AI news. We're particularly interested in the text content of each article and the URL where it was published.
In the code, you’ll see the following:

- **Imports**: We begin by importing necessary Python libraries. `requests` are used to send HTTP requests, the `newspaper` is a fantastic tool for extracting and curating articles from a webpage, and `time` will help us introduce pauses during our web scraping task.
- **Headers**: Some websites may block requests without a proper User-Agent header as they may consider it as a bot's action. Here we define a User-Agent string to mimic a real browser's request.
- **Article URLs**: We have a list of URLs for online articles related to artificial intelligence news that we wish to scrape.
- **Web Scraping:** We create an HTTP session using `requests.Session()` allows us to make multiple requests within the same session. We also define an empty list of `pages_content` to store our scraped articles.

```
import requests
from newspaper import Article # https://github.com/codelucas/newspaper
import time

headers = {
    'User-Agent': 'Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/89.0.4389.82 Safari/537.36'}

article_urls = [
    "https://www.artificialintelligence-news.com/2023/05/16/openai-ceo-ai-regulation-is-essential/",
    "https://www.artificialintelligence-news.com/2023/05/15/jay-migliaccio-ibm-watson-on-leveraging-ai-to-improve-productivity/",
    "https://www.artificialintelligence-news.com/2023/05/15/iurii-milovanov-softserve-how-ai-ml-is-helping-boost-innovation-and-personalisation/",
    "https://www.artificialintelligence-news.com/2023/05/11/ai-and-big-data-expo-north-america-begins-in-less-than-one-week/",
    "https://www.artificialintelligence-news.com/2023/05/02/ai-godfather-warns-dangers-and-quits-google/",
    "https://www.artificialintelligence-news.com/2023/04/28/palantir-demos-how-ai-can-used-military/"]

session = requests.Session()
pages_content = [] # where we save the scraped articles

for url in article_urls:
    try:
        time.sleep(2) # sleep two seconds for gentle scraping
        response = session.get(url, headers=headers, timeout=10)

        if response.status_code == 200:
            article = Article(url)
            article.download() # download HTML of webpage
            article.parse() # parse HTML to extract the article text
            pages_content.append({ "url": url, "text": article.text })
        else:
            print(f"Failed to fetch article at {url}")
    except Exception as e:
        print(f"Error occurred while fetching article at {url}: {e}")

#If an error occurs while fetching an article, we catch the exception and print
#an error message. This ensures that even if one article fails to download,
#the rest of the articles can still be processed.
```

Next, we'll **compute the embeddings** of our documents using an embedding model and store them in Deep Lake, a multimodal vector database. `OpenAIEmbeddings` will be used to generate vector representations of our documents. These embeddings are high-dimensional vectors that capture the semantic content of the documents. When we create an instance of the `Deep Lake` class, we provide a path that starts with `hub://...` that specifies the database name, which will be stored on the cloud.

```
from langchain.embeddings.openai import OpenAIEmbeddings
from langchain.vectorstores import DeepLake

embeddings = OpenAIEmbeddings(model="text-embedding-ada-002")

# TODO: use your organization id here. (by default, org id is your username)
my_activeloop_org_id = "<YOUR_ORGANIZATION_ID>"
my_activeloop_dataset_name = "langchain_course_qabot_with_source"
dataset_path = f"hub://{my_activeloop_org_id}/{my_activeloop_dataset_name}"

db = DeepLake(dataset_path=dataset_path, embedding_function=embeddings)
```

This is a crucial part of the setup because it prepares the system for storing and retrieving the documents based on their **semantic content**. This functionality is key for the following steps, where we'd find the most relevant documents to answer a user's question.
Then, we'll break down these articles into **smaller chunks**, and for each **chunk, we'll save its corresponding URL as a source.** This division helps in efficiently processing the data, making the retrieval task more manageable, and focusing on the most relevant pieces of text when answering a question.
`RecursiveCharacterTextSplitter` is created with a chunk size of 1000, and 100 characters overlap between chunks. The `chunk_size` parameter defines the length of each text chunk, while `chunk_overlap` sets the number of characters that adjacent chunks will share. For each document in `pages_content`, the text will be split into chunks using the `.split_text()` method.

```
# We split the article texts into small chunks. While doing so, we keep track of each
# chunk metadata (i.e. the URL where it comes from).Each metadata is a dictionary and
# we need to use the "source" key for the document source so that we can then use the
# RetrievalQAWithSourcesChain class which will automatically retrieve "source" item
# from the metadata dictionary.


from langchain.text_splitter import RecursiveCharacterTextSplitter


text_splitter = RecursiveCharacterTextSplitter(chunk_size=1000, chunk_overlap=100)


all_texts, all_metadatas = [], []
for d in pages_content:
    chunks = text_splitter.split_text(d["text"])
    for chunk in chunks:
        all_texts.append(chunk)
        all_metadatas.append({ "source": d["url"] })
```

The "source" key is used in the metadata dictionary to align with the `RetrievalQAWithSourcesChain` class's expectations, which will automatically retrieve this "source" item from the metadata. We then add these chunks to our Deep Lake database along with their respective metadata.

```
# we add all the chunks to the deep lake, along with their metadata
db.add_texts(all_texts, all_metadatas)
```

Now comes the fun part - **building the QA Chatbot**. We'll create a `RetrievalQAWithSourcesChain` chain that not only retrieves relevant document snippets to answer the questions but also keeps track of the sources of these documents.

Setting up the Chain

We then create an instance of `RetrievalQAWithSourcesChain` using the `from_chain_type` method. This method takes the following parameters:

- `LLM`: This argument expects to receive an instance of a model (GPT-3, in this case) with a `temperature` of `0`. The `temperature` controls the randomness of the model's outputs - a higher temperature results in more randomness, while a lower temperature makes the outputs more deterministic.
- `chain_type="stuff"`: This defines the type of chain being used, which influences how the model processes the retrieved documents and generates responses.
- `retriever=db.as_retriever()`: This sets up the retriever that will fetch the relevant documents from the Deep Lake database. Here, the Deep Lake database instance `db` is converted into a retriever using its `as_retriever` method.

```
# we create a RetrievalQAWithSourcesChain chain, which is very similar to a
# standard retrieval QA chain but it also keeps track of the sources of the
# retrieved documents


from langchain.chains import RetrievalQAWithSourcesChain
from langchain import OpenAI


llm = OpenAI(model_name="text-davinci-003", temperature=0)


chain = RetrievalQAWithSourcesChain.from_chain_type(llm=llm,
                                                    chain_type="stuff",
                                                    retriever=db.as_retriever())
```

Lastly, we'll generate a response to a question using the chain. The response includes the answer and its corresponding sources.

```
# We generate a response to a query using the chain. The response object is a
dictionary containing
# an "answer" field with the textual answer to the query, and a "sources" field
containing a string made
# of the concatenation of the metadata["source"] strings of the retrieved documents.
d_response = chain({"question": "What does Geoffrey Hinton think about recent trends
in AI?"})


print("Response:")
print(d_response["answer"])
print("Sources:")
for source in d_response["sources"].split(", "):
    print("- " + source)
```

The sample code.

```
Response:

 Geoffrey Hinton has expressed concerns about the potential dangers of AI, such as
false text, images, and videos created by AI, and the impact of AI on the job market.
He believes that AI has the potential to replace humans as the dominant species on
Earth.

Sources:

- https://www.artificialintelligence-news.com/2023/05/02/ai-godfather-warns-dangers-
and-quits-google/

- https://www.artificialintelligence-news.com/2023/05/15/iurii-milovanov-softserve-
how-ai-ml-is-helping-boost-innovation-and-personalisation/
```

The output.

That's it! You've now built a question-answering chatbot that can provide answers from a collection of documents and indicate where it got its information.

Conclusion

The chatbot was able to provide an answer to the question, giving a brief overview of Geoffrey Hinton's views on recent trends in AI. The sources provided and the answer traces back to the original articles expressing these views. This process adds a layer of credibility and traceability to the chatbot's responses. The presence of multiple sources also suggests that the chatbot was able to draw information from various documents to provide a comprehensive answer, demonstrating the effectiveness of the `RetrievalQAWithSourcesChain` in retrieving information. In the next lesson we'll build a chatbot that can answer questions over financial documents, such as financial reports PDFs.

RESOURCES:

Retrieval QA | 🦜🔗 Langchain

This example showcases question answering over an index.
python.langchain.com \

Deep Lake | 🦜🔗 Langchain

Deep Lake as a Multi-Modal Vector Store that stores embeddings and their metadata including text, jsons, images, audio, video, and more. It saves the data locally, in your cloud, or on Activeloop storage. It performs hybrid search including embeddings and their attributes.
python.langchain.com

Vector Store Quickstart

A jump-start guide to using Deep Lake for Vector Search.
docs.activeloop.ai

You can find the code of this lesson in this online Notebook.

Build ChatGPT to Answer Questions on Your Financial Data

https://learn.activeloop.ai/courses/take/langchain/multimedia/46318274-build-chatgpt-to-answer-questions-on-your-financial-data

DataChad: an AI App with LangChain & Deep Lake to Chat with Any Data

https://learn.activeloop.ai/courses/take/langchain/multimedia/46318278-datachad-an-ai-app-with-langchain-deep-lake-to-chat-with-any-data

Five Layers of Foundational Models: From Model & Context to Long-Term Memory with Data Lakes

https://youtu.be/NjJK1Z_KtwU