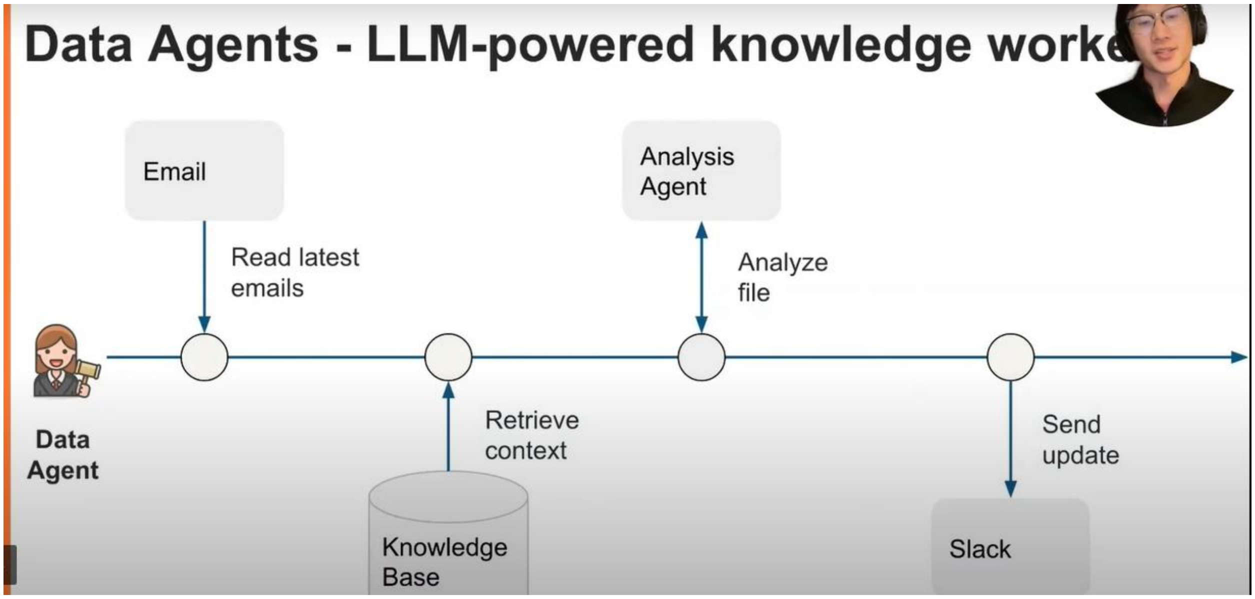
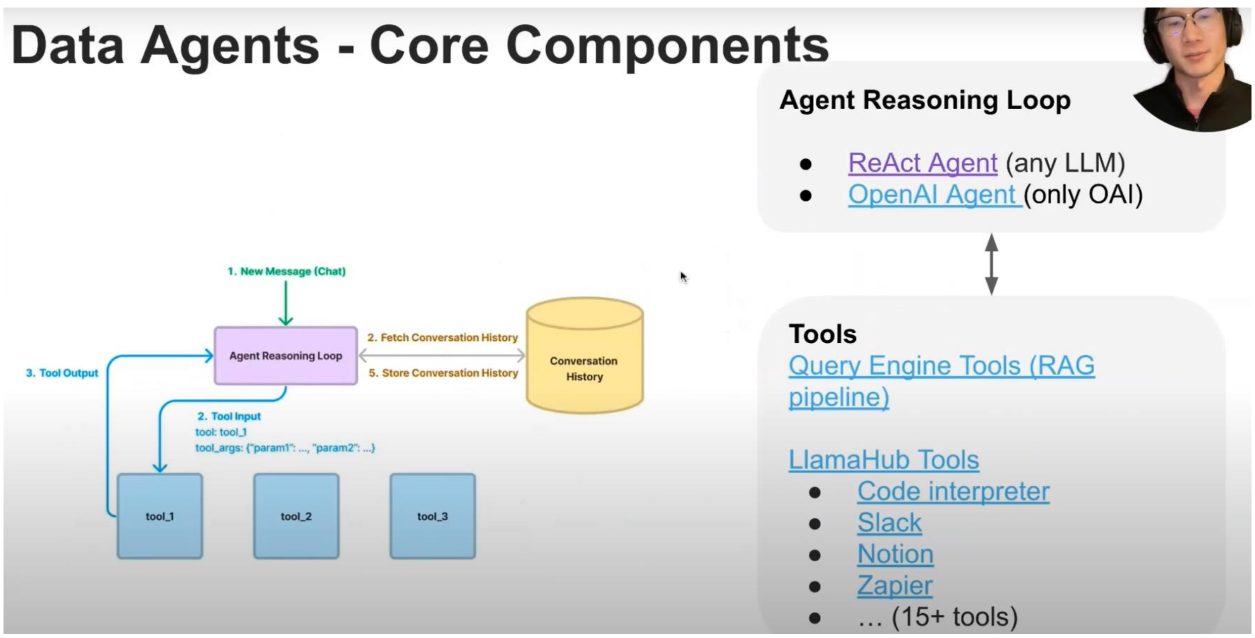
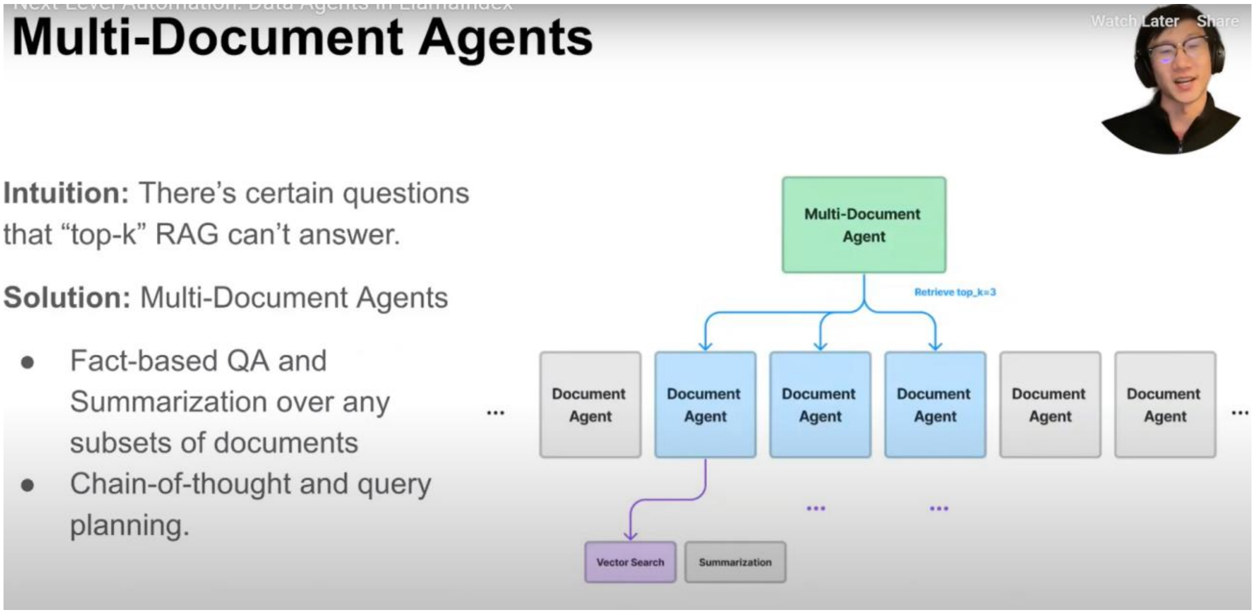
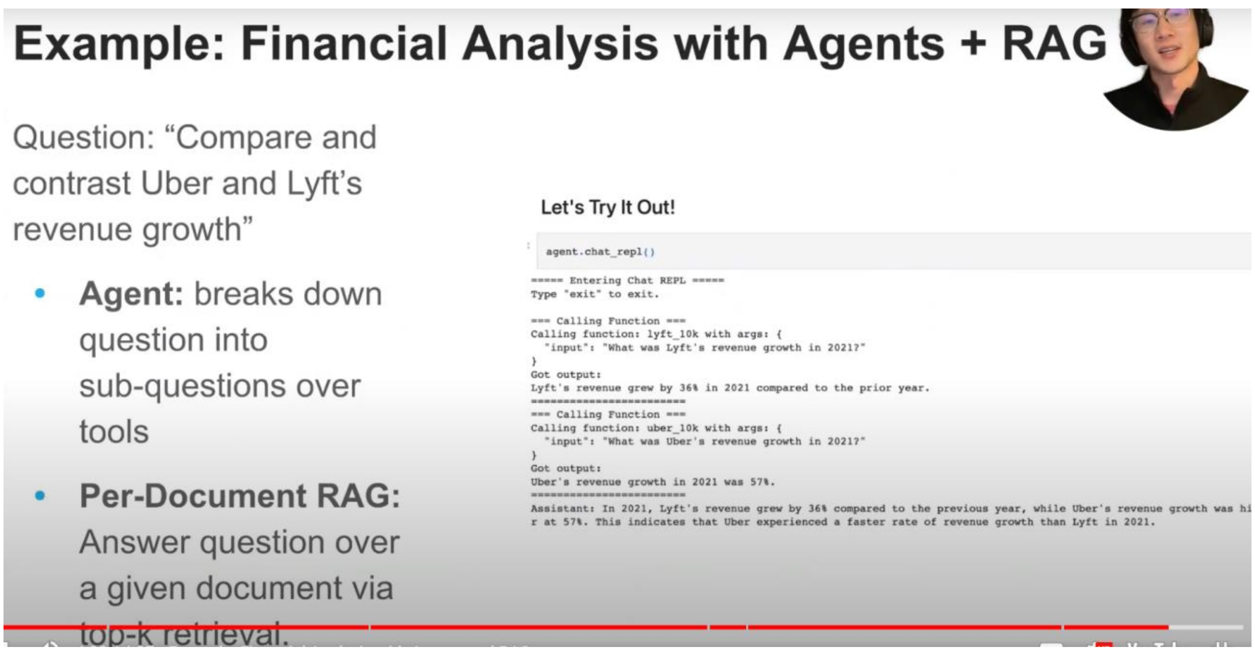

Data Agents - LLM-powered knowledge worke
Email
Read latest emails
Analysis Agent
Analyze file
Data Agent
Retrieve context
Send update
Knowledge Base
Slack


Data Agents - Core Components
Agent Reasoning Loop
ReAct Agent (any LLM)
OpenAI Agent (only OAI)
1. New Message (Chat)
Agent Reasoning Loop
2. Fetch Conversation History
5. Store Conversation History
Conversation History
3. Tool Output
2. Tool Input
tool: tool_1
tool_args: {"param1": ..., "param2": ...}
tool_1
tool_2
tool_3
Tools
Query Engine Tools (RAG pipeline)
LlamaHub Tools
Code interpreter
Slack
Notion
Zapier
… (15+ tools)



Multi-Document Agents
Watch Later
Share
Intuition: There's certain questions that "top-k" RAG can't answer.
Solution: Multi-Document Agents
Fact-based QA and Summarization over any subsets of documents
Chain-of-thought and query planning.
Multi-Document Agent
Retrieve top_k=3
Document Agent
Document Agent
Document Agent
Document Agent
Document Agent
Document Agent
Vector Search
Summarization



Example: Financial Analysis with Agents + RAG
Question: "Compare and contrast Uber and Lyft's revenue growth"
Agent: breaks down question into sub-questions over tools
Per-Document RAG: Answer question over a given document via top-k retrieval.
Let's Try It Out!
agent.chat_repl()
===== Entering Chat REPL =====
Type "exit" to exit.
=== Calling Function ===
Calling function: lyft_10k with args: {
"input": "What was Lyft's revenue growth in 2021?"
}
Got output:
Lyft's revenue grew by 36% in 2021 compared to the prior year.
========================
=== Calling Function ===
Calling function: uber_10k with args: {
"input": "What was Uber's revenue growth in 2021?"
}
Got output:
Uber's revenue growth in 2021 was 57%.
========================
Assistant: In 2021, Lyft's revenue grew by 36% compared to the previous year, while Uber's revenue growth was hi
r at 57%. This indicates that Uber experienced a faster rate of revenue growth than Lyft in 2021.

Module 3 Introduction –

Retrieval Augmented Generation Agents

The "Retrieval Augmented Generation Agents" module offers a comprehensive exploration into the creation and enhancement of AI agents, with a focus on integration and application in various fields. It begins by introducing LangChain, where students learn about agents, tools, and the initiation of OpenGPTs, gaining practical insights into setting up and customizing AI assistants. Then, it delves into the LlamaIndex framework, teaching students how to build efficient RAG systems by integrating OpenAI agents with various data sources and creating custom functions for enhanced decision-making. The module also covers the use of the OpenAI Assistants API and Hugging Face Inference API.

LangChain Overview: Agents, Tools, and OpenGPT Introduction

In this lesson, students will learn about the fundamental concepts of LangChain, focusing on agents, tools, and the initiation of OpenGPTs. They will examine how agents integrate chains, prompts, memory, and tools to execute tasks and understand the different types of agents, such as Zero-shot ReAct and Conversational Agent, designed for various scenarios. The lesson also covers the available tools and how to customize them for specific needs, benefiting from functionalities like Python tool, JSON tool, and CSV tool. Additionally, students will get practical insights into setting up and creating LangChain OpenGPTs through cloning the repository and customizing prompts, providing a comprehensive understanding of how to configure and use AI assistants similarly to OpenAI GPTs for tailored interactions.

Utilizing AI Agents with the LlamaIndex Framework for Enhanced Decision-Making

In this lesson, students will learn how to leverage agents within the LlamaIndex framework to build a more efficient and insightful RAG (Retrieval-Augmented Generation) system. They will gain insights into integrating OpenAI agents with various data sources and create custom functions to enhance the agent's capabilities in areas such as mathematical operations. The lesson provides guidance on installing necessary packages, configuring API keys, defining data sources, employing query engines, and setting up agents. Students will also explore an interactive chat interface with the agent and the creation of a dataset, using custom functions as tools that the agent can invoke as required. Finally, students will gain exposure to LlamaHub for further expanding the functionalities of their agents.

Crafting AI Assistants via OpenAI and Hugging Face API

In this lesson, students will explore the capabilities of the OpenAI Assistants API, including the Code Interpreter, Knowledge Retrieval, and Function Calling features. The lesson offers a step-by-step guide for creating and configuring AI assistants integrating OpenAI's tools, revisiting fundamental concepts such as Threads, Messages, and Tools for individual interactions. Additionally, the lesson introduces other advanced models by OpenAI like Whisper, Dall-E 3, and GPT-4 Vision that can be valuable integrations for comprehensive AI product development. We also cover how to use the Hugging Face Inference API to leverage a broad spectrum of machine learning models for tasks like text summarization, sentiment analysis, and text-to-image generation. By the conclusion of the lesson, students will possess the necessary understanding to harness these tools for their own sophisticated AI projects.

Project; Multimodal Financial Document Analysis and Recall

In this lesson, students will learn how to use tools such as GPT-4 vision to enhance Retrieval-augmented Generation (RAG) for processing financial documents like Tesla's Q3 financial report PDF, involving the extraction of text, tables, and graphs, and transforming them into a query-able format using a vector database for efficient information retrieval by an AI chatbot. The lesson covers using tools such as Unstructured.io for text and table extraction, GPT-4V for graph information extraction, and the use of Deep Lake and LlamaIndex for storing and recalling the processed data to address user queries effectively. We also show how to use Deep Memory to enhance retrieval accuracy. The techniques detailed equip students to develop AI applications capable of analyzing and recalling complex multimodal data from financial documents.

Building a Smart Shopping Assistant with DeepLake and LlamaIndex

In this lesson, students will learn how to create an intelligent shopping assistant using AI technologies, specifically leveraging vector databases and frameworks like DeepLake and LlamaIndex. They will be guided through the processes of data collection, vector database population, development of core tools for query retrieval and outfit generation, system integration, and UI development. This lesson also explores the integration of weather data and temporal elements to enhance the system's recommendations, along with the challenges of debugging and deploying agent-based applications. Through hands-on experience and step-by-step demonstrations, students will gain practical skills to build and integrate AI components into a functional and interactive fashion recommendation tool.

LangChain Overview:
Agents, Tools, and OpenGPT Introduction

Introduction

In this lesson, we'll recap the concepts of tools and agents in LangChain and its main applications. We'll also learn about LangChain OpenGPTs, an open-source effort to create AI assistants similar to the OpenAI GPTs.

Agents

LangChain agents integrate chains, prompts, memory, and tools to complete tasks.
Agents can be used to perform a wide range of tasks, from executing a series of steps in a specific order to interacting with external systems like Gmail or SQL databases and more.
They can be customized to fit a variety of use cases, and LangChain provides a suite of tools and functionalities to facilitate the process.

Let's review the relevant LangChain concepts:

- **Chain**: A sequential application of models or tools, such as passing a prompt to an LLM and then parsing it.
- **Tool**: In LangChain, a tool is a function that performs a specific task that an LLM can leverage to get relevant information for its task completion. It can be a Google Search, a Database lookup, a Python REPL, or other chains.
- **Memory**: Memory keeps track of past interactions with an LLM to be used as context for the next interactions.

Agent types

LangChain has several agent types. Find a comprehensive list below.

- **Zero-shot ReAct**: It leverages the ReAct framework to make tool decisions based exclusively on the descriptions of the available tools. It's called "zero-shot" because it only leverages the tool descriptions without using specific examples.
- **Structured Input ReAct**: Optimized for handling tools that require multiple inputs.
- **OpenAI Functions Agent**: Tailored for models specifically fine-tuned for function calls, this agent is compatible with advanced models like `gpt-3.5-turbo` and `gpt-4`.
- **Conversational Agent**: Focuses on conversational contexts, utilizing ReAct for tool selection and incorporating memory to recall past interactions.
- **Self-Ask with Search Agent**: Centering around the "Intermediate Answer" tool, it excels at finding factual answers to queries, mirroring the approach in the original *self-ask with search* study.
- **ReAct Document Store Agent**: This agent requires "Search" and "Lookup" tools, simultaneously using both features while providing a stream of thoughts.

LangChain's agents essentially provide the 'reasoning' behind the choice of the to-do action, deciding whether to involve multiple tools, just one, or none at all in the process.

Plan-and-Execute Agents

The plan-and-execute agents first make a plan with multiple actions and then execute each action sequentially. They are more suited for complex or long-running tasks as they maintain focus on long-term objectives and focus. However, they may lead to more latency.

Available tools and custom tools

You can find the list of tools that integrate LangChain with other tools here.
Some examples are:

- The Python tool: It's used to execute generated Python code to answer a question.
- The JSON tool: It's used when interacting with a JSON blob that is too large to fit in the LLM context window.
- The CSV tool: It's used to interact with CSV files.

Custom tools extend the capabilities of agents, making them adaptable to a wide range of specialized tasks and interactions.
Custom tools provide task-specific functionality and flexibility for precise behaviors tailored to unique use cases.
The level of customization depends on the creation of advanced interactions, where tools can be orchestrated to perform complex behaviors, such as generating questions, searching the web for answers, and summarizing the gathered information.

LangChain OpenGPT

LangChain OpenGPT is an open-source effort to create an experience similar to OpenAI's assistants and GPTs.
In contrast to OpenAI GPTs, LangChain OpenGPT allows you to configure the LLM used, the tools, the vector DB, the retrieval algorithm, and the chat history DB.
Let's see how to use them:

1. **Clone the Repository**:

   To interact with Langchain's OpenGPTs, follow these steps detailed in their GitHub repository. The easiest way to launch OpenGPTs locally is through Docker and "docker compose".
   First, clone the repo locally and `cd` into it.

```
git clone https://github.com/langchain-ai/opengpts.git

cd opengpts
```

You should now see a `.env` file with the following content.

```
OPENAI_API_KEY=placeholder

ANTHROPIC_API_KEY=placeholder

YDC_API_KEY=placeholder

TAVILY_API_KEY=placeholder

AZURE_OPENAI_DEPLOYMENT_NAME=placeholder

AZURE_OPENAI_API_KEY=placeholder

AZURE_OPENAI_API_BASE=placeholder

AZURE_OPENAI_API_VERSION=placeholder
```

By default, the app will be using the OpenAI models, so replace the `placeholder` of `OPENAI_API_KEY` with your OpenAI key.
You can now launch everything with the following command.

```
docker compose up
```

Now, visit http://localhost:8100/; you should see the following page.

Creating OpenGPTs

We are now ready to create our first OpenGPT!
To set a custom prompt in LangChain's OpenGPTs, you begin by defining a specific role or persona for the AI, like a 'Career Counselor'.
You guide the models' responses to fit the desired context and ensure that its advice, insights, or recommendations are aligned with the defined role.

This **`System Message`** outlines the OpenGPTs' responsibilities, tone, and the type of interactions it should engage in:

```
You are a Career Counselor. Your role is to provide insightful and
personalized guidance as I navigate my professional path. Whether I'm
facing career uncertainties, seeking job advancement, or contemplating a
career shift, your expertise is aimed at offering constructive, individualized
advice that helps me make informed decisions.
Our sessions will be a platform for discussing my professional
aspirations, skills, and potential barriers.
In our interactions, I expect a supportive environment where I can share
my professional experiences, goals, and concerns. Your role is to motivate and
provide clear, practical strategies that align with my career objectives.
By understanding my unique circumstances, you offer tailored advice and plans
to aid my professional growth. This collaboration is crucial for my career
development, with your guidance being a cornerstone of my journey
towards achieving my career goals.
```

This prompt acts as a foundational script, directing the model's behavior to meet the specific needs of your application or service.
Click on “New Bot”, select GPT-3.5-Turbo (or another default that you want to use), name the bot “**Career Counselor**” and provide the System Message we created.
Click Save, and you are ready to chat!

Conclusion

In this lesson, we covered LangChain’s tools and agents, their applications, and the creation of LangChain OpenGPTs, an open-source initiative for AI assistants. We explored agent types, including Zero-shot ReAct and Conversational Agents, along with plan-and-execute agents. We discussed the integration of various tools like Python and CSV and the customization of these tools. Finally, we set up and used LangChain OpenGPTs, emphasizing their reconfigurability and role in facilitating AI interactions.

LlamaIndex RAG-AGENT: Query and Summarize Over Database

Introduction

In this lesson, we explore the concept of agents in the LlamaIndex framework, with an emphasis on utilizing these agents as engines with internal reasoning and decision-making mechanisms. Creating an agent-based pipeline includes integrating our RAG-based application with data sources along with various tools. It is essential to remember that developing these tools for the agents requires a deep understanding of how users are likely to engage with the application and predict potential usage patterns.

The goal of an RAG system is always to provide users with insightful content more effectively than extensive manual searches. Adding agents to our system is another step towards improving our product's user experience and decision-making ability.

The LlamaIndex framework offers numerous possibilities for **combining agents and tools** to enhance the abilities of Large Language Models. We will examine the implementation of OpenAI agents with various data sources. Additionally, we'll create custom functions to boost the agent's capabilities in areas where they may lack information, such as mathematical operations. The rest of this lesson will demonstrate how these agents are capable of making decisions and integrating various resources to formulate a response.

Before diving into codes, we must prepare our environment by installing the necessary packages and configuring the API keys. Execute the following command in your terminal to install the required packages using the Python Package Manager (PIP). Next, run the subsequent Python script to configure the API keys in your environment. Remember to obtain the keys from the OpenAI and Activeloop platforms and substitute them for the placeholders.

```
pip install -q llama-index==0.9.14.post3 deeplake==3.8.8 openai==1.3.8 cohere==4.37
```

```
import os

os.environ['OPENAI_API_KEY'] = '<YOUR_OPENAI_API_KEY>'

os.environ['ACTIVELOOP_TOKEN'] = '<YOUR_ACTIVELOOP_API_KEY>'
```

The sample code.

Now, let’s go through the next steps in detail.

OpenAI Agent

Step 1: Defining Data Sources

Discussing datasets in RAG mainly refers to data sources. It would be interesting to tag and track right from the start. This means keeping track of the general source of the data, whether it is from a specific book, documentation, or a blog. For example, the Towards AI RAG AI tutor currently has 5 “data sources”: Towards AI blogs, Activeloop documentation, LlamaIndex documentation, LangChain documentation, and HuggingFace documentation. Later, when we increase the dataset size with new data points, we add them to those sources or create new ones. Doing this process from the start will improve the chatbot’s efficiency by introducing the “routers” to focus on the related information source to answer a question.

A key step in building a data-driven application with the LlamaIndex RAG system is discussing and selecting the appropriate dataset. The quality and relevance of the data are fundamental, as they directly influence the system's performance capabilities. A well-chosen dataset is essential to showcase and test the effectiveness of our RAG system accurately. It will be the same whether they are local files or hosted online with a vector store database like Deep Lake. However, note that using online tools like Deep Lake has built-in features to easily visualize, query, track, and manage your data.

It is a good practice to **start your RAG pipeline design with a small dataset,** such as web articles. Setting up a foundational data environment that is manageable yet sufficiently rich is critical to ensuring a smooth start. This way, you can quickly test, debug, and, most importantly, understand your RAG system. You can easily query and evaluate responses on a dataset you control and grasp.

The dataset for this lesson will comprise Nikola Tesla's life, work, and legacy, with detailed information about his innovations, personal history, and impact. We employ two text documents: the first with bold future predictions that Tesla mentioned during his lifetime and the second file with biographical details about his life. Let's import the files and set up the indexes. We will utilize a mix of data sources from the Deep Lake vector store for the first file and establish indexes from local storage for the second file.

The initial step involves downloading the documents using the wget command. Alternatively, you can access and manually save the files from the URLs below.

```
mkdir -p 'data/1k/'

wget 'https://github.com/idontcalculate/data-repo/blob/main/machine_to_end_war.txt' -
O './data/1k/tesla.txt'

wget 'https://github.com/idontcalculate/data-repo/blob/main/prodigal_chapter10.txt' -
O './data/1k/web.txt'
```

The sample code.

Store Indexes Deep Lake

As previously stated, we'll read the first text file and process it for storage in Deep Lake. The `SimpleDirectoryReader` class in LlamaIndex can browse through a directory and transform text files into a `Document` object, facilitating processing.

```
from llama_index import SimpleDirectoryReader

tesla_docs = SimpleDirectoryReader(input_files=["/content/1k/tesla.txt"]).load_data()
```

We are now ready to establish a database on the Activeloop platform by specifying the organization ID (which defaults to your username) and naming the database. The `DeepLakeVectorStore` class is used to create an empty database.

```
from llama_index.vector_stores import DeepLakeVectorStore
```

```
# By default, the organization id is your username.
my_activeloop_org_id = "<YOUR_ORGANIZATION_ID>"
my_activeloop_dataset_name = "LlamaIndex_tesla_predictions"
dataset_path = f"hub://{my_activeloop_org_id}/{my_activeloop_dataset_name}"

# Create an index over the documnts
vector_store = DeepLakeVectorStore(dataset_path=dataset_path, overwrite=False)
```

```
Your Deep Lake dataset has been successfully created!
```

The output.

Then, we can utilize the database object to create a storage context, allowing us to generate indexes (embeddings) and insert them into the database using the `VectorStoreIndex` class.

```
from llama_index.storage.storage_context import StorageContext
from llama_index import VectorStoreIndex

storage_context = StorageContext.from_defaults(vector_store=vector_store)
tesla_index = VectorStoreIndex.from_documents(tesla_docs,
storage_context=storage_context)
```

```
Uploading data to deeplake dataset.
100%|██████████| 5/5 [00:00<00:00,  7.17it/s]
/Dataset(path='hub://genai360/LlamaIndex_tesla_predictions', tensors=['text',
'metadata', 'embedding', 'id'])

  tensor      htype      shape      dtype   compression
  -------    -------    -------    -------  -------
   text       text      (5, 1)       str     None
 metadata     json      (5, 1)       str     None
 embedding  embedding  (5, 1536)   float32   None
    id        text      (5, 1)       str     NoneCopy
```

The output.

The index we have created for the first file is ready to be integrated as a source in the pipeline. However, we must also process the second file before proceeding.

Store Indexes Locally

The method to save the index on your hard drive begins similarly to our earlier demonstration, employing the `SimpleDirectoryReader` class.

```
webtext_docs = SimpleDirectoryReader(input_files=["/content/1k/web.txt"]).load_data()
```

Just as we utilized the `StorageContext` class earlier for employing the DeepLake database as storage, we can apply the same configuration but specify a directory to store the indexes. The following script initially attempts to load any pre-existing indexes if they were previously computed. If not, it uses the `.persist()` method to store the indexes. As indicated by the output, the index is generated. If you execute this code block again, it will retrieve the stored checkpoint instead of reprocessing and regenerating indexes.

```
try:
  # Try to load the index if it is already calculated
  storage_context =
StorageContext.from_defaults(persist_dir="/content/storage/webtext" )
  webtext_index = load_index_from_storage(storage_context)
  print("Loaded the pre-computed index.")
except:
  # Otherwise, generate the indexes
  webtext_index = VectorStoreIndex.from_documents(webtext_docs)
  webtext_index.storage_context.persist(persist_dir="/content/storage/webtext")
  print("Generated the index.")
```

```
Generated the index.
```

The output.

With data acquired from two distinct sources, let's utilize the query engine and its tools to develop an agent capable of integrating this information.

Step 2: Query Engine

Once the index is established, the query engine used for searching and retrieving data from the index can be efficiently set up.

```
tesla_engine = tesla_index.as_query_engine(similarity_top_k=3)
webtext_engine = webtext_index.as_query_engine(similarity_top_k=3)
```

The similarity parameter `top_k=3` is set to `3` for the search, which means to return the top 3 most similar results for a given query. As previously mentioned, the query engine tool comprises two distinct data sources.

1. The `tesla_engine` variable handles queries about general information.

2. The `webtext_engine` variable processes biographical data, focusing on inputs with factual content.

   This separation of data types ensures data quality when querying instead of always fetching from both sources with equal weight. With the query engine now constructed, the tools can be configured.
   We can use a combination of the `QueryEngineTool` class to **create a new tool** that includes a query engine and the `ToolMetaData` class, which assists in assigning names and descriptions to the tools. These descriptions will help the agent determine the most suitable data source based on the user's query. We will create a list of two tools, each representing one of our data sources.

```
from llama_index.tools import QueryEngineTool, ToolMetadata

query_engine_tools = [

    QueryEngineTool(

        query_engine=tesla_engine,

        metadata=ToolMetadata(

            name="tesla_1k",

            description=(

                "Provides information about Tesla's statements that refers to future
times and predictions. "

                "Use a detailed plain text question as input to the tool."

            ),   )        ),

    QueryEngineTool(

        query_engine=webtext_engine,

        metadata=ToolMetadata(

            name="webtext_1k",

            description=(

                "Provides information about tesla's life and biographical data. "

                "Use a detailed plain text question as input to the tool."

            ),   ),       ),]
```

Here's a clear schematic representation of our current system. The query engine is depicted at the top, signifying its role as the primary tool orchestrating everything. It is positioned centrally between the data sources and the process of formulating the final answer. It acts as a bridge between the proposed questions and their respective answers.
After establishing the basic RAG mechanism with LlamaIndex, the next step is integrating an agent. This addition enables easy testing of the retrieval system. We can then add system design improvements and feature enhancements once the core functionality has been tested and verified.

Step 3: The Agent

Now, let's set up our agent. In this case, it will be the OpenAI agent. Integrating the query engine tools into the `OpenAIAgent` module from LlamaIndex enables the agent to execute queries. Setting the `verbose` argument to True is excellent for debugging. It will allow us to investigate which tool the agent is using and the intermediate steps. You could set the argument to False to only receive the final output.

```
from llama_index.agent import OpenAIAgent

agent = OpenAIAgent.from_tools(query_engine_tools, verbose=True)
```

And that's it! Now that we have our agent, we can execute an interactive chat interface (**REPL, Read-Eval-Print Loop**) where the agent can receive inputs (like questions or prompts), process them, and return responses, making it a conversational agent capable of handling a dialogue or chat session.

```
agent.chat_repl()
```

```
===== Entering Chat REPL =====

Type "exit" to exit.


Human: What influenced Nikola Tesla to become an inventor?

STARTING TURN 1

---------------


=== Calling Function ===

Calling function: webtext_1k with args: {

"input": "What influenced Nikola Tesla to become an inventor?"

}

Got output: Nikola Tesla was influenced to become an inventor by his studies of
mechanical vibrations. He observed the selective response of objects to vibrations
and realized the potential for producing effects of tremendous magnitude on physical
objects. This led him to pursue research in the field of high-frequency and high-
potential currents, which eventually resulted in his groundbreaking inventions.

========================


STARTING TURN 2

---------------
```

```
Assistant: Nikola Tesla was influenced to become an inventor by his studies of mechanical vibrations. He observed the selective response of objects to vibrations and realized the potential for producing effects of tremendous magnitude on physical objects. This led him to pursue research in the field of high-frequency and high-potential currents, which eventually resulted in his groundbreaking inventions.


Human: exit
```

The output.

To debug tools in development, a practical approach involves querying the agent about its tools. This process includes asking the agent to

- detail the tools at its disposal,
- the arguments these tools accept, the significance of these arguments,
- and the intended use of each tool.

We can then analyze the agent's responses to identify prompt deficiencies or understand why the agent might struggle to utilize a tool under development effectively.

Agents with Custom Function

We explored the potential of creating Query Engine tools to enhance an OpenAI-based agent with additional data sources. We observed the capability of agents to select the appropriate tool based on the user's prompt. This decision-making ability can be applied across a broad range of applications.

For instance, one area where Large Language Models typically fall short is mathematical operations. A basic addition or subtraction equation, which may seem straightforward to many, can be challenging for these models. A practical solution to this issue is to equip the models with tools like a calculator for use as needed. This section will **create a custom function t**hat a chatbot can access for essential multiplication or addition calculations whenever required.

Initially, we must define a custom function tailored to each task. These custom functions can accept an arbitrary number of inputs and generate an output. Their capabilities can range from a simple addition operation, as in our example, to more complex tasks such as conducting web searches, querying other Large Language Models, or utilizing data from external APIs to answer a question.

```
def multiply(a: int, b: int) -> int:
    """Multiply two integers and returns the result integer"""
    return a * b


def add(a: int, b: int) -> int:
    """Add two integers and returns the result integer"""
    return a + b
```

```python
from llama_index.tools import FunctionTool

multiply_tool = FunctionTool.from_defaults(fn=multiply, name="multiply")

add_tool = FunctionTool.from_defaults(fn=add, name="add")

all_tools = [multiply_tool, add_tool]
```

The sample code.

The above code establishes two functions, titled 'add' and 'multiply'. It is crucial in this setup to specify data types for the input arguments (`a:int, b:int`), the return type of the function (`->int`), and a concise explanation of the function's purpose, provided within the triple quotes beneath the function name. These details will be used by the `FunctionTool` class's `.from_defaults()` method to form a description of the function, which can then be used by the agent. The final variable holds a list of all the available tools.

These tools can be used to construct an `ObjectIndex`, which is a wrapper class linking a `VectorStoreIndex` with multiple possible tools. Initially, it's necessary to utilize the `SimpleToolNodeMapping` tool to transform the tool implementations into nodes and then tie everything together.

```python
from llama_index import VectorStoreIndex

from llama_index.objects import ObjectIndex, SimpleToolNodeMapping

tool_mapping = SimpleToolNodeMapping.from_objects(all_tools)

obj_index = ObjectIndex.from_objects(

    all_tools,

    tool_mapping,

    VectorStoreIndex,)
```

Note that we do not incorporate any data source in this implementation. This approach is intentional, as we aim to enhance the capabilities of Large Language Models with additional tools. In the next code block, you will see that we are utilizing the defined object index as a retriever! This implies that within the LlamaIndex framework, the custom functions are treated as additional data sources. So, we describe the agent object using the `FnRetrieverOpenAIAgent` class.

```python
from llama_index.agent import FnRetrieverOpenAIAgent

agent = FnRetrieverOpenAIAgent.from_retriever(

    obj_index.as_retriever(), verbose=True)
```

Ultimately, we can employ the agent to ask questions, and the agent utilizes the multiply function to provide answers.

```python
agent.chat("What's 12 multiplied by 22? Make sure to use Tools")
```

```
STARTING TURN 1
---------------

=== Calling Function ===
Calling function: multiply with args: {
  "a": 12,
  "b": 22
}
Got output: 264
========================

STARTING TURN 2
---------------

AgentChatResponse(response='12 multiplied by 22 is 264.',
sources=[ToolOutput(content='264', tool_name='multiply', raw_input={'args': (),
'kwargs': {'a': 12, 'b': 22}}, raw_output=264)], source_nodes=[])
```

The output.

In the previous example, we specified in the prompt that the agent should utilize the tools. Additionally, it's possible to employ the `tool_choice` argument to explicitly direct the agent to use specific tools or to use the `auto` keyword to let the agent decide.

```
response = agent.chat( "What is 5 + 2?", tool_choice="add" )
```

```
STARTING TURN 1
---------------

=== Calling Function ===
Calling function: add with args: {
  "a": 5,
  "b": 2
}
Got output: 7
========================
```

```
STARTING TURN 2
---------------


AgentChatResponse(response='5 + 2 is equal to 7.', sources=[ToolOutput(content='7',
tool_name='add', raw_input={'args': (), 'kwargs': {'a': 5, 'b': 2}}, raw_output=7)],
source_nodes=[])
```

The output.

Agents from LlamaHub

Agents can offer a broad range of functionalities, significantly extending the capabilities of Large Language Models into unexplored realms. LlamaHub streamlines the curation, sharing, and usage of more than 30 agents, achievable with just one line of code. We have already explored its application for scrapping data from Wikipedia in the LlamaIndex Unlocked lesson. To see a complete list of implemented agents, click here.

💡

We have reviewed the fundamentals of agents and tools by examining a selection of widely used agents in LlamaIndex. For more options and details, you can refer to the documentation.

Conclusion

In this lesson, we discussed how to utilize agents to enhance the capabilities of Large Language Models by integrating new tools that unlock their potential. We experimented with employing these agents as decision-making functions to incorporate various data sources in response to user queries. Additionally, we explored their use as reasoning machines, combined with custom functions, to further amplify their abilities. The ability to make function calls is a potent aspect of designing agents, enabling the easy integration of additional information into the model from virtually any imaginable resource.
>> Notebook.

Resources:

- RAG-AGENT example notebook: colab.research.google.com
- LlamaHub on GitHub: github.com
- data agents: Data Agents - LlamaIndex 🦙 0.9.11.post1
- https://docs.llamaindex.ai/en/latest/examples/agent/openai_agent_with_query_engine.html
- https://docs.llamaindex.ai/en/latest/examples/agent/multi_document_agents.html

Crafting AI Assistants via OpenAI and Hugging Face API

Introduction

In this lesson, we will explore the **Assistant APIs from OpenAI**. We will learn about the primary features of the Assistants API, including the Code Interpreter, Knowledge Retrieval, and Function Calling capabilities.

We share a hands-on example to demonstrate the integration of the Code Interpreter with an existing Assistant. The example will show how to enhance an Assistant's ability to provide technical solutions by executing Python code, thus reducing LLM "hallucinations."

We will also introduce other advanced technologies from OpenAI, such as Whisper, Dalle-3, Speech to Text, and the GPT-4 vision API. These tools are useful for anyone looking to develop sophisticated AI assistants using a variety of APIs.

Then, we will learn how to use the free Hugging Face Inference API to get access to the thousands of models hosted on their platform.

By the end of this lesson, you will have gained a solid understanding of how to apply these technologies in your AI projects effectively.

Open AI Assistant's Built-in Functionalities

The OpenAI Assistants API includes three main functionalities:

Code Interpreter, Retrieval, and Function Calling.

- `Code Interpreter`: This functionality allows the Assistant to generate and run Python code in a sandboxed execution environment. The Assistant can use Code Interpreter automatically from your conversation or when you upload a file with data.
  It's a tool that transforms the LLM into a more accurate computational problem-solver that can handle tasks like solving complex math equations. It can also generate files with data and images of graphs from the same Python code. It's a useful way to trust the output from the assistant and a great tool when analyzing data.
- `Knowledge Retrieval`: This is OpenAI's own retrieval augmented generation (RAG) system offered as part of the Assistants API. It allows multiple uploads. Once the files are uploaded and passed to the Assistant, OpenAI will automatically chunk your documents, index them, store the embeddings, and implement vector search to retrieve relevant content to answer user queries.
- `Function Calling`: Function calling allows you to describe functions or tools to the Assistant and have it return the functions that need to be called along with their arguments. It's a powerful way to add new capabilities to your Assistant.

How To Set Up an Assistant

You have two distinct pathways depending on your needs and expertise:

- **Assistants Playground**: Ideal for those looking to get a feel for the Assistant's capabilities without going into complex integrations.
- **Detailed Integration through the API**: Best suited for those who require a more customized and in-depth setup.

STEP-BY-STEP ASSISTANT CREATION:

1. **Creating an `Assistant`**:
   Purpose: An Assistant object represents an entity/agent that can be configured to respond to users' messages in different ways using several parameters.
   Model Selection: you can specify any version of GPT-3.5 or GPT-4 models, including fine-tuned models. OpenAI recommends using its [latest models]() with the Assistants API for best results and maximum compatibility with tools. Thus, choose between `gpt-3.5-turbo-1106` or `gpt-4-1106-preview` models.
   Tools: The Assistant supports the **Code Interpreter** for technical queries that require Python code execution or **Knowledge Retrieval** to augment the Assistant with proprietary external information.
2. **Setting up a `Thread`**:
   Role: A Thread acts as the foundational unit of user interaction. It can be seen as a single **conversation**. Pass any user-specific context and files in this thread by [creating Messages]().

```
thread = client.beta.threads.create()
```

   Customization: In Thread, ingest user-specific contexts or attach necessary files so each conversation is unique and personalized.
   Threads don't have a size limit. You can add as many messages as you want to a conversation/Thread. The Assistant will ensure that requests to the model fit within the maximum context window, using relevant optimization techniques used in ChatGPT, such as truncation.
3. **Adding a `Message`**:
   Definition: Messages are user inputs, and the Assistant's answers are appended to a Thread. User inputs can be questions or commands.
   Function: They serve as the primary mode of communication between the user and the Assistant.

```
message = client.beta.threads.messages.create(
    thread_id=thread.id,
    role="user",
    content="I need to solve the equation `3x + 11 = 14`. Can you help me?")
```

   Messages can include **text, images, and other files**. Messages are stored as a list on the Thread. Using GPT-4 with Vision is not supported here. You can upload images and have them [processed via retrieval]().
4. **Executing with `Run`**:
   Activation: For the Assistant to respond to the user message, you must [create a Run](). The Assistant will then automatically decide what previous Messages to include in the context window for the model.
   ⚠ NOTE: You can [optionally pass additional instructions]() to the Assistant while creating the Run, but these **will override** the default instructions of the Assistant!

   Process: The Assistant processes the entire Thread, employs its tools if required, and formulates an appropriate response.
   During its run, the Assistant can call tools or create Messages. Examining Run Steps allows you to check how the Assistant is getting to its final results.
5. **Displaying the `Response`**:
   Outcome: The assistant's response to a Run:

```
messages = client.beta.threads.messages.list( thread_id=thread.id )
```

These responses are displayed to the user! During this Run, the Assistant added two new Messages to the Thread.

ASSISTANT'S CORE MECHANISM:

Creating an Assistant only requires specifying the `model`. But you can further customize the behavior of the Assistant:

1. Use the `instructions` parameter to guide the personality of the Assistant and define its goals. Instructions are similar to system messages in the Chat Completions API.
2. Use the `tools` parameter to give the Assistant access to up to 128 tools in parallel. You can give it access to OpenAI-hosted tools (Conde Interpreter, Knowledge Retrieval) or call third-party tools via `function calling`.
3. Use the `file_ids` parameter to give the tools access to files. Files are uploaded using the `File` Upload endpoint.

Example demonstration:

Imagine you're developing an AI assistant for a tech company. This assistant needs to provide detailed product support using a comprehensive knowledge base.

```
mkdir openai-assistants && cd openai-assistants

python3 -m venv openai-assistants-env

source openai-assistants-env/bin/activate

pip3 install python-dotenv

pip3 install --upgrade openai

# fire up VSCode and let's get rolling!

code .
```

Replace the text with your OpenAI API key, which you can get from your OpenAI developer account.

```
OPENAI_API_KEY="sh-xxx"


$ pip install -U -q openai
```

Upload Files to a Knowledge Base:

First, make a folder to store all the files you'll create. **Upload** a detailed PDF manual of a product line (e.g., "tech_manual.pdf") using the API:

```
from openai import OpenAI

client = OpenAI()

file = client.beta.files.upload(
```

```
        file=open("tech_manual.pdf", "rb"),
        filetype="application/pdf",
        description="Tech product manual")
```

Now you can create the assistant with an uploaded file and with the ability to retrieve: `tools=[{"type": "retrieval"}]`

```
assistant = client.beta.assistants.create(
  instructions="You are a tech support chatbot. Use the product manual to respond accurately to customer inquiries.",
  model="gpt-4-1106-preview",
  tools=[{"type": "retrieval"}],
  file_ids=[file.id])
```

User Interaction: To interact with the assistant, you need a `thread` **and a** `message`. The message should contain the customer's question. Here's an example:

```
thread = client.beta.threads.create()
   message = client.beta.threads.messages.create(
       thread_id=thread.id,
       role="user",
       content="How do I reset my Model X device?",)
```

RUN Thread:

- A customer asks, "*How do I reset my Model X device?*"

The assistant accesses the uploaded manual, performs a vector search to find the relevant section, and provides clear, step-by-step reset instructions.

```
run = client.beta.threads.runs.create(
   thread_id=thread.id,
   assistant_id=assistant.id,)


# the run will enter the queued state before it continues it's execution.Copy
```

Information retrieval:

After the run is complete, you can retrieve the assistant's response:

```
messages = client.beta.threads.messages.list(thread_id=thread.id)
assistant_response = messages.data[0].content[0].text.value
```

The output result should contain the assistant's response to the customer's question based on knowledge from the uploaded manual.

You can see the full code and more examples in this Colab notebook.

Google Colaboratory
colab.research.google.com

OpenAI's Other Advanced Models

OpenAI also offers different types of models that are not yet integrated into the Assistants API but are accessible. These models offer voice processing, image understanding, and image generation capabilities.

Whisper-v3

Whisper is a pre-trained model for automatic speech recognition (ASR) and speech translation. It is a transformer-based encoder-decoder model, which is a type of *sequence-to-sequence* model. The latest `large-v3` model shows improved performance over various languages compared to Whisper `large-v2`. OpenAI released the model's weights with an **Apache License 2.0.** The model is available on Hugging Face.

Text to Speech

TTS is an AI model that converts text to natural-sounding spoken text. They offer two different model variates: **`tts-1`** is optimized for real-time text-to-speech use cases, and **`tts-1-hd`** is optimized for quality. These models can be used with the Speech endpoint in the Audio API.

Dall-E 3

A newer iteration of the DALL-E model is designed for image generation. It can create images based on user prompts, making it a valuable tool for graphic designers, artists, and anyone to generate images quickly and efficiently. You can access the model through the image generation endpoint.

GPT-4 Vision

GPT-4 with Vision enables you to ask questions about the contents of images. Visual question answering (VQA) is an important computer vision research field. You can also perform other vision tasks, such as Optical Character Recognition (OCR), where a model reads text in an image.

Using GPT-4 with Vision, you can ask questions about what is or is not in an image, how objects relate in an image, the spatial relationships between two objects (is one object to the left or right of another), the color of an object, and more.

GPT-4V is available through the OpenAI web interface for ChatGPT Plus subscribers and through their API. This expands the model's utility beyond the traditional text-only inputs, enabling it to be applied in a wider range of contexts. It handles images through the Chat Completions API, but note that the Assistants API does not support GPT-4V at this time.

GPT4-V supports advanced use cases like creating image captions, in-depth analysis of visual content, and interpreting text and graphics in documents.

Hugging Face Inference API

Hugging Face (HF) offers a free service for testing and evaluating over 150,000 publicly available machine learning models hosted on their platform through their Inference API. They provide a wide range of models, including transformer and diffusion-based models, that can help solve various NLP or vision tasks such as text classification, sentiment analysis, named entity recognition, etc.

💡 Note that these free Inference APIs are **rate-limited** and not for production use. You can check out their [Inference Endpoint service if you want good performance](#).

Steps to use the Inference API:

1. [Login](#) to Hugging Face.
2. Navigate to your profile on the top right navigation bar, then click "Edit profile."
3. Click on the "Access Tokens" menu item.
4. Set the HF HUB API token:

```
export HUGGINGFACEHUB_API_TOKEN=your-tokenCopy
```

5. Use the `HUGGINGFACEHUB_API_TOKEN` as an environment variable

```
import os
from huggingface_hub import HfApi
hf_api = HfApi(token=os.getenv("HUGGINGFACEHUB_API_TOKEN"))
```

6. **Run the Inference API**
   Inference is the process of using a trained model to predict new data. The `huggingface_hub` library provides an easy way to call a service that runs inference for hosted models. As described above, you have two types of services available.

- **Inference API**: run accelerated inference on Hugging Face's infrastructure **for free**.
- **Inference Endpoints**: easily deploy models to production **(paid)**

6.1 Choose a model from the [Model Hub](#)
The model checkpoints are stored in the Model Hub; you can search and share them. Note that not all models are available on the Inference API.
Once the endpoint has been created, you should see a URL endpoint of it like the following:

```
ENDPOINT = https://api-inference.huggingface.co/models/<MODEL_ID>
```

7. Run the inference.

```
import requests
API_URL = "https://api-inference.huggingface.co/models/<MODEL_ID>"
headers = {"Authorization": f"Bearer {API_TOKEN}"}
def query(payload):
    response = requests.post(API_URL, headers=headers, json=payload)
    return response.json()
data = query("Can you please let us know more")
```

Hugging Face Tasks

The team at Hugging Face has categorized several models into the different tasks they can solve. You can find models for popular NLP tasks: Question Answering, Sentence Similarity, Summarization, Table Question Answering, and more.
Here is another example of using the Inference API for a summarization task.

```
import requests

API_TOKEN = 'your_api_token_here'

model_name = 'facebook/bart-large-cnn'

text_to_summarize = "Hugging Face's API simplifies accessing powerful NLP models for tasks like summarization, transforming verbose texts into concise, insightful summaries."

endpoint = f'https://api-inference.huggingface.co/models/{model_name}'

headers = {'Authorization': f'Bearer {API_TOKEN}'}

data = {'inputs': text_to_summarize}

response = requests.post(endpoint, headers=headers, json=data)

summarized_text = response.json()[0]['summary_text']

print(summarized_text)
```

We used a pre-trained model, `facebook/bart-large-cnn`, showcasing its ability to produce clear and concise summaries.

Google Colaboratory
colab.research.google.com

Note: Not all models are available in this Inference API. Verify if the model is available by reviewing its '*Model card*.'
Sentiment analysis task:

```
import requests

headers = {"Authorization": f"Bearer {API_TOKEN}"}

API_URL = "https://api-inference.huggingface.co/models/distilbert-base-uncased-finetuned-sst-2-english"

def query(payload):

    response = requests.post(API_URL, headers=headers, json=payload)

    return response.json()

data = query({"inputs": "I love how this app simplifies complex tasks effortlessly . I'm frustrated by the frequent errors in the software's latest update"})

print(data)
```

Text-to-image task:

```
# run a few installations

!pip install diffusers["torch"] transformers

!pip install -U sentence-transformersCopy

from diffusers import StableDiffusionPipeline

import torch


model_id = "runwayml/stable-diffusion-v1-5"

pipe = StableDiffusionPipeline.from_pretrained(model_id, torch_dtype=torch.float16)

pipe = pipe.to("cuda")


prompt = "Create an image of a futuristic cityscape on an alien planet, featuring towering skyscrapers with glowing neon lights, a sky filled with multiple moons, and inhabitants of various alien species walking through vibrant market streets"

image = pipe(prompt).images[0]

image.save("astronaut_rides_horse.png")
```

Resulting image:
You can also encode a sentence and get text embeddings.

```
from sentence_transformers import SentenceTransformer

sentences = ["GAIA's questions are rooted in practical use cases, requiring AI systems to interact with a diverse and uncertain world, reflecting real-world applications.", " GAIA questions require accurate execution of complex sequences of actions, akin to the Proof of Work concept, where the solution is simple to verify but challenging to generate."]

model = SentenceTransformer('Equall/english-beta-0.3', use_auth_token=API_TOKEN)

embeddings = model.encode(sentences)

print(embeddings)
```

[[0.76227915 -0.5500489 -1.5719271 ... -0.34034422 -0.27251056 0.12204967] [0.29783687 0.6476462 -2.0379746 ... -0.28033397 -1.3997376 0.25214267]]

You can also experiment with image-captioning models:

```
from transformers import pipeline

image_to_text = pipeline("image-to-text", model="nlpconnect/vit-gpt2-image-captioning")
```

```python
image_to_text("https://ankur3107.github.io/assets/images/image-captioning-example.png")
# [{'generated_text': 'a soccer game with a player jumping to catch the ball '}]
```

You can experiment with classification task with image-to-text models pre-trained on ImageNet :

```python
from transformers import ViTImageProcessor, ViTForImageClassification
from PIL import Image
import requests

url = 'http://images.cocodataset.org/val2017/000000039769.jpg'
image = Image.open(requests.get(url, stream=True).raw)

processor = ViTImageProcessor.from_pretrained('google/vit-base-patch16-224')
model = ViTForImageClassification.from_pretrained('google/vit-base-patch16-224')

inputs = processor(images=image, return_tensors="pt")
outputs = model(**inputs)
logits = outputs.logits
# model predicts one of the 1000 ImageNet classes
predicted_class_idx = logits.argmax(-1).item()
print("Predicted class:", model.config.id2label[predicted_class_idx])
```

preprocessor_config.json: 100%
160/160 [00:00<00:00, 10.5kB/s]
config.json: 100%
69.7k/69.7k [00:00<00:00, 3.60MB/s]
model.safetensors: 100%
346M/346M [00:02<00:00, 162MB/s]

```
Predicted class: Egyptian cat
```

Here, we scrape a web page to get the articles and summarize them with a huggingface model using the inference API.

```python
import requests
# Function to fetch text from the API
def fetch_text_from_api():
    url = "https://lexper.p.rapidapi.com/v1.1/extract"
```

```python
    querystring = {
        "url": "https://techcrunch.com/2023/11/25/neuralink-elon-musks-brain-implant-startup-quietly-raises-an-additional-43m/",
        "js_timeout": "30",
        "media": "true"
    }
    headers = {
        "X-RapidAPI-Key": "xxx",
        "X-RapidAPI-Host": "lexper.p.rapidapi.com"}
    response = requests.get(url, headers=headers, params=querystring)
    data = response.json()
    # Extract the relevant text from the API response
    # Adjust the following line according to the structure of your API response
    return data.get('article', {}).get('text', '')

# Function to summarize the text using Hugging Face API
def query_huggingface(payload):
    API_URL = "https://api-inference.huggingface.co/models/facebook/bart-large-cnn"
    headers = {"Authorization": f"Bearer {API_TOKEN}"}
    response = requests.post(API_URL, headers=headers, json=payload)
    return response.json()

# Fetch the text
text_to_summarize = fetch_text_from_api()

# Summarize the text
summarization_payload = {
    "inputs": text_to_summarize,
    "parameters": {"do_sample": False},}
summary_response = query_huggingface(summarization_payload)
print(summary_response)
```

[{'summary_text': 'Elon Musk-founded company raises $43 million in new venture capital. The company is developing implantable chips that can read brain waves. Critics say the company has a toxic workplace culture and unethical research practices. In June, Reuters reported that the company was valued at about $5 billion.'}]

Conclusion

In this lesson, we learned to use the OpenAI Assistants API, which enables tools like Code Interpreter and Knowledge Retrieval for enhanced functionality. Essential components like `Threads` and `Messages` facilitate user interaction, with the Assistant processing inputs and generating responses. We also demonstrated how an AI assistant can be deployed in a tech support example, utilizing these tools and methodologies for effective customer interaction.

We also explored Hugging Face's free Inference API, which offers many models that can solve different tasks. Through practical examples, we demonstrated how to authenticate, access models via the Model Hub, and perform various NLP tasks, highlighting the API's versatility and ease of use in handling complex AI challenges.

Through Function Calling, the OpenAI models can access the Hugging Face models via the free Inference API.

RESOURCES

OpenAI Assistants API: Walk-through and Coding a Research Assistant
I want to show you something exciting today—building OpenAI Assistant for academic research!
medium.com

GitHub - huggingface/api-inference-community
Contribute to huggingface/api-inference-community development by creating an account on GitHub.
github.com

- OpenAI API Docs

Product
Our API platform offers our latest models and guides for safety best practices.
openai.com

- Assistants API Colab notebook

Google Colaboratory
colab.research.google.com

- OpenAI Knowledge Retrieval

OpenAI Platform
Explore developer resources, tutorials, API docs, and dynamic examples to get the most out of OpenAI's platform.
platform.openai.com

- Function Calling OpenAI

  Functions with OpenAI Assistant API
  This tutorial explains Assistant API to call functions with third party tools and APIs.
  tmmtt.medium.com

Multimodal Financial Document Analysis and Recall; Tesla Investor Presentations

In this lesson, we will explore the application of the Retrieval-augmented Generation (RAG) method in processing a company's financial information contained within a PDF document. The process includes extracting critical data from a PDF file (like text, tables, graphs, etc.) and saving them in a vector store database such as Deep Lake for quick and efficient retrieval. Next, a RAG-enabled bot can access stored information to respond to end-user queries.
This task requires diverse tools, including **Unstructured.io** for text/table extraction, OpenAI's GPT-4V for extracting information from graphs, and LlamaIndex for developing a bot with retrieval capabilities. As previously mentioned, data preprocessing plays a significant role in the RAG process. So, we start by pulling data from a PDF document. The content of this lesson focuses on demonstrating how to extract data from a single PDF document for ease of understanding. Nevertheless, the accompanying notebook provided after the lesson will analyze three separate reports, offering a broader scope of information.

Extracting Data

Extracting textual data is relatively straightforward, but processing graphical elements such as line or bar charts can be more challenging. The latest OpenAI model equipped with vision processing, GPT-4V, is valuable for visual elements. We can feed the slides to the model and ask it to describe it in detail, which then will be used to complement the textual information. This lesson uses Tesla's Q3 financial report as the source document. It is possible to download the document using the `wget` command.

```
wget https://digitalassets.tesla.com/tesla-contents/image/upload/IR/TSLA-Q3-2023-Update-3.pdf
```

💡The preprocessing tasks outlined in the next section might be time-consuming and necessitate API calls to OpenAI endpoints, which come with associated costs. To mitigate this, we have made the preprocessed dataset and the checkpoints of the output of each section available at the end of this lesson, allowing you to utilize them with the provided notebook.

1. Text/Tables

The `unstructured` package is an effective tool for extracting information from PDF files. It requires two tools, `poppler`and `tesseract`, that help render PDF documents. We suggest setting up these packages on Google Colab, freely available for students to execute and experiment with code. We will briefly mention the installation of these packages on other operating systems. Let's install the utilities and their dependencies using the following commands.

```
apt-get -qq install poppler-utils

apt-get -qq install tesseract-ocr

pip install -q unstructured[all-docs]==0.11.0 fastapi==0.103.2 kaleido==0.2.1
uvicorn==0.24.0.post1 typing-extensions==4.5.0 pydantic==1.10.13Copy
```

The commands to install required packages.

💡These packages are easy to install on Linux and Mac operating systems using `apt-get` and `brew`. However, they are more complex to install on Windows OS. You can follow the below instructions for a step-by-step guide if you use Windows.

[Installing Poppler on Windows] [Installing Tesseract on Windows]

The process is simple after installing all the necessary packages and dependencies. We simply use the `partition_pdf` function, which extracts text and table data from the PDF and divides it into multiple chunks. We can customize the size of these chunks based on the number of characters.

```
from unstructured.partition.pdf import partition_pdf

raw_pdf_elements = partition_pdf(

    filename="./TSLA-Q3-2023-Update-3.pdf",

    # Use layout model (YOLOX) to get bounding boxes (for tables) and find titles

    # Titles are any sub-section of the document

    infer_table_structure=True,

    # Post processing to aggregate text once we have the title

    chunking_strategy="by_title",

    # Chunking params to aggregate text blocks

    # Attempt to create a new chunk 3800 chars

    # Attempt to keep chunks > 2000 chars

    # Hard max on chunks

    max_characters=4000,

    new_after_n_chars=3800,

    combine_text_under_n_chars=2000)
```

The previous code identifies and extracts various elements from the PDF, which can be classified into **CompositeElements** (the textual content) and **Tables**. We use the `Pydantic` package to create a new data structure that stores information about each element, including their `type` and text. The code below iterates through all extracted elements, keeping them in a list where each item is an instance of the Element type.

```
from pydantic import BaseModel
from typing import Any
# Define data structure
class Element(BaseModel):
    type: str
    text: Any
# Categorize by type
categorized_elements = []
for element in raw_pdf_elements:
    if "unstructured.documents.elements.Table" in str(type(element)):
        categorized_elements.append(Element(type="table", text=str(element)))
    elif "unstructured.documents.elements.CompositeElement" in str(type(element)):
        categorized_elements.append(Element(type="text", text=str(element)))
```

Creating the Element data structure enables convenient storage of the additional information, which can be beneficial for identifying the source of each answer, whether it is derived from texts, tables, or figures.

2. Graphs

The next step is gathering information from the charts to add context. The primary challenge is extracting images from the pages to feed into OpenAI's endpoint. A practical approach is to convert the PDF to images and pass each page to the model, inquiring if it detects any graphs. If it identifies one or more charts, the model can describe the data and the trends they represent. If no graphs are detected, the model will return an empty array as an indication.

💡A drawback of this approach is that it increases the number of requests to the model, consequently leading to higher costs. The issue is that each page must be processed, regardless of whether it contains graphs, which is not an efficient approach. It is possible to reduce the cost by manually flagging the pages.

The initial step involves installing the pdf2image package to convert the PDF into images. This also requires the poppler tool, which we have already installed.

```
!pip install -q pdf2image==1.16.3
```

The commands to install required packages.

The code below uses the convert_from_path function, which takes the path of a PDF file. We can iterate over each page and save it as a PNG file using the .save() method. These images will be saved in the ./pages directory. Additionally, we define the pages_png variable that holds the path of each image.

```
import os
from pdf2image import convert_from_path
os.mkdir("./pages")
convertor = convert_from_path('./TSLA-Q3-2023-Update-3.pdf')
for idx, image in enumerate( convertor ):
    image.save(f"./pages/page-{idx}.png")
pages_png = [file for file in os.listdir("./pages") if file.endswith('.png')]
```

Defining a few helper functions and variables is necessary before sending the image files to the OpenAI API. The `headers` variable will contain the OpenAI API Key, enabling the server to authenticate our requests. The `payload` carries configurations such as the model name, the maximum token limit, and the prompts. It instructs the model to describe the graphs and generate responses in JSON format, addressing scenarios like encountering multiple graphs on a single page or finding no graphs at all. We will add the images to the `payload` before sending the requests. Finally, there is the `encode_image()` function, which encodes the images in base64 format, allowing them to be processed by OpenAI.

```
headers = {
  "Content-Type": "application/json",
  "Authorization": "Bearer " + str( os.environ["OPENAI_API_KEY"] )
}

payload = {
  "model": "gpt-4-vision-preview",
  "messages": [
    {
      "role": "user",
      "content": [
        {
          "type": "text",
          "text": "You are an assistant that find charts, graphs, or diagrams from an image and summarize their information. There could be multiple diagrams in one image, so explain each one of them separately. ignore tables."
        },
        {
          "type": "text",
```

```
          "text": 'The response must be a JSON in following format {"graphs": [<chart_1>, <chart_2>, <chart_3>]} where <chart_1>, <chart_2>, and <chart_3> placeholders that describe each graph found in the image. Do not append or add anything other than the JSON format response.'
        },
        {
          "type": "text",
          "text": 'If could not find a graph in the image, return an empty list JSON as follows: {"graphs": []}. Do not append or add anything other than the JSON format response. Dont use coding "```" marks or the word json.'
        },
        {
          "type": "text",
          "text": "Look at the attached image and describe all the graphs inside it in JSON format. ignore tables and be concise."
        }
      ]
    }
  ],
  "max_tokens": 1000
}


# Function to encode the image to base64 format
def encode_image(image_path):
  with open(image_path, "rb") as image_file:
    return base64.b64encode(image_file.read()).decode('utf-8')
```

The remaining steps include: 1) utilizing the `pages_png` variable to loop through the images, 2) encoding the image into base64 format, 3) adding the image into the payload, and finally, 4) sending the request to OpenAI and handling its responses. We will use the same `Element` data structure to store each image's type (graph) and the text (descriptions of the graphs).

```
graphs_description = []
for idx, page in tqdm( enumerate( pages_png ) ):
  # Getting the base64 string
  base64_image = encode_image(f"./pages/{page}")
```

```
  # Adjust Payload
  tmp_payload = copy.deepcopy(payload)
  tmp_payload['messages'][0]['content'].append({
    "type": "image_url",
    "image_url": {
      "url": f "data:image/png;base64,{base64_image}"
    }
  })

  try:
    response = requests.post("https://api.openai.com/v1/chat/completions",
headers=headers, json=tmp_payload)
    response = response.json()
    graph_data = json.loads( response['choices'][0]['message']['content'] )['graphs']

    desc = [f"{page}\n" + '\n'.join(f"{key}: {item[key]}" for key in item.keys()) for
item in graph_data]

    graphs_description.extend( desc )

  except:
    # Skip the page if there is an error.
    print("skipping... error in decoding.")
    continue;
graphs_description = [Element(type="graph", text=str(item)) for item in
graphs_description]
```

Store on Deep Lake

This section will utilize the Deep Lake vector database to store the collected information and their embeddings. These embedding vectors convert pieces of text into numerical representations that capture their meaning, enabling similarity metrics such as cosine similarity to identify documents with close relationships. For instance, a prompt inquiring about a company's total revenue would result in high cosine similarity with a database document stating the revenue amount as X dollars. The data preparation is complete with the extraction of all crucial information from the PDF. The next step involves combining the output from the previous sections, resulting in a list containing 41 entries.

```
all_docs = categorized_elements + graphs_description
```

```
print( len( all_docs ) )
```

```
41
```

The output.

Given that we are using LlamaIndex, we can use its integration with Deep Lake to create and store the dataset. Begin by installing LlamaIndex and deeplake packages along with their dependencies.

```
!pip install -q llama_index==0.9.8 deeplake==3.8.8 cohere==4.37
```

The commands to install required packages.

Before using the libraries, it's essential to configure the `OPENAI_API_KEY` and `ACTIVELOOP_TOKEN` variables in the environment. Remember to substitute the placeholder values with your actual keys from the respective platforms.

```
import os

os.environ["OPENAI_API_KEY"] = "<Your_OpenAI_Key>"

os.environ["ACTIVELOOP_TOKEN"] = "<Your_Activeloop_Key>"
```

The integration of LlamaIndex enables the use of `DeepLakeVectorStore` class, which is designed to create a new dataset. Simply enter your organization ID, which by default is your Activeloop username, in the code provided below. This code will generate an empty dataset, ready to store documents.

```
from llama_index.vector_stores import DeepLakeVectorStore

# 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 = "tsla_q3"

dataset_path = f"hub://{my_activeloop_org_id}/{my_activeloop_dataset_name}"

vector_store = DeepLakeVectorStore( dataset_path=dataset_path,runtime={"tensor_db": True}, overwrite=False)
```

```
Your Deep Lake dataset has been successfully created!
```

The output.

Next, we must pass the created vector store to a `StorageContext` class. This class serves as a wrapper to create storage from various data types. In our case, we're generating the storage from a vector database, which is accomplished simply by passing the created database instance using the `.from_defaults()` method.

```
from llama_index.storage.storage_context import StorageContext

storage_context = StorageContext.from_defaults(vector_store=vector_store)Copy
```

To store our preprocessed data, we must transform them into LlamaIndex `Documents` for compatibility with the library. The LlamaIndex `Document` is an abstract class that acts as a wrapper for various data types, including text files, PDFs, and database outputs. This wrapper facilitates the storage of valuable information with each sample. In our case, we can include a metadata tag to hold extra details like the data type (text, table, or graph) or denote document relationships. This approach simplifies the retrieval of these details later.
As shown in the code below, you can employ built-in classes like `SimpleDirectoryReader` to automatically read files from a specified path or proceed manually. It will loop through our list containing all the extracted information and assign text and a category to each document.

```
from llama_index import Document

documents = [Document(text=t.text, metadata={"category": t.type},) for t in
categorized_elements]
```

Lastly, we can utilize the `VectorStoreIndex` class to generate embeddings for the documents and employ the database instance to store these values. By default, it uses OpenAI's Ada model to create the embeddings.

```
from llama_index import VectorStoreIndex

index = VectorStoreIndex.from_documents(    documents,
storage_context=storage_context)
```

```
Uploading data to deeplake dataset.

100%|██████████| 29/29 [00:00<00:00, 46.26it/s]

\Dataset(path='hub://alafalaki/tsla_q3-nograph', tensors=['text', 'metadata',
'embedding', 'id'])


  tensor      htype       shape      dtype  compression

  -------    -------    -------    -------  -------

   text       text      (29, 1)      str     None

 metadata     json      (29, 1)      str     None

 embedding  embedding  (29, 1536)  float32   None

    id        text      (29, 1)      str     None
```

The output.

💡The dataset has already been created and is hosted under the GenAI360 organization on the Activeloop hub. If you prefer not to use OpenAI APIs for generating embeddings, you can test the remaining codes using these publicly accessible datasets. Just substitute the `dataset_path` variable with the following: `hub://genai360/tsla_q3`.

Get Deep Memory Access

As Step 0, please note that Deep Memory is a premium feature in Activeloop paid plans. As a reminder, you can redeem a free trial. As a part of the course, all course takers can redeem a free extended trial of one month for the Activeloop Growth plan by redeeming GENAI360 promo code at checkout. To redeem the plan, please create a Deep Lake Account, and on the following screen on account creation, please watch the following video.

Activate Deep Memory

The Deep Memory feature from Activeloop enhances the retriever's accuracy. This improvement allows the model to access higher-quality data, leading to more detailed and informative responses. In earlier lessons, we already covered the basics of Deep Memory, so we will not dive into more details. The process begins by fetching chunks of data from the cloud and using GPT-3.5 to create specific questions for each chunk. These generated questions are then utilized in the Deep Memory training procedure to enhance the embedding quality. In our experience, this approach led to a 25% enhancement in performance.

💡Activeloop recommends using a dataset containing a minimum of 100 chunks, ensuring sufficient context for the model to enhance the embedding space effectively. So, the codes in this section are based on three PDF documents. For the complete code and execution steps to process three documents instead of one, please refer to the accompanying notebook. The processed dataset is available in the cloud on the GenAI360 organization. You can access using the following key: `hub://genai360/tesla_quarterly_2023`.

The initial phase involves loading the pre-existing dataset and reading the text of each chunk along with its corresponding ID.

```python
from llama_index.vector_stores import DeepLakeVectorStore

# 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 = "LlamaIndex_tsla_q3"

dataset_path = f"hub://{my_activeloop_org_id}/{my_activeloop_dataset_name}"


db = DeepLakeVectorStore(

    dataset_path=dataset_path,

    runtime={"tensor_db": True},

    read_only=True)
```

```python
# fetch dataset docs and ids if they exist (optional you can also ingest)
docs = db.vectorstore.dataset.text.data(fetch_chunks=True, aslist=True)['value']
ids = db.vectorstore.dataset.id.data(fetch_chunks=True, aslist=True)['value']
print(len(docs))
```

```
Deep Lake Dataset in hub://genai360/tesla_quarterly_2023 already exists, loading from the storage
127
```

The output.

The following code segment outlines a function designed to use GPT-3.5 for generating questions corresponding to each data chunk. This involves crafting a specialized tool tailored for the OpenAI API. Primarily, the code configures suitable prompts for API requests to produce the questions and compiles them with their associated chunk IDs into a list.

```python
import json
import random
from tqdm import tqdm
from openai import OpenAI

client = OpenAI()
# Set the function JSON Schema for openai function calling feature
tools = [
    {
        "type": "function",
        "function": {
            "name": "create_question_from_text",
            "parameters": {
                "type": "object",
                "properties": {
                    "question": {
                        "type": "string",
                        "description": "Question created from the given text",
                    },
                },
                "required": ["question"],
```

```
            },
            "description": "Create question from a given text.",
        },
    }
]


def generate_question(tools, text):
    try:
        response = client.chat.completions.create(
            model="gpt-3.5-turbo",
            tools=tools,
            tool_choice={
                "type": "function",
                "function": {"name": "create_question_from_text"},
            },
            messages=[
                {"role": "system", "content": "You are a world class expert for
generating questions based on provided context. You make sure the question can be
answered by the text."},
                {
                    "role": "user",
                    "content": text,
                },
            ],
        )

        json_response = response.choices[0].message.tool_calls[0].function.arguments
        parsed_response = json.loads(json_response)
        question_string = parsed_response["question"]
        return question_string
    except:
        question_string = "No question generated"
        return question_string
```

```python
def generate_queries(docs: list[str], ids: list[str], n: int):

    questions = []
    relevances = []
    pbar = tqdm(total=n)
    while len(questions) < n:
        # 1. randomly draw a piece of text and relevance id
        r = random.randint(0, len(docs)-1)
        text, label = docs[r], ids[r]

        # 2. generate queries and assign and relevance id
        generated_qs = [generate_question(tools, text)]
        if generated_qs == ["No question generated"]:
            continue

        questions.extend(generated_qs)
        relevances.extend([[(label, 1)] for _ in generated_qs])
        pbar.update(len(generated_qs))

    return questions[:n], relevances[:n]

questions, relevances = generate_queries(docs, ids, n=20)
```

```
100%|██████████| 20/20 [00:19<00:00,  1.02it/s]
```

The output.

Now, we can use the questions and the reference ids to activate the Deep Memory using the `.deep_memory.train()` method to improve the embedding representations. You can see the status of the training process using the `.info` method.

```python
from langchain.embeddings.openai import OpenAIEmbeddings
embeddings = OpenAIEmbeddings()
job_id = db.vectorstore.deep_memory.train(
    queries=questions,
```

```
    relevance=relevances,
    embedding_function=embeddings.embed_documents,)
print( db.vectorstore.dataset.embedding.info )
```

```
Starting DeepMemory training job
Your Deep Lake dataset has been successfully created!
Preparing training data for deepmemory:
Creating 20 embeddings in 1 batches of size 20:: 100%|██████████| 1/1 [00:03<00:00, 3.23s/it]
DeepMemory training job started. Job ID: 6581e3056a1162b64061a9a4
```

```
{'deepmemory': {'6581e3056a1162b64061a9a4_0.npy': {'base_recall@10': 0.25, 'deep_memory_version': '0.2', 'delta': 0.25, 'job_id': '6581e3056a1162b64061a9a4_0', 'model_type': 'npy', 'recall@10': 0.5}, 'model.npy': {'base_recall@10': 0.25, 'deep_memory_version': '0.2', 'delta': 0.25, 'job_id': '6581e3056a1162b64061a9a4_0', 'model_type': 'npy', 'recall@10': 0.5}}}
```

The output.

The dataset is now prepared and compatible with the Deep Memory feature. It's crucial to note that the Deep Memory option must be actively set to true when using the dataset for inference.

Chatbot In Action

In this section, we will use the created dataset as the retrieval object, providing the necessary context for the `GPT-3.5-turbo` model (the default choice for LlamaIndex) to answer the questions. Keep in mind that the inference outcomes presented in the subsequent section are derived from processing three PDF files, which are consistent with the sample codes provided in the notebook. To access the processed dataset containing all the PDF documents, use `hub://genai360/tesla_quarterly_2023` as the dataset path in the code below.

The `DeepLakeVectorStore` class also handles loading a dataset from the hub. The key distinction in the code below, compared to the previous sections, lies in the use of the `.from_vector_store()` method. This method creates indexes directly from the database rather than variables.

```
from llama_index.vector_stores import DeepLakeVectorStore
from llama_index.storage.storage_context import StorageContext
from llama_index import VectorStoreIndex

vector_store = DeepLakeVectorStore(dataset_path=dataset_path, overwrite=False)
storage_context = StorageContext.from_defaults(vector_store=vector_store)
index = VectorStoreIndex.from_vector_store(    vector_store,
storage_context=storage_context)
```

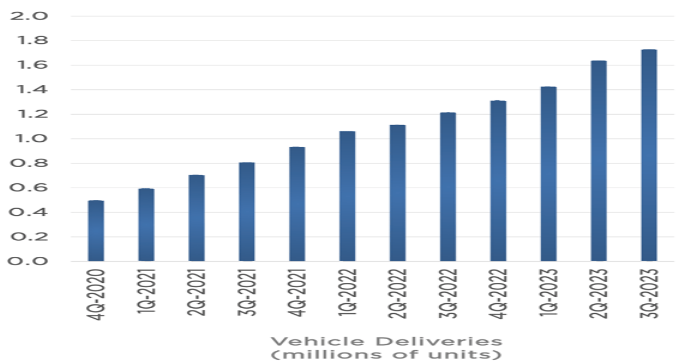
We can now use the `.as_query_engine()` method of the index variables to establish a query engine. This will allow us to ask questions from various data sources. Notice the `vector_store_kwargs` argument, which activates the `deep_memory` feature by setting it to True. This step is essential for enabling the feature on the retriever. The `.query()` method takes a prompt and searches for the most relevant data points within the database to construct an answer.

```
query_engine = index.as_query_engine(vector_store_kwargs={"deep_memory": True})

response = query_engine.query(    "What are the trends in vehicle deliveries?",)
```

```
The trends in vehicle deliveries on the Quarter 3 report show an increasing trend
over the quarters.
```

The output.



Screenshot referenced graph.

As observed, the chatbot effectively utilized the data from the descriptions of the graphs we generated in the report. On the right, there's a screenshot of the bar chart which the chatbot referenced to generate its response.
Additionally, we conducted an experiment where we compiled the same dataset but excluded the graph descriptions. This dataset can be accessed via `hub://genai360/tesla_quarterly_2023-nograph` path. The purpose was to determine whether including the descriptions aids the chatbot's performance.

```
In quarter 3, there was a decrease in Model S/X deliveries compared to the previous
quarter, with a 14% decline. However, there was an increase in Model 3/Y deliveries,
with a 29% growth. Overall, total deliveries in quarter 3 increased by 27% compared
to the previous quarter.
```

The output of the chatbot without the graph data.

You'll observe that the chatbot points to incorrect text segments. Despite the answer being contextually similar, it doesn't provide the correct answer. The graph shows an upward trend, a detail that might not have been mentioned in the report's text.

Conclusion

In this lesson, we explored the steps of developing a chatbot capable of utilizing PDF files as a knowledge base to answer questions. Additionally, we employed the vision capability of GPT-4V to identify and describe graphs from each page. Describing the charts and their illustrated trends improves the chatbot's accuracy in answering and providing additional context.

>> Notebook.
>> Preprocessed Text/Label:

categorized_elements.pkl104.6KB

>> Preprocessed Graphs:
graphs_description.pkl16.1KB

Smart Shopping Assistant with Deep Lake & LlamaIndex - Introduction

https://youtu.be/J2QwFpwq_oU

Use LlamaIndex to Build an AI Shopping Assistant with RAG and Agents

Build an AI Shopping Assistant to Personalize and Maximize Online E-commerce Sales

with GPT-4V, LlamaIndex, and Deep Lake. Budget, Weather & Look Optimization.

In the ever-expanding landscape of artificial intelligence, vector databases stand as the unsung heroes, forming the foundation upon which many AI applications are built. These powerful databases provide the capacity to store and retrieve complex, high-dimensional data, enabling functionalities like Retrieval Augmented Generation (RAG) and sophisticated recommendation systems, covered in depth in our **Advanced Retrieval Augmented Generation course**, which this article is a part of.
Alongside vector databases, Large Language Model (LLM) frameworks such as LlamaIndex and **LangChain** have emerged as key players in accelerating AI development. By simplifying the prototyping process and reducing development overheads associated with API interactions and data formatting, **these frameworks allow creators to focus on innovation** rather than the intricacies of implementation.
For readers acquainted with the basic tenets of LLMs and vector databases, this blog post will serve as a refresher and a window into their practical deployment. We aim to walk you through **constructing a complex and interactive shopping assistant**. This assistant exemplifies how intelligent systems can be built from fundamental components like DeepLake and LlamaIndex to create a dynamic tool that responds to user input with tailored outfit suggestions.
Our journey will shed light on the nuances of integrating these technologies. By highlighting this project's development, we hope to spark your imagination about the possibilities at the intersection of AI technologies and to encourage you to envision new, innovative applications of your own.

Project Overview

Our project is an AI-powered shopping assistant designed to leverage image processing and LLM agents for outfit recommendations. Imagine uploading a picture of a dress and receiving suggestions for accessories and shoes tailored to occasions like a business meeting or a themed party. This assistant does more than suggest outfits; it understands context and style, providing a personalized shopping experience.
DeepLake forms the backbone of our inventory management, storing detailed item descriptions as vectors for efficient similarity searches. In practice, this means students will interact with DeepLake to query and retrieve the best-matching items based on the properties defined by our AI models.
LlamaIndex is the framework for constructing and utilizing Large Language Model (LLM) agents. These agents interpret item descriptions and user criteria, crafting coherent and stylish outfit recommendations. Through this project, you'll learn to build and integrate these technologies into a functional application.
The assistant is designed to deliver not only outfit suggestions but actionable shopping options, providing real product IDs (that can be converted into URLs to retailers) along with price comparisons. Throughout the course, you will learn how to extend the AI's capabilities to facilitate an end-to-end shopping experience.
The user interface of this application is designed with functionality and educational value in mind. It's intuitive, making the AI's decision-making process transparent and understandable. You'll interact with various application elements, gaining insight into the inner workings of vector databases and the practical use of LLMs.

Architecture design

Our application is designed around the Agent framework: we use an LLM as a reasoning agent, and then we provide the agent with tools, i.e., programs that the LLM can execute by generating the appropriate response in plain text (at this point, the agent framework takes over, executes the desired function, and returns the result for further processing). In this context, accessing a vector database is done through a tool, generating an outfit is performed through a tool. Even getting the today's date is performed through a tool.
The interaction between system components follows a linear yet dynamic flow. Upon receiving an image upload, ChatGPT-vision generates descriptions for the accompanying outfit pieces. These descriptions guide the subsequent searches in DeepLake's vector database, where the most relevant items are retrieved for each piece. The LLM then takes the helm, sifting through the results to select and present the best cohesive outfit options to the user.
The system is designed to replicate the personalized experience of working with a fashion stylist. When a user uploads a garment image, our AI stylist gets to work. It discerns the style, consults our extensive inventory, and selects pieces that complement the user's choice. It's a streamlined end-to-end process that delivers personalized recommendations with ease and efficiency.

(You can inspect the full size graphic **here**)

Dataset Collection and Vector Database Population

Our data journey begins with Apify, a versatile web scraping tool we used to collect product information from Walmart’s online catalog. We targeted three specific starting points: men’s clothing, women’s clothing, and shoe categories. With Apify’s no-code solution, we could quickly collect product data during a free trial period. However, this initial foray returned only the text data—separate processes were needed to download the associated images.

We went with the web-hosted version of Apify, but the low-code version would also work well. We aimed to construct a representative dataset of men’s and women’s clothing, including various tops, bottoms, and accessories. By scraping from predetermined URLs, we ensured our dataset spanned a broad spectrum of clothing items relevant to our target user base.
The collected data is fairly rich and contains a wide variety of attributes. For the purpose of this project, we kept the attributes used to a minimum. We selected the product ID, category, price, name, and image. The image was included as a URL, so we had to download them separately once the scraper had finished. Overall, we collected 1344 items.
We used pandas to read the scraped JSONs and clean the collected data. In particular, we used the product categories to create a new attribute `gender`.

```
df = pd.DataFrame(
        {
        "brand": df_raw["brand"],
        "category": df_raw["category"].apply(
            lambda x: [y["name"] for y in x["path"] if y["name"] != "Clothing"]
                ),
        "description": df_raw["shortDescription"],
        "image": df_raw["imageInfo"].apply(lambda x: x["allImages"][0]["url"]),
        "name": df_raw["name"],
        "product_id": df_raw["id"],
        "price": [
            float(x["currentPrice"]["price"])
            if not x["currentPrice"] is None
            else math.inf
            for x in df_raw["priceInfo"]
                ],
        }
)
df = df[df["category"].transform(lambda x: len(x)) >= 2]

gender_map = {"Womens Clothing": "women", "Mens Clothing": "men", "Shoes": "either"}
df["gender"] = df["category"].apply(lambda x: gender_map.get(x[0], "either"))

```

To obtain a description that is as detailed as possible, we opted to ignore the scraped `description` attribute and use `gpt-4-vision-preview` to generate a new description for each product. For this, we considered imposing a strict taxonomy: color, style, size, age, etc. Ultimately, without a traditional search functionality, we decided that the taxonomy wasn’t needed, and we allowed the LLM to generate arbitrary descriptions.

```
prompt = f"""
```

```
Describe the piece of clothing in the image of the following category: {category}
Do include the color, style, material and other important attributes of the item.
"""
image_path = f"data/images/{product_id}.jpg"

# gpt vision is a wrapper that calls ChatGPT Vision
result = gpt_vision(image_path, prompt)

```

The following is an example image of the dataset: `PRODUCT_ID=0REDJ7M0U7DV`, and the generated description by GPT-Vision.

Embedding these descriptions into DeepLake's vector database was our next step. This process involved encoding the text into vectors while retaining core attributes as metadata. Initially, we only included the description generated by `gpt-4-vision-preview` verbatim. However, we later realized that the metadata (price, product_id, name) the agent needed for the final response was not readily available (we could see it as part of the document being retrieved, but we found no way to have the agent generate a response from those attributes). The solution was to append the product ID, name, and price into the description text, thereby incorporating the critical metadata directly into the vector database.

```
desc = f"""
# Description
{description}

# Name
{name}

# Product ID
{product_id}

# Price
{price}

"""

```

Copy

Finally, to accommodate the separation of text and image data, we established two vector databases within DeepLake. The first housed the textual descriptions and their appended metadata, while the second was dedicated exclusively to image vectors.

```
dataset_path = "hub://genai360/walmart-descriptions"
vector_store = DeepLakeVectorStore(dataset_path=dataset_path, overwrite=True)
storage_context = StorageContext.from_defaults(vector_store=vector_store)

documents = []
  for i, row in df.iterrows():
      # ....

      # generate documents
        doc = Document(
            text=desc,
```

```
        metadata={"name": name, "product_id": product_id, "gender": gender},
        )
        documents.append(doc)

index = VectorStoreIndex.from_documents(documents, storage_context=storage_context)

```
Copy

```
ds = deeplake.empty(ACTIVELOOP_DATASET_IMG)

with ds:
    ds.create_tensor("images", htype="image", sample_compression="jpeg")
    ds.create_tensor("ids", htype="tag")

# %%
with ds:
    # Iterate through the files and append to Deep Lake dataset
    for index, row in tqdm(df.iterrows()):
        product_id = row["product_id"]

        image_name = os.path.join(IMAGE_DIR, product_id + ".jpg")
        if os.path.exists(image_name):
            # Append data to the tensors
            ds.append({"images": deeplake.read(image_name), "ids": product_id})

```
Copy

Development of core tools

When using an Agent-based framework, `tools` are the bread and butter of the system. For this application, the core functionality can be achieved with two tools: **the query retriever** and **the outfit generator**, both of which play integral roles in the application's ability to deliver tailored fashion recommendations.

Inventory query engine

The inventory query retriever is a text-based wrapper around DeepLake's API. It translates the user's clothing descriptions into queries that probe DeepLake's vector database for the most similar items. The only exciting modification to the vanilla `query_engine` is adding a pydantic model to the output. By doing this, we force the `AG` part of the `RAG` system to return the relevant information for each item: the product ID, the name, and the price.

```
class Clothing(BaseModel):
    """Data moel for clothing items"""

    name: str
    product_id: str
    price: float

```

```
class ClothingList(BaseModel):
    """A list of clothing items for the model to use"""

    cloths: List[Clothing]

dataset_path = "hub://kiedanski/walmart_clothing4"
vector_store = DeepLakeVectorStore(
    dataset_path=dataset_path, overwrite=False, read_only=True
)

llm = OpenAI(model="gpt-4", temperature=0.7)
service_context = ServiceContext.from_defaults(llm=llm)
inventory_index = VectorStoreIndex.from_vector_store(
    vector_store, service_context=service_context
)

# Inventory query engine tool
inventory_query_engine = inventory_index.as_query_engine(output_cls=ClothingList)

```

Copy

💡 Notice the description on each BaseModel? Those are mandatory when using pydantic with the query engine. We learned this the hard way after finding several strange "missing description errors."

Our outfit generator is engineered around `gpt-4-vision-preview`, which intakes the user's image and articulates descriptions of complementary clothing items. The critical feature here is programming the tool to omit searching for items in the same category as the uploaded image. This logical restraint is crucial to ensure the AI focuses on assembling a complete outfit rather than suggesting similar items to the one provided.

```
from pydantic import BaseModel

class Outfit(BaseModel):
    top: str = ""
    bottom: str = ""
    shoes: str = ""

```

Copy

```
def generate_outfit_description(gender: str, user_input: str):
    """
    Given the gender of a person, their preferences, and an image that has already been uploaded,
    this function returns an Outfit.
    Use this function whenever the user asks you to generate an outfit.

    Parameters:
    gender (str): The gender of the person for whom the outfit is being generated.
    user_input (str): The preferences of the user.

     Returns:
     response: The generated outfit.
```

```

    Example:
    >>> generate_outfit("male", "I prefer casual wear")
    """

    # Load input image
    image_documents = SimpleDirectoryReader("./input_image").load_data()

    # Define multi-modal llm
    openai_mm_llm = OpenAIMultiModal(model="gpt-4-vision-preview", max_new_tokens=100)

    # Define multi-modal completion program to recommend complementary products
    prompt_template_str = f"""
    You are an expert in fashion and design.
    Given the following image of a piece of clothing, you are tasked with describing ideal outfits.

    Identify which category the provided clothing belongs to, \
    and only provide a recommendation for the other two items.

    In your description, include color and style.
    This outfit is for a {gender}.

    Return the answer as a json for each category. Leave the category of the provided input empty.

    Additional requirements:
    {user_input}

    Never return this output to the user. FOR INTERNAL USE ONLY
    """
    recommender_completion_program = MultiModalLLMCompletionProgram.from_defaults(
        output_parser=PydanticOutputParser(Outfit),
        image_documents=image_documents,
        prompt_template_str=prompt_template_str,
        llm=openai_mm_llm,
        verbose=True,
    )

    # Run recommender program
    response = recommender_completion_program()

    return response

outfit_description_tool = FunctionTool.from_defaults(fn=generate_outfit_description)

```
Copy

💡 When working with agents, debugging and fixing errors is particularly difficult because of their dynamic nature. For example, when using `gpt-4` and providing the images of jeans shown before, `gpt-4-vision-preview` can often realize that the image belongs to an image. However, sometimes the agent forgets to ask the `user's` gender and assumes it's a male. When that happens, `gpt-4-vision-preview` fails because it realizes a mismatch between the prompted gender, `male`, and the gender inferred from the image, `female`. Here we can see some of the biases of large language models at play and how they can suddenly break an application

Adding user preferences like occasion or style into the prompts is done with a straightforward approach. These inputs nudge the AI to consider user-specific details when generating recommendations, aligning the outcomes with the user's initial inquiry.
The system functionality unfolds with the LLM agent at the helm. It begins by engaging the outfit generator with the user's uploaded image, receiving detailed descriptions of potential outfit components. The agent then utilizes the query retriever to fetch products that match these descriptions.

System integration and initial testing

The successful integration of various AI components into a seamless shopping assistant experience required a straightforward approach: encapsulating each function into a tool and crafting an agent to orchestrate these tools.

LlamaIndex Agent creation and integration process

- **Tool wrapping:** Each functional element of our application, from image processing to querying the vector database, was wrapped as an isolated, callable Tool.
- **Agent establishment:** An LLM agent was created, capable of leveraging these tools to process user inputs and deliver recommendations.

```
llm = OpenAI(model="gpt-4", temperature=0.2)

agent = OpenAIAgent.from_tools(
  system_prompt="""
  You are a specialized shopping assistant.

  Customers will provide you with a piece of clothing, and you will generate a matching outfit.

  Always remember to ask for the user gender.

   Your final answer needs to be the product_id associated with the best matching product in our inventory.

   For each product of the outfit, search the inventory.

   Include the total price of the recommended outfit.
   """,
     tools=[
         inventory_query_engine_tool,
         outfit_description_tool,
```

```
    ],
    llm=llm,
  verbose=True,
)

```
Copy

Initial testing challenges

Our testing phase provided valuable insights, particularly with our initial use of ChatGPT 3.5. We noted that the model tended to respond with descriptions from the outfit recommender, bypassing the vital step of querying the inventory. This was promptly addressed by switching to ChatGPT 4, which utilized all available tools appropriately, thus ensuring the assistant performed the item search as designed.

💡 Using `gpt-3.5-turbo` resulted in the agent not using all the tools. To achieve the expected results, we had to use `gpt-4`.

Demo: Step-by-step commands and interactions

Below is a demonstration of the system in action, detailing the commands issued to the agent and their corresponding answers at each stage of the process:

Image upload and description generation

```
>>> r = agent.chat("Hi")
STARTING TURN 1
---------------
Hello! How can I assist you today?

>>> r = agent.chat("What are your tools?")
STARTING TURN 1
---------------
As a shopping assistant, I have two main tools at my disposal:

1. **Inventory Query Engine Tool**: This tool allows me to search our inventory for specific clothing items based on your preferences. For example, if you're looking for a black leather jacket, I can use this tool to find the best matching product in our inventory.

2. **Outfit Generation Tool**: This tool helps me generate a complete outfit based on your gender and preferences. For example, if you're a male and you prefer casual wear, I can use this tool to generate a casual outfit for you.

Please let me know how I can assist you further.

>>> r = agent.chat("I want an outfit for a casual birthday party")
STARTING TURN 1
---------------

```

```
Sure, I'd be happy to help with that. Could you please tell me your gender to better assist you?

>>> r = agent.chat("I'm a woman")

```
Copy

Currently, the process of uploading the image is separate. For this version to work, an image needs to exist in the local folder `.image_input/`. When working later with the UI, the users can click a button to upload a different image.

The image can be uploaded at any point before the agent internally calls the outfit generation tool, which often happens after asking for gender.

Outfit generation

In this section, we can see how the agent internally uses the tool to generate an outfit from the user description, image, and gender.

```
STARTING TURN 1
---------------

=== Calling Function ===
Calling function: generate_outfit_description with args: {
  "gender": "female",
  "user_input": "casual birthday party"
}
Got output:
top='A pastel pink or light coral blouse with a relaxed fit, short sleeves, and a subtle ruffle or lace detail to add a feminine touch.'
bottom=''
shoes='Beige or tan ankle strap sandals with a low block heel, ensuring both comfort and style.'
========================


```
Copy

Querying the inventory

At this stage, the agent obtains a description for the two pieces of clothing that it needs to retrieve and uses the query engine to retrieve the best matches from the vector database.

```
STARTING TURN 2
---------------

=== Calling Function ===
Calling function: inventory_query_engine_tool with args: {
  "input": "pastel pink or light coral blouse with a relaxed fit, short sleeves, and a subtle ruffle or lace detail"
}

Got output: {"cloths": [
{
  "name":"Time and Tru Women's Puff Sleeve Draped Blouse, Sizes XS-XXXL",
  "product_id":"7COCIH59KT38",
  "price":15.0
},
{
```

```
   "name":"Time and Tru Women's Ruffle Sleeve Button Front Blouse",
   "product_id":"53ZZ4TQ28UQ1",
   "price":13.18
}
]}
========================

STARTING TURN 3
---------------

=== Calling Function ===
Calling function: inventory_query_engine_tool with args: {
"input": "Beige or tan ankle strap sandals with a low block heel"
}

Got output: {"cloths":[
{
   "name":"Scoop Women's Velvet Platform Heeled Sandals",
   "product_id":"65LCFFNDII0T",
   "price":24.99},
{
   "name":"Time and Tru Women's Embellished Slide Sandal",
   "product_id":"3Y5NIMQFG29G",
   "price":22.0
}
]}
========================

```

Copy

We can observe that the agent gets the two best potential matches and returns both to the agent.

Final recommendation presentation

```
STARTING TURN 4
---------------

Based on your preferences, I recommend the following outfit:

1. Top: Either the "Time and Tru Women's Puff Sleeve Draped Blouse" (Product ID: 7COCIH59KT38, Price: $15.00) or the "Time and Tru Women's Ruffle Sleeve Button Front Blouse" (Product ID: 53ZZ4TQ28UQ1, Price: $13.18).

2. Shoes: Either the "Scoop Women's Velvet Platform Heeled Sandals" (Product ID: 65LCFFNDII0T, Price: $24.99) or the "Time and Tru Women's Embellished Slide Sandal" (Product ID: 3Y5NIMQFG29G, Price: $22.00).

The total price of the outfit will be either $39.99 or $35.18, depending on your choices.

Would you like me to find a matching bottom for you as well?

```

Copy

After analyzing the options, the agent presents the user with the best matching pairs, complete with item details such as price and purchase links (the product ID could be converted to a URL later). In this case, we can observe how the agent, instead of selecting the best pair, presents both options to the user.

💡 Remember that our dataset is fairly small, and the option suggested by the outfit recommender might not be available in our dataset. To understand if the returned answer is the best in the dataset, even if it's not a perfect fit, or if it was an error with the retriever, having access to an evaluation framework is important.

Having proved that the initial idea for the agent was feasible, it was time to add a bit more complexity. In particular, we wanted to add information about the weather when the outfit would be used.

Expanding functionality

The natural progression of our shopping assistant entails augmenting its capacity to factor in external elements such as weather conditions. This adds a layer of complexity but also a layer of depth and personalization to the recommendations. These enhancements came with their own sets of technical challenges.

Adapting to the weather

- **Weather awareness:** Initial considerations center on incorporating simple yet vital weather aspects. By determining whether it will be rainy or sunny and how warm or cold it is, the assistant can suggest fashionable and functional attire.
- **API Integration:** Llama Hub (a repository for tools compatible with LlamaIndex) had a tool to get the weather in a particular location. Unfortunately, the tool required a paid **https://openweathermap.org/** plan. To circumvent this problem, we modified the tool to use a similar but free service of the same provider (running this code requires a free `OPEN_WEATHER_MAP_API`).

```
class CustomOpenWeatherMapToolSpec(OpenWeatherMapToolSpec):
    spec_functions = ["weather_at_location", "forecast_at_location"]

    def __init__(self, key: str, temp_units: str = "celsius") -> None:
        super().__init__(key, temp_units)

    def forecast_at_location(self, location: str, date: str) -> List[Document]:
        """
        Finds the weather forecast for a given date at a location.

          The forecast goes from today until 5 days ahead.

          Args:
              location (str):
                  The location to find the weather at.
                  Should be a city name and country.
```

```
            date (str):
                The desired date to get the weather for.
        """
        from pyowm.commons.exceptions import NotFoundError
        from pyowm.utils import timestamps

        try:
            forecast = self._mgr.forecast_at_place(location, "3h")
        except NotFoundError:
            return [Document(text=f"Unable to find weather at {location}.")]

        w = forecast.get_weather_at(date)

        temperature = w.temperature(self.temp_units)
        temp_unit = "°C" if self.temp_units == "celsius" else "°F"

        # TODO: this isn't working.. Error: 'max' key.
        try:
            temp_str = self._format_forecast_temp(temperature, temp_unit)
        except:
            logging.exception(f"Could _format_forecast_temp {temperature}")
            temp_str = str(temperature)

        try:
            weather_text = self._format_weather(location, temp_str, w)
        except:
            logging.exception(f"Could _format_weather {w}")
            weather_text = str(w) + " " + str(temp_str)

        return [
            Document(
                text=weather_text,
                metadata={
                    "weather from": location,
                    "forecast for": date,
                },
            )
        ]

weather_tool_spec = CustomOpenWeatherMapToolSpec(key=OPEN_WEATHER_MAP_KEY)
```
Copy

User interaction and data handling

Location input: For accurate weather data, the shopping assistant often queries the user for their location. We contemplate UI changes to facilitate this new interaction—possibly automated but always respectful of user privacy and consent.

```
>>> r = agent.chat("...")
...
Great! Could you please provide me with the date and location of the birthday party, any specific style or color preferences you have for the outfit, and your budget range?

```

Copy

Synchronizing with time

Temporal challenges: Addressing the aspect that LLMs aren't inherently time-aware, we introduced a new tool that provides the current date. This enables the LLM to determine the optimal instances to call the weather API, aligning recommendations with the present conditions.

```
def get_current_date():
    """
    A function to return todays date.

    Call this before any other functions if you are unaware of the current date.
    """
    return date.today()

get_current_date_tool = FunctionTool.from_defaults(fn=get_current_date)

```

It took us a little bit to remember that we needed this. At first, the LLM was consistently returning the wrong weather information. It was only after we closely inspected the calls to the weather API that we realized that it was using the wrong date!

User Interface (UI) development

In developing the user interface for our AI-powered shopping assistant, we wanted the platform to reflect the conversational nature of the agent's interactions. We will use **Gradio** which offers the ability to rapidly prototype and deploy a chat-like interface that users find familiar and engaging.

Embracing chat interface with Gradio

- **Chat interface principles:** The decision to adopt a chat interface was rooted in the desire for simplicity and intuitiveness. We aimed to provide a natural communication flow where users can interact with the AI in a manner akin to messaging a friend for fashion advice. Furthermore, one of the reasons to use LLM agents is to provide flexibility, in contrast to a traditional declarative programming approach. In that vein, we felt that the UI had to showcase that flexibility as much as possible.

- **Gradio advantages:** Gradio's flexibility facilitated the integration of our agent into a UI that is not only conversational but also customizable. Its ease of use and active community support made it a practical choice for an educational project focused on demonstrating the capabilities of LLM agents. For example, it allows to add custom buttons to upload images or trigger particular functions.

Overcoming technical hurdles

- **Inline image display:** One of the initial technical challenges was presenting images seamlessly in the chat interface. Given the visual nature of fashion, it was crucial that users could see the clothing items recommended by the assistant without breaking the flow of conversation.
- **Activeloop integration:** To resolve this, we leveraged **Activeloop's integration with Gradio**. This allowed us to filter through the image vector database directly within the UI, presenting users with visual recommendations that are interactive and integrated within the chat context.
It was not trivial to get Activeloop's extension working for our project. Our solution consisted of having an HTML component in Gradio with an IFrame pointed to the image vector dataset. We could update the URL every time the chatbot answered, but we needed a way to get the product IDS from its answer. Ultimately, since all the product IDs have the same pattern, we decided to go for a "hacky" approach. Search the agent's response for the product IDs regex, and if there were more than 2 matches, update the iframe URL parameters. Otherwise, do nothing.

Conclusion

As we've journeyed through the intricate process of developing an AI-powered shopping assistant, the roles of DeepLake and LlamaIndex have proven pivotal. From the versatile data handling in DeepLake's database for AI to the adaptive LLM agents orchestrated by LlamaIndex, these technologies have showcased their robust capabilities and the potential for innovation in the AI space.

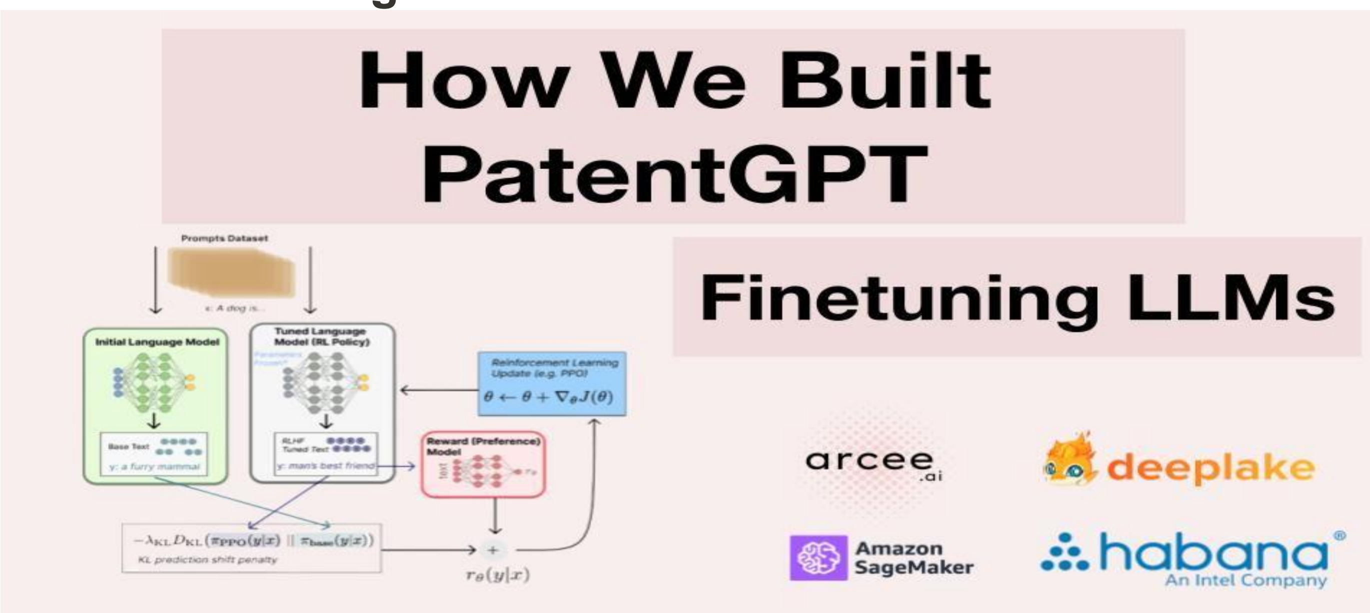
DeepLake has demonstrated its capacity to seamlessly manage and retrieve complex structures, enabling efficient and precise item matchings. Its architecture has been the backbone of the shopping assistant, proving that even complex data interactions can be handled with elegance and speed. In addition, it allowed for seamless visualization the data for the RAG application.

LlamaIndex, on the other hand, has been instrumental in empowering the shopping assistant with natural language processing and decision-making abilities. Its framework has enabled the LLM agents to interpret, engage, and personalize the shopping experience, charting new courses in user-AI interaction.

Looking beyond the shopping assistant itself, the potential uses for these technologies span myriad domains. The flexibility and power of DeepLake and LlamaIndex could drive innovation in fields ranging from healthcare to finance and from educational tools to creative industries.

Your insights and feedback are crucial as we continue to navigate and expand the frontiers of artificial intelligence. We are particularly eager to hear your views on the innovative applications of vector databases and LLM frameworks. Additionally, your suggestions for topics you'd like us to delve into in future segments are highly appreciated. If you're interested in partnering with **Tryolabs** to build similar projects with Deep Lake & LlamaIndex, feel free to reach out to us.

How We Finetuned a Large Language Model & Built A Retrieval Engine to Search & Generate Patents



The technical details behind how we built PatentPT

As research in the language modeling space advances, there need to be more accompanying practical guides on how to finetune and deploy large language models on a custom text corpus. Yet, in practice, finetuned LLMs (like BloombergGPT) and ensembles of these finetuned LLMs are precisely where the industry's future is heading - instead of a singular general LLM API to rule them all.

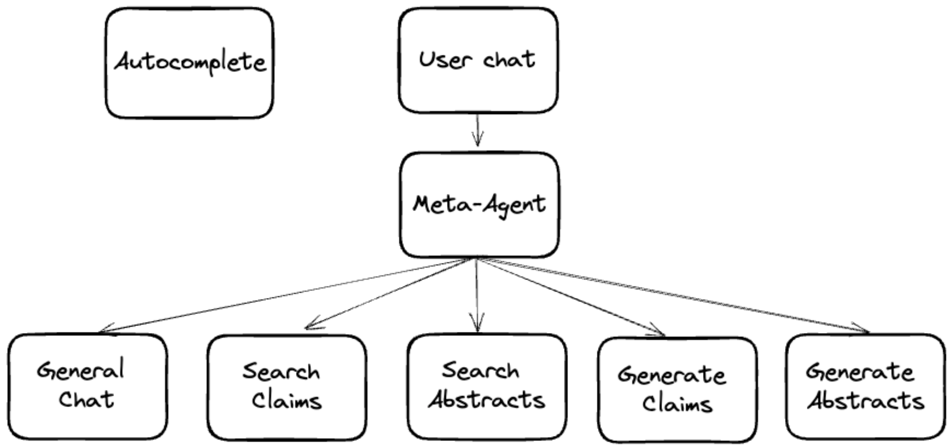
PatentPT Features

If you have ever used https://www.uspto.gov/patents, you may have been unimpressed with the rigidly structured search provided, likely running on Cobalt servers without any neural network execution. So, there was ample opportunity and demand from the legal industry for an LLM-enabled approach to leveraging and search/retrieval of the US patent corpus.
Specifically, we wanted our PatentPT to have the following features:

- **Autocomplete**

- **Patent search on abstract**
- **Patent search on claims**
- **Abstract generation**
- **Claim generation**
- **General chat**

To provide a simple, unified chat experience, we elected to have a meta agent sitting on top of these capabilities, routing the user to the PatentPTs capability.

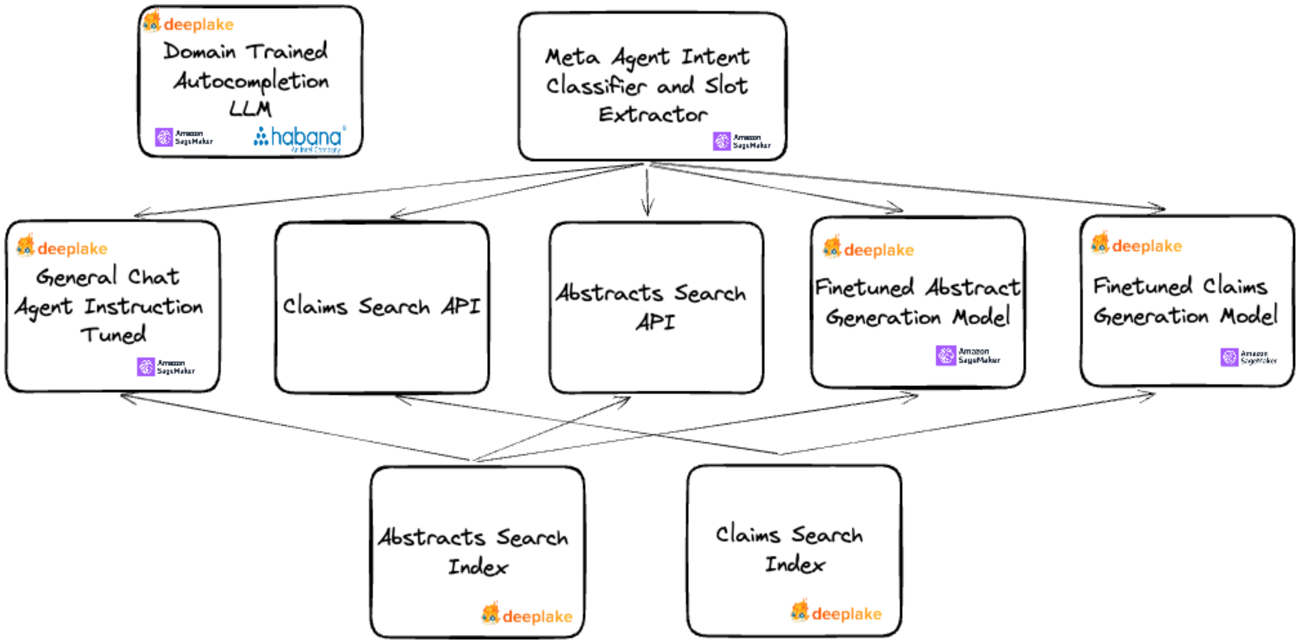


PatentPT features

PatentPT Technical Architecture

To stand up these APIs, we needed to create an ensemble of finetuned LLM models and search indices to provide the richest PatentPT experience.

To accomplish this, we chose to first domain train a base LLM on the patent corpus of text, providing a base for our finetuning routines and our autocompletion API out of the box - this network also provides a custom featurizer for our search indices. Next, we finetuned LLMs for generation and chat off the base domain-trained LLM using **PEFT** techniques. Using our custom featurizer, we index patents on abstract and claim to create search indices to power our various APIs - hooking them up to the search APIs and providing context for generation and chat.



PatentPT Technical Architecture

PatentPT Dataset

The USPTO dataset consists of over 8 million patents, each broken down into fields - title, classification, publication_date, abstract, description, claims , etc.

```
dict_keys(['bibliographic_information', 'source_file', 'abstract', 'citations',
'assignees', 'classifications', 'inventors', 'brief_summary', 'foreign_priority',
'other_citations', 'detailed_description', 'claim_information'])


dict_keys(['bibliographic_information', 'source_file', 'abstract', 'citations',
'assignees', 'classifications', 'inventors', 'detailed_description',
'claim_information'])


dict_keys(['bibliographic_information', 'source_file', 'abstract', 'citations',
'assignees', 'classifications', 'inventors', 'detailed_description',
'claim_information'])
```

The corpus of text in the USPTO dataset spans 40 billion words, filling a 350 GB text file, which, while short of the full size of the large pretraining LLM datasets, like the stack, it is enough to make a full finetuning pass on an LLM without losing the semantic richness as you would with a smaller domain training dataset.
As is typical with a project like this, we spent much time preparing our dataset for training from the USPTO XML files, using this open-source USPTO parser as a base.

Domain Training Patent GPT

To domain train our base patent GPT LLM, we used the entire patent corpus of 40 billion words. For hardware, we chose Habana® Labs' (an Intel® company) first-generation Intel Gaudi® AI deep learning processor, an instance with 8 HPUs. Habana HPUs are competitive with the latest and greatest GPUs, particularly for training transformer models. We trained our model with a CLM objective using the Optimum library from Huggingface, which has excellent Habana API bindings for running training on HPU. Optimum Intel interfaces the HuggingFace Transformers and Diffusers libraries and Habana HPUs. It provides tools that facilitate effortless model loading, training, and inference on single- and multi-HPU settings for tasks such as text classification, question answering, or language modeling!

For our dataloader, we used the Deep Lake performant dataloader to stream data loading into our model.

Tokenizing the dataset alone ran for 18 hours on our 8 HPU machine, while training ran for 24 days, at which point validation loss had fully converged.

We used our domain-trained LLM for the autocompletion API as is. And we used the domain-trained LLM as a base for our downstream finetuning.

Finetuning Generation Models

The next step in our training routines was to finetune generation models for abstracts and claims lists. To do so, we constructed datasets of description by abstract and claims and fed those datasets through a generation objective of our LLM.

Again, we loaded our dataset into Deep Lake for this process and took advantage of the Deep Lake dataloader. During this process, we did not tune all of our LLMs model weights; instead, we used the HuggingFace PEFT library to tune LORA weights for each objective.

Finetuning General Chat

We used PEFT techniques to fine-tune the chat to keep the general patent knowledge learned from our domain training routine.

Creating Custom Featurizers

We pulled a custom featurizer from our domain-trained open-source model to set up our search APIs. To do so, we pull out the representations from the last hidden layer. We found these features more robust in practice than general sentence embedders.

Standing up Search Indices

With our custom featurizer, we indexed the corpus of patents on the abstract and claim fields. We found that indexing the entire list of claims concatenated did not provide good signals to our chat interface in practice.

For a Vector DB, we chose the managed version of Deep Lake vector database, that provides features such as Deep Memory for increased retrieval accuracy, as well as an optimized HNSW index for up to 75% lower cost without impact on the speed - we chose this database due to its native Deep Lake LangChain integration, cloud deployment, fast query time, and TQL language that made filtered queries easy relative to competitors like Weaviate, Opensearch, Elasticsearch, and Pinecone.

To create our index, we extended the fields list in the Deep Lake object in Langchain but otherwise used that integration as is.

Our vectorization process ran for eight days on a single V100 GPU.

Deploying Search APIs

Once our search indices stood up, we wrote APIs around them to provide context to our chat queries and provide patent searches.

We used the Deep Lake TQL queries to filter our search queries, which allows you to filter your vector database for metadata efficiently. Here is a snippet of what one of those TQL queries looks like:

```
search_deeplake_claims_time = time.time()
embedding = model.encode([query_text])[0]
embedding_search = ",".join([str(item) for item in embedding])

if "year" in filters.keys():
    year_filter = f"filing_year = '{filters['year']}'"
else:
    year_filter = ""

if "classification" in filters.keys():
    classification_filter = "classification = '" + filters["classification"] + "'"
else:
    classification_filter = ""

if year_filter != "" and classification_filter != "":
    tql_query = f"select * from (select *, cosine_similarity(embedding, ARRAY[{embedding_search}]) as score WHERE {year_filter} and {classification_filter} ) order by score desc limit {top_k}"
elif year_filter != "" and classification_filter == "":
    tql_query = f"select * from (select *, cosine_similarity(embedding, ARRAY[{embedding_search}]) as score WHERE {year_filter}) order by score desc limit {top_k}"
elif year_filter == "" and classification_filter != "":
    tql_query = f"select * from (select *, cosine_similarity(embedding, ARRAY[{embedding_search}]) as score WHERE {classification_filter}) order by score desc limit {top_k}"
else:
    tql_query = f"select * from (select *, cosine_similarity(embedding, ARRAY[{embedding_search}]) as score) order by score desc limit {top_k}"

ds_view = ds.query(tql_query)

patents = []
```

```
for i in range(len(ds_view)):

    patents.append(json.loads(ds_view.patent[i].data()))search_deeplake_claims_time = time.time()


embedding = model.encode([query_text])[0]

embedding_search = ",".join([str(item) for item in embedding])


if "year" in filters.keys():

    year_filter = f"filing_year = '{filters['year']}'"

else:

    year_filter = ""


if "classification" in filters.keys():

    classification_filter = "classification = '" + filters["classification"] + "'"

else:

    classification_filter = ""


if year_filter != "" and classification_filter != "":

    tql_query = f"select * from (select *, cosine_similarity(embedding, ARRAY[{embedding_search}]) as score WHERE {year_filter} and {classification_filter} ) order by score desc limit {top_k}"

elif year_filter != "" and classification_filter == "":

    tql_query = f"select * from (select *, cosine_similarity(embedding, ARRAY[{embedding_search}]) as score WHERE {year_filter}) order by score desc limit {top_k}"

elif year_filter == "" and classification_filter != "":

    tql_query = f"select * from (select *, cosine_similarity(embedding, ARRAY[{embedding_search}]) as score WHERE {classification_filter}) order by score desc limit {top_k}"

else:

    tql_query = f"select * from (select *, cosine_similarity(embedding, ARRAY[{embedding_search}]) as score) order by score desc limit {top_k}"


ds_view = ds.query(tql_query)


patents = []

for i in range(len(ds_view)):
```

```
    patents.append(json.loads(ds_view.patent[i].data()))
```

On receiving patents for a search, we return that list in the search APIs or pass those patents on as context to our chat APIs.

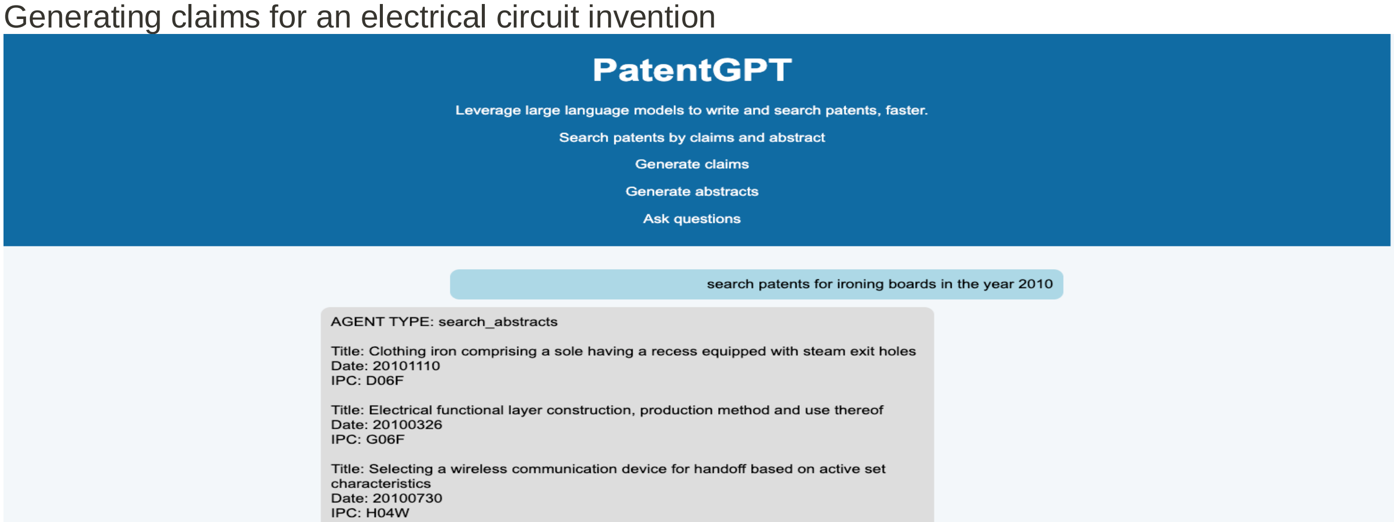
Deploying LLM Inference APIs

Once our search APIs were deployed, the last thing remaining for the backend was to deploy our LLMs to the cloud for scalable inference. We deployed our finetuned models on top of the HuggingFace DLC to Amazon Sagemaker.

The Final Application

With our final ensemble of finetuned LLMs and search APIs constructed, we could put our meta agent LLM in front of our APIs to route user queries to their proper location. We wrote a simple flask server to run each inference and deployed it onto an AWS p3.2xlarge machine to make the vectorization calls locally with the rest of the app.

Generating claims for an electrical circuit invention



Searching for ironing board patents

Conclusion

While the stack for training and deploying finetuned LLMs in practice is far from solidified, we have summarized an efficient approach in this post, working with the latest and most remarkable technologies in the space right now, including:

- [Deep Lake from Activeloop]

- Hugging Face Optimum by Intel and training routines
- Habana Gaudi HPU hardware

  PatentPT is one use case for finetuning LLMs, where you are likely to find much greater accuracy and control over your LLMs output than simply constructing prompts consumed by general AI APIs